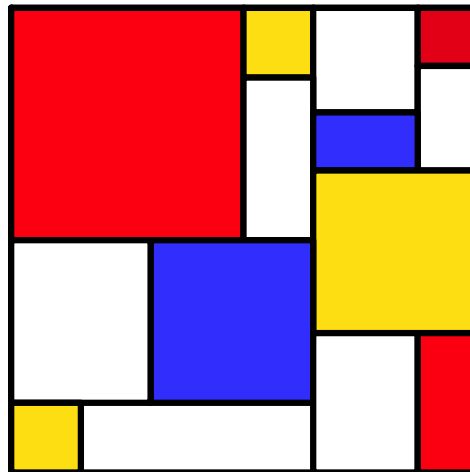


# SLiM: An Evolutionary Simulation Framework

**Benjamin C. Haller and Philipp W. Messer**

Dept. of Computational Biology  
Cornell University, Ithaca, NY 14853

Correspondence: [bhaller@benhaller.com](mailto:bhaller@benhaller.com)



Last revised 14 September 2022, for SLiM version 4.0.1.

## Author Contributions:

SLiM 2 and later were conceived and designed by BCH and PWM, based upon the previous design of SLiM by PWM. BCH designed and implemented the Eidos scripting language, wrote almost all of the code for SLiM 2 and later (but see the acknowledgements below), and wrote this manual. PWM provided feedback and edited this manual.

## Acknowledgements:

The authors want to thank Ian Caldas, Sam Champer, Isabel Kim, Mitch Lokey, Nathan Oakes, Aaron Sams, and all members of the Messer lab for helpful feedback and bug reports. Thanks to Kevin Thornton and Ryan Hernandez for many discussions and for their general help in promoting forward population genetic simulations. Thanks to Jared Galloway, Jerome Kelleher, and Peter Ralph for their considerable work in implementing tree-sequence recording in SLiM 3; and to Peter Ralph as well for his invaluable `pyslim` framework, and for ideas and contributions to this manual. Thanks to Bryce Carson, Russell Dinnage, and Graham Gower for major contributions to the SLiM ecosystem, including new platform installers and porting work. Thanks to Simon Aeschbacher, Pamela Alamilla, Rodrigo Pracana Fragoso De Almeida, Jorge Amaya, Bill Amos, Chenling Antelope, Jaime Ashander, Hannes Becher, Emma Berdan, Jeremy Berg, Gertjan Bisschop, Dan Bolnick, Tom Booker, Gideon Bradburd, Jason Bragg, Vince Buffalo, Yoann Buoro, Richard Burns, Bryce Carson, Deborah Charlesworth, Jeremy Van Cleve, Jonathan Cocker, Zuxi Cui, Jean Cury, Michael DeGiorgio, A.P. Jason de Koning, Emily Dennis, Jordan Rohmeyer Dherby, Russell Dinnage, Alan Downey-Wall, Ian Dworkin, Julia Frank, Jared Galloway, Jesse Garcia, Kimberley Gilbert, Graham Gower, Alexandre Harris, Kelley Harris, Rebecca Harris, Matthew Hartfield, Ding He, Jody Hey, Marcus Hicks, Kathryn Hodgins, Christian Huber, Melissa Jane Hubisz, Emilia Huerta-Sanchez, Chaz Hyseni, Jacob Malte Jensen, Peter Keightley, Jerome Kelleher, Andy Kern, Bhavin Khatri, Bernard Kim, Jere Koskela, Athanasios Kousathanas, Peter Krawitz, Chris Kyriazis, Benjamin Laenen, Anna Maria Langmüller, Áki Láruson, Stefan Laurent, Clancy Lawler, Anita Lerch, Mitchell Lokey, Qiming Long, Eugenio Lopez, Kathleen Lotterhos, Nicolas Lou, Ailene MacPherson, Andrew Marderstein, Sebastian Matuszewski, Mikhail Matz, Rupert Mazzucco, Christopher McAllester, Maéva Mollion, Miguel Navascués, Chase Nelson, Dominic Nelson, Bruno Nevado, Tram Nguyen, Etsuko Nonaka, Boyana Norris, Nick O'Brien, Omar Eduardo Cornejo Ordaz, Matt Osmond, Greg Owens, Josephine Paris, Harvinder Pawar, Martin Petr, Denis Pierron, Fernando Racimo, Peter Ralph, Erik Regla, David Rinker, Murillo Fernando Rodrigues, Andrew Sackman, Kieran Samuk, Sara Schaal, Sebastian Schreiber, Derek Setter, Elissa Sorojsrisom, Onuralp Söylemez, Fabian Staubach, Stefan Strütt, Michelle Su, Aurelien Tellier, Anastasia Teterina, Silas Tittes, Rob Unckless, Magnus Dehli Vigeland, Nathan Villiger, Christos Vlachos, Silu Wang, Clemens L. Weiß, James Whiting, Alexander Whitwam, Aaron Wolf, Yan Wong, Yannick Wurm, Alexander Xue, Sam Yeaman, Justin Yeh, and Yulin Zhang for comments and feedback that have led to improvements in SLiM. Thanks also to everyone on stackoverflow, an invaluable resource and a great community; likewise to the Qt community on Qt Interest, particularly the ever-helpful Thiago Macieira. Finally, we want to thank Dmitri Petrov, whose support was instrumental in the initial conception of SLiM.

## Text conventions:

A standardized color-coding scheme is used throughout this manual:

sections that apply only to WF models

sections that apply only to nonWF models

sections that apply only to nucleotide-based models

Eidos script, and snippets from script such as variable names, class names, and numeric values, are generally written in a `monospace` font to distinguish them from other text.



## Citation:

To cite **SLiM 3** in a publication, please cite:

Haller, B.C., and Messer, P.W. (2019). SLiM 3: Forward genetic simulations beyond the Wright–Fisher model. *Molecular Biology and Evolution* 36(3), 632–637. DOI: <https://doi.org/10.1093/molbev/msy228>

Papers using **tree-sequence recording** should probably also cite that paper:

Haller, B.C., Galloway, J., Kelleher, J., Messer, P.W., & Ralph, P.L. (2019). Tree–sequence recording in SLiM opens new horizons for forward–time simulation of whole genomes. *Molecular Ecology Resources* 19(2), 552–566. DOI: <https://doi.org/10.1111/1755-0998.12968>

Papers which use **SLiM 2** can still cite that paper:

Haller, B.C., and Messer, P.W. (2017). SLiM 2: Flexible, interactive forward genetic simulations. *Molecular Biology and Evolution* 34(1), 230–240. DOI: <http://dx.doi.org/10.1093/molbev/msw211>

Papers which wish to cite **this manual** (perhaps because they make reference to a recipe) should cite:

Haller, B.C., and Messer, P.W. (2016). SLiM: An Evolutionary Simulation Framework.  
URL: [http://benhaller.com/slim/SLiM\\_Manual.pdf](http://benhaller.com/slim/SLiM_Manual.pdf)

And if you wish to cite a publication about **Eidos**, please cite the Eidos manual:

Haller, B.C. (2016). Eidos: A Simple Scripting Language.  
URL: [http://benhaller.com/slim/Eidos\\_Manual.pdf](http://benhaller.com/slim/Eidos_Manual.pdf)

## URL:

<http://messerlab.org/slim>

## License:

Copyright © 2016–2022 Philipp Messer. All rights reserved.

SLiM is a free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

## Disclaimer:

The program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License (<http://www.gnu.org/licenses/>) for more details.

# Contents

## PART I: THE SLiM COOKBOOK

1.	<a href="#">SLiM overview</a>	14
1.1	<a href="#">Introduction</a>	14
1.2	<a href="#">Why SLiM?</a>	15
1.3	<a href="#">A quick summary of SLiM</a>	16
1.4	<a href="#">The typical SLiM usage pattern</a>	19
1.5	<a href="#">Conceptual overview</a>	21
1.5.1	<a href="#">Individuals and genomes</a>	21
1.5.2	<a href="#">Mutations and substitutions</a>	23
1.5.3	<a href="#">Mutation stacking</a>	25
1.5.4	<a href="#">Genomic elements, genomic element types, mutation types, and the chromosome</a>	29
1.5.5	<a href="#">Subpopulations and migration</a>	30
1.5.6	<a href="#">Recombination, gene conversion, and biased gene conversion</a>	32
1.5.7	<a href="#">Community, Species, ticks, cycles, and generations</a>	36
1.5.8	<a href="#">Other concepts</a>	38
1.6	<a href="#">Wright–Fisher (WF) versus non-Wright–Fisher (nonWF) models</a>	38
1.7	<a href="#">Tree-sequence recording</a>	41
1.8	<a href="#">Nucleotide-based models</a>	44
1.9	<a href="#">Multispecies models</a>	47
1.10	<a href="#">Online resources for SLiM users</a>	49
2.	<a href="#">Installation</a>	54
2.1	<a href="#">Installation on macOS</a>	54
2.1.1	<a href="#">Installing the prebuilt SLiM package on macOS</a>	54
2.1.2	<a href="#">Building SLiM from sources on macOS</a>	54
2.2	<a href="#">Installation on Linux and other Un*x platforms</a>	59
2.2.1	<a href="#">Installing SLiMgui using Flatpak on Linux</a>	59
2.2.2	<a href="#">Installing SLiM with a platform-dependent binary package on Linux</a>	59
2.2.3	<a href="#">Building SLiM from sources on Linux</a>	62
2.3	<a href="#">Installation on Windows</a>	64
2.3.1	<a href="#">Installing SLiM with an installer on Windows</a>	64
2.3.2	<a href="#">Building SLiM from sources on Windows</a>	66
2.4	<a href="#">Building SLiMgui</a>	67
2.4.1	<a href="#">Building SLiMgui on macOS</a>	67
2.4.2	<a href="#">Building SLiMgui on Linux / Un*x</a>	69
2.4.3	<a href="#">Troubleshooting SLiMgui build issues on Linux / Un*x</a>	72
2.4.4	<a href="#">Building SLiMgui on Windows</a>	73
2.4.5	<a href="#">Troubleshooting SLiMgui build issues on Windows</a>	74
2.4.6	<a href="#">Building SLiMgui on Windows with WSL</a>	74
2.5	<a href="#">Installation with conda</a>	80
2.6	<a href="#">Testing the SLiM installation</a>	81
2.7	<a href="#">Post-installation issues</a>	82
3.	<a href="#">Running simulations in SLiMgui</a>	85
3.1	<a href="#">The SLiMgui simulation window</a>	85
3.2	<a href="#">The script help window</a>	87

3.3	<a href="#">The Eidos console</a>	88
3.4	<a href="#">The Eidos variable browser</a>	89
3.5	<a href="#">Automatic code completion and command syntax lookup</a>	89
3.6	<a href="#">Script prettyprinting</a>	91
3.7	<a href="#">Further SLiMgui features</a>	92
4.	<a href="#">Getting started: Neutral evolution in a panmictic population</a>	94
4.1	<a href="#">A basic neutral simulation</a>	94
4.1.1	<a href="#">initialize() callbacks</a>	96
4.1.2	<a href="#">Mutation rate</a>	97
4.1.3	<a href="#">Mutation types</a>	97
4.1.4	<a href="#">Genomic element types</a>	98
4.1.5	<a href="#">Genomic elements</a>	99
4.1.6	<a href="#">Recombination rate</a>	100
4.1.7	<a href="#">Eidos events</a>	101
4.1.8	<a href="#">Subpopulations</a>	102
4.1.9	<a href="#">Executing the simulation</a>	102
4.2	<a href="#">Basic output</a>	104
4.2.1	<a href="#">Entire population</a>	104
4.2.2	<a href="#">Random population sample</a>	106
4.2.3	<a href="#">Sampling individuals rather than genomes</a>	107
4.2.4	<a href="#">Substitutions</a>	109
4.2.5	<a href="#">Automatic logging with LogFile</a>	110
4.2.6	<a href="#">Custom output with Eidos</a>	112
4.2.7	<a href="#">The simulation endpoint</a>	115
5.	<a href="#">Demography and population structure</a>	116
5.1	<a href="#">Subpopulation size</a>	116
5.1.1	<a href="#">Instantaneous changes</a>	116
5.1.2	<a href="#">Exponential growth</a>	116
5.1.3	<a href="#">The population visualization graph</a>	120
5.1.4	<a href="#">Cyclical changes</a>	121
5.1.5	<a href="#">Context-dependent changes: Muller's Ratchet</a>	121
5.2	<a href="#">Population structure</a>	122
5.2.1	<a href="#">Adding subpopulations</a>	122
5.2.2	<a href="#">Removing subpopulations</a>	124
5.2.3	<a href="#">Splitting subpopulations</a>	125
5.2.4	<a href="#">Joining subpopulations</a>	125
5.3	<a href="#">Migration and admixture</a>	126
5.3.1	<a href="#">A linear stepping-stone model</a>	126
5.3.2	<a href="#">A non-spatial metapopulation</a>	128
5.3.3	<a href="#">A two-dimensional subpopulation matrix</a>	129
5.3.4	<a href="#">A random, sparse spatial metapopulation</a>	130
5.3.5	<a href="#">Reading a migration matrix from a file</a>	133
5.4	<a href="#">The Gravel et al. (2011) model of human evolution</a>	135
5.5	<a href="#">Rescaling population sizes to improve simulation performance</a>	141
6.	<a href="#">Sexual reproduction</a>	145
6.1	<a href="#">Recombination</a>	145
6.1.1	<a href="#">Crossing over: Making a random recombination map</a>	145
6.1.2	<a href="#">Crossing over: Reading a recombination map from a file</a>	146

6.1.3	<a href="#">Gene conversion</a>	148
6.1.4	<a href="#">Multiple chromosomes</a>	148
6.2	<a href="#">Separate sexes</a>	150
6.2.1	<a href="#">Enabling separate sexes</a>	150
6.2.2	<a href="#">Sex ratios</a>	150
6.2.3	<a href="#">Modeling sex-chromosome evolution</a>	152
6.3	<a href="#">Selfing and cloning</a>	153
6.3.1	<a href="#">Selfing in hermaphroditic populations</a>	153
6.3.2	<a href="#">Cloning</a>	154
7.	<a href="#">Mutation types, genomic elements, and chromosome structure</a>	155
7.1	<a href="#">Mutation types and fitness effects</a>	155
7.2	<a href="#">Genomic element types</a>	157
7.3	<a href="#">Chromosome organization</a>	158
7.4	<a href="#">Custom display colors in SLiMgui</a>	160
8.	<a href="#">SLiMgui visualizations for polymorphism patterns</a>	164
8.1	<a href="#">Mutation frequency spectra</a>	165
8.2	<a href="#">Mutation frequency trajectories</a>	166
8.3	<a href="#">Times to fixation and loss</a>	167
8.4	<a href="#">Population fitness over time</a>	168
9.	<a href="#">Selective sweeps</a>	171
9.1	<a href="#">Introducing adaptive mutations</a>	171
9.2	<a href="#">Making sweeps conditional on fixation</a>	173
9.3	<a href="#">Making sweeps conditional on establishment</a>	175
9.4	<a href="#">Partial sweeps</a>	177
9.5	<a href="#">Soft sweeps from de novo mutations</a>	178
9.5.1	<a href="#">A soft sweep from recurrent de novo mutations in a large population</a>	178
9.5.2	<a href="#">A soft sweep with a fixed mutation schedule</a>	180
9.5.3	<a href="#">A soft sweep with a random mutation schedule</a>	182
9.6	<a href="#">Sweeps from standing genetic variation</a>	184
9.6.1	<a href="#">A sweep from standing variation at a random locus</a>	184
9.6.2	<a href="#">A sweep from standing variation at a predetermined locus</a>	186
9.7	<a href="#">Adaptive introgression</a>	187
9.8	<a href="#">Fixation probabilities under Hill–Robertson interference</a>	189
9.9	<a href="#">Keeping a reference to a sweep mutation</a>	191
9.10	<a href="#">Tracking the fate of background mutations</a>	192
9.11	<a href="#">Effective population size versus census population size</a>	194
10.	<a href="#">Context-dependent selection using mutationEffect() callbacks</a>	201
10.1	<a href="#">Temporally varying selection</a>	201
10.2	<a href="#">Spatially varying selection</a>	202
10.3	<a href="#">Fitness as a function of genomic background</a>	204
10.3.1	<a href="#">Epistasis</a>	204
10.4	<a href="#">Fitness as a function of population composition</a>	206
10.4.1	<a href="#">Frequency-dependent selection</a>	206
10.4.2	<a href="#">Cultural effects on fitness</a>	208
10.4.3	<a href="#">Kin selection and the green-beard effect</a>	210

10.5	<a href="#">Changing selection coefficients with setSelectionCoeff()</a>	214
10.6	<a href="#">Varying the dominance coefficient among mutations</a>	215
11.	<a href="#">Complex mating schemes using mateChoice() callbacks</a>	219
11.1	<a href="#">Assortative mating</a>	219
11.2	<a href="#">Sequential mate search</a>	225
11.3	<a href="#">Gametophytic self-incompatibility</a>	229
12.	<a href="#">Direct child modifications using modifyChild() callbacks</a>	234
12.1	<a href="#">Social learning of cultural traits</a>	234
12.2	<a href="#">Lethal epistasis</a>	236
12.3	<a href="#">Simulating gene drive</a>	237
12.4	<a href="#">Suppressing hermaphroditic selfing</a>	241
12.5	<a href="#">Tracking separate sexes in script</a>	242
13.	<a href="#">Phenotypes, fitness functions, quantitative traits, and QTLs</a>	245
13.1	<a href="#">Polygenic selection</a>	246
13.2	<a href="#">A simple model of variable QTL effect sizes</a>	248
13.3	<a href="#">A model of discrete QTL effects with a structured chromosome</a>	251
13.4	<a href="#">A quantitative genetics model with heritability</a>	257
13.5	<a href="#">A QTL-based model with two quantitative phenotypic traits and pleiotropy</a>	262
13.6	<a href="#">A variety of fitness functions</a>	269
14.	<a href="#">Advanced models</a>	276
14.1	<a href="#">Relatedness, inbreeding, and heterozygosity</a>	276
14.2	<a href="#">Mortality-based fitness</a>	278
14.3	<a href="#">Reading initial simulation state from an MS output file</a>	281
14.4	<a href="#">Modeling chromosomal inversions with a recombination() callback</a>	284
14.5	<a href="#">Modeling both X and Y Chromosomes with a Pseudo-Autosomal Region (PAR)</a>	291
14.6	<a href="#">Estimating model parameters with ABC</a>	295
14.7	<a href="#">Tracking true local ancestry along the chromosome</a>	298
14.8	<a href="#">Live plotting with R using system()</a>	302
14.9	<a href="#">Modeling haploid organisms</a>	306
14.10	<a href="#">Using mutation rate variation to model varying functional density</a>	307
14.11	<a href="#">Modeling microsatellites</a>	309
14.12	<a href="#">Modeling transposable elements</a>	314
14.13	<a href="#">Modeling opposite ends of a chromosome</a>	320
14.14	<a href="#">Visualizing ancestry and admixture with mutation() callbacks</a>	323
14.15	<a href="#">Modeling biallelic loci with a mutation() callback</a>	326
15.	<a href="#">Continuous-space models and interactions</a>	334
15.1	<a href="#">A simple 2D continuous-space model</a>	334
15.2	<a href="#">Spatial competition</a>	336
15.3	<a href="#">Boundaries and boundary conditions</a>	338
15.4	<a href="#">Mate choice with a spatial kernel</a>	340
15.5	<a href="#">Mate choice with a nearest-neighbor search</a>	342
15.6	<a href="#">Divergence due to phenotypic competition with an interaction() callback</a>	344
15.7	<a href="#">Modeling phenotype as a spatial dimension</a>	349

15.8	<a href="#">Sympatric speciation facilitated by assortative mating</a>	352
15.9	<a href="#">Speciation due to spatial variation in selection</a>	355
15.10	<a href="#">A simple biogeographic landscape model</a>	360
15.11	<a href="#">Local adaptation on a heterogeneous landscape map</a>	365
15.12	<a href="#">Periodic spatial boundaries</a>	371
15.13	<a href="#">Density-dependent fecundity with summarizeIndividuals()</a>	375
16.	<a href="#">Going beyond Wright–Fisher models: nonWF model recipes</a>	382
16.1	<a href="#">A minimal nonWF model</a>	382
16.2	<a href="#">Age structure (a life table model)</a>	385
16.3	<a href="#">Monogamous mating and variation in litter size</a>	388
16.4	<a href="#">Beneficial mutations and absolute fitness</a>	390
16.5	<a href="#">A metapopulation extinction-colonization model</a>	393
16.6	<a href="#">Habitat choice</a>	396
16.7	<a href="#">Evolutionary rescue after environmental change</a>	400
16.8	<a href="#">Pollen flow</a>	405
16.9	<a href="#">Litter size and parental investment</a>	407
16.10	<a href="#">Spatial competition and spatial mate choice in a nonWF model</a>	411
16.11	<a href="#">A spatial model with carrying-capacity density</a>	416
16.12	<a href="#">Forcing a specific pedigree in a nonWF model</a>	420
16.13	<a href="#">Modeling clonal haploids in a nonWF model with addRecombinant()</a>	425
16.14	<a href="#">Modeling clonal haploid bacteria with horizontal gene transfer</a>	426
16.15	<a href="#">Implementing a Wright–Fisher model with a nonWF model</a>	429
16.16	<a href="#">Alternation of generations</a>	434
16.17	<a href="#">Meiotic drive</a>	437
16.18	<a href="#">A spatial epidemiological S-I-R model</a>	440
16.19	<a href="#">Range expansion in a stepping-stone model in two recipes</a>	446
16.20	<a href="#">Logistic population growth with the Beverton–Holt model</a>	452
16.21	<a href="#">Dynamic population structure in nonWF models</a>	456
16.22	<a href="#">Sperm storage with a survival() callback</a>	460
16.23	<a href="#">Tracking separate sexes in script, nonWF style</a>	465
16.24	<a href="#">Modeling haplodiploidy with addRecombinant()</a>	467
17.	<a href="#">Tree-sequence recording: tracking population history and true local ancestry</a>	471
17.1	<a href="#">A minimal tree-seq model</a>	471
17.2	<a href="#">Overlaying neutral mutations</a>	472
17.3	<a href="#">Simulation conditional upon fixation of a sweep, preserving ancestry</a>	474
17.4	<a href="#">Detecting the “dip in diversity”: analyzing tree heights in Python</a>	476
17.5	<a href="#">Mapping admixture: analyzing ancestry in Python</a>	479
17.6	<a href="#">Measuring the coalescence time of a model</a>	482
17.7	<a href="#">Analyzing selection coefficients in Python with tskit</a>	483
17.8	<a href="#">Starting a hermaphroditic WF model with a coalescent history</a>	485
17.9	<a href="#">Starting a sexual nonWF model with a coalescent history</a>	487
17.10	<a href="#">Adding a neutral burn-in after simulation with recapitation</a>	490

18.	<a href="#">Modeling explicit nucleotides</a>	495
18.1	<a href="#">A simple neutral nucleotide-based model</a>	495
18.2	<a href="#">Reading an ancestral nucleotide sequence from a FASTA file</a>	497
18.3	<a href="#">Sequence output from nucleotide-based models</a>	498
18.4	<a href="#">Back-mutations, independent mutational lineages, and VCF output</a>	501
18.5	<a href="#">Modeling elevated CpG mutation rates and equilibrium nucleotide frequencies</a>	503
18.6	<a href="#">A nucleotide-based model with introduced non-nucleotide-based mutations</a>	507
18.7	<a href="#">Using standard SLiM fitness effects with nucleotides: modeling synonymous sites</a>	509
18.8	<a href="#">Defining sequence-based fitness effects at the nucleotide level</a>	511
18.9	<a href="#">Defining sequence-based fitness effects at the amino acid level</a>	513
18.10	<a href="#">Varying the mutation rate along the chromosome in a nucleotide-based model</a>	515
18.11	<a href="#">Modeling GC-biased gene conversion (gBGC)</a>	516
18.12	<a href="#">Reading VCF files to create nucleotide-based SNPs</a>	518
18.13	<a href="#">Tree-sequence recording and nucleotide-based models</a>	522
18.14	<a href="#">Modeling identity by state (IBS): uniquing mutations with a mutation() callback</a>	525
18.15	<a href="#">Modeling identity by state (IBS): uniquing back-mutations to the ancestral state</a>	528
19.	<a href="#">Multispecies modeling</a>	532
19.1	<a href="#">A simple multispecies model</a>	532
19.2	<a href="#">A two-species model</a>	533
19.3	<a href="#">A deterministic host–parasitoid model</a>	536
19.4	<a href="#">An individual-based host–parasitoid model</a>	540
19.5	<a href="#">A continuous-space host–parasitoid model</a>	547
19.6	<a href="#">A coevolutionary host–parasitoid trait-matching model</a>	553
19.7	<a href="#">A coevolutionary host–parasite matching-allele model</a>	558
19.8	<a href="#">Within-host reproduction in a host–pathogen model</a>	563
20.	<a href="#">Runtime control</a>	572
20.1	<a href="#">The random number generator</a>	572
20.2	<a href="#">Defining constants on the command line</a>	573
20.3	<a href="#">Other command-line options</a>	576
20.4	<a href="#">File input and output</a>	577
20.5	<a href="#">Lambda execution</a>	578
20.6	<a href="#">Debugging</a>	580
21.	<a href="#">Implementation and performance</a>	584
21.1	<a href="#">Writing fast SLiM simulations</a>	584
21.2	<a href="#">Performance evaluation</a>	586
21.3	<a href="#">Memory usage considerations</a>	588
21.4	<a href="#">Mutation runs and runtime optimization</a>	589
21.5	<a href="#">Profiling simulations in SLiMgui</a>	592
21.6	<a href="#">Profiling memory usage in SLiMgui, or with outputUsage()</a>	599
 PART II: THE SLiM REFERENCE		
22.	<a href="#">SLiM architecture (WF models)</a>	603
22.0	<a href="#">Step 0: Execution of first() Eidos events</a>	603

22.1	<a href="#">Step 1: Execution of early() Eidos events</a>	604
22.2	<a href="#">Step 2: Generation of offspring</a>	604
22.2.1	<a href="#">The order of offspring generation</a>	604
22.2.2	<a href="#">Mate choice</a>	605
22.2.3	<a href="#">Mutation and recombination</a>	605
22.2.4	<a href="#">Child modification</a>	607
22.2.5	<a href="#">Child generation</a>	607
22.3	<a href="#">Step 3: Removal of fixed mutations</a>	608
22.4	<a href="#">Step 4: Offspring become parents</a>	608
22.5	<a href="#">Step 5: Execution of late() Eidos events</a>	608
22.6	<a href="#">Step 6: Fitness value recalculation</a>	609
22.7	<a href="#">Step 7: Tick/cycle count increment</a>	610
23.	<a href="#">SLiM architecture (nonWF models)</a>	611
23.0	<a href="#">Step 0: Execution of first() Eidos events</a>	611
23.1	<a href="#">Step 1: Generation of offspring</a>	611
23.1.1	<a href="#">The order of offspring generation</a>	612
23.1.2	<a href="#">Individual-based reproduction with reproduction() callbacks</a>	612
23.1.3	<a href="#">Mutation and recombination</a>	613
23.1.4	<a href="#">Child modification</a>	613
23.1.5	<a href="#">Child generation</a>	613
23.2	<a href="#">Step 2: Execution of early() Eidos events</a>	613
23.3	<a href="#">Step 3: Fitness value recalculation</a>	614
23.4	<a href="#">Step 4: Viability/survival</a>	614
23.5	<a href="#">Step 5: Removal of fixed mutations</a>	616
23.6	<a href="#">Step 6: Execution of late() Eidos events</a>	616
23.7	<a href="#">Step 7: Tick/cycle count increment</a>	616
24.	<a href="#">SLiM classes and built-in functions</a>	617
24.1	<a href="#">Simulation initialization: initialize() callbacks</a>	617
24.2	<a href="#">Class Chromosome</a>	629
24.2.1	<a href="#">Chromosome properties</a>	629
24.2.2	<a href="#">Chromosome methods</a>	632
24.3	<a href="#">Class Community</a>	635
24.3.1	<a href="#">Community properties</a>	635
24.3.2	<a href="#">Community methods</a>	636
24.4	<a href="#">Class Genome</a>	639
24.4.1	<a href="#">Genome properties</a>	639
24.4.2	<a href="#">Genome methods</a>	640
24.5	<a href="#">Class GenomicElement</a>	647
24.5.1	<a href="#">GenomicElement properties</a>	647
24.5.2	<a href="#">GenomicElement methods</a>	647
24.6	<a href="#">Class GenomicElementType</a>	647
24.6.1	<a href="#">GenomicElementType properties</a>	648
24.6.2	<a href="#">GenomicElementType methods</a>	648
24.7	<a href="#">Class Individual</a>	648
24.7.1	<a href="#">Individual properties</a>	649
24.7.2	<a href="#">Individual methods</a>	652



24.8	<a href="#">Class InteractionType</a>	654
24.8.1	<a href="#">InteractionType properties</a>	658
24.8.2	<a href="#">InteractionType methods</a>	658
24.9	<a href="#">Class LogFile</a>	665
24.9.1	<a href="#">LogFile properties</a>	666
24.9.2	<a href="#">LogFile methods</a>	666
24.10	<a href="#">Class Mutation</a>	669
24.10.1	<a href="#">Mutation properties</a>	669
24.10.2	<a href="#">Mutation methods</a>	670
24.11	<a href="#">Class MutationType</a>	671
24.11.1	<a href="#">MutationType properties</a>	671
24.11.2	<a href="#">MutationType methods</a>	674
24.12	<a href="#">Class SLiMEidosBlock</a>	674
24.12.1	<a href="#">SLiMEidosBlock properties</a>	675
24.12.2	<a href="#">SLiMEidosBlock methods</a>	676
24.13	<a href="#">Class SLiMgui</a>	676
24.13.1	<a href="#">SLiMgui properties</a>	676
24.13.2	<a href="#">SLiMgui methods</a>	676
24.14	<a href="#">Class Species</a>	677
24.14.1	<a href="#">Species properties</a>	677
24.14.2	<a href="#">Species methods</a>	678
24.15	<a href="#">Class Subpopulation</a>	689
24.15.1	<a href="#">Subpopulation properties</a>	689
24.15.2	<a href="#">Subpopulation methods</a>	691
24.16	<a href="#">Class Substitution</a>	702
24.16.1	<a href="#">Substitution properties</a>	702
24.16.2	<a href="#">Substitution methods</a>	703
24.17	<a href="#">SLiM built-in functions</a>	703
24.17.1	<a href="#">Nucleotide utilities</a>	703
24.17.2	<a href="#">Population genetics utilities</a>	705
24.17.3	<a href="#">Other utilities</a>	708
25.	<a href="#">Writing Eidos events and callbacks</a>	711
25.1	<a href="#">Defining Eidos events</a>	711
25.2	<a href="#">Defining mutation effects with a mutationEffect() callback</a>	712
25.3	<a href="#">Defining individual-level fitness effects with a fitnessEffect() callback</a>	715
25.4	<a href="#">Defining mate choice with a mateChoice() callback</a>	716
25.5	<a href="#">Defining child generation with a modifyChild() callback</a>	718
25.6	<a href="#">Defining recombination behavior with a recombination() callback</a>	720
25.7	<a href="#">Defining interaction behavior with a interaction() callback</a>	722
25.8	<a href="#">Defining reproduction behavior with a reproduction() callback</a>	724
25.9	<a href="#">Defining mutation behavior with a mutation() callback</a>	725
25.10	<a href="#">Defining survival and mortality with a survival() callback</a>	727
25.11	<a href="#">Further details on Eidos events and callbacks</a>	728
26.	<a href="#">SLiM output formats</a>	730
26.1	<a href="#">Species output methods</a>	730

26.1.1	<a href="#">outputFull()</a> .....	731
26.1.2	<a href="#">outputFixedMutations()</a> .....	733
26.1.3	<a href="#">outputMutations()</a> .....	734
26.2	<a href="#">Subpopulation output methods</a> .....	735
26.2.1	<a href="#">outputSample()</a> .....	735
26.2.2	<a href="#">outputMSSample()</a> .....	736
26.2.3	<a href="#">outputVCFsample()</a> .....	737
26.2.4	<a href="#">outputVCFsample() in nucleotide-based models</a> .....	739
26.3	<a href="#">Genome output methods</a> .....	742
26.3.1	<a href="#">output()</a> .....	743
26.3.2	<a href="#">outputMS()</a> .....	743
26.3.3	<a href="#">outputVCF()</a> .....	743
26.4	<a href="#">SLiM additions to the .trees file format</a> .....	744
27.	<a href="#">SLiM extensions to the Eidos language</a> .....	753
27.1	<a href="#">Extensions to the Eidos grammar</a> .....	753
27.2	<a href="#">SLiM scoping rules</a> .....	754
28.	<a href="#">SLiM reference sheet</a> .....	758
29.	<a href="#">Revision history</a> .....	766
30.	<a href="#">Credits and licenses for incorporated software</a> .....	780
31.	<a href="#">References</a> .....	786

## PART I: THE SLIM COOKBOOK

# 1. SLiM overview

## 1.1 Introduction

SLiM is an evolutionary simulation package that provides facilities for very easily and quickly constructing genetically explicit individual-based evolutionary models. By default, SLiM is based upon a Wright–Fisher or “WF” model of evolution; in particular, (1) generations are non-overlapping and discrete, (2) the probability of an individual being chosen as a parent for a child in the next generation is proportional to the individual’s fitness, (3) individuals are diploid, and (4) offspring are generated by recombination of parental chromosomes with the addition of new mutations. Some of these assumptions can be relaxed in WF models using techniques described in this manual, and an alternative non-Wright–Fisher or “nonWF” type of model can be used for even greater flexibility; nevertheless, the default WF model type is the conceptual foundation of SLiM, and it should be understood thoroughly before venturing into more advanced models.

The original version of SLiM (through version 1.8; Messer 2013) was written by Philipp Messer, now of Cornell University; its name stands for **S**election on **L**inked **M**utations. SLiM 2 and later – the subject of this manual, hereafter simply referred to as SLiM – is a ground-up redesign of SLiM (by Benjamin C. Haller, now of the Messer Lab at Cornell) that provides much greater power, flexibility, and speed on top of the same foundational architecture as the original.

SLiM is based upon two main components: a simple scripting language called Eidos that was invented for use with SLiM, and a set of Eidos classes that implement entities such as subpopulations, mutations, and chromosomes. A minimal SLiM simulation, such as you will see in section 4.1, comprises just a few lines of Eidos code; virtually all of the simulation details are handled by SLiM, so the Eidos script needs only to set up basic parameters such as the population size and mutation rate. Because of the extensibility provided by Eidos, however, it is straightforward to extend such a simulation to model almost any evolutionary scenario.

Regardless of the specific problem studied, evolutionary simulations will often entail common design elements: multiple subpopulations connected by migration, for example, or selective sweeps, or spatial and temporal variation in selection. Such design elements will come up over and over for users of SLiM, and it might not be obvious – particularly to biologists with little programming experience – how to model them using the toolbox provided by Eidos and SLiM. The first part of this manual has thus been structured as a “cookbook”, an assemblage of recipes showing how to build different sorts of models in SLiM. The design of each recipe will be explained, so that users of SLiM feel comfortable modifying the recipes to build their own models.

Under the hood, SLiM is a complex piece of software, with dozens of source files devoted to implementing both the Eidos language and the Eidos classes provided by SLiM. However, users of SLiM should not need to confront that complexity; SLiM users should literally never need to delve into the C++ and C code that drives SLiM. All that you need to understand, as an end user, is the Eidos scripting interface that SLiM presents for your use. Understanding the Eidos language itself is an important part of using SLiM effectively; this manual will briefly introduce Eidos concepts as they arise, but for a more thorough and complete introduction to Eidos it is recommended that you refer to its separate manual (Haller 2016). Eidos is similar to the popular R language (R Core Team 2015); if you have used R, Eidos should feel fairly natural. (The Eidos manual discusses why we invented a new language for SLiM, rather than using an existing language.)

This manual does not need to be read from cover to cover; each recipe is designed to stand alone to the extent possible, although concepts do build on each other. We recommend that all SLiM users read at least this introductory chapter, which lays out the conceptual foundations of SLiM. We also urge beginning users to start by doing the **SLiM Workshop**, which is available both in-person and online; see section 1.10 for more information on it.

## 1.2 Why SLiM?

Many evolutionary simulation packages already exist, from custom-built one-off models that address only a single problem, to simulation toolkits designed to fit a wide variety of tasks. It would be reasonable to ask why we have made yet another toolkit, and what sets SLiM apart. This section will explain what SLiM is designed to do and why it adds something important and unique to existing modeling tools. Note that here we are discussing SLiM 2 and later, which is quite different in its design philosophy and approach than earlier versions of SLiM.

The primary reason for SLiM is flexibility. Most evolutionary modeling toolkits, including SLiM 1.8, are quite limited in their abilities. They are not easily extensible or modifiable; typically, if you wish to make such a toolkit do something new, you need to modify the actual source code (often C or C++) in which the toolkit is written – a non-trivial task. Some toolkits embrace this shortcoming as a feature, and are thus designed as a set of C++ templates or other reusable objects; but again, although this approach is flexible, a great deal of programming experience is needed. Because existing toolkits are so difficult to modify and extend, it is often simpler for researchers to write their own purpose-built model from scratch; however, this also entails substantial drawbacks. First, it involves reinventing the wheel over and over. Second, it limits the level of complexity attainable, since each research project begins again more or less from scratch. Third, it is a very bug-prone approach, since the code underlying such simulations is often complex, and all of the debugging and testing that went into previous models is lost with each new model. SLiM is designed to radically simplify the process of making an evolutionary model, because of the way that the inner mechanisms of the SLiM simulation are exposed in the Eidos scripting language. Modifying a simulation to add a new and complex behavior such as epistasis or sequential mate choice can often be expressed in just a couple of lines of simple Eidos code – a much simpler proposition than trying to do the same thing in the underlying C++ in which the toolkit is written. The underlying SLiM engine is quite complex – it contains a full interpreter for the Eidos language – but it can be treated as a black box, and never needs to be modified or understood by the end user at all. The script that drives a particular simulation, on the other hand, can usually be trivially short, easily understood, and quickly modified. The end result is power and flexibility coupled with simplicity and reusability.

A second reason for SLiM's existence is performance. When writing a one-off model, it is often prohibitive – in terms of both time and effort – to optimize the model for fast execution, and once code is optimized it becomes much harder to maintain and modify, hindering reusability. Because SLiM will be used for many different models, however, we deemed it worthwhile to spend the effort on optimization. Months of hard work have gone into making SLiM run fast, including a number of complex and non-obvious algorithmic optimizations. By using SLiM, you get all of the speed benefits of those optimizations for free. Individual-based simulations are often speed-limited; one is often forced to explore only a limited range of parameter space, or use smaller population sizes than desired, or make other such compromises because of limited computing resources. SLiM's speed helps to lift that constraint.

A third reason for SLiM's existence is to provide interactive execution and graphical debugging. With SLiMgui, you can visualize your simulation as it runs, with graphical depictions of mutations, genomic elements, subpopulations, migration patterns, and simulation metrics. You can single-step through your simulations, examine the values of all of the underlying objects, and even execute arbitrary Eidos code to modify your simulation as it runs. This allows much more rapid and bug-free simulation development. We highly recommend that you use SLiMgui for model development even if your production runs will be on a cluster; the value of graphical model development and debugging is immense (Grimm 2002).

We believe that flexibility, performance, and graphical interactivity make SLiM a worthy addition to the ecosystem of evolutionary modeling toolkits. We hope that you will agree.

### 1.3 A quick summary of SLiM

This section is a quick summary of SLiM's design, as a brief introduction to provide context for the recipes in this cookbook. See part II of this manual, the SLiM Reference, for further details.

**Glossary.** We begin with a glossary of terms that have a special meaning in SLiM:

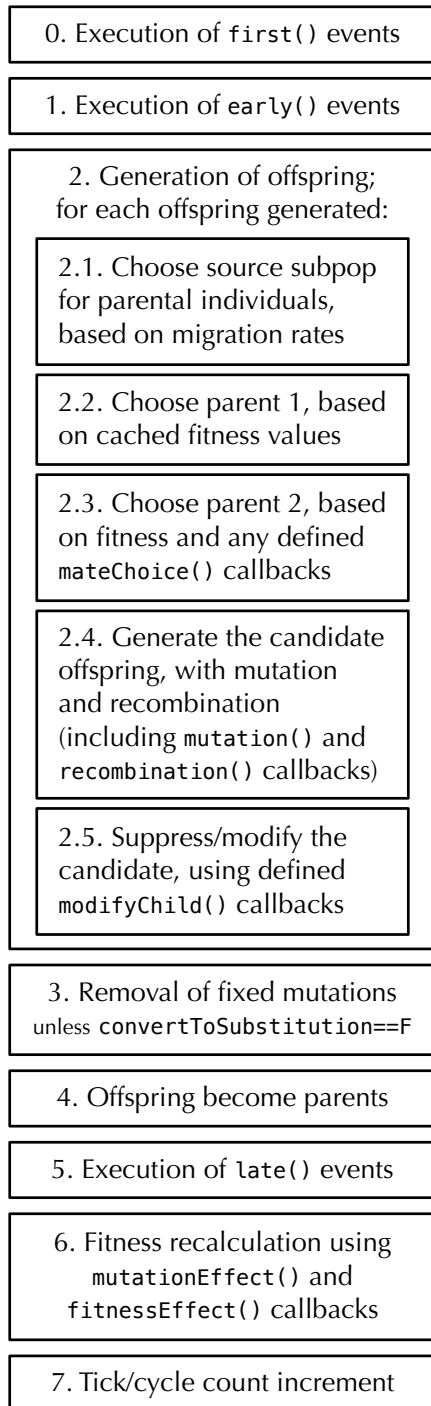
- community*: The *community* in SLiM, of Eidos class *Community*, is the top-level object that manages the simulation. It contains all of the species objects in the simulation.
- species*: A *species* in SLiM, of class *Species*, corresponds to one species in the simulation. All other objects related to the species are contained within the species object.
- population*: A *population* in SLiM comprises all of the individuals being simulated by one species in SLiM. There is no Eidos object corresponding to the population; the species object handles everything related to the population.
- subpopulation*: A population is divided into *subpopulations*, discrete groups of individuals which may or may not be connected to each other by migration. The Eidos class *Subpopulation* is used to represent the subpopulations in a species.
- individual*: An *individual*, in SLiM, is a diploid organism composed of two haploid *genomes* (see below). Individuals are represented by the Eidos class *Individual*. The individual is the conceptual level at which fitness is computed, mate choice is conducted, and so forth.
- genome*: A *genome* is the haploid set of all mutations occurring in one of the two homologous chromosomes of a diploid individual; diploid individuals thus contain two genome objects. (In haploid individuals, the second genome in SLiM is not used and is kept empty.) The Eidos class *Genome* is used to represent genomes.
- mutation*: A *mutation* is a change in the genetic information of an individual, represented by the Eidos class *Mutation*. Mutations have a position in the genome, a selection coefficient, information about when/where they arose, and (in nucleotide-based models) an associated nucleotide. Each mutation references a *mutation type* that governs additional properties of the mutation, such as its dominance coefficient.
- mutation type*: Mutations are drawn from a particular *mutation type*, representing simulation-dependent categories of mutations (neutral, beneficial, lethal, synonymous, etc.). In general, the mutation type determines the distribution of fitness effects (DFE) from which mutations of that mutation type are drawn. The mutation type also determines the dominance coefficient of all mutations of that type. The Eidos class *MutationType* is used to represent mutation types in SLiM.
- chromosome*: In SLiM's terminology, the *chromosome* is the positional map of regions, such as genes, being modeled for a species by SLiM; the term does *not* refer to a single chromosome carried by a particular individual (the term *genome* is used for that purpose). The chromosome, represented by the Eidos class *Chromosome*, defines regions according to both their recombination rate and their mutational profile.
- genomic element*: A chromosome is spanned by non-overlapping *genomic elements*, of Eidos class *GenomicElement*, each referencing a *genomic element type* (see below).
- genomic element type*: A *genomic element type* defines the particular mutation types that can occur in genomic elements of the given type. It is represented by Eidos class *GenomicElementType*. Biological examples of genomic element types could be introns, exons, or non-coding regions. All of the genomic elements referencing a type use that type's mutational profile to generate new mutations.
- substitution*: When mutations reach fixation in an entire population, they are often replaced by substitution objects, of Eidos class *Substitution*, for efficiency reasons. The substitution provides a permanent record of the fixed mutation's characteristics.



**Population structure.** SLiM allows arbitrary population structure; you can model any number of subpopulations, connected by any pattern of migration, and subpopulations may be created, change size, and be removed at any time. In WF models, mate choice occurs within subpopulations – adult organisms do not migrate or mate between subpopulations – and migration occurs at the juvenile stage. (These restrictions are relaxed in nonWF models, which are more flexible in many ways; see section 1.6). Individuals in SLiM are always diploid, and gametes are

always haploid; SLiM does not provide much intrinsic support for other ploidy levels (but haploids can easily be modeled by modifying the default behavior with scripting; see section 14.9). See chapter 5 for further discussion.

#### *The WF model's tick cycle*



**Sexual reproduction.** SLiM can model either hermaphroditic individuals (no distinction between sexes) or sexual individuals (distinct males and females). In either case, individuals normally undergo biparental mating to produce offspring through sexual recombination (including both crossing over and, optionally, gene conversion). Clonal reproduction is also supported, instead of or in addition to biparental mating, and in the hermaphroditic case, SLiM also supports self-fertilization (“selfing”). In sexual models the sex ratio is controllable, and sex chromosomes may be modeled. These topics are discussed further in chapter 6.

**Genetics.** SLiM is genetically explicit: it models mutations at specific base positions in genomes with an explicit chromosome structure. SLiM does not, by default, model nucleotide sequences, but it can; see section 1.8 and chapter 18. The chromosome is composed of genomic elements (e.g., sections of a gene), each of a particular genomic element type (e.g., intron versus exon) that defines the mutational profile of genomic elements of that type, using a set of mutation types and associated probabilities. These topics are discussed further in chapter 7.

**Fitness.** By default, SLiM calculates fitness multiplicatively, based upon all of the mutations possessed by each individual. The selection coefficient  $s$  of a given mutation defines the mutation’s fitness effect when homozygous ( $1+s$ ); when heterozygous, the fitness effect is modified by a dominance coefficient  $h$  ( $1+hs$ ). The effects of mutations may be altered by `mutationEffect()` callbacks that provide full control over how the fitness of an individual is calculated given the particular set of mutations present in its genome, and possibly other properties of the population. These topics are discussed further in chapter 10.

**Tick cycle.** SLiM is based, by default, on an extended Wright–Fisher or “WF” model with non-overlapping, discrete generations (a non-Wright–Fisher or “nonWF” model can also be used, as discussed in section 1.6 and chapters 16 and 23, but that is an advanced topic that we will pass over here). Within each generation, events occur in a fixed order (see left). A generation in SLiM is also often

called a *tick* or a *cycle*, but those terms having slightly different meanings, especially in nonWF models and multispecies models, discussed further in section 1.5.7; we will lean towards using the term “tick” since it applies correctly in all of those situations. Each tick begins with the execution of user-defined Eidos scripts called `first()` and `early()` events. Examples of such events might be demographic events, such as changes in population size, population splits, changes in migration rates, etc. Offspring are then generated by drawing gametes from the parent population according to fitness; this default mating scheme may be modified to implement non-standard mating scenarios via user-defined `mateChoice()` callbacks (see chapter 11). Gametes are generated from the genomes of the candidate parents, modified by mutation and recombination; furthermore, the standard mutation behavior can be modified using a `mutation()` callback (see section 25.9), and the standard user-defined recombination map can be modified arbitrarily for each gamete generated using a `recombination()` callback (see sections 14.4 and 25.6). After offspring have been created, their genomes and other attributes can be modified according to user-defined rules using `modifyChild()` callbacks (see chapter 12). After (optional) removal of fixed mutations from the model, the offspring become the parents. Next is another opportunity for Eidos events – in this case, `late()` events – to run. This is where output events, such as drawing a random sample of individuals from the population, would typically be specified. Fitness values are then calculated, modified by `mutationEffect()` callbacks (see chapter 10). Finally, the simulation advances to the next tick. This sequence of events is called the “tick cycle”.

**Tags.** User-defined “tag” values can be attached to almost all of the objects defined by SLiM in order to associate your own information with SLiM’s objects, whether short-term flags or long-term state. Tags are used in many recipes in this cookbook; see those recipes for a variety of examples of the utility of tags. One may also attach values to most SLiM objects using a dictionary-like `getValue()` / `setValue()` mechanism that associates a *value* (which may be an entire vector or matrix) with a named *key* on the target object. This facility, which is provided by the Eidos class `Dictionary` (acting as the superclass for many Eidos classes in SLiM), provides an even broader and more flexible way to attach model state to objects; see the Eidos manual for details on the `getValue()` / `setValue()` methods, and see the recipe in section 11.1 for an example of their use.

**Continuous space.** Beginning in SLiM 2.3, SLiM provides support for continuous space. If this optional feature is enabled, individuals in SLiM maintain a spatial position – either (x), (x, y), or (x, y, z) – within their subpopulation. These spatial positions can be changed at any time (simulating phenomena such as foraging and dispersal, for example), and are used to create a spatial visualization of the subpopulation in SLiMgui. Positions can be used in script in any way, allowing models to incorporate the consequences of continuous space in any way desired. In particular, spatial positions may be used as the basis for spatial interactions between individuals (see below), and may be used in conjunction with spatial maps that define variation in environmental variables across continuous space. This advanced feature is first introduced in recipes in chapter 15.

**Interactions.** Beginning in SLiM 2.3, SLiM adds a new class, `InteractionType`, which can govern interactions between individuals. Interactions can still be handled with pure Eidos code, but the use of `InteractionType` automates and accelerates many common tasks, such as finding the total interaction strength felt by an individual (as a result of competition, for example). `InteractionType` can also manage spatial interactions, providing features such as interaction strengths that vary according to distance, and handling spatial queries such as nearest-neighbor searches. This advanced feature is first introduced in recipes in chapter 15.

**Multiple species.** Beginning in SLiM 4.0, SLiM allows the simulation of more than one species in a single SLiM model, opening the door to ecological interactions and coevolutionary dynamics. Each species has its own genetic structure (chromosome, mutation types, genomic element types), its own mutations, and its own population and subpopulations; the only configuration choice that



all species must share is that they must all use the same model type – WF or nonWF. The tick cycles of species can be run in an interleaved manner, or can run separately. This advanced feature is summarized in section 1.9, and first used in chapter 19.

At this point, let's take a step back to see where we're heading. Beyond the brief overview above, this chapter will introduce some more specialized topics. Section 1.4 sketches out some practical details about how SLiM is typically used. Section 1.5 provides a more detailed overview of some of the concepts above, including how mutations are represented in SLiM, how SLiM models recombination between genomes, and what happens when mutations fix; it is essential reading. Section 1.6 then introduces non-Wright–Fisher or “nonWF” models, section 1.7 introduces tree-sequence recording, section 1.8 introduces nucleotide-based models, and section 1.9 introduces multispecies models; these are all advanced topics, but it is good to be aware of the existence of these features and the reasons why you might wish to use them. Finally, section 1.10 wraps up this introductory chapter with a summary of various online resources available for SLiM users – importantly, including free **SLiM Workshops** that can be attended in-person or completed online. That can be a good way for beginners to get going on SLiM.

Chapter 2 provides instructions on building and installing SLiM on various platforms. Chapter 3 gives an introduction to SLiMgui, the graphical modeling environment provided for use on macOS, Linux, and Windows. Most of the remainder of Part I of this manual, the SLiM Cookbook, then provides “recipes” demonstrating the core concepts of SLiM. Two chapters at the end of Part I are not so recipe-focused: chapter 20 discusses a variety of runtime control issues such as the random number generator used by SLiM, command-line options supported by SLiM, and so forth, while chapter 21 covers issues related to implementation and performance: how to write models that are fast and memory-efficient, and how to profile the performance of your model to find “hot spots” in need of optimization.

Finally, Part II of this manual, the SLiM Reference, provides technical reference documentation for SLiM, including such aspects as the tick cycle in WF and nonWF models, the Eidos classes provided by SLiM, the various types of events and callbacks, and the output formats supported by SLiM, beginning in chapter 22.

## 1.4 The typical SLiM usage pattern

Before delving more deeply into the concepts introduced in the previous section, it might be helpful to clarify how a typical user would use SLiM, in practical terms.

First of all, many or most users will use the SLiMgui modeling environment for their model development and testing, and perhaps for exploratory, non-production runs as well. SLiMgui provides many tools for model development, such as code completion, syntax coloring, online documentation, interactive model execution, live graphing of common model metrics, performance analysis, and graphical debugging. It also makes model development logistically simpler; there is no need to write a dispatch script, no need to execute Unix commands in a terminal, etc. When learning how to use SLiM, it is therefore strongly recommended that you use SLiMgui, which now runs on macOS, Linux, and Windows. Note that all of the recipes in this manual are directly accessible in SLiMgui through the Open Recipe submenu of the File menu.

Second, many or most SLiM users will do their “production” model runs on a computing cluster. One reason is that individual-based models often take a long time to execute; SLiM is highly optimized, but simulating the genomic details of a large number of individuals over many generations is inevitably slow. Another reason is that many replicate runs are typically needed; one run of an individual-based model provides just one data point, one solitary example of what *could* happen, so one usually needs to perform many runs and then use statistical methods to draw inferences (just as one often would in field-based or lab-based research). Finally, most studies

involving individual-based modeling explore a “parameter space”, examining how the model’s outcome depends upon the parameters of the model; each set of parameter values explored implies another full set of replicated runs. Together, these facts mean that a single study using SLiM might entail months or even years of processor time; the use of a computing cluster is thus often necessary.

For this reason, SLiM is designed to fully utilize a single processor; it is not designed to take advantage of multiple processors using multithreading or MPI. A single run of the `slim` command runs a given model a single time on a single processor; to conduct the many runs that are typically needed, `slim` will be run many times. This single-threaded design makes it straightforward for the user to run a separate instance of a model on each processor on a multicore machine or a computing cluster. This can be done manually in some cases, but is more typically done using a batch-queueing system such as Open Grid Scheduler. Either way, some sort of a dispatch script is generally needed to schedule each of the individual runs of `slim`. Because there are so many different possible ways that the user might want to run SLiM, and so many different computing environments in which it might be run, a standard dispatch script is not provided as a part of the SLiM package; however, this is usually straightforward. It can usually be done in whatever scripting language you prefer, from R or Python to a Bash shell script, and often just consists of a loop over all of the parameter values and replicates desired, with a call to launch or schedule a run of `slim` inside that loop. In Python, sublaunching a Unix process can be done with the `subprocess` package; in R, with `system()` or `system2()`; in a Bash shell script, by just invoking `slim` directly. A couple of examples of dispatch scripts for use on a local machine are provided in the SLiM-Extras repository at <https://github.com/MesserLab/SLiM-Extras/tree/master/sublaunching>, including a sublaunching tutorial by Sam Champer (see section 1.10). If you are working on an institutional computing cluster, the cluster administrator may be able to provide you with examples of dispatch scripts appropriate for that environment.

Finally, SLiM users will typically want to collect results from model runs and perform statistics and other analyses on them. This can sometimes be done directly in the dispatch script; that script might collect the model output and tabulate simple results as runs complete. In other cases, each invocation of `slim` will be set up to produce its own output files, and then a separate analysis script – typically written in a language like R or Python that has support for statistics and plotting – will read in those output files, parse the relevant information out of them, and conduct the desired analyses. In this undertaking, you are on your own. However, it is worth noting that SLiM can generate output in some standard file formats, such as FASTA, VCF, and MS (as well as CSV and TSV for non-genetic data), and that many tools already exist to read in and analyze such standard-format files, so in some cases you might be able to use pre-existing software for at least some of your analysis. If your model uses tree-sequence recording (see section 1.7), you can also output a `.trees` file that can be read and processed in Python using the `tskit`, `msprime`, and `pyslim` packages, making many types of post-run analysis much easier (see examples in chapter 17); indeed, this could in itself be a compelling reason to use tree-sequence recording, since parsing output files to do analysis is otherwise such an annoyance. Finally, in some cases you might want to do some of the needed analysis inside the SLiM model itself, in Eidos code, to simplify the post-processing needed. For example, if your analysis needs the number of mutations fixed at the end of each model run, it might be simpler to count the fixed mutations inside the SLiM model, and just output that count, rather than parsing full genomic output files from each model run just to extract the count of fixed mutations. The `LogFile` class in SLiM is designed to facilitate automatic logging of summary statistics and other metrics (see section 4.2.5). It is even possible to generate plots with R or Python from inside a SLiM model with sublaunching (see section 14.8), a technique that can be useful in some scenarios. In short, output and analysis present complex issues, and it

will be beneficial to think, up front, about how to design your model and your analysis code so that they communicate as cleanly and simply as possible.

## 1.5 Conceptual overview

This section will delve into further detail on some of the concepts set out in section 1.3 (which should be read before this section), to present a more complete picture of how SLiM works at a conceptual level. We will not show any Eidos code here; that will be left for the recipes in the “cookbook” that begins in chapter 4. We will, however, make reference to the Eidos classes used by SLiM to represent various concepts, and to some of the properties and methods of those classes. We will gloss over some minor details here in order to present the big picture as clearly as possible; for more comprehensive information, see Part II of this manual, the SLiM Reference that begins in chapter 22.

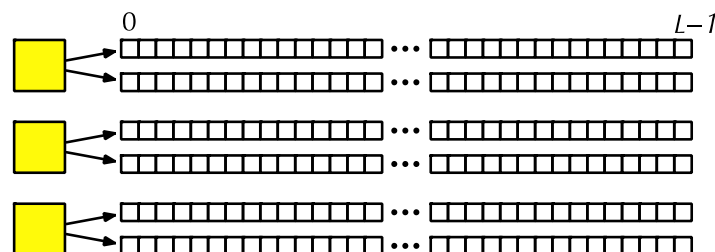
### 1.5.1 Individuals and genomes

SLiM is a framework for running individual-based models; this means that every individual organism in the model is simulated explicitly. Each individual is represented in SLiM as an instance of the `Individual` class (see section 24.7) in the Eidos scripting language. At the most minimal level individuals are born and die, and in between they find mates and produce offspring (or they reproduce by selfing or cloning); these actions are built into SLiM. In more complex models individuals may also do things like gather resources, learn from experience or through social learning, interact with other individuals, be subject to events that alter their state, move in discrete or continuous space, and exhibit various other behaviors; these actions are not built into SLiM (although supporting infrastructure is provided for some of them), but they can easily be implemented in Eidos script.

Perhaps most importantly, since SLiM models are genetically explicit simulations, individuals contain genetic information. Individuals in SLiM are diploid; each individual thus possesses two homologous chromosomes (or one X and one Y chromosome, if sex chromosomes are being simulated), referred to as the *genomes* of that individual. (It is possible to model more than one chromosome in SLiM, conceptually, but this is done by using a recombination map that specifies free recombination at particular positions, effectively subdividing the chromosome into unlinked sub-chromosomes; see section 6.1.4.) Each of the two genomes of an individual is represented using an instance of the `Genome` class in Eidos (see section 24.4).

A genome is essentially a container that holds a set of mutations. If both of an individual’s genomes contain exactly the same mutation (a surprisingly subtle concept, which will be defined rigorously in the next subsection), the individual is homozygous for that mutation; if a given mutation is contained in only one of the two genomes, the individual is heterozygous for that mutation. Note that SLiM does not model nucleotides explicitly by default, but it does model explicit, discrete base positions along the genome. (Beginning with version 3.3, SLiM can in fact model nucleotides explicitly, but that is an advanced feature that will not be discussed in this overview; see section 1.8 and chapter 18.)

The overall picture, then, looks like this:



Each yellow square represents one individual. Each individual contains two genomes; and each genome contains the state of all of the base positions along the chromosome, from beginning (position 0) to end (position  $L-1$ , so that the chromosome is of length  $L$ ). Each base position is represented here as an empty box, and all of the boxes from 0 to  $L-1$  together represent a genome.

A key concept in SLiM is that genomes begin, by default, as empty: they contain no mutations and no genetic information. This can be thought of as the “wild type”, in a sense, and we will refer to it in that way sometimes in what follows, but it does not have to represent the wild-type state of the organism you are modeling. Instead, it simply represents the base, un-mutated state of individuals, whatever you want to consider that to be. All mutations in SLiM can be thought of as modifications that are layered on top of this empty base state. The base state can also be thought of as “neutral”, in terms of fitness, but it does not have to actually be neutral (i.e., 1.0) in absolute fitness. Instead, the base state can have any absolute fitness value you like – but in general that is unimportant, since SLiM’s core engine is only concerned with relative fitness (at least in WF models, the default mode of operation, which we will limit ourselves to in this discussion). When the simulation begins, and all genomes are empty, it does not matter to SLiM what the absolute fitness of those empty genomes is; since they all have the *same* absolute fitness, they all have a *relative* fitness of 1.0, and that is what matters to SLiM (again, in WF models). The fitness effects of mutations then modify those relative fitnesses, multiplicatively. (See section 8.4 for a more complete discussion of what “fitness” really means in SLiM.)

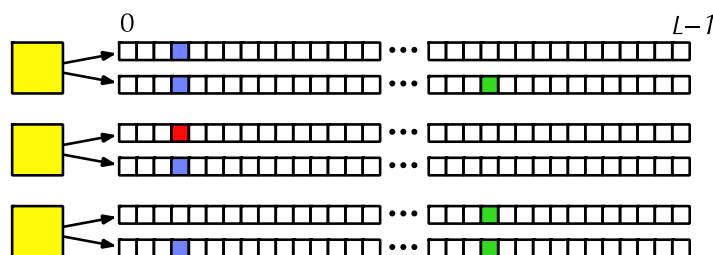
You can, of course, set up your simulation to begin with whatever mutational state you want, by explicitly adding mutations to particular genomes, or by loading mutational state from a VCF or MS file. Even more commonly, a simulation will begin with a “burn-in” period that establishes an equilibrium level of genetic diversity through mutation–selection–migration balance before the more interesting part of the simulation begins. It is important to understand that such genetic diversity is always built on top of empty chromosomes in SLiM, however. In general, if two mutations are segregating at a given base position, there are effectively *three* alleles in the population at that base position: the first mutation, the second mutation, and what you could think of as the “wild-type allele” represented by the *absence* of either of those mutations. In script, you could force a mutation to exist at every base position in every genome, so that there are no empty positions in any of the genomes in your simulation; if you do so, however, you will find that your simulation then runs quite slowly, since SLiM is having to track and manage all of those mutations, so such a strategy is usually undesirable.

Learning to let go of the idea of chromosomes filled with genetic information at every position (as is the case biologically, when regarding a DNA molecule), and thinking instead in terms of mutations layered on top of the empty “wild-type” state, is an essential conceptual leap to make in using SLiM. To move from one conceptual model to the other, imagine the “wild-type” nucleotide sequence for your study organism: some specified genome sequence of A, C, G, and T. Whatever that sequence might be, it is represented in SLiM by the absence of any genetic information at all: empty chromosomes. SLiM tracks only mutations on top of that sequence: SNPs, in the nucleotide-based paradigm. But any individual that does not possess a SNP at a given location instead possesses the “wild-type” nucleotide, conceptually – represented in SLiM by an empty base position. This conceptual model is preserved even when modeling nucleotides explicitly in SLiM, although that is beyond the scope of our discussion here (see section 1.8).

We will not dwell upon it here, but it is worth noting that genomes in SLiM can sometimes be so-called *null genomes*. Null genomes are placeholders, representing the *absence* of a genome, because every individual in SLiM always has two genomes but sometimes you do not, in fact, want two genomes. If you are modeling haploids, for example, every individual would have one non-null genome, containing the individual’s haploid genetic information; the second genome could just be kept empty, or it could be a null genome signifying that the second genome does not exist.

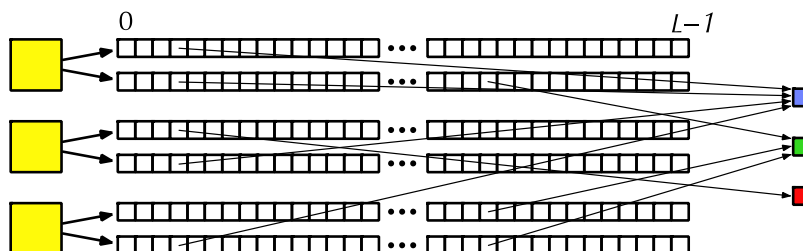
### 1.5.2 Mutations and substitutions

With the foundation of individuals and genomes laid by the previous section, let's now explore the idea of mutations in SLiM in more detail. In SLiM, a mutation is an instance of class `Mutation` in Eidos (see section 24.10). A mutation has various properties – its base position and its selection coefficient, most importantly. (In nucleotide-based models, each mutation also has an associated nucleotide.) In SLiM, the multiplicative fitness effect of a mutation with selection coefficient  $s$  is  $1+s$  for a homozygote; in a heterozygote it is  $1+hs$ , where  $h$  is the dominance coefficient (kept by the mutation type; see section 1.5.4). Let's update our conceptual schematic to include some mutations:



This simulation has three mutations, represented here with blue, red, and green. At position 3, the first individual is homozygous for the blue mutation, the second individual is heterozygous for red and heterozygous for blue, and the third individual is heterozygous for blue (the other allele in that individual being the empty “wild-type allele” as discussed above).

An important concept to absorb is that the genomes that contain the blue mutation do not just contain their own particular copies of blue-mutation-type information; they actually contain *references* to the very same shared blue-mutation object. A more accurate conceptual diagram might therefore look like this:

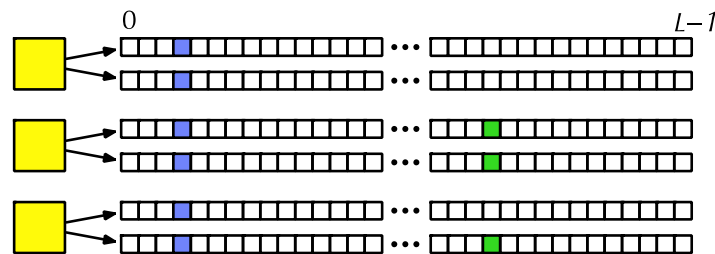


Here the particular base positions in the genomes have references to the three shared mutation objects in the model. Since this diagram is quite difficult to interpret visually, we will stick with showing mutations as residing inside genomes; but you should always keep in mind that mutations are really shared objects. A new mutation object is created either (1) when a random mutation event occurs in SLiM (as governed by the overall mutation rate set for the simulation), or (2) when requested by the simulation script with the `addNewMutation()` or `addNewDrawnMutation()` methods of `Genome` (see section 24.4.2). In both cases, *these events always create a new mutation object*, even if a mutation with exactly the same properties – position, selection coefficient, etc. – already exists in SLiM. In our conceptual diagram above, the blue and red mutations might be identical in every detail; they are nevertheless considered to be different mutations by SLiM, and will be tracked separately and never merged into a single identity. You can therefore think of SLiM as tracking mutational lineages, following a sort of identity by descent; the red and blue mutations might represent the very same SNP, but they arose due to separate mutational events, and thus they seeded mutational lineages that SLiM keeps separate.

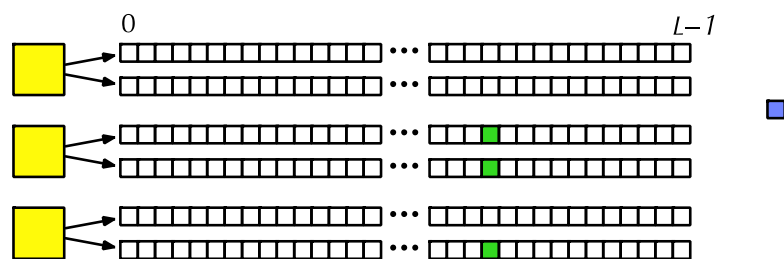


This distinction becomes particularly important when you ask whether two genomes contain “the same mutation” – if you ask, for example, whether a given individual is heterozygous or homozygous for a given mutation. The second individual in the diagram is heterozygous for blue and heterozygous for red; even if the blue and red mutations are identical, the individual is not considered by SLiM to be homozygous. The combined multiplicative fitness effect of being red/blue,  $(1+hs) \times (1+hs)$ , is therefore different from the fitness effect of being red/red or blue/blue,  $(1+s)$ , and – depending upon the selection coefficient and dominance coefficient involved – this difference could be quite large. Often this can be ignored; if mutations are neutral then this clearly doesn’t matter, and if mutational effects are drawn from a distribution of fitness effects then in general no two new mutations will be the same anyway so the point is moot. However, if some mutations in a simulation use a fixed, non-neutral fitness effect then this might become important; this could be important in a model of a soft sweep by recurrent *de novo* mutations, for example. In such cases, you may wish to ensure that all references to identical mutations are transmuted into references to the same mutation object, enforcing a single mutational lineage. When new mutations are being introduced in script, rather than by SLiM, you can ensure that an existing mutation object is used by using the `addMutations()` method of `Genome` (see section 24.4.2), which adds already-existing mutations to a genome instead of creating new mutation objects (and thereby new mutational lineages). For new mutations introduced automatically by SLiM, the technique shown in section 18.14 may prove useful.

Another key concept involving mutations is that by default, mutations are removed from the simulation when they become fixed. Suppose that, after mate choice and biparental mating, the next generation of our conceptual diagram looked like this:



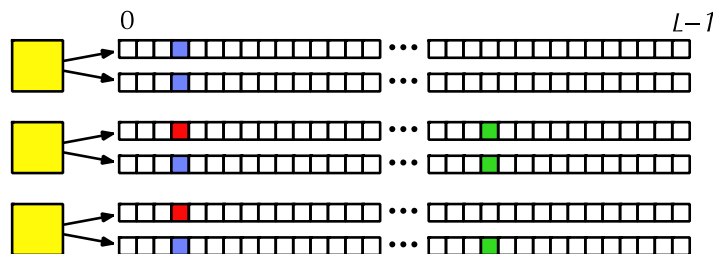
This genomic state will be visible to the simulation script only momentarily, if at all, because in each tick, at a particular point in the tick cycle, it will be detected by SLiM and replaced by this state instead:



The fixed mutation has been removed from the simulation and stored as a “substitution” object (represented here by the blue square to the right). This substitution object will be kept by SLiM forever, to remember the fixed mutation. The substitution object is available to the script; but the mutation is no longer contained by the genomes of each individual, and it no longer influences SLiM’s fitness computations. This is usually a good idea, because it allows SLiM to run much faster than it otherwise would; without removal of fixed mutations, simulations would slowly bog down under the weight of more and more accumulated fixed mutations. It is also usually safe, since a

mutation that is possessed by every individual will usually have an effect on *absolute* fitness but not on *relative* fitness – since its multiplicative fitness effect is the same in every individual, it can be neglected. However, simulations that involve epistasis, or that model additive mutational effects upon quantitative traits, or that otherwise depend upon mutations in ways that go beyond their direct effect on relative fitness, may wish to disable this automatic conversion for the mutations involved in such effects; this can be done easily in SLiM using the `convertToSubstitution` property (see section 24.11.1).

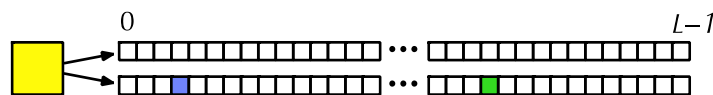
The above ideas, about the distinct identity of each mutational lineage and the way that fixation is defined in SLiM, combine in a way that is worth mentioning since it might be unexpected. Suppose that there are two mutations, blue and red, segregating at base position 3, as above, and suppose further that these mutations are identical in every detail but arose separately, as described above. Finally, suppose that the population reaches this state:



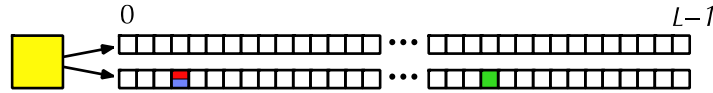
It might be natural to suppose that the mutations at position 3 would be considered to have fixed, and would be removed, as above, given that they are all identical and merely represent independent mutational lineages of what you might think to be the same mutation. As above, however, this is not how SLiM thinks! Since the blue and red mutations are different mutation objects, neither is considered to have reached fixation; fixation in SLiM occurs when a specific mutation reaches a population-wide frequency of `1.0`, without any consideration for other identical mutations segregating in the population. In a scenario like this, such as a soft-sweep model, if you want fixation of independent mutational lineages to be detected you will need to either merge the independent lineages yourself in script, as described above (and see section 18.14), or simply detect fixation yourself directly. You could detect fixation by checking that every genome in the model contains an appropriate mutation (either red or blue, in this case), or by summing the counts of all of the appropriate mutations in the population to confirm that the total count is equal to  $2N$  (where  $N$  is the population size and the constant factor of 2 accounts for diploidy). (You might be inclined to sum the mutation frequencies instead and compare to `1.0`, but this strategy is vulnerable to floating-point roundoff error, so it is not advisable.) This is all simpler than it sounds; the soft-sweep recipes in section 9.5 provide some examples.

### 1.5.3 Mutation stacking

There is one more key concept about mutations in SLiM to consider, and it is this: by default, a given base position in a given genome may actually contain more than one mutation – indeed, it may contain an arbitrarily large number of mutations. This is referred to as “mutation stacking”; the multiple mutations at a single base position in a given genome are referred to as being “stacked”. For example, imagine that this individual exists in a SLiM simulation:

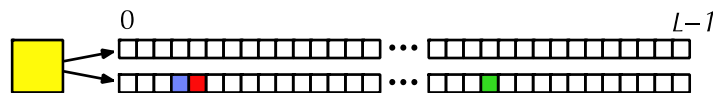


And then imagine that a new mutation, which we will show as red, occurs at the same position, in the same genome, where the blue mutation already exists. By default, here is what will happen:



The red mutation has stacked on top of the blue mutation; both mutations now exist at that position in that genome. This does not fit terribly well into the concept of “genotype” – SLiM does not really think in terms of genotypes. You could perhaps say that the genotype of this individual is something like “wild/red-blue”, if you wished, where “red-blue” is the allele created by the stacking of a red and a blue mutation together at the same locus. SLiM, however, just thinks in terms of the mutations actually possessed by each genome, so in SLiM’s terms, rather than talking about genotype, we would simply say that the individual is wild-type (i.e., empty) at the given position in one genome and has a red mutation and a blue mutation at that position in the other genome. We could also say that the individual is heterozygous for red and heterozygous for blue; that is true, but is a bit ambiguous since the same would be said of a red/blue heterozygote that had a red allele in one genome and a blue allele in the other; unless the red and blue mutations are epistatic, however, that distinction is probably unimportant. Note that further mutations at the same position could occur as well, and would stack on top of those already present, making all this even more complex. Probably the simplest thing is to learn to think in the same terms in which SLiM thinks: which mutations are present in which genomes.

This behavior of SLiM is perhaps quirky, but usually harmless. It is not common for mutations to end up stacked in practice, since normal mutation–selection balance in finite populations usually clears out genetic diversity quickly enough that it is unusual for a new mutation to occur right on top of another mutation that is still segregating in the population. Furthermore, when mutations do occasionally stack, it is usually not important to the dynamics of the model; in most cases the resulting behavior is identical, for practical purposes, to if the new mutation had occurred at an immediately adjacent base position instead, which would result in the two mutations being extremely tightly physically linked rather than actually being stacked:



In principle these mutations could be separated by recombination, whereas the stacked mutations cannot be, but in practice that is unlikely enough that it will probably not make a difference to the model’s dynamics. It is therefore important to understand that SLiM works in this way, but in most cases models do not need to concern themselves with stacking.

However, there are cases where it does present problems. If you want to simulate actual nucleotides (see section 1.8 and chapter 18), for example, then each base position must be unambiguously either A, C, G, or T; it makes no biological sense for an A and a G to be “stacked” at a single position. Similarly, if you want to make a quantitative-genetics model with particular discrete quantitative effects for the alleles at each QTL (see section 13.3), such as a  $-1/0/+1$  allelic system, you would not want mutation stacking to occur since stacking of more than one mutation at a given QTL would violate your design. The default mutation stacking behavior may therefore be modified.

The stacking behavior of mutations is governed by their *mutation type*, a concept we haven’t yet discussed. All mutations in SLiM belong to a mutation type, represented by the Eidos class `MutationType` (section 24.11). Mutation types are important primarily because they dictate the distribution of fitness effects from which mutations are drawn; for example, all of the mutations of a given mutation type might be neutral, or they might be deleterious and drawn from a gamma distribution. When a new mutation is generated by SLiM, the selection coefficient for the mutation is drawn from the distribution specified by the relevant mutation type, as will be discussed in detail



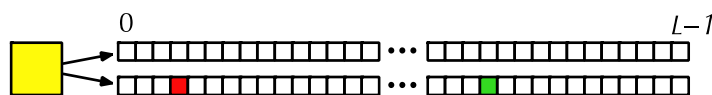
in section 1.5.4. Simulations may define as many mutation types as desired, but most simulations contain just one or a few mutation types.

Besides defining the distribution of fitness effects from which mutations are drawn, mutation types define a few other behaviors for mutations too. For example, the `convertToSubstitution` property mentioned in section 1.5.2, which determines whether a mutation will be removed when it fixes, is actually a property of `MutationType`, not of each individual mutation, since it makes sense for this behavior to be uniform across each class of mutations. The dominance coefficient of mutations is also a property of the mutation type, not of individual mutations, for the same reason.

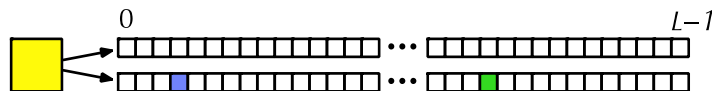
For our discussion of mutation stacking here, however, mutation types are important because stacking behavior is controlled by the `mutationStackGroup` and `mutationStackPolicy` properties of `MutationType` (see section 24.11.1); all of the mutations of a given mutation type follow the same stacking policy. In fact, more than one mutation type can be joined together into a single “mutation stacking group” using `mutationStackGroup`, and then all of the mutations in that stacking group follow the same policy. This is a power-user feature that we will mostly gloss over for the rest of the discussion here, however; we will just assume that each mutation type constitutes a separate stacking group (which is the default behavior).

Given this design, in the example above the red mutation needs to be of the same mutation type (or in the same stacking group) as the blue mutation before stacking can be prevented. If they are not of the same mutation type (or more precisely, not in the same stacking group), then they will stack, regardless of what stacking policy has been set for each mutation type. The idea is that you generally don’t want different kinds of mutations to interfere with each other in a model. If a QTL mutation exists at a given position, for example, then you might want a new QTL mutation at the same locus to replace the old one, rather than stacking – but if a neutral mutation occurred at the same locus, you would probably not want that neutral mutation to replace the existing QTL mutation. Even more clearly, if you used a particular mutation type to represent epigenetic modifications along the genome, such as methylation – which is certainly possible in SLiM – then you certainly would want those epigenetic modifications to be able to stack with mutations of other types. Whatever you might think of that argument, you can always modify it by placing all of your mutation types into a single stacking group.

So let’s now assume that the red and blue mutations are in the same stacking group – of the same mutation type, most trivially. Now the stacking policy can be modified. The default policy is referred to as type “s” (for “stack”), and yields the stacked result we saw above. Instead, you can set the policy to type “l” (for “last”), which produces this result after the red mutation occurs:

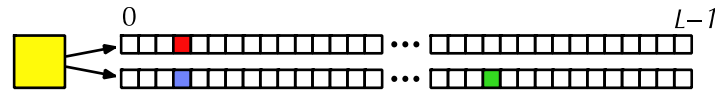


The red mutation has replaced the mutation in this genome, because this stacking policy dictates that the *last* mutation at a given position should be kept. This is the common alternative to the “s” policy, but you can also set a policy of “f” (for “first”) which produces this result:

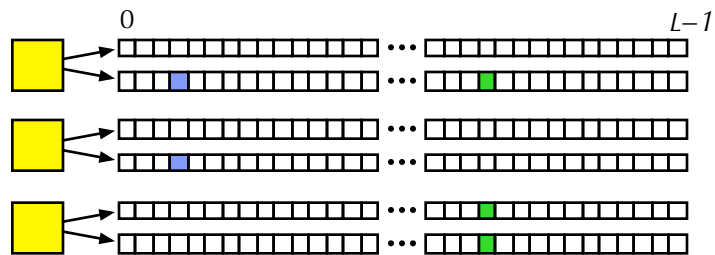


Here the red mutation has simply been thrown away, because the blue mutation was there first. The post-mutation state is therefore identical to the pre-mutation state (note that this means the effective mutation rate will be lower than the requested mutation rate, since some mutations will be suppressed). This policy is less commonly used, since its biological motivation is unclear, but it is provided as an option in SLiM just in case it is called for (see, e.g., section 9.5.1).

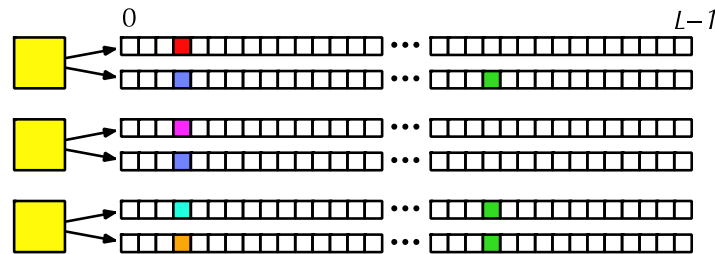
Note that all three stacking policies – "s", "l", and "f" – depend only upon the existing mutation(s) *at the specific base position in the specific genome* where a new mutation is occurring. The state at other positions, or in other genomes, is completely irrelevant. Changing the mutation stacking policy is not a way to prevent more than one allele from existing in a population at a given locus; it is only a way to prevent more than one mutation from existing *in one genome* at a given locus. Suppose, in our ongoing example, that the new red mutation had occurred in the *other* genome of the individual instead; regardless of the stacking policy, the result would then be:



That base position in that genome was empty; the stacking policy is therefore irrelevant, since there is no pre-existing mutation to stack on top of, so the red mutation is always added. More generally, suppose we have a population of three individuals, with a few mutations:



Different new mutations could occur at each of the empty sites at base position 3, regardless of the stacking policy. So whether the stacking policy is "s", "l", or "f", new mutations could easily lead to this state:



If the stacking policy were "f", further new mutations at position 3 beyond this could not be introduced; now that every genome has a mutation at position 3, those pre-existing mutations would prevent the new mutations from being added. Under stacking policy "l", new mutations would still be possible at position 3 even in this state; the new mutations would just replace the old mutations. Under policy "s", the new mutations would stack, the default behavior.

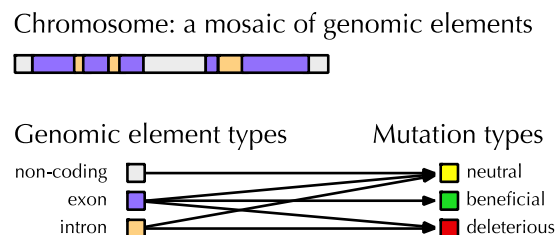
Sometimes it is desirable to allow for the possibility of back-mutation in a model. This can be accomplished in several ways in SLiM. One way is to use a mutational distribution of fitness effects that provides only a limited set of possible values, and use type "l" stacking so that new mutations replace existing mutations; new mutations are then automatically sometimes back-mutations. (This would occur automatically in a nucleotide-based model; see section 1.8). Another possibility, if you only need back-mutation to the “wild type” empty-genome state, is to remove existing mutations yourself, in script; if done with the appropriate probability, this could simulate back-mutation to the wild type. For nucleotide-based models, see section 18.15 for further discussion of such approaches; there is a lot of complexity here!

Finally, note that the stacking policy is applied to new mutations introduced in script, as well as to new mutations added by SLiM as a result of the overall mutation rate. New mutations that you add will be allowed to stack unless you change the stacking policy; and conversely, if you change the stacking policy then new mutations that you add might result in the removal of pre-existing mutations, or might not be added at all, in accordance with the policy you have chosen. However, when the stacking policy is changed in mid-run it is not retroactively enforced upon existing mutations, and when a saved population is loaded the current stacking policy is not enforced upon the mutations loaded.

The issue of mutation stacking is a bit complicated and confusing; if the discussion above has left you with more questions than answers, rest assured that this is an advanced topic that generally does not impact simple models at all. Play around with SLiM for a while, and then when you revisit this section it will probably make much more sense.

#### 1.5.4 Genomic elements, genomic element types, mutation types, and the chromosome

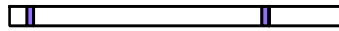
SLiM allows you to model complex chromosomes with genomic regions that have different mutational effects. For example, exons would be expected to sustain beneficial, deleterious, and neutral mutations, whereas introns might sustain only deleterious and neutral mutations, and non-coding regions might only allow neutral mutations. You set up this structure in SLiM using a hierarchical configuration: the chromosome (class `Chromosome`, section 24.2) contains genomic elements (class `GenomicElement`, section 24.5), which are each of a given genomic element type (class `GenomicElementType`, section 24.6), and each genomic element type draws new mutations from a weighted set of mutation types (class `MutationType`, section 24.11), each of which specifies a distribution of fitness effects for that type of mutation. In an exon/intron/non-coding model, for example, each type of genomic region – “exon”, “intron”, and “non-coding” – would be a genomic element type, and the chromosome would be a mosaic of genomic elements using these three genomic element types. Each of these genomic element types would draw its mutations from a different set of mutation types – perhaps “beneficial”, “deleterious”, and “neutral”, in this case. In practice, that might look something like this:



Here the chromosome contains two genes, each of which is composed of an alternation of exons and introns, and non-coding regions are interspersed around the genes. Those regions along the chromosome are defined by genomic elements, which reference the three genomic element types defined here (non-coding, exon, and intron). Those three genomic element types each utilize some subset of the mutation types that have been defined here (neutral, beneficial, and deleterious). Although this diagram simply shows arrows from genomic element types to mutation types, there are in fact weights associated with the mutation types of a given genomic element type; in this case, you might specify that 90% of mutations within introns are neutral and 10% are deleterious, by giving those weights to SLiM when the intron genomic element type is configured. In this way, new mutations that are automatically generated by SLiM will have appropriate fitness effects depending upon the genomic region in which they occur.

Note that this example is entirely arbitrary; you can define whatever mutation types, genomic element types, and genomic elements you wish. You could make your chromosome entirely neutral, or you could model the specific mutational effects of each region in an empirical genome

map, right down to the full level of detail known for your study organism. There is no practical limit to the number of mutation types and genomic element types you may define. In the opposite direction, you can also make a much simpler model than the one above. Not all of the chromosome needs to be assigned to a genomic element; much of the chromosome can simply be empty. For example, you could make a model of epistatic interactions between two loci with a chromosome like this:



Here most of the chromosome is empty, containing no genomic elements. No mutations will be generated in these regions, since mutation generation in SLiM is governed by genomic elements. Only the purple loci will be active; we have one genomic element type and one mutation type in this model, and quite possibly only two mutations (one at each locus). A simple genetic structure like this will often run much faster in SLiM than simulating a whole chromosome; if you are not interested in what is going on in some regions of the chromosome, just don't simulate them. The only limitation in setting up your chromosome structure in SLiM is that genomic elements must be non-overlapping.

Note also that in general, this chromosomal structure only influences the way that SLiM automatically generates new mutations; it has no effect upon mutations added in script. When offspring are generated, the overall mutation rate is used to draw the number of mutations that have occurred in a given gamete. The position of each mutation is then drawn, and SLiM determines which genomic element the mutation falls within. Given that, SLiM can find the genomic element type, and it then chooses a mutation type from the weighted set of mutation types used by that genomic element type. Finally, knowing the mutation type for the mutation, it draws a selection coefficient from the distribution of fitness effects for that mutation type. That yields a new mutation object, which is placed in the gamete. That is the *only time* that SLiM uses the genomic elements and genomic element types you have defined; none of the other machinery inside SLiM's core cares about those constructs at all. This means – following the earlier example – that your script is free to explicitly add a beneficial mutation within an intron; the fact that introns are defined as not experiencing beneficial mutations is irrelevant. You may consult the chromosomal structure in your script and use it in whatever way you wish, but doing so would be quite unusual. By and large, the behavior of SLiM simulations depends upon the mutations contained by the genomes of individuals, without reference to the chromosomal structure apart from the moment when a new mutation is created.

Mutation types, however, do continue to be used in a minimal fashion, as we saw in section 1.5.3; each mutation knows its mutation type, and some properties of mutations, such as their dominance coefficient, their stacking behavior, and their fixation behavior, are specified by their mutation type.

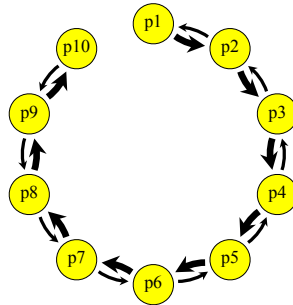
### 1.5.5 Subpopulations and migration

The previous section discussed hierarchy levels from the individual to the genome to the mutation, and the dependence of mutations upon mutation types, genomic element types, genomic elements, and ultimately the chromosome itself. However, there are also higher hierarchy levels: individuals live within subpopulations, and all of the subpopulations together exist within the whole modeled population.

Subpopulations are represented by the class `Subpopulation` in Eidos. A subpopulation is a set of individuals, and is characterized primarily by the fact that random mating occurs (weighted by individual fitness) within each subpopulation. In other words, subpopulations primarily influence reproductive isolation; each subpopulation is internally panmictic (again, weighted by individual

fitness), but externally isolated. Migration rates can be configured between subpopulations in order to allow gene flow, but by default the migration rate among subpopulations is zero.

For example, one might have a population structure like this:



Here we have ten subpopulations linked into a “stepping-stone” model that might represent a river system; subpopulation p1 is upstream, and relatively high levels of migration produce gene flow in the downstream direction, while much lower levels of migration exist in the upstream direction (as shown by the relative widths of the arrows). This is just an example; you can have as many subpopulations as you wish, linked by any pattern of migration. The effect of the population structure on SLiM will manifest in the pattern of reproductive isolation among individuals.

The details of this conceptual model can be important. Offspring are always generated by parents within a single subpopulation; parental individuals do not move between subpopulations for the purposes of mating. Instead, migration occurs at the juvenile stage in SLiM; generated offspring are placed into particular subpopulations in accordance with the specified migration rates. SLiM allows you to define spatial simulations involving one-, two-, or three-dimensional landscapes for each subpopulation; again, even in such spatial models, panmixis (weighted by individual fitness) is the default, although effects like spatial competition and spatial mate choice preference may be added in script. The subpopulation is the fundamental unit of reproductive isolation in SLiM. (All of this refers to WF models only, to which we are still limiting our discussion; in nonWF models, individuals can mate with other individuals regardless of subpopulation structure, and can migrate at any time; see section 1.6.)

An important conceptual point is that when setting the size of a subpopulation in SLiM, this should be thought of as a request for a future change, not as a present-time change. If a subpopulation contains 800 individuals and the script requests that it be 900 or 700 instead, that change does not happen immediately (which individuals would be removed? how exactly would the new individuals be created – with what genetic state?); instead, the subpopulation size change is a request that will take effect in the next generation. That is, when offspring are generated, 900 or 700 offspring will be generated from the current subpopulation of 800, and the requested subpopulation size will thus take effect in the child generation. This is always the case in SLiM (in WF models); there is actually no straightforward way to create new individuals or kill existing individuals within a single generation (although the fitness of an individual can be set to zero, which has much the same effect as killing it, all else being equal).

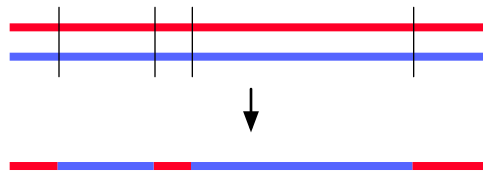
Another key concept is that zero-size subpopulations do not exist in SLiM (in WF models). If a subpopulation is set to size zero, it will (in the child generation) cease to exist entirely. For this reason, a new subpopulation cannot be created with a size of zero, since that has no meaning in SLiM. Instead, you should create the new subpopulation with a non-zero size at the moment that its size grows above zero. This can be inconvenient in metapopulation models that involve dynamic local extinction and re-colonization; such models are better written as nonWF models, which follow very different rules (see section 1.6, and section 16.5 for an example of a nonWF extinction/recolonization model).



### 1.5.6 Recombination, gene conversion, and biased gene conversion

In sexual models, recombination is an important process because it breaks down linkage between different areas of the chromosome over time. SLiM has two fairly different ways of modeling recombination: a simple “crossover breakpoints” model, and a much more realistic “double-stranded break (DSB)” model. We will discuss both in this section.

The “**crossover breakpoints**” model has existed since SLiM 1.x, and continues to be supported. It is a model of crossover only; gene conversion is not modeled. The only parameter to this model is the recombination rate per base position per gamete (although it can be supplied as a recombination map, varying the rate along the chromosome, if desired). When generating a gametic genome during (non-clonal) offspring generation, SLiM begins with one of the two parental genomes as the *copy strand*, and draws the number and the positions of *crossover breakpoints* based upon the recombination rate (or rate map). At each crossover breakpoint, the copy strand switches to the opposite strand. The gametic genome is generated by copying from the current copy strand, switching copy strands at each crossover breakpoint, until the end of the chromosome is reached. New mutations are incorporated as a part of this process, changing the state at a given base position from the state in the copy strand, but mutations will not be discussed further in this section. The process of gamete generation in the “crossover breakpoints” model therefore looks like this:



The red and blue lines at top are the two parental strands, and the thin black lines are crossover breakpoints that occur between one base position and the next, switching the copy strand at that point. The resulting gametic genome is shown below the parental strands; it is a patchwork of red and blue sections as a result of the crossovers.

The “crossover breakpoints” model is chosen by simply setting a recombination rate (or rate map) with `initializeRecombinationRate()`. It is the recombination model used by most SLiM models; it is conceptually simpler and faster to execute, and is sufficiently realistic for most purposes. However, it does not support gene conversion.

The “**double-stranded break (DSB)**” model is an extension of the basic gene conversion facility that existed in SLiM 1.x and onward. In SLiM 3.3 it was cleaned up and redesigned, and backward compatibility with the old scheme was deliberately *not* preserved (see below), so it can be considered to be essentially new. This model is designed with two main goals: (1) to allow gene conversion at both crossover and non-crossover points to be modeled reasonably accurately, not at the level of a single generation (where these processes are quite complex and biological realism would be difficult to achieve), but at the level of producing realistic long-term average behavior, and (2) to provide intrinsic support for biased gene conversion with SLiM 3.3’s new nucleotide-based models (about which see section 1.8). The mechanics of the “DSB” model described here are a simplified version of the biological recombination mechanisms described in Duret & Galtier (2009), attempting to achieve the two aforementioned goals while preserving conceptual simplicity and ease of implementation.

As in the “crossover breakpoints” model, in the “DSB” model when SLiM is generating a gametic genome it copies from a “copy strand”, one of the two homologous parental strands. Similarly, too, breakpoints are chosen based upon the rate or rate map; in a hypothetical example that we will follow through the recombination process, the breakpoints might look like this:



In the “DSB” model, these breakpoints represent double-stranded breaks, or DSBs, in a more biologically accurate manner than the breakpoints of the “crossover breakpoint” model, as we will see below. We will therefore refer to them interchangeably as either “breakpoints” or “DSBs” hereafter.

Next, at each DSB, *gene conversion* is configured. Gene conversion is a phenomenon in which a short stretch of DNA surrounding a DSB – a *gene conversion tract* – is taken from the opposite strand. (A technical aside: if a recombination rate of exactly 0.5 occurs between two bases, to produce the effect of independent loci/chromosomes as in section 6.1.4, then no gene conversion tract will be constructed around a DSB at that position; it will be considered a simple crossover point, as in the “crossover breakpoints” model.) A gene conversion tract is constructed around each DSB by drawing an extent for the tract to the left of the DSB, and independently drawing an extent for the tract to the right of the DSB. These two draws are each taken from a geometric distribution with mean  $\lambda/2$ ; the expected length of the gene conversion tract as a whole, then, is  $\lambda$  (where  $\lambda$  is a model parameter), but the length distribution is not geometric (it is negative binomial, in fact), and the tract may not be centered on the DSB. The gene conversion tracts around the DSBs shown above might look like this:



Note that two of the tracts here overlap. This is considered to be a bad draw, and is rejected. As a result, new DSBs are drawn (the *number* of DSBs is retained, and the *tract lengths* around the DSBs are retained, to be precise; only the DSB *locations* are re-drawn). This process repeats until a good draw is achieved. (In practice, for biologically realistic parameters, bad draws are very unlikely to occur since DSBs are not common and  $\lambda$  is small, but we illustrate the procedure here for completeness.) In our hypothetical example, the redrawn configuration looks like this:



The gene conversion tracts are now non-overlapping, and so the recombination process continues. In the “DSB” model, each DSB can result in a switch in the copy strand (a crossover event), or can be repaired with no switch in the copy strand (a non-crossover event). For each DSB this determination is made independently, according to a given probability of non-crossover (which is a model parameter). In our hypothetical example, the DSBs might look like this after the determination has been made, with crossovers represented by a solid black line and non-crossovers by a dashed black line:



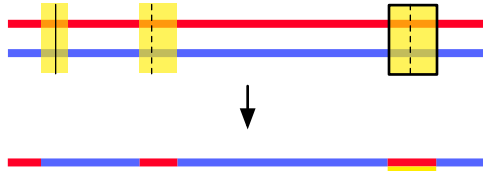
The leftmost DSB has been designated as a crossover; the other two are non-crossovers. The next step is that each gene conversion tract is designated as either *simple* or *complex*. This determination is made, again, independently for each tract according to the probability of a simple gene conversion tract (which is a model parameter). This determination is independent of whether the DSB is a crossover or non-crossover.

If, for our ongoing example, the first two gene conversion tracts are designated as simple tracts, while the third is designated as a complex tract, then the recombination configuration now looks like this, showing complex tracts outlined with a box:



In a simple gene conversion tract, the copy strand simply switches at the beginning of the tract, and then switches back again at the end of the tract. A simple gene conversion tract thus behaves very similarly to two closely-spaced crossover breakpoints (and indeed, it is modeled as such under the hood). In a complex gene conversion tract, on the other hand, the red and the blue parental strands actually end up paired with each other within the gene conversion tract (*why* this occurs gets into the topic of Holliday junctions, etc., which we will not describe here). Where the two parental strands are identical, this is fine; one could think of either one as being the “copy strand” but the distinction doesn’t matter since they are identical. Where they are *not* identical, however, this pairing means that *heteroduplex mismatches* exist, at which mismatched bases such as A-C or G-T are paired. Each of these heteroduplex mismatches is repaired independently, in SLiM, making complex gene conversion tracts a sort of hybrid between the red and blue parental strands on a mismatch-by-mismatch basis. Conceptually, however, and for the purposes of tree-sequence recording (see section 1.7), we will think of their ancestry as being just like simple gene conversion tracts, entailing a switch in the copy strand at the beginning and end of the tract. Any mismatches that are repaired toward the non-copy strand will be represented as new mutations rather than as a change in ancestry, as we will see below.

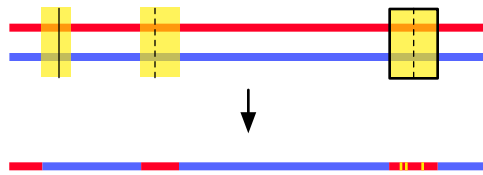
Continuing our example, then, we can now actually construct the resulting gametic genome, which looks like this:



Note that the first DSB, which was designated as a crossover, switched the copy strand at its beginning and did not switch it back (or, to be precise, it switched back at the end of the gene conversion tract and then immediately switched over again to model the crossover). The other two DSBs, designated as non-crossovers, switched the copy strand at both their beginning and their end. Each stretch of the gametic genome has a designated ancestry, either red or blue, including the complex gene conversion tract (which is recorded as red, entailing a temporary switch in the copy strand, just as with the preceding simple gene conversion tract). The complex gene conversion tract, however, might have heteroduplex mismatches that will need to be resolved, case by case, and it is therefore shown with a yellow underlying stripe; that stripe represents the non-copy strand that is paired with the copy strand within the tract.

The final step is repair of heteroduplex mismatches within any complex gene conversion tracts (such as the region marked with the yellow stripe above). This process looks at each mismatch between the red and blue strands within each complex gene conversion tract and chooses probabilistically whether to adopt the state of the red strand or the blue strand. Normally this choice is done with equal probabilities for either option, but the choice can be biased if desired (see below). After all of the heteroduplex mismatches have been found and repaired, the final genome might look like this:





Sites where a heteroduplex mismatch was repaired toward the non-copy (blue) strand are shown as tiny fragments of yellow, representing single SNPs that were present on the non-copy strand and were added as new mutations (but sharing the same SLiM mutation objects as the original SNPs) in the final gametic genome. Their ancestry is *not* tracked as blue in tree-sequence recording, which is why they are shown here as yellow rather than blue.

Note that if a mutation is present in the copy strand and not in the non-copy strand, the repair toward the non-copy strand could really involve the *removal* of a copy-strand mutation. Also, mutations that were newly created during the copying process would exist in the copy strand, and heteroduplex mismatch repair could immediately remove such mutations by repairing them back toward the non-copy strand. The yellow fragments shown in the diagram above could therefore represent (1) addition of a non-copy-strand mutation not present in the copy strand, (2) removal of a pre-existing copy-strand mutation not present in the non-copy strand, or (3) removal of a newly added mutation in the copy strand. In all three cases, however, the yellow fragments would represent the fixing of a heteroduplex mismatch toward the non-copy strand. Finally, note that heteroduplex mismatch repair is performed position by position, not mutation by mutation; if stacked mutations exist at a position (see section 1.5.3), mismatch repair will involve choosing either the copy-strand stack or the non-copy-strand stack, never breaking up a stack.

The final concept to discuss is *biased* gene conversion. When repairing heteroduplex mismatches in complex gene conversion tracts, the repair can be unbiased – repairing an A-C heteroduplex mismatch to either A-T or G-C with equal probability, for example – or biased toward a tendency to fix the mismatch in a particular way. The typical bias in biological systems is called GC-biased gene conversion, or gBGC, and entails a preference for G and C alleles in the repair process. This means that an A-C mismatch would be more likely to be repaired to G-C than to A-T. The degree of bias is represented by a gene conversion bias coefficient (a model parameter). If it is 0.0, there is no bias. A value of 1.0 indicates that a G or C allele will always be chosen in preference to an A or T allele; a value of -1.0, the opposite. (Other possible mismatches are unaffected by the gene conversion bias coefficient, and are always resolved with equal probability.) Since biased gene conversion is a nucleotide-based phenomenon, it may only be enabled in a nucleotide-based model (see section 1.8).

The “DSB” model, then, has five parameters: (1) the recombination rate (or rate map), as before; (2) the mean length  $\lambda$  of gene conversion tracts (drawn with two independent geometric draws with mean  $\lambda/2$ ); (3) the fraction of DSBs that result in non-crossovers (the remainder becoming crossovers); (4) the fraction of gene conversion tracts that are “simple” (the remainder becoming “complex”); and (5) the gene conversion bias coefficient, influencing resolution of heteroduplex mismatches in complex gene conversion tracts, which is 0.0 by default. The “DSB” model is selected by calling `initializeRecombinationRate()` to set the recombination rate (or rate map), and then calling `initializeGeneConversion()` with the remaining parameters. Specifically, the call to `initializeGeneConversion()` switches SLiM to using the “DSB” model, rather than the simpler “crossover breakpoints” model.

It is worth briefly mentioning that prior to SLiM 3.3 the `initializeGeneConversion()` function did not have a parameter for the fraction of gene conversion tracts that are simple, nor a parameter for the gene conversion bias coefficient. Old code that uses `initializeGeneConversion()` will no longer run (because of the missing parameters), and will need to be fixed manually. This lack of

backward compatibility is deliberate, since the underlying conceptual model has changed; existing users of gene conversion in SLiM need to think about how they wish to adapt their scripts. The three key conceptual differences, prior to SLiM 3.3, were that (1) all gene conversion tracts were “simple”, (2) DSBs that resulted in a crossover event did not have any associated gene conversion tract, and (3) gene conversion tracts started at the DSB point, rather than extending to both sides of the DSB point.

#### 1.5.7 Community, Species, ticks, cycles, and generations

SLiM 2 supported only non-overlapping generations, because it supported only simulations based upon the Wright–Fisher model, which assumes non-overlapping generations. In SLiM 3, support was added for non-Wright–Fisher models that allowed overlapping generations (see section 1.6), and the conceptual model became more complex. In SLiM 4, support for modeling more than one species was added (see section 1.9), and so the conceptual model became more complex again – especially since those multiple species could run on different timescales (since mosquitoes, for example, reproduce much more frequently than humans). This section will attempt to clear up some of the resulting conceptual confusion.

One area where confusion arises is: what object has the property/method I’m looking for? In SLiM 2 and SLiM 3, the top of the population hierarchy was the `SLiMSim` object, named `sim`. If you wanted anything having to do with the simulation as a whole, you would look in that class – to get the segregating mutations, or the active subpopulations, or the generation counter, for example, or to add a new subpopulation, or create a new log file, or generate output with a standard method like `outputFull()`. In SLiM 4, in order to support multiple species, `SLiMSim` was broken into two pieces. Now we have `Species`, which represents one species in the model, and `Community`, which is the new top-level object that contains all the species being simulated. In both single-species and multispecies models, there is now always one `Community` object, named `community`. If you’re writing a single-species model (which will be the topic for the first 500-plus pages of this manual), there is also one `Species` object, named `sim`. The name `sim` was kept to preserve some degree of backward compatibility across this transition; most, but not all, of the functionality that used to be in `SLiMSim` is now in `Species`, so most legacy code will run with few or no modifications.

So now, when you’re looking for a property or method having to do with “the simulation” in a single-species model, where do you look – `Community` or `Species`? To answer this, think about the responsibilities, at a conceptual level, of each of these classes. If you’re looking for something that has to do with the particular species being simulated, then look in `Species`. To cover the examples given previously, the segregating mutations belong to the species; a different species would have different segregating mutations. Similarly for the active subpopulations of a species, or adding a new subpopulation to a species; and similarly for generating output with `outputFull()`, since that produces an output file about the individuals and mutations in the species. On the other hand, if you’re looking for something that has to do with the entire simulation, or with *all* of the species (if you were to have more than one), then look in `Community`. From the previous examples, this would include creating a new log file (since these can log information about the whole simulation, not just one species). You would also look in `Community` for properties and methods that provide information spanning multiple species, in a multispecies model; to get a vector of *all* the subpopulations in the model, not just those belonging to one species, look in `Community`.

You might have noticed that one thing in that previous list of examples was not clarified above: “the generation counter”. That used to live in `SLiMSim`’s `generation` property, but that property no longer exists in SLiM 4. Where do you look now for information about time in the simulation?

The answer is that there are three answers, because there are three different ways of thinking about time now. This can be confusing. The three concepts of time are *ticks*, *cycles*, and *generations*.

*Ticks* are the unit of time kept by *Community*, and the tick counter is available from `community.tick`. You can think of ticks as being an objective measure of time in the simulation, like the ticking of a clock. Each tick represents one time unit. You get to decide what length of time a tick represents in your model; it could be a year or a season or a day or anything else. You don't even need to tell SLiM what it is; SLiM doesn't care. SLiM executes a set of standard behaviors in every tick: things like reproduction, fitness assessment, and mortality. This set of standard behaviors, and the exact order in which they occur, is referred to as the “tick cycle”. The tick cycle for WF and nonWF models is not the same, but they both have a tick cycle. You can see a brief overview of the WF tick cycle in section 1.3; complete reference documentation for the WF and nonWF tick cycles is in chapters 22 and 23, respectively. At the end of the tick cycle, the tick counter kept in `community.tick` is incremented by one, and then the next tick cycle begins. So the tick counter begins at 1, and increments at the end of every tick cycle.

*Cycles* are the unit of time kept by *Species*, and the cycle counter is available from `sim.cycle` in single-species models (or a similar syntax in multispecies models). This counter simply indicates the number of tick cycles that the species has completed. When the tick cycle finishes, the cycle counter for the species is incremented by one – *if* the species was active during that tick. In single-species models, the species is active in every tick, always, and so the cycle counter and the tick counter always have exactly the same value; there is a conceptual difference between them, but there is no practical difference. In multispecies models, a given species might be active in some ticks and inactive in others (because, e.g., mosquitoes reproduce more often than humans), and so the cycle counter for a given species might count upward more slowly than the tick counter.

*Generations* are a biological unit of time having to do with things like the lifespan of an organism, the mean age at first reproduction, and so forth. There is no one “right” definition of a “generation”, for organisms with overlapping generations and age structure, and it can be hard to pin down what one even means by the term in some cases (if, for example, males and females have very different life histories). SLiM makes no attempt to measure generations for you; if you want to do that, you would write some Eidos script that would calculate it based upon the definition of “generation” that you want to use. However – confusingly – SLiM used to use the term “generation” to refer to its unit of time, before SLiM 4. Back in those days, there were no “ticks” or “cycles”; both of those time units were added in SLiM 4, and what they represent was called a “generation” back then. What has happened in SLiM 4 is quite parallel, then: SLiMSim split into *Community* and *Species*, and SLiMSim itself no longer exists; and `sim.generation` split into `community.tick` and `sim.cycle`, and `sim.generation` no longer exists.

Which should you use, then? If you want the overall “objective” measure of time that is the same for any species in the model, and that represents something like seasons or days or years, then use `community.tick`. If you want a view of time from the perspective of the species, representing how many opportunities that species has had to reproduce since the simulation started, then use `sim.cycle`. And if you want the biological concept of “generations”, whatever exactly you mean by that, then use that term (and calculate it yourself).

Which will this manual use, in its recipes and discussion? Unfortunately, it will use all three, since all three concepts are important. Times for events and script blocks in your code are always specified in ticks – everything runs off of the objective clock, not subjective species times. But when you're talking about what a species does, and the behaviors it exhibits, talking in terms of cycles often makes the most sense. And we will use the term “generation” too, especially in the context of WF models (with non-overlapping generations), since the concepts of the “parental generation” versus the “offspring generation” are important, and the number of times that the parents have died and the children have taken over the world is conventionally called the number of generations.

On the bright side, as long as you are writing single-species WF models all three are the same; with a single species and non-overlapping generations, ticks, cycles, and generations are all the same length of time (although the conceptual differences between the terms remain). And as long as you are writing single-species nonWF models, ticks and cycles remain the same; they just don't necessarily match up with biological generations any more (if your nonWF model has non-overlapping generations and age structure). But when you get into writing multispecies models, all three terms refer to different units of time, and really, different ways of thinking about the passage of time; so then you need to be careful. We will try to retain a clear conceptual separation between these terms throughout the manual, but sometimes it can be difficult.

### 1.5.8 Other concepts

That completes our overview of the foundational concepts underlying SLiM. There is a lot more to SLiM that will be covered in the following chapters, but if you understand these fundamental ideas the rest should be fairly straightforward.

Other sections of this manual also provide important conceptual information. Section 1.3 contains a brief introduction to some other key SLiM concepts that did not merit an in-depth discussion here, such as fitness, the tick cycle, continuous space, interactions, and tags; it would be good to read now, if you haven't already. The following sections of chapter 1 provide conceptual introductions to various types of advanced models that can be written in SLiM: non-Wright-Fisher or "nonWF" models (section 1.6), models using tree-sequence recording (section 1.7), nucleotide-based models (section 1.8), and multispecies models (section 1.9); these are recommended reading so that you understand the possibilities that exist in SLiM, even if you do not intend to use these features now.

Part II of this manual, the SLiM Reference, also has some conceptual sections; chapter 22 contains detailed information on the stages of the tick cycle in SLiM for WF models (and chapter 23, for nonWF models), including details on how mate choice, migration, offspring generation, fitness calculation, and other stages are implemented, and chapter 25 describes how to modify those default cycle stages through the use of several different types of scripted callbacks, which provides much of the power and flexibility of SLiM. Reading those chapters might be deferred until you have become more familiar with SLiM, however, as they are more technical; if you are a beginning SLiM user, it is recommended that you finish reading this chapter and then proceed with installing SLiM and begin experimenting with the recipes presented in the chapters that follow.

The learning curve for SLiM can be steep, and this manual is unfortunately rather long. Perhaps the best conceptual introduction to SLiM is provided by the free **SLiM Workshop**, which can be attended in-person or completed online. See section 1.10 for more information.

## 1.6 Wright-Fisher (WF) versus non-Wright-Fisher (nonWF) models

SLiM 1.x and 2.x supported only one type of model, Wright-Fisher models (usually referred to as WF models). In SLiM 3.0, support was added for a second type of model, non-Wright-Fisher models (nonWF models). The choice between these types of models is made with an optional initialization call, `initializeSLiMModelType()`; if no call to that function is made, the WF model type is used by default, providing complete backward compatibility with SLiM 2.x.

Use of the nonWF model type is an advanced topic, recommended only for users experienced with SLiM. Most of this manual will be about the default WF model type. Only three areas of this manual will discuss nonWF models in any detail: section 1.6 (e.g., this section, which summarizes the differences between WF and nonWF models), chapter 16 (which introduces nonWF models in more detail and presents nonWF model recipes), and the SLiM Reference (Part II of this manual).

The differences between WF and nonWF models are pervasive, but may be regarded as falling into a few major categories:

- **Age structure.** In WF models, generations are discrete and non-overlapping. Each “tick” of SLiM’s tick counter is associated with the creation of a new offspring generation and the demise of the previous parental generation. There is thus no concept of the age of an individual, since all individuals live for a single tick. In nonWF models, generations may instead be overlapping. Each “tick” of SLiM’s tick counter is associated with the opportunity for creation of new offspring and the opportunity for mortality among existing individuals. SLiM keeps track of the age of individuals, which may live for many ticks. This makes it simple to construct models of overlapping generations with any type of age structure and any age-related behaviors desired.
- **Offspring generation.** In WF models, offspring are generated automatically by SLiM each tick. This process may be modified by various callbacks, but the process itself – how many offspring to generate, from which parental individuals, into which subpopulations – is managed by SLiM in such a way as to “fill out” each subpopulation with a fresh batch of offspring while satisfying the constraints imposed by parameters such as subpopulation size, sex ratio, cloning rate, and selfing rate. In nonWF models, offspring are instead generated in response to a request from the model script, made in a `reproduction()` callback, and the script itself is in charge of managing the process. In nonWF models, SLiM no longer attempts to enforce any particular subpopulation size, sex ratio, cloning rate, or selfing rate; typically, these are instead emergent properties of the individual-based dynamics of the model. This approach is somewhat more complex, but allows the genetics and state of each individual to influence the way that individual reproduces – its expected litter size, its reproductive behavior (cloning, selfing, biparental mating), the sex of its offspring, and so forth.
- **Population regulation.** In WF models, population regulation (keeping population size within bounds) is managed automatically by SLiM, which keeps each subpopulation at the initial size it is given, or at whatever new size is set by a `setSubpopulationSize()` call. Subpopulation size is therefore a parameter of the model, in effect. In nonWF models, population regulation is instead an emergent property, a side effect of how many offspring are created versus how many individuals die due to selection in each tick. This makes it much more natural to construct models with realistic population dynamics such as density-dependence, fitness-dependence, resource limitation (i.e., carrying capacity), and so forth. In a nonWF model, the parameters of the model that result in population regulation would more likely be things like carrying capacity, the strength of density-dependent fitness, or the size of a limited resource pool.
- **Fitness.** In WF models, all offspring survive to maturity, and fitness specifies the probability that an individual will be chosen as a parent in the next generation. Higher-fitness individuals thus have a larger expected litter size, but fitness in WF models is *relative* fitness because the population size is held to whatever size is set by the model as described above. In nonWF models, fitness influences survival instead (mating success and fecundity can be modified also, but in script, not through SLiM’s automatic fitness evaluation mechanism). Fitness differences are thus expressed through the likelihood that a given individual will survive to maturity. This allows a more realistic modeling of the



variance in reproductive output, as well as simplifying model dynamics that are influenced by the survival of individuals, such as competition. In nonWF models, fitness is *absolute* fitness, which is more realistic but can be more challenging to model since it forces you to think explicitly about population regulation in concrete, individual-based, mechanistic terms. (See section 8.4 for further discussion of fitness in SLiM.)

- **Migration.** In WF models, migration is governed by model parameters set on each subpopulation, specifying what rate of migration SLiM should enforce with each other subpopulation. Migration is simulated in these models as the movement of an offspring individual, immediately after it is generated in the parental subpopulation; thus, only juvenile migration can be modeled. In nonWF models, migration is instead implemented in script, by explicitly moving individuals from subpopulation to subpopulation. This can be done at any time; migration of adults as well as juveniles can be modeled, or multiple migrations over the course of an individual's life. This design also makes it much simpler to implement migration that depends upon the circumstances of each individual: habitat choice, condition-dependent migration, genetic variation in dispersal, sex differences in migration behavior, and so forth.
- **Subpopulation splits.** In WF models, the splitting of a subpopulation is modeled as a new subpopulation being founded by a wave of such migrant offspring. In nonWF models, subpopulation splits are modeled as the migration of a set of individuals (of any age) to form a new subpopulation. Particularly in models with small population sizes, this can produce more realistic splits, particularly when migration of parental individuals, rather than juveniles, is desired to found new populations.

The general trend across all of the above points is that nonWF models are more individual-based, more script-controlled, and potentially more biologically realistic – but also more complex in some respects, because the SLiM core is managing fewer details of the model's dynamics automatically. In particular, all nonWF models must implement at least one `reproduction()` callback in order to generate new offspring. Each type of model has its appropriate uses; nonWF models are not “better”, although they are more flexible in some respects. WF models may be simpler to design, and may run somewhat faster; and of course staying within the WF conceptual framework may make it easier to compare simulation results with theoretical expectations from analytical models that are based in the Wright–Fisher paradigm.

As mentioned above, most of the remainder of this manual will discuss WF models, since they are SLiM's original mode of operation and remain the default model type. Whenever a given section does not explicitly state otherwise, it should be assumed that the focus is on WF models. All of the recipes in Part I of the manual are WF recipes up until chapter 16, which specifically presents nonWF models.



The reference section of this manual, Part II, will provide reference information covering both WF and nonWF models. A color-coding convention will thus be used in Part II: items which apply only to WF models will be highlighted in green, and items which apply only to nonWF models (or primarily so) will be highlighted in blue. For example:

```
– (void)a_WF_only_method(...)
```

and:

```
– (void)a_nonWF_only_method(...)
```

Again: nonWF models are an advanced topic. As a beginning SLiM user, it's good to know that nonWF models exist, but don't worry if all of the above information is not clear. You can forget about nonWF models for now, and focus on familiarizing yourself with the default, WF models.

## 1.7 Tree-sequence recording

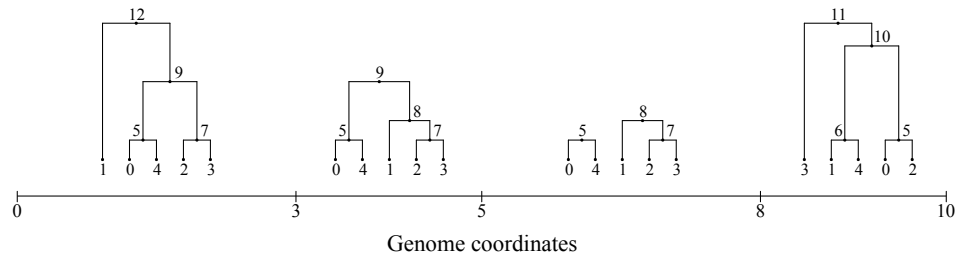
SLiM 3 introduces a major feature called tree-sequence recording. This is essentially a method of tracking the true local ancestry of every chromosome position in every individual as a SLiM model runs. Such ancestry information can be saved out to files called `.trees` files, and can be loaded in to SLiM from `.trees` files as long as they are in the correct SLiM-compliant format. These `.trees` files can also be loaded into Python, where their ancestry information can be browsed, analyzed, and even modified using the packages `tskit` (for tree sequences) and `msprime` (for coalescent simulation). When moving data between SLiM, `tskit`, and `msprime`, the `pyslim` Python package is also essential, since it knows how to translate some types of SLiM-specific information in `.trees` files from SLiM into a form the other packages can work with, and vice versa. Models using tree-sequence recording will sometimes be referred to as “tree-seq” models.

Use of tree-sequence recording is an advanced topic, recommended only for users experienced with SLiM. Most of this manual will be about models that do not use tree-sequence recording. Only three areas of this manual will discuss tree-seq models in detail: section 1.7 (e.g., this section, which summarizes the concept of tree-sequence recording), chapter 17 (which presents tree-seq model recipes), and the SLiM Reference (Part II of this manual).

Tree-sequence recording does not record the full pedigree of a model. Instead, it records only the specific ancestry information needed to reconstruct the mutational and recombinational history of each extant individual's genomes. Over time, some information that originally needed to be recorded in the tree sequence may become unnecessary to keep; perhaps a whole branch of the evolutionary tree went extinct and so the information recorded about it is no longer relevant, for example. Through the process of *simplification*, which is performed periodically upon the recorded tree sequence, all information that is not relevant to any extant genomes gets pruned away, which keeps the memory requirements of tree-sequence recording manageable.

This recorded information is referred to as a “tree sequence” because it is literally a sequence of trees. Conceptually, each position along the chromosome has its own ancestry tree, which is the result of recombination at that position; as one walks along the chromosome, one encounters a sequence of such ancestry trees, from which the pattern of inheritance can be traced from every extant genome back to the most recent common ancestor for the population at that chromosome position (and thence all the way back to the original ancestor). The ancestry tree at a given position may have multiple roots, if there is no common ancestor for all of the individuals in the population at that position; forward simulations begin with every individual unrelated to every other, and common ancestry is only built over time through the process of coalescence. The ancestry trees at adjacent chromosome positions are generally highly correlated, and indeed are often identical (since those adjacent positions may never have been split by recombination, or

traces of such recombination might not have survived in any living individual). Tree-sequence recording accounts for this correlation, tracking each distinct ancestry tree along successive chromosome intervals rather than at every position. This is one reason why the tree sequence is sometimes called a *succinct tree sequence*; it is much more compact than the full set of trees for every position, because it omits the redundant information shared between positions with identical ancestry. This multiplicity of trees is not easy to depict, but here is a sketch of the concept:



The intervals between the ticks on the x axis are intervals on the chromosome that have distinct ancestry trees (this example chromosome is only ten base positions long, and has four intervals with distinct trees). Each interval's ancestry tree reflects its particular pattern of inheritance along the chromosome; however, the trees at adjacent sites tend to be highly correlated, with redundant information that is represented concisely by the tree sequence data structure. Within each tree, the leaf nodes at the bottom (labeled 0 through 4) might be extant genomes with no descendants, whereas the internal nodes might be ancestral genomes that are no longer extant; however, with overlapping generations things might be less clear-cut, since ancestors might still be extant. As mentioned above, the tree for a given interval might have multiple roots, if coalescence is not yet complete; that situation is found in the ancestry tree for the third chromosome interval pictured above.

This is just a brief summary of tree-sequence recording, and may or may not be comprehensible; for a full overview of tree-sequence recording, including details of how simplification works, please refer to the definitive paper on the topic, Kelleher et al. (2018).

Tree-sequence recording enables some very powerful techniques, such as:

- **Overlaying neutral mutations.** You can run a model without any neutral mutations (which is generally much faster), and then overlay neutral mutations afterwards using `msprime`. This gains tremendous efficiency from the fact that neutral mutations only need to be overlaid on branches of the ancestry tree that lead to extant individuals; neutral mutations on all branches that went extinct before the end of the simulation do not need to be considered at all. For models that contain many neutral mutations, this can result in a speedup of an order of magnitude or more.
- **Analyzing ancestry directly.** You can sometimes avoid simulating neutral mutations altogether, when it is actually the pattern of ancestry you are interested in. Often the pattern of neutral mutations from a forward simulation is used to infer things about a population's history – selection, bottlenecks, immigration, and so forth. Using neutral mutations for this purpose is a blunt instrument, however, since they are sparse and occur stochastically; inference from them is difficult, time-consuming, and inexact. Instead, it is often possible to draw such inferences from the recorded tree sequence itself, which in a sense embodies every possible mutational history that the population could have had given its actual history of inheritance and recombination. The inferential power can therefore be much higher. This means that inferences from the tree sequence can often be

exact, and can also often be computed much more quickly and easily than from neutral mutations.

- **Detecting coalescence during forward simulation.** Often you want to “burn in” a model until it has reached an equilibrium state, which can often be defined as occurring some time after full coalescence across the whole chromosome has occurred. The time at which this happens, however, is often hard to know. A heuristic of “ $10N$ ” (running for a number of generations equal to ten times the population size) is often used, but even for simple models it can be an underestimate. For models with a variable population size, multiple subpopulations, or non-neutral dynamics of any kind, the proper burn-in duration is often just a guess, and it is often necessary to make a large overestimation “just to be safe”. With tree-sequence recording enabled, SLiM can tell you whether your model has coalesced fully or not; it has all the information needed. You can then use that information to decide when to end the burn-in period of the model. (The moment of coalescence itself is special in some respects, and so it might not itself be a good equilibrium point at which to end burn-in, but it provides a baseline.)
- **Moving between coalescent and forward simulation methods.** Tree-sequence recording and the `.trees` file format build a sort of bridge between coalescent and forward simulation methods, allowing both methods to be used in a single simulation. For example, a neutral “burn-in” period for a model could be simulated using the coalescent in `msprime` without mutations, and the results saved to a `.trees` file in the proper SLiM-compatible format using `pyslim`. That `.trees` file could then be loaded into SLiM as the starting state for forward simulation; perhaps the evolutionary dynamics become non-neutral at that point in time, due to a change in the environment, and so now one wishes to forward simulate those non-neutral dynamics. Since the coalescent is so fast, this can result in much quicker burn-in compared to simulating the burn-in with SLiM. You could then overlay neutral mutations after forward simulation is completed (see above); this is a common strategy for moving between coalescent and forward simulation. Moving between methods in this manner allows the strengths of each method to be leveraged, while avoiding their weaknesses; “recapitation”, described below, is another such strategy. As mentioned above, the `pyslim` package is essential to this sort of interoperability, as we will see in chapter 17. Several ways of moving `.trees` data between SLiM and `msprime` using `pyslim` will be shown in chapter 17. The `pyslim` documentation has even more examples of such interoperability; for example, the vignette at [https://tskit.dev/pyslim/docs/stable/vignette\\_coalescent\\_diversity.html](https://tskit.dev/pyslim/docs/stable/vignette_coalescent_diversity.html) shows how to generate a neutral burn-in with `msprime`, including overlaid neutral mutations, and import that whole burn-in into SLiM (as opposed to overlaying the neutral mutations after completing forward simulation, which is the more typical workflow); this can be useful if the forward simulation actually needs to use or modify mutations that come from the burn-in period.
- **“Recapitating” to construct the ancestral history of a simulation.** With this technique, you could run a model forward with no neutral burn-in period at all, from a set of empty genomes, and then reconstruct the ancestry of the initial genomes using the coalescent after the forward simulation has completed. This technique, called *recapitation*, is much more efficient even than generating a burn-in state using the coalescent directly, because only the ancestry trees present at the end of forward simulation need to be coalesced back; any part of any genome that was present in the initial state of the model but was

later lost does not need to be coalesced, so most of the work of burn-in can be avoided. As before, neutral mutations can be overlaid at the end of the model run, after recapitation has constructed the ancestral history, rather than having to be simulated. This technique can allow very large models to be simulated relatively efficiently, since most of the work of burn-in can be eliminated. The use of this technique is shown in section 17.10.

Tree-sequence recording has a large impact on the performance of models, in terms of both runtime and memory usage. It is therefore disabled by default, and needs to be enabled with a call to an initialization function, `initializeTreeSeq()`. When it is enabled, you can control the frequency of simplification (more frequent simplification means a lower memory footprint but a longer runtime), and you can turn on checking for coalescence if your model needs to know when full coalescence has been attained (for a small additional performance penalty on top of each simplification).

Again, tree-seq models are an advanced topic; it is good to know that tree-sequence recording is an option, but beginning SLiM users should postpone trying to use it until they have become fairly familiar with the basics of SLiM. However, if you're finding that performance is an issue, and you're bogged down simulating huge numbers of neutral mutations or running endless burn-in periods, tree-sequence recording may be able to help. Similarly, if you want to detect coalescence, or obtain a record of the ancestry at every chromosome position, tree-sequence recording can provide a solution. Tree-sequence recording also provides an output format for saving simulation state that can be much more compact than either VCF or SLiM's native output file format (even though it includes information about ancestry that those file formats do not), and that can be analyzed and manipulated in Python with `pyslim`; those advantages can also be a reason to use tree-sequence recording.

Section 1.10 of this manual contains some useful links for those interested in learning more.

## 1.8 Nucleotide-based models

Beginning with SLiM 3.3, SLiM now supports models in which an explicit nucleotide sequence along the chromosome is tracked. Such models, henceforth called *nucleotide-based models*, are fundamentally quite similar to other SLiM models, but there are a few key differences, particularly in the areas of model initialization and mutation generation. Those differences will be summarized here; beyond this, chapter 18 contains several recipes that provide examples of nucleotide-based models. This section assumes that the reader is already familiar with non-nucleotide-based SLiM models; nucleotide-based models are an advanced feature.

Nucleotide-based models may be WF or nonWF models (see section 1.6), and they may use tree-sequence recording or not (see section 1.7); the decision as to whether a model is nucleotide-based or not is orthogonal to those other choices. To make a nucleotide-based model, the first step is to notify SLiM by passing `nucleotideBased=T` to the `initializeSLiMOptions()` function in the model's `initialize()` callback. Flipping this switch has various consequences and provides various new features:

- **Keeping an ancestral nucleotide sequence.** Nucleotide-based models must supply an ancestral nucleotide sequence in their `initialize()` callback, by calling `initializeAncestralNucleotides()`. The ancestral nucleotide sequence specifies the nucleotide that SLiM will consider to be present at a given position of a genome unless a nucleotide-based mutation is present at that position (overriding the ancestral sequence with a derived nucleotide). This ancestral sequence could also be thought of as the “wild-

type” sequence, in the sense introduced in section 1.5.1; it represents the nucleotide sequence for the empty genomes that SLiM begins with by default. Keeping an ancestral nucleotide sequence allows SLiM to avoid keeping a separate nucleotide sequence for each genome; instead, it can track the mutational differences from a common genetic background, just as it does for non-nucleotide-based models. SLiM provides a convenience function, `randomNucleotides()`, for generating a random ancestral nucleotide sequence; alternatively, an ancestral nucleotide sequence may be loaded from a file (such as a FASTA file).

- Defining nucleotide-based mutations.** The `initializeMutationType()` function now has a nucleotide-based twin, `initializeMutationTypeNuc()`, that is used to declare a nucleotide-based mutation type. (Nucleotide-based models may still also use non-nucleotide-based mutation types, as discussed below.) Mutations that belong to a nucleotide-based mutation type will keep track of the nucleotide they represent, available through the `nucleotide` property on `Mutation` (as a string value, "A" / "C" / "G" / "T") or through the `nucleotideValue` property on `Mutation` (as an integer value, 0 / 1 / 2 / 3 respectively). (The same properties exist on `Substitution` as well, carrying over from the original mutation.) In all other respects nucleotide-based mutations behave identically to non-nucleotide-based mutations; for example, the standard SLiM fitness machinery using the mutation type's distribution of fitness effects and dominance coefficient still applies. When a nucleotide-based mutation fixes it is converted to a `Substitution` object unless `convertToSubstitution` is F, as usual (see section 1.5.2); in addition, the ancestral nucleotide sequence will be updated to use the new fixed nucleotide instead of the nucleotide that was originally present. Nucleotide-based mutations are automatically set up by SLiM to comprise a single “stacking group” with a “last” stacking policy (see section 1.5.3), ensuring that a new nucleotide-based mutation at a site that already contains a nucleotide-based mutation will correctly replace the old mutation (while stacking, by default, with any non-nucleotide-based mutations also present at the site).
- Defining sequence-based mutation rates for genomic element types.** The mutation rate along the chromosome is no longer determined by an overall mutation rate or a mutation rate map; indeed, in nucleotide-based models it is not legal to call `initializeMutationRate()` at all. Instead, the `initializeGenomicElementType()` function takes a new parameter named `mutationMatrix`. This supplies a matrix of mutation rates that depend upon the original nucleotide present at a given site. For example, the `mutationMatrix` might state that if the existing nucleotide is an A, then it will mutate to a G with a probability of  $2.23 \times 10^{-8}$  (per base position per gamete, as usual in SLiM); it could provide different rates for an A-to-T mutation, etc. The `mutationMatrix` is thus a  $4 \times 4$  matrix containing 16 values, providing rates for mutation from any nucleotide to any nucleotide (the values along the diagonal, representing the rate of mutation from a given nucleotide to the *same* nucleotide, must be 0.0 by convention). Alternatively, `mutationMatrix` may be a  $4 \times 64$  matrix providing rates for mutation from any given trinucleotide sequence (with the potentially mutating base being the center of the trinucleotide) to a new central nucleotide (again, the entries representing the mutation of a nucleotide to itself must be 0.0). This allows the mutation rate to depend upon the nucleotide sequence immediately surrounding the point of mutation, so that (for example) an elevated mutation rate at CpG sites can be implemented. SLiM defines helper functions to set up these matrices for common cases such as the Jukes–Cantor mutational model. Examples of this are provided in chapter 18.



- **A new nucleotide-based mutational model.** The previous point is one aspect of what is really a new mutational model in SLiM, used in all nucleotide-based models. In this new mutational model, the overall mutation rate per gamete is not constant because the mutation rate at each position is sequence-dependent; a genome with many CpG sites (for example) might mutate more rapidly than a genome with lower CpG content. For reasons of conceptual clarity, this new mutational model allows only nucleotide-based mutations to be auto-generated by SLiM (the ratio between mutation types specified to `initializeGenomicElementType()` would not be easily interpretable if one mutation type were nucleotide-based and another weren't). Note that non-nucleotide-based mutations can still be added in script, and can coexist with nucleotide-based mutations in a nucleotide-based model; they just can't be auto-generated by SLiM, to keep the new mutational model simple.
- **A new mutational hotspot map.** In non-nucleotide-based models the mutation rate along the chromosome can be varied, providing mutational “hot spots” and “cold spots”; in nucleotide-based models, since `initializeMutationRate()` is not used, such mutation-rate variation along the chromosome can instead be supplied with a new function, `initializeHotspotMap()`. The values of the hotspot map combine multiplicatively with the base rates specified by the `mutationMatrix` properties of genomic element types, making mutation overall more or less likely within a region while preserving the relative frequencies of particular mutations. If no hotspot map is supplied, a default value of 1.0 is assumed across the chromosome, meaning that the rates supplied by the `mutationMatrix` properties of genomic element types are used unmodified as the absolute mutation rates for the model.
- **Access to genome nucleotide sequences for any purpose in script.** The nucleotide sequence of any genome is available in script, in whole or within a given range, with the `nucleotides()` method of `Genome`. The nucleotide sequence for a genome is based upon the ancestral sequence, with the nucleotides for any nucleotide-based mutations in the genome overlaid on top. The ancestral sequence itself is also available in script, from the `ancestralNucleotides()` method of the `Chromosome` object (`sim.chromosome`). For example, if one wanted to implement a fitness effect that depended upon the codon sequence within an exon (making stop codons lethal and synonymous mutations neutral, for example), one could get the nucleotide sequence for the exon from a given individual's genomes and, based upon examination of their nucleotide sequences in script (perhaps in comparison to the ancestral sequence), assign a fitness effect to the individual using the `fitnessScaling` property or with a `fitnessEffect()` callback. Many other uses for these nucleotide sequences are also possible, such as outputting nucleotide sequences from a sample of genomes/individuals at key points in a run, assessing GC richness in different populations, and computing sequence-based  $F_{ST}$  values.
- **Intrinsic support for biased gene conversion.** With these new nucleotide-based models, SLiM now provides intrinsic support for biased gene conversion (see section 1.5.6). This can be configured, if biased gene conversion is desired, simply by providing a `bias` parameter to the `initializeGeneConversion()` function that gives the probability that, at mismatched sites within a gene conversion tract that need to be repaired, the mismatch will be repaired by taking the G or C nucleotide in preference to the A or T nucleotide. For



example, if there is an A/G mismatched pairing, where one nucleotide comes from one parental strand and the mismatched nucleotide comes from the homologous parental strand that invaded along the gene conversion tract, *either* the G will be repaired to T or the A will be repaired to C, and the bias indicates any preference between those alternatives in the mismatch repair machinery. The default bias of **0.0** indicates that the choice is random; a positive value up to **1.0** provides GC-biased gene conversion (gBGC), whereas a negative value down to **-1.0** provides AT-biased gene conversion instead (which is probably non-biological, in general, but is provided for completeness).

The reference section of this manual, Part II, will provide reference information covering both non-nucleotide-based models and nucleotide-based models. Since non-nucleotide-based models are the default (and were the only option available until SLiM 3.3), material specifically related to nucleotide-based models will be highlighted in purple in this manual. For example:

```
- (void)aNucleotideModelOnlyMethod(...)
```

Again: nucleotide-based models are an advanced topic. As a beginning SLiM user, it's good to know that one can create a nucleotide-based model, but don't worry if all of the above information is not clear; this section assumed a fairly detailed familiarity with the default, non-nucleotide-based models, as a basis for discussing how nucleotide-based models are different. If you are just starting out with SLiM, you should probably focus on non-nucleotide-based models first.

## 1.9 Multispecies models

Beginning with SLiM 4, SLiM now supports models in which more than one species is simulated simultaneously. Such models, henceforth called *multispecies models*, are based upon all of the same fundamental concepts as other SLiM models, but utilize an extended SLiM script syntax to allow the various species in the model to be independently declared, configured, and managed. These extensions and related concepts will be summarized here; beyond this, chapter 19 contains recipes that provide examples of multispecies models. This section assumes that the reader is already familiar with single-species SLiM models; multispecies models are an advanced feature. Most of this manual will avoid discussion of multispecies models, for clarity.

Each species in a multispecies model may be independently configured in most respects. One might be sexual while another is hermaphroditic; one might use tree-sequence recording while another does not; one might be nucleotide-based while another is not; one might live in continuous space while another lives in discrete subpopulations connected by migration. The various species may also have completely different genetics, different scripted behavior (including callbacks), and different schedules for reproduction, migration, and mortality.

In one respect, however, all of the species in a multispecies model must match: they must all use the same model type. In other words, they must all be based upon the WF model, *or* all be based upon the nonWF model (see section 1.6); the two model types cannot be mixed in a multispecies model. This is because the model type determines the phases of the tick cycle; in a multispecies model, the species typically share the tick cycle and execute in an interleaved fashion, so the structure of their tick cycles must match.

To make a multispecies model, the first step is to switch to *explicit species declarations*. In a single-species SLiM model, the declaration of the single species is implicit; that species is always named `sim`, and the global symbol `sim` representing that species is declared for you automatically by SLiM. Multispecies models instead declare each species in the model explicitly, giving each species a unique name; perhaps a model might have `fox` and `mouse` species, for example. In that case, global symbols named `fox` and `mouse` will exist, representing the two declared species, and

the symbol `sim` will no longer exist. Chapter 19 will show how to write such explicit species declarations to create a multispecies model.

Once a model has explicitly declared one or more species, SLiM's behaviors change in various ways, providing new features related to multispecies simulation:

- **Species-specific initialization.** Each species must have its own `initialize()` callback, in which that species is configured independently of the other species in the model. No configuration may be shared between species; each species must declare its own genomic elements types, mutation types, interaction types, and so forth. (The identifiers for these objects must be unique across the whole model, however; for example, if one species declares a mutation type `m1`, no other species may use that `m1`, and no other species may declare its own `m1`). For configuration at the whole-simulation level (the community level, in SLiM's terminology), such as setup of model parameter values, a non-species-specific `initialize()` callback can be written, as we will see in chapter 19.
- **Species-specific callbacks.** In general, every callback in a multispecies model must be declared with a *species specifier* that tells SLiM which species that callback applies to. Every callback applies to exactly one species, in general; callbacks may not be shared by multiple species. If more than one species needs to have the same callback-driven behavior, that behavior can be implemented in a shared user-defined function, called by the callbacks of each species; this can be a bit roundabout, but is worth it for the conceptual clarity provided by the rule that every callback belongs to exactly one species, always. (The hedge phrase “in general” was used because `interaction()` callbacks are always non-species-specific, configuring behavior at the community level.)
- **Independent tick schedules.** The objective measure of time (years, months, days) in SLiM models is the tick, and a counter with the number of ticks that have elapsed – the number of times that the tick cycle has executed – is kept by the `Community` object, available in `community.tick`. Each species also keeps a counter of the number of times its tick cycle has executed, available in single-species models in `sim.cycle`. In single-species models these are the same; the species executes its tick cycle in every tick. In multispecies models, on the other hand, ticks pass at the same rate for all species, but each species may choose to be *active* or *inactive* in any given tick, independent of the other species. When a species is inactive in a given tick, it does not execute the tick cycle, and its cycle counter does not increment. As a result, the cycle counter for different species may differ from the tick counter, and from each other; species live on independent tick schedules. Furthermore, if non-overlapping generations are modeled both of these measures of time may be different from the biological concept of the number of generations that have passed. See section 1.5.7 for further discussion of these concepts.
- **Interleaved tick cycle execution.** In ticks for which more than one species is active (see above), the tick cycles of the species are *interleaved*. In essence, this means that each stage of the tick cycle executes for *all* species, one by one, before SLiM proceeds to the next stage. For example, each species would be given an opportunity to reproduce, one after the other, and then after all reproduction is done, each species would undergo mortality, one after the other. The precise way in which this works will be discussed in chapter 19; for now, the point is that the tick cycle events for species can occur (almost)

simultaneously, if you wish, through this interleaving; if you do not want that to happen, simply configure the model so that in any given tick only one species is active.

Not every model that involves multiple species ought to be a SLiM multispecies model. There are two cases where it is particularly likely that you will *not* want to use a multispecies model. One case is when you are modeling the process of speciation itself. Multispecies models are required to declare every species in the model explicitly; when modeling the process of speciation, you might not know whether multiple species will emerge, how many there will be, what their characteristics will be, etc. In such cases, it is often easier to model a single species, and allow divergence within that species to emerge, reproductive isolation to develop, etc., without explicitly declaring each species. The other case is when species are so similar that they are allowed to hybridize (even if they are partially reproductively isolated). As mentioned above, every species in a multispecies model is completely distinct, using different genomic element types, mutation types, etc.; hybridization between individuals of different species is thus strictly not allowed. Again, in such cases it will typically be better to model a single SLiM species, with individuals tagged according to which species they belong; this will allow hybridization to occur. If you are unsure whether to approach a given problem as a single-species or multispecies model, it might be useful to try both approaches; one will likely feel much simpler and more natural.

Again: multispecies models are an advanced topic. As a beginning SLiM user, it's good to know that one can create a multispecies model, but don't worry if all of the above information is not clear; this section assumed a fairly detailed familiarity with SLiM's default single-species models, as a basis for discussing how multispecies models are different. If you are just starting out with SLiM, you should probably focus on single-species models first, as this manual will do.

## 1.10 Online resources for SLiM users

Users of SLiM should be aware of various online resources that are available to support their work. This section summarizes them and provides links. Let's start with resources from the Messer Lab itself (where SLiM is developed by yours truly):

- First of all, there is the **SLiM home page** at the Messer lab website. This is the primary place from which to download SLiM, and provides a history of SLiM releases as well as a list of publications that have used SLiM. <https://messerlab.org/slim/>
- The **slim-announce mailing list** is only for announcements from our group, such as new versions of SLiM, new SLiM publications related to SLiM, and plans for conferences where you could connect with us. <https://groups.google.com/forum/#!forum/slim-announce>
- The **slim-discuss mailing list** is for questions from SLiM users that might be of general interest. Please feel free to post your own questions – and even to answer other people's questions, if you can. <https://groups.google.com/forum/#!forum/slim-discuss>
- The **SLiM Workshop download page** provides downloadable materials for the SLiM Workshop, a 3–5 day workshop for beginning-to-intermediate SLiM users. We have offered the workshop in person several times already; see the SLiM home page for the dates and places of any upcoming scheduled workshops. We have also made the materials available for download, so you can do it on your own, or with a group of friends on the online platform of your choice. If you find it hard to climb the learning curve with

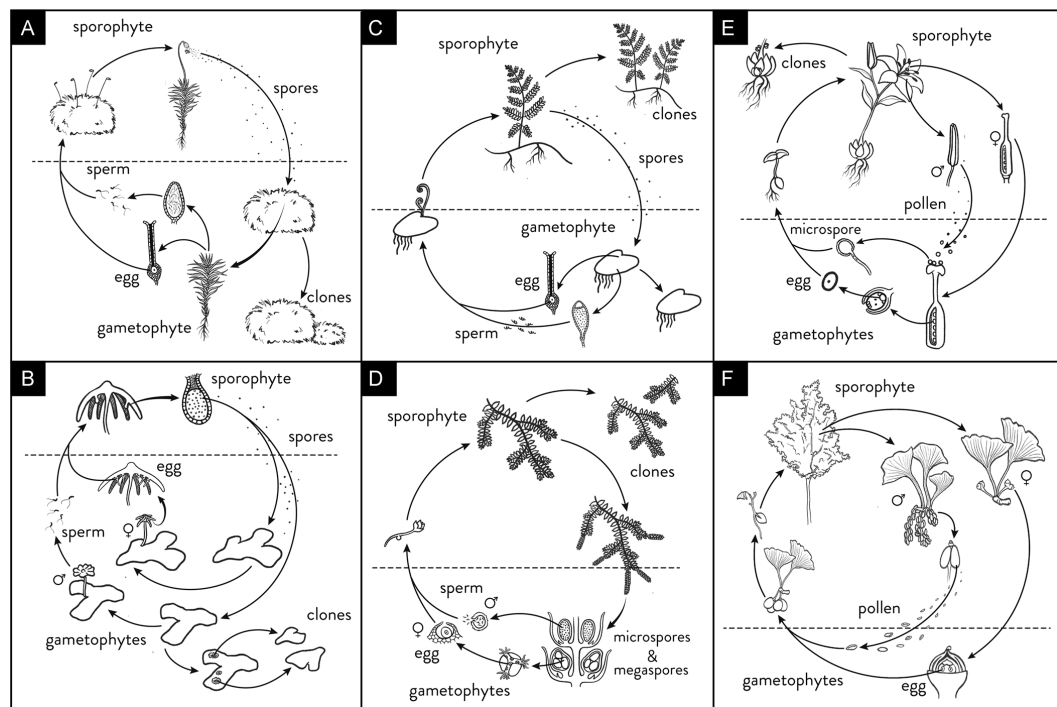
this manual, the workshop is recommended as a more methodical and gentle introduction to SLiM. <http://benhaller.com/workshops/workshops.html>

- The **SLiM-Extras GitHub repository** is a place for the SLiM community to share useful resources related to SLiM. This could include reusable Eidos functions, full SLiM models that might be of general interest, code for reading or analyzing SLiM output in languages like R or Python, code for sublaunching multiple SLiM runs on computing clusters (see section 1.4), and so forth. There is already lots of useful stuff there – notably, a whole tutorial written by Sam Chamber on sublaunching SLiM runs from R, Python, or the Bash shell. Please feel free to email us your own submissions for additions to SLiM-Extras, or better, submit them as pull requests on GitHub. <https://github.com/MesserLab/SLiM-Extras>
- The **SLiM GitHub repository** is where SLiM’s source code lives. SLiM is open-source, but unless you have some specific reason to access the source, you probably don’t need to. As described in chapter 2, installers now exist for all major platforms supported by SLiM (macOS, Linux, and Windows), so building SLiM from sources is uncommon. The GitHub repository, then, is mostly useful for people who want to be on the cutting edge, running the latest version of SLiM before it has been publicly released. Be aware that doing this will mean you are more exposed to bugs, and that sometimes the sources on GitHub may not even build (although we try to avoid that). <https://github.com/MesserLab/SLiM>

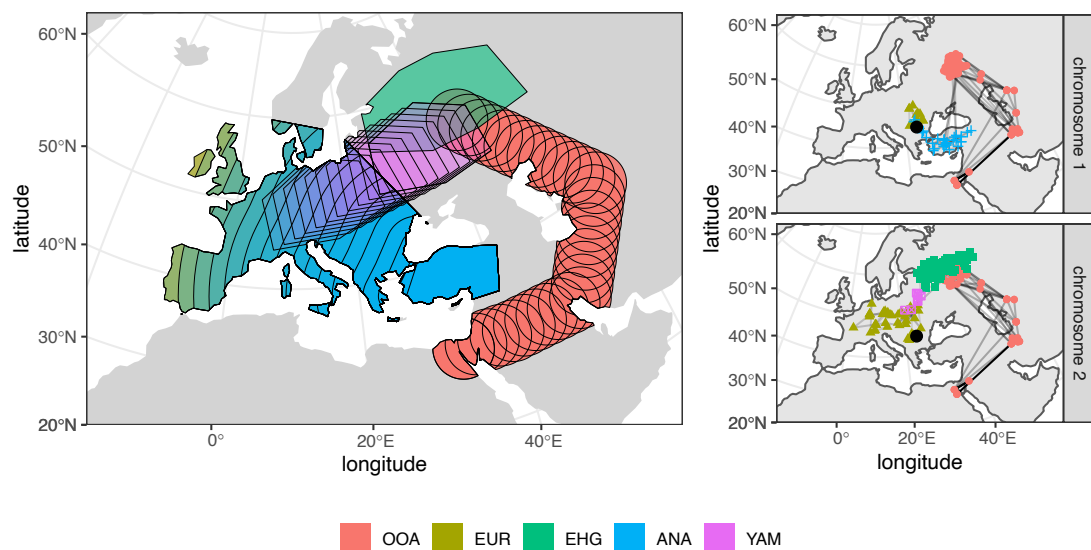
Then there are online resources from related labs and software development groups; these are software packages that are documented, supported, and maintained:

- The **tskit**, **pyslim**, and **msprime** software packages, which are increasingly important for modeling using SLiM (see chapter 17), all have excellent online documentation and resources hosted at <https://tskit.dev>. The **pyslim** docs are particularly helpful for SLiM modelers; start with the Introduction and continue on to the Tutorial and the Vignettes, which are essential for understanding what’s going on with tree sequences on the Python side. Incidentally, if you would like to develop a deeper understanding of tree sequences, how they work and what they’re for, an excellent lecture by Wilder Wohns and Jerome Kelleher is at <https://www.youtube.com/watch?v=X1GEuQrF1jQ>.
- The **stdpopsim** software package, developed by the PopSim Consortium, can simulate a wide variety of standard population genetics models using both **msprime** and SLiM as the “back end” for the simulation. This provides easy access to sophisticated simulations using published demographic models, empirical recombination maps, and more, for a growing collection of species. The best place to start with **stdpopsim** is with their manual, at <https://stdpopsim.readthedocs.io/en/latest/index.html>, or more specifically with the tutorials at <https://stdpopsim.readthedocs.io/en/latest/tutorial.html>.
- The **shadie** Python package, developed by Elissa Sorojrisom, provides a simple Python front end (using SLiM as a back end) for simulations of complex plant mating systems and life cycles, including detailed and biologically accurate simulation of the alternation of generations (i.e., explicitly modeling both the sporophyte and gametophyte stages). It is on GitHub at <https://github.com/elissasoroj/shadie>, and a paper about it can be obtained at <http://doi.org/10.1002/aps3.11472>. Here is a gorgeous diagram, drawn by Elissa Sorojrisom, showing some of the diversity of plant biology that **shadie** makes it easy to

simulate (A–E: monoicous and dioicous bryophytes, homosporous and heterosporous pteridophytes, and monoecious and dioecious spermatophytes):



- The **sleendr** R package, developed by Martin Petr and others, provides a framework for programmable, visual, interactive encoding of spatial population boundaries on real and abstract landscapes, as well as specifying other standard demographic events (population divergences, changes in population size, and gene flow). Here is a quick look at plots produced by **sleendr** for a simple demographic model running on a map of Europe:





The left panel above shows a summary of subpopulation range boundary changes over time in the model, for the five subpopulations shown in the legend at bottom. The right panels show spatial information about the ancestors of one individual (the black dot near the center), for that individual's two chromosomes, illustrating how that individual's spatial pattern of ancestry differs markedly between its maternal and paternal sides.

Spatio-temporal population models in `slendr` are written as simple R scripts and then executed by a SLiM back-end script built into the package. For convenience, a collection of R functions for analyzing and processing output tree sequences and calculating population genetic statistics on them is provided via an R interface to the `tskit` Python module. As you can see, `slendr` is a remarkably powerful tool. Martin has a preprint on `slendr`, available at <https://www.biorxiv.org/content/10.1101/2022.03.20.485041v1>. It includes tutorial vignettes, including one vignette that goes into depth on the plots shown above. You can find `slendr` itself at <https://www.slendr.net>.

- The **`slimr`** R package, developed by Russell Dinnage and others, is for running SLiM models in R. It is on GitHub at <https://rdinnager.github.io/slimr/>, and Russell plans to get it onto CRAN soon. It is “designed to create a seamless link between SLiM 3 and the R development environment”; a preprint describing its many bells and whistles can be found at <https://www.biorxiv.org/content/10.1101/2021.08.05.455258v1>.

And finally, there are miscellaneous third-party online materials provided by individual users of SLiM, which may be less polished / supported. We can't vouch for the accuracy or usefulness of these materials, but they're out there and you might benefit from them:

- Miguel Guardado has produced a Python-based tool called `ped_slim` that can simulate genomic data onto **complex family pedigrees**, represented in `networkx` format (or `.ped` or `.fam` formats, which it can convert to `networkx` format), using SLiM as a back end. Miguel says it is still under development, but is ready for early adopters to try it out. You can find it at [https://github.com/MiguelGuardado/ped\\_slim](https://github.com/MiguelGuardado/ped_slim).
- Zuxi Cui has a GitHub repo about doing **whole-genome simulations**, based upon empirical VCF/FASTA data: [https://github.com/zxc307/GWAS\\_simulation\\_handbook](https://github.com/zxc307/GWAS_simulation_handbook). At the time I added this note it was a work in progress, but it looks very helpful already!
- Cyril Monjeaud has posted a solution for installing SLiM 3 with **Docker**, although I'm not a Docker user and can't vouch for it; it is at <https://github.com/cmonjeau/docker-slim>.
- Jared Galloway has posted some **benchmarking scripts** for SLiM simulations onto GitHub, at [https://github.com/jgallowa07/SLiM\\_TSBenchmarks](https://github.com/jgallowa07/SLiM_TSBenchmarks).
- Peter Ralph has posted **lecture notes and teaching materials** for a course he teaches in Population Biology using SLiM, at <https://petrelharp.github.io/popbio/>, including a very helpful introduction to SLiM using Jupyter notebooks; this would be a good starting point for anyone thinking of using SLiM in the classroom.



- Graham Gower has posted tools for using **Demes demographic models** in SLiM (as described [here](#)), at <https://github.com/grahamgower/demes-slim>.
- Chris McAllester posted a **SLiM syntax file** for the Sublime Text editor onto the slim-discuss list, at <https://groups.google.com/g/slim-discuss/c/bYh-ZiGf8rY>. Downloading and installing this as he describes will provide SLiM syntax coloring in Sublime Text, if you want to use that editor instead of SLiMgui. He notes that it may not be complete and bug-free; improvements are welcome, as are SLiM syntax files for other popular editors.
- There are lots of papers out there that use SLiM for one purpose or another, and many of them even have open-source SLiM model code posted on GitHub. This can be a valuable resource – for seeing how other people are using SLiM, for finding specific SLiM coding solutions, and even for finding collaborators.

There are probably other resources out there as well; if you're aware of something useful that should be added to this list, please let us know. Similarly, drop us a line if any of the links above have become stale, or are no longer useful.

## 2. Installation

The `slim` command-line tool for SLiM is designed to be portable; it is written in pure C++, with no external dependencies. Code from various third-party libraries such as the GNU Scientific Library (Galassi et al. 2009), Boost (Boost 2015), and `tskit` (Kelleher et al. 2016) is used in SLiM (see chapter 30), but that code has been integrated directly into SLiM, so those libraries are not required in order to build or run SLiM. SLiM should be buildable on macOS, Linux, other Unix platforms, Windows, and – possibly with minor modifications – on other platforms as well.

The SLiMgui application is also fairly portable; it is written on top of the Qt 5 widget kit (Qt is pronounced “cute”), which exists on macOS, Linux, and Windows, and also runs under the Windows Subsystem for Linux. Since the code for the Qt 5 widget kit is not included inside SLiMgui’s code, the Qt 5 library has to be installed first as an external dependency (if it is not already present) in order to build SLiMgui. (The original SLiMgui application was written for macOS only, in Objective-C++ using Apple’s Cocoa library; it is now called SLiMguiLegacy, and is no longer supported. The new Qt-based SLiMgui was originally called QtSLiM, and you will still see that term referenced occasionally in this chapter, because the QtSLiM name lives on in SLiM’s build system.)

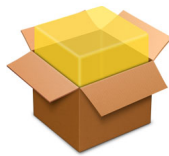
The steps to install SLiM and SLiMgui depend upon the platform you are on; please refer to the appropriate subsection below.

### 2.1 Installation on macOS

On macOS you have a choice between (1) installing a prebuilt package containing the `slim` command-line tool and the SLiMgui application (as well as EidosScribe, the `eidos` command-line tool, and manual PDFs), or (2) cloning the SLiM GitHub repository and building the targets yourself using Apple’s Xcode development environment. Unless you are an experienced developer, the first option is recommended. Building SLiM on macOS at the command line is *not* recommended, although it is possible; use of the Xcode project will provide the proper compiler settings, etc., for a standard macOS build.

#### 2.1.1 Installing the prebuilt SLiM package on macOS

Installing SLiM with the prebuilt installer package is quite straightforward. First, download the installer package from SLiM’s home page at <http://messerlab.org/slim/>, which looks like this:



If multiple versions are available, be sure to download the appropriate version for your system. Once it is downloaded, double-click the package in the Finder to run the installer. Click through all steps in the installer, and SLiM will now be installed on your system. The applications (SLiMgui and EidosScribe) will be installed in your `/Applications` folder; the command-line tools (`slim` and `eidos`) will be installed in `/usr/local/bin/`. Pre-existing Terminal windows may not find `slim` and `eidos`; open a new Terminal window to get the updated paths.

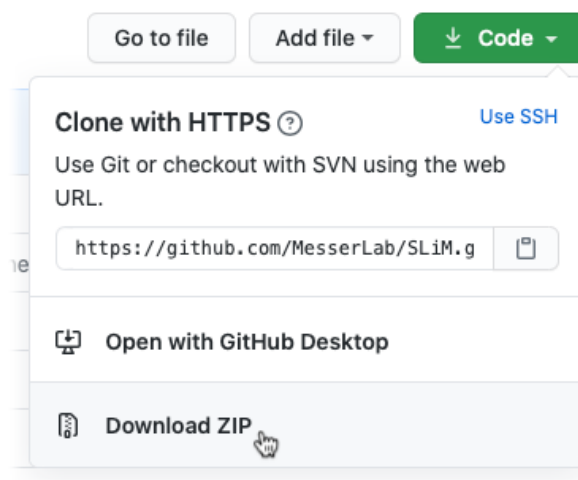
#### 2.1.2 Building SLiM from sources on macOS

This option is relatively complex. If you are not an experienced developer it is recommended that you install SLiM using the prebuilt package instead (see the previous section). There is no

advantage to building SLiM from sources unless you wish to run it under the debugger, modify its code, or other similarly advanced tasks, or wish to run the current development version of SLiM.

First of all, you need to have Xcode and Apple's other developer tools installed on your machine. SLiM's project is generally kept synchronized with the current version of Xcode; older versions of Xcode may be unable to open the SLiM project. To run the current version of Xcode, you generally need to be on the current version of macOS as well, so you may need to upgrade your macOS installation before installing Xcode. Once you have done that, you will need to obtain and install Xcode itself. This will probably involve registering as a developer with Apple (which is free, for the basic level) at their developer website, [developer.apple.com](https://developer.apple.com), and then downloading and installing the developer tools package including the latest version of Xcode (which may be possible through the App Store, otherwise through the developer website's Member Center). It is quite a large download, so do it through a fast connection. Once you have completed this step, you should see Xcode.app in your /Applications folder, and you should be able to launch it by double-clicking it.

Second, you need to obtain the SLiM source code. If you are following these instructions, you probably want to obtain the current development version of SLiM from its GitHub repository at <https://github.com/MesserLab/SLiM>. You can of course clone the SLiM repository if you know how to do so; otherwise, just go to <https://github.com/MesserLab/SLiM>, click the big green "Code" button, and then click "Download ZIP":

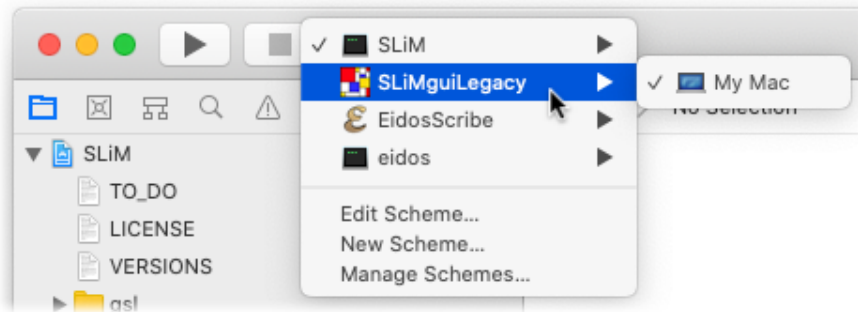


Locate the downloaded file and double-click it to decompress the zip file if that has not already been done for you automatically by your browser. This distribution will include the sources to allow you build both the `slim` command-line tool and the SLiMguiLegacy interactive graphical modeling environment (building SLiMgui, rather than SLiMguiLegacy, is done at the command line; see section 2.4). Be aware that the current development version on GitHub may not be thoroughly tested – indeed, it may not even compile. We try to keep it working, though.

Third, open the SLiM project in Xcode. To do so, locate the Xcode project file within the archive you have just downloaded; it is at the root level, with the name `SLiM.xcodeproj`. Double-click it to open the project in Xcode. You should see a big project window.

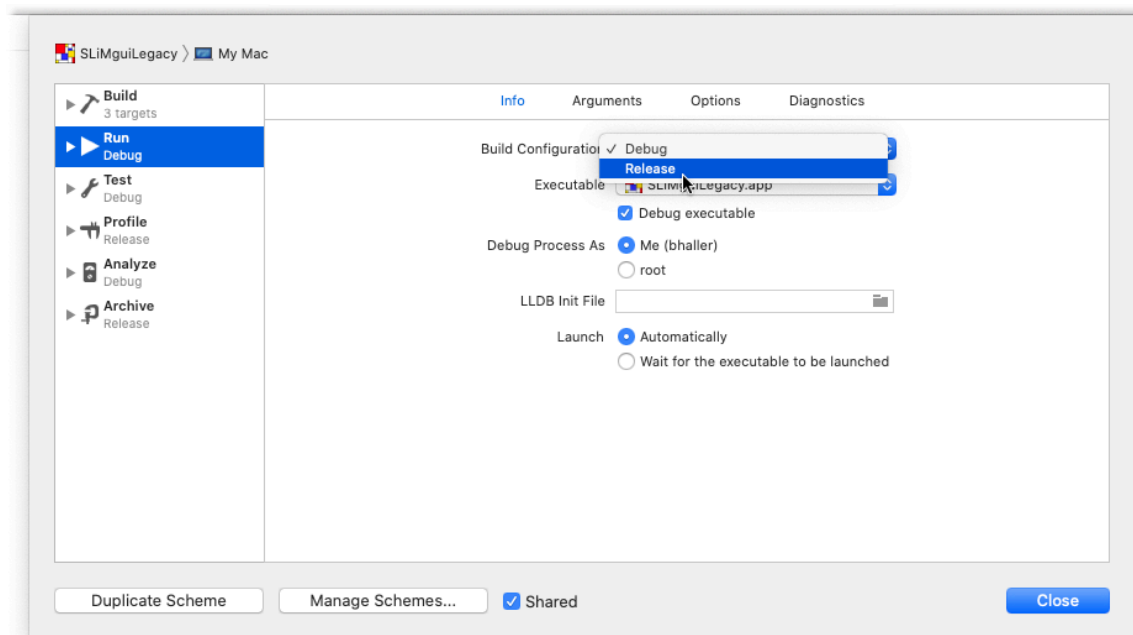
Fourth, choose a build target. A target is basically a product that an Xcode project knows how to build. The SLiM project has four targets that may be selected: SLiM (the `slim` command-line tool), SLiMguiLegacy, EidosScribe (an interactive Eidos script development environment), and Eidos (the Eidos interpreter command-line tool, `eidos`). For now, select SLiMguiLegacy; it is no longer

supported (SLiMgui should be used instead), but it is still usable, and it will allow us to explore Xcode's options a bit more. You select the SLiMguiLegacy target in the pop-up near the upper left of the project window:



Fifth, build and run the selected target by pressing command-R (which is the Run command in the Product menu). It should build fairly cleanly (perhaps apart from some nib warnings that are difficult to eliminate). Once it finishes building (which may take several minutes, depending on your machine), SLiMguiLegacy should launch automatically. Assuming that worked, quit SLiMguiLegacy for now, as we are not quite done setting things up in Xcode.

Sixth, there is an additional twist: the build configuration and other build scheme options. These options are accessed by choosing the “Edit Scheme...” menu item that can be seen in the pop-up menu in the above screenshot. Choose this, and you will see a sheet:



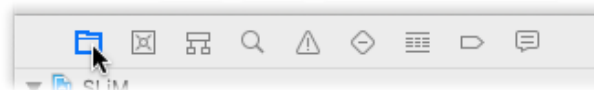
In this screenshot, the “Run” action has been selected on the left, so we are configuring what Running this target (the SLiMguiLegacy target) will do. At the top, the Info tab is selected, which provides the most basic configuration options. Finally, the pop-up menu next to the label “Build Configuration” has been clicked in order to choose the Release build configuration. In general, you will want to build and run SLiM and SLiMguiLegacy using the Release build configuration unless you have a specific reason to wish to do otherwise; Release builds are much faster than

Debug builds, partly because of optimization, and partly because additional runtime checks are turned on in SLiM's code when built in the Debug configuration. After choosing Release, you can click the Close button, and command-R (run) will now build and run a Release build of SLiMguiLegacy.

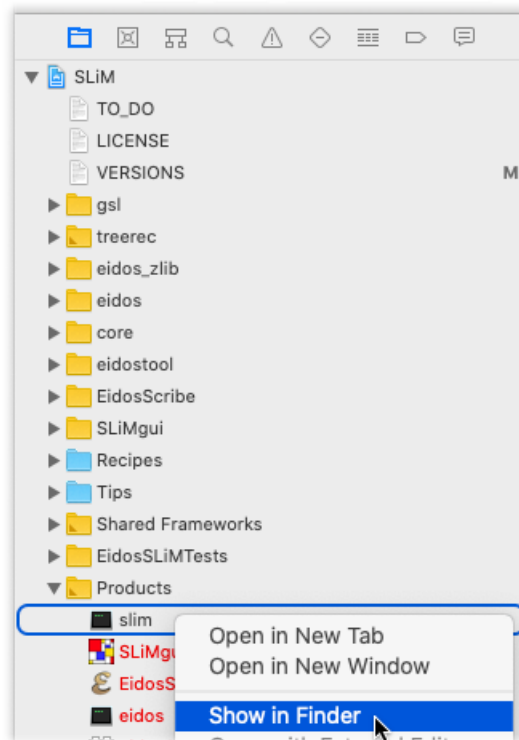
If your goal is to run SLiMguiLegacy, you can simply run it from within Xcode; there is no particular disadvantage to doing so. If your goal is to run the `slim` command-line tool, doing so from within Xcode is a little bit inconvenient (since it is a little bit complicated to supply command-line arguments to it, for example), so the simplest course is to build `slim` in Xcode and put the built executable wherever you want it to be so that you can run it in Terminal as usual.

To do this, first select the SLiM target from the target pop-up in the project window, as described above (since the SLiMguiLegacy target is presently selected, if you have been following along). Second, choose Edit Scheme... again and make sure that the SLiM target is using the Release build configuration for its Run action, as described above (since we set that for the SLiMguiLegacy target above, not for the SLiM target – each target has its own settings), and close the Edit Scheme... sheet. Third, press command-B to build the target (this is the Build command in the Product menu).

Once the build completes, you will want to locate the built product in the Finder. To do this in Xcode 12 and earlier (see below for Xcode 13 and later), first select the Project Navigator in the project window by clicking the folder icon at the left edge of the strip of icons at the top left of the project window:



Then click disclosure triangles to open the SLiM top-level item and the Products subfolder in the outline view shown there (if you don't see this, read on below):



The `slim` product may be shown in red even if it has been successfully built; this appears to be a bug in Xcode. Right-click (i.e., click with the right-hand mouse button) or control-click (i.e., hold down the control key and click) on the `slim` product, and choose “Show in Finder” from the context menu displayed, as shown in the screenshot above. Your computer should switch to the Finder and open a new window in the Finder, showing the contents of a folder that is probably named “Release” (because the Release build configuration has been chosen). A file named `slim` should be selected in this window. This is the built executable for the `slim` command-line tool. (There are other, more standard ways to get to this point, but they are a bit more complicated.)

For Xcode 13 and later, the workflow above no longer works because the Product folder is no longer shown in the Project Navigator. Instead, choose the “Show Build Folder in Finder” menu item from the Product menu. That will open a Finder window showing the folder within which all of your build products are located. Inside that folder should be a Products folder, and inside that should be Release and Debug subfolders (if you have built Release and Debug targets, at least), and inside those are your individual build products such as the Release build of `slim`.

Having found the built `slim` product in the Finder, you should copy this file to whatever location you wish, and then run `slim` using that copy. The standard install location for `slim` is at `/usr/local/bin/`, but since this is a custom build you might not wish to put it at that location to avoid confusion. Instead, a location like `~/bin/` inside your home directory might be appropriate (you might need to create this folder first, in the Finder). In any case, once you have installed `slim` at the desired location, you can open a Terminal window and `cd` to that location (your Terminal shell prompts may look different from mine, of course):

```
darwin:~ bhallar $ cd ~/bin
darwin:~/bin bhallar $
```

Then you can run the local copy of `slim` that is at that location:

```
./slim <rest of command line>
```

The syntax `./slim` tells Terminal to run the copy of SLiM in the current directory, rather than running the standard version that might be installed at `/usr/local/bin/` on your machine. You can verify that the correct version of SLiM is being run using the `-v` (version) option:

```
darwin:~/bin bhallar $ ./slim -v
SLiM version 2.3, built Apr 18 2017 17:20:34
```

The build date and time should correspond with the build you just did in Xcode; if not, you have done something wrong and should re-check the steps above.

You can in fact follow the same steps to locate the built SLiMguiLegacy application in the Finder and install it somewhere on your machine for later use; the `~/Applications/` folder might be a good choice of location. However, it is probably simpler to run it from within Xcode – or to build and run the new SLiMgui app instead (see section 2.4.1).

Obviously Xcode is a very complicated application, and we can’t provide a thorough introduction to using it, but the steps above should allow you to build and run both `slim` and SLiMguiLegacy using the current development head sources from GitHub. If you run into any problems with these instructions, please let us know.



## 2.2 Installation on Linux and other Un\*x platforms

For users on Linux, there may be an installer for SLiM and SLiMgui available; see section 2.2.2. If not, you can build the `slim` and `eidors` command-line tools following the instructions in section 2.2.3, and the SLiMgui modeling environment following section 2.4.2.

Building the command-line tools from sources is very easy, whereas building SLiMgui from sources is more complicated. For this reason, as of SLiM 4.0.1, a portable option is also provided for installing *only* the SLiMgui graphical modeling environment; see the Flatpak installation instructions in section 2.2.1. Particularly if a platform-specific installer is not available for your Linux distribution, you may wish to use the Flatpak to get SLiMgui (following section 2.2.1) and then build the command-line tools from sources yourself (following section 2.2.3). If all you want is SLiMgui, the Flatpak may also be useful.

### 2.2.1 Installing SLiMgui using Flatpak on Linux

Flatpak is an encapsulated, or containerized, application runtime and platform target for thirty-five major Linux distributions. The chances are very high that if you use a graphical environment with the Linux kernel your environment is supported by Flatpak, whatever user-space software you might run, whether GNU or non-GNU.

To install SLiMgui on Linux with Flatpak:

1. Complete the Flatpak Setup procedure for your operating system: <https://flatpak.org/setup/>.
2. If the Flatpak Setup procedure did not enable the Flathub repository, enable it following the additional instructions found in that link.
3. Install SLiMgui using a “software centre” or “store” (this is a repository, not a marketplace) like GNOME Software or KDE Discover, if those applications have Flatpak and Flathub integration (they should). If not, navigate to <https://flathub.org/home> and search for SLiMgui, and install it using the link provided on that page.
4. If neither of step 3’s options work for you, then you can instead install SLiMgui with the command-line `flatpak` tool:

```
flatpak install SLiMgui
```

The instructions above require you to enable the Flathub repository. If you are not comfortable with that, for whatever reason, you may instead request a Flatpak bundle (a non-updating version of a Flatpak application) from the package maintainer, Bryce Carson.

### 2.2.2 Installing SLiM with a platform-dependent binary package on Linux

With extensive help from the SLiM community, we now have installers for SLiM on many Linux platforms as detailed below. If an installer is not available for your platform, you will need to build from SLiM and/or SLiMgui from source (see sections 2.2.3 and 2.4.2) – or set up an installer for your platform that the whole community can use, if you know how to do that. Check the SLiM home page (<https://messerlab.org/slim/>), the `slim`-announce list on Google Groups (<https://groups.google.com/g/slim-announce>), or SLiM’s GitHub repository (<https://github.com/MesserLab/SLiM>) for information on new installers or beta versions of installers that need community testing and feedback.

### Debian and Ubuntu:

The only non-Flatpak installation option for Debian and Ubuntu users is a Bash script, `DebianUbuntuInstall.sh`. Unlike the Flatpak, the Bash script also builds and installs both of the command-line tools alongside the SLiMgui modelling environment. It provides desktop integration for SLiMgui, but there is no defined uninstallation or removal method other than what a system administrator should be capable of performing after reading the installation script.

*(Advanced users: If you would prefer a DEB package, please understand that the packaging process and distribution methodology for these platforms is difficult, and despite significant time investment no satisfactory option has been found. If you are an experienced Debian user or developer, with experience deploying independently compiled binaries for multiple Debian (and preferably also Ubuntu) versions/releases, and you wish to enter SLiM (both the command-line tools and SLiMgui) into the official repositories and maintain that package, please open a pull request against the `debian/` folder sources on GitHub, tag @bryce-carson, and you'll receive thanks for any improvements to the methodology discussed in the now-closed [Issue 288](#).)*

The rest of this section will discuss the use of the installer script, which is the recommended solution for both Debian and Ubuntu at this time. The installation script downloads the source code archive for the current release version of SLiM, builds its targets with `cmake` and `make`, and installs `slim`, `eidos`, and `SLiMgui` in `/usr/bin`. The `/usr/bin` directory is normally in the search path defined by the Unix `$PATH` environment variable, so the `slim` and `eidos` command-line programs should be available in your terminal without any extra work; `SLiMgui` can also be launched from the terminal, but is integrated with the desktop environment in FreeDesktop compliant environments. If this script works for you, then you are completely done.

The installer script is hosted in the SLiM-Extras repository on GitHub; you can view the script at <https://raw.githubusercontent.com/MesserLab/SLiM-Extras/master/installation/DebianUbuntuInstall.sh>. Note that it requires `cmake`, `qmake`, `Qt`, and either `curl` or `wget` to be installed on your system first; if they are not, it will print instructions on how to install them. Follow those instructions to install the needed packages, and then try running the install script again.

Note that because this script installs the built components into `/usr/bin`, the `sudo` command is used to run the script with root privileges, so you will need to enter your system's root password. If you do not have root access to your system, you won't be able to use this script directly; you might request that your system administrator run it for you. Alternatively, you could download and build SLiM yourself, without installing it into `/usr/bin`, either following the script's outline by hand, or using the build instructions in sections 2.2.3 and 2.4.2.

If you have `wget` installed (which appears to be present by default on Ubuntu), you can run the install script directly from the web with the following single command line (which should be entered as a single line in Terminal):

```
wget --output-document /dev/stdout - --quiet https://
raw.githubusercontent.com/MesserLab/SLiM-Extras/master/installation/
DebianUbuntuInstall.sh | sudo bash -s
```

or if you have `curl` installed and prefer to use it, you can execute this single command line to install directly from the web:

```
curl --silent https://raw.githubusercontent.com/MesserLab/SLiM-Extras/
master/installation/DebianUbuntuInstall.sh | sudo bash -s
```

These two options should be identical for all practical purposes. They download the install script from its URL and pipe it directly into `bash` to be executed.

However, since the script is run as root, you may wish to examine it locally before running it (if you don't trust us – and why would you?). To do so, first fetch the script with `wget`:

```
wget https://raw.githubusercontent.com/MesserLab/SLiM-Extras/master/
installation/DebianUbuntuInstall.sh
```

or with `curl`:

```
curl https://raw.githubusercontent.com/MesserLab/SLiM-Extras/master/
installation/DebianUbuntuInstall.sh > DebianUbuntuInstall.sh
```

or with the browser of your choice, using the link above. Examine the script to your satisfaction, then make it executable with `chmod` and run the script with `sudo`:

```
chmod +x ./DebianUbuntuInstall.sh
sudo ./DebianUbuntuInstall.sh
```

You may remove the script afterwards if you like, or retain it so you know what changes were made to your system. An uninstallation script is not provided; if you need help removing the software installed by the script, please make a request for help with uninstallation instructions at <https://github.com/MesserLab/SLiM-Extras/issues/new>, or study the installation script and simply remove the files that were installed.

Please direct issues and pull requests to SLiM-Extras on GitHub.

### **RPM-based distributions: Fedora, CentOS, RedHat Enterprise Linux, openSUSE, and SUSE Linux Enterprise:**

Bryce Carson has kindly created binary packages for RPM-based Linux distributions, especially those in the Fedora family. The packages are hosted on a COPR repository (“copper”); COPR is a Fedora community service that builds packages and hosts third-party software repositories.

**Fedora-based distributions** (including RHEL and CentOS) need only enable the Copr repository, and then the package will be available for installation:

```
sudo dnf copr enable bacarson/SLiM-Selection_on_Linked_Mutations
sudo dnf install SLiM
```

**openSUSE and SUSE Linux Enterprise** users must follow a slightly different set of instructions, since these distributions do not use the DNF package manager, but instead use Zypper. The repository file must be added to the directory containing all such files which Zypper manages.

*openSUSE Leap 15.3:*

```
sudo zypper add-repo https://copr.fedorainfracloud.org/coprs/bacarson/
SLiM-Selection_on_Linked_Mutations/repo/opensuse-leap-15.3/bacarson-SLiM-
Selection_on_Linked_Mutations-opensuse-leap-15.3.repo
```

*openSUSE Leap 15.4:*

Download the RPM file from <https://packagecloud.io/bryce-a-carson/SLiM>. Then open a terminal and navigate to the folder it was saved in, and install it by running:

```
sudo zypper install SLiM-4.0-1.x86_64.rpm
```

Note that deployment of the RPM for this platform will typically be delayed by approximately two weeks.

*openSUSE Tumbleweed:*

```
sudo zypper add-repo https://copr.fedorainfracloud.org/coprs/bacarson/
SLiM-Selection_on_Linked_Mutations/repo/opensuse-tumbleweed/bacarson-SLiM-
Selection_on_Linked_Mutations-opensuse-tumbleweed.repo
```

*Either Leap 15.3 or Tumbleweed:*

```
sudo zypper refresh
sudo zypper install SLiM
```

The installed executables (`slim`, `eidos`, and `SLiMgui`) are placed in `/usr/bin`. The `/usr/bin` directory is normally in the search path defined by the `$PATH` environment variable, so the `slim` and `eidos` command-line programs should be available in your terminal without any extra work; `SLiMgui` can also be launched from the terminal, but is integrated with the desktop environment in FreeDesktop compliant environments (GNOME, KDE, MATE, XFCE, LXQt, and so forth).

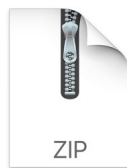
### Arch Linux:

Graham Gower has kindly created an installer package for Arch. The first step is to acquire the PKGBUILD file from <https://aur.archlinux.org/packages/slim-simulator>; then install with `makepkg -si`. This package is hosted on the Arch User Repository; more detailed instructions for installing packages from that repository can be found on the Arch Linux wiki at [https://wiki.archlinux.org/index.php/Arch\\_User\\_Repository#Installing\\_packages](https://wiki.archlinux.org/index.php/Arch_User_Repository#Installing_packages). If that procedure works for you, you now have `slim` and `SLiMgui`! If you are using a standard desktop environment, such as GNOME or KDE, and `SLiMgui` has not been integrated with the desktop environment by this procedure, you may want to run the relevant commands in the `DebianUbuntuInstall.sh` script described in the instructions for Debian and Ubuntu users before this subsection. The script is hosted in the `SLiM-Extras` repository on GitHub, at <https://raw.githubusercontent.com/MesserLab/SLiM-Extras/master/installation/DebianUbuntuInstall.sh>.

#### 2.2.3 Building SLiM from sources on Linux

It is recommended that users on Linux use their platform's installer for `SLiM` if possible; that will avoid these complicated instructions. See section 2.2.2 for details.

Details of building `SLiM` at the command line on Linux and other Un\*x platforms (including macOS) may vary depending on the platform, but the basic gist should be the same. First, you need to obtain the `SLiM` source code. It is recommended that you obtain the code for the latest supported release from `SLiM`'s home page at <http://messerlab.org/slim/>:



However, you may also obtain the current development version from `SLiM`'s GitHub repository at <https://github.com/MesserLab/SLiM>. Be aware that the version on GitHub may not be thoroughly tested – indeed, it may not even compile. We try to keep it working, though.

The following steps have changed considerably with the release of `SLiM` 3, since we have switched from using the `make` build tool to using a tool called `cmake` that is considerably more modern (but also a bit more complicated to use). The `cmake` tool is often preinstalled on Linux systems, but if it is not installed on your machine, you will need to install it before proceeding. You can tell whether `cmake` is installed on your system by executing this in a terminal window:

```
which cmake
```

If a path is printed in response, `cmake` is installed; if not, it needs to be installed. On macOS, `cmake` can be installed using MacPorts (<https://www.macports.org>) or Homebrew (<https://brew.sh>); which installation system you use seems to be largely a matter of taste, although you can read strong opinions in both directions online. On Linux and other Un\*x systems, you can install `cmake` using your package manager (e.g., `sudo apt-get install cmake` on Ubuntu and Debian), or directly from the `cmake` home page at <https://cmake.org/download>. The `cmake` tool builds a makefile for us, which can then be used to build with `make`, as we will see next.

With `cmake` installed, go to a new Un\*x shell window and type:

```
cd SLiM
cd ..
mkdir build
cd build
cmake ../SLiM
make slim
```

The first `cd` command changes the current directory to your SLiM source directory (you will need to supply the appropriate path there instead of just `SLiM`), and the next `cd` command changes to the enclosing directory (of course you can just `cd` there directly). We then use `mkdir` to create a build directory adjacent to the SLiM source directory in the filesystem; this keeps all of the cruft associated with the build process separate from SLiM's sources. We `cd` into that build directory, and then tell `cmake` to update its information regarding SLiM. The last command actually builds the `slim` command-line tool. Note that SLiM uses C++11 extensions; the `g++` compiler installed on your machine must be recent enough to support C++11 or your build will fail. The `slim` tool will appear as an executable file in the `build` directory once the build is finished (not, as in SLiM 2.x, in a `bin` directory inside the source directory).

You can copy the built executable to a different location, such as `/usr/bin/`, if you wish; the standard location for user-built executables depends upon your Un\*x flavor, so consult your documentation to install it in a standard location. Depending upon your system configuration, executing:

```
make install slim
```

may (or may not) automatically copy the `slim` executable to a standard location on your system. You may need administrator privileges to do that, in which case you may need to do:

```
sudo make install slim
```

The `eidos` command-line tool can also be built on Un\*x platforms following the same procedure, with the final command `make eidoss`; a simple `make` command will build both. Building SLiMgui is a more complicated and platform-dependent topic since it depends upon the Qt library (see section 2.4).

Once `cmake`'s information has been set up, rebuilding after minor source changes can generally be done just by re-executing the `make slim` command in the build directory. If you make major changes to the SLiM sources – and in particular, if you add or remove source files, or do a `git pull` of new changes from GitHub – you might need to tell `cmake` to update its information before you run `make` to rebuild the project. This is necessary because SLiM's `CMakeLists.txt` configuration file uses a `cmake` feature called `GLOB` to collect a list of source files to be used for building, and that means that `cmake` cannot automatically re-update in all cases. If you make such changes to the sources, you should do the following:

```
cd SLiM
touch ./CMakeLists.txt
```

Then `cd` to the build directory and execute `make slim` to rebuild; `cmake`'s cached information will be rebuilt, since the configuration file was touched, so you do not need to re-run `cmake`.

The build instructions above use a build directory named `build`, and use a default “build type” with `cmake` (producing a Release build), and this will suffice for almost all SLiM users; you need not read further unless you really want to complicate your life. In fact, however, you can name your build directory whatever you wish, and you can have more than one build directory if you



wish; and you can specify a build type of either Release or Debug with `cmake`. For example, you could do this to make a build directory named `Release`, using the Release build type:

```
cd SLiM
cd ..
mkdir Release
cd Release
cmake -D CMAKE_BUILD_TYPE=Release ../SLiM
make slim
```

A Release build has optimization turned on and debugging symbols turned off, giving you a lean, fast executable suitable for most purposes. This is `cmake`'s default when building SLiM; the original build instructions above produced a Release build since no build type was specified. Similarly, you could make a Debug build in a directory named `Debug`:

```
cd SLiM
cd ..
mkdir Debug
cd Debug
cmake -D CMAKE_BUILD_TYPE=Debug ../SLiM
make slim
```

A Debug build has optimization turned off and debugging symbols turned on, giving a large symbolicated build suitable for runtime debugging. Debug builds also have some additional runtime error checking turned on in SLiM, designed to catch a variety of internal error states that might cause a Release build to crash or produce incorrect output. Most users, however, will be fine with just the standard build directory named `build` and the default `cmake` build type.

## 2.3 Installation on Windows

For users on Windows (64-bit only, Windows 7 or later; only tested on Windows 10 and 11, so those are recommended), an installer that provides both `slim` and `SLiMgui` is available; see section 2.3.1. If you wish to build from sources (perhaps to get the latest changes from GitHub), see section 2.3.2. Note that SLiM can also be installed on Windows under the WSL (Windows Subsystem for Linux) as detailed in section 2.4.6; which environment is the better choice may be a matter of taste, but may also depend upon what other tools you plan to use alongside SLiM.

The Windows port of SLiM, the `pacman` installer package, and the writeup of these installation instructions were done by Russell Dinnage, the author of the `slimr` R package (see section 1.10). Thanks Russell!

### 2.3.1 Installing SLiM with an installer on Windows

On Windows, a full installation of SLiM, including the `slim` and `eidos` command-line tools and `SLiMgui`, can be done using the MSYS2 Software Distribution and Building Platform for Windows. MSYS2 can be downloaded from <https://www.msys2.org/>. After following the MSYS2 installation instructions, you can install SLiM using the following steps:

- (1) Click on “search” in the Start menu, and start typing `msys2`. Click on “MSYS2 MSYS” when it appears. This will open an MSYS2 command prompt.

- (2) At the MSYS2 command prompt, make sure the MSYS2 package manager is fully up to date by typing:

```
pacman -Syu
```

- (3) Then install SLiM by typing:



`pacman -S mingw-w64-x86_64-slim-simulator`

You should see something like this, ending with a prompt:

```
MSYS ~
$ pacman -Sy mingw-w64-x86_64-slim-simulator
:: Synchronizing package databases...
mingw32             1465.2 KiB   392 KiB/s 00:04 [#####] 100%
mingw64             1472.4 KiB   318 KiB/s 00:05 [#####] 100%
ucrt64             1614.9 KiB   376 KiB/s 00:04 [#####] 100%
clang64            1457.4 KiB   385 KiB/s 00:04 [#####] 100%
msys                387.5 KiB   108 KiB/s 00:04 [#####] 100%
resolving dependencies...
looking for conflicting packages...
warning: dependency cycle detected:
warning: mingw-w64-x86_64-harfbuzz will be installed before its mingw-w64-x86_64-freetype dependency

Packages (18) mingw-w64-x86_64-dbus-1.12.20-4
               mingw-w64-x86_64-double-conversion-3.1.5-1
               mingw-w64-x86_64-fontconfig-2.13.94-1
               mingw-w64-x86_64-freetype-2.11.0-2  mingw-w64-x86_64-glib2-2.70.0-1
               mingw-w64-x86_64-graphite2-1.3.14-2
               mingw-w64-x86_64-harfbuzz-2.9.1-1  mingw-w64-x86_64-icu-69.1-1
               mingw-w64-x86_64-libjpeg-turbo-2.1.1-1
               mingw-w64-x86_64-libpng-1.6.37-6  mingw-w64-x86_64-md4c-0.4.8-1
               mingw-w64-x86_64-pcre-8.45-1  mingw-w64-x86_64-pcre2-10.36-1
               mingw-w64-x86_64-qt5-base-5.15.2+kde+r96-1
               mingw-w64-x86_64-vulkan-headers-1.2.179-2
               mingw-w64-x86_64-vulkan-loader-1.2.179-2
               mingw-w64-x86_64-wineditline-2.205-3
               mingw-w64-x86_64-slim-simulator-3.7-1

Total Installed Size: 367.34 MiB

:: Proceed with installation? [Y/n] Y
```

Type Y at this prompt to continue. You should then get output like this:

```
:: Proceed with installation? [Y/n] Y
(18/18) checking keys in keyring [#####] 100%
(18/18) checking package integrity [#####] 100%
(18/18) loading package files [#####] 100%
(18/18) checking for file conflicts [#####] 100%
(18/18) checking available disk space [#####] 100%
:: Processing package changes...
(1/18) installing mingw-w64-x86_64-wineditline [#####] 100%
(2/18) installing mingw-w64-x86_64-pcre [#####] 100%
(3/18) installing mingw-w64-x86_64-glib2 [#####] 100%
(4/18) installing mingw-w64-x86_64-dbus [#####] 100%
(5/18) installing mingw-w64-x86_64-double-con... [#####] 100%
(6/18) installing mingw-w64-x86_64-libpng [#####] 100%
(7/18) installing mingw-w64-x86_64-graphite2 [#####] 100%
(8/18) installing mingw-w64-x86_64-harfbuzz [#####] 100%
Optional dependencies for mingw-w64-x86_64-harfbuzz
  mingw-w64-x86_64-icu: harfbuzz-icu support [pending]
  mingw-w64-x86_64-cairo: hb-view program
(9/18) installing mingw-w64-x86_64-freetype [#####] 100%
(10/18) installing mingw-w64-x86_64-fontconfig [#####] 100%

Fontconfig configuration is done via /mingw64/etc/fonts/conf.avail and conf.d.
Read /mingw64/etc/fonts/conf.d/README for more information.

updating font cache... done.
(11/18) installing mingw-w64-x86_64-icu [#####] 100%
(12/18) installing mingw-w64-x86_64-libjpeg-turbo [#####] 100%
(13/18) installing mingw-w64-x86_64-md4c [#####] 100%
(14/18) installing mingw-w64-x86_64-pcre2 [#####] 100%
(15/18) installing mingw-w64-x86_64-vulkan-headers [#####] 100%
(16/18) installing mingw-w64-x86_64-vulkan-loader [#####] 100%
(17/18) installing mingw-w64-x86_64-qt5-base [#####] 100%
Optional dependencies for mingw-w64-x86_64-qt5-base
  mingw-w64-x86_64-libmariadbclient
  mingw-w64-x86_64-firebird2
  mingw-w64-x86_64-postgresql
(18/18) installing mingw-w64-x86_64-slim-simulator [#####] 100%
:: Running post-transaction hooks...
(1/1) Updating fontconfig cache...
```

SLiM should now be installed. You can find `slim.exe`, `eidos.exe`, and `SLiMgui.exe` in your MSYS2 installation directory under `<msys2_install>\mingw64\bin`. In a typical MSYS2 install, for example, you could find your executables in `C:\msys64\mingw64\bin`.

All executables should now work if you run them at a command prompt. You may also double-click on the `SLiMgui.exe` executable to run the SLiMgui graphical user interface. You may want to create a shortcut to SLiMgui for easier access.

### 2.3.2 Building SLiM from sources on Windows

Rather than installing SLiM on Windows using the pacman installer tool (see section 2.3.1), you can build SLiM from sources if you wish. This process is relatively complicated and error-prone; therefore, it is recommended that you use the installer instead, as described above, unless you need to run SLiM under the debugger, or obtain the latest changes from GitHub.

First, you will need the MSYS2 Software Distribution and Building Platform for Windows, which has the build tools necessary to compile SLiM on Windows. MSYS2 can be downloaded from <https://www.msys2.org/>. After following the MSYS2 installation instructions, you can compile and install SLiM using the following steps:

(1) Download and unzip SLiM's source code, either from <https://messerlab.org/slim/> (for the last released version) or from SLiM's GitHub repository at <https://github.com/MesserLab/SLiM> (for any release version, or for the current development head).

(2) Click on "search" in the Start menu, and start typing `msys2`. Click on "MSYS2 MinGW 64-bit" when it appears. This will open an MSYS2 command prompt. Note that it is important that you choose the "MinGW 64-bit MSYS" command, rather than the "MSYS2 MSYS" command, because this makes sure the environment is set up correctly to run the correct build tools.

(3) At the MSYS2 command prompt, install prerequisites for SLiM by typing:

```
pacman -Syu
pacman -S mingw-w64-x86_64-cmake
```

(4) Now `cd` into the directory where SLiM was unzipped, and create a new directory to hold the build by typing:

```
mkdir build
```

(5) Build SLiM with the following commands (note the trailing double dots should be included):

```
cd build
cmake -G"MSYS Makefiles" ..
make
```

(6) Install into the directory of your choice with:

```
DESTDIR="<install_dir>" make install
```

For `<install_dir>` you might wish to use the MSYS2 bin directory (`C:\msys64\mingw64\bin` for a typical install), but that is not required.

Note that building from sources can take some time. At the price of slowing down everything else you're doing on your machine, you can speed up compilation by passing the `-j` option to `make`, which tells it to use multiple cores for compilation tasks. For example, if you wish to use six cores for building SLiM, then instead of using just `make`, as above, use:

```
make -j6
```

The instructions above do not build SLiMgui, since that requires a couple of additional steps. Once you have a build of SLiM working with the above instructions, see section 2.4.4 to add SLiMgui to your build.

## 2.4 Building SLiMgui

The SLiMgui application uses a cross-platform object kit called “Qt” (pronounced “cute”). It was originally named QtSLiM (“cute SLiM”), and you will see that name still used internally in SLiM’s code. In SLiM 3.5, it replaced the old Cocoa-based SLiMgui (now called SLiMguiLegacy). This port from Cocoa to Qt provides a graphical modeling environment on Linux, macOS, Windows, and the Windows Subsystem for Linux (WSL), based upon a single unified code base.

This section will discuss how to build SLiMgui, which is fairly complex because of its external dependency on the Qt 5 widget library. An installer that installs both `slim` and SLiMgui might be available for your platform (see the previous sections); if so, that provides a much simpler alternative unless you really need to build SLiM from sources. If you only need to build the `slim` command-line tool, see section 2.1.2 for macOS instructions, section 2.2.3 for Linux instructions (these should work for the Windows Subsystem for Linux also), and 2.3.2 for Windows instructions. If you also need to build SLiMgui, see section 2.4.1 for macOS, section 2.4.2 for Linux, section 2.4.4 for Windows, and section 2.4.6 for the Windows Subsystem for Linux.

### 2.4.1 Building SLiMgui on macOS

You should be running macOS 10.13 or later to run SLiMgui, in general; this is the minimum system version for Qt 5.15.2, which is strongly recommended on macOS. If you’re on an older version of macOS, you can probably install an older version of Qt 5 to match; Qt 5.11.x supports macOS 10.11 and later, for example. However, these combinations have not been tested as extensively, and SLiMgui’s code will produce a compile-time error (which you can disable) if you try to build against any Qt 5 version earlier than 5.15.2, as a safeguard against people using an old Qt 5 version unintentionally. Note that Qt 6 has been released, but SLiMgui has not been ported to it; Qt 5 is required. Alternatively, you can continue to use the SLiMguiLegacy app, which can be built on macOS in Xcode (see section 2.1.2); it is no longer supported, but may work for you.

Please note that for end users of SLiM **there is usually no need to build SLiMgui on macOS**; it is included in the standard double-click installer for SLiM on macOS. If you download and run that installer (see section 2.1.1), you should then be all set; you do not need to read this section.

For those who do want to build SLiMgui themselves (perhaps to get bug fixes and other changes from the GitHub head before they are released, or to be able to run SLiMgui under the debugger), this section will show how to build SLiMgui on macOS.

**The first step is to install Qt**, which is not installed by default on macOS. There are several ways to do this: via MacPorts, via Homebrew, or via the Qt Online Installer; read on.

If you are a MacPorts user, you can install it with `sudo port install qt5`; this will give you Qt 5.15.2 as of this writing, so it should work well. Other versions are also available, such as `qt59` (Qt 5.9.9), `qt511` (Qt 5.11.3), and `qt513` (Qt 5.13.2), but see the warning above; in any case, you should not use a Qt version older than 5.9.5. The last time I checked, MacPorts did not automatically set up a symlink to `qmake`, so after installing you should probably execute the command `sudo ln -s /opt/local/libexec/qt5/bin/qmake /opt/local/bin/qmake` to do so (after checking that that is where MacPorts put `qmake` for you). If you want the Qt Creator IDE as well (perhaps to run SLiMgui under the debugger), do `sudo port install qt5-qtcreator`; Qt Creator will then be available at `/Applications/MacPorts/Qt5/Qt Creator.app`.

If you are a Homebrew user, the package named `qt5` currently provides Qt 5.15.3; do *not* use the `qt` package, as that will install Qt 6, which you do not want. The command would thus be

`brew install qt5`; using `sudo` should not be needed, but if the install fails due to permissions then try `sudo brew install qt5`. You can get `qmake` on your `$PATH` (so Terminal can find it) after this install by doing `echo 'export PATH="/opt/homebrew/opt/qt@5/bin:$PATH"' >> ~/.profile` or, alternatively, you can set up a symlink to it with `sudo brew link qt5 --force`, or if that doesn't work, `sudo ln -s /usr/local/Cellar/qt/<QT_VERSION_HERE>/bin/qmake /usr/local/bin/qmake`. If you want the Qt Creator IDE as well (perhaps to run SLiMgui under the debugger), you should be able to install it with `brew install qt-creator` and a link to it in your `/Applications` folder should be created by Homebrew automatically. (You probably also want to do `brew install git` to install git, and `brew install cmake` to get cmake; both may be needed below.)

If you are not a MacPorts or Homebrew user, you can install Qt using their “Online Installer”. Start by surfing to <https://www.qt.io/download-open-source>. Scroll down past all the verbosity about their different license types; you want to download the free, open source license version of Qt. At the bottom of the page is a large green button labeled “Download the Qt Online Installer”. Click that and you will land on a page with a large green button that says “Download”. Click that and the installer will download. Double-click the downloaded `.dmg` file in Finder, run the installer it contains, and follow its instructions. It will likely make you sign up for a “Qt Account”; this is free, albeit somewhat annoying. (If you install with MacPorts or Homebrew you can avoid this step.) It defaults to installing in a directory named Qt inside your home directory; that is fine. After you choose the destination, the installer will give you a choice of versions to install. As discussed above, 5.15.2 is the strongly recommended choice for SLiMgui on macOS. The installer defaults to installing Qt Creator for you, which is harmless and might prove useful; it will be located inside the install folder – at `/Users/bhaller/Qt/Qt Creator.app`, for me.

**Next, confirm your installed Qt version.** Try executing `qmake --version` in Terminal:

```
bhaller@lanois ~ % qmake --version
zsh: command not found: qmake
```

If you installed with MacPorts or Homebrew following the above procedures, the correct path *might* already be set up; having installed with the Online Installer, it is not, so you will get an error, as above. If you get an error, or if the version of Qt that `qmake` reports is not what you expected (i.e., it is finding some other installation of Qt), see the troubleshooting guide in section 2.4.3.

**Next, build SLiMgui.** We will use the `qmake` tool to do so; it is also possible to use `cmake`, as in the Linux build instructions in the next section, but on macOS `qmake` currently produces a better build of the app (as a “bundle”, properly formed for macOS). If you experience a problem with `qmake`, you can try following the Linux build instructions below to use `cmake`, starting with setting up the `Qt5_DIR` environment variable as described in section 2.4.3, but it is not recommended.

Here, we’re proceeding with `qmake`, then. As with building the `slim` and `eidoss` command-line tools, we want to use a build directory that is parallel to SLiM’s source directory – and let’s use a different one than we perhaps used for building `slim` and `eidoss`, since we built those with `cmake`. So execute the following:

```
cd SLiM
cd ..
mkdir SLiMgui_QMAKE
cd SLiMgui_QMAKE
qmake ../SLiM
make
```

The first line, `cd SLiM`, should be a `cd` to the SLiM source code directory that you previously obtained from the Messer Lab’s SLiM home page or from GitHub (see section 2.1 and 2.2). Then you `cd` to the directory above that, and make a build directory named `SLiMgui_QMAKE` parallel to



the source code directory (the name of the build directory is arbitrary). Do a `cd` into the build directory, execute `qmake` providing the path to the source code directory (and supplying the full path to `qmake` if you did not set it up to be in your path). Finally, execute `make` to do the build.

If all that went without a hitch, after a minute or two the build should be complete. The SLiMgui application should be located at `./QtSLiM/SLiMgui.app`; you can find it in Finder and double-click it, or launch it from Terminal with `./QtSLiM/SLiMgui.app/Contents/MacOS/SLiMgui`.

**As a final step, install SLiMgui** in the standard folder `/Applications` rather than having it just live in its build directory. This will help macOS find it – for Launchpad, for opening `.slim` model files automatically when they are double-clicked, and for other such purposes. After `make` finishes, you can simply execute the command `sudo make install`, which should install it for you after you type in your administrator password. Alternatively, you can copy `SLiMgui.app` to `/Applications` in the Finder, if you prefer. After installing it, you might want to drag SLiMgui from `/Applications` into your Dock for easy access.

#### 2.4.2 Building SLiMgui on Linux / Unix

First of all, please note that for end users of SLiM **there is often no need to build SLiMgui on Linux**; standard installers for SLiM are available on several Linux platforms. If you can use an installer (see section 2.2.2), you should then be all set; you do not need to read this section.

Building SLiMgui with `cmake` on Linux is fairly straightforward, but is not enabled by default since we don't want to cause trouble with builds of SLiM on computing clusters and other platforms where SLiMgui is not needed (and might fail to build). Before getting into the details, let me first say up front: I am not an experienced user of Linux, I don't know anything about different Linux distros, I don't know anything about software installation on Linux, nor am I particularly good with obscure platform-dependent compiler/linker/toolchain/makefile issues. The instructions below were puzzled out with some effort, but may be wrong on some distros/versions. If you run into a problem with these instructions, and you figure out a solution, *please let me know* so that I can promulgate that information.

**The first step is to check your existing Qt version.** On some Linux platforms Qt may already be preinstalled. You can check the version on your system by executing `qmake --version`. On Ubuntu 18.04 LTS and 20.04 LTS, however, `qmake` is not present, because Qt is not preinstalled, and so that command will likely give an error of some kind – indeed, it may urge you to install the `qtchooser` package, but that is *not* recommended (see section 2.4.3 for discussion). If you *do* have Qt already installed, you might see something like this:

```
bhaller@darwin-ubuntu:~$ qmake --version
QMake version 3.1
Using Qt version 5.9.5 in /usr/lib/x86_64-linux-gnu
```

That indicates that Qt 5.9.5 is already present on the system.

What version of Qt 5 is best for SLiMgui? Officially, we support Qt 5.9.5 and later (that is the earliest version we test on). Qt, like Ubuntu, designates some versions as “LTS” (Long-Term Support); these are considered the most stable versions, and receive bug fixes and patches for the longest time. As of this writing, the most recent LTS version of Qt 5 is 5.15.2, and it works well for me. I have not noticed any important issues with any Qt 5 version from 5.9.5 forward, though, so in general it is probably fine to go with whatever the default version is for the installer that you choose to use (see below). They do fix lots of bugs in each Qt release, though, so a more recent version might be better, especially if you experience problems with an older version. Note that Qt 6 has been released, but SLiMgui has not been ported to it; Qt 5 is required.

**The next step is to install Qt** (assuming whatever may be preinstalled is insufficient). There are several possible ways to do this, depending upon the distro you're on. In Debian/Ubuntu, do:

```
sudo apt-get install qt5-default
```

On Ubuntu 18.04 LTS this installs Qt 5.9.5, which is the earliest version of Qt we support; on Ubuntu 20.04 LTS it gives you Qt 5.12.8, at present. On Debian stable it currently yields Qt 5.11.3; you could use the “unstable” version, which gets you Qt 5.12.5 at present, or the “experimental” version, which currently gives Qt 5.14.2, but probably the “stable” default of Qt 5.11.3 on Debian will be good unless you encounter problems. Of course the version provided by the default install will change over time, so the versions mentioned here are just for reference.

On Arch, you probably just want to use the SLiMgui installer created by Graham Gower (see above), which will pull in Qt as a dependency automatically, but if you do want to install Qt yourself for some reason, you can run:

```
pacman -S qt5-base
```

If those installers don't work well for you, or if you want more control over which version of Qt you're getting, where it goes, etc., you can use the Qt “Online Installer”. To do that, here is a link about installing Qt on Ubuntu: [https://wiki.qt.io/Install\\_Qt\\_5\\_on\\_Ubuntu](https://wiki.qt.io/Install_Qt_5_on_Ubuntu). This is a much more complicated route, however, and is not recommended. To run the online installer you will need to create a “Qt Account” (free, but annoying), and things like the path to `qmake` and the Qt configuration for `cmake` will not be set up automatically. Fixes for those issues will be covered in the troubleshooting guide in section 2.4.3, but if you use the standard distro installer instead, you probably won't have to deal with them in the first place.

**Next, confirm your installed Qt version.** Try executing `qmake --version` again:

```
bhaller@darwin-ubuntu:~$ qmake --version
QMake version 3.1
Using Qt version 5.9.5 in /usr/lib/x86_64-linux-gnu
```

Here we see that Qt version 5.9.5 is installed (because here we installed with `apt-get` on Ubuntu 18.04 LTS). If Qt was preinstalled, your system might still find the old preinstalled `qmake`; or if Qt was not preinstalled, your system might be unable to find the newly installed `qmake`, and so you might get an error. In those cases, see the troubleshooting guide in section 2.4.3.

**Next, build SLiMgui.** Now that you have a good version of Qt installed, the rest is (maybe) easy. You will use `cmake` as described in section 2.2, but with an additional flag that tells `cmake` to include SLiMgui as a target (in addition to the `eidos` and `slim` targets, which will build as usual). If you have already built `slim` and `eidos`, you can use the same build directory to build SLiMgui; otherwise you will need to make a new build directory:

```
cd SLiM
cd ..
mkdir build
cd build
cmake -D BUILD_SLIMGUI=ON ../SLiM
make SLiMgui
```

The `-D BUILD_SLIMGUI=ON` flag passed to `cmake` tells it to include SLiMgui in its list of targets, since it does not do so by default. The final line, `make SLiMgui`, builds only the SLiMgui target; if you want to build `slim` and `eidos` as well, you can simply do `make`.

If you get an error at this stage that `cmake` can't find necessary configuration information for Qt, you will need to help it find Qt with `Qt5_DIR` or `CMAKE_PREFIX_PATH`; see section 2.4.3.



Once the build is finished, you should have a built SLiMgui app in your build directory that you can launch with the command:

```
./SLiMgui
```

As usual you can copy/move the executable to a standard location if you wish; the command `make install` might do this automatically, defaulting to the standard location `/usr/local/bin`. Alternatively, you could add the directory where SLiMgui lives to your `PATH` environment variable if you wish to execute it from there.

If you find that SLiMgui won't run, particularly after you do `make install` or otherwise copy/move it, with an error like "error while loading shared libraries: libQt5Widgets.so.5: cannot open shared object file", you may need to give the loader a path to your Qt libraries; see the troubleshooting guide in section 2.4.3.

Finally, if you open the About panel in SLiMgui, in the lower left it will show the version of Qt against which it was built. If the version you see there is not the version of Qt you installed and intended to use, or if SLiMgui gives you an error on launch that the run-time Qt version does not match the build-time Qt version, you will probably need to set up the `Qt5_DIR` environment variable after all; again, see section 2.4.3.

Before we finish, it is worth mentioning that there are two alternative ways to build SLiMgui. The first way is to use `qmake` instead of `cmake`; because SLiMgui is based on Qt, the `qmake` build tool provided with Qt works with it. You can do this at the command line in much the same way as with `cmake`; `mkdir` and `cd` to a parallel build directory as above, execute `qmake ../SLiM`, and then execute `make`. The `qmake` configuration is only set up to build SLiMgui; it will not build `eidos` or `slim`. With `qmake`, you can find the built executable at `./QtSLiM/SLiMgui` if you don't use `make install` to install it elsewhere (`/usr/local/bin` by default). The macOS installation instructions in the previous subsection use `qmake`, so you might refer to them. If you have trouble getting the `cmake` build of SLiMgui to work, try using `qmake` instead; it often smoothes out Qt-specific issues.

The second alternative way to build and run SLiMgui is with Qt Creator, the development environment for Qt; you can double-click the `SLiM.pro` project file at the top level of your SLiM local repository to launch Qt Creator (or, if double-clicking the file doesn't work, launch Qt Creator and then open the `SLiM.pro` file from within the app). The main reason that you might want to use Qt Creator is that it provides a debugger, which might be useful to pin down a crash or other problem. However, this is really only useful for experienced programmers; *caveat lector*.

OK, now you have SLiMgui! You may notice that SLiMgui does not have an application icon until it is launched, and that it does not appear in the application launcher, and that the `.slim` file extension is not registered to SLiMgui (and so double-clicking `.slim` files in your Linux windowing system doesn't launch SLiMgui and open the file), and so forth. These issues are external to SLiMgui itself; they have to do with how SLiMgui is registered as an application in the desktop environment. The standard SLiM installers for most Linux platforms perform this "desktop integration" for you, but if you build SLiMgui yourself, as we have done here, this desktop integration remains to be done. If you want to do it, you can find the appropriate commands in the installer script for your platform (see section 2.2.2), and try running them for yourself. However, there is no harm to *not* doing those steps, if you don't mind the lack of integration with the desktop environment.

When you run the app, if things seem poorly scaled (small buttons, tiny text), or if you just want things to be larger for readability on a screen with high pixel density, see section 2.4.3. Also, if you get square "missing symbol" icons where little pythons should be next to Python recipes under the File menu, see section 2.4.3.

### 2.4.3 Troubleshooting SLiMgui build issues on Linux / Unix

In this subsection I'll try to anticipate some of the most common difficulties with getting SLiMgui to build on Linux. See section 2.7 for runtime issues after you have built SLiMgui.

#### The qmake command can't be found

If you execute `qmake --version` and get some kind of error that `qmake` can't be found, you'll want to track down the location of `qmake`. (If you're trying to use `cmake` to build, finding `qmake` is not strictly essential, but it helps you to find where your whole Qt installation is located, and to check what version of Qt you installed, so it's best to find it if you can.)

Since we are juggling a variety of distros and installation methods with these instructions, it could be in several different places. The simplest thing is to try these commands:

```
find /usr -name qmake -print
find /opt -name qmake -print
find ~ -name qmake -print
```

If there's more than one and you can't tell which one is the one you just installed, you can run each one with `--version` to find out its version. Having installed Qt with the Online Installer (which is a common reason why it can't be found), I find that `qmake` is at `/home/bhaller/Qt/5.14.2/gcc_64/bin/qmake`. You will want to use the `qmake` that is under a directory named something like `gcc_64` or `clang`, not other `qmake` installs that might be present under platform-specific directories named `android`, `wasm_32`, etc., or under directories like `Src`, `Examples`, or `Docs`.

Having located the right `qmake`, we now want to help the system automatically find it. The simple way is to make a symlink:

```
sudo ln -s /home/bhaller/Qt/5.14.2/gcc_64/bin/qmake /usr/local/bin/qmake
```

Use the correct path for `qmake` on your system, of course.

Alternatively, on Linux there is a tool called `qtchooser` is responsible for finding `qmake` and other Qt tools. It is most useful if you want to have multiple Qt versions installed on your machine, and switch between them. I'm not going to get into how to set it up and use it (which is a bit technical); if you wish to use it, help is available online.

#### Tools like gcc, g++, cmake, git, etc. can't be found, OR cmake can't find OpenGL

If you experience problems that seem to be due to uninstalled tools or software, here are some standard packages that you can try installing. They will be installed automatically for many installation strategies, but might need to be manually installed in some cases, especially if you use the Qt Online Installer. Try these commands:

```
sudo apt-get install build-essential
sudo apt-get install libfontconfig1
sudo apt-get install mesa-common-dev
sudo apt-get install libglu1-mesa-dev
sudo apt-get install cmake
```

If you still get an error from `cmake` that it can't find OpenGL, try this:

```
sudo apt-get install libgl1-mesa-dev
```

You might also wish to install these additional packages, although they're not required:

```
sudo apt-get install git
sudo apt-get install r-base-core
```

## The cmake build fails because it can't find Qt, or it builds with the wrong Qt version

If cmake builds against the wrong version of Qt, or if it gives an error like:

```
Could not find a package configuration file provided by "Qt5" with any of
the following names:
```

```
Qt5Config.cmake
qt5-config.cmake
```

then cmake needs help finding your Qt installation. The cmake build tool refers to an environment variable named Qt5\_DIR to find the various components of Qt that it needs. This seems to be automatically set up by some installation procedures (or perhaps cmake is smart enough to find Qt automatically when it is in certain standard install locations), but in other cases Qt5\_DIR needs to be set up in order for cmake to work. The simplest thing is to just set it in Terminal, like:

```
export Qt5_DIR=/home/bhaller/Qt/5.14.2/gcc_64
```

Use the corresponding path for your installation; again, the path I have given above is for an Online Installer installation of Qt 5.14.2 in my home directory. Note that the path needed for Qt5\_DIR ends at the gcc\_64 directory under the Qt install directory.

If you want this definition of Qt5\_DIR to be more global and permanent, so you don't need to keep remembering to set it, you could put it in a startup script like ~/.bashrc.

Another option that also seems to work is to help cmake find Qt by set CMAKE\_PREFIX\_PATH, a define that cmake uses to find libraries and such. Sticking with the example above of Qt 5.14.2 installed in my home directory by the Online Installer, I could build using cmake with:

```
cmake -D CMAKE_PREFIX_PATH=/home/bhaller/Qt/5.14.2/gcc_64
-D BUILD_SLIMGUI=ON ../SLiM
```

That's all one line in the terminal. The -D CMAKE\_PREFIX\_PATH=... define tells cmake where to look for Qt and other libraries; the rest is the standard cmake build line for building SLiMgui. Again, note that the path provided should be to the gcc\_64 directory inside the directory for the Qt version you want cmake to use.

### Other issues

For other build issues, you might try building with qmake instead of cmake (see the end of section 2.4.2). For everything else, perhaps Google will help; please let us know any useful tips.

For post-install issues (when actually running SLiMgui), see section 2.7.

#### 2.4.4 Building SLiMgui on Windows

Section 2.3.2 provides instructions for building SLiM on Windows. Those instructions do not build SLiMgui, since it involves a few additional complications; that topic is covered here. See section 2.3.2 for the base instructions, including installation of the MSYS2 build tools. It is recommended that you successfully build SLiM before attempting to build SLiMgui, so that if problems occur it is easier to debug them.

To build SLiMgui, you will first need to install the Qt 5 library, prior to running cmake. You can install it with MSYS2 with the command:

```
pacman -S mingw-w64-x86_64-qt5-base
```

With Qt 5 successfully installed, you can then follow the instructions in section 2.3.2, but substitute the following command line in place of the cmake command line there:

```
cmake -G"MSYS Makefiles" -DBUILD_SLIMGUI=ON ..
```

If you encounter problems, please see the next section.

#### 2.4.5 Troubleshooting SLiMgui build issues on Windows

When you try to run the SLiMgui.exe file, you may get an error about “missing DLLs”. If so, you need to add the MSYS2 bin directory to your PATH environment variable so that the Qt 5 DLLs can be found. To do this, just click the search box in the Start menu and begin typing environment, then click on “Edit the system environment variables” when it appears. In the subsequent dialog box, click the “Environment Variables” button in the lower right corner. In the “System variables” pane, find the entry for “PATH” (it may be spelled “Path” or “path” also; it is not case-sensitive). Double-click on that entry, and in the next dialog box, click “New”. Now type the MSYS2 bin path and click OK when finished. As discussed in section 2.3.1, the MSYS2 bin path is C:\msys64\mingw64\bin for a typical installation of MSYS2 at C:\msys64, but if you chose a different installation directory the bin path will be correspondingly different. After fixing PATH, when you try to run SLiMgui.exe again it should work.

You might also get a more esoteric error message along the lines of “The procedure entry point \_Z20qResourceFeatureZlibv could not be located in the dynamic link library ... \SLiMgui.exe”. This error message means that Windows is using the wrong version of Qt 5, perhaps from a previous installation. You *could* make sure Windows is using the version that has been installed into your MSYS2 bin directory by putting it higher on your PATH environment variable, before any other Qt 5 installations; unfortunately, this could break other software that relies upon a different Qt 5 version. Happily, there is a simple solution that should always work and won’t break any other software: copy the Qt 5 DLLs (prefixed with Qt5, e.g., Qt5Core.dll) from the MSYS2 bin directory, where they should have been installed by pacman, into the same directory as your SLiMgui.exe. Windows will look there before it looks in directories specified in the PATH variable, so SLiMgui should then run without any change to PATH.

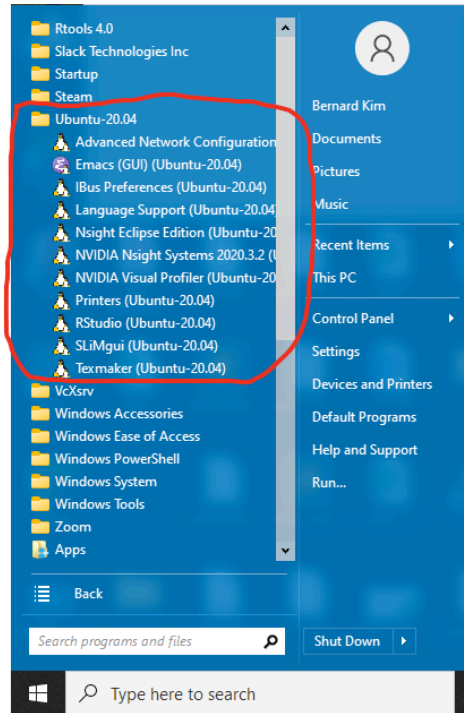
For other post-install issues (when actually running SLiMgui), see section 2.7.

#### 2.4.6 Building SLiMgui on Windows with WSL

SLiM now builds and runs natively for Windows; see section 2.3 for details. Alternatively, it is also possible to run SLiMgui on Windows using the “Windows Subsystem for Linux” (WSL), which – despite its name – is essentially a Linux subsystem for Windows. These instructions for building and running SLiMgui under the WSL come from Bernard Kim (thanks Bernard!); I have not tested them at all, since I am not on Windows. If you have any improvements for these instructions, please let us know (thanks to Chaz Hyseni for a few additions).

There are two ways to approach running SLiMgui under the WSL. The **first way** is to use Microsoft’s new GPU-accelerated WSL 2, called the WSLg. At the time of this writing, the WSLg is currently in “preview”, so this is prerelease software. Also, this method of running SLiMgui has, thus far, only been demonstrated to work with Ubuntu 20.04 LTS; Ubuntu 18.04 produced only a black window, and other Linux distros have not been tried. To use the WSLg you will have to update your Windows to the latest version on the developer channel, which can introduce instability to your system and is irreversible unless you do a fresh installation of Windows. Installation instructions for the WSLg are at <https://github.com/microsoft/wslg>. Once the WSLg is installed, you may be able to simply install SLiMgui using the standard Linux installer for the Linux version you have installed inside the WSLg; see section 2.4.2 for the standard Linux installation instructions. If the installer doesn’t work for you, you can follow Bernard’s build instructions below, but *ignore everything having to do with VcXsrv or X11* – skip steps 4, 7, 8, 9, and 10.

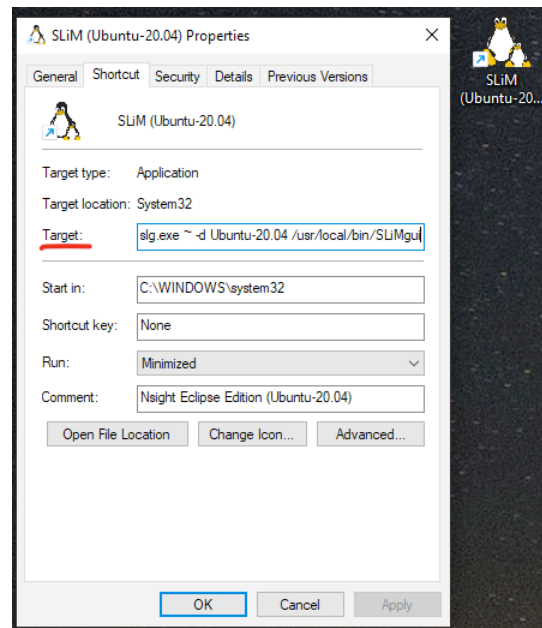
Whichever way you get SLiMgui built and installed to /usr/bin, it should probably then appear under the Windows Start menu, as shown here:



If that works, then to run SLiMgui, you can simply click on the icon there; you do not need to set up a Windows X server, or worry about any of the related X server configuration steps, or set up a batch script for launching as described below. You are done.

If SLiMgui does *not* appear in the Start menu (apparently this sometimes happens, for reasons unknown), you should still be able to run it in Linux, like any other binary, and again there should be no need to worry about X server config or batch scripts.

Having gotten SLiMgui to run under the WSLg, you should be able to set up a Windows “shortcut” to it if you wish. You can initiate this by right-clicking on the desktop (or wherever you want the shortcut to reside) and choosing **New > Shortcut**. Configure it something like this:





For the target of the shortcut (highlighted red in the screenshot above), make sure that the latter half of the target's specification is consistent with your Linux and SLiMgui installations. For example, if you installed SLiMgui to `/usr/local/bin` with `sudo make install` and your WSLg installation is using Ubuntu 20.04, then the shortcut's target would be:

```
C:\Windows\System32\wslg.exe ~ -d Ubuntu-20.04 /usr/local/bin/SLiMgui
```

The **second way**, if you choose not to use the WSLg as described above, is to build SLiMgui under the WSL 1 or WSL 2 and run it under an X server, VcXsrv, that handles its windowing and graphics. You should be running a 64-bit version of Windows 10 with the Anniversary Update; the Creator's Update is recommended also. (I have no idea what these things are.)

There are ten steps:

**Step 1** is to set up the Windows Subsystem for Linux. Microsoft provides installation instructions; see <https://docs.microsoft.com/en-us/windows/wsl/install-win10>. **The workflow described here is for WSL 1.** WSL 2 is included in the Windows 10 version 2004 update, which was recently released (not, apparently, in 2004). Based on preliminary testing, changes to this workflow for WSL 2 are given at the very end, after Step 10.

**Step 2** is installing Ubuntu 20.04 LTS from the Windows Store. Other Linux distros might also work, but have not been tested with SLiMgui.

**Step 3** is optional: install Windows Terminal from the Windows Store for a nicer terminal interface. If you don't install Windows Terminal, just execute the bash command at the Windows command prompt to get to a Linux shell where you can enter the commands below.

**Step 4** is to set up VcXsrv. This is an X11 server for Windows – a windowing system server, allowing Linux apps that use X11 to run inside windows on Windows. Note that Xming, an alternative X11 server for Windows, seems to have problems with Qt apps that use OpenGL, such as SLiMgui, so it should not be used. VcXsrv is found at <https://sourceforge.net/projects/vcxsrv/>. Download the installer by clicking the big green button near the top of the page, then run it. More extensive instructions can be found with Google if you hit bumps, of course. Do not run VcXsrv yet; we will configure it below before running it.

**Step 5** is to install the Linux dependencies you will need. Bernard recommends this:

```
sudo apt-get update
sudo apt-get install -y cmake gcc g++ qt5-default mesa-utils
sudo apt-get install -y libgl1-mesa-glx x11-apps unzip wget
```

It may be that not all of these packages are necessary, but they are what Bernard had installed when he got all this to work, and if some are extra, it should be harmless.

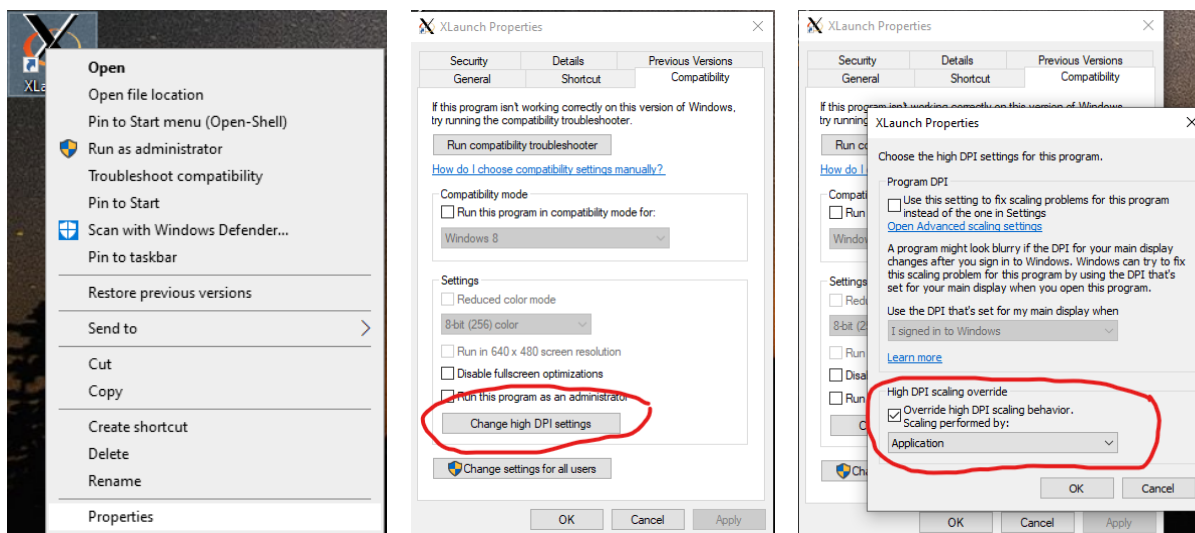
**Step 6** is to use an installer to install SLiM (see section 2.2.2) – which may or may not work, the installers have not been tested under the WSL – or to download and build SLiMgui following the instructions in section 2.4.2, which can be summarized as:

```
wget http://benhaller.com/slim/SLiM.zip
unzip SLiM.zip
cd SLiM
cd ..
mkdir build
cd build
cmake -D BUILD_SLIMGUI=ON ../SLiM
make SLiMgui
```

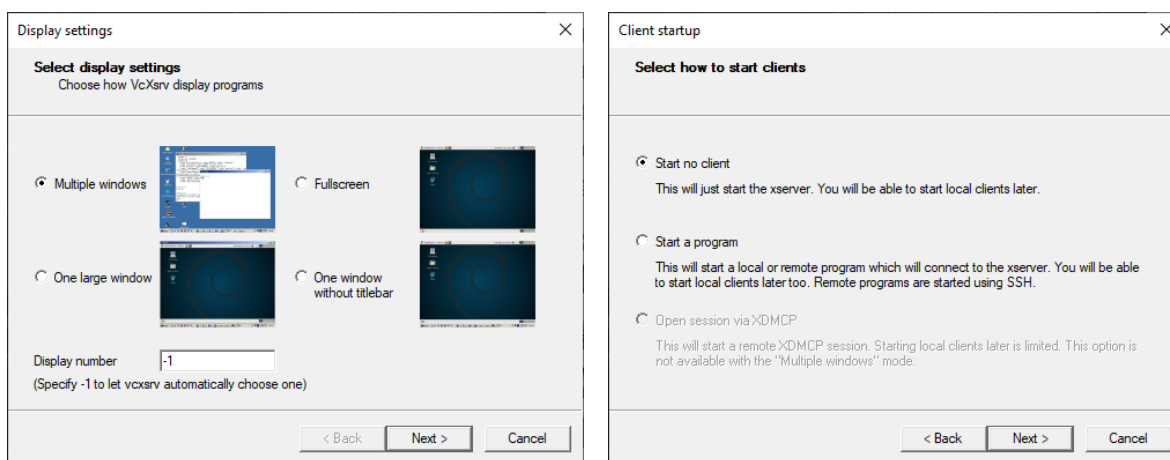
You can just execute `make` as the last step, instead of `make SLiMgui`, if you also want to build the `slim` and `eidos` command-line tools.

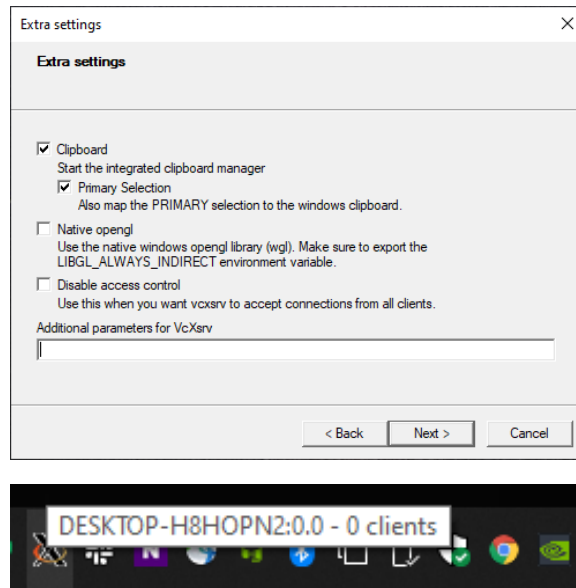


**Step 7**, to be done **before** running VcXsrv, is to fix an issue with blurry fonts on high-resolution monitors. Right-click on the VcXsrv icon (named “XLaunch”), go to the “Compatibility” tab, and click the “Change high DPI settings” button. In the resulting panel, enable “High DPI scaling behavior” and change the “Scaling performed by” popup to “Application” or “System Enhanced”. Here are some screenshots of this step:



**Step 8** is to run VcXsrv in “Multiple windows” and “Start no client” mode, with “Native opengl” unchecked. Note the display server address here, or (as shown in the last screenshot below) by hovering over the VcXsrv icon in your taskbar. The display server address will be in the format `hostname:display.screen`. The number to note is the first number after `hostname:` – as long as this display server number is provided, SLiMgui should be able to connect to the X server. It is possible to have multiple display servers running, each with multiple screens. However, if VcXsrv is allowed to automatically set the display server number by using `-1` in the Display settings dialog box (as in the screenshot below), and if no additional instances of VcXsrv are running, it will default to one display server numbered `0` and one screen numbered `0`. In the last screenshot below, the end result of this automatic configuration is `DESKTOP-H8H0PN2:0.0`. Here are the screenshots for this step:





**Step 9** is to unset the environment variable `LIBGL_ALWAYS_INDIRECT` in the Linux terminal by entering:

```
export LIBGL_ALWAYS_INDIRECT=
```

This must always be done before running SLiMgui. The best solution to this is given in step 10, so rather than doing this now in your terminal, you can just proceed to step 10.

**Step 10**, finally, is to run SLiMgui. You can run it from the build directory at the Linux command line with `./SLiMgui`. Since `LIBGL_ALWAYS_INDIRECT` needs to be unset, and since it is best to feed the display server number to VcXsrv via an environment variable as well (to avoid issues with the display server not being found), the best solution is to make a Windows batch script that opens SLiMgui for you. To do this, open the Notepad app in Windows and type the following commands into a new empty window:

```
bash -c 'export LIBGL_ALWAYS_INDIRECT=; export DISPLAY=:0; ~/build/SLiMgui'
pause
```

The path to SLiMgui should be its path in the Linux filesystem (e.g., `~/build/SLiMgui` for the example given here), not the path within Windows (e.g., `C:\Users\Program Files`). Also note the display server number here; this should match the display server number configured for VcXsrv. The display server address we noted before for this example was `DESKTOP-H8HOPN2:0.0`, so we want to tell SLiMgui to connect to display server `0` by executing `export DISPLAY=:0`. It is typically unnecessary to provide the full address, but doing `export DISPLAY=:0.0` or `export DISPLAY=localhost:0.0` will also work (where `localhost` is `DESKTOP-H8HOPN2:0.0` in this example, but will be different for you). If you change the display server number when running VcXsrv (e.g., manually set it to 1), this file must be edited to match the new number.

Save this file as type “All Files” and name it `SLiMgui.bat`. As long as you have VcXsrv running, double-clicking this file will automatically run SLiMgui with the correct settings.

An error that might bite you at this stage, when you double-click the batch file to launch SLiMgui, is:

```
error while loading shared libraries: libQt5Core.so.5: cannot open shared
object file: No such file or directory
```

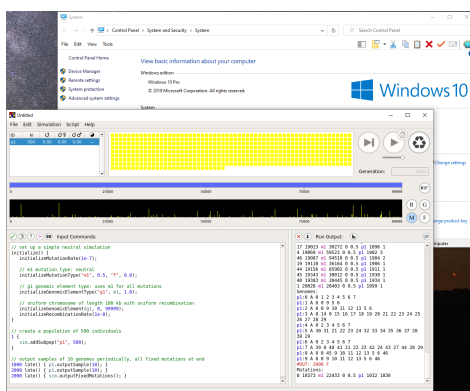
This appears to happen on some systems because WSL 1 is missing a Linux kernel call that Qt relies upon. You can work around it by stripping out the offending bit of the Qt core library, with one of the following two commands:

```
sudo strip --remove-section=.note.ABI-tag /usr/lib64/libQt5Core.so.5
sudo strip --remove-section=.note.ABI-tag /usr/lib/x86_64-linux-gnu/libQt5Core.so.5
```

Use whichever command corresponds to the path to `libQt5Core.so.5` on your system. I don't know why this works, but apparently it does. More information on this can be found at: <https://github.com/dnschneid/crouton/wiki/Fix-error-while-loading-shared-libraries:-libQt5Core.so.5>. This problem does not seem to occur for WSL 2.

If you have problems with the SLiMgui build in step 6, the troubleshooting tips in section 2.4.3 might be helpful; but then again they might not, since Windows with WSL is a different beast from a normal Ubuntu installation.

If you get it all working, here is the glorious end result – SLiMgui running on Windows:



These instructions will launch SLiMgui in a standalone window on your Windows desktop, as in the screenshot above. If you prefer to run it inside a Linux desktop environment (like running under a VM), or if the standalone window approach doesn't work for you, you might try using `xfce4` (<https://medium.com/@dhanar.santika/installing-wsl-with-gui-using-vcxsrv-6f307e96fac0>) or `Unity` (<https://www.zdnet.com/article/how-to-run-run-the-native-ubuntu-desktop-on-windows-10/>). However, this manual will not attempt to cover the details of those configurations. Chaz Hyseni, who supplied this information, says that `Unity` didn't work very well for him under WSL 1, but `xfce4` worked fine.

Further polish – installing SLiMgui in a standard location, making it findable by the standard Windows mechanisms for launching apps so that you don't have to launch it with a batch script, making `.slim` files show the correct icon and launch SLiMgui when they are double-clicked, etc. – may be possible, but these things remain a mystery at this time.

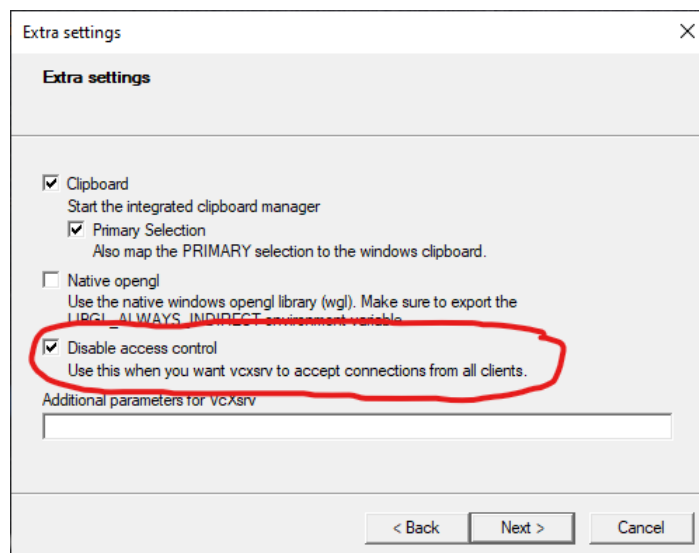
**Changes for WSL 2.** The instructions above are for WSL 1. If you install WSL 2 instead, a few alterations need to be made. SLiMgui seems to work well in both, but in WSL 2 disk access is *much* faster, so if you plan to generate large amounts of output WSL 2 might be preferable. However, due to differences in virtualization, Ubuntu in WSL 2 has its own IP address and does not share one with the machine, so connecting to the display server is a bit more complex, as we'll see below.

First of all, with WSL 2, every time Windows is rebooted one must run the Windows Powershell as administrator and run the following command to allow connections to VcXsrv from Ubuntu:

```
Set-NetFirewallProfile -DisabledInterfaceAliases 'vEthernet (WSL)'
```

This is not trivial to set up to run automatically on startup, since it has to run as administrator; one can hope that Microsoft will find a way to make it unnecessary in a future version.

Second, when you run VcXsrv (Step 8 above), in the “Extra settings” panel the “Disable access control” checkbox should be checked when using WSL 2, as in this screenshot:



The other settings in this panel remain the same. This stackoverflow thread discusses this setting: <https://stackoverflow.com/questions/61110603/how-to-set-up-working-x11-forwarding-on-wsl2>.

Third, at the Ubuntu command prompt the display address should now be set with the command:

```
export DISPLAY=$(awk '/nameserver / {print $2; exit}' /etc/resolv.conf 2>/dev/null):0
```

This looks up the correct IP address for the VcXsrv display server, which as mentioned earlier is different from hostname on WSL 2 because Linux gets its own IP address.

Note that the LIBGL\_ALWAYS\_INDIRECT environment variable still needs to be unset with:

```
export LIBGL_ALWAYS_INDIRECT=
```

As before (Step 10), these two commands can be placed in a Windows batch script, so that you can simply double-click the batch script to run SLiMgui with the correct configuration.

Note that if you install WSL 2, it actually allows you to choose whether to use WSL 1 or WSL 2 for your Linux installation(s); WSL 2, in a sense, contains both WSL 1 and WSL 2 within it (for now, at least). Given that you installed WSL 2, if you forget which WSL version you have chosen to use for a given Linux installation, you can always check by executing `wsl -l -v` at a Linux shell prompt. If you install WSL 2 but choose to actually use WSL 1 for your Linux installation, then you should follow the WSL 1 instructions, not the WSL 2 instructions. Could they make this any more confusing? Who's on first?

## 2.5 Installation with conda

An alternative to the aforementioned ways of installing SLiM (building from sources using `cmake`, or using the macOS double-click installer), beginning with SLiM 3.3.2, is to install SLiM using `conda`. `conda` is a package manager and environment manager that is available on many Unix and Linux platforms, including macOS. It is particularly useful for configuring self-contained *environments* within which particular versions of particular packages are installed, without

affecting the overall configuration of your machine. It is often used to set up a whole software stack in a standard and reproducible way, such as on a computing cluster. If you're already happy with how you build/install SLiM, you can ignore this section, but if you use conda this information may be of interest. Note that the conda installer does not install SLiMgui.

To install SLiM with conda, assuming you already have conda set up and you are already within the conda environment that you want to work in, just do:

```
conda install -c conda-forge slim
```

This means “install the package named `slim` from the `conda-forge` channel” (which is where SLiM happens to live). If you're an experienced conda user, you may of course want to do some variant of this. If you don't yet have a conda environment set up and activated, you might want to do something like this:

```
conda create -n test-slim
source activate test-slim
conda install -c conda-forge slim
```

The first command creates a new environment named `test-slim`, which we activate in the second line. We are now working within a self-contained, quarantined environment, and software that we install here will only be available when this environment is activated; it will not be installed on the machine in the usual way, only within `test-slim`. Once you have done this, you can check that the installation worked, as described in the next subsection. The conda-built version of SLiM ought to be functionally identical to SLiM built from sources yourself, with the same performance; there's nothing magic going on here.

When you are finished, executing:

```
source deactivate
```

will then take you out of the `test-slim` environment, which makes the `slim` command unavailable (or reverts you to whatever version of `slim` is installed on your machine outside of conda), until you activate the environment again.

For more information on conda, a good place to start might be the [anaconda installation documentation](https://docs.anaconda.com/anaconda/install/) at <https://docs.anaconda.com/anaconda/install/>. Let us know if you have any issues – but please, don't contact us with conda technical support questions unless they are specifically related to SLiM's conda configuration. Thanks to Ian Caldas for testing, and for help with this documentation; and a big thank-you to Jerome Kelleher for making this happen, and to Kevin Thornton and Peter Ralph for contributions as well.

## 2.6 Testing the SLiM installation

Regardless of your platform or method of installation, it is a good idea to run some self-tests after installing SLiM. In your Terminal window, Unix shell window, or platform equivalent, run the following two commands (while in the build directory where the `slim` executable resides):

```
./slim -testEidos
./slim -testSLiM
```

Each command should print a result line beginning with `SUCCESS` and then a count of successful self-tests. If any other lines print, indicating failure of a test, you should probably contact us to ask how to proceed; but see section 2.7, below, for a couple of exceptions to that.

## 2.7 Post-installation issues

This section is about problems that occur once you have successfully built/installed SLiM and SLiMgui, when you try to run them. See sections 2.4.3 and 2.4.5 for other issues that might arise; the dividing line between this section and those sections is not entirely clear. If you figure out fixes for other issues that should be documented here, please let us know.

### **This build of Eidos does not detect integer arithmetic overflows**

You may see a warning message when you run the Eidos self-tests, something like:

```
WARNING: This build of Eidos does not detect integer arithmetic overflows.
Compiling Eidos with GCC version 5.0 or later, or Clang version 3.9 or
later, is required for this feature. This means that integer addition,
subtraction, or multiplication that overflows the 64-bit range of Eidos
(-9223372036854775808 to 9223372036854775807) will not be detected.
```

The error message says pretty much what the problem is: the compiler you are using is too old to support integer arithmetic overflows (which an operation on integer values produces a result that is outside of the representable range for integer values). Note that this means you are using a very old compiler; but some computing clusters, in particular, use very old compilers by default because they don't want to break old software. Your system administrator, if you have one, should be able to tell you how to switch to building SLiM with a more modern toolchain.

However, in practice this warning is unlikely to cause you real problems. The representable range for 64-bit integers, such as Eidos uses, is extremely wide (as stated in the error message). Unless you expect that your code might actually trigger such overflows somehow, you might get away with ignoring this message.

### **This build of Eidos does not have a working <regex> library**

You may see a warning message when you run the Eidos self-tests, something like:

```
WARNING: This build of Eidos does not have a working <regex> library, due
to a bug in the underlying C++ standard library provided by the system.
This may cause problems with the Eidos functions grep() and readCSV(); if
you do not use those functions, it should not affect you. If a case where
a problem does occur is encountered, an error will result. This problem
might be resolved by updating your compiler or toolchain, or by upgrading
to a more recent version of your operating system.
```

As with the previous warning, this likely indicates that you are building with a very old compiler or toolchain. If you don't use the regular-expression facilities provided by Eidos, this may not affect you at all; if you get runtime errors when you try to do things with regular expressions, you will need to update your toolchain.

### **This system does not appear to have a writeable temporary directory**

You may see a warning message when you run the Eidos or SLiM self-tests, something like:

```
WARNING: This system does not appear to have a writeable temporary
directory. Filesystem tests are disabled, and functions such as
writeTempFile() and system() that depend upon the existence of the
temporary directory will raise an exception if called (and are therefore
also not tested). Other self-tests that rely on writing temporary files,
such as of readCSV() and Image, will also be disabled. If this is
surprising, contact the system administrator for details.
```

This problem generally comes up on computing clusters. Normally, the operating system provides a directory such as /tmp on Linux where temporary files can be stored. Some computing



clusters do not provide this, because there is little or no local storage on the individual cluster nodes. This is not, in itself, a problem for Eidos or SLiM; but as the warning message says, some of the Eidos and SLiM self-tests will not be run because they need a temporary directory to write to. If you have no other reason to believe that there is a problem, you can *probably* ignore this warning, but be aware that some components of Eidos/SLiM have not been tested and might have undiscovered problems. The lack of a temporary directory will also prevent you from using some Eidos/SLiM features, particularly the `tempdir()` and `writeTempFile()` functions. If you want to resolve this problem, try asking your system administrator for guidance.

### **The built app won't launch because of shared library issues**

This issue was written up for Linux, but may also be relevant on Windows, and certainly for Windows under the WSL.

If SLiMgui builds successfully, but it won't launch because either (a) Qt-related shared object files can't be found, or (b) it says there is a mismatch between the run-time and compile-time Qt versions, you may need to help the system's loader find the Qt libraries. To do this, add the appropriate path to the `LD_LIBRARY_PATH` environment variable:

```
export LD_LIBRARY_PATH=/home/bhaller/Qt/5.14.2/gcc_64/lib:$LD_LIBRARY_PATH
```

Use the corresponding path for your installation; again, the path I have given above is for an Online Installer installation of Qt 5.14.2 in my home directory. Note that the path needed for `LD_LIBRARY_PATH` ends at the `gcc_64/lib` directory under the version-numbered directory inside the Qt install directory. You may wish to put this in a startup script like `~/ .bashrc`.

If you do this and you still get an error that the run-time Qt version does not match the compile-time Qt version, you probably need to set up the `Qt5_DIR` variable as explained above; your run-time version may now be correct, but you may be building against the wrong Qt version. Set up `Qt5_DIR` and then do a new, clean build of SLiMgui. “Missing symbol” boxes after the Python recipes in the File menu

### **Emojis are not displaying in SLiMgui**

SLiMgui uses emojis in a couple of places. For example, it shows the “snake” emoji, 🐍, after Python recipes in the Open Recipe menu. Also, multispecies models typically set up emojis as “avatars” for the species being simulated, which SLiMgui uses to represent the species in its user interface. If you notice “missing symbol” boxes where emojis ought to be, or if they don't display at all, you may need to install an emoji font that provides the “snake” emoji. In the Arch distro, at least, the relevant package is named `noto-fonts-emoji`. Ubuntu 18.04 LTS and 20.04 LTS seem to come with the necessary font preinstalled, but you could check that they are correctly installed on your system (see <https://fonts.google.com/noto>).

If you still have problems with emoji display in SLiMgui, Graham Gower recommends surfing to <https://gist.github.com/charveey/091b11ea554436d15c7ffcf01129a8a> and copying the file there named `75-noto-color-emoji.conf` into `/etc/fonts/conf.d/`. Then run `fc-cache -f`, and then restart SLiMgui. Presumably more recent versions of Linux distros will fix these issues out of the box; but note that since new emojis are being added all the time, some emojis may display correctly while others do not. Furthermore, some emojis may display in SLiMgui in full color, while others display only as an outline. Numerous bugs in Qt seems to be entangled with all of this; there is not much SLiMgui can really do about it; the problem is really manifesting at lower software levels. For further discussion, see <https://github.com/MesserLab/SLiM/issues/306>.

### **If widgets or text are small, or the SLiMgui window layout is screwed up**

This issue was written up for Linux, but may also be relevant on Windows, and certainly for Windows under the WSL.

If you have a high-DPI screen that Qt doesn't recognize as high-DPI, SLiMgui's buttons and widget text may be at 50% size, with rather awful consequences. You may be able to fix this by launching SLiMgui from Terminal while setting two environment variables, like this:

```
QT_SCALE_FACTOR=2.0 QT_FONT_DPI=144 SLiMgui
```

These scale up the user interface and the fonts; users who simply want the interface to be larger may also find them useful. Depending on your system, `QT_SCALE_FACTOR=1.5` and/or `QT_FONT_DPI=128` might work better; experiment to find the optimum settings. If you decide that you want to make such a configuration option permanent, and have it apply when you launch the app from the desktop environment as well, you can set up the environment variable in the appropriate Linux configuration script. If you want to launch SLiMgui from the desktop environment, not from Terminal, then setting these variables up in `~/.bashrc` may not suffice, because the desktop environment may not use that file; you may need to put these settings in a special config file that your distro and desktop environment uses. For Ubuntu some advice is at <https://help.ubuntu.com/community/EnvironmentVariables>).

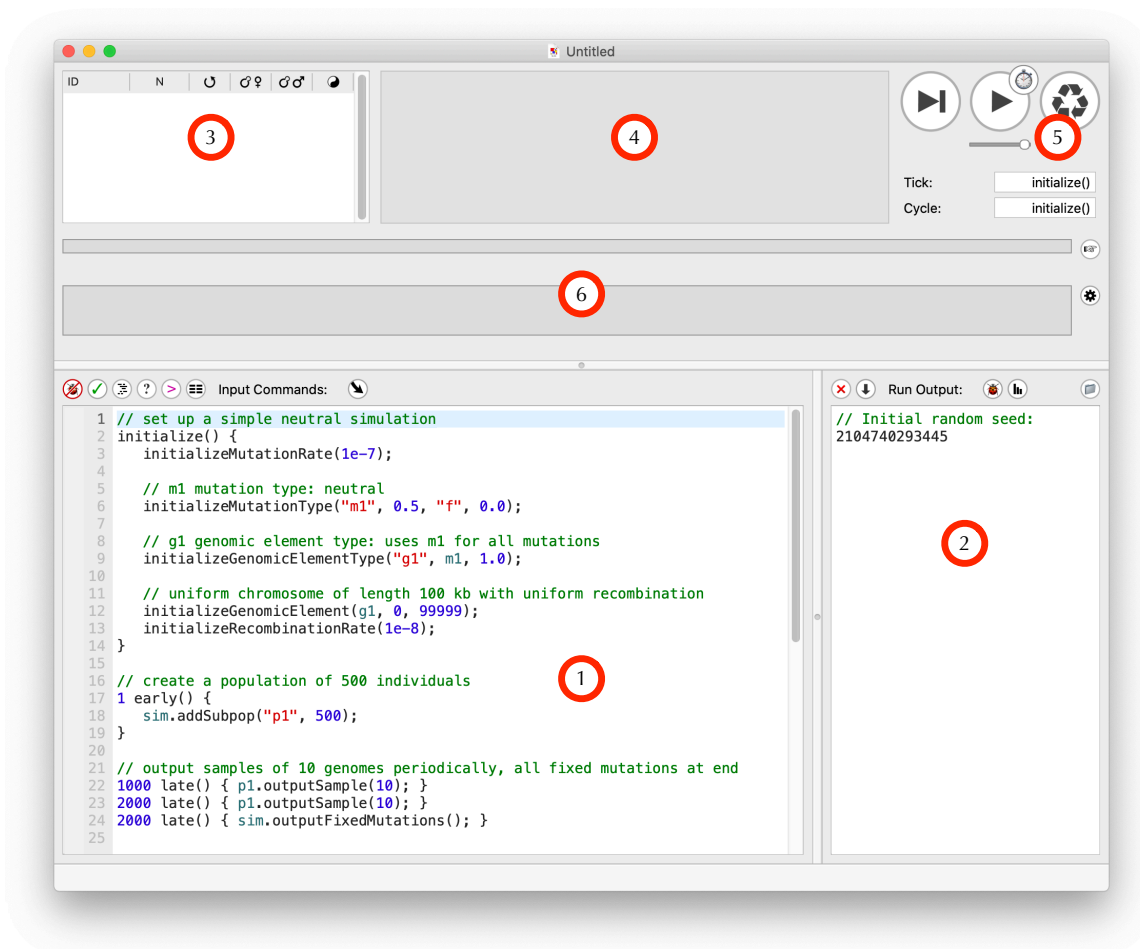
### 3 Running simulations in SLiMgui

This chapter introduces the SLiMgui graphical development environment for SLiM. The screenshots throughout this manual are taken in SLiMgui or SLiMguiLegacy on macOS; the appearance on Linux and Windows will of course be different in minor ways.

On macOS, SLiMgui is typically located in /Applications/ if you installed from the prebuilt package; if you built SLiM from sources, you may run SLiMguiLegacy directly from Xcode. On Linux and Windows, the location of SLiMgui depends upon how you built it; see chapter 2.


#### 3.1 The SLiMgui simulation window







When you first launch SLiMgui, you should see a window similar to this:








The main sections of the window are indicated with red numbered circles; they are:

**1. The scripting pane.** This is where input commands for SLiM – in the form of an Eidos script – are entered. A simple script is given by SLiMgui by default, as a starting point. At the top of this pane you can see the pane’s title, “Input Commands”, and to the left of that title are six buttons, and another one to the right of the title. All of the buttons in the window have tooltips, so just hover the mouse over them to be reminded of what they do. We will see some of these buttons again in later sections, but for now, in brief, here is what they do (left to right):

-  **Clear Debug Points:** This clears “debug points” that you can set in SLiMgui to obtain runtime debugging information about your model’s execution (see section 20.6).

-  **Check Script:** This checks your script for errors. Since your script is also checked when you restart (i.e., “recycle”) your model, checking your script with this button is usually unnecessary, but can occasionally be useful.
-  **Prettyprint Script:** This reformats your script by adding (or removing) tab characters at the start of each line to indent them nicely, showing the logical structure of your script. If you hold down the option (alt) key and click this button, it performs a more extensive reformatting, producing a script in a completely standard format.
-  **Script Help:** This brings up a documentation window that provides help on both Eidos and SLiM, including information on functions, classes, and methods, as well as Eidos operators and keywords, SLiM events and callbacks, and so forth.
-  **Show Eidos Console:** This shows a console window in which you can enter Eidos commands and get immediate, interactive output.
-  **Show Variable Browser:** This shows a browser window in which you can examine the values of all defined Eidos constants and variables, including elements of vectors and properties of objects. This can be useful for debugging.
-  **Jump:** This button pops up a menu that makes it easy to jump to different parts of your script – different SLiM events or callbacks that you have defined. This is very useful for navigating around in a large model script.


**2. The output pane.** This is where run output from SLiM – diagnostic output as well as output explicitly generated by your simulation script – is shown. At the top of this pane you can see the pane’s title, “Run Output”, surrounded by four buttons – and a fifth over at the right-hand edge. Again, we will see many of these more later, but for now (left to right):

-  **Clear Output:** This clears any output that has been dumped into the output pane, providing a clean slate for further output.
-  **Dump Population State:** This outputs the current population state to the output pane. This is essentially the same output as you would get from a call to the `outputFull()` method of `Species`, which we will see later on.
-  **Show Debugging Output:** This opens a window that shows debugging output, which SLiMgui keeps separately from the output of your model. Section 20.6 discusses debugging output in detail, including the use of “debug points” in SLiMgui.
-  **Show Graph:** This button pops up a menu, from which you can choose to display one of a variety of built-in graphs supported by SLiMgui.
-  **Change Working Directory:** This button changes the “working directory” used by SLiMgui. If your script reads or writes a file using a simple filename like `foo.txt` rather than a full file path like `/Users/bhaller/Desktop/foo.txt`, SLiM will use the current working directory as the location of the file. By default the working directory is the `~/Desktop` folder, if you have one. Note this can be changed with the `setwd()` function.


**3. The population view.** This is a table view that displays all of the subpopulations in the current simulation population. Since the simulation has not yet been started in the screenshot above, there are presently no subpopulations, and thus the table is empty. This table will be covered in more detail once we have a running simulation.

**4. The individual view.** This is where individual “organisms” in the simulation are displayed from the subpopulation(s) selected in the population view. At present this is empty since there are no individuals to display; the model has not yet started to execute.

**5. The tick controls.** The three large buttons are, from left to right:

-  **Step:** This runs one tick of the SLiM model and then stops.

- **▶ Play:** This runs SLiM model ticks until you click it again to stop execution.
- **♻️ Recycle:** This resets SLiM back to the initialization stage of the simulation. Recycling also clears the output pane, and re-parses the script in the script pane (which cannot happen while you're in the middle of executing a script). Whenever you change your script, you will need to recycle for your changes to take effect.


The slider below the Play button controls the speed at which the simulation runs; usually you will want this to be the maximum speed, but occasionally it is useful to slow the simulation down to better see what is happening on short timescales. The small stopwatch button , overlaying the Play button, is the **Profile** button, used to profile the speed and memory usage of a simulation (see sections 21.5 and 21.6). The **Tick** textfield shows the tick that SLiMgui is *about to execute*; if you press the Step button, the tick shown will execute. Similarly, the **Cycle** textfield shows the cycle that SLiM will next execute; unless you are doing multispecies modeling (see section 1.9 and chapter 19), this will be the same as the Tick textfield. At present, in the screenshot, the simulation is paused prior to the execution of `initialize()` callbacks, a special initialization stage that happens first before the simulation begins to run.

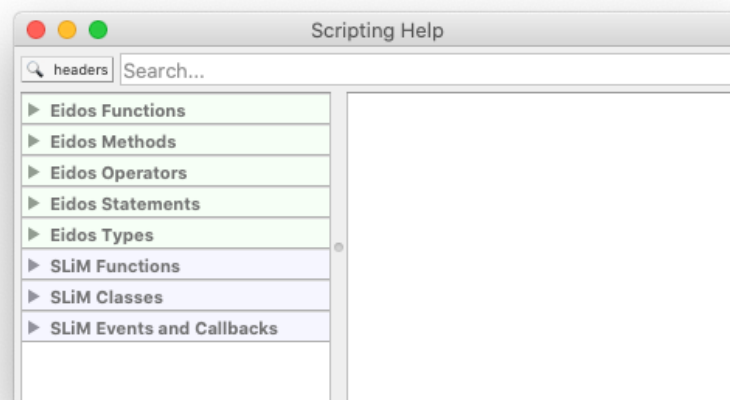
**6. The chromosome view.** The two wide stripes show views onto the simulated chromosome (empty at present since the simulation has not yet started); these will be discussed later. To the right of those views are two buttons:

- **👉 Show Object Tables:** This button, with the right-pointing hand icon, opens an auxiliary window that contains several tables of useful information about the running model.
- **⚙️ Chromosome View Actions:** This button, with the gear icon, pops up a menu with various configuration options for the chromosome view. This gear button is used in other places in SLiMgui's user interface too, always to indicate an "action button" providing commands and options that are specific to a particular part of the user interface.

This is a lot of information, and to avoid drowning in details we will now move on to other parts of SLiMgui's interface. We will see most of these buttons and commands again later, in applied contexts that will make their use more clear.

### 3.2 The script help window


Before moving on to writing your first Eidos script for SLiM, there are a few other components of SLiMgui that it might be useful to briefly mention. One such is the script help window. If you click the Script Help button , the script help window will open:

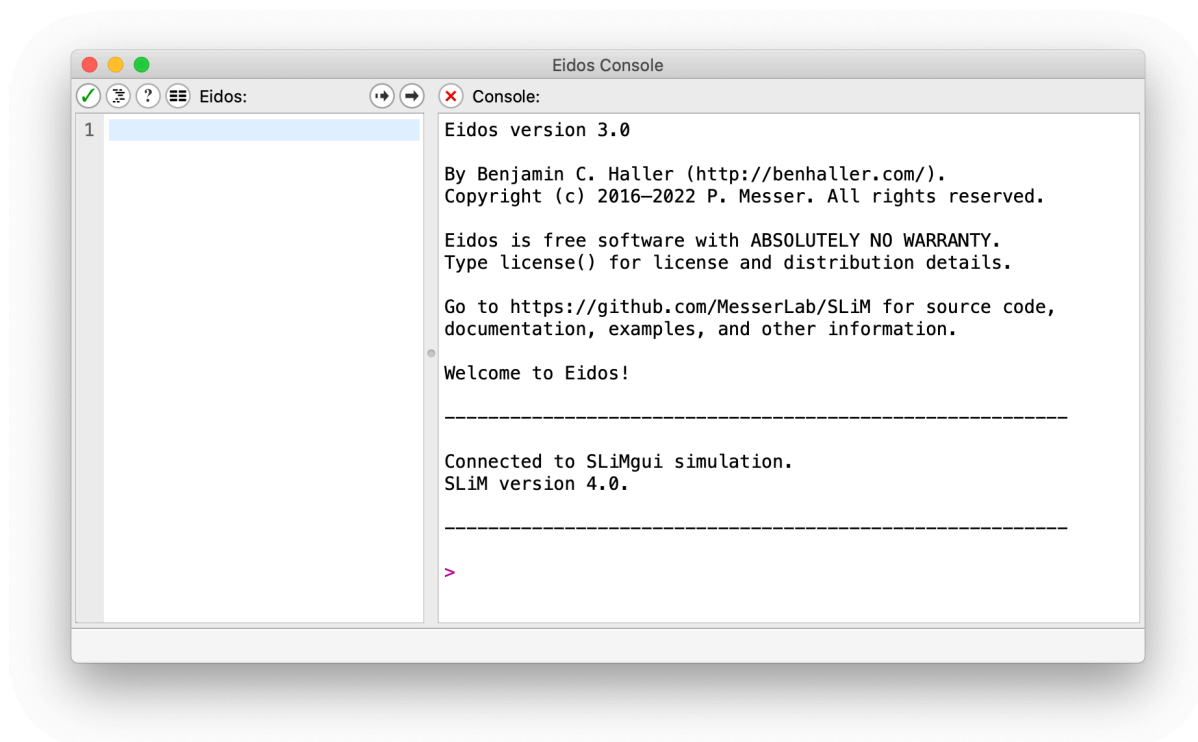





This window can be used to browse information about both Eidos and SLiM. A list of top-level topic headings appears on the left; you can click on these headings to expand them to show the sub-headings and sub-sub-headings they contain. When you click on a “leaf” – an actual topic – the information about that topic will be shown in the pane on the right. You can also use the search field at the top of the window to search on a given topic; the button to the left of the search field allows you to choose whether the search is done on topic headers only (for fewer and probably more relevant results), or is done on the full help text of each topic (for more comprehensive results).

A useful shortcut: an **option-click** (or alt-click, depending on your keyboard’s labels) on anything in the Eidos script of the simulation window’s scripting pane will pop up the script help window with search results for the item you option-clicked. This is very useful for quickly looking up functions, methods, language keywords, and other topics.

### 3.3 The Eidos console


Clicking the Show Eidos Console button  will show the console:

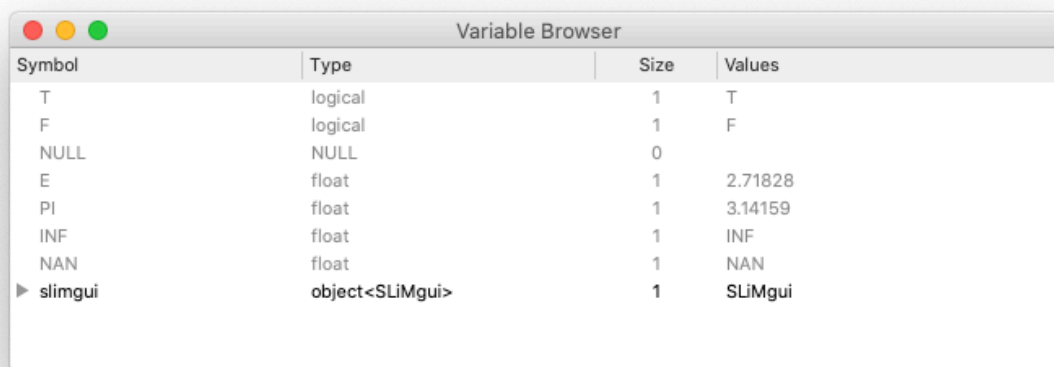


We won’t go into detail about this now, since we haven’t really discussed Eidos at all yet; but in essence, this console is a window where you can execute arbitrary Eidos commands at any point in your simulation. Commands are entered, and their output is shown, in the console pane on the right-hand side of the window, following the `>` prompt. You can clear the console pane’s previous output using the  button, just as in the main window. The left-hand side is a scratch area where you can work on longer blocks of script, using the arrow buttons at top to execute either the selected line only, with , or the entire scratch area, with . If you want to experiment with Eidos and try out ideas interactively, this is the place to do it; as new Eidos concepts are introduced in this manual, you may wish to try them out.



### 3.4 The Eidos variable browser

Finally, clicking the Show Variable Browser button  will show a variable browser that looks like this:



Symbol	Type	Size	Values
T	logical	1	T
F	logical	1	F
NULL	NULL	0	
E	float	1	2.71828
PI	float	1	3.14159
INF	float	1	INF
NAN	float	1	NAN
► slimgui	object<SLiMgui>	1	SLiMgui

Only a few variables are presently defined; these are all constants – unmodifiable values defined for your use by Eidos or SLiM. These constants include the values T and F representing true and false, the values PI and E representing those mathematical constants (3.1415... and 2.7182...), the value INF representing infinity and NAN representing “Not A Number” (both numerical constants used in floating-point arithmetic), and some others; these will be discussed as they come up.

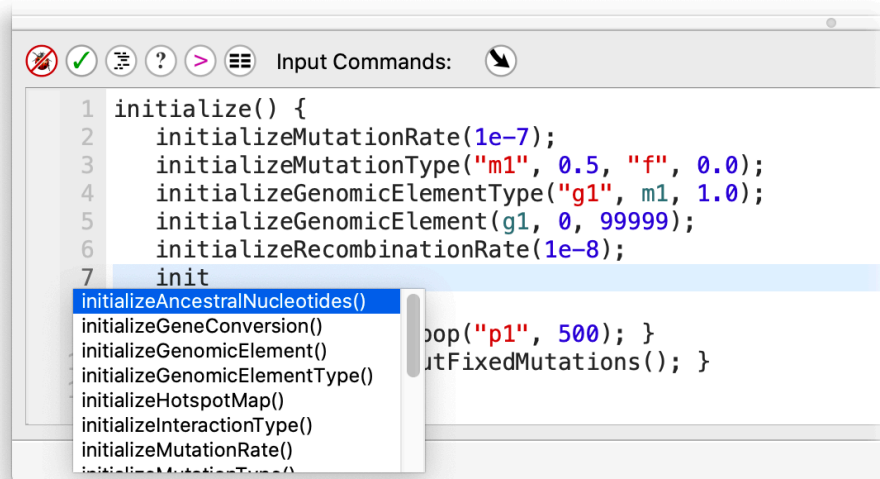
The built-in Eidos constants are shown in gray in the variable browser since they are unmodifiable and, frankly, uninteresting. The variable browser will show other symbols in black rather than gray (whether constants or variables). For example, you might try now entering `x=10` at the console prompt in the Eidos console window. After you press return to execute that command, you will see the variable x appear in the variable browser, defined with a type of integer, a size of 1 (because 10 is a single value, as opposed to a larger vector of values), and a value of 10.

We will return to the variable browser in later sections to examine some of the objects defined by SLiM. In general, however, just be aware that the variable browser exists, and that you can use it to examine all of the Eidos variables that you will be working with in later sections. It can be a useful tool for debugging.

### 3.5 Automatic code completion and command syntax lookup

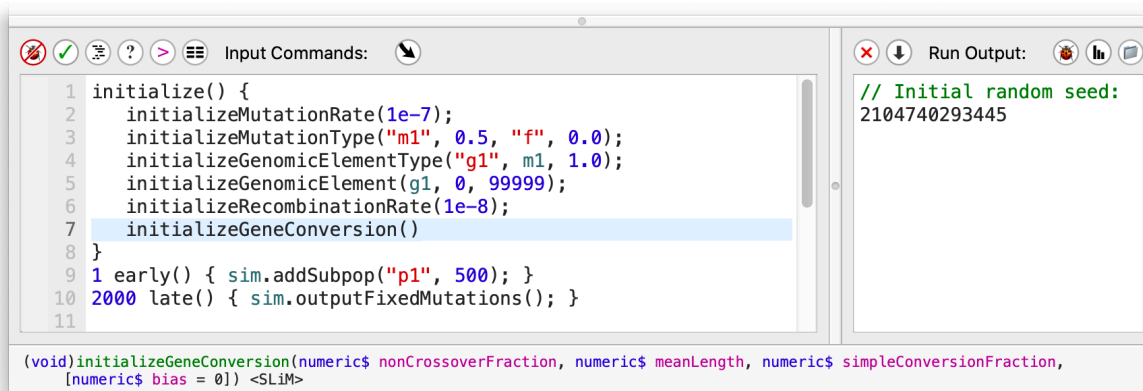
In the next chapter, we will start exploring a simple neutral model in SLiM. As you start writing scripts like that, you will sometimes find it difficult to remember the name of something you need to use in your script – a function, a property, or a method. We are getting a bit ahead of ourselves, but there is a final feature of SLiMgui to discuss here. Suppose you were writing a script, but you couldn’t remember the name of the `initializeGeneConversion()` function. You could look it up in the documentation, but in SLiMgui there is an easier way: *code completion*.

To try this out, first click in your script at the appropriate spot inside an `initialize()` callback, and then start typing with `init` since you know you’re looking for an `initialize...()` function. Now press the <esc> key, which requests the code completion feature from SLiMgui. In response, you should see something like this:



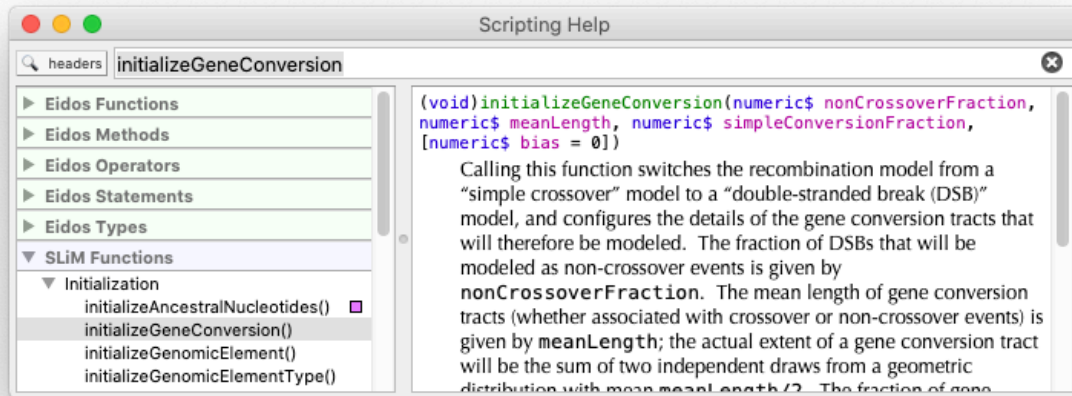
This is a pop-up list of all of the things Eidos knows about that start with `init`. The entry for `initializeGeneConversion()` is second in the list; to accept it you could click on it, or press down-arrow and then press return. You could also type a G, which dismisses the auto-complete pop-up; now you have `initG`, and if you press `<esc>` again, `initializeGeneConversion()` will conveniently be the first choice listed. (If you decide you don't want to auto-complete after all, just press `<esc>` again, or click elsewhere in the window.)

Having accepted `initializeGeneConversion()` as your choice, you might now also be unable to remember what its parameters are. Again, SLiMgui has a feature that can help. Simply click between the parentheses of the function call, and then look at the status bar at the bottom of the window:



The function signature shown there reminds you of the parameters needed, including their types and their names. Section 4.1.3 has further discussion about this feature, getting into details of the structure and symbolism of the function signature, which would have little meaning before we have started scripting.

As described in section 3.2, there is a script help facility provided in SLiMgui, which can be called up with an option-click on any part of your script. If the function signature shown above is not sufficient to jog your memory, and you want to see the full documentation for the `initializeGeneConversion()` function, just option-click on the function name, `initializeGeneConversion`, and the script help window will come up showing the results of a search on that term:



The full documentation for `initializeGeneConversion()` is shown, which should clarify any remaining questions.

The code completion (with <esc>) and context help (with option-click) features work for properties and methods as well. This may not mean much to you yet, since functions, properties, and methods have not yet been formally introduced, but keep all this in the back of your mind as you start exploring scripting in chapter 4.


### 3.6 Script prettyprinting

As you start writing your own scripts in SLiMgui, you may find that they look a bit raggedy because their line indentation doesn't follow standard coding conventions, like this:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  subpopCount = 5;

      for (i in 1:subpopCount)
sim.addSubpop(i, 500);
    for (i in 1:subpopCount)
      for (j in 1:subpopCount)
        if (i != j)
          sim.subpopulations[i-1].setMigrationRates(j, 0.05);
    }
10000 late() {
  sim.outputFull();
}
```

This is a contrived example (based on the recipe in section 5.3.2), but you really will find that your script indentation starts to suffer as you change the structure of your code, moving code from block to block, wrapping some code in loops and unwrapping other code, and so forth. Indentation issues can make your code hard to read and maintain, but manually fixing the indentation of each line is a hassle. Instead, you can just use SLiMgui's automatic script

prettyprinting facility. Select “Prettyprint Script” from the Script menu, or click the  button just above the scripting pane, and your code will be reformatted:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  subpopCount = 5;

  for (i in 1:subpopCount)
    sim.addSubpop(i, 500);
  for (i in 1:subpopCount)
    for (j in 1:subpopCount)
      if (i != j)
        sim.subpopulations[i-1].setMigrationRates(j, 0.05);
}
10000 late() {
  sim.outputFull();
}
```

Only the line indentation is changed; all other aspects of your coding style are preserved, including blank lines, brace style, spacing around operators, and so forth. As mentioned before, you can option-click (alt-click) the Prettyprint Script button to have SLiMgui do more extensive reformatting that does not preserve those things. In either case, prettyprinting your script is an undoable operation, so try it and see whether you like the results.

### 3.7 Further SLiMgui features

SLiMgui has many more features, but they won’t make much sense until we have gotten into writing scripts. For future reference, here is a cross-reference to some sections that present more features of SLiMgui:

- Running a simulation: section 4.1.9.
- Viewing script block registration/deregistration: section 5.1.2.
- Executing a simulation up to a specified tick: section 5.1.2.
- Visualizing population structure and migration: sections 5.1.3 and 5.3.
- Using the Play speed slider: section 5.2.2.
- Using the Jump pop-up menu: section 5.4.
- Viewing the recombination rate map and mutation rate map: sections 6.1.1 and 14.10.
- Viewing the sex ratio, cloning rate, and selfing rate: sections 6.2.2, 6.3.1, and 6.3.2.
- Viewing the registered mutation types and their distributions of fitness effects: section 7.1.
- Viewing the registered genomic element types: section 7.2.
- Viewing genomic elements and selecting a subrange of the chromosome: section 7.3.
- Customizing display colors for mutations, genomic elements, and individuals: section 7.4.
- Graphing of mutation and population dynamics in SLiMgui: chapter 8, section 11.1.
- Alternative population display modes: section 12.3.
- Haplotype display in the chromosome view: chapter 8, section 14.4.
- Haplotype plots (not in the chromosome view): sections 13.3, 14.4, 15.9.
- Live plots generated by R inside SLiMgui: sections 13.5 and 14.8.
- Multispecies modeling in SLiMgui: chapter 19, particularly section 19.2, 19.3, and 19.5.

Debugging output and “debug points” in SLiMgui: section 20.6.

Profiling the speed and memory usage performance of models: sections 21.5 and 21.6.

Manipulating SLiMgui from script using the `SLiMgui Eidos` class: section 24.13.

Many of these features of SLiM are covered in detail in the SLiM workshop, which is available online, and is free and open to all; see section 1.10 for further information. Doing the workshop is a great way to learn to use SLiMgui effectively.

Finally, note that SLiMgui knows the recipes presented in this cookbook, and can open them for you directly. Just choose the recipe you want from the “Open Recipe” submenu under SLiMgui’s File menu, and the recipe will open in a new SLiMgui window.

## 4. Getting started: Neutral evolution in a panmictic population

This chapter will introduce the basic concepts involved in making, configuring, running, and obtaining output from a simple neutral simulation.

### 4.1 A basic neutral simulation

There is a tradition in computer programming of introducing a new language by writing a “hello, world” program. In Eidos, this minimal “hello, world” program is quite simple:

```
print("hello, world");
```

This single-line program calls a built-in function named `print()`, which prints whatever you tell it to. Here, `print` is called with a single argument, the string “hello, world”, enclosed in quotes that indicate it is a string. The semicolon at the end indicates to Eidos that the statement – the line of code – ends at that point. If you are using SLiMgui, you can open the Eidos console window now and enter this command at the prompt; after you press return, “hello, world” will be shown as output. In SLiM, we use Eidos as a tool for building and controlling SLiM simulations; for a complete introduction to Eidos, see the manual *Eidos: A Simple Scripting Language*.

That was a minimal Eidos program; now let’s look at a minimal SLiM simulation script, which is a bit more involved. Here’s a truly minimal “hello, world” SLiM script:

```
initialize()
{
}

1 early()
{
    print("hello, world");
}
```

This shows the basic structure of a SLiM script: it is made up of *blocks*, enclosed by braces {}, that can contain Eidos code inside them; there are two blocks in the script above. Each block is *declared* with a heading that tells SLiM what kind of block it is, when it should run, and so forth; in the above script we declare one block as an `initialize()` callback that performs initialization tasks, and the other as an `early()` event that runs in tick 1, providing an opportunity to say hello to the world with a call to `print()`. We will go into all those concepts in detail soon. This script is not a representative SLiM script, however, in that it doesn’t set up any genetics at all; we haven’t told SLiM anything about the length of the genome, the mutation or recombination rate, and so forth. This is called a “no-genetics model”, and it is something SLiM allows you to do, but in general it is not terribly useful – the point of SLiM, in general, is to simulate genetics! We will not see another no-genetics model until chapter 19, when we delve into the world of multispecies modeling; when simulating more than one species, you might wish for one of the species to evolve while another is being simulated purely for its ecological effects, so no-genetics models become more useful in that context. The above model also doesn’t create any subpopulations or individuals, so it doesn’t actually simulate anything at all. Let’s leave it behind, and start instead with a minimal SLiM script that actually simulates genetics, since that is our goal.

We’ll start with a basic neutral simulation that models a genomic region of length 100 kb in a population of 500 diploid individuals, evolving over 10000 generations. Neutral mutations occur uniformly in this region at a rate of  $10^{-7}$  per bp per gamete. Recombination also occurs uniformly at a rate of  $10^{-8}$  per bp per gamete (corresponding to 1 cM/Mbp). The Eidos commands for specifying this simulation are as follows:



```

// set up a simple neutral simulation
initialize()
{
    // set the overall mutation rate
    initializeMutationRate(1e-7);

    // m1 mutation type: neutral
    initializeMutationType("m1", 0.5, "f", 0.0);

    // g1 genomic element type: uses m1 for all mutations
    initializeGenomicElementType("g1", m1, 1.0);

    // uniform chromosome of length 100 kb
    initializeGenomicElement(g1, 0, 99999);

    // uniform recombination along the chromosome
    initializeRecombinationRate(1e-8);
}

// create a population of 500 individuals
1 early()
{
    sim.addSubpop("p1", 500);
}

// run to tick 10000
10000 early()
{
    sim.simulationFinished();
}

```

Running this script at the command line on a Unix system (including macOS), or on Windows, is very simple. If it is saved in the current directory as a file named `test.txt`, and if the `slim` command-line tool is in the shell's executable path, then the command:

```
slim test.txt
```

should suffice to run it. Otherwise, paths will be needed, which depend upon where the `slim` command-line tool and the `test.txt` file are located. On macOS, the SLiM installer will install `slim` in `/usr/local/bin`; if `test.txt` is in your home directory, then the command would be:

```
/usr/local/bin/slim ~/test.txt
```

Running this script in the SLiMgui application on macOS, Linux, or Windows is also straightforward. Just launch SLiMgui (installed in `/Applications` by the macOS SLiM installer; see chapter 2 on installation of SLiMgui on Linux and Windows), and then copy/paste the script into the script area of SLiMgui's window – or, even easier: select the section 4.1 recipe from the Open Recipe submenu of SLiMgui's File menu. Then press the big Recycle button at the upper right of the window to reset the simulation with the new script, and press the Play button just to the left of the Recycle button. Chapter 3 provided a brief introduction to SLiMgui and the parts of its window, and the sections below will walk through running this particular simulation in SLiMgui in detail.

There are a couple of general things to say first about the whole script. First of all, comments in Eidos begin with two slashes, `//`, and continue to the end of their line; the script above has a comment above most of the lines of executable code.

Second, this code snippet utilizes syntax coloring to make the meaning of the script clearer, just as shown in SLiMgui's input area. Numeric constants are shown in blue, string constants in red, SLiM object constants such as `m1`, `g1`, and `sim` in sage, and comments in green. Syntax coloring will generally be used in this manual for Eidos scripts.

Third, braces `{}` are used in Eidos to enclose whole blocks of code. The code above has three such blocks: an `initialize()` callback that sets up the simulation, an Eidos event that is set to execute in tick 1 – the beginning of simulation execution – to add a subpopulation, and an Eidos event that runs in tick 10000 and stops the simulation. (We could somewhat equivalently say that the events run in cycle 1 and cycle 10000, or in generation 1 and generation 10000; until we look at nonWF models in chapter 16 and multispecies models in chapter 19, “generation”, “tick”, and “cycle” will be effectively synonymous. However, they are conceptually different, and events are, in fact, scheduled in ticks, not cycles or generations. See section 1.5.7 for further discussion.)

Fourth, whitespace characters such as spaces, tabs, and newlines are not generally significant in Eidos; comments, also, are considered whitespace and do not matter to the execution of your code. The above script could thus be written more compactly as:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
```

With that prelude, the following subsections will explore each of the commands in this script in detail. If you wish to follow along in SLiMgui, you can open the section 4.1 recipe in SLiMgui (from the Open Recipe submenu of the File), then press the Recycle button to reset the simulation to the current script (redundant here, in fact, since the newly opened window begins in a Recycled state), and then press Play to see the simulation run.

#### 4.1.1 *initialize()* callbacks

Before a simulation can really begin running, some initialization tasks need to be done. SLiM needs to know basic simulation parameters like the mutation rate, the length of the chromosome, and so forth. Setting up this foundational state is done before the execution of the first tick, at what is called “initialization time”. At initialization time, SLiM calls any blocks in the script that are designated as `initialize()` callbacks (simply by having `initialize()` before their starting brace). Our sample script defines one `initialize()` callback; you are allowed to have more than one, in which case they are called sequentially in the order in which they are defined in the script. This would be useful mostly for conceptual division of your code into discrete sections.

An `initialize()` callback may contain arbitrary Eidos code; you can define variables, execute loops, and call functions (all topics we will explore in future sections). Mostly what you will do in your `initialize()` callbacks, in practice, is call initialization functions that are built into SLiM to help set up your simulation. These functions have names that begin with the word `initialize`; the `initialize()` callback here calls five such functions, discussed in the following sections. (A full reference of all of the `initialize...()` functions provided by SLiM is given in Part II of this manual, the reference section; in general all of the topics discussed in Part I are summarized in a more reference-oriented format there).

#### 4.1.2 Mutation rate

The first line of our `initialize()` callback is:

```
initializeMutationRate(1e-7);
```

This calls the SLiM function `initializeMutationRate()`, passing a single parameter, the numeric constant `1e-7`. This is written in a sort of scientific notation commonly used in programming; `1e-7` means  $1.0 \times 10^{-7}$ , and could also be written in Eidos as `0.0000001`. Numeric values in Eidos may be of type `integer`, like `6` or `-17`, or of type `float`, like `1e-7` or `0.0000001`.

The effect of this statement is to tell SLiM that the simulation will use a uniform mutation rate of `1e-7` (per base position per gamete) across the whole chromosome. SLiM uses this rate to determine how many mutations arise in each offspring that it generates in each generation in the genomic elements being simulated. It is also possible to set up a mutation rate map that varies the mutation rate along the chromosome, but we will defer that until later. Precisely what mutations can arise in a given element is governed by other aspects of the simulation configuration, discussed next.

#### 4.1.3 Mutation types

The next line in the `initialize()` callback is:

```
initializeMutationType("m1", 0.5, "f", 0.0);
```

This calls the SLiM function `initializeMutationType()` to set up a new mutation type. You may define as many mutation types in SLiM as you wish. Each is given a unique symbolic name; the mutation type defined here is given the name `m1`, as requested by the first parameter to the function call. A mutation type encapsulates a few key pieces of information about a particular type of mutation: the dominance coefficient for mutations of this type (`0.5` here), the distribution of fitness effects (DFE) to be used (a fixed fitness effect here, as represented by `"f"`), and any parameters that configure the distribution of fitness effects (`0.0` here, giving the fixed selection coefficient that will be used by all mutations of this type). This call creates a new Eidos variable named `m1`, and so henceforth we can use the symbol `m1` to refer to this mutation type.

This mutation type, `m1`, thus represents neutral mutations – always with a selection coefficient of `0.0`. Mutation types might represent things like neutral mutations, beneficial mutations, deleterious mutations, nearly neutral mutations, etc., with different distributions of fitness effects. Each time that a mutation is created, its selection coefficient is drawn from the distribution of fitness effects specified by the mutation type to which the new mutation belongs. It can be useful to use different mutation types to represent mutations that are conceptually different in your simulation even if they share the same DFE. For example, you might use mutation type to represent neutral mutations created by SLiM randomly as a result of normal mutational processes, and a different mutation type with exactly the same DFE to represent a particular neutral mutation that you deliberately create in your script and want to track separately during the simulation.

You might notice that it seems hard to remember what all four parameters are and which order they are supposed to go in. If you are using SLiMgui, a helpful feature is provided to address this problem (introduced in section 3.5, with a screenshot). If you click anywhere inside the parentheses of the `initializeMutationType()` call, a summary of the syntax of the call is shown in the status bar at the bottom of the window:

```
(object<MutationType>$)initializeMutationType(is$ id,  
      numeric$ dominanceCoeff, string$ distributionType, ...)
```

The Eidos manual has a complete description of meaning of these summaries, called *function signatures*. Briefly, this function signature first gives the type of value returned by the function – here, an object value of class `MutationType`, representing the new mutation type created by the function call. As shown by the trailing `$`, this returned value is a *singleton*, meaning that there will be only a single value present, rather than a vector containing multiple values. You could assign the result of `initializeMutationType()` into a variable:

```
x = initializeMutationType("m1", 0.5, "f", 0.0);
```

This would define `x` to refer to the new mutation type. Since the variable `m1` is defined automatically with the same value, this is usually not necessary; you would use `m1` to refer to the new mutation type. In some cases, however, it can be useful, particularly if you are defining more than one mutation type using a loop.

Now we get to the parameters listed in the function signature. The first parameter may be either an integer or a string (thus the designation `is`, from the leading character of each permitted type name). This parameter should be a singleton (as shown by the `$`, again), and is named `id`; it gives the identifier to be used for the new mutation type, either as an integer like `1` or as a string like `"m1"` (both of which would lead to a variable named `m1` being defined). The second parameter must be numeric (meaning either an integer or a float), is a singleton, and is named `dominanceCoeff`; this is self-explanatory. The third parameter must be a singleton string, and is named `distributionType`. Then there is `...`, a signifier that zero or more additional parameters might be supplied of unspecified type and name. In the case of `initializeMutationType()`, these additional parameters configure the distribution of fitness effects for the new mutation type, and their number and type depend upon the distribution specified by `distributionType`. Exponential, gamma, and normal DFEs are also supported by SLiM, for example, and require different parameters for their specification; consult the reference manual (section 24.11) for details about all of the DFE types currently supported in SLiM.

For now, the overall point is that the function signature is always available in the status bar whenever you click inside a function, and can be used as a quick reference to remind you of the meaning and type of each parameter. Remember that in SLiMgui you can also option-click in Eidos code to bring up the script help window showing a search on the clicked term (see section 3.5); an option-click on `initializeMutationType` would bring up not only the function signature for the function, but the full text from the reference section regarding the function. You might try this now, as an experiment.

#### 4.1.4 Genomic element types

Let's now turn to the third line of the `initialize()` callback:

```
initializeGenomicElementType("g1", m1, 1.0);
```

This creates a new genomic element type named `g1`. A genomic element type represents a particular type of chromosomal region – introns, exons, UTRs, etc. As with mutation types, you might wish to use a special genomic element type for a particular chromosomal region that you want to track separately in your simulation, even if it has the same characteristics as other similar regions – an exon of particular interest, for example.

Each genomic element type has a particular mutational profile. Mutations occur in all genomic elements at the same uniform rate, as set by the overall mutation rate; but the types of mutations that can occur in a particular genomic element type are determined by the mutational profile of that genomic element type. Here, genomic element type `g1` is defined as using mutation type `m1` for all of its mutations (as specified by the proportion `1.0` supplied as the third parameter).

Suppose we wanted to define `g1` as using a mix of three mutation types, `m1`, `m2`, and `m3`? Let's look at the function signature for `initializeGenomicElementType()` to see how we might do this:

```
(object<GenomicElementType>$)initializeGenomicElementType(is$ id,  
  io<MutationType> mutationTypes, numeric proportions)
```

In some ways this is quite similar to the function signature for `initializeMutationType()` that we examined above; it returns an object (this time of class `GenomicElementType`), and it takes a singleton integer or string named `id` to specify the identifier for the new object. The second parameter is named `mutationTypes`, and can be either of type integer or object (with class `MutationType`); so we could specify the mutation type for the genomic element type either using an object like `m1`, as we did in the example script, or using an integer like `1` (identifying mutation type `m1`), which might be more convenient. The third parameter is of type numeric (integer or float, remember) and specifies the proportion of all mutations that will be drawn from the given mutation type.

The second and third parameters are not designated as singletons (they do not have a `$` in their type specifier). This means that they can be *vectors* of values, which allows us to specify multiple mutation types for a genomic element type:

```
initializeGenomicElementType("g1", c(m1,m2,m3), c(1,2,10));
```

The `c()` built-in function returns all of its parameters pasted together into a single vector; we use it here to make a vector containing the three mutation types, and another vector containing proportions. Notice the proportions don't have to sum to 1; they are just relative proportions.

In Eidos, *all* values are in fact vectors; singletons are just vectors containing exactly one value. Even when you write a numeric constant like `10`, that is actually an Eidos vector that happens to be a singleton. Many Eidos operators and functions are built to work with whole vectors; this simplifies your code by removing the need for many of the loops that would be necessary in other languages in order to loop over the elements in an array. It also makes Eidos much faster, since a whole vector can be processed in a single statement. For example, take this Eidos statement:

```
sum(1:10);
```

This adds the numbers from 1 to 10 using the built-in `sum()` function. A vector containing the numbers from 1 to 10 is generated using the sequence operator of Eidos, `:`, which counts upwards (or downward) from its first operand to its second operand. Once you get used to the way vectors work in Eidos, you will find that they often make complicated tasks very easy.

#### 4.1.5 Genomic elements

Having set up genomic element type `g1` in the third line, the fourth line of the `initialize()` callback now uses `g1` to set up a genomic element:

```
initializeGenomicElement(g1, 0, 99999);
```

A genomic element is simply a region of the chromosome that uses a particular genomic element type. For example, you might have one genomic element type that represents introns, and you might then have dozens (or thousands) of genomic elements in your chromosome that use that genomic element type to represent a specific intron at a particular position. The call here sets up a single genomic element that stretches from base position 0 to base position 99999, and is thus 100000 bases long, using genomic element type `g1`.

A chromosome can consist of many genomic elements, but two genomic elements cannot overlap. Genomic elements also do not have to cover the entire chromosome. For example, you

can run a simulation with only two genomic elements of length 1 kb each, separated by 50 kb of “empty space” (see section 14.13). Mutations are only simulated in those regions of the chromosome where a genomic element has been specified. Recombination events, however, are still simulated across the whole chromosome. You can see the genomic elements you have defined by turning on the display of genomic elements in SLiMgui’s chromosome view (see section 3.1).

The following example shows how we can set up a chromosome consisting of ten genomic elements of type g1, with gaps between them at regular intervals:

```
for (index in 1:10)
  initializeGenomicElement(g1, index*1000, index*1000 + 499);
```

This introduces a few new Eidos concepts. First of all, `1:10` makes a vector containing the sequence from 1 to 10, as we saw above; it is equivalent to `c(1,2,3,4,5,6,7,8,9,10)`.

Second, the `for...in` construct loops over that vector; for every value in `1:10`, the value will be assigned to the variable `index` and then the body of the loop – the statement on the next line – will be executed. In this way, `initializeGenomicElement()` will be called ten times, first with `index` equal to 1, then with `index` equal to 2, and on up to `index` equal to 10.

Third, you can see that Eidos, like most programming languages, allows you to write mathematical expressions that are evaluated when the script executes. The `*` operator indicates multiplication and the `+` operator indicates addition, so the expressions here calculate start and end positions based upon the current value of `index`. All in all, this `for` loop is essentially equivalent to:

```
initializeGenomicElement(g1, 1000, 1499);
initializeGenomicElement(g1, 2000, 2499);
...
initializeGenomicElement(g1, 10000, 10499);
```

It should now be fairly obvious how one might extend the `for` loop above to create a much more complex chromosome involving multiple genes, each with UTRs and introns and exons, interspersed with non-coding regions, and so forth. You would likely want to use additional features of Eidos that are described in the Eidos language manual, such as nested `for` loops and the modulo operator `%`. You could also potentially read in a chromosome map from a file on disk, parse that map in whatever format it is written in, and execute the corresponding commands to build the map in Eidos; Eidos has all of the tools you would need to do this, including file input/output and string processing.

#### 4.1.6 Recombination rate

The final line of the `initialize()` callback in our script is:

```
initializeRecombinationRate(1e-8);
```

This specifies the recombination rate for the whole chromosome to be  $1e-8$ , which means that a crossing-over event will occur between any two adjacent bases with a probability of  $1e-8$  per gamete (see section 1.5.6 for a discussion of how exactly SLiM then models those crossing-over events). In this case, a single recombination rate is used along the whole chromosome. The function signature for `initializeRecombinationRate()`, however, is:

```
(void)initializeRecombinationRate(numeric rates, [Ni ends = NULL],
  [string$ sex = "*"])
```

Parameters named `ends` and `sex` are listed; however, they are in brackets `[]`. This indicates that these parameters are optional and may be omitted (in which case they are assigned the default



values shown in the signature; see the Eidos manual for further discussion of optional arguments). That is what we did in the example script, so `ends` was assigned its default value of `NULL` (and `sex` received its default value of `"*"`; other values can be used to define sex-specific recombination rates/maps). If we included `ends`, which must be either `NULL` or an integer value according to its `Ni` type-specifier in the signature, then it should be the last base position of the chromosome, like this:

```
initializeRecombinationRate(1e-8, 99999);
```

Notice that the `rates` and `ends` parameters are not singletons (no `$`). In fact, we may supply a vector of end positions and a matching vector of rates, defining a series of recombination regions, each starting at the base position after the previous range ends (or at the beginning of the chromosome, for the first range). For a simple example, we could make the first half of the chromosome experience a much higher recombination rate than the second half:

```
initializeRecombinationRate(c(1e-7, 1e-8), c(49999, 99999));
```

You may supply as many recombination regions as you wish, specifying recombination hotspots, etc. These recombination regions are not related to genomic elements; the boundaries of recombination regions and genomic elements do not need to match up at all. You can see the recombination regions you have defined by turning on the display of recombination regions in SLiMgui's chromosome view (see section 3.1). Note also that recombination can be tailored on an individual-level basis using a `recombination()` callback to model things such as chromosomal inversions; see sections 14.4 and 25.6.

As mentioned in section 4.1.2, it is also possible to define a mutation rate map that varies the mutation rate along the chromosome. In fact, that is done with exactly the same syntax that we have just seen here to configure a recombination rate map; a vector of rates and end positions is supplied to `initializeMutationRate()` instead a singleton rate.

With this, we're done discussing the `initialize()` callback section of the script. After the `initialize()` callback finishes, the tick counter will be set to 1 and the first tick will be ready to execute, as discussed next. If you're following along in SLiMgui (which you can do by pressing the Step button in the simulation window once), the tick counter will change from `initialize()` to 1, indicating that the initialization phase is done and tick 1 is next up to be executed (but has not begun yet), and likewise for the cycle counter shown in SLiMgui, which is in sync with the tick counter for this simple model (see section 1.5.7 for discussion of timescale concepts, but the distinctions between ticks, cycles, and generations are not really important this point). If you have the variables browser open, you will see that the variables `m1` and `g1`, defined by the `initialize()` callback, are now listed, along with another variable named `sim` that will be discussed below.

#### 4.1.7 Eidos events

The next section of our script is:

```
1 early()
{
    sim.addSubpop("p1", 500);
}
```

This defines an *Eidos event* that is scheduled to run in tick 1 – at the very beginning of the simulation, but after the `initialize()` callbacks have completed. This is the usual time at which new subpopulations are constructed, and that is what this event does, as will be discussed in the next section. For now, however, we're interested in this idea of defining Eidos events.

Eidos events are scheduled to run in a specific tick or range of tick. A single tick is specified with a single number, as here. A range of ticks is specified with the Eidos sequence operator; we could make an event run in ticks 10 through 19, for example, by writing:

```
10:19 early() { ... }
```

After the tick range for the event is `early()`, which specifies the point in the tick cycle when the event will run (*early* in the tick cycle, in this case; see the tick cycle diagram in section 1.3). The `...` here is the script for the event, of course; it can be whatever Eidos code you wish. We will see a great many Eidos events in later examples.

#### 4.1.8 Subpopulations

The Eidos event scheduled to run in tick 1 has a single line in its body:

```
sim.addSubpop("p1", 500);
```

This looks a bit like a function named `sim.addSubpop()` is being called – and that is almost right, but not quite. After `initialize()` callbacks complete, SLiM defines a new Eidos constant named `sim` that represents the species being simulated. The `sim` object is used to access all sorts of species-level properties, as we will see later. It is also a sort of gateway for a specialized sort of function calls referred to as *methods*. A method is a call that can be made to a particular object, to request that that object perform some operation. All values of type object in Eidos support a handful of basic methods (as you can read about in the Eidos manual), and the object classes defined by SLiM often support several more specialized methods as well.

So here we call the `addSubpop()` method of the `sim` object. This adds a new subpopulation to the species; it will be represented by the new variable `p1`, as specified by the first parameter, and it will have an initial size of 500 diploid individuals (the second parameter). The individuals in the new subpopulation are blank slates; they contain no mutations as of yet. The method call is done using the member operator of Eidos, a period (`.`); this operator selects one member, such as a method, from an object. Apart from this syntactic difference, methods are in many ways quite similar to functions, but they encapsulate an “object-oriented” perspective; instead of a globally defined function performing an operation, a specific object is asked to perform the operation. Subpopulations “belong” to the species, and therefore the species – the `sim` object – is asked to add new subpopulations.

If you have been following in SLiMgui, press Step again and you will see that a new variable named `p1` appears in the variable browser to represent the new subpopulation. You might now open the Eidos console and enter `sim.methodSignature()`. This is a method that returns the signatures for all of the methods supported by an object. Among other methods in this list (all documented in the reference section of this manual), you will see the signature for `addSubpop()`:

```
- (object<Subpopulation>$)addSubpop(is$ subpopID, integer$ size,  
  [float$ sexRatio = 0.5])
```

Notice there is a third parameter, which is optional, named `sexRatio`. This will be discussed when sexual simulations are covered in chapter 6. The dash at the very beginning indicates that the signature is for a method, as opposed to a function signature, which has no leading dash. Methods are sometimes referred to by name with this leading dash; the `addSubpop()` method is thus sometimes called `-addSubpop()`. The meanings are identical.

#### 4.1.9 Executing the simulation

We have already executed the initialization phase and the first tick of the simulation (if you are following along in SLiMgui). Now try pressing the Play button. The simulation will suddenly run

at full speed, which is probably pretty fast in this case. You will see mutations flicker in and out of existence, and rise and fall in frequency, along the length of the chromosome, as displayed in the chromosome view. If you let the simulation run, it will continue until it reaches tick **10000**, at which point it will stop because of the final snippet of code in our example script, defining a terminating event:

```
10000 early()
{
    sim.simulationFinished();
}
```

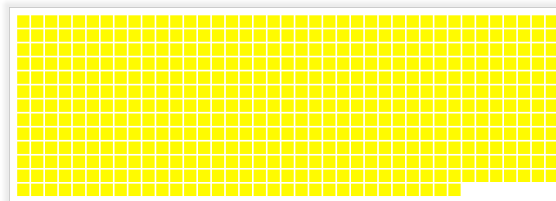
This defines an Eidos event that executes in tick **10000**. The event calls a method named `simulationFinished()` on the `sim` object, and that method declares that the simulation is finished (although it continues to execute until the end of the current tick). This is actually not the typical way that a simulation ends, as we will see in the next section; but it will serve for now. When the simulation stops, the tick counter will read **10001** because tick **10000** finished executing and the next tick to execute (were the simulation not finished) would be **10001**.

Having reached the end of the simulation, let's look at a few parts of the simulation window. The population table view now looks something like this:

ID	N				
p1	500	0.00	0.00	0.00	—

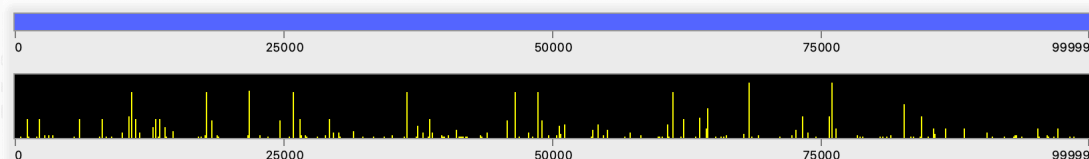
This shows the subpopulation that the tick 1 Eidos event created, named **p1** as shown in the ID column, along with its size (**500**) and its selfing and cloning rates (all **0.00**). The last column shows the sex ratio (M:M+F); species in SLiM are hermaphroditic by default, so the sex ratio in this model is undefined.


The individual view now looks something like this:



Each yellow square represents one individual; there are 500 squares since the subpopulation size is 500. Each individual is colored according to the calculated fitness of that individual; however, in this simulation all mutations are neutral, so all individuals have the same relative fitness, 1.0, and so they are all yellow (as expressed by the color stripe for fitness values, discussed in section 3.1). In a simulation with beneficial and deleterious mutations you would see a range of colors, giving you an immediate visual sense of the fitness distribution across the subpopulation.

Finally, the chromosome view area looks something like this:



The top band shows the whole chromosome; the numbers and ticks below indicate base positions. You can click and drag here to select a subrange of the chromosome for display in the bottom view; a simple click in the top band will return you to viewing the full chromosome. The bottom band displays the selected range of the chromosome (here, the full range). Each yellow bar is a mutation that exists at that position in the chromosome. The height of the bar indicates the frequency of the mutation; if the bar reaches the full height of the band, it has reached fixation and is removed from the display (but you can turn on display of fixed mutations by choosing Display Substitutions using the  action button to the right, in which case they will display as full-height blue bars, by default). Here, all the bars are yellow because all the mutations in this simulation are neutral, and neutral mutations are colored yellow in SLiM by default.

You might have noticed that whole sets of yellow bars tend to rise and fall in synchrony. These are haplotypes that have become associated with each other through genetic drift. The dynamics of mutations, haplotypes, and genetic drift is immediately visually apparent here, underlining the value of having an interactive graphical interface in which you can develop, debug, test, and experiment with your simulations. This visual, interactive workflow also makes SLiMgui a potentially valuable tool for classroom instruction in population genetics and evolutionary biology.

## 4.2 Basic output

So far, our basic neutral simulation simply stops in tick **10000**. Typically, it is desirable for a simulation to produce some output. We will look at a few output options in this section.

### 4.2.1 Entire population

One option is to output the full state of the population (i.e., all individuals in all subpopulations of the species). To do this, we might change our script just a bit:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 late() { sim.outputFull(); }
```

The only change here is that the final event, including the `sim.simulationFinished()` call, has been replaced by a different event to generate output using the `sim.outputFull()` method:

```
10000 late() { sim.outputFull(); }
```

Note the keyword `late()` here. SLiM can run user-defined Eidos events at three different points in the tick cycle, as shown in section 1.3. These three points are referred to as `first()`, `early()`, and `late()` events.

Often, running events early in the tick, prior to the generation of offspring, is desirable. In some cases, however, it is better for an event to run toward the end of the tick, after the generation of offspring – and events that produce output are usually such a case. If the `late()` specifier were `early()` instead, the output would be generated at the beginning of tick **10000**, instead of at the end, and would thus reflect the results of running the model for only 9999 ticks. After the output was generated at the start of the tick, the model would then run for the final tick, with no effect on the model's output.

That would be quite a subtle bug, and easy to miss. In fact, generating output at the beginning of a tick, in an `early()` event, is so likely to be a bug that SLiM will output a warning if you try to do it using one of SLiM's standard output methods. For example, if we change the `late()` designation on the output event above to `early()`, SLiM will generate a warning when the model executes:

```
#WARNING (Species::ExecuteInstanceMethod): outputFull() should probably
not be called from an early() event; the output will reflect state at the
beginning of the cycle, not the end.
```

There are a few other common situations in which you want to use a `late()` event instead of `early()` events, most notably when you introduce new mutations into the simulation in script; these situations will be discussed as they arise, as will `first()` events.

Once you have got this recipe open in a simulation window in SLiMgui (from the Open Recipe submenu under the File menu), press Recycle and then Play, and let the simulation run to the end. Notice that the full state of the population has been printed, in an output section that looks something like this (with large chunks of output skipped over with ellipses):

```
#OUT: 10000 10000 A
Populations:
p1 500 H
Mutations:
29 68306 m1 34640 0 0.5 p1 6848 450
2 68503 m1 7251 0 0.5 p1 6867 561
...
Individuals:
p1:i0 H p1:0 p1:1
p1:i1 H p1:2 p1:3
...
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
p1:1 A 16 17 18 7 8 19 9 20 21 22 23 24 25 26 27 28
...
```

The format here is fairly simple and easily parsable by something like an R script (full output format details are given in section 26.1.1); it can also be read by SLiM in order to read in a population from a saved state, as we will see later. It begins with an output prefix, `#OUT:`, followed by the tick (10000), the cycle (again 10000), and the type of output (A for all).

The next section describes the subpopulations; `p1` is of size 500 and is hermaphroditic (H).

Next comes a list of all mutations present; each line shows one mutation, including (1) a temporary unique identifier used only in this output section, (2) a permanent unique identifier kept by SLiM throughout a run, (3) the mutation type, (4) the base position, (5) the selection coefficient (here always 0 since this is a neutral model), (6) the dominance coefficient (here always 0.5), (7) the identifier of the subpopulation in which the mutation first arose, (8) the tick in which it arose, and (9) the prevalence of the mutation (the number of genomes that contain the mutation, where – in the way that SLiM uses the term “genome” – there are two genomes per individual).

Next comes a list of individuals. The first line of this section as shown above, for example, shows that individual 0 in subpopulation 1 (`p1:i0`) is a hermaphrodite (H), and is comprised of two genomes, `p1:0` and `p1:1`, that will be listed in the following section. This section makes it easier to match genomes with individuals, and can provide additional individual-level information.

The final section lists all of the genomes in the simulation. The first line in the Genomes section as shown above, for example, shows that genome `p1:0` (which the Individuals section told us

belonged to individual p1:i0) is an autosome (A) and contains the mutations with the identifiers 0, 1, 2, ... 15.

From all of this information, the complete state of the population can be reconstructed – every individual, and every mutation contained by every individual. Thus, the method name `outputFull()`. It does not contain the complete information needed to recreate the entire simulation, however; just the current population state.

#### 4.2.2 Random population sample

Often the `outputFull()` method is overkill; you might just want a sample of genomes that is randomly drawn from a particular subpopulation. For example, to output a 10-genome sample at the halfway point of the simulation, here is a modified script:

```
initialize() {  
    initializeMutationRate(1e-7);  
    initializeMutationType("m1", 0.5, "f", 0.0);  
    initializeGenomicElementType("g1", m1, 1.0);  
    initializeGenomicElement(g1, 0, 99999);  
    initializeRecombinationRate(1e-8);  
}  
1 early() { sim.addSubpop("p1", 500); }  
5000 late() { p1.outputSample(10); }  
10000 late() { sim.outputFull(); }
```

The only change relative to the previous script is the added line:

```
5000 late() { p1.outputSample(10); }
```

This calls the method `outputSample()` on the subpopulation from which the sample should be taken, p1, with a requested sample size of 10 genomes (not 10 individuals, note; `outputSample()` outputs randomly selected haploid genomes). If you open this recipe in SLiMgui, Recycle and Play, you will see that at tick 5000 some output appears:

```
#OUT: 5000 5000 SS p1 10  
Mutations:  
21 203188 m1 92838 0 0.5 p1 4489 4  
4 203247 m1 73991 0 0.5 p1 4495 6  
...  
Genomes:  
p1:0 A 0 1 2 3 4 5 6  
p1:1 A 0 7 1 3 4 5 8 6 9  
...
```

The format here is similar to that of `outputFull()`, but the header `#OUT: 5000 5000 SS p1 10` indicates that in tick 5000 (also cycle 5000) a sample (S) was taken from p1 of size 10 genomes and was output in SLiM (S) format. The list of mutations and genomes is identical in format to `outputFull()`, except that only those mutations are shown that are actually present in the sample, and prevalence now refers to the sample rather than the entire population. The list of populations and the list of individuals are also omitted in this output style. (Full output format details are given in section 26.2.)

SLiM also supports output of samples in MS format. The previous recipe can be modified to use the `outputMSSample()` method instead of the `outputSample()` method; running the recipe in SLiMgui would then produce MS-style output at tick 5000:



```

segsites: 73
positions: 0.0262003 0.0351204 0.0445104 0.0597206 0.0647306 0.0702307
0.0871209 0.1039710 0.1046410 0.1162212 0.1222912 0.1407114 0.1608916
0.1728717 0.1775018 0.2036920 0.2366524 0.2462625 0.2468625 0.2475225
0.2531725 0.2858329 0.2880929 0.2913729 0.2998330 0.3045530 0.3048630
0.3132831 0.3279533 0.3400434 0.3484835 0.3544035 0.3547335 0.3586336
0.3594236 0.3765438 0.3845438 0.4385444 0.4441644 0.4460845 0.4483745
0.4692747 0.4801248 0.5282953 0.5387154 0.5493255 0.5563656 0.5660457
0.5810758 0.5871859 0.5880459 0.6183762 0.6271763 0.6368964 0.6389064
0.6678067 0.6880269 0.6961170 0.6961970 0.7082871 0.7340373 0.7362074
0.8214782 0.8450485 0.8694387 0.8696087 0.9022290 0.9114191 0.9187892
0.9264793 0.9364694 0.9835098 0.9969800
1011010000000100001100110001001010100010000010010000000011001011001010001
001101000001010000111111010100100110000000001001000000000101011011000011
0001010000000100000000001000010000000101100101100100111000010000000001000
1011010000100100001100110001001010100010000010010000000011001111001010101
10110100000001000011001100110010101010000010010000000011001011001010001
001101000001010000111011010100100110000000001001000000000101011011000011
0000101111001011111000100011011001100000110000110110001000010101100000
01010100000001000000000010000100000001011001011001011000010000000001000
1011010000000100001100110010101010000010010000000011001011001010001
1011010000100100001100110001001010100010000001001000000011001011001010001

```

Finally, SLiM supports output of samples in VCF format. If the `outputVCFsample()` method is used instead of `outputSample()` in the above recipe, VCF-style output would be produced:

```

##fileformat=VCFv4.2
##fileDate=20160609
##source=SLiM
##slimGenomePedigreeIDs=4999898,4999899,...
##INFO=<ID=S,Number=1,Type=Float,Description="Selection Coefficient">
##INFO=<ID=DOM,Number=1,Type=Float,Description="Dominance">
##INFO=<ID=PO,Number=1,Type=Integer,Description="Population of Origin">
##INFO=<ID=TO,Number=1,Type=Integer,Description="Tick of Origin">
##INFO=<ID=MT,Number=1,Type=Integer,Description="Mutation Type">
##INFO=<ID=AC,Number=1,Type=Integer,Description="Allele Count">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=MULTIALLELIC,Number=0,Type=Flag,Description="Multiallelic">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT p1:i166 p1:i100
p1:i462 p1:i8 p1:i0 p1:i314 p1:i318 p1:i488 p1:i81 p1:i287
1 547 . A T 1000 PASS S=0;DOM=0.5;PO=1;TO=4822;MT=1;AC=4;DP=1000
GT 1|0 0|1 1|0 0|0 0|0 0|0 0|0 0|0 0|1
1 826 . A T 1000 PASS S=0;DOM=0.5;PO=1;TO=4342;MT=1;AC=5;DP=1000
GT 0|0 0|0 0|0 1|1 0|1 0|0 1|0 0|0 0|0 1|0
...

```

#### 4.2.3 Sampling individuals rather than genomes

The previous section showed how to use methods of the `Subpopulation` class to output a sample of genomes from a subpopulation, in either SLiM's own format, or in MS or VCF format. However, in many situations you want to output information about a sample of *individuals*, rather than a sample of genomes – you want to ensure that *pairs* of genomes, each belonging to an individual, are the level of granularity at which the sampling for output occurs. Also, the `Subpopulation` output methods only support sampling from a single subpopulation at a time, but sometimes you want to output a sample drawn from multiple subpopulations, or from the full population. Finally, sometimes you want to output a custom sample that your script chooses itself, rather than a random, equally-weighted sample of the sort that the `Subpopulation` methods allow.

These tasks are all quite straightforward using lower-level methods of the `Genome` class. As an illustration of this approach, we will look at a recipe that outputs the genomes of a weighted sample of individuals drawn from the whole population:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", 0.01);
  initializeGenomicElementType("g1", c(m1,m2), c(1.0,0.01));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
  sim.addSubpop("p2", 500);
  p1.setMigrationRates(p2, 0.01);
  p2.setMigrationRates(p1, 0.01);
}
10000 late() {
  allIndividuals = sim.subpopulations.individuals;
  w = asFloat(allIndividuals.countOfMutationsOfType(m2) + 1);
  sampledIndividuals = sample(allIndividuals, 10, weights=w);
  sampledIndividuals.genomes.output();
}
```

This model involves a few things that we haven't looked at yet, such as multiple subpopulations connected by migration (see section 5.2.1) and multiple mutation types (see section 7.1). However, those details are not our focus here. For our current purposes, it suffices to recognize that there are two subpopulations, `p1` and `p2`, connected by migration, and there are two mutation types, `m1` and `m2`, with `m1` representing common neutral mutations and `m2` representing less common beneficial mutations. Our focus is on the `late()` event in tick **10000**, which produces the output for the model. We will follow through its logic one line at a time.

The first line defines the set of individuals from which our sample will be drawn, here called `allIndividuals`. The `sim.subpopulations` property gives us a vector of all of the subpopulations in the model (i.e., `p1` and `p2`). The `individuals` property of that vector is then all of the individuals (that is, objects of class `Individual`; see section 24.7) gathered from across those subpopulations.

The second line determines the weights that we will use for sampling, here called `w`. Note that the use of weights here is optional; if you want an unweighted sample, you can skip this line and omit the weights vector in the call to `sample()` that follows. In this recipe, however, we want the likelihood that an individual is chosen for sampling to be related to the number of `m2` mutations the individual possesses, so we use `countOfMutationsOfType(m2)`. We add 1 to that count, so that individuals with no `m2` mutations have a weight of 1, individuals with one `m2` mutation have a weight of 2, and so forth. This is somewhat arbitrary, but it does have the nice property that we are guaranteed that the weights vector is not all zero (which would cause a runtime error, since it would not be possible to draw a sample). If you draw your own samples, you probably want to ensure that the sample can always be drawn, to avoid runtime errors.

The third line draws a sample of 10 individuals from `allIndividuals`, using the `sample()` function and passing it the weights vector `w` (using the named arguments syntax of Eidos, for readability). The sample is put into the temporary variable `sampledIndividuals`.

Finally, the fourth line outputs the genomes of the sampled individuals using the `output()` method of `Genome`. This outputs the sample in SLiM's own format, similarly to what we saw in the previous sections; there are also `outputMS()` and `outputVCF()` methods on `Genome` that can be used

similarly to produce MS and VCF output. Since there are two genomes per individual, twenty genomes will be output. Each pair of genomes in the output will come from one sampled individual; in the SLiM and MS output formats this is not explicit, but in the VCF format, since it is a diploid format (as used by SLiM), this pairing into individuals is explicit in the output. See section 24.4.2 for more information on these output methods, and section 26.3 for details on the format of output they generate.

While this recipe implemented one particular sampling scheme, the Genome methods used here can be used to output any vector of genomes, whether a random sample or a specifically selected set. The higher-level output methods of Species and Subpopulation are a little easier to use, but these low-level methods provide greater power and generality.

#### 4.2.4 Substitutions

The outputFull() method of Species does output the full state of the population, but there is some historical state that it does not output. Most notably, it does not output substitutions – mutations which have been fixed and have thus been removed from the population by SLiM for efficiency (see section 1.5.5). If fixed mutations are of interest, SLiM does keep a record of them and can output that record on request. For example, we could output substitutions, in addition to other state, with this script:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
5000 late() { p1.outputSample(10); }
10000 late() { sim.outputFull(); }
10000 late() { sim.outputFixedMutations(); }
```


The important change relative to the previous script is the added line:

```
10000 late() { sim.outputFixedMutations(); }
```

If you open this recipe in SLiMgui, Recycle and Play, you will see that at simulation end some additional output appears:

```
#OUT: 10000 10000 F
Mutations:
0 220169 m1 98564 0 0.5 p1 107 1053
1 221802 m1 1217 0 0.5 p1 268 1152
...
```

The header #OUT: 10000 10000 F indicates simply that in tick 10000 (also cycle 10000) fixed mutations (F) were output. The rest is a list of mutations in almost the same format as before. However, the final number in each line is no longer a prevalence; since the mutation fixed, we know that it was present in every genome. Instead, the final number shows the tick number in which the mutation reached fixation. The output format used here is documented in section 26.1.2. Note that the default behavior of SLiM – that substitutions are removed from all individual genomes – can be deactivated if necessary (see sections 1.5.2 and 24.11.1).

Display of fixed mutations in SLiMgui can also be enabled by clicking the  action button to the right of the chromosome view and checking Display Substitutions; when this is enabled, fixed mutations will be displayed as full-height bars (blue, by default) in the chromosome view.

#### 4.2.5 Automatic logging with LogFile

Often it is desirable to produce an output file that, rather than being a dump of genetic information about a population or a sample, is a table of summary statistics or metrics regarding the simulation. It can be useful to append such table-based output to a file periodically, so that the state of the simulation can be observed through time as the simulation runs. To support this common style of output, SLiM (as of version 3.5) provides a built-in class called `LogFile` (documented in full in section 24.9).

The recipe we will use is quite simple:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 999999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 1000);
    sim.addSubpop("p2", 1000);
    p1.setMigrationRates(p2, 0.001);
    p2.setMigrationRates(p1, 0.001);
    log = community.createLogFile("sim_log.txt", logInterval=10);
    log.addCycle();
    log.addCustomColumn("FST", "calcFST(p1.genomes, p2.genomes);");
}
20000 late() { }
```

Like section 4.2.3, we have two subpopulations, but with a lower migration rate, and neutral mutations only. The new and interesting change here is the addition of automatic logging using `LogFile`:

```
log = community.createLogFile("sim_log.txt", logInterval=10);
log.addCycle();
log.addCustomColumn("FST", "calcFST(p1.genomes, p2.genomes);");
```

The first line creates a new `LogFile` object that is attached to the simulation. We request that it log to a file named `sim_log.txt` in the Desktop folder of our home directory (which is symbolized with a tilde, `~`, in macOS and Linux); `LogFile` outputs to a file, not to SLiM's output stream as we have seen thus far. We also request that it log automatically every ten ticks, with `logInterval=10`; automatic logging is optional, but often useful. Finally, the `createLogFile()` method returns the `LogFile` object, which we put in the local variable `log`.

This method is called on an object we have not seen before, `community`. The `community` object represents the entire ecological community being modeled; as we will see in chapter 19, multispecies models can contain more than one species, and `community` manages them all. The `community` object is of class `Community`, whereas `sim` is of class `Species`; the *class* of an object tells you what methods that object has, and where to look in the reference documentation for more information about it. We will use `community` to access simulation-level facilities that are shared by all species; `LogFile` creation with `createLogFile()` is one such shared facility.

The next two lines configure the data that we want to be logged. The first calls `addCycle()` to add a standard column, named `cycle`, that contains the current cycle number. The second calls `addCustomColumn()` to add a column named `FST` that contains custom data calculated by our script – in this case, the  $F_{ST}$  (a standard metric of divergence) between our two subpopulations `p1` and `p2`. The data for the `FST` column is generated by a snippet of Eidos code that is passed to

`addCustomColumn()` as a string (called a “lambda” in Eidos; see section 20.5). Stripped of its quotes, that code is:


```
calcFST(p1.genomes, p2.genomes);
```

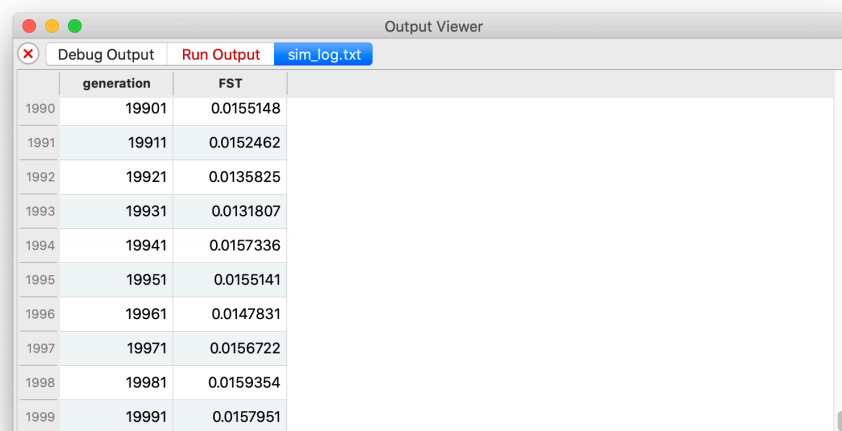
The `LogFile` will run the lambda provided to `addCustomColumn()` whenever it is generating a new row of output. In this case, `calcFST()` is a built-in function in SLiM that calculates the  $F_{ST}$  between two vectors of genomes and returns it as a float (see section 24.17.2), which `LogFile` then outputs as the data value for the FST column.

If we run this model, the `sim_log.txt` file looks like this at the end of the run (with many lines elided by the ellipsis):

```
cycle,FST
1,0.000250063
11,0.000681749
21,0.00102468
...
19971,0.0140662
19981,0.0121124
19991,0.0108835
```

There are several things to notice here. The log file is in CSV (comma-separated value) format, which is `LogFile`’s default; its optional `sep` parameter allows the creation of a TSV (tab-separated value) file instead, by passing `sep="\t"`, if desired. A header line was automatically emitted with the names of the columns; `LogFile` always generates a header. New rows have been generated every ten cycles, as requested. Finally, note that although our model finished at the end of cycle 20000, the last line emitted is for cycle 19991; this is because automatic logging would have next triggered at the end of cycle 20001, so we might wish to make our model end with that cycle instead so that the log file covers the full simulation run.

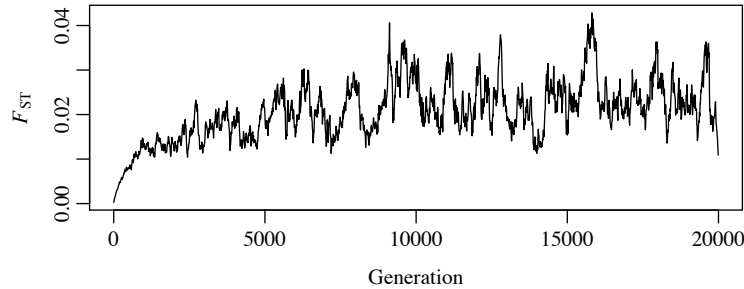
Beginning in SLiM 3.7, we can view this `LogFile` output directly in SLiMgui. If you click the debugging output viewer button  in the main model window, it opens the debugging output viewer, and clicking on the `sim_log.txt` tab button provides the output (shown here from a different run):



	generation	FST
1990	19901	0.0155148
1991	19911	0.0152462
1992	19921	0.0135825
1993	19931	0.0131807
1994	19941	0.0157336
1995	19951	0.0155141
1996	19961	0.0147831
1997	19971	0.0156722
1998	19981	0.0159354
1999	19991	0.0157951

This can be useful for viewing output as a simulation is running, for debugging purposes.

We can also easily open this CSV file in R and plot it:



This shows that the  $F_{ST}$  between the two subpopulations starts at zero and grows over time, eventually reaching a dynamic equilibrium around 0.02 or so. This equilibrium depends upon parameters such as the migration rate and the mutation rate, of course. It is interesting to integrate the beneficial mutations of section 4.2.3 and see what they do to the  $F_{ST}$  over time (with, perhaps, a much lower  $m_2$  fraction, like 0.0001, and a higher selection coefficient, like 0.1).

This is a rather trivial example of the use of `LogFile`. It has some other interesting features: it can output a compressed file, and log out new data rows on request rather than on an automatic schedule (see section 24.9). We will see `LogFile` again in later sections, such as sections 9.11 and 16.19.

#### 4.2.6 Custom output with *Eidos*

The `outputFull()`, `outputSample()`, `outputMSSample()`, `outputVCFSample()`, and `outputFixedMutations()` methods described in the preceding sections generate output in fixed formats that may or may not be useful to you, and you may or may not wish to avail yourself of the table-based logging provided by `LogFile`. There is one more built-in output method that is described in the reference section – `outputMutations()` – but it, too, may not be what you need. What to do when you want to produce output in a format not supported by SLiM?

This is an area where the power of *Eidos* really shines, since you can write whatever *Eidos* script you want, to generate whatever kind of output you want. Some of the more advanced recipes in this cookbook will generate custom output of interesting kinds; section 11.1's recipe will include code to calculate and print the  $F_{ST}$  between two subpopulations to assess their genetic divergence, for example, and section 14.1's recipe calculates and prints the heterozygosity of a population to assess the effects of inbreeding. In this section, we will look at a much simpler example, as an introduction to the topic of custom output using *Eidos*.

Suppose you are interested only in the base positions of all of the mutations in the population; you can achieve this with the following output event:

```
100 late() { cat(paste(sim.mutations.position, sep="\n")); }
```

With our basic neutral simulation script, this generates output like:

```
31113
57761
8790
...
```

This is precisely what we want. How does it work? Let's dissect it, step by step, since it is more complex than the *Eidos* code we've seen so far.

First of all, we have seen the `sim` object before, representing the simulated species. This object, which is an *Eidos* object of class `Species`, has a property named `mutations` that yields a vector



containing all of the mutations in the simulation (a vector of type `object` and class `Mutation`, to be precise). Apart from section 4.2.3, where we quickly glossed over this issue, this is the first use we have seen of a *property*; a property is somewhat like a method, in that it is a member of an object, but whereas a method performs an operation (perhaps involving a lot of computation, and perhaps altering the state of the simulation), a property simply corresponds to a value that exists inside the object. The `sim` object knows all of the mutations it contains; it doesn't have to calculate them or create them, it just has to return the value it already has. It is therefore a property – the mutations property. The member operator is used to access properties, just as it is used to access methods, so the value of `sim.mutations` is the vector of mutations contained by the simulation.

That value is itself an object vector. As mentioned before, all values in Eidos are actually vectors; the vector of mutations might contain many elements (each of which is called an *object-element*), but it is nevertheless a single Eidos value, an object vector. The `Mutation` class in SLiM defines a property, `position`, that is the base position of the mutation; we can access this property on our vector of mutations to get a vector of positions. This involves a certain amount of work behind the scenes; each mutation has its own position, and Eidos must loop over all of the mutations in the vector of mutations, getting the position of each one and pasting all the positions together to make a new vector. Eidos does this for you, though, since Eidos is a vector-based language; so `sim.mutations.position` is all the Eidos code that is needed to get a vector of the positions of all mutations in the simulation.

Peeling the onion back a layer, this expression is contained in a function call:

```
paste(sim.mutations.position, sep="\n")
```

This function takes a vector argument (`sim.mutations.position`) and pastes all of the elements together to form a singleton value of type `string`. We haven't really talked about `string` yet; a `string` is simply a sequence of characters, like "hello, world". Eidos can paste strings together, split them apart, and print them out; it can also convert most other types into `string` for the purpose of output. The `paste()` function always produces a `string` as its result; it converts the elements of the vector it is given into `string` elements in order to do so. The second parameter of `paste()`, `"\n"`, is a `string` containing a single character, the newline character, represented with the escape sequence `\n` (you can read more about strings and escape sequences in the Eidos manual; it is not worth getting into here). It is assigned to the `sep` named argument of `paste()`, which controls the separator character used when pasting. The final result is that each of the integer positions in `sim.mutations.position` is converted to a `string` representation and then pasted together with the others, with newlines in between, to produce the desired output as seen above.

Note that this output event is designated as a `late()` event, for the reasons outlined in section 4.2.1; in this case, we wish to see the positions of mutations at the end of tick 100, not the beginning. With custom output code of this sort, SLiM is not able to infer that it is likely to be an error if it runs in an `early()` event instead, however; so if the `late()` designation is changed to `early()` here, SLiM will not produce a warning. When writing custom output script events, you need to think carefully about whether they should be `first()`, `early()`, or `late()` events (but a `late()` event is usually what you want for output).

The simplicity of this example – just a single line of Eidos code! – should make it clear that generating output in whatever format you desire is likely to be straightforward once you get the hang of how Eidos works.

Since the power of this may not be entirely apparent yet, let's consider a problem that might arise in using SLiM. You might wish to produce MS-style output for a sample of genomes spanning the whole population, but the built-in `outputMSSample()` method of `Subpopulation` (section 4.2.2) only supports generation of MS-style output from a sample of a single subpopulation. **This recipe**

is **probably of mainly historical interest**, since the addition of the `Genome` method `outputMS()`, but let's pretend that method doesn't exist; the same general problem will arise whenever you want to output data in a format that SLiM does not intrinsically support. Generating MS-style output using Eidos is trivial:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
  sim.addSubpop("p2", 500);
}
// custom MS-style output from a multi-subpop sample
2000 late() {
  // obtain a random sample of genomes from the whole population
  g = sample(sim.subpopulations.genomes, 10, T);

  // get the unique mutations in the sample, sorted by position
  m = sortBy(unique(g.mutations), "position");

  // print the number of segregating sites
  cat("\n\nsegsites: " + size(m) + "\n");

  // print the positions
  positions = format("%.6f", m.position / sim.chromosome.lastPosition);
  cat("positions: " + paste(positions, sep=" ") + "\n");

  // print the sampled genomes
  for (genome in g)
  {
    hasMuts = (match(m, genome.mutations) >= 0);
    cat(paste(asInteger(hasMuts), sep="") + "\n");
  }
}
```

This model is quite straightforward; the only complication is that there are two subpopulations, and we wish to produce MS-style output for a sample across both of them. The tick 2000 event does precisely that. First it obtains the genomes upon which the output will be based; here this is done using `sample()` on the vector of all genomes in the simulation, but any other sampling scheme could be used instead. To sample specifically from subpopulations p1 and p2, for example, without sampling from any other subpopulations that might exist, you could do:

```
g = sample(c(p1.genomes, p2.genomes), 10, T);
```

The sample size is 10, and sampling is done with replacement (the T parameter to `sample()`), but that can obviously be customized. Next `unique()` and `sortBy()` are used to get a vector of the mutations present in the sample, sorted by position. The `size()` of that vector is the number of segregating sites, so that is trivial to output. Printing the positions is also simple; `format()` is used to guarantee that every position is formatted with six digits to the right of the decimal, in decimal rather than scientific notation even for very small values. Finally, the sampled genomes are printed in MS format, using `asInteger()` to convert logical values from `match()` into 0s and 1s.

This recipe brings in a lot of Eidos functions that we haven't seen before; we will not explain them all in detail, since they are all documented in the Eidos manual. The point is simply to illustrate that even relatively sophisticated output can be generated very easily in Eidos.

#### 4.2.7 The simulation endpoint

The astute reader will have noticed that in the previous sections we not only added output events, we also removed one line from the original script of section 4.2.1:

```
10000 early() { sim.simulationFinished(); }
```

We can do this because SLiM will generally stop after the last tick in which an event is scheduled. In our basic neutral simulation, however, we had no output commands; if we had omitted the line above with the `simulationFinished()` call the simulation would have ended at the end of tick 1. Alternatively, we could have just written:

```
10000 early() { }
```

This “empty event” would have caused SLiM to run out to the end of tick 10000, because there was still a future event scheduled. At the end of tick 10000 SLiM would then stop, since there was no future event after tick 10000 scheduled. In fact, `simulationFinished()` is really useful only when you wish to end a simulation before it would naturally end on its own. You might check for an equilibrium condition, for example, and end the simulation if it has been at equilibrium for the past 1000 ticks, even though it would otherwise have run further.

There is one further caveat to mention regarding how simulations end. It is possible to define an Eidos event that runs in every tick. For example, we could write the following script:

```
initialize() {  
    initializeMutationRate(1e-7);  
    initializeMutationType("m1", 0.5, "f", 0.0);  
    initializeGenomicElementType("g1", m1, 1.0);  
    initializeGenomicElement(g1, 0, 99999);  
    initializeRecombinationRate(1e-8);  
}  
1 early() { sim.addSubpop("p1", 500); }  
late() { p1.outputSample(10); }
```

Notice that there is no tick number before the Eidos event defined in the last line; this indicates that the event should be run in every tick, from here to eternity. If you open this recipe in SLiMgui and Recycle and Play, it will stop at the end of tick 1, however; there is always a scheduled future event (the output event), but SLiM does not use it in determining when the simulation should end, precisely because it has no expiration date. SLiM assumes that it is not your wish to run a simulation that runs forever – commonly known as an “infinite loop” in programming – so the simulation end is determined without reference to that event.

To define a final tick for the simulation in this situation, you would use precisely the trick described above: just add an “empty event”, like:

```
100 early() { }
```

The simulation will now run to the end of tick 100. Note that there is no need to designate the event as a `late()` event; SLiM always runs the last tick to completion, even if `simulationFinished()` is called. The only way to halt a simulation immediately, before completing the current tick, is to call the Eidos error function `stop()`.

## 5. Demography and population structure

The previous chapter discussed the structure of a basic neutral simulation in SLiM, including `initialize()` callbacks, Eidos events, the `Species` class, and many conceptual underpinnings of SLiM such as mutation types, genomic element types, and chromosome organization. It also covered some basic features of the Eidos language, such as vectors and vector operations, function calls, objects, and method calls. From here, we assume a working knowledge of these topics; consult the Eidos manual and the SLiM reference section of this manual as needed.

In this chapter, we will focus on building on this foundation to make some simple recipes for simulations that do interesting things with demography and population structure.

### 5.1 Subpopulation size

#### 5.1.1 Instantaneous changes

As we saw in section 4.1.8, we can create a new subpopulation with `sim.addSubpop()`, a method call which takes the initial size of the subpopulation as a parameter. Importantly, this sets the *census* population size, not the *effective* population size; see section 9.11 for discussion of that distinction. What if we want the population size to change later? This is very straightforward; for example, here is a simple script that models a population that goes through a bottleneck:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 1000); }
1000 early() { p1.setSubpopulationSize(100); }
2000 early() { p1.setSubpopulationSize(1000); }
10000 late() { sim.outputFixedMutations(); }
```

The `initialize()` callback is unchanged from previous models. In tick 1 a subpopulation named `p1` is set up with an initial size of 1000. In tick 1000, a method named `setSubpopulationSize()` is called on `p1` to change its size to 100, the beginning of the bottleneck. In tick 2000 the size is set back to 1000, ending the bottleneck. The simulation then executes until it ends in tick 10000 with output of fixed mutations.

If you run this in SLiMgui (don't forget to Recycle), the population size changes as expected.

#### 5.1.2 Exponential growth

Sometimes you may want the size of a subpopulation to vary in a continuous fashion, rather than experiencing discrete changes as in the previous recipe. This is straightforward in SLiM because of the power of Eidos to express mathematical relationships. We will look at several different versions of this recipe in order to explore different aspects of the problem.

As a first example, to make a population experience a period of exponential growth:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
```

```

1 early() { sim.addSubpop("p1", 100); }
1000:1099 early() {
    newSize = asInteger(p1.individualCount * 1.03);
    p1.setSubpopulationSize(newSize);
}
10000 late() { sim.outputFixedMutations(); }

```

Most of the script is unchanged; the work is done in the second Eidos event, which is scheduled to run from tick 1000 to 1099, producing exponential growth for 100 generations. This event calculates a new size, assigning it into `newSize`, and then sets the subpopulation size using that variable. The use of the variable is for readability; it would be essentially equivalent to write:

```
p1.setSubpopulationSize(asInteger(p1.individualCount * 1.03));
```

Let's look more closely at that statement. First, the current size of the subpopulation is accessed through the `individualCount` property of `p1`. That size is then multiplied by 1.03 to produce a new size based on an exponential growth rate of 1.03. Finally, the size is converted to an integer by the `asInteger()` function (since `setSubpopulationSize()` does not allow you to set a non-integer size), and the integer size is passed to `setSubpopulationSize()`. There is a suite of `as...` functions in Eidos that allow you to convert values to various new types, but converting float values to integer with `asInteger()` is probably the most common conversion.

An important point here is that the `setSubpopulationSize()` call does not change the current size of the subpopulation; after all, what would the genetics of the new individuals be? Instead, it sets a new target size that will be used the next time that an offspring generation is created; a few additional offspring will be made to reach the new target size. If you access the `individualCount` property immediately after calling `setSubpopulationSize()`, you therefore observe that the individual count has not changed. This is the reason why `setSubpopulationSize()` is not called `setIndividualCount()`; the difference in name is intended to emphasize how SLiM works.

Since the population size gets rounded down to an integer in each tick, this code does not actually achieve the precise exponential growth rate of 1.03 that we wanted. Let's fix that:

```

initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 100); }
1000:1099 early() {
    newSize = asInteger(round(1.03^(sim.cycle - 999) * 100));
    p1.setSubpopulationSize(newSize);
}
10000 late() { sim.outputFixedMutations(); }

```

Running this in SLiMgui shows that the population reaches a size of 1922, whereas the previous version reached a size of 1630 – a significant difference! Beware accumulated roundoff error.

How does the new version work? The expression `sim.cycle-999` computes the number of generations of exponential growth that have occurred; in cycle 1000 this is 1, in cycle 1099 it is 100. Next, `1.03^(sim.cycle-999)` raises 1.03 to that power; `^` is the exponentiation operator in Eidos, as in many programming languages. That calculates the result of the exponential growth over the requisite number of generations. Finally, that result is rounded to the nearest whole number by `round()`, which produces a float result, and then that whole number is converted to an integer by `asInteger()`.

Sometimes we may want the population size to increase exponentially until a specific target size is reached, without knowing how many generations that might take. This is a bit trickier. One possible implementation does this by brute force (leaving out the `initialize()` code):

```
1 early() { sim.addSubpop("p1", 100); }
1000:2000 early() {
    if (p1.individualCount < 2000)
    {
        newSize = asInteger(round(1.03^(sim.cycle - 999) * 100));
        p1.setSubpopulationSize(newSize);
    }
}
10000 late() { sim.outputFixedMutations(); }
```

This uses an `if` statement to increase the size of the subpopulation only if it is still less than 2000 (this is the first time we've seen the `if` statement in Eidos, but it should be pretty obvious what it does here; see the Eidos manual for clarification). The tick range for the event has been expanded to 1000:2000, because we don't know what tick the target size will be reached in; it will be well before tick 2000, so this is good enough, but is a bit sloppy. Alternatively, we could calculate the exact tick in which the target size is reached (1101, as it happens), and use that instead of 2000; if we did that, we wouldn't even need the `if` statement, since the exponential growth would reach its target in the event's last invocation.

To ensure that the last generation of growth produces exactly 2000 individuals, we can use the following solution:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 100); }
1000: early() {
    newSize = round(1.03^(sim.cycle - 999) * 100);
    if (newSize > 2000)
        newSize = 2000;
    p1.setSubpopulationSize(asInteger(newSize));
}
10000 late() { sim.outputFixedMutations(); }
```

First of all, notice that the tick range for the exponential growth event is now written as 1000:, omitting the end tick. Just as supplying no tick range at all for an event means "run in every tick", omitting the end tick means "run in every tick after the specified start tick". One may similarly omit the start tick with the syntax :end, meaning "run in every tick from the beginning until the specified end tick".

The logic inside the event has also changed. Now the code *always* calculates a new size. If that size is greater than 2000 it gets clamped to 2000, which enforces the maximum population size we wanted. The clamped size is then set on the subpopulation.

The drawback to this solution is that the event runs in every tick from 1000 onward, calling `setSubpopulationSize()` to set the size to 2000 over and over in ticks after the target size has been reached. That is inefficient; more importantly, it might interfere with other changes we might want to make to the population size later in the simulation. We'd really like our event to run for exactly the needed duration to reach the target size, and then not run in subsequent ticks, without having



to hard-code a final tick for it. There is a simple solution to this problem – and this, you will be relieved to read, is the final, polished version of our exponential growth recipe (so the `initialize()` callback is provided to make the recipe complete):

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 100); }
1000: early() {
  newSize = asInteger(round(1.03^(sim.cycle - 999) * 100));
  if (newSize >= 2000)
  {
    newSize = 2000;
    community.deregisterScriptBlock(self);
  }
  p1.setSubpopulationSize(newSize);
}
10000 late() { sim.outputFixedMutations(); }
```


Braces have been added around the `if` statement's consequent to make a group of statements; all of the statements in the group will be executed if the condition of the `if` statement is true. This use of braces makes what is called a *compound statement*; compound statements are legal anywhere in Eidos that a single statement is legal.

The interesting change here, though, is the addition of the line:

```
community.deregisterScriptBlock(self);
```

This line prevents the event from continuing to run after the target size has been reached. It is a common pattern in SLiM scripting, so it is worth a bit of discussion. SLiM keeps track of *script blocks* that are registered for execution in the simulation. All of the events and callbacks that you write in your input script are automatically registered; it is also possible to construct and register new script blocks dynamically, as we will see in a later chapter. Script blocks that are registered can be deregistered, using the `deregisterScriptBlock()` method of `Community`, as we are doing here. Once a block is deregistered, SLiM forgets about it and no longer executes it.

The other thing to be explained here is `self`. This is a variable that is defined whenever SLiM is executing an event or callback; it refers to the currently executing script block. We use it here so that our exponential growth event can deregister itself. Once deregistered, the event is no longer executed, and will remain deregistered until the Recycle button is pressed.

You can see the dynamics of script registration and deregistration graphically in SLiMgui, by the way. Start by pressing Recycle, with the recipe above already open in a SLiMgui window. Now if you open the drawer for the simulation window by pressing the drawer button  to the right of the chromosome view area, you will see a list of registered scripts:

**Eidos Blocks:**

ID	Start	End	Type
—	0	0	initialize()
—	1	1	early()
—	1000	MAX	early()
—	10000	10000	late()

The exponential growth event is listed as the third entry there; if you hover the mouse over its entry for a second or two, you will even see the code for it, shown in a tooltip. Now click in the tick field, enter 1101, and press return:


Tick:

This causes SLiMgui to execute the simulation up to the beginning of the requested tick, which is just before the target population size is attained. If you now click Step, you will see the exponential growth event disappear from the list of registered events.

If your scripting gets complicated, with script blocks registering and deregistering frequently, this facility for viewing all of the registered script blocks can prove quite useful.

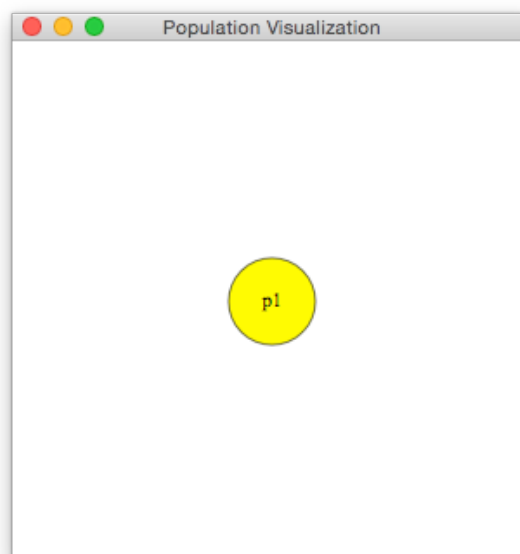
### 5.1.3 The population visualization graph

In the previous section, we saw several different ways to make a subpopulation grow exponentially for a period of time. In this section, we show how to visualize the dynamics of demography and population structure graphically in SLiMgui.

Start by copying and pasting the last recipe into a new SLiMgui simulation window, and then Recycle to parse the script and get ready to execute it. Now click on the Show Graph button, , and from its pop-up menu select “Graph Population Visualization”. This will bring up a small window, presently empty.

If you click Step twice, in order to execute the initialization phase and then tick 1, you should see a small yellow circle labeled “p1” appear. This circle represents subpopulation p1; in particular, its radius represents the size of the subpopulation, and its color represents the mean fitness value of the subpopulation.

Now press Play, and keep your eyes on the subpopulation circle. When the simulation reaches tick 1000 the circle for p1 will begin to grow, and it will continue growing until the subpopulation size reaches its target, 2000:



In this instance the visualization is fairly trivial; still, it is useful for verifying that the population expansion happens in the way that it was planned. In future sections we will see that the

population visualization graph can also help us to visualize migration dynamics, fitness dynamics, and other such things, making it a very useful tool for simulation debugging.

#### 5.1.4 Cyclical changes

Another common demographic dynamic is a cyclically varying subpopulation size, perhaps representing seasonality, or a decadal oscillation in resource availability. Implementing this in SLiM is quite trivial; for example:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 1500); }
early() {
    newSize = cos((sim.cycle - 1) / 100) * 500 + 1000;
    p1.setSubpopulationSize(asInteger(newSize));
}
10000 late() { sim.outputFixedMutations(); }
```

The `initialize()` callback is unchanged. The interesting work is done by the second event – the one with no tick range, which you might recall means that it will be executed in every tick. This event first calculates a new size, and then sets the new size in the subpopulation (converting to integer as usual). The new size is calculated based upon the current cycle count, `sim.cycle`, passed through the `cos()` function, which calculates a cosine. (Note that the cycle count is the same as the tick count, for this simple model; see section 1.5.7 for discussion of timescales). The cycle is scaled and translated in order to arrange that the first offspring generation has a size of 1500, matching the size of the parental generation set up by `addSubpop()`. Of course that is not required; this particular arrangement is just for demonstration purposes.

The `cos()` function calculates a cosine based on a given angle in radians, as is standard. The periodicity of the cyclical population size is therefore based upon the fact that there are  $2\pi$  radians in a circle; given the scaling factor of 100, the population will complete a full cycle in  $200\pi$  generations, which is about 628. To make a subpopulation cycle with whatever periodicity you wish, just adjust the scaling factor accordingly. Similarly, since cosine varies between -1 and 1, the scaling factor of 500 on that, plus the translation of 1000, means that the population size cycles between 500 and 1500. You might wish to view these dynamics in SLiMgui’s population visualization graph, as described in the previous section.

This recipe uses `cos()` to generate cyclical dynamics, but you can plug in whatever formula you wish. The population size does not have to depend only upon the tick, either; the flexibility of Eidos allows you to implement whatever demographic model you wish, including demography that depends upon the model dynamics themselves, as we will see in the next section.

#### 5.1.5 Context-dependent changes: Muller’s Ratchet

Sometimes you might want subpopulation size to depend upon something other than time. An example would be a situation in which population size depends upon mean fitness (e.g., a model of so-called “hard” selection); if the mean fitness is low then the subpopulation size would be small (perhaps because the subpopulation is getting outcompeted by individuals of some other species, or perhaps because the subpopulation is just so unfit that individuals are having trouble surviving and feeding themselves even without competition). Here we will examine such dynamics in a model of Muller’s Ratchet:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "e", -0.01);
  m2.convertToSubstitution = F;
  initializeGenomicElementType("g1", c(m1,m2), c(1,1));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 100); }
early() {
  meanFitness = mean(p1.cachedFitness(NULL));
  newSize = asInteger(100 * meanFitness);
  p1.setSubpopulationSize(newSize);
}
10000 late() { sim.outputFixedMutations(); }

```

The `initialize()` callback here has been modified so that there are two mutation types, `m1` and `m2`, produced with equal probability in genomic element type `g1`. The `m1` type represents neutral mutations as usual (a fixed DFE with selection coefficient `0.0`); the `m2` type represents deleterious mutations (an exponential DFE with mean selection coefficient `-0.1`). In our model we want deleterious mutations to continue to affect fitness even after fixation, progressively reducing the population size. For this reason, the `convertToSubstitution` property of `m2` is set to `F` to prevent fixed mutations of that type from being replaced by `Substitution` objects (see sections 22.3 and 24.11.1 for further information on this property).

The `cachedFitness()` method of class `Subpopulation`, called on `p1`, provides a vector of the fitness values of all individuals in `p1` (the `NULL` value passed simply requests fitness values for all individuals; a vector of indices for individuals in `p1` could be passed instead). The `mean()` function calculates the arithmetic mean of the vector it is passed, resulting in the mean subpopulation fitness. A new size for the subpopulation is then calculated using a base size of `100`, representing the size if the population were of mean fitness `1.0`. In our scenario, the base size is simply multiplied by the mean fitness.

This recipe is a very primitive toy model of fitness-based population dynamics; nevertheless, it is interesting to look at it in the population visualization graph, where the population size changes visibly and is clearly correlated with mean fitness, shown as the color of the subpopulation. As deleterious mutations accumulate in the subpopulation, its circle both reddens and shrinks, showing visually that demography is being driven by mean fitness. When the subpopulation reaches extinction due to the accumulation of deleterious mutations by Muller's Ratchet, this model terminates with an error ("undefined identifier p1"), because after subpopulation `p1` is set to a size of `0` the symbol `p1` ceases to exist; setting a size of `0` tells SLiM to remove the subpopulation entirely. One could end the simulation more gracefully by testing for (`newSize == 0`) and calling `sim.simulationFinished()`, perhaps after producing some sort of output regarding the tick and the number of deleterious mutations fixed.

This model would in some ways be simpler and more natural to write as a nonWF model, since in nonWF models fitness is absolute and selection is "hard" by default anyway (see section 1.6).

## 5.2 Population structure

### 5.2.1 Adding subpopulations

We have already seen in previous recipes how to add a single subpopulation by calling the `addSubpop()` method of `sim` (which is of class `Species`). Creating population structure by adding multiple subpopulations – of different sizes, linked by migration – is a very simple extension:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
  sim.addSubpop("p2", 100);
  sim.addSubpop("p3", 1000);
  p1.setMigrationRates(c(p2,p3), c(0.2,0.1));
  p2.setMigrationRates(c(p1,p3), c(0.8,0.01));
}
10000 late() { sim.outputFixedMutations(); }

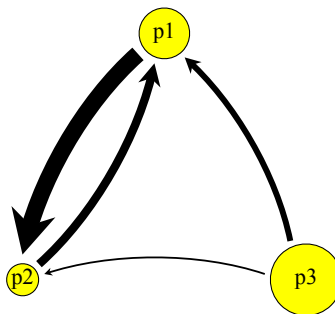
```

For this recipe we have returned to the simple `initialize()` callback we started with, with only one mutation type. The new action happens in the tick 1 event, where we now add three subpopulations of different sizes. Those populations are then connected by migration using calls to the `setMigrationRates()` method of Subpopulation. Let's examine the first such call:

```
p1.setMigrationRates(c(p2,p3), c(0.2,0.1));
```

Since this method call is made on `p1`, it is setting up *immigration into* `p1`. Both `p2` and `p3` are given as sources of immigration, with rates of 0.2 and 0.1 respectively. In SLiM's design, this does not mean that individuals from `p2` and `p3` will move from those populations into `p1`. Rather, it means that when `p1` generates an offspring generation, and parents are chosen for a new offspring individual, 20% of the time both parents will be chosen from `p2`, 10% of the time both will be chosen from `p3`, and the remaining 70% of the time both will be chosen from `p1` itself. In effect, this models immigration of newly generated juveniles from the source subpopulations. This is how migration in WF models works; see sections 1.6 and 22.2 for further discussion. The next line sets up migration into `p2`, in the same way. Since migration into `p3` is not configured, `p3` acts as a source only. Note that, as discussed further in section 22.2.1, migration rates specify the probability that any given offspring individual will come from a particular source subpopulation; the actual number of migrants in a given generation is thus stochastic, not deterministic.

It can be hard to visualize complex population structure just from code like this; happily, SLiMgui's population visualization graph provides a nice way to see what's going on. If you open this recipe in a SLiMgui simulation window, Recycle, press Step twice so that tick 1 has been executed, and then open the population visualization graph, you will see a nice graphical representation of the population structure:



As before, the circle sizes represent the subpopulation sizes, and the circle colors indicate the mean fitness of each subpopulation. The arrows show migration links, with the thicknesses of the

arrows indicating the strength of each link. Here it is immediately obvious that p3 is a source, and that p2 is close to being a sink but does contribute some immigrants to p1.

Of course there is no need for the population structure to all be set up in tick 1; you can add subpopulations, remove subpopulations, and change migration rates in each tick if you wish, as we will explore in the next section. The population visualization graph will update to show the current population structure as your simulation runs.

### 5.2.2 Removing subpopulations

The population structure set up in the previous recipe was quite static, established in tick 1 and unchanging subsequently. Let's explore more dynamic population structure by both adding and removing subpopulations over time and dynamically changing migration. The recipe here is longer because it does more things, but it is only a small extension of previous concepts:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
100 early() { sim.addSubpop("p2", 100); }
100:150 early() {
  migrationProgress = (sim.cycle - 100) / 50;
  p1.setMigrationRates(p2, 0.2 * migrationProgress);
  p2.setMigrationRates(p1, 0.8 * migrationProgress);
}
1000 early() { sim.addSubpop("p3", 10); }
1000:1100 early() {
  p3Progress = (sim.cycle - 1000) / 100;
  p3.setSubpopulationSize(asInteger(990 * p3Progress + 10));
  p1.setMigrationRates(p3, 0.1 * p3Progress);
  p2.setMigrationRates(p3, 0.01 * p3Progress);
}
2000 early() { p2.setSubpopulationSize(0); }
10000 late() { sim.outputFixedMutations(); }
```

The `initialize()` callback is as before. Subpopulations p1, p2, and p3 are now set up in ticks 1, 100, and 1000 respectively. Subpopulation p2 is removed in tick 2000 by setting its size to 0; that is all that is needed to remove a subpopulation.

The rest of the code – the 100:150 event and the 1000:1100 event – introduces some continuous change into the population structure. In the 100:150 event the migration rates between p1 and the newly established p2 grow over time until they reach a target rate. In the 1000:1100 event the new subpopulation p3 grows in size over time, from a very small founder population, and its migrational contribution to p1 and p2 grows over time as p3 grows in size.

Watching these dynamics in SLiMgui's population visualization graph is useful for confirming that they are functioning correctly. However, the simulation may go too fast for the dynamics to be seen clearly. You can use the speed slider, directly below the Play button, to slow it down:





If you set a slower speed, as can be seen in the position of the speed slider shown above, and then Recycle and Play, it should be easier to follow the action.

### 5.2.3 Splitting subpopulations

By default, when new subpopulations are added in SLiM they are composed of “brand-new” individuals with no mutations. To produce more realistic dynamics, we might like the new subpopulations to be split off from an existing subpopulation. Modifying the recipe above, this is quite a simple change:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
100 early() { sim.addSubpopSplit("p2", 100, p1); }
100:150 early() {
    migrationProgress = (sim.cycle - 100) / 50;
    p1.setMigrationRates(p2, 0.2 * migrationProgress);
    p2.setMigrationRates(p1, 0.8 * migrationProgress);
}
1000 early() { sim.addSubpopSplit("p3", 10, p2); }
1000:1100 early() {
    p3Progress = (sim.cycle - 1000) / 100;
    p3.setSubpopulationSize(asInteger(990 * p3Progress + 10));
    p1.setMigrationRates(p3, 0.1 * p3Progress);
    p2.setMigrationRates(p3, 0.01 * p3Progress);
}
2000 early() { p2.setSubpopulationSize(0); }
10000 late() { sim.outputFixedMutations(); }
```

The only change is that the `addSubpop()` calls that established `p2` and `p3` have been changed to `addSubpopSplit()`. This call is very similar to `addSubpop()`, but takes an extra parameter: the existing subpopulation from which the new subpopulation should be split. The split is accomplished by copying the individuals for the new subpopulation from the source subpopulation. In other words, each new individual is an exact genetic clone of an existing individual in the source population. This might seem a bit odd; but after the next generation has been generated, it is as if the new subpopulation had been created empty and then filled with new offspring generated by matings within the source subpopulation. It therefore models juvenile migration from a source subpopulation, just as migration is normally modeled in WF models in SLiM (as we saw before in section 5.2.1). The initial filling of the source subpopulation with clones is, in effect, just choosing which source subpopulation individuals will be the parental individuals that generate the first offspring generation in the new subpopulation.

### 5.2.4 Joining subpopulations

As well as splitting subpopulations off from other existing subpopulations, subpopulations can be joined together. In SLiM this is treated as essentially a special case of normal subpopulation creation and migration, rather than being implemented with a special method like `addSubpopSplit()`.

For example, suppose we begin with two subpopulations, `p1` and `p2`, and wish to merge them – with a particular proportion of admixture – to form a joined subpopulation, `p3`. The recipe for this is quite simple:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
  sim.addSubpop("p2", 1000);
}
1000 early() {
  // set up p3 to generate itself entirely from migrants
  sim.addSubpop("p3", 300);
  p3.setMigrationRates(c(p1, p2), c(0.75, 0.25));
}
1000 late() {
  // remove the source subpopulations
  p3.setMigrationRates(c(p1, p2), c(0.0, 0.0));
  p1.setSubpopulationSize(0);
  p2.setSubpopulationSize(0);
}
10000 late() { sim.outputFixedMutations(); }

```

The 1000 early() event runs before offspring generation. It creates a new subpopulation, p3, at the desired size; here we happen to use 300, but we could use 1500 if we wanted p1 and p2 to join together without any diminishment in size. This new subpopulation is created with 300 new empty individuals, as requested; but that is not what we want. We therefore set migration rates into p3 from p1 and p2 to be 0.75 and 0.25 respectively (again, that is just what we happen to do in this recipe; if we wanted p1 and p2 to admix proportionally to their sizes, we could use rates of 0.3333 and 0.6664).

After this early() event, SLiM will generate offspring, and the empty individuals in p3 will be discarded and replaced by migrant offspring from p1 and p2 as requested. The 1000 late() event executes immediately after this occurs; p3 is now the joined subpopulation, but p1 and p2 still exist – at this point, we have modeled something more like a joint colonization event than a joining of subpopulations. Now we want to remove p1 and p2 from the model, after zeroing out the migration from them. (Technically, zeroing out the migration is not necessary; SLiM will do that automatically when the subpopulations are removed. It is shown here to illustrate the technique that would be used if one wanted to retain p1 or p2 in the model for some reason.)

Of course this recipe can be generalized to admix any number of source subpopulations in any proportions to produce a joined subpopulation of any size, either in a single step or through successive joinings in different ticks; and these mechanics can also be combined with subpopulation splits and size changes as well, to produce any desired demographic history.

### 5.3 Migration and admixture

The previous subsections already showed how to set up patterns of migration among multiple subpopulations. Here, we will focus on recipes for a few standard types of population structure to show how the flexibility of Eidos makes setting up complex population structure easy.

#### 5.3.1 A linear stepping-stone model

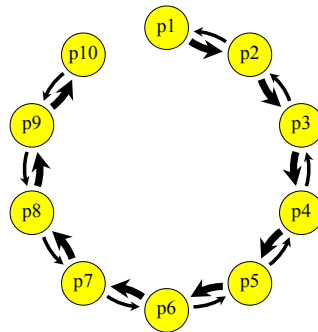
This model describes a linear chain of subpopulations, each of which is connected by migration only to its nearest neighbors. This is quite easy to set up:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  subpopCount = 10;
  for (i in 1:subpopCount)
    sim.addSubpop(i, 500);
  for (i in 2:subpopCount)
    sim.subpopulations[i-1].setMigrationRates(i-1, 0.2);
  for (i in 1:(subpopCount-1))
    sim.subpopulations[i-1].setMigrationRates(i+1, 0.05);
}
10000 late() { sim.outputFixedMutations(); }

```

If you open this recipe in SLiMgui, Recycle, do Step twice to step through tick 1, and open the population visualization graph, you will see the resulting structure:



As you can see, migration is stronger in one direction than in the other; this might represent gene flow in a river system, for example, where gene flow is more likely to go downstream than upstream. Indeed, it would be trivial to interrupt the upstream gene flow completely in certain spots to represent the effect of waterfalls that prevent upstream migration. Using the techniques shown in previous sections, it would also be trivial to make the migration pattern change over time; a major flooding event in one generation could allow gene flow to pass upstream past a small waterfall, for example.

This population structure is achieved using three for loops. We saw for loops briefly in section 4.1.5; let's revisit the concept now. The first loop in this recipe is:

```

for (i in 1:subpopCount)
  sim.addSubpop(i, 500);

```

This causes the statement `sim.addSubpop(i, 500);` to be executed repeatedly as the loop index variable `i` varies from 1 up to `subpopCount`, following the sequence defined by `1:subpopCount`. The result is that new subpopulations are created in order: p1, p2, p3, and on up to `subpopCount`.

The second loop is almost as straightforward:

```

for (i in 2:subpopCount)
  sim.subpopulations[i-1].setMigrationRates(i-1, 0.2);

```

This sets up migration into each subpopulation from the previous subpopulation: into p2 from p1, into p3 from p2, and so forth. Since p1 receives no such migration (there being no p0

subpopulation), the loop starts at 2, not 1. For each value of *i*, the corresponding subpopulation is looked up in the simulations subpopulation array, which is a property of *Species*, using `sim.subpopulations[i-1]`; the use of *i-1* here instead of *i* is because *p1* will be at index 0 in the subpopulation array (since vectors in Eidos are numbered starting at 0, not 1). Given the proper subpopulation, the `setMigrationRates()` call then sets up migration from the previous subpopulation (which is the purpose of *i-1* in that call).

Given that explanation, the operation of the third loop should be fairly obvious. The power of this algorithmic approach to setting up population structure should be clear; if we want 1000 subpopulations arranged in this manner, all we have to do is change `subpopCount = 10`; to `subpopCount = 1000`;. The population visualization graph in SLiMgui will find this a bit challenging to display, but it should be possible to test and debug such a model with just 10 subpopulations, and then scale up to 1000 subpopulations for your production runs.

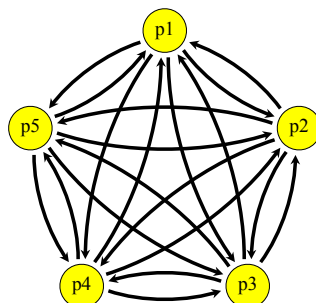
### 5.3.2 A non-spatial metapopulation

Another possible scenario is a non-spatial metapopulation in which migration between each pair of subpopulations occurs at some constant rate. This can be set up as follows:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  subpopCount = 5;
  for (i in 1:subpopCount)
    sim.addSubpop(i, 500);
  for (i in 1:subpopCount)
    for (j in 1:subpopCount)
      if (i != j)
        sim.subpopulations[i-1].setMigrationRates(j, 0.05);
}
10000 late() { sim.outputFixedMutations(); }
```

Here a nested pair of for loops using index variables *i* and *j* sets up the migration from *j* into *i* for each pair of subpopulations. The test `if (i!=j)` prevents the code from attempting to set the migration rate from a subpopulation into itself, which is illegal.

SLiMgui's visualization of this population structure looks like what we wanted:



This recipe uses only five subpopulations (`subpopCount = 5`), but again the code is general and may be scaled up to however many subpopulations you want in your metapopulation (although SLiMgui may not be very good at visualizing these scenarios once they become too complex).

### 5.3.3 A two-dimensional subpopulation matrix

A third common population structure is a two-dimensional grid or matrix of subpopulations, representing a spatial metapopulation in which each subpopulation exchanges migrants only with its direct neighbors. SLiM has no intrinsic concept of geographic space in its simulations (unless continuous space is enabled; see chapter 15), but with a population structure like this a pseudo-geographic regime can be imposed upon a SLiM simulation such that new beneficial mutations, for example, will spread from subpopulation to subpopulation in a similar manner to how they might spread across a real landscape.

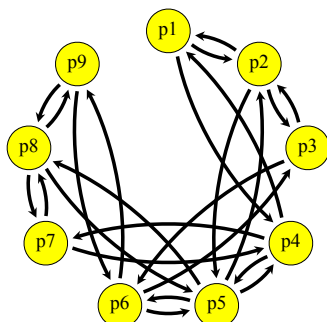
```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    metapopSide = 3; // number of subpops along one side of the grid
    metapopSize = metapopSide * metapopSide;
    for (i in 1:metapopSize)
        sim.addSubpop(i, 500);

    subpops = sim.subpopulations;
    for (x in 1:metapopSide)
        for (y in 1:metapopSide)
        {
            destID = (x - 1) + (y - 1) * metapopSide + 1;
            destSubpop = subpops[destID - 1];
            if (x > 1) // left to right
                destSubpop.setMigrationRates(destID - 1, 0.05);
            if (x < metapopSide) // right to left
                destSubpop.setMigrationRates(destID + 1, 0.05);
            if (y > 1) // top to bottom
                destSubpop.setMigrationRates(destID - metapopSide, 0.05);
            if (y < metapopSide) // bottom to top
                destSubpop.setMigrationRates(destID + metapopSide, 0.05);
        }
}
10000 late() { sim.outputFixedMutations(); }
```

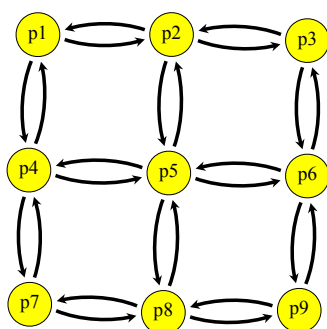
This code is a bit more complex than previous recipes. In the tick 1 event, we first decide how large of a metapopulation we want; `metapopSide=3` means that we will make a 3x3 metapopulation (trivially small, but easier to check for correctness). This can be scaled up arbitrarily; it works well in SLiMgui for values as large as a 1000x1000 metapopulation with 10 individuals per subpopulation. We calculate the number of subpopulations we will need, and make them all with the first for loop.

Then comes the trickier part. We loop over the grid of subpopulations using `x` and `y`, which each range from 1 to `metapopSide`. For each subpopulation in the grid, as determined by `x` and `y`, we calculate the ID of the subpopulation in `destID`, and then fetch the subpopulation itself into `destSubpop`, getting it from the simulation's subpopulation array as before. We then set the migration into `destSubpop` from each of its four sides; if it is at the edge of the matrix, it receives no migrants from that side (although it would be trivial to modify this code to make a wrap-around pattern of migration simulating a toroidal world). Simple arithmetic is used to determine the identifier of neighboring subpopulations based upon `destID`.

SLiMgui's visualization of this setup is complicated; it is not immediately obvious that it represents a two-dimensional metapopulation, but it does:



However, SLiMgui has an alternate method of display for these population visualizations. If you control-click or right-click on the graph, you will get a pop-up menu. Select “Optimized Positions” from that menu, and you should see something more like this:



Here it is much more obvious that the migration pattern is as desired. This positioning optimization algorithm is based on a concept called force-directed layout. It is a somewhat experimental feature in SLiMgui, and may not always give layouts that look good; it is also quite slow for layouts involving more than a dozen or so subpopulations. However, it is worth a try if you want to get a publication-worthy picture of your population structure.

Speaking of “publication-worthy”, note that when you control-click or right-click on the visualization graph, the pop-up menu also has an item entitled “Copy Graph”. That copies the graph to the clipboard as a PDF – very handy for pasting into documents or slides.

#### 5.3.4 A random, sparse spatial metapopulation

The recipe in section 5.3.3 showed how to make a small spatial metapopulation in which subpopulations are arranged spatially in a grid that is connected by orthogonal migration. Here we will extend that recipe to a larger size and a more random metapopulation configuration, and we'll look at a very simple sweep of a beneficial mutation across the metapopulation.

The `initialize()` callback for this model is largely unchanged from section 5.3.3:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 0.5, "f", 0.3);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
```



The only modification here is the addition of a new mutation type, m2, which has been defined to represent beneficial mutations (with a selection coefficient of 0.3). New mutations are still always neutral (mutation type m1); we will introduce the sweep mutation ourselves in script.

The final output event is also unchanged:

```
10000 late() { sim.outputFixedMutations(); }
```

The interesting changes are to the 1 late() event that configures the initial state of the metapopulation:

```
1 late() {
  mSide = 10; // number of subpops along one side of the grid
  for (i in 1:(mSide * mSide))
    sim.addSubpop(i, 500);

  subpops = sim.subpopulations;
  for (x in 1:mSide)
    for (y in 1:mSide)
    {
      destID = (x - 1) + (y - 1) * mSide + 1;
      ds = subpops[destID - 1];
      if (x > 1) // left to right
        ds.setMigrationRates(destID - 1, runif(1, 0.0, 0.05));
      if (x < mSide) // right to left
        ds.setMigrationRates(destID + 1, runif(1, 0.0, 0.05));
      if (y > 1) // top to bottom
        ds.setMigrationRates(destID - mSide, runif(1, 0.0, 0.05));
      if (y < mSide) // bottom to top
        ds.setMigrationRates(destID + mSide, runif(1, 0.0, 0.05));

      // set up SLiMgui's population visualization nicely
      xd = ((x - 1) / (mSide - 1)) * 0.9 + 0.05;
      yd = ((y - 1) / (mSide - 1)) * 0.9 + 0.05;
      ds.configureDisplay(c(xd, yd), 0.4);
    }

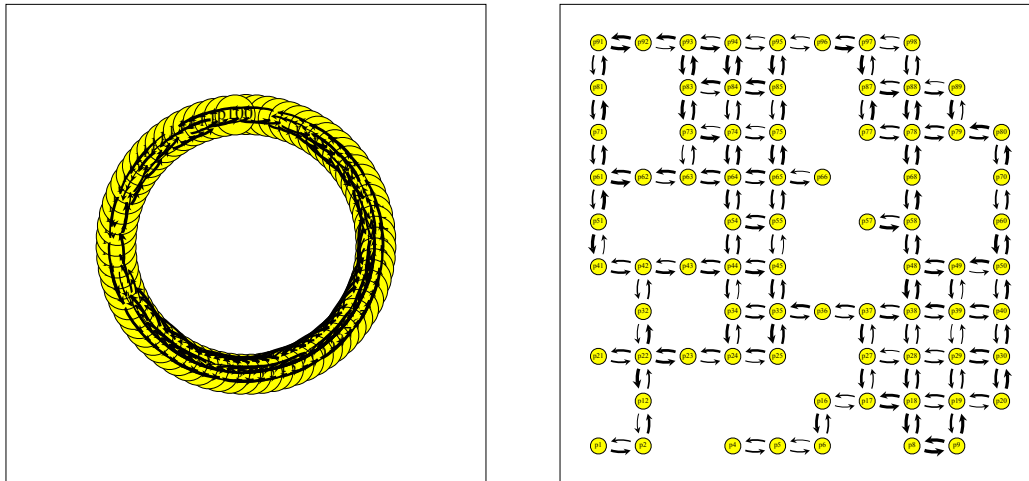
  // remove 25% of the subpopulations
  subpops[sample(0:99, 25)].setSubpopulationSize(0);

  // introduce a beneficial mutation
  target_subpop = sample(sim.subpopulations, 1);
  sample(target_subpop.genomes, 10).addNewDrawnMutation(m2, 20000);
}
```

This creates a 10×10 spatial metapopulation in much the same way as section 5.3.3 created a 3×3 metapopulation: first by creating the subpopulations themselves in a simple loop, and then setting up the migration among them with a pair of nested loops over x and y (see section 5.3.3 for further comments on that basic design).

For each subpopulation, however, it also does something new, under the “// set up SLiMgui” comment: it calculates a visual position for the subpopulation, as xd and yd, and then calls configureDisplay() on the subpopulation to tell SLiMgui to use that position in its display. The coordinate system used by SLiMgui for subpopulation display spans [0,1] in x and y, so the xd and yd values calculated here are within that range. We pass a value of 0.4 for the second parameter to configureDisplay(); this is a scaling factor for the circle used to represent the subpopulation, so here we are telling SLiMgui to use unusually small circles (so that all 100 subpopulations fit into the display without overlapping).

If we ran the model *without* this display configuration (just comment out the call to `configureDisplay()`), we would get the left-hand plot below; the subpopulations are all overlapping and nothing can be discerned at all. (The “Optimized Positions” technique shown in section 5.3.3 is not up to the task either; optimizing such a large network is a difficult problem. Perhaps that experimental feature will eventually be improved to the point that it produces acceptable results for this model, but at present it does not.) If we run it *with* the display configuration, we get the right-hand plot below, which is obviously much better:



Looking at the right-hand visualization, it is now immediately apparent that this is not a complete metapopulation, but rather a sparse metapopulation in which some subpopulations are missing. That brings us to the next lines in the event:

```
// remove 25% of the subpopulations
subpops[sample(0:99, 25)].setSubpopulationSize(0);
```

This takes a sample, of size 25, from the sequence 0:99, selects those subpopulations (using the subset operator on `subpops`), and sets their size to zero. As we saw in section 5.2.2, this removes those subpopulations from the simulation; any connections they have to other subpopulations by migration are broken. That one line, then, very easily produces a random sparse metapopulation. The only caveat is that, depending upon which subpopulations get removed, the remaining metapopulation might not be connected; there might be two or more isolated networks with no connection between them via migration at all. If that is undesirable, further steps would have to be taken to avoid that possibility.

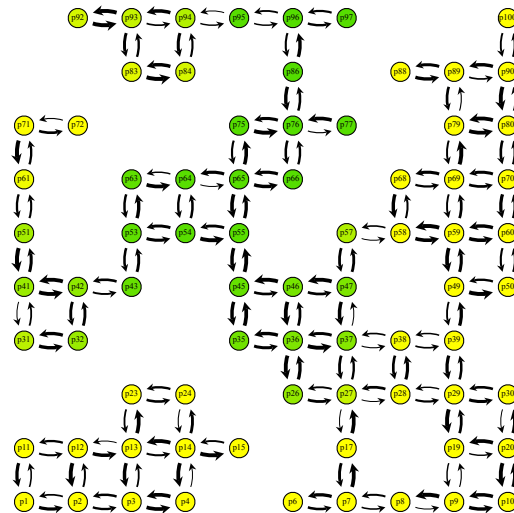
You might also notice that the arrows in the population visualization now vary in their thickness; this is because we now draw random migration rates using `runif()`, rather than using a fixed migration rate as we did in section 5.3.3. This code draws migration rates from a uniform distribution, and makes no attempt to ensure that migration rates between a given pair of subpopulations are symmetric. One could, of course, implement any pattern of migration, random or otherwise, that one wished.

That brings us to the final lines of the event:

```
// introduce a beneficial mutation
target_subpop = sample(sim.subpopulations, 1);
sample(target_subpop.genomes, 10).addNewDrawnMutation(m2, 20000);
```

This manual will cover introduced mutations and selective sweeps in great detail in chapter 9, so this is just a tiny bit of foreshadowing of that complex topic. The idea here is simple: we choose one subpopulation from the subpopulations that remain in the model using `sample()`, and then we choose ten genomes at random from the chosen subpopulation again using `sample()` (starting with ten rather than just one to make it more likely that the mutation will not be lost due to drift early on, for pedagogical purposes), and finally we call `addNewDrawnMutation()` to add a new `m2` mutation to those ten genomes at position 20000. This code, then, introduces a beneficial mutation that will, we hope, sweep through our sparse metapopulation.

And if it doesn't get lost, and if the metapopulation is connected, sweep it does. Here is a screenshot from midway through the sweep (in a different random run than was shown above):



As per the default behavior in SLiM, subpopulations are colored according to their mean fitness; populations where the sweep mutation is approaching fixation are thus a dark green, while populations that are still neutral are yellow. Note that the `configureDisplay()` method has a third (optional) argument that would allow us to customize the color of each subpopulation as well, coloring them according to whatever model variable we wish; we have not used that here, since the default fitness-based coloring is in fact exactly what we want.

### 5.3.5 Reading a migration matrix from a file

Sometimes, when modeling an empirical population, migration rates are specified in a file, and you would like to read that file in and create a corresponding SLiM model. This is very easy to accomplish using Eidos in SLiM. Let's start by looking at a very simple migration matrix file:

```
// For the recipe of section 5.3.5.
// Format: <src>, <dest>, <rate>.
1,1,0.78
1,2,0.10
1,3,0.12
2,1,0.01
2,2,0.96
2,3,0.03
3,1,0.33
3,2,0.17
3,3,0.50
```

This file expresses the model's migration rates as a series of lines, each of which is a series of comma-separated values; this format is often called a CSV file. Let us suppose that it exists on disk with the filename `migration.csv`. The first two lines are comments, describing the file. The rest of the lines each have three values: the identifier of the source subpopulation, the identifier of the destination subpopulation, and the migration rate (sometimes the destination is listed before the source in these sorts of files, so be careful). Note that there are lines expressing the fraction of each subpopulation that does *not* migrate – the remainder after all migrants have left. This is not optimal for SLiM's purposes, but we want to work with the file as it is.

We can read this file in and create a model based on it with a simple script:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  for (i in 1:3)
    sim.addSubpop(i, 1000);
  subpops = sim.subpopulations;

  lines = readFile("migration.csv");
  lines = lines[substr(lines, 0, 1) != "//"];

  for (line in lines)
  {
    fields = strsplit(line, ",");
    i = asInteger(fields[0]);
    j = asInteger(fields[1]);
    m = asFloat(fields[2]);

    if (i != j)
    {
      p_i = subpops[subpops.id == i];
      p_j = subpops[subpops.id == j];
      p_j.setMigrationRates(p_i, m);
    }
  }
}
10000 late() { sim.outputFixedMutations(); }
```

The tick 1 event is where the action is. It first creates the three subpopulations of the model with a simple for loop. It would be easy to extend this model to determine the number of subpopulations from the contents of the `migration.csv` file, and to read subpopulation sizes in from that file, and so forth. Here, however, we hard-code those values for simplicity. Once the subpopulations have been created, we cache the vector of subpopulations in `subpops` for brevity in the script that follows.

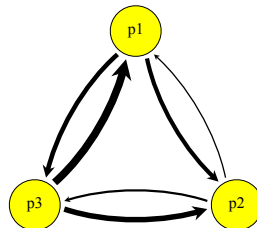
Next, the script reads in the contents of the `migration.csv` file using `readFile()`. This creates a string vector, with each line of the file being a separate element in the vector. The following line uses the `substr()` function to find lines that begin with `//`, and removes those lines by subsetting the `lines` vector, stripping out the comment lines from the file.

Now we can loop through the lines with a for loop and handle each line in turn. The `strsplit()` function is used to split line into its components, separated by commas; this is often a

very convenient way to parse CSV files (but using the `readCSV()` function is often even easier – see section 6.1.2 for an example). The three values of the line are then extracted from the `fields` vector, which is of type `string`, and are converted to their appropriate types.

The last block actually sets the migration rate. First it checks that `i` and `j` are not the same; this filters away the lines that express the non-migrating fractions, which SLiM does not need to know about. Then it looks up the subpopulations referenced by `i` and `j`, using the `id` property, which corresponds to the numeric part of the subpopulation’s symbol (i.e., subpopulation `p3` has an `id` of 3). Finally, it sets the migration rate from `p_j` to `p_i` to be the rate `m` that was read from the file.

When executed, SLiMgui’s visualization of this population structure looks like this:



That looks like what we would expect from looking at the file; `p2` is a sink, `p3` is a source, and `p1` is close to balanced. This is a very simple population model, but this script could just as easily read in a migration matrix file for hundreds or even thousands of subpopulations; its code is quite general. As mentioned above, it could also easily be extended to read subpopulation sizes from the CSV file as well; indeed, other subpopulation properties, such as selfing rates and sex ratios, could also easily be added to the file format and set up by this script, and the script could easily be adapted to work with whatever empirical data files already exist.

## 5.4 The Gravel et al. (2011) model of human evolution

In this section we will look at a recipe that brings together all of the elements of demography and population structure that have been discussed in this chapter. This is a SLiM implementation of a model of human evolution presented by Gravel et al. (2011); in particular, we here model the “Low-coverage + exons” model described in their Table 2. The recipe:

```

initialize() {
  initializeMutationRate(2.36e-8);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 9999);
  initializeRecombinationRate(1e-8);
}

// INITIALIZE the ancestral African population
1 early() { sim.addSubpop("p1", asInteger(round(7310.370867595234))); }

// END BURN-IN; EXPAND the African population
73105 early() { p1.setSubpopulationSize(asInteger(round(14474.54608753566))); }

// SPLIT Eurasians (p2) from Africans (p1) and SET UP MIGRATION
76968 early() {
  sim.addSubpopSplit("p2", asInteger(round(1861.288190027689)), p1);
  p1.setMigrationRates(c(p2), c(15.24422112e-5));
  p2.setMigrationRates(c(p1), c(15.24422112e-5));
}

```

```

// SPLIT p2 into European (p2) and East Asian (p3); RESIZE; MIGRATION
78084 early() {
    sim.addSubpopSplit("p3", asInteger(round(553.8181989)), p2);
    p2.setSubpopulationSize(asInteger(round(1032.1046957333444)));

    p1.setMigrationRates(c(p2, p3), c(2.54332678e-5, 0.7770583877e-5));
    p2.setMigrationRates(c(p1, p3), c(2.54332678e-5, 3.115817913e-5));
    p3.setMigrationRates(c(p1, p2), c(0.7770583877e-5, 3.115817913e-5));
}

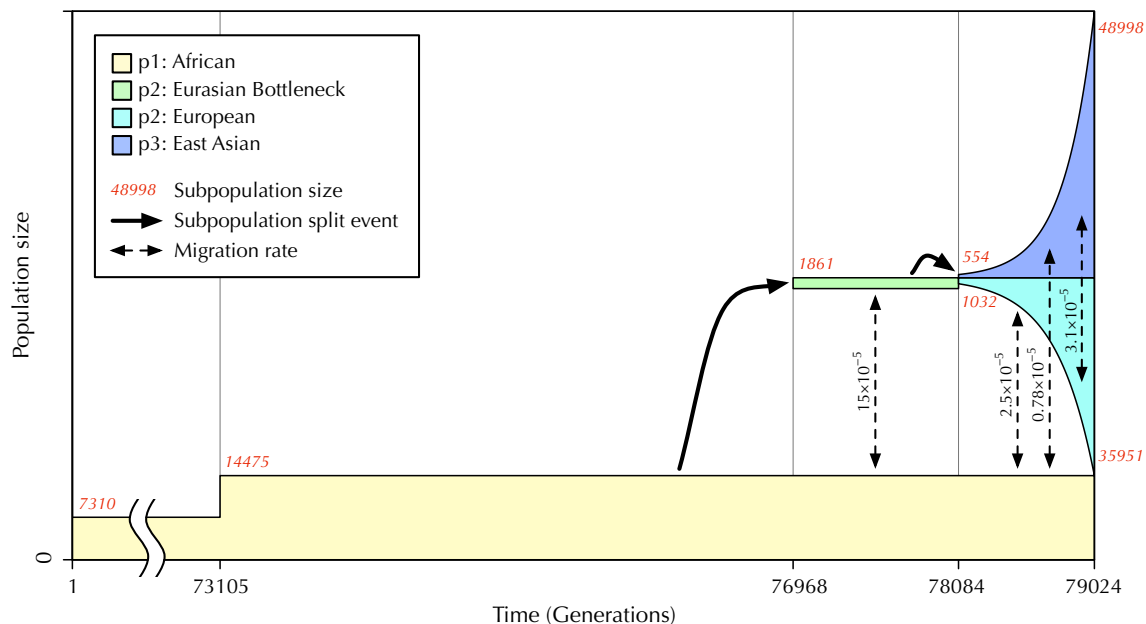
// SET UP EXPONENTIAL GROWTH in Europe (p2) and East Asia (p3)
78084:79024 early() {
    t = sim.cycle - 78084;
    p2_size = round(1032.1046957333444 * (1 + 0.003784324268)^t);
    p3_size = round(553.8181989 * (1 + 0.004780219543)^t);

    p2.setSubpopulationSize(asInteger(p2_size));
    p3.setSubpopulationSize(asInteger(p3_size));
}

// OUTPUT AND TERMINATE
79024 late() {
    p1.outputSample(216); // YRI phase 3 diploid sample of size 108
    p2.outputSample(198); // CEU phase 3 diploid sample of size 99
    p3.outputSample(206); // CHB phase 3 diploid sample of size 103
}

```

Three subpopulations are modeled in this recipe, as shown in the diagram below (solid arrows show subpopulation splitting events, dotted arrows show migration rates between subpopulations, and red italic numbers show subpopulation effective sizes). Note that, while exact parameter values from the original model are used in the recipe (Simon Gravel, pers. comm.), rounded values are reported in the Gravel et al. (2011) manuscript. The full version of this recipe, available online and in SLiMgui, provides extensive comments regarding the rounding of values and the timing of simulation events; those comments are not shown here for brevity.





The first subpopulation, present from the beginning of the simulation, is `p1`; it represents the African population YRI (Yoruba in Ibadan, Nigeria). The second, `p2`, initially represents the ancestral Eurasian bottleneck, and then becomes the European population CEU (Utah residents with Northern and Western European ancestry). The third, `p3`, represents the East Asian population CHB (Han Chinese in Beijing, China). As you can see in the diagram and the recipe, `p2` splits from `p1` at tick 76968, and then `p3` splits off from `p2` at tick 78084. Beginning with the split of `p3` from `p2`, both `p2` and `p3` undergo exponential growth until the end of the model.

The model begins 79024 generations ago (~1.98 million years if we assume 25 years per generation), which is also 79024 ticks ago (given non-overlapping generations). It spends quite a long time doing neutral burn-in – 73104 ticks, to be exact, calculated as ten times the ancestral population size (see section 17.6). Then the action really starts in tick 73105 with the expansion of the African subpopulation. Running the full model only takes a couple of minutes, but if you're impatient, you can change the tick 1 event that creates `p1` to run in tick 73000 instead, providing an 105-tick burn-in that should suffice for illustrative purposes (but will not produce the correct pattern of neutral diversity at the end of the run). Note that the final population sizes, at the end of the run, may not exactly match the values given in Gravel et al. (2011) due to roundoff error.

This model is a neutral model; the only mutation type modeled is `m1`, which is neutral. At the end of the model, random samples are output from each of the three subpopulations to provide a view of the neutral diversity present in each subpopulation. The empirical samples that this output is intended to match were taken from (diploid) humans; the `outputSample()` method of `Subpopulation`, on the other hand, takes as its argument the number of (haploid) genomes to sample and output. The sample sizes in SLiM are thus double the empirical sample sizes.

A frequently asked question is: how can one produce output encompassing particular genomes that have been sampled from multiple subpopulations? The `outputSample()` method used in this recipe is a method of `Subpopulation`, and therefore takes its sample from the single subpopulation that is the target of the method. But there are also output methods on `Genome` that can be called on any vector of genomes, obtained from any source (see section 24.4.2). For example, we could rewrite the final output event for this recipe as follows:

```
79024 late() {
    sample1 = p1.sampleIndividuals(108).genomes; // YRI
    sample2 = sample(p2.individuals, 99).genomes; // CEU
    sample3 = sample(p3.genomes, 206); // CHB
    c(sample1, sample2, sample3).output();
}
```

This snippet shows three different ways to take a sample from a population: with the `sampleIndividuals()` method of `Subpopulation` (see section 24.15.2), with the `sample()` function applied to the vector of individuals in a subpopulation, and with `sample()` applied to the vector of genomes in a subpopulation. Note that in the first two cases individuals are sampled and then the `genomes` property of `Individual` is used to get the two genomes of each sampled individual, so these samples consist of diploid pairs of genomes from individuals, whereas in the third case the genomes of the subpopulation are sampled directly, so this sample will consist of unassociated haploid genomes. This is why the sample size in the first two cases is half what it was before; we end up with two genomes for each sampled individual. Given these three vectors of genomes, we use `c()` to construct a single vector of genomes from them, and then call `output()` to produce output from that vector. `Genome` also has `outputVCF()` and `outputMS()` methods (see section 24.4.2), so any style of output supported by SLiM can be produced in this manner.

It is worth noting that the population sizes used in this model are effective population sizes. The actual population sizes in human history were likely much larger, but geography and other

factors greatly reduced the effective population size. This is also the reason that the sizes of p2 and p3 post-split are not modeled as adding up to the same size as the pre-split p2 subpopulation. For discussion of effective population size versus census population size, see section 9.11.

It is also worth noting that this model was estimated using a mutation rate of  $2.36 \times 10^{-8}$  per base position per gamete. As this rate is somewhat higher than what has been estimated by more recent studies of human parent/offspring ‘trios’, it might be appropriate to utilize some of the rescaling techniques discussed in the next section for certain applications. For example, to simulate evolution with a mutation rate of  $1.2 \times 10^{-8}$  per base pair per gamete, Amorim et al. (2017) multiply branch lengths (generations) and population sizes by a scaling factor of  $1.97$  ( $2.36 \times 10^{-8} / 1.2 \times 10^{-8}$ ). Additionally, a more recent version of the Gravel model has been estimated by Jouganous et al. (2017) using a generation time of 29 years and mutation rate of  $1.44 \times 10^{-8}$  per base position per gamete, meant to reflect synonymous sites in protein-coding genes. Rescaling is necessary, rather than simply changing the mutation rate used in the model, because if a lower mutation rate is used with the same population sizes and event times, a lower level of genetic diversity will result, no longer fitting empirical patterns of genetic diversity (see section 5.5).

The model for this recipe was written by Aaron Sams, formerly of the Messer Lab at Cornell, and revised by Chase W. Nelson in SLiM 3.5. Incidentally, if you are interested in exploring other human demographic models, and a wide variety of other standard population genetic models, the `stdpopsim` software package may be of interest. It uses both SLiM and `msprime` as “back ends”, including leveraging tree-sequence recording for efficient burn-in and analysis, and can be very useful for reproducing and exploring standard models. See section 1.10 for discussion and links.

Before moving on, let’s wander off on a little digression, because this model is the longest SLiM model we’ve seen yet. If you open it in SLiMgui, you may not be able to see all of its code unless you make the model window fairly large. If you use SLiM for very long, your models may be even longer and more complex than this recipe, and when you get to that point, scrolling around trying to find the event you want can be confusing and annoying. There is a need for some kind of organizational tool – and happily, SLiMgui provides such a tool.

For illustration purposes, here’s a simplified Gravel model script in which all of the old comments in the code have been deleted – but there are a few new comments:

```
/// # Gravel Model in SLiM
/// ####_(with Jump Menu annotations)_
///

initialize() {
  initializeMutationRate(2.36e-8);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 9999);
  initializeRecombinationRate(1e-8);
}

///
/// **Demography:**

1 early() /* create p1 */ {
  sim.addSubpop("p1", asInteger(round(7310.370867595234)));
}

73105 early() /* end burn-in */ {
  p1.setSubpopulationSize(asInteger(round(14474.54608753566)));
}
```

```

76968 early() /* split p2 from p1 */ {
    sim.addSubpopSplit("p2", asInteger(round(1861.288190027689)), p1);
    p1.setMigrationRates(c(p2), c(15.24422112e-5));
    p2.setMigrationRates(c(p1), c(15.24422112e-5));
}

78084 early() /* split p3 from p2 */ {
    sim.addSubpopSplit("p3", asInteger(round(553.8181989)), p2);
    p2.setSubpopulationSize(asInteger(round(1032.1046957333444)));

    p1.setMigrationRates(c(p2, p3), c(2.54332678e-5, 0.7770583877e-5));
    p2.setMigrationRates(c(p1, p3), c(2.54332678e-5, 3.115817913e-5));
    p3.setMigrationRates(c(p1, p2), c(0.7770583877e-5, 3.115817913e-5));
}

78084:79024 early() /* exponential growth */ {
    t = sim.cycle - 78084;
    p2_size = round(1032.1046957333444 * (1 + 0.003784324268)^t);
    p3_size = round(553.8181989 * (1 + 0.004780219543)^t);

    p2.setSubpopulationSize(asInteger(p2_size));
    p3.setSubpopulationSize(asInteger(p3_size));
}

/**/
/** **Final output:** */

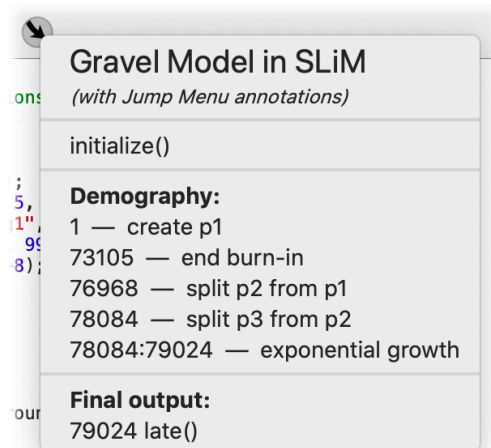
79024 late() {
    p1.outputSample(216);
    p2.outputSample(198);
    p3.outputSample(206);
}

```

These special comments annotate the script in a way that SLiMgui can understand. (This modified script is available as “Recipe 5.4 - The Gravel et al. (2011) model of human evolution II” in SLiMgui’s Open Recipe menu.) To see the point of this, click on the Jump button at the top of the scripting area:



The button pops up a menu like this:



And now you can see the point of this. Each item in this menu will jump you to the corresponding spot in your code – to one of the events you have defined (or callbacks, or user-defined functions), or to one of the bold header comments you have put in the code. All of the labels come from the comments present in the code. Most code comments do not appear in the Jump menu; there are three specific kinds of comments that *do* appear.

First of all, there are comments that add a label to an event, callback, or user-defined function. These must appear on the same line as the declaration of the script block, before the starting brace for the block. These code formatting styles would all produce the same result shown above:

```
1 early() /* create p1 */ {
    sim.addSubpop("p1", asInteger(round(7310.370867595234)));
}

1 early() /* create p1 */
{
    sim.addSubpop("p1", asInteger(round(7310.370867595234)));
}

1 early() // create p1
{
    sim.addSubpop("p1", asInteger(round(7310.370867595234)));
}
```

So either a `/* */` comment or a `//` comment may be used, and the opening brace can be on the same line or the next line, as you prefer.

Second, there are comments that add a header line, like the **Demography:** header above. These use a conventional comment format sometimes called a “Doxygen” comment, after the popular code-documentation tool. The format involves an extra `/` at the start of a `//` comment, or an extra `*` at the start of a `/* */` comment, like these comments in the script above:

```
/// **Demography:**
/** **Final output:** */
```

When SLiMgui sees a comment in this format, it knows that it is a Jump menu header item. These comments have some other extra asterisks, too, highlighted in red here:

```
/// **Demography:**
/** **Final output:** */
```

These asterisks are the reason that these header lines appear in bold. They are so-called “Markdown” annotations, telling SLiMgui that the text inside the double asterisks should be shown in bold. You can read more about it on Wikipedia at <https://en.wikipedia.org/wiki/Markdown>, and there are tons of other online resources about it, but really it’s extremely simple – especially since SLiMgui supports only a small subset of Markdown annotations. Basically, you can put text in italics by surrounding it with single underscores or single asterisks, like `_italic_` or `*italic*`; you can put it in bold with double underscores or double asterisks, like `__bold__` or `**bold**`; you can do both by doing both, like `***bold-italic***` or `__*bold-italic*__` or similar variants; and you can change the font size with header annotations from large, like `# H1`, to small, like `##### H6`, using 1–6 hash marks, followed by a space (required), followed by your label text.

Now we can understand the comments at the top of the file, which use Markdown annotations to produce the nice-looking header at the top of the Jump menu. Let’s look at them again, with the Markdown annotations shown in red:

```

/// # Gravel Model in SLiM
/// ##### _(with Jump Menu annotations)_

```

The first comment uses a single hash mark (#), followed by the required space, to make the line into a top-level (H1) header using the largest font size.

The second comment uses four hash marks (####) to make the line into a subhead (H4), resulting in a font size slightly smaller than normal; in SLiMgui, the default font size in the Jump menu corresponds to header level H3 (###). This line also uses single underscores (\_) around the label text to make it italic.

Note that SLiMgui's support for Markdown is only rudimentary. You might want to do something like this, for example (Markdown annotations again in red):

```

/// _italics here_ and **bold here**

```

This does not work; the entire label must be in a single style. The Markdown annotations for things such as links, images, bullet lists, etc., are also not supported. Markdown annotations also work only in standalone header comments like these; they do not work in comments on script blocks (the first category of formatted comments we looked at above). (These limitations are imposed by what the Qt widget kit, used by SLiMgui, allows in menu items.)

Finally, there are comments that add a separator line in the Jump menu, like these comments in the script above:

```

///
/****/

```

Separator lines must be exactly the two styles above; variants will not work. All other comments in your code will be ignored by the Jump menu, so you can (and should!) continue to comment your code as usual.

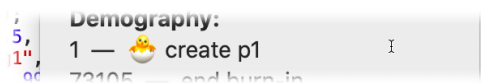
When writing your own scripts, you can use these special comments to customize the Jump menu in SLiMgui to suit your preferred workflow. Oh, and you can even put emojis in your comments; they will appear in the Jump menu, which can be a very quick, visual way to mark your most commonly used script blocks! For example, you might add a hatching chick emoji to your event that creates the initial subpopulation, like this:

```

1 early() /* 🐣 create p1 */ {

```

That would appear like this in the Jump menu:



That might seem silly; but humans are extremely visual, and can find visual patterns much faster than text patterns. If you mark the events that you most commonly hunt for in your script with appropriate emojis, you will find them much more easily, even if you do feel a bit silly doing so!

## 5.5 Rescaling population sizes to improve simulation performance

The limiting factor in most forward population genetic simulations tends to be the actual number of individuals simulated. In SLiM, every individual in the population is modeled explicitly. Thus, in larger populations more time is consumed per generation creating the children that make up the population, and more memory is needed for storing their genetic information.

Perhaps we can approximate the evolution of a large population using a simulation of a smaller population? In some ways, we can: analytical theory predicts that under certain assumptions many important population-genetic parameters, such as the expected levels of diversity, polymorphism frequency spectra, levels of linkage disequilibrium, etc., should primarily depend on products of the form  $N\mu$ ,  $Nr$ , and  $Ns$ , where  $N$  here is the effective population size (often written  $N_e$ ),  $\mu$  is the mutation rate,  $r$  the recombination rate, and  $s$  the selection coefficient of a given mutation.

Thus, for many analyses we do not have to simulate a population of the true size, but can obtain similar results by simulating a much smaller population while rescaling  $\mu$ ,  $r$ , and  $s$  such that the products  $N\mu$ ,  $Nr$ , and  $Ns$  remain the same. Importantly, when rescaling population sizes, time has to be rescaled as well, because drift will be faster in a smaller population. The amount of drift in a Wright–Fisher population of 1000 individuals, observed over 100 generations, will be similar to that in a population of 100 individuals observed over just 10 generations. In this way, rescaling a simulation to a smaller population size not only helps us by having to model fewer individuals per generation, but also by reducing the number of generations we need to simulate.

This rescaling “trick” can be tremendously helpful in practice, allowing the simulation of scenarios featuring large populations that would not be feasible to simulate more directly. However, there are clear limitations to rescaling as well. Most obviously, there will be limits to how far downscaling of population sizes can be pushed in the light of an increasing impact of discretization effects: as simulated population sizes become smaller, there will be fewer possible population frequencies at which mutations can segregate, with the lowest possible frequency being  $1/(2N)$ . Thus, the rescaling approach will break down if the goal is to study the characteristics of very low-frequency polymorphisms.

The discreteness of generations can increasingly become a problem as time is rescaled downwards. For example, if a selective sweep that takes 100 generations in a population of 10000 individuals takes the same amount of “time” in a downscaled population of 500 individuals, it would complete in only 5 generations (assuming selection coefficients were rescaled upwards accordingly). In that case, the frequency changes of the selected allele will no longer be small over the timescale of a single generation, violating a key assumption in many analytical models. Furthermore, since the time for a beneficial allele to sweep scales with  $\log(N)$ , we actually expect the sweep in the smaller population to complete even *faster* than we would expect after accounting for rescaling.

This discretization can also have unexpected and undesirable side effects on processes such as adaptation. Since  $N\mu = (N/Q)\mu Q$ , rescaling by a factor  $Q$  preserves the influx of mutations per generation. One rescaled generation is  $Q$  original generations, so this implies a lower influx of mutations per unit of time, but since mean TMRCA scales with  $N$ , genetic diversity at neutral sites is preserved. Since the probability of fixation of a beneficial mutation scales with  $s$ , and since  $N\mu s dt = (N/Q)\mu Q s Q(dt/Q)$ , the influx of successfully established beneficial mutations per unit time is also preserved, implying a smaller number of fixed selected mutations since the mean TMRCA in the population. Since rescaled values of  $s$  are larger, this has the net effect of substituting many mutations of small effect with a single one of large effect (with  $Q = 100$ , replacing 100 mutations of  $s = 0.001$  by a single one of  $s = 0.1$ ), a very different scenario.

While it may be tempting to always rescale any given scenario to a very small population size, then, one must be careful that finite-size effects do not distort results too much. It is therefore generally advisable to test any rescaled scenario at larger and smaller sizes in order to make sure that the results are consistent. When tree-sequence recording can be used to speed up a simulation, by allowing simulation of neutral mutations to be deferred (see section 17.2), or even by allowing the coalescent to be used for neutral burn-in with recapitation (see section 17.10), that



will usually be strongly preferable, since tree-sequence recording does not introduce such artifacts. Nevertheless, since tree-sequence recording may not be applicable to a given model, or may provide insufficient performance enhancement, rescaling is still sometimes needed. Rescaling may be used in conjunction with tree-sequence recording, with the appropriate adjustment of parameters such as  $\mu$  and  $r$  for mutation overlay and recapitation.

As an example of the use of rescaling, consider the following recipe, in which we model neutral and deleterious mutations occurring over a 10 kb locus in a population of initial size 5000. In tick 50000, the population experiences a bottleneck that lasts for 5000 generations. A random population sample is then taken in tick 60000:

```
initialize() {
  initializeMutationRate(1e-8);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", -0.01);
  initializeGenomicElementType("g1", c(m1,m2), c(0.8,0.2));
  initializeGenomicElement(g1, 0, 9999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 5000); }
50000 early() { p1.setSubpopulationSize(1000); }
55000 early() { p1.setSubpopulationSize(5000); }
60000 late() { p1.outputSample(10); }
```

The patterns of diversity observed in the population sample retrieved at the end of the simulation should be very similar to those obtained from the following recipe, in which we downscaled the population size by a factor of ten:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", -0.1);
  initializeGenomicElementType("g1", c(m1,m2), c(0.8,0.2));
  initializeGenomicElement(g1, 0, 9999);
  initializeRecombinationRate(1e-7);
}
1 early() { sim.addSubpop("p1", 500); }
5000 early() { p1.setSubpopulationSize(100); }
5500 early() { p1.setSubpopulationSize(500); }
6000 late() { p1.outputSample(10); }
```

Note how the population sizes, times, mutation rates, recombination rates, and selection coefficients were all rescaled in this scenario. On a 2.26 GHz Intel Xeon Mac Pro running the models in SLiMgui, the first recipe runs in 218 seconds, whereas the second runs in 12 seconds – quite a significant difference.

One caveat is that if the original model uses very high recombination rates, those rates should not be scaled in the simple multiplicative fashion described above. In particular, consider that a recombination rate of 0.5 represents completely independent assortment from one base to the next, due to a probability of crossover between bases of 0.5 (see section 24.1's documentation of `initializeRecombinationRate()`). If a model uses a rate of 0.5 between sites, a rescaled version of that model would still use a rate of 0.5 between those sites, because completely independent assortment can't get *more* independent; indeed, if the rescaling factor were, e.g., 10, then multiplying the original recombination rate by the rescaling factor would result in a nonsensical scaled rate of 5.0 that would not even be interpretable as a probability. In point of fact, the simple multiplication of rates when rescaling a model is always just an approximation, but for rates less

than 0.001 and rescaling factors of 10.0 or less it will be such a close approximation that the difference shouldn't matter. The correct formula for rescaling of recombination rates in SLiM, that gives the rescaled, per-locus recombination rate  $r_{scaled}$  corresponding to an original per-locus recombination rate of  $r$  with a rescaling factor of  $n$ , is (thanks to Peter Ralph):

$$r_{scaled} = \frac{1}{2}(1 - (1 - 2r)^n)$$

For small values of  $r$  this produces an essentially linear scaling by  $n$ , but as  $r$  approaches 0.5 the scaling saturates in the desired manner. For an original rate of 0.001 and a rescaling factor  $n$  of 10, this suggests a rescaled recombination rate of 0.00991, which is only very slightly lower than the rate of 0.01 produced by naive multiplication. If the original rate were 0.01, however, the rescaled rate would be 0.0915, almost 10% off from the rate of 0.1 provided by naive multiplication.

This formula is based upon the probability that a binomial draw will be odd; as a region of length  $n$  squeezes down to a single site, the important question for rescaling the recombination rate is the probability that the original region of length  $n$  would have an even number of recombination events (cancelling out to produce no effect, once rescaled) or an odd number of recombination events (producing the same effect as a single crossover, once rescaled). In practice, however, if you are rescaling a model in a parameter regime where the effects of this formula matter to your results (besides the fact that a rate of 0.5 should stay 0.5 to produce independence), you may be pushing the limits of safe rescaling, and should proceed with extreme caution. See section 14.13 for a somewhat unusual application of this formula to scaling one particular region of the chromosome in a model.

For further discussion of model rescaling, Uricchio & Hernandez (2014) may be of interest, and citations therein.

## 6. Sexual reproduction

The standard model of reproduction in SLiM assumes (1) hermaphroditic diploid individuals, (2) reproduction by biparental sexual mating, (3) a uniform rate of crossing over without gene conversion during recombination, and (4) modeling of an autosome rather than a sex chromosome. In this chapter, we will explore how each of these assumptions can be modified to study various other scenarios of reproduction. The only limitations are that SLiM remains restricted to diploid organisms (but haploids and polyploids can be simulated with some creative scripting; see section 14.9), hermaphroditism or two-sex systems, and X–Y sex chromosome systems.

### 6.1 Recombination

#### 6.1.1 Crossing over: Making a random recombination map

Section 4.1.6 introduced the `initializeRecombinationRate()` method in a basic neutral simulation. That section also discussed the possibility of supplying different recombination rates for different stretches of the chromosome, since `initializeRecombinationRate()` takes a vector of rates and a vector of end positions.

Here, then, let's examine a recipe for setting up random variation in the recombination rate along a chromosome, to simulate the presence of “hot spots” and “cold spots”:

```
initialize() {  
  initializeMutationRate(1e-7);  
  initializeMutationType("m1", 0.5, "f", 0.0);  
  initializeGenomicElementType("g1", m1, 1.0);  
  initializeGenomicElement(g1, 0, 99999);  
  
  // 1000 random recombination regions  
  ends = c(sort(sample(0:99998, 999)), 99999);  
  rates = runif(1000, 1e-9, 1e-7);  
  initializeRecombinationRate(rates, ends);  
}  
1 early() { sim.addSubpop("p1", 500); }  
10000 early() { sim.simulationFinished(); }
```

First of all, please note that this recipe is *not empirically based*; the choice of 1000 recombination regions, and the choice of uniform distribution from which the recombination rates are drawn, are both arbitrary. However, it would be straightforward to tailor this recipe to match whatever empirical information about recombination rates and regions one might have.


Let's look at how this code works. Everything new is in the `initialize()` callback, under the comment “// 1000 random recombination regions”. The `initializeRecombinationRates()` call sets all of the rates in one fell swoop, using a vector named `rates` that contains the recombination rates (in recombination events per base pair per gamete), and a vector named `ends` that contains the ends of the recombination regions (in base pairs along the chromosome). Each of these vectors has 1000 elements.

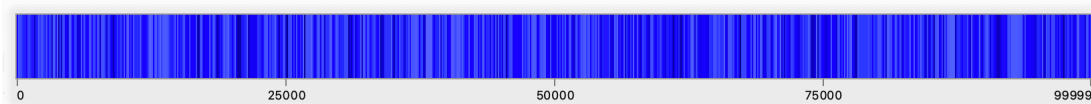
The previous script line creates the `rates` vector by calling the `runif()` function of Eidos. This function generates random draws from a specified uniform distribution. The first parameter requests 1000 samples; the next two parameters give the minimum and maximum values for the uniform distribution to be used. Note that Eidos has several other functions for drawing from other distributions, such as `rnorm()` for a normal distribution, `rpois()` for a Poisson distribution, `rbinom()` for a binomial distribution, and `rexp()` for an exponential distribution. Using these facilities, it is quite easy to make simulations based upon some sort of random configuration or behavior, as in this recipe.

Continuing to work backwards, the preceding script line sets up the vector of recombination region endpoints:

```
ends = c(sort(sample(0:99998, 999)), 99999);
```

Let's analyze this from the inside out. The innermost call is `sample(0:99998, 999)`. The Eidos function `sample()` returns a random sample from a given vector. The given vector here is the sequence `0:99998`, containing every base pair position along the chromosome except for the very last position (for reasons we will see momentarily). The second parameter requests 999 samples. The `sample()` function draws its samples without replacement by default, which is what we want; each recombination region should end at a different base pair. Next, the result from `sample()` is passed to the `sort()` function, which sorts the vector (because `initializeRecombinationRates()` requires the vector of end positions to be in sorted order). Finally, the `c()` function is used to concatenate the value `99999` onto the end of the vector, providing the final entry for the vector; this is the reason that only 999 samples were drawn, from positions up to only 99998. The last end position is required by `initializeRecombinationRates()` to be the last position in the chromosome.

If open this recipe in SLiMgui and do a Recycle and a Step, the random recombination map is loaded into the simulation. To see that it has worked, click the action button  to the right of the chromosome view, and select Display Rate Maps to turn it on (displaying both recombination and mutation rate maps if they exist – in this case, just the recombination rate map, since that is the one that has been set). The chromosome view should then show you something like this:



The regions shown in the darkest blues are cold spots, with low rates closer to  $10^{-9}$ , whereas the regions shown in lighter shades are hot spots, with high rates closer to  $10^{-7}$ . Remember that you can drag out a display range in the upper chromosome view, which changes what you see in the lower chromosome view – including the recombination map.

Note that SLiM also allows the recombination map to be tailored at an individual level using a `recombination()` callback; see sections 14.4 and 25.6. Also, note that a mutation rate map may be configured in exactly the same way as this recipe's recombination rate map, using `initializeMutationRate()`.

### 6.1.2 Crossing over: Reading a recombination map from a file

Rather than generating a random recombination map, you might want to read in and use an empirically determined recombination map such as the map from *Drosophila melanogaster* presented by Comeron et al. (2012) (dataset available in Fiston-Lavier & Petrov 2013). Let's work with chromosome arm 2L, which has a recombination map that looks like this:

```
1 0
100001 0
200001 0
300001 0.234550139
400001 0.234550139
500001 1.993676178
...
22800001 0.234550139
22900001 0
23000001 0
```

The length of the 2L arm is given as 23011544 bases; positions in SLiM will thus range from 0 to 23011543 (always beware off-by-one errors!). The file gives start positions in bases, beginning with 1, with rates in the second column in cM/Mbp (centimorgans per megabasepair). We have a little format conversion to do, but reading this file in and using it in SLiM is quite straightforward:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 23011543);

  // read Drosophila 2L map from Comeron et al. 2012
  map = readCSV("Comeron_100kb_chr2L.txt", c("ends", "rates"), sep="\t");
  ends = map.getValue("ends");
  ends = c(ends[1:(size(ends)-1)] - 2, 23011543);
  rates = map.getValue("rates") * 1e-8;
  initializeRecombinationRate(rates, ends);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
```

The action is in the `initialize()` callback after the comment, as before. The first line calls the `readCSV()` function of Eidos to read in a text file at the given filesystem path; you will need to change this path to the correct path to the file on your computer. The full details of `readCSV()` are documented in the Eidos manual, but its usage here is straightforward. After the filesystem path, the next parameter gives the names of the columns of the data file (since the file does not contain a header line). Finally, the named `sep` parameter tells `readCSV()` that values within each row of data are separated by a tab character, `"\t"`, rather than `readCSV()`'s default of comma separators. The result of `readCSV()` is a `DataFrame` object, with one column for each column of data in the file. Again, `DataFrame` is documented in the Eidos manual, but for our purposes here, we can simply fetch the values for the two columns using its `getValue()` method with the column names.

A brief digression. Before `readCSV()` was added to Eidos, this recipe read the contents of the file directly (as in section 5.3.5). The code for doing that may still be of interest:


```
lines = readFile("/Users/bhaller/Desktop/Comeron_100kb_chr2L.txt");
rates = NULL;
ends = NULL;

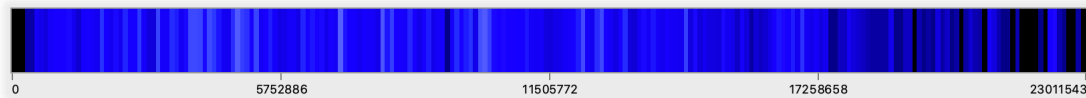
for (line in lines)
{
  components = strsplit(line, "\t");
  ends = c(ends, asInteger(components[0]));
  rates = c(rates, asFloat(components[1]));
}
```

This read the file as a vector of string lines, split them by tabs using `strsplit()`, and assembled the columns using `c()`. Using `readCSV()` and `DataFrame` is much easier!

We still need to convert the data into the format expected by SLiM. The input file specifies start positions, but we want end positions. The expression `ends[1:(size(ends)-1)]` thus strips off the first value, the start position 1, which is not needed. The `-2` correction does two things: (1) it shifts from starts to ends (each region ends at a position one less than the position where the next region starts), accounting for `-1`, and (2) it shifts from the 1-based system of the input file, in which the first base is at position 1, to the 0-based system of SLiM, with the first base is at position 0, for another `-1`. Finally, the `c()` call adds the last base position, 23011543, to the tail of the vector.

The rate conversion line just converts the recombination rates from cM/Mbp to a rate per base pair per gamete, as used by SLiM, by multiplying by  $10^{-8}$ , a conversion ratio that comes from the units involved. 1 cM means that there is a probability of 0.01 of crossover during meiosis. Therefore,  $1 \text{ cM/Mbp} = 10^{-6} \text{ cM/bp} = 10^{-8}$  probability of crossover per base pair.

If you open this recipe in SLiMgui, Recycle, Step, and turn on display of rate maps in the chromosome view with the  button to its right, as before, you should see something like this:



This is the *Drosophila melanogaster* 2L recombination map according to Comeron et al. (2012). Note that a mutation rate map could be read in and set up in SLiM in much the same manner.

### 6.1.3 Gene conversion

In addition to crossing over, recombination can be caused by gene conversion, which is – roughly speaking – the copying of a stretch of the genetic sequence from one chromosome to its homologous chromosome (Chen et al. 2007; Duret & Galtier 2009). By default gene conversion is not enabled in SLiM, but it can be turned on easily:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
    initializeGeneConversion(0.2, 500, 1.0);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
```

When `initializeGeneConversion()` is called, SLiM switches over to a “double-stranded break (DSB)” recombination model in which every recombination breakpoint is a double-stranded break that generates a gene conversion tract. The `initializeGeneConversion()` call has three required parameters (a fourth, governing biased gene conversion, is optional; see section 18.11). The first parameter, `nonCrossoverFraction`, is the fraction of recombination events that do not result in crossover (only in gene conversion); here we decide that 0.2 or 20% will be non-crossovers. The second, `meanLength`, is the mean length of each gene conversion tract, in base pairs; here we specify 500. When a gene conversion tract is constructed, the length of the tract is drawn by SLiM from a distribution with the specified mean length. Finally, `simpleConversionFraction`, the third parameter, specifies the fraction of gene conversion tracts that are “simple”, as opposed to “complex” tracts that involve heteroduplex mismatch repair. Section 1.5.6 provides a thorough discussion of the “DSB” recombination model and the meaning of these parameters.

SLiM does not support variation in the gene conversion parameters along the chromosome; however, a `recombination()` callback might allow a similar effect (see section 25.6).

### 6.1.4 Multiple chromosomes


At present, SLiM models only a single chromosome; there is no built-in support for multiple chromosomes in SLiM. However, modeling multiple chromosomes is nevertheless quite straightforward. A recombination rate of 0.5 between one base and the next means that a breakpoint will be generated at that position 50% of the time; this has the effect of dividing the simulated chromosome in two, such that the two halves of the chromosome assort independently

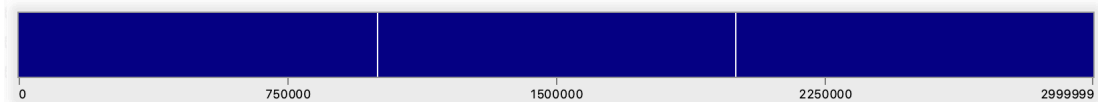


during meiosis. The end result is essentially to simulate two separate chromosomes. More than one such rate 0.5 point can be configured, and so any number of chromosomes can be modeled. The recipe for this is very simple:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", 0.1);
  initializeGenomicElementType("g1", c(m1,m2), c(10000,1));
  initializeGenomicElement(g1, 0, 2999999);

  rates = c(1e-9, 0.5, 1e-9, 0.5, 1e-9);
  ends = c(999999, 1000000, 1999999, 2000000, 2999999);
  initializeRecombinationRate(rates, ends);
}
1 early() { sim.addSubpop("p1", 500); }
10000 late() { sim.outputFixedMutations(); }
```

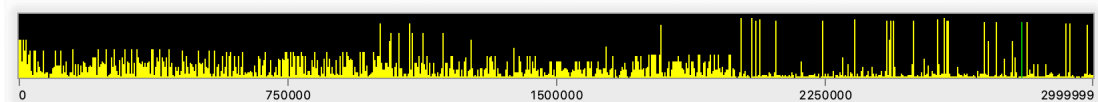
This sets up a chromosome with free recombination between bases 999999 and 1000000, and again between bases 1999999 and 2000000. The model is thus of three “effective chromosomes”, each of length 1e6. Turning on only the display of the recombination map in SLiMgui using the chromosome view’s  button, this can be seen visually:



The two white lines indicate points with a very high recombination rate; these are the rate 0.5 “joins” between the effective chromosomes.

This model also sets up two mutation types: m1, representing neutral mutations, and m2, representing beneficial mutations. Neutral mutations are 10,000 times more common than beneficial mutations, according to the proportions given to `initializeGenomicElementType()`.

If we run this model forward in SLiMgui (switching to viewing mutations rather than the recombination map), it is quite visually apparent that the three effective chromosomes are independent. Most obviously, whenever a beneficial mutation sweeps it carries along only the effective chromosome in which it resides. Here, for example, is a snapshot mid-sweep on the rightmost effective chromosome:



The mutation that is sweeping is visible as a green line fairly near the right edge of the chromosome. Neutral diversity within that chromosome is obviously hitchhiking along with the sweep, due both to the strong beneficial selection coefficient and the low recombination rate. The neutral diversity in the other two chromosomes is unaffected, and all three chromosomes show different patterns of neutral diversity because different lengths of time have elapsed since they last experienced a sweep.

Any number of autosomes can be simulated trivially with this method. Simulating autosomes together with sex chromosomes can also be done, although it is more complex; see section 14.5.

The way in which recombination map positions with a rate of **0.5** are handled internally in SLiM turns out to be rather subtle; section 14.13 has an extended discussion of this issue, but it is of no practical consequence for users of SLiM.

## 6.2 Separate sexes

### 6.2.1 Enabling separate sexes

Simulations in SLiM involve hermaphroditic individuals by default. If you wish to simulate separate sexes, however, doing so is essentially just the flip of a switch:

```
initialize() {  
    initializeMutationRate(1e-7);  
    initializeMutationType("m1", 0.5, "f", 0.0);  
    initializeGenomicElementType("g1", m1, 1.0);  
    initializeGenomicElement(g1, 0, 99999);  
    initializeRecombinationRate(1e-8);  
    initializeSex("A");  
}  
1 early() { sim.addSubpop("p1", 500); }  
10000 early() { sim.simulationFinished(); }
```

This recipe is identical to the basic neutral recipe from section 4.1, apart from the addition of a call to the function `initializeSex()`:

```
initializeSex("A");
```

The parameter here specifies the type of chromosome that is to be modeled; "A" specifies modeling of an autosome, but it is also possible to model the X or Y chromosome in SLiM (see section 6.2.3).

Once sex is turned on in a simulation, there is no way to turn it off; it is not possible to make a SLiM simulation in which individuals switch between being sexual and being hermaphroditic. That effect could easily be produced with a customized mate choice algorithm in a hermaphroditic model, however (see chapter 11).

Having turned sex on, all subpopulations will keep track of female and male individuals separately, and biparental matings will always involve a female parent and a male parent. A few things work a bit differently when sex is enabled; each subpopulation has a sex ratio that can be specified and modified (see section 6.2.2), for example, and selfing (see section 6.3.1) is not allowed when separate sexes are enabled. Usually you will know whether your code is running with sex enabled or disabled, but if you wish to write general-purpose code that works in either environment, the `sexEnabled` property of `Species` will tell you whether sex is presently enabled.

Section 4.2 discussed output of basic simulation state using the `outputFull()` and `outputSample()` methods of `Species`. When sex is enabled, the output generated by these methods changes slightly. In particular, the H symbol that was used to designate individuals as hermaphrodites will change to indicate the sex of each individual with M or F. In addition, if sex chromosomes are modeled (see section 6.2.3), the A that designated genomes in the output as autosomes will change to X or Y as appropriate. See chapter 26 for more on file output formats.

### 6.2.2 Sex ratios

When individuals are one of two sexes, rather than being hermaphroditic, the question of the sex ratio immediately arises. A sex ratio of **0.5** is maintained by default in SLiM; if that is what you want, no further action is required. To set a sex ratio (i.e., male fraction; see below) of **0.6**, on the

other hand, you simply supply an extra parameter to `addSubpop()` (or to `addSubpopSplit()`, which works in the same way):

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
  initializeSex("A");
}
1 early() { sim.addSubpop("p1", 500, 0.6); }
10000 early() { sim.simulationFinished(); }
```

If you open this recipe in SLiMgui, Recycle, and Step twice to get through tick 1, you can see in the population table view that this sex ratio has been used by SLiM:

ID	N	♂	♀	♂/♀	☉
p1	500	—	0.00	0.00	0.60

The column labeled ☉ shows the current sex ratio of each subpopulation; the subpopulations in the simulation do not all need to have the same sex ratio. The sex ratio of a subpopulation can be accessed in script through the `sexRatio` property of `Subpopulation`; in the recipe above, for example, `p1.sexRatio` would be equal to `0.6`.

The “sex ratio” in SLiM refers to the *male fraction*, the fraction of the total subpopulation that is male. Symbolically, it is therefore  $M/(M+F)$ , where  $M$  and  $F$  are the number of males and females respectively. A sex ratio of `0` would imply all females, a sex ratio of `1` would imply all males, and the default sex ratio of `0.5` is half males and half females.

You are free to set almost any sex ratio you wish; SLiM will raise an error, however, if the simulation is forced into a situation in which a subpopulation would become unisexual (whether due to the sex ratio, cloning rate, or other factors). All subpopulations must be viable, which means that at least one parent of each sex must be available to produce offspring.

It is possible for the sex ratio of a subpopulation to change over time. For example, here is a recipe for a simulation in which the sex ratio fluctuates randomly around `0.5`:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
  initializeSex("A");
}
1 early() { sim.addSubpop("p1", 500); }
1: early() { p1.setSexRatio(runif(1, 0.3, 0.7)); }
10000 early() { sim.simulationFinished(); }
```

The `setSexRatio()` method of `Subpopulation` simply takes a float representing the new sex ratio. As with `setSubpopulationSize()`, the change is not effected immediately; instead, the call sets the target sex ratio that will be used when offspring are generated. The recipe above draws a new random sex ratio between `0.3` and `0.7` in each tick using `runif()`.

### 6.2.3 Modeling sex-chromosome evolution

So far we have always seen simulations of autosomal evolution, but SLiM also supports simulation of sex chromosome evolution. The choice of chromosome type is made in the `initializeSex()` call, so to simulate X chromosome evolution one would simply do:

```
initialize() {  
    initializeMutationRate(1e-7);  
    initializeMutationType("m1", 0.5, "f", 0.0);  
    initializeGenomicElementType("g1", m1, 1.0);  
    initializeGenomicElement(g1, 0, 99999);  
    initializeRecombinationRate(1e-8);  
    initializeSex("X");  
}  
1 early() { sim.addSubpop("p1", 500); }  
10000 early() { sim.simulationFinished(); }
```

The "X" passed to `initializeSex()` sets SLiM to model the X chromosome. SLiM will handle all of the necessary details; females in the simulation will be XX, whereas males will be XY. Since the Y chromosome is not being modeled in this scenario, it will be a “null genome”, a placeholder kept by SLiM simply to make the diploid bookkeeping balance. Null genomes have no structure, receive no mutations, and raise an error if you attempt to do much of anything with them. It is also possible to supply "Y" to `initializeSex()`, to model the Y chromosome; in this case it is the X chromosome that will be a “null genome”. If you need to work directly with `Genome` objects in your script (we will see some examples of this later), you can find out whether a given genome is null or not with the `isNullGenome` property of `Genome`.

When modeling sex chromosomes, recombination works as you would expect it to. To be specific, if you are modeling the X chromosome, recombination will occur between the two X chromosomes of female parents when generating their gametes, whereas no recombination will occur between the X and Y chromosomes of a male. If you are modeling the Y chromosome, recombination does not occur, since the X chromosome in this case is a null chromosome, and individuals never possess two Y chromosomes.

When modeling the X chromosome, there are two different ways in which an individual can be heterozygous for a given mutation: the individual might be XX (i.e., female) and possess the mutation on just one X chromosome, or the individual might be XY (i.e., male) and possess the mutation on the one X chromosome present (technically called “hemizygous” rather than heterozygous). These cases are handled differently by SLiM when it calculates the fitness effect of a mutation. The first case is handled in the same way as when modeling an autosome: the dominance coefficient for the mutation type, as supplied to `initializeMutationType()`, specifies a multiplicative modifier for the selection coefficient of the mutation (see sections 1.5.2 and 4.1.3). The second case is handled using a special dominance coefficient, which SLiM calls the *haploid dominance coefficient* for lack of a better term; it covers both the haploid case, as we will see later, and the hemizygous case we see here. This coefficient is 1.0 by default, meaning that an XY individual possessing an X-linked mutation will experience the same fitness effect from that mutation as an XX individual that is homozygous for that mutation (a reasonable assumption because of X inactivation and dosage compensation, perhaps). The haploid dominance coefficient may be changed by setting its value on the mutation type (either during `initialize()` or later):

```
m1.haploidDominanceCoeff = 0.8;
```

Each mutation type has its own haploid dominance coefficient, allowing some degree of control, but (as with the regular dominance coefficient) it cannot be varied on a per-mutation

basis. Of course it would be possible to introduce a more complex fitness calculation scheme to achieve that using a `mutationEffect()` or `mutation()` callback, as discussed in section 10.6.

When modeling the Y chromosome, individuals can only possess zero or one copy of a given mutation, since males possess one Y and females possess none. For consistency, SLiM also uses the haploid dominance coefficient in this case. It would be unusual to change the coefficient in such a model, however, since Y-linked mutations are *always* hemizygous; you might as well just select the selection coefficient you want for them and be done with it.

It is worth noting that the way in which the Y chromosome is modeled in SLiM is in some ways similar to how one might model mitochondrial genomes. If SLiM is used in this way, the XY “males” are really the females, with a single genome possessed by all of the mitochondria in a given female. The XX “females” are really the males; they possess mitochondria too, of course, but they do not pass them down to their offspring, so their mitochondria are an evolutionary dead end that does not need to be modeled. This equivalence should work unless you need to model the fitness effects of mitochondrial mutations in males; that should also be possible in SLiM, using an autosomal model with some tricky scripting. We will not show such a model here, but section 14.5 provides a recipe that illustrates the sort of techniques that would be used.


## 6.3 Selfing and cloning

### 6.3.1 Selfing in hermaphroditic populations

Selfing, or self-fertilization, is the mating of an individual with itself: male and female gametes from one individual combine to form a zygote. Selfing is different from cloning in that gametes are produced and fertilize, so offspring are not clones of their parent; notably, recombination occurs in selfing. It is an essentially hermaphroditic phenomenon, since hermaphroditic individuals can produce both eggs and sperm (i.e., can produce both X and Y gametes). There are probably counterexamples somewhere in biology, where it sometimes seems that anything that is possible exists; but in SLiM, at least, selfing is limited to the hermaphroditic case.

Returning to a hermaphroditic model, then, selfing can be turned on with a single call:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
    p1.setSelfingRate(0.8);
}
10000 early() { sim.simulationFinished(); }
```

The `setSelfingRate()` method call here on `p1` tells SLiM that selfing should be used to generate 80% of the offspring in subpopulation `p1`. The selfing rate may be anything from `0.0` (the default) to `1.0`. The `setSelfingRate()` method may be called at any time, so the selfing rate can be varied over time or in response to other simulation conditions. The current selfing rate for a subpopulation is shown in SLiMgui’s population table view under the  column.

Usually it is not important, but it should be noted that in non-sexual simulations SLiM does not prevent a parent from being chosen twice, as both parents in a biparental mating event. Even when the selfing rate is set to `0`, therefore, a low background rate of selfing may occur. This can easily be prevented if necessary; see the recipe in section 12.4.

### 6.3.2 Cloning

SLiM also supports clonal reproduction, in which an offspring individual is an exact genetic copy of a single parent (except for new mutations introduced during the copying of the DNA). Indeed, hermaphroditic simulations may have a combination of biparental mating, self-fertilization, and cloning. In a hermaphroditic model, setting up clonal reproduction is a single call:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
    p1.setCloningRate(0.1);
}
10000 early() { sim.simulationFinished(); }
```

The `setCloningRate()` call here tells SLiM to generate 10% of the offspring in p1 clonally (the remaining 90% will be generated biparentally as usual). The cloning rate may be anything from 0.0 (the default) to 1.0. The `setCloningRate()` method may be called at any time, so the cloning rate can be varied over time or in response to other simulation conditions. The current cloning rate for each subpopulation is shown in SLiMgui's population table view under the ♂♀ and ♂♂ columns; in the hermaphroditic case the same number will be shown in both columns (the icon depicts a little Athena budding parthenogenically from the head of Zeus, if that is not immediately obvious).

Clonal reproduction is also supported in the case of separate sexes. It works in essentially the same way:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
    initializeSex("A");
}
1 early() {
    sim.addSubpop("p1", 500);
    p1.setCloningRate(c(0.5,0.0));
}
10000 early() { sim.simulationFinished(); }
```

The only difference is that when separate sexes are enabled by `initializeSex()`, the `setCloningRate()` method can take a vector of two rates, as above. Here the cloning rate is set to 0.5 for females, but 0.0 for males, reflecting a somewhat common situation in some taxa, in which females can reproduce parthenogenically but males can reproduce only sexually. You may still pass a singleton value to `setCloningRate()`; when separate sexes are enabled, that value is then taken as the cloning rate for both males and females. As before, the current cloning rates for the females and males in each subpopulation are shown in SLiMgui's population table view under the ♂♀ and ♂♂ columns.



## 7. Mutation types, genomic elements, and chromosome structure

A number of concepts such as mutation types, genomic element types, genomic elements, and chromosome structure were already introduced in chapter 4 using a basic neutral simulation. Here we return to those foundational concepts and explore them in more detail, showing how simulations might use multiple mutation types, multiple genomic element types, and many genomic elements to simulate more realistic chromosome structures with mutations of varying selective effects.

The structure of this chapter will be a bit different from that of previous chapters. In this chapter each subsection will build upon the recipe introduced by the previous subsection, working towards making a relatively large final recipe for a full-chromosome simulation.

### 7.1 Mutation types and fitness effects

In section 4.1.3 the concept of a mutation type was introduced: a category of mutations that represents some particular subset of the mutations in a simulation. Examples might include “neutral mutations”, “beneficial mutations introduced by the simulation script in tick 10”, or “mutations that will be forced to sweep to fixation”. Mutation types are represented in SLiM with the `MutationType` class, and each defined mutation type has a unique symbolic identifier of the form `mX`, like `m1` or `m27`. If you want to be able to generate or refer to a particular set of mutations separately from other mutations, you probably want to define a new mutation type. This can be true even if the mutation types are otherwise identical; you might use distinct but identical mutation types for “mutations I am tracking” versus “mutations I am not tracking”, for example.

Section 4.1.3 also introduced the function used to create new mutation types at initialization time, `initializeMutationType()`. This function can only be called inside an `initialize()` callback; in fact, it is not even defined at other times. Mutation types can therefore be set up only at initialization time; you must set up all of the types that your simulation will need for the entire run. After initialization, mutation types are typically static entities; however, there is a method, `setDistribution()`, that may be used to change a mutation type’s distribution of fitness effects, affecting new mutations generated from that point onward.

A mutation type is basically defined by two things: its dominance coefficient, and its distribution of fitness effects (DFE). Both of these properties are important only because they affect new mutations that are generated from a given mutation type: (1) each mutation of a given mutation type receives a selection coefficient drawn from the mutation type’s DFE, and (2) individuals that are heterozygous for a mutation will use a dominance coefficient, obtained from the mutation type of the mutation, to modify the selection coefficient of the mutation.


For the purposes of this recipe, let’s define four mutation types: two for neutral mutations (non-coding and synonymous), one for mildly deleterious mutations, and one for strongly beneficial mutations:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);           // non-coding
  initializeMutationType("m2", 0.5, "f", 0.0);           // synonymous
  initializeMutationType("m3", 0.1, "g", -0.03, 0.2);     // deleterious
  initializeMutationType("m4", 0.8, "e", 0.1);           // beneficial
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 5000); }
10000 early() { sim.simulationFinished(); }
```

The neutral mutation types, m1 and m2, are defined with a dominance coefficient of 0.5 (which does not matter for neutral mutations), and mutations drawn from them receive a fixed selection coefficient (DFE type "f") of 0.0 as specified by the DFE parameter. Note that they are exactly identical in their parameters; however, they will be used to represent different conceptual types of neutral mutations, with m1 representing neutral mutations in non-coding regions and m2 representing synonymous mutations in coding regions. Drawing this distinction will allow us to distinguish these two classes of mutations later on, when we are observing the simulation running.

The deleterious mutation type, m3, has a dominance coefficient of only 0.1, meaning that heterozygous individuals feel very little effect from these mutations; they are almost recessive. These mutations are drawn from a gamma DFE (see section 24.11) with a mean of -0.030, with a shape parameter (alpha) of 0.2.

Finally, the beneficial mutation type, m4, has a dominance coefficient of 0.8, representing incomplete dominance. Mutations of this type are drawn from an exponential DFE (again, see section 24.11) with a mean of 0.1.

All defined mutation types can be viewed in the drawer on the simulation window. If you open this recipe in SLiMgui, Recycle, and Step, then open the drawer by pressing the  button, you can see the mutation types listed, which can be quite useful if mutation types are manufactured *en masse* with a for loop or similar automated construction method:

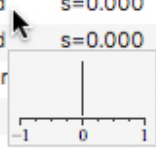
**Mutation Types:**

ID	h	DFE	Params
m1	0.500	fixed	s=0.000
m2	0.500	fixed	s=0.000
m3	0.100	gamma	s=-0.030, α=0.200
m4	0.800	exp	s=0.100

We are not using the new mutation types yet, just defining them; we'll make more progress with this recipe in the next section. Before we move on to that, however, there is one hidden feature worth mentioning. The mutation type table that is depicted in the screenshot above has a nice hidden addition: tooltips that show the mutation type's DFE graphically. If you place the mouse cursor over the m1 line without moving it for a second or a bit more (often called a "hover" of the cursor), a "tooltip" – a little informational tab – will appear:

**Mutation Types:**

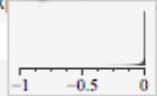
ID	h	DFE	Params
m1	0.500	fixed	s=0.000
m2	0.500	fixed	s=0.000
m3	0.100	gamma	s=-0.030, α=...
m4	0.800	exp	s=0.100



(Ignore the somewhat different appearance of the two screenshots above; they were taken on different versions of macOS, and Apple changed the standard system font and other aspects of table appearance from one macOS release to the next.) The tooltip shown above contains a plot of the mutation type's distribution of fitness effects; the x-axis is the selection coefficient, and the y-axis is the distribution's relative density. In this case, since the mutation type has a fixed selection coefficient of 0.0, the distribution consists of a single peak at 0.0. Hover over m3, and a more interesting tooltip appears:

### Mutation Types:

ID	h	DFE	Params
m1	0.500	fixed	s=0.000
m2	0.500	fixed	s=0.000
m3	0.100	gamma	$\bar{s}=-0.030$ , $\alpha=...$
m4	0.800	exp	$\bar{s}=0.100$

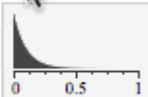


This plot shows the specific gamma distribution specified by the parameters for m3; note that the x-axis now spans the range  $-1$  to  $0$  only, since this DFE does not include any positive selection coefficients. The distribution shown has most of its density very close to zero, but a tail is visible extending downward perhaps as far as  $-0.25$  (well, it extends to  $-\infty$ , of course).

Hovering over m4 shows its DFE:

### Mutation Types:

ID	h	DFE	Params
m1	0.500	fixed	s=0.000
m2	0.500	fixed	s=0.000
m3	0.100	gamma	$\bar{s}=-0.030$ , $\alpha=...$
m4	0.800	exp	$\bar{s}=0.100$



This is a non-negative DFE, so the x-axis spans  $0$  to  $1$ , and this distribution is much broader. This visualization capability can prove quite helpful in configuring and debugging DFEs.

## 7.2 Genomic element types

Now that we have a couple of mutation types defined, we can make some genomic element types that use these mutation types. Genomic element types were introduced in section 4.1.4; they represent a type of region in the genome with a particular mutational profile. For this simple toy model, let's focus on exons, introns, and non-coding regions:




```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);           // non-coding
  initializeMutationType("m2", 0.5, "f", 0.0);           // synonymous
  initializeMutationType("m3", 0.1, "g", -0.03, 0.2);     // deleterious
  initializeMutationType("m4", 0.8, "e", 0.1);           // beneficial
  initializeGenomicElementType("g1", c(m2,m3,m4), c(2,8,0.1)); // exon
  initializeGenomicElementType("g2", c(m1,m3), c(9,1)); // intron
  initializeGenomicElementType("g3", c(m1), 1);         // non-coding
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 5000); }
10000 early() { sim.simulationFinished(); }
```

Genomic element type `g1` defines exons, which often suffer deleterious mutations, sometimes get neutral (synonymous) mutations, and very rarely get beneficial mutations. Type `g2` defines introns, which often get neutral (non-coding) mutations, and occasionally get deleterious mutations. Type `g3` defines non-coding regions, which get neutral (non-coding) mutations only. Each genomic element type is defined with a call to `initializeGenomicElementType()`, as previously discussed in section 4.1.4; its parameters supply the identifier for the new type, a vector of mutation types, and a vector of relative proportions for those mutation types among the new mutations that occur in the genomic elements of this type.

Notice that there is *not* a one-to-one correspondence between mutation types and genomic element types; this is typical, since the two classes represent quite different objects. A genomic element type often draws from a variety of different mutation types, with various probabilities. If this model were expanded to be more biologically realistic, it might have quite a few more genomic element types (3' and 5' UTRs, promoters and enhancers and silencers, various different types of non-coding regions...) but might have only a couple more mutation types (a strongly deleterious mutation type, to represent things like premature stop codons and broken promoters, for example).

Again, this recipe is not yet complete; now that we have defined genomic element types, we need to make genomic elements that use those types. However, at this point we can Recycle, Step, and observe the defined genomic element types in the simulation window's drawer:

**Genomic Element Types:**

ID	Color	Mutation types
g1		m2=2.000, m3=7.000, m4=1...
g2		m1=9.000, m3=1.000
g3		m1=1.000

Note that each genomic element type is automatically assigned a color by SLiMgui. We will soon see how these colors are used.

### 7.3 Chromosome organization

As previously discussed in section 4.1.5, genomic elements are regions of a chromosome that use a particular genomic element type. The genomic element types define what possibilities exist in the chromosome; the genomic elements determine what actually does exist, and where. Having defined genomic element types for exons, introns, and non-coding regions, we now need to create genomic elements to express how those genomic element types are distributed along the chromosome.

As has been the case all along with this recipe, the goal here is not biological realism; nevertheless, it is interesting to try to come up with a recipe that broadly approximates the structure of a real chromosome, just to show how the problem might be approached. In this recipe, then, the formulas used for the lengths of exons and introns are very loosely based on empirical length distributions.

This is by far the longest recipe we've seen so far, so a bit of preamble to prepare the way for it is helpful. We have previously used for loops to iterate over a specified vector. This example uses two new looping constructs, a `do` loop and a `do-while` loop. These both iterate as long as a given condition remains true; when the condition tests false, the loop terminates. The only difference between them is that `do-while` tests its condition at the end of the loop body, and thus the loop body always executes at least once. The other new thing in this recipe is use of the `rlnorm()` function, which draws samples from a lognormal distribution. In addition to the number of

samples to draw, it takes the mean and standard deviation of the distribution as parameters, on the log scale. Oh, and the `rdunif()` function draws samples from a discrete uniform distribution; this is the same as `runif()`, which we have seen before, except it only generates discrete (i.e., integer) values.

With that preface, here is the recipe:

```
initialize() {
  initializeMutationRate(1e-7);

  initializeMutationType("m1", 0.5, "f", 0.0);      // non-coding
  initializeMutationType("m2", 0.5, "f", 0.0);      // synonymous
  initializeMutationType("m3", 0.1, "g", -0.03, 0.2); // deleterious
  initializeMutationType("m4", 0.8, "e", 0.1);      // beneficial

  initializeGenomicElementType("g1", c(m2,m3,m4), c(2,8,0.1)); // exon
  initializeGenomicElementType("g2", c(m1,m3), c(9,1));      // intron
  initializeGenomicElementType("g3", c(m1), 1);              // non-coding

  // Generate random genes along an approximately 100000-base chromosome
  base = 0;

  while (base < 100000) {
    // make a non-coding region
    nc_length = rdunif(1, 100, 5000);
    initializeGenomicElement(g3, base, base + nc_length - 1);
    base = base + nc_length;

    // make first exon
    ex_length = asInteger(rlnorm(1, log(50), log(2))) + 1;
    initializeGenomicElement(g1, base, base + ex_length - 1);
    base = base + ex_length;

    // make additional intron-exon pairs
    do
    {
      in_length = asInteger(rlnorm(1, log(100), log(1.5))) + 10;
      initializeGenomicElement(g2, base, base + in_length - 1);
      base = base + in_length;

      ex_length = asInteger(rlnorm(1, log(50), log(2))) + 1;
      initializeGenomicElement(g1, base, base + ex_length - 1);
      base = base + ex_length;
    }
    while (runif(1) < 0.8); // 20% probability of stopping
  }


  // final non-coding region
  nc_length = rdunif(1, 100, 5000);
  initializeGenomicElement(g3, base, base + nc_length - 1);

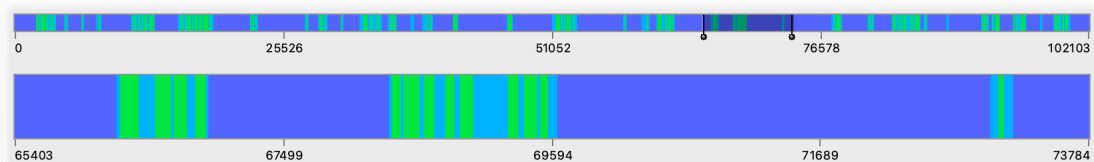
  // single recombination rate
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 5000); }
10000 early() { sim.simulationFinished(); }
```

This is too much code to examine line by line, but it should be fairly clear what is going on. The outer loop adds one non-coding region and then one gene each time that it iterates; the inner loop adds one intron-exon pair with each iteration. The overall algorithm is not targeted to end at a fixed chromosome length (doing that without biasing the metrics of the final gene generated would be a bit tricky); instead, genes are added until the chromosome length exceeds **100000** and then the process is ended with a final non-coding region. Eidos does not have any built-in facility for graphing, so you can't directly see the distributions used to generate the lengths of the exons and introns; however, since the syntax of Eidos is so similar to that of R, it is quite easy to use R to plot these distributions. If you are so inclined, try running this code in R:

```
x = as.integer(rlnorm(100000, log(50), log(2))) + 1
hist(x[x < 250], breaks=100)
```

That will produce a plot of the distribution of exon lengths, which may be compared to the distribution shown in Deutsch & Long (1999), which was loosely used as a reference for this model. The distribution of intron lengths may be compared similarly.

If you open this recipe in SLiMgui, Recycle, and Step, you will see the random chromosome structure that it generated, which might look something like this (after turning on display of genomic elements with the  button, and selecting a subrange in the upper chromosome view to show some detail in the lower view):



The chromosome structure consists of non-coding regions (dark blue) interspersed with genes made up of exons (cyan) alternating with introns (green), as intended. It should now be clear, of course, how SLiMgui uses the colors that it assigns to genomic element types, as seen in the previous section.

Rather than generating a random chromosome organization, you might wish to read in an actual chromosome map and generate genomic regions based upon that information, to simulate the evolution of a particular organism. That is left as an exercise for the reader, but with the recipe in section 6.1.2 as guidance it should not be too difficult.

## 7.4 Custom display colors in SLiMgui

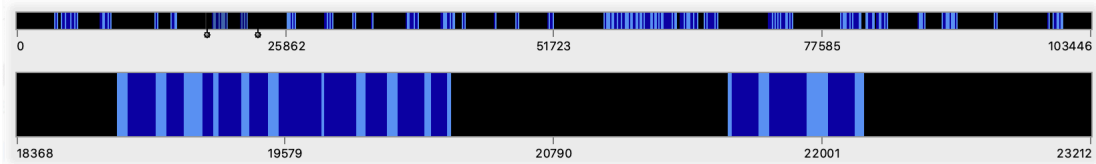
The model that we built in the previous three sections uses various default color schemes supplied by SLiMgui: mutations are colored according to their selection coefficients, genomic element types are automatically assigned colors from a standard palette, and individuals are colored according to their fitness. For simple models these default colors generally suffice, but when constructing a complex model it may be helpful to customize the color scheme used in SLiMgui to improve the clarity of the model's visual representation. In this section we will briefly explore SLiM's facilities for doing so.

First of all, the colors used for genomic regions, as shown in the figure above in section 7.3, is perhaps less than ideal. It would be nice if non-coding regions were shown in a distinctive color suggestive of the unimportance of those regions to the organism, such as black. Similarly, perhaps exons could be in the brightest color, connoting their importance, whereas introns could be shown in a more muted fashion. This can be accomplished by simply adding these lines after the calls to `initializeGenomicElementType()` have set up the genomic element types:



```
g1.color = "cornflowerblue";
g2.color = "#00009F";
g3.color = "black";
```

This produces a chromosome view like this (again, turning on display of genomic elements with the chromosome view action button, ⚙️, and selecting a subrange to zoom in on):



It works by setting values for the `color` property of the genomic element types. By default, these color properties are the empty string, `""`, which tells SLiM to use a color from its default color palette. Here we have instead told SLiM to use a light blue named "cornflowerblue" for the exons represented by `g1`, and "black" for non-coding regions represented by `g3`. These names are two of the 657 named colors supported by SLiM. The complete list of named colors is identical to the named color list provided by the R language, for simplicity, and so there are various web resources available that show all of the names and their corresponding colors (one such resource: <http://research.stowers-institute.org/efg/R/Color/Chart/ColorChart.pdf>). You can browse such a list, and then simply use the name for the color you want in SLiM.

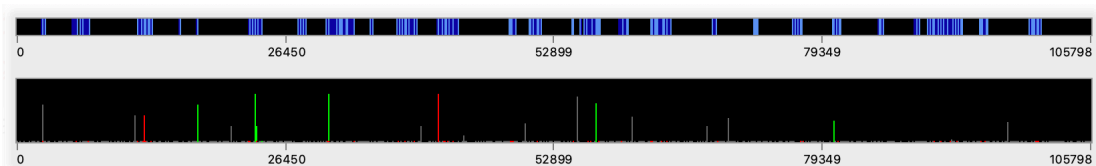
The color used for `g2` is not a named color, however; instead, it is the rather cryptic string value `"#00009F"`. This specifies the color in hexadecimal, or base 16, as two-digit values for the red, green, and blue components of the color. With values of zero for red and green, and a value of 9F for blue, this string represents a dark blue that works well for introns here.

The Eidos manual has further discussion of how colors are specified in SLiM (this is actually part of the Eidos language). Note that setting colors on SLiM objects in this way only has an effect in SLiMgui, but it is entirely legal even when running SLiM models at the command line; the color properties still exist, but are unused by SLiM outside of SLiMgui.

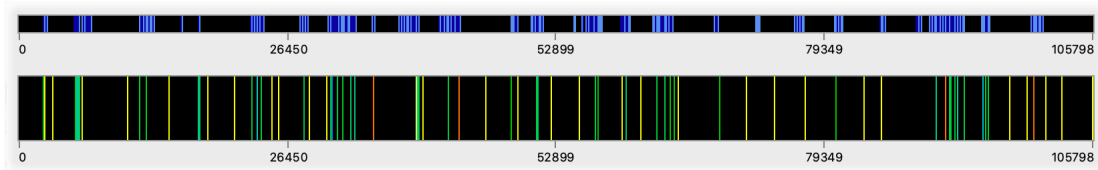
Now that the genomic elements are colored nicely, let's set up new colors for the mutations. By default, SLiMgui displays neutral mutations in yellow, beneficial mutations in shades of green and blue, and deleterious mutations in shades of orange and red. Let's suppose we want to de-emphasize neutral mutations in this model; we want them to display, but in a less visible color than the default bright yellow color. We also want all beneficial mutations to be a single shade of green (the blue doesn't show up well against the blues we've chosen for our genomic elements), and we want all deleterious mutations to be bright red so we can see them very clearly. This can be achieved with a few lines placed after the mutation types have been defined:

```
m1.color = "gray40";
m2.color = "gray40";
m3.color = "red";
m4.color = "green";
```

That produces a display like this (with display of the genomic elements turned off now, for greater clarity):



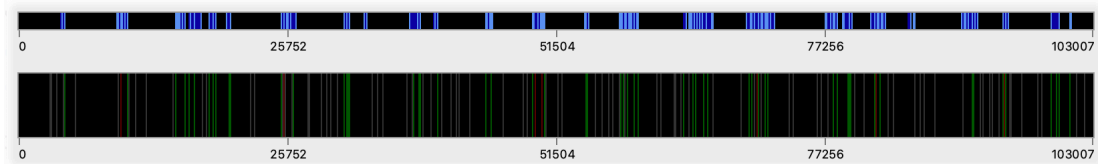
That looks like what we had in mind. However, if we use the ⚙ button to turn on display of fixed mutations (check Display Substitutions) and turn off segregating mutations (uncheck Display Mutations), we see that those are still being displayed in SLiM's default color scheme:



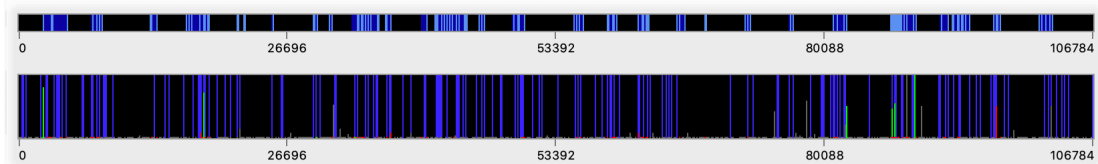
This is obviously not what we want. Instead, let's use darker shades of the same colors used for the mutations when they are still segregating. This can be done by setting the `colorSubstitution` property of the mutation types:

```
m1.colorSubstitution = "gray20";
m2.colorSubstitution = "gray20";
m3.colorSubstitution = "#550000";
m4.colorSubstitution = "#005500";
```

That produces the desired result:



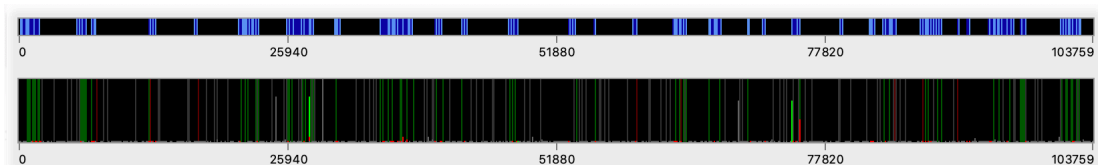
There's one thing left to tweak. If we turn on display of both segregating and fixed mutations, SLiMgui displays the fixed mutations using a shade of blue, rather than our custom colors:



This is deliberate, to prevent too much clutter and chaos in the chromosome view. However, we deliberately chose dark colors above for our fixed mutations with the intention of having them coexist aesthetically, so we'd like to tell SLiMgui to use our color scheme in all cases. The default dark blue color SLiMgui uses for fixed mutations when both are being displayed is a property on the `Chromosome` object named `colorSubstitution`. By default, it is set to `"#3333FF"`, the dark blue color above. We can set it to the empty string, `""`, instead; this eliminates the use of this default color. Since the `Chromosome` object is not available until the simulation has been fully initialized, we can do this at the beginning of tick 1, by adding a line to the tick 1 event:

```
sim.chromosome.colorSubstitution = "";
```

Now our model displays as intended:



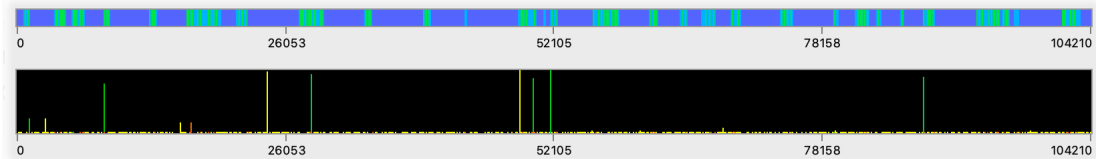
The fixed mutations are now in dark, dimmed colors, whereas the currently segregating mutations are in bright, clear colors that are easily visible against that background.


We won't give an example of it here, but the `Individual` class in SLiM also has a `color` property that can be set to override the default fitness-based color scheme for display of individuals. Setting this property should be done in a `late()` event, since SLiMgui updates its display in between ticks; `first()` events are executed after SLiMgui has already displayed, and so setting display colors for individuals at that time, or in an `early()` event, will have no visible effect. Generally SLiMgui's fitness-based coloring works well, but in some special cases it might be useful to give a specific color to individuals that possess some particular property or are in some particular state. Section 15.11 provides an example of such a model, where individuals are colored according to their phenotype (as determined by the additive effects of QTL mutations they possess).

## 8. SLiMgui visualizations for polymorphism patterns

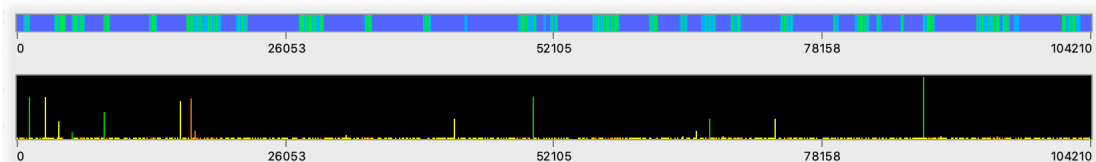
The previous chapter developed a model of evolutionary dynamics in a scenario involving neutral, deleterious, and beneficial mutations. Given the full recipe from section 7.3 (*not* 7.4) of the previous chapter, we will look briefly in this chapter at the interplay of selection, drift, and hitchhiking when it executes, since it's the most complex model we've made thus far. We will start by looking at how these dynamics can be observed in the chromosome view, and then we will move on to look at some of the graphs that SLiMgui provides.


First of all, if you Recycle and Play, you should see a fairly complicated dance of mutations, some fixing quickly, others struggling, but most flickering in and out of existence quickly:

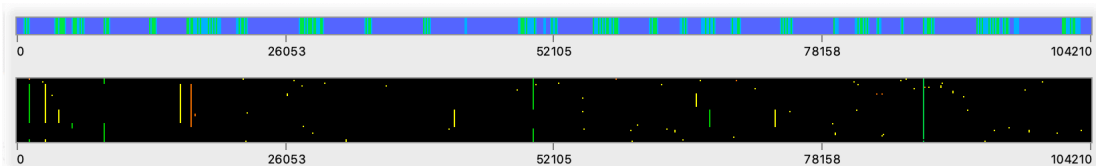


Using the  action button to the right, the main chromosome view on the bottom is displaying only the segregating mutations, while the chromosome overview on top shows the genetic structure of the model by displaying its genomic elements. Notice that the mutation lines are distinct colors; neutral mutations are yellow, beneficial are green, and deleterious are orange or sometimes even red.

A cloud of neutral and deleterious mutations occupies the bottom row of pixels; most mutations in the model are neutral or deleterious. A handful of beneficial mutations are visible in green, including several that appear to be on the verge of fixation. The beneficial mutations are all, as expected, located within the gene regions shown in green in the chromosome overview; beneficial mutations do not occur in the blue non-coding (g3) regions of the chromosome (see chapter 7 for discussion of the model design). Two neutral mutations, in yellow, have apparently been dragged up to high frequency by the beneficial mutations that are sweeping, which they are presumably linked to. A hundred ticks later, this is the situation:



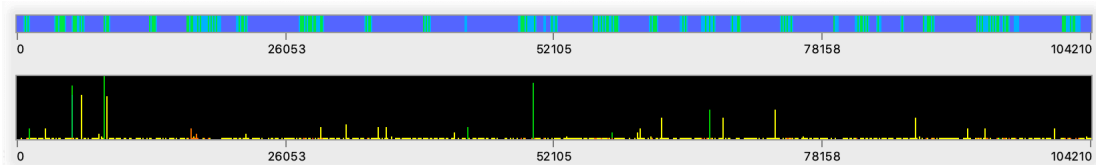
Several beneficial mutations have fixed, and their linked neutral mutations as well; they are thus no longer displayed. Several other beneficial mutations that were previously present have still not managed to fix, such as those near 50000 and 88000. The really interesting thing at this stage, however, is that a deleterious mutation near 17000 has risen to a frequency of more than 50%. How has it managed to do that? Let's switch the chromosome view's mode by clicking the  action button and choosing Display Haplotypes. This is the result, still at the same tick:



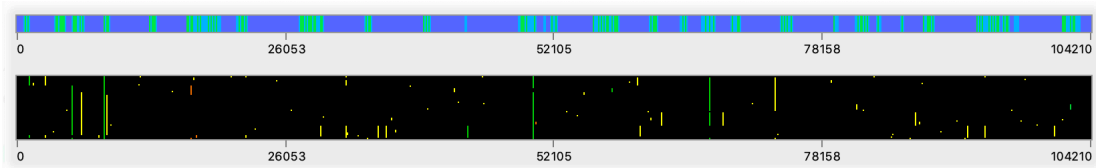
This is a different view on the same genetic data. In this mode, each horizontal row of pixels in the chromosome view represents one simulated genome, showing all of its mutations along the

chromosome; in other words, we are not looking at *frequency bars* any more, we are looking at a set of *randomly sampled genomes* from the model. These sampled genomes have been sorted by their genetic similarity, from top to bottom, to provide a more coherent picture; genomes that are genetically similar have been grouped together vertically. What we can see, then, is that the deleterious mutation near 17000 is strongly associated with a beneficial mutation very close to the left edge of the chromosome, around 1200; the correlation between the two mutations (their linkage disequilibrium) is very high – almost perfect. The deleterious mutation also co-occurs with several other beneficial mutations, but its fate is not tied tightly to them; it occurs in many genomes that those beneficial mutations do *not* occur in, and vice versa, so the linkage disequilibrium of those pairs is low. If you look back to the very first screenshot above, the deleterious/beneficial pair that exhibit high linkage disequilibrium were actually already visible back then; they seem to have originated more or less together, more than a hundred ticks ago, and they have traveled together since.

Two hundred ticks later, here is the situation (back in frequency mode):




Our deleterious/beneficial pair is back down to low frequency again; they appear to have lost out in the competition between the various beneficial mutations. Let's switch back to haplotype view to see this more clearly:

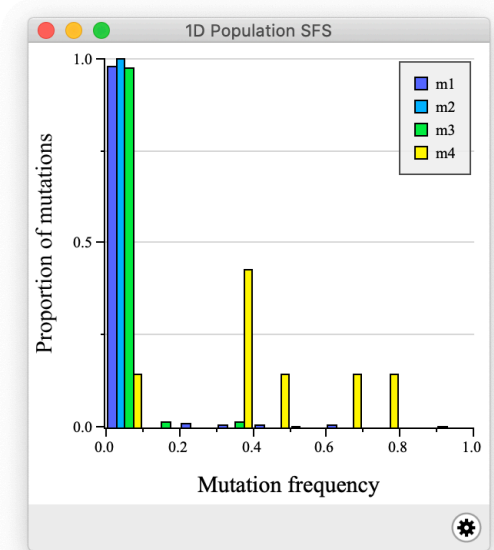


They have actually become unlinked from each other! The genomes that contain one do not contain the other any more; the correlation is gone. We could see this even more clearly by creating an actual haplotype plot in SLiMgui, as shown in section 13.3, 14.4, and 15.9; but perhaps this digression has gone on long enough. The take-home point here is: the chromosome view is quite rich in information, and can allow you to follow the fates of individual mutations in some detail, including their patterns of linkage, particularly if the Display Haplotypes mode is used.

Incidentally: as the simulation is running, you can watch the colors of the individuals, shown in the top center view, to see the mix of different fitnesses present. As a beneficial mutation sweeps, more and more individuals will turn green; once the mutation fixes, they will suddenly revert toward yellow, since the fitness benefit of that mutation is no longer part of SLiM's fitness calculations (discussed further in section 1.5.2).

## 8.1 Mutation frequency spectra

SLiMgui has a number of graphs it can display to help analyze and understand the simulation. While running this recipe, for example, if you click the  button and select "Graph 1D Population SFS" you should see something like this:

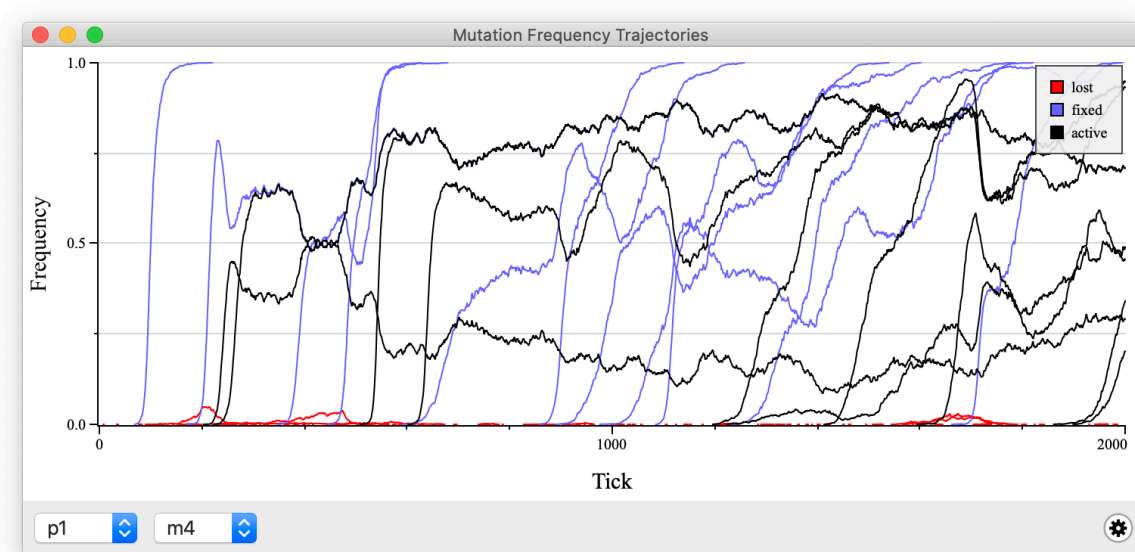


Remember that m1 and m2 are neutral, m3 is deleterious, and m4 is beneficial. What this plot tells us is that out of all mutations that are of type m1, for example, the vast majority are at very low frequency; the same is true of m2 and m3. Out of all mutations that are of type m4, however – the beneficial mutations – a fairly large proportion are at middle or high frequency, because they are on their way to fixation.

Note that graph windows in SLiMgui have their own ⚙️ action button; if you click it, you will see various options for reconfiguring the graph's appearance and exporting it or its data.

## 8.2 Mutation frequency trajectories

The next graph in SLiMgui's graph menu is "Graph Mutation Frequency Trajectories". Whereas the previous graph showed us only an analysis of the frequencies of different mutation types at the present moment in time (try opening it while the simulation is actually playing), this graph shows us similar information across the whole run of the model so far (here showing 2000 ticks):





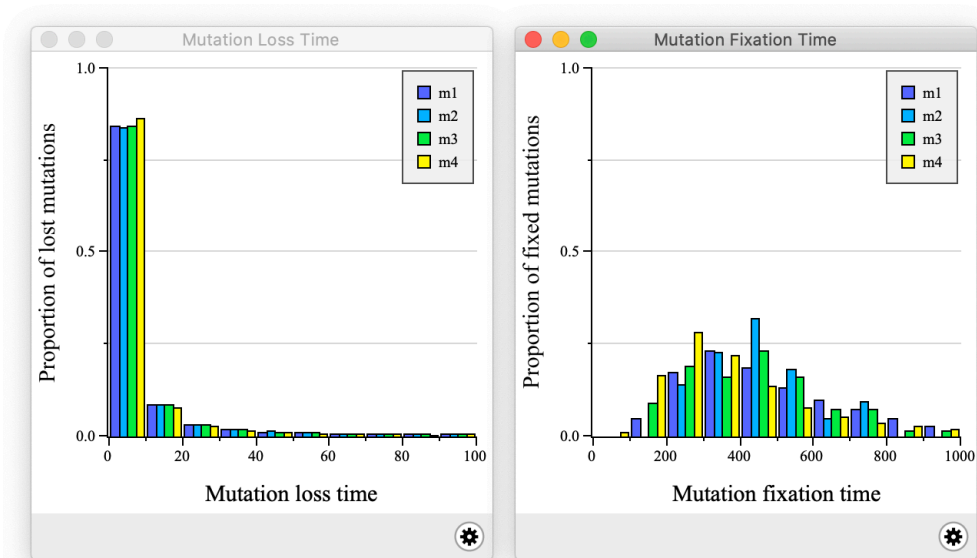
Using the popups at the lower left of the window, you can select a particular subpopulation and mutation type. The data collected for this graph can be quite large, and quite slow to collect. For this reason, the data are not normally collected when you run a model; doing so would slow SLiM down too much. Instead, the data are collected only when the graph window is open, and only for the subpopulation and mutation type chosen. If you want a plot for an entire simulation run, as shown above, you should therefore (1) Recycle, (2) Step once to advance to tick 1, (3) open the graph window, (4) select the mutation type you want from the popup, and (5) press Play to run your simulation with data collection.

The result at the end of a simulation run might look something like the graph shown above. The complete trajectory of every mutation of the selected mutation type in the selected subpopulation is shown with an individual curve in this plot; here we're looking at beneficial mutations (type m4), and there is only one subpopulation. The color of each curve indicates whether the mutation was lost (red), fixed (blue), or is still segregating (black).

SLiMgui provides various options to configure the visual appearance of plots. If you control-click or right-click on the graph window here, for example, you will get a context menu with menu items that allow you to show/hide grid lines, show/hide the legend, and so forth. If you're following along in SLiMgui, try experimenting with these options; you can't do any harm. SLiMgui graph windows also make their data available to you, if you want to regenerate them or perform further analysis; from the context menu, just select "Copy Data" to copy the underlying data for the plot to the clipboard, or "Export Data..." to export the data to a text file readable by other programs such as R. The format of the data should be pretty self-explanatory.

### 8.3 Times to fixation and loss

Next let's look at the plots for "Graph Mutation Loss Time Histogram" and "Graph Mutation Fixation Time Histogram", side by side:



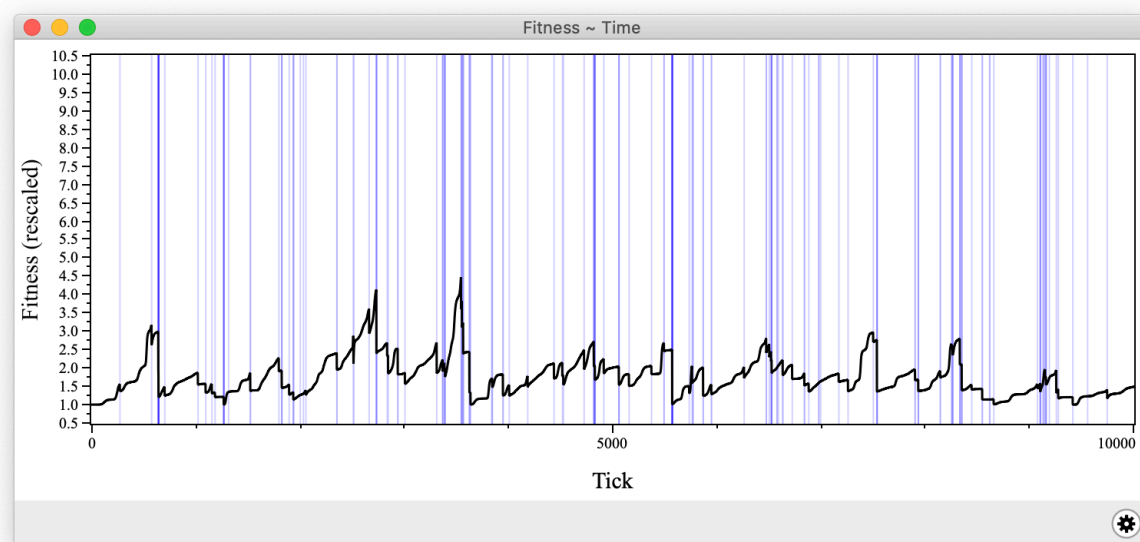
These plots show an analysis of metrics gathered over the course of an entire run (lengthened to 100,000 ticks, in fact, to get a more stable distribution). The loss time plot, on the left, shows the distribution of loss times, in ticks (not cycles, although they are the same in this simple model), for each of the four mutation types. Remember that this is *not* saying that beneficial mutations are just as likely to be lost as deleterious mutations, but only that *when* a beneficial mutation is lost, the

amount of time it takes to be lost is similar to the amount of time it takes a deleterious mutation to be lost. In other words, this plot is conditional upon mutations being lost; it implies nothing about the likelihood of loss. The loss profile for all four types is quite similar, but m4, the beneficial mutation type, appears to be a bit more likely to be lost quickly; if a beneficial mutation is lost at all, it is perhaps more likely to be lost very early, due to drift when it is still at very low frequency.

The fixation time plot, on the right, similarly shows the distribution of fixation times, in ticks (again, not cycles), for each of the four mutation types (conditional, similarly, upon the mutations having fixed). Here we can see that neutral mutations inside exons (mutation type m2) appear to have a different distribution of fixation times than neutral mutations inside introns and non-coding regions (mutation type m1), presumably due to linkage with mutations under selection in the exons (both hitchhiking and background selection). We can also see that beneficial m4 mutations tend to fix quickly (more skewed towards the left) compared to deleterious m3 mutations (more skewed towards the right).

## 8.4 Population fitness over time

The next graph we will examine is “Graph Fitness ~ Time”. At the end of a run of this recipe, to 10,000 ticks, it might look something like this (with a bit of customization of its appearance using the ⚙ action button – hiding the grid, adding a box, hiding subpopulation display):



This plot shows the fitness of the population over time (and depending on settings, it might show the fitness of each subpopulation too). More specifically, it shows mean fitness, and (following the SLiM engine) it rescales whenever a mutation fixes, dropping the fixed mutation from the calculation (see section 1.5.2). Without that rescaling, the plot would be nearly a monotonically rising curve; with the rescaling, a drop in the curve occurs each time that a beneficial mutation fixes (shown with a vertical blue line). In fact neutral and deleterious mutations that fix are also shown with a blue line in this plot, but since, in this model, they almost always fix at the same time as a beneficial mutation, due to hitchhiking, there are only a few fixation events in the plot that do not correspond to a drop in mean fitness due to that rescaling.

A few things can be observed in this plot. First of all, a bunch of mutations fixed over 10000 ticks; the probability of beneficial mutations arising in this model is probably much higher than the typical empirical rate. For the same reason, the strong-selection–weak-mutation assumption

emphatically does not hold here; beneficial mutations are stacking up and competing with each other, as can be seen from both the complex wiggling of the curve in between fixation events, and from the fact that fixation events usually do not drop the mean fitness back down to 1.0 (indicating that at least one other beneficial mutation exists at intermediate frequency in the population at the same time).

This may be a good moment for a digression about exactly how SLiMgui displays “fitness” to the user, which is a surprisingly complicated topic. First of all: SLiM calculates fitness multiplicatively. The fitness of an individual, as calculated by SLiM, is the product of all of the fitness effects influencing that individual. More specifically, it is the product of the fitness effects of all mutations possessed by the individual (perhaps influenced by `mutationEffect()` callbacks), multiplied by any additional fitness effects from `fitnessEffect()` callbacks, Individual-level `fitnessScaling` values, and Subpopulation-level `fitnessScaling` values. This is documented at an introductory level in sections 1.3 and 1.5.2, and in detail in the reference sections 22.6 and 23.3 for WF and nonWF models respectively; reference documentation for `mutationEffect()` and `fitnessEffect()` callbacks are in sections 25.2 and 25.3, and reference documentation for the `fitnessScaling` property is in sections 24.7.1 and 24.15.1 for Individual and Subpopulation respectively. Many of these sources of fitness effects have not been discussed yet, and will come up in recipes in later chapters; chapter 10, for example, is all about `mutationEffect()` callbacks. So we’re getting ahead of ourselves a bit, but the point is: all of these fitness effects get multiplied together to produce a final fitness value for each individual, which SLiM usually just calls “fitness”. The mean of this, across the whole population and also, optionally, across each subpopulation, is essentially what is displayed in the graph shown above; however, SLiMgui applies a rescaling to those values prior to display, as described in detail below.


In nonWF models, these fitness values are used directly as each individual’s probability of survival during the viability/survival tick cycle stage (see section 23.4); they can therefore be considered to represent “absolute” fitness. In WF models, these fitness values influence the probability of mating instead, and the absolute fitness value for an individual is unimportant; the important thing is the magnitude of an individual’s fitness value *relative* to the fitness of other individuals in the same subpopulation, since individuals with higher fitness are more likely to be chosen as mates. In WF models, then, these fitness values represent a non-normalized metric of “relative” fitness. SLiM internally normalizes them to sum to 1.0 across each subpopulation to represent mating probabilities; these normalized values are user-visible as the `weights` vector in `mateChoice()` callbacks, but are not otherwise available. See section 1.6 for further discussion of the meaning of fitness in WF versus nonWF models.

It is worth noting that SLiM’s fitness values may not include everything that a biologist would consider to be a part of “fitness”; they are simply calculated values that influence viability (in nonWF models) or mating (in WF models). One could easily write a SLiM model in which other factors (whether genetic or non-genetic) influenced the probability that an individual would mate, and those other factors would not be included in SLiM’s fitness values. For example, a WF model could have a `mateChoice()` callback that made particular individuals unlikely to mate even though their fitness was high (due to juvenile exposure to environmental chemicals, say); or a nonWF model could have a `reproduction()` callback that decreased individual fecundity based upon particular mutations (life-history genes influencing the individual’s investment in growth versus reproduction, say). Such scripted effects would not influence SLiM’s calculated fitness values, but they would figure into a biologist’s assessment of what “fitness” was doing in the model. SLiMgui displays only SLiM’s calculated fitness values; it makes no attempt to assess the true biological fitness of individuals. Biologists sometimes talk about fitness *components*: conceptual sub-partitions of the overall fitness of individuals. From this perspective, SLiM’s fitness values can be thought of as one fitness component, among others that might be present in a model.

Returning to the plot above, we see the y axis is titled “Fitness (rescaled)”. What does that mean, exactly? This term actually captures two different things that are going on. One, as discussed above, is the way that fixed mutations get removed by default in the WF model, resulting in the downward jumps observed in the plot. This is one kind of “rescaling” being applied: once the population is fixed for a given mutation in the WF model, the fitness effect of that mutation can be ignored because the WF model depends upon relative fitness, not absolute fitness, as explained above. Removing the mutation from the simulation and creating a `Substitution` object for it therefore “rescales” fitness values in the model to no longer contain that mutation’s fitness effect.

The other sense in which fitness values here are “rescaled” is that they do not include the effects of density-dependent population regulation in nonWF models – or, to be more precise, they do not include the effect of the `Subpopulation` property `fitnessScaling`. As we will see when we start working with nonWF models in chapter 16 (see section 1.6 for a quick overview), nonWF models are responsible for their own population regulation, and typically do so by setting a subpopulation-wide fitness effect in the `fitnessScaling` property of the subpopulation. This serves to decrease the absolute fitness of the whole subpopulation if it is too large, or increase the absolute fitness of the whole subpopulation if it is too small. Because the subpopulation size is often fairly stochastic, and might be affected by other factors like variation in migration, fecundity, etc., variation in this density-dependence term of fitness can introduce a lot of noise that obscures the signal that the user of SLiMgui is typically interested in: the extent to which individuals are carrying beneficial or deleterious alleles. To better reflect that signal, then, SLiMgui does not include the effect of the `Subpopulation`-level `fitnessScaling` value in its display. The fitness values displayed are therefore *not* the final values used by SLiM to determine survival in the nonWF model; this may sometimes be confusing, but overall it seems more likely to be useful. Note that in spatial nonWF models (see section 16.10), density-dependent effects are put into the `Individual` `fitnessScaling` property, not the `Subpopulation` `fitnessScaling` property, because different individuals feel different strengths of density-dependence; SLiMgui has no way to separate this out from other fitness effects the model might use, since it is embedded into the `Individual` `fitnessScaling` value (along with, perhaps, other individual-level fitness effects), so in this case SLiMgui’s display does not include this type of rescaling.

All of the above discussion applies to “fitness” as displayed in SLiMgui in other contexts too. In particular, fitness values that are “rescaled” in the senses defined above are displayed in the `Population Fitness Distribution` and `Subpopulation Fitness Distribution` plots also, and are used to determine the colors for individuals in the individuals view in SLiMgui. SLiMgui never uses the final fitness value for individuals, including the `Subpopulation`-level `fitnessScaling` value; it always rescales to remove that scaling factor.

OK, that ends our digression about fitness calculations in SLiM and SLiMgui. Moving on; we have seen the “Graph Population Visualization” plot before, and it is not interesting here since there is no population structure in this recipe; so this concludes our tour of SLiMgui’s graphing facilities (the “Create Haplotype Plot” command is shown in sections 13.3, 14.4, and 15.9). Some new plots have been added to SLiMgui since this chapter was written, and thus are not described here at the moment; note that for all of SLiMgui’s plots you can always select “About This Graph” from the action button  in the graph window to get a brief explanation of what the graph displays.

If you want to graph other things, you can of course generate an output file with the data you need, read the data into R, and generate your plots there (or you can sublaunch R from within your model to generate live plots as your model runs in SLiMgui; see section 14.8). Also, don’t forget that you can copy SLiMgui’s graphs to the clipboard, or save them out as PDF files, using the “Copy Graph” and “Export Graph...” commands in the action button’s pop-up menu.

## 9. Selective sweeps

This chapter shows how SLiM can be used to model selective sweeps and various associated phenomena such as hitchhiking, background selection, and genetic draft. SLiM's ability to accurately model such scenarios is the origin of its name, **S**election on **L**inked **M**utations. This chapter will use very simple one-population models; of course these recipes can be combined with the recipes of previous chapters to model sweeps with complex demography and population structure, complex genetic architecture, spatiotemporal variation in selection, and so forth.

### 9.1 Introducing adaptive mutations

The simplest selective sweep scenario is of an explicitly introduced adaptive mutation sweeping against a background of neutral mutations. We can simulate such a sweep in just a few lines:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
1000 late() {
    target = sample(p1.genomes, 1);
    target.addNewDrawnMutation(m2, 10000);
}
10000 late() { sim.simulationFinished(); }
```

The call to `initializeMutationType()` sets up `m2`, a new mutation type used to represent the introduced beneficial mutation. Note that a dominance coefficient of `1.0` (fully dominant) is used with a selection coefficient of `0.5`; these values provide strong selection for the introduced mutation, making it less likely to be lost due to genetic drift while still at low frequency. This is useful for illustrative purposes here, but of course in real simulations less favorable values would likely be needed, making it more probable that the mutation would be lost rather than sweeping. There are two ways to handle this. One is to simply accept that some runs of the simulation will fail to sweep – do many runs of the model, and collect statistics only across those runs where the sweep succeeds. The second way to handle the problem is to write additional Eidos code to make the simulation run conditional on fixation (see the next section).

The other interesting code here is the Eidos event at tick `1000`. The first line of this event selects a target genome into which the added mutation will be placed. It does this using the `sample()` function, by drawing a single sample from subpopulation `p1`'s vector of genomes. This is important, because SLiM does *not* – for reasons of speed – guarantee a random order to the individuals in a subpopulation. In this simple model all individuals are produced identically, so this is not an issue, but if features of SLiM are used such as migration, separate sexes, selfing, cloning, or `mateChoice()/modifyChild()/recombination()` callbacks, the individuals in the subpopulation will be produced in a specific and non-random order, and thus whenever a random individual is desired `sample()` must be used (rather than just using, e.g., the genome at index `0`).

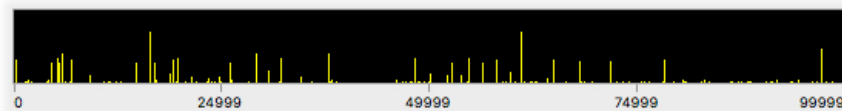
The second line then adds a new mutation to the target genome, drawn from mutation type `m2`; the remaining parameter specifies the position of the mutation in the chromosome as `10000` (this could be drawn randomly as well, if desired, using a function such as `rdunif()`).

Note that this event is a `late()` event, specified in its declaration. These were introduced back in section 4.2.1 as a way to make scripted events happen late in the tick, after offspring generation



has occurred (see the tick cycle overview in section 1.3). It is important that introducing new mutations occurs in a `late()` event, because it makes script-introduced mutations act in exactly the same way as mutations added by SLiM due to the normal process of mutation during offspring generation. If this event were an `early()` event instead, the mutation would be introduced immediately before offspring generation, and it would have no effect on fitness values since fitnesses would have been calculated before it was added (toward the end of the preceding tick). The mutation would therefore be subject to one tick of drift before its correct fitness effect would be accounted for. This would likely not be the desired behavior, so SLiM will issue a warning if you add a mutation during an `early()` event – you can change the `late()` designation here to `early()` to see that.

The introduced mutation in this model typically sweeps quite quickly (or is lost quickly due to drift), so perhaps the simplest way to see what is going on in SLiMgui is to recycle and then enter **1000** in the tick textfield and press return (as illustrated in section 5.1.2). The simulation will advance to the beginning of tick **1000**, and you can then press Step to single-step forward and watch the progress of the introduced mutation. Alternatively, you could slow down the simulation speed using the speed slider (as illustrated in section 5.2.2) to see the simulation play more slowly. In either case, you should observe that at tick **1000**, just before the beneficial mutation is introduced, there is lots of neutral diversity across the chromosome:



But just before the mutation finishes sweeping (assuming it is not lost – you may need to recycle and repeat this procedure several times to get a run in which the sweep establishes), most of that diversity has been lost, and just a few neutral sites have hitchhiked along with the introduced mutation, producing the characteristic signature of a selective sweep (Smith & Haigh 1974):



Note the green line at position **10000**, representing the beneficial mutation itself.

We can automatically halt the model immediately after the introduced mutation has fixed (or has been lost). This requires just a simple modification of the final line of the model, with the call to `simulationFinished()`:

```
1000:10000 late() {
    if (sim.countOfMutationsOfType(m2) == 0)
        sim.simulationFinished();
}
```

Now `simulationFinished()` is called immediately when the number of mutations of mutation type `m2` falls to zero, which happens either when the introduced mutation fixes (because SLiM then converts the mutation into a substitution object), or when it is lost. (See section 9.9 for an alternative approach that may be better in some situations, based upon remembering the sweep mutation object itself.) Note the tick range of **1000:10000** on this event; this means that this termination condition will be checked each tick from **1000** (when the introduced mutation originates) until **10000** (when the model ends, regardless). Also note that the `late()` designation ensures that the simulation ends in the same tick that the `m2` mutation is lost or fixed.



It might be nice to have some indication, in the output stream of the simulation, as to whether the introduced mutation fixed or was lost. One simple way to do this is to check for a Substitution object of the right mutation type, in a small extension to the event above:

```
1000:100000 late() {
  if (sim.countOfMutationsOfType(m2) == 0)
  {
    fixed = (sum(sim.substitutions.mutationType == m2) == 1);
    cat(ifelse(fixed, "FIXED\n", "LOST\n"));
    sim.simulationFinished();
  }
}
```

The variable `fixed` is assigned a logical value that comes from checking the list of substitutions for elements with mutation type `m2`. Each match will produce a value of T in the logical vector that results from the `==` operator; and since T is equal to 1 in Eidos, whereas F is equal to 0, the `sum()` function then adds up the total number of matches. If the sum is 1, the mutation fixed; otherwise (when it is 0) the mutation was lost.

The next line calls `cat()` to concatenate a string to the output stream. The string comes from the `ifelse()` function, which looks at the logical value of its first parameter (`fixed`) and returns its second parameter ("FIXED\n") if that value is T; otherwise, it returns its third parameter ("LOST\n"). This function is based on the `ifelse()` function of R; it is vectorized, and can thus perform this operation across whole vectors at once – a useful tool. An if–else here would be less compact:

```
if (fixed)
  cat("FIXED\n");
else
  cat("LOST\n");
```

Another alternative, while we’re on the subject, would be to use the “ternary conditional” operator of Eidos. This is similar to the `ifelse()` function, but it is a built-in operator in Eidos, and it is not vectorized; the Eidos manual has further details about it. Its use would look like this:

```
cat(fixed ? "FIXED\n" else "LOST\n");
```

## 9.2 Making sweeps conditional on fixation

The recipe in the previous section did not guarantee that the introduced mutation would sweep to fixation. That is not necessarily a flaw; sometimes one is interested in the outcome of a model both when a sweep completes and when it does not. Sometimes, however, one wishes to make a model that guarantees that a sweep completes:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  // save this run's identifier, used to save and restore
  defineConstant("simID", getSeed());

  sim.addSubpop("p1", 500);
}
```

```

1000 late() {
  // save the state of the simulation
  sim.outputFull(tempdir() + "slim_" + simID + ".txt");

  // introduce the sweep mutation
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m2, 10000);
}
1000:100000 late() {
  if (sim.countOfMutationsOfType(m2) == 0)
  {
    fixed = (sum(sim.substitutions.mutationType == m2) == 1);

    if (fixed)
    {
      cat(simID + ": FIXED\n");
      sim.simulationFinished();
    }
    else
    {
      cat(simID + ": LOST - RESTARTING\n");

      // go back to tick 1000
      sim.readFromPopulationFile(tempdir() + "slim_" + simID + ".txt");

      // start a newly seeded run
      setSeed(rdunif(1, 0, asInteger(2^62) - 1));

      // re-introduce the sweep mutation
      target = sample(p1.genomes, 1);
      target.addNewDrawnMutation(m2, 10000);
    }
  }
}

```

To make a model that is conditional on fixation, there are some simple “solutions” that don’t work. You can’t force the introduced mutation to increase in frequency each tick; that would produce abnormal evolutionary trajectories that would distort the model dynamics. Similarly, you can’t just reintroduce a new mutation if the previous one is lost; in the process of getting lost, the previous mutation may have affected the genetic background. What you really want is that if the introduced mutation has been lost, the simulation gets reset to precisely the same state as it was in prior to the previous introduction, but with a different random number seed to ensure that events take a different course. If you do this repeatedly until the mutation fixes, you have a proper simulation of a selective sweep conditional upon fixation. SLiM provides a convenient solution for this by allowing the user to save the relevant state of a simulation to disk. Doing this is actually pretty simple; there are just a few key steps that need to be taken to save the state and then restore it when we want to try a different trajectory.

First of all, the tick 1 event now stores the initial random number seed, obtained from `getSeed()`, as a constant named `simID`, using the Eidos function `defineConstant()`. Such constants persist until the simulation terminates or is recycled; unlike local variables, defined constants do not disappear at the end of the current event in SLiM, but instead go into the global scope. This value is thus a unique identifier for the model run; if many model runs were being done, each would presumably have a different seed, and thus the initial seed identifies the run.

The event in tick 1000 saves the current population state to a file in a temporary directory; on all Unix systems (including macOS) this will be `/tmp` or a subdirectory of it. The filename used is

generated using string concatenation with the + operator (described in detail in the Eidos manual), based on the value stored earlier in `simID`; the temporary file therefore gets named according to the initial random number seed of the run, with a name like "slim\_92631.txt". This event also adds the new mutation that we want to sweep. Note that this event is designated as a `late()` event; this is logical, both because it produces output (the saved simulation state), and because it adds a mutation to the simulation just before fitness recalculation.

The tick `1000:100000` event checks, each cycle, whether the introduced mutation is still present. If it has fixed, it prints a message and terminates the simulation. If it has been lost, it prints a message and reads the saved population state back in (which sets the tick counter back to `1000`). It then changes the random number seed, to start a new evolutionary trajectory (reproducible by starting again at the new seed – see section 20.1 for discussion), and then re-introduces the sweep mutation as before. This event is also a `late()` event; this way if the simulation has to be restored to the saved state, it picks up exactly where it was saved out (it should also be a `late()` event because it adds a new mutation). Finally, note that when we set the tick back to `1000`, the tick `1000` event that did our initial setup does not execute again; the events that will run for a given tick are determined at the beginning of that tick and do not change even if `community.tick` is changed. If we were to set `community.tick` to `999` instead, then the tick `1000` event would execute again, in the next tick.

If you recycle and run this model, sometimes the introduced mutation will fix on the first attempt, but sometimes it will take repeated attempts and you will see output like this:

```
1452090492983: LOST – RESTARTING
1452090492983: LOST – RESTARTING
1452090492983: LOST – RESTARTING
1452090492983: FIXED
```

This demonstrates that the model is working as intended; three introduced mutations were lost before the fourth attempt resulted in fixation. You could change the dominance coefficient and selection coefficient of `m2` to be less favorable, which would result in more restarts but should work fine. You could even make `m2` neutral or deleterious, while still making runs conditional on fixation; we'll see an example of that in section 9.6.2. Since the probability of a new neutral mutation drifting to fixation is small, many restarts will be needed, but no harm is done by that.

One major caveat is that `outputFull()` writes out only a specific set of information: the subpopulations that exist, the individuals they contain, and the mutations contained by the genomes of those individuals. When `readFromPopulationFile()` is called, any other state associated with the subpopulation, individuals, genomes, and mutations is wiped away. If the model had set up other properties – setting a cloning rate on a subpopulation, say, or setting up values with `tag` or `setValue()` – that state will be lost, and will need to be set up again.

Given the relative complexity of this recipe, most of the other selective sweep recipes in this chapter will assume that conditionality on fixation is not desired, so that their different topics are not obscured by the conditionality machinery. However, the key elements of this recipe can be combined with any selective sweep strategy – or indeed, can be used to make a simulation that is conditional upon any outcome you wish. In the next section, we will see how to make a simulation conditional upon establishment of a mutation, rather than upon fixation.

### 9.3 Making sweeps conditional on establishment

In the previous section, we saw how to make a model conditional on fixation of the introduced mutation. Sometimes one may want to relax this condition and require only that the mutation reaches a certain threshold frequency at which selection outweighs drift, rendering subsequent loss

unlikely. This threshold frequency is commonly referred to as the establishment frequency, and is a function of the selection coefficient of the mutation. Making our model conditional on establishment, rather than fixation, requires just a simple modification to the previous recipe:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  // save this run's identifier, used to save and restore
  defineConstant("simID", getSeed());

  sim.addSubpop("p1", 500);
}
1000 late() {
  // save the state of the simulation
  sim.outputFull(tempdir() + "slim_" + simID + ".txt");

  // introduce the sweep mutation
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m2, 10000);
}
1000: late() {
  mut = sim.mutationsOfType(m2);

  if (size(mut) == 1)
  {
    if (sim.mutationFrequencies(NULL, mut) > 0.1)
    {
      cat(simID + ": ESTABLISHED\n");
      sim.deregisterScriptBlock(self);
    }
  }
  else
  {
    cat(simID + ": LOST - RESTARTING\n");

    // go back to tick 1000
    sim.readFromPopulationFile(tempdir() + "slim_" + simID + ".txt");

    // start a newly seeded run
    setSeed(rdunif(1, 0, asInteger(2^62) - 1));

    // re-introduce the sweep mutation
    target = sample(p1.genomes, 1);
    target.addNewDrawnMutation(m2, 10000);
  }
}
10000 early() {
  sim.simulationFinished();
}
```

Much of the machinery here is carried over from the previous recipe; see section 9.2 for discussion of the way the simulation state is saved and restored. Here, however, the 1000: event

checks for the existence of the introduced mutation, using `size()`. If it exists, its frequency is checked against the threshold frequency `0.1` (an arbitrary choice that can be adjusted to whatever threshold frequency you wish). If the mutation's frequency is above that threshold, a message is printed and the script block deregisters itself so that further checks are not done – once the mutation has established, the simulation runs freely to its end, whether the mutation is subsequently fixed or lost. (See section 5.1.2 for discussion of `deregisterScriptBlock()`, as well as sections 9.5.3 and 24.13.2). On the other hand, if the introduced mutation no longer exists, the simulation is reset back to the point of introduction, just as in section 9.2.

When this model is run, you will typically see output like:

```
1459603349880: LOST – RESTARTING
1459603349880: LOST – RESTARTING
1459603349880: LOST – RESTARTING
1459603349880: ESTABLISHED
```

## 9.4 Partial sweeps

Sometimes it is desirable to make a selective sweep stop or change when the sweep mutation reaches a specific frequency. This recipe will initiate a selective sweep with a beneficial mutation, which will convert into a neutral mutation when it reaches a frequency of `0.5`.

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
1000 late() {
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m2, 10000);
}
1000:10000 late() {
  mut = sim.mutationsOfType(m2);
  if (size(mut) == 0)
    sim.simulationFinished();
  else if (mut.selectionCoeff != 0.0)
    if (sim.mutationFrequencies(NULL, mut) >= 0.5)
      mut.setSelectionCoeff(0.0);
}
```

Most of this is similar to previous recipes in this chapter; the difference lies in the `1000:10000` event. This event now uses `mutationsOfType()` to recover the introduced mutation object (see section 9.9 for an alternative, and section 27.2 for discussion). If the mutation is no longer present – if `size(mut)` is zero – the simulation terminates; this is like the `countOfMutationsOfType(m2)` check done in previous recipes. Otherwise, it uses `mutationFrequencies()` to check the frequency of the introduced mutation; if it has cleared the `0.5` threshold, it is converted to neutral with `setSelectionCoeff()`. That last bit of code is protected by a test that the selection coefficient has not already been changed; that check is purely for speed, as `mutationFrequencies()` can be slow.

This recipe could incorporate parts of the previous recipes, in order to determine whether the introduced mutation was lost or had fixed, or in order to make the simulation conditional on the mutation reaching the threshold frequency; the basic idea is easily adapted.

## 9.5 Soft sweeps from *de novo* mutations

In the previous recipes in this chapter, we've seen "hard" selective sweeps originating from a single new mutation. If, on the other hand, a sweep proceeds from multiple copies of the same mutation, present in different individuals, it is termed a "soft" sweep, because it preserves a more diverse genetic background (Messer & Petrov 2013). Soft sweeps can arise from standing genetic variation (seen in section 9.6), or can occur when multiple copies of the same *de novo* mutation arise during a sweep. This section will model the latter scenario, in three different ways.

### 9.5.1 A soft sweep from recurrent *de novo* mutations in a large population

In this recipe, we will model a soft sweep by *de novo* mutations generated by SLiM. In order to get a soft sweep to occur from recurrent *de novo* mutations at the same locus, the population needs to be very large or the mutation rate needs to be very high, so that new copies of the mutation tend to arise before the previous copy has fixed. In this recipe, unlike most recipes in this book, we will not model a background of neutral mutations; instead, this is a model of competing beneficial mutations at a single site. We are, in effect, modeling the sweep of multiple, separate instances of a particular single-nucleotide change, such as the transition of a particular nucleotide from G to A (only conceptually; we will see explicitly nucleotide-based models in chapter 18). The model terminates when the sweep completes: when every individual genome possesses an instance of the mutation, regardless of where and when this instance arose. With no further ado, the recipe:

```
initialize() {
  initializeMutationRate(1e-5);
  initializeMutationType("m1", 0.45, "f", 0.5); // sweep mutation
  m1.convertToSubstitution = F;
  m1.mutationStackPolicy = "f";
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 0);
  initializeRecombinationRate(0);
}
1 early() {
  sim.addSubpop("p1", 100000);
}
1:10000 early() {
  counts = p1.genomes.countOfMutationsOfType(m1);
  freq = mean(counts > 0);

  if (freq == 1.0)
  {
    cat("\nTotal mutations: " + size(sim.mutations) + "\n\n");

    for (mut in sortBy(sim.mutations, "originTick"))
    {
      mutFreq = mean(p1.genomes.containsMutations(mut));
      cat("Origin " + mut.originTick + ": " + mutFreq + "\n");
    }

    sim.simulationFinished();
  }
}
```

The chromosome is defined as having a single genomic element that spans from position 0 to position 0 – a single base position. This recipe uses a relatively high mutation rate (1e-5) with a large population size (100000) so that multiple mutations arise at that single site before the sweep



completes. An even larger population size could be used with a lower mutation rate ( $N=1e6$  and  $u=1e-6$ , say, or even  $N=1e7$  and  $u=1e-7$ ), but the model would take somewhat longer to complete.

There is one unusual complication in this model: SLiM, by default, allows multiple mutations of a given mutation type to occur at the same site in the same individual ("stacked" mutations; see section 1.5.3). In other words, an individual that has already undergone the G-to-A transition might nevertheless be given another mutation at the same site, representing the same G-to-A transition. While this behavior is desirable in many instances, it is inconvenient in this particular model, and so this recipe tells SLiM to use a different behavior: stacked mutations will not be allowed, and new mutations that collide with an existing mutation at the same site will be suppressed. This is achieved by setting the `mutationStackPolicy` property of `m1` to "f", which tells SLiM to prevent stacking by keeping only the *first* (thus "f") mutation of type `m1` at a given site (see section 1.5.3 for details). This simplifies the model's code considerably; without this change in policy, the model would have to carefully take account of the possibility of stacking and compensate for it. Note that if this model contained other mutation types as well, those would still be allowed to coexist, both with each other and with an `m1` mutation at the same site; only stacking of `m1` mutations with other `m1` mutations is prevented by default.

Second, the dominance coefficient of `0.45` for `m1` is actually a special and important value. SLiM tracks each independent origin of the sweep mutation as a separate `Mutation` object; those mutations are all of type `m1`, but they are distinct mutations as far as SLiM is concerned (see section 1.5.2). If an individual has different versions of the sweep mutation in its two genomes, SLiM will evaluate the fitness of that individual according to the fact that it contains two different mutations, each of which is heterozygous, and each of which therefore uses the mutation type's dominance coefficient. The value `0.45` makes this work out, because the relative fitness for one heterozygous mutation is  $1.0 + 0.45 \times 0.5$ , which is  $1.225$ , and then when the relative fitness values for the two mutations are multiplied together,  $1.225 \times 1.225 = 1.5$  (almost exactly), and  $1.5$  is the relative fitness of the sweep mutation when homozygous. In other words,  $h = (\sqrt{1+s} - 1)/s$ , and therefore  $(1+hs)^2 = 1+s$ . This model could be written to use a dominance coefficient that did not satisfy this relationship, but a `mutationEffect()` callback would be needed to produce the desired effects.

The `1:10000` event tallies up the overall frequency of the sweep mutation, regardless of which particular `Mutation` objects are possessed by each individual. To do this, it first uses the `countOfMutationsOfType()` method to produce a vector of the number of mutations in each genome in the population. Then it tests `counts > 0`, producing a `logical` vector that is `T` if a genome contains at least one mutation, `F` otherwise. The `mean()` function computes the frequency of the sweep as the average of those values (implicitly converting `T` to `1` and `F` to `0`, as a convenience, following Eidos convention; one could call `asInteger()` to do this explicitly). If the frequency is equal to `1.0`, the sweep has completed and the simulation is stopped.

This model executes and stops, but it is hard to tell what's going on; there is only a single base position on the chromosome, so all of the mutations are displayed in a single column in SLiMgui's chromosome view. We can sort of see the sweep progressing, but we can't tell how many mutations are involved, or what the distribution of their frequencies might be. The model therefore has some custom output code that processes the final state of the simulation and prints a summary. Because stacking is prevented by `mutationStackPolicy`, as described above, this output code is quite simple; it just loops through all of the mutations in the simulation (sorted by origin tick, using `sortBy()`), and calculates the frequency of each mutation as the average of the values returned by `p1.genomes.containsMutations()` for that mutation – a very similar strategy to the calculation of the overall frequency of the sweep, as described above. Typical output might indicate that 25 mutations existed in the population at the end of the run (and thus participated in the soft sweep), and would then list the final frequencies of those mutations, sorted in ascending order by origin tick:

Total mutations: 25

Origin 4: 0.22628  
Origin 6: 0.04301  
Origin 7: 0.18139  
Origin 7: 0.215995  
Origin 10: 0.10995  
Origin 12: 0.0709  
...

### 9.5.2 A soft sweep with a fixed mutation schedule

The previous recipe relied upon SLiM's standard mutation-generation machinery to generate new instances of a mutation that executed a soft sweep. Sometimes one might wish to have more control over the process than that; our next recipe therefore adds the multiple copies of the sweep mutation at predetermined times during the run, according to a scripted schedule:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 0.45, "f", 0.5); // sweep mutation
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
    p1.tag = 0; // indicate that a mutation has not yet been seen
}
1000:1100 late() {
    if (sim.cycle % 10 == 0) {
        target = sample(p1.genomes, 1);
        if (target.countOfMutationsOfType(m2) == 0)
            target.addNewDrawnMutation(m2, 10000);
    }
}
1:10000 late() {
    if (p1.tag != sim.countOfMutationsOfType(m2)) {
        if (any(sim.substitutions.mutationType == m2)) {
            cat("Hard sweep ended in cycle " + sim.cycle + "\n");
            sim.simulationFinished();
        } else {
            p1.tag = sim.countOfMutationsOfType(m2);
            cat("Cycle " + sim.cycle + ": " + p1.tag + " lineage(s)\n");

            if ((p1.tag == 0) & (sim.cycle > 1100)) {
                cat("Sweep failed to establish.\n");
                sim.simulationFinished();
            }
        }
    }
}
if (all(p1.genomes.countOfMutationsOfType(m2) > 0)) {
    cat("Soft sweep ended in cycle " + sim.cycle + "\n");
    cat("Frequencies:\n");
    print(sim.mutationFrequencies(p1, sim.mutationsOfType(m2)));
    sim.simulationFinished();
}
}
```

There's a lot going on here; let's unpack it. First of all, there are a bunch of `cat()` calls to print out diagnostic information about the sweep. A run of this model might produce output like:

```
Cycle 1000: 1 lineage(s)
Cycle 1010: 2 lineage(s)
Cycle 1020: 3 lineage(s)
Cycle 1040: 4 lineage(s)
Cycle 1041: 3 lineage(s)
Cycle 1042: 2 lineage(s)
Soft sweep ended in cycle 1067
Frequencies:
0.638 0.362
```

Each time a new copy of the sweep mutation is introduced, it produces a new lineage that is conceptually distinct from other lineages containing the same mutation; although each copy of the mutation is identical (same position, same selection coefficient, etc.), SLiM tracks each introduced copy separately, since each is generated with a separate call to `addNewDrawnMutation()`; see section 1.5.2 for discussion of this. (If you wanted SLiM to simulate just a single mutation object, without tracking lineages in this way, you could look up the existing mutation object and add it to new individuals with the `addMutation()` method instead, as we will see in the next section.) As in section 9.5.1, we use a “magic” value for the dominance coefficient  $h$  so that  $(1+hs)^2=1+s$ , so the calculated fitness values for individuals heterozygous for two different mutational lineages are correct. In the next section we will see how to collapse mutational lineages together, as another way of avoiding this issue, but here we are specifically interested in the distinct lineages and do not wish to collapse them.

As the output shows, the number of lineages increases as new copies get added during the soft sweep; sometimes the lineage count goes down, though, as particular lineages go extinct due to genetic drift. The model uses a value stored in `p1.tag` to track the number of lineages; each time that the current lineage count differs from that tag value, a new output line is generated. That's the function of the first half of the `1:10000` event: to track the number of lineages, produce output when it changes, and halt the simulation if a hard sweep completes from a single introduced mutation (indicated by the existence of a substitution of type `m2`) or if the sweep fails to establish (indicated by a lack of active sweep mutations, if we're past the end of the mutation introduction period).

The second half of the `1:10000` event detects when the soft sweep has completed, and prints a message giving the cycle at which it completed, along with the frequencies of each of the mutational lineages that ended up being part of the completed sweep. These frequencies sum to 1, indicating that every individual in the population possesses mutations from one or another of those lineages; given that all of the mutational lineages are genetically identical, this means that the mutation has really fixed, even though it is divided into distinct lineages by the design of the simulation. The way this event detects completion might need a bit of explanation. First, `p1.genomes` gives a vector containing the genomes for all individuals in the subpopulation. The method `countOfMutationsOfType(m2)` performs a count on each genome in that vector, and returns a new vector containing the corresponding counts. The `> 0` test then generates a logical vector containing T for each genome that had a copy of the sweep mutation; if any genome is still missing the sweep mutation, this vector will have an F in that position. Finally, `all()` returns T if all of the elements of that logical vector are T; if any genome failed the test, `all()` returns F. You can compare this to the strategy for detecting soft sweep fixation used in the previous section, which actually computes the frequency of the sweep.

The `1000:1100` event is the engine that generates the new mutational lineages of the soft sweep. For simplicity, it starts a new lineage every 10 cycles, using the modulo operator, `%`, to test for

cycles which are evenly divisible by 10; it would of course be trivial to modify this to generate the new lineages at whatever fixed ticks or cycles you wished, or to generate a new lineage randomly with a given probability. In any case, when it is decided that a new lineage should be created this event adds a new mutation, identical to the previous mutations, to a randomly chosen genome, much as we have seen before.

The only tricky bit here is that we want to prevent more than one sweep mutation from existing in the same genome, so before we add the new mutation, we check whether one is already there. Normally SLiM allows two mutations to occupy exactly the same position in the chromosome (see section 1.5.3); the check here avoids that possibility, so as to guarantee that the final lineage frequencies will sum to exactly 1. In the previous section, we used the `mutationStackPolicy` property of `MutationType` to prevent such “stacked” mutations; that solution would work equally well here, since the mutation type’s stacking policy applies to introduced mutations as well as to mutations generated by SLiM. A different strategy is employed here simply to show a different approach to the same issue.

### 9.5.3 A soft sweep with a random mutation schedule

As mentioned above, the previous recipe could be modified to follow a random schedule quite easily; for example, the `(sim.cycle % 10 == 0)` test could be changed to `(runif(1) < 0.1)` to provide a new mutational lineage at random times averaging every ten ticks. But perhaps your model would like to know the mutation schedule ahead of time, for some reason; you would like to have, at the outset, a list of the ticks in which a new lineage will be created. This would allow you to prune out schedules that didn’t satisfy some criterion, or run statistics on the schedule in advance, or other such tasks. Here we look at an alternative recipe, then, that generates a schedule ahead of time and then uses it to generate lineages.

Sections 9.5.1 and 9.5.2 used a “magic” dominance coefficient to guarantee that  $(1+hs)^2=1+s$ . This allowed them to keep the various mutational lineages of the soft sweep distinct, while still producing the correct fitness value for individuals that were heterozygous for each of two different mutational lineages (see those sections and section 1.5.2 for further discussion of this problem). Here we will take a different approach: we will collapse the mutational lineages into one, by reusing the already existing `m2` sweep mutation whenever it is present, rather than creating a new `m2` mutation for each introduction. This guarantees that SLiM will calculate correct fitness values even though we use a dominance coefficient of 0.1 here (which is far from the “magic” value of 0.45 for the selection coefficient chosen), at the cost of losing the information about each distinct lineage. As a result, the sweep mutation will be fix and be substituted by SLiM when the sweep completes, unlike in the previous sections, and we won’t be able to print out the frequencies for each lineage that comprised the sweep; but in terms of the evolutionary dynamics it is still a soft sweep model since the same sweep mutation can arise on multiple genetic backgrounds as the sweep is in progress. To get the best of both worlds – to model the mutational lineages independently while also getting correct fitness values without using a “magic” dominance coefficient – we could essentially treat the mutational lineages as being epistatic; see section 10.3.1 for discussion of modeling epistasis in SLiM.

Here’s the complete recipe:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.1, "f", 0.5);    // sweep mutation
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
```

```

1 early() {
    sim.addSubpop("p1", 500);

    gens = cumSum(rpois(10, 10)); // make a vector of start gens
    gens = gens + (1000 - min(gens)); // align to start at 1000
    defineConstant("Z", max(gens)); // remember the last gen

    for (gen in gens)
        community.registerLateEvent(NULL, s1.source, gen, gen);

    community.deregisterScriptBlock(s1);
}
s1 1000 late() {
    target = sample(p1.genomes, 1);
    mut = sim.mutationsOfType(m2);
    if (mut.size() > 0)
        target.addMutations(mut);
    else
        target.addNewDrawnMutation(m2, 10000);
}
1:10000 late() {
    if (any(sim.substitutions.mutationType == m2))
    {
        catn("Sweep completed in cycle " + sim.cycle + ".");
        sim.simulationFinished();
    }
    else if ((sim.countOfMutationsOfType(m2) == 0) & (sim.cycle > Z))
    {
        catn("Soft sweep failed to establish.");
        sim.simulationFinished();
    }
}
}

```

The 1:10000 event checks for loss or completion of the sweep and reacts accordingly, much as we have seen previously. More interesting is how the lineage addition works now. There is an event declared using a syntax that has not been seen up until now:

```
s1 1000 late() {
```

This event is *named*; an Eidos constant, `s1`, will be defined as referring to the script block for the event. To some extent this is just a shorthand for convenience; all defined script blocks in a simulation are available through `community.allScriptBlocks`, as objects of type `SLIMEidosBlock`, and this block could be looked up from that vector using the properties of that class (by looking for a block running only in tick 1000, for example). Being able to just use the name `s1` is obviously more convenient, and mirrors the syntax of being able to refer to a subpopulation as `p1`, a genomic element type as `g1`, or a mutation type as `m1`.

Putting that aside, this event schedules the addition of a new lineage in much the same way as in the previous recipe. Notably, however, the tick 1000 event now schedules a new lineage just once. The trick behind this recipe is that this block gets re-scheduled by the model to run in an assortment of other ticks. The code to do this is in the tick 1 event. First, a vector of all the ticks in which the event will run is constructed by drawing from a Poisson distribution to get waiting times, using `rpois()`, and then calculating cumulative sums from that vector of waiting times, using `cumSum()`; the Eidos documentation gives details on these functions, of course. This vector is then realigned so that its smallest element is equal to 1000, giving us the final schedule of lineage additions that will be followed. The event then loops through that vector, and for each tick in it,



calls `community.registerLateEvent()` to schedule a lineage addition in that tick. It does this by scheduling the source code from the `s1` event over again, obtained simply with `s1.source`. Finally, the `s1` event itself is deregistered; the loop has already scheduled a lineage addition in tick 1000, so `s1` is not needed. In this way, `s1` itself never runs at all, but its source code is used as a template for ten other events that do run.

The fact that Eidos is an interpreted language gives you the freedom to play all sorts of games like this with the code of your simulation, even as it runs; you can register new events and callbacks, deregister existing ones, and even generate code on the fly and then execute it with the `executeLambda()` command. These sorts of techniques should be used in moderation, as they can make the code of a model difficult to understand and maintain; but in some situations, as here, they can prove exceedingly useful for implementing highly dynamic model behavior.

Incidentally, SLiMgui provides graphical tools for inspecting the event list, which can be useful for understanding the behavior of models like this. This feature was previously described in section 5.1.2, in a very different context; you might try opening the window drawer as described in that section, and then recycling this model and stepping through tick 1. You will see that initially, the event list contains `s1`; after tick 1 has executed, `s1` has disappeared from the list, because it has been deregistered, and ten new events have been added programmatically.

So the `s1` event performs the introduction of the sweep mutation in a particular tick. It does so a bit differently than we have seen before, however. After choosing a target genome, it attempts to look up the existing sweep mutation with `mutationsOfType()`. If that does find an existing mutation, that mutation will be added to the target genome with `addMutations()`, reusing it rather than creating a new mutational lineage as we did in previous sections. Only if it fails to find an existing sweep mutation does it create a new mutation with `addNewDrawnMutation()`. This strategy avoids the fitness calculation issue with multiple mutational lineages discussed above; it also avoids any issue with “stacking” of the introduced mutations, so we don’t need to change the stacking policy here or otherwise worry about stacking. (If the target genome already contains the existing `m2` mutation, `addMutations()` does nothing, since a given mutation can exist only once in any one genome.)

## 9.6 Sweeps from standing genetic variation

The previous section showed recipes for simulating soft selective sweeps with *de novo* mutations – either introduced explicitly, or resulting from the normal mutational processes of SLiM. Soft sweeps can also originate from standing genetic variation, however. In this case, a mutation that was previously neutral becomes beneficial (presumably due to a change in the environment), suddenly exposing the mutation to selection. Because the mutation was initially neutral, it will generally occur in the population against a variety of genetic backgrounds; a soft selective sweep will therefore generally occur, although a hard sweep is possible if just one of the mutation-containing lineages happens to eliminate all of the others.

There are several ways that a sweep from standing genetic variation might be modeled. We will examine two recipes in this section: a sweep from standing variation at a randomly chosen locus, and a sweep from standing variation at a predetermined locus that is triggered when a mutation at that locus reaches a threshold frequency.

### 9.6.1 A sweep from standing variation at a random locus

In this first recipe, a randomly chosen mutation will be picked out of the standing neutral genetic diversity that has accumulated in the model; the only criterion for the selection of the mutation will be that it must be above a threshold frequency. The chosen mutation will be



changed to be beneficial (reflecting an environmental change), and the model will terminate when the mutation has either been lost or has completed a sweep:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 1.0, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
}
1000 late() {
  muts = sim.mutations;
  muts = muts[sim.mutationFrequencies(p1, muts) > 0.1];

  if (size(muts))
  {
    mut = sample(muts, 1);
    mut.setSelectionCoeff(0.5);
  }
  else
  {
    cat("No contender of sufficient frequency found.\n");
  }
}
1000:10000 late() {
  if (sum(sim.mutations.selectionCoeff) == 0.0)
  {
    if (sum(sim.substitutions.selectionCoeff) == 0.0)
      cat("Sweep mutation lost in cycle " + sim.cycle + "\n");
    else
      cat("Sweep mutation reached fixation.\n");
    sim.simulationFinished();
  }
}

```

The tick 1000 event chooses the target mutation and transmogrifies it. The `muts` variable is set up initially as the full set of mutations in the simulation, and is then whittled down to only those mutations above a minimum threshold frequency of 0.1. Note that it would be straightforward to modify this further and also require that the mutation is below a given maximum threshold frequency; in that case, we would be studying a scenario of a selective sweep from a standing genetic variant with starting frequency within the interval (min, max). The `sample()` function is then used to choose one mutation from that set, at random, and the chosen mutation has its selection coefficient altered to be beneficial. Note that this event is a `late()` event; this is for essentially the same reasons as explicated in section 4.2.1. In short, we are changing the selection coefficient of an existing mutation; for the fitness effect of that change to be realized immediately, it must happen in a `late()` event so that the change occurs after offspring generation but before fitness recalculation.

The 1000:10000 event just checks for termination conditions: either the chosen mutation has been lost, or it has fixed. Since a separate mutation type is not used for the sweep mutation in this model, unlike the other sweep models we have seen, these termination conditions are detected using the property that the chosen mutation has a non-zero selection coefficient.

This could be tailored in all sorts of ways; for example, the transition from neutral to beneficial could be made gradual, or the new selection coefficient could be drawn from a distribution rather than being a fixed value.

### 9.6.2 A sweep from standing variation at a predetermined locus

In this recipe we want to model a soft sweep from standing genetic variation. Specifically, we want to assume that a previously neutral mutation suddenly becomes beneficial as a consequence of an environmental change, and then subsequently sweeps in the population. In this scenario, we can choose the “starting frequency” of the mutation at the time the environment changes. Our recipe for this scenario is based on the conditional-on-establishment machinery of section 9.3, which restarts the simulation if the mutation is lost prior to reaching the chosen starting frequency (here defined as a frequency of 0.1). If the mutation does manage to drift to this frequency, it is then converted to be beneficial. After that point, the model runs without conditionality, and detects whether the mutation fixes or is lost prior to tick 10000.

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 1.0, "f", 0.0); // introduced mutation
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  // save this run's identifier, used to save and restore
  defineConstant("simID", getSeed());

  sim.addSubpop("p1", 500);
}
1000 late() {
  // save the state of the simulation
  sim.outputFull(tempdir() + "slim_" + simID + ".txt");

  // introduce the sweep mutation
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m2, 10000);
}
1000: late() {
  mut = sim.mutationsOfType(m2);

  if (size(mut) == 1)
  {
    if (sim.mutationFrequencies(NULL, mut) > 0.1)
    {
      cat(simID + ": ESTABLISHED – CONVERTING TO BENEFICIAL\n");
      mut.setSelectionCoeff(0.5);
      community.deregisterScriptBlock(self);
    }
  }
  else
  {
    cat(simID + ": LOST BEFORE ESTABLISHMENT – RESTARTING\n");

    // go back to tick 1000
    sim.readFromPopulationFile(tempdir() + "slim_" + simID + ".txt");
  }
}
```

```

// start a newly seeded run
setSeed(rdunif(1, 0, asInteger(2^62) - 1));

// re-introduce the sweep mutation
target = sample(p1.genomes, 1);
target.addNewDrawnMutation(m2, 10000);
}
}
1000:10000 late() {
  if (sim.countOfMutationsOfType(m2) == 0)
  {
    fixed = (sum(sim.substitutions.mutationType == m2) == 1);
    cat(simID + ifelse(fixed, ": FIXED\n", ": LOST\n"));
    sim.simulationFinished();
  }
}
}

```

Most of the code here is identical to the recipe in section 9.3, so please refer to that section for further discussion. Compared to that recipe, mutation type `m2` now has an initial selection coefficient of `0.0`, making it neutral. If the mutation reaches the threshold frequency of `0.1`, an extra line of code calls `setSelectionCoeff()` to convert the mutation to beneficial. Finally, a `1000:10000` event has been added that checks for fixation or loss. Note that this event runs after the `1000:` event, because it is declared later in the source code; the `1000:` event catches cases of loss prior to establishment and restarts the model before the `1000:10000` event notices the loss.

If you run this model, it will probably emit a long string of restart messages before it fixes, due to loss of the introduced mutation while it is still neutral. Once it reaches the threshold frequency and is converted to beneficial, it will usually fix. Typical output is therefore something like:

```

1460585630465: LOST BEFORE ESTABLISHMENT – RESTARTING
1460585630465: LOST BEFORE ESTABLISHMENT – RESTARTING
...
1460585630465: LOST BEFORE ESTABLISHMENT – RESTARTING
1460585630465: LOST BEFORE ESTABLISHMENT – RESTARTING
1460585630465: ESTABLISHED – CONVERTING TO BENEFICIAL
1460585630465: FIXED

```

It would be simple to convert the introduced mutation to be initially deleterious; the model would still work. In that case, however, it would usually take a very long time to complete a run, because the introduced mutation would frequently be lost unless we chose a very low threshold frequency or a very small deleterious selection coefficient.

## 9.7 Adaptive introgression

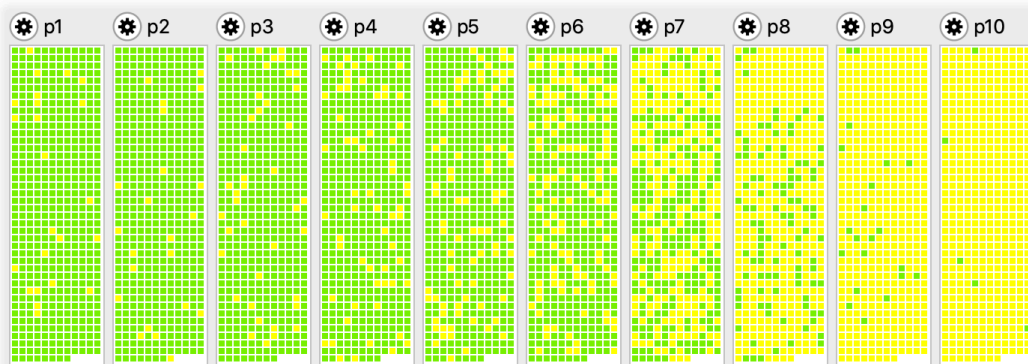
All of the selective sweep recipes thus far have involved just a single subpopulation. Here we want to extend this to model adaptive introgression, the progress of an adaptive allele from its subpopulation of origin into another subpopulation via migration and gene flow. Section 5.3 introduced techniques for setting up structured populations with migration; all we need to do is add a selective sweep to such a model. Here's a simple model combining the population structure of section 5.3.1's recipe with the hard sweep dynamics of section 9.1 (both slightly modified) to model adaptive introgression:

```

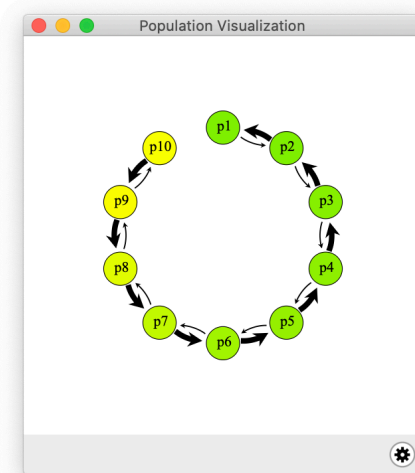
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    subpopCount = 10;
    for (i in 1:subpopCount)
        sim.addSubpop(i, 500);
    for (i in 2:subpopCount)
        sim.subpopulations[i-1].setMigrationRates(i-1, 0.01);
    for (i in 1:(subpopCount-1))
        sim.subpopulations[i-1].setMigrationRates(i+1, 0.2);
}
100 late() {
    target = sample(p1.genomes, 1);
    target.addNewDrawnMutation(m2, 10000);
}
100:100000 late() {
    if (sim.countOfMutationsOfTypes(m2) == 0)
    {
        fixed = (sum(sim.substitutions.mutationType == m2) == 1);
        cat(ifelse(fixed, "FIXED\n", "LOST\n"));
        sim.simulationFinished();
    }
}

```

If you run this model and stop the run in the middle, while the adaptive introgression is in progress (assuming it is not lost by chance early on – you may have to recycle and restart several times before it catches), the population shown in SLiMgui may look something like this:



Individuals possessing the introgressing allele are colored green; note that there are no intermediate-colored individuals because *m2* is fully dominant. The mutation was introduced into subpopulation *p1*, and so it is closest to fixation there, whereas it has only just begun to sweep in *p10*. Of course it is not possible to see the connectivity between the subpopulations, and the migration rates between them, in this view. As described in section 5.1.3, you can open the population visualization window to see what the population structure looks like:



Here we see that we have a stepping-stone model, with gene flow mostly from p10 toward p1, opposing the spread of the introduced mutation. Nevertheless, the mutation introgresses quite rapidly given the parameters for this model; you could play with migration rates, selection coefficients, etc. to test how they affect the time until completion of the sweep. It would be easy to introduce spatial variation in selection into this model, too, using the techniques of section 10.2, in order to allow the introgression to proceed only so far along the chain of subpopulations before further introgression is blocked by local selection against it. Section 12.3 provides an example of such spatial variation resisting the spread of an introgressing mutation, and then models a CRISPR gene drive that allows the introgression to proceed despite the selection against it.

## 9.8 Fixation probabilities under Hill–Robertson interference

In this final section, we’ll see how a simple sweep model can test the predictions of population genetic theory. The recipe here will model Hill–Robertson interference, the interference of beneficial mutations that have arisen in different lineages and thus compete against each other (Hill & Robertson 1966). The recipe’s code then compares the predicted mean fixation time for a beneficial allele without such interference to the mean fixation time observed by the simulation.

The design of this recipe is that the first 5000 ticks of the run are a burn-in period during which a dynamic equilibrium is reached, and then the final 1000 ticks are measured. The formulas for the probability of fixation of a beneficial mutation without interference and the expected number of fixed mutations are calculated according to standard population genetic theory (Kimura 1962). The actual number of fixed mutations is obtained from the information stored in SLiM’s substitution objects, which record the tick of fixation of all mutations; the number of fixed mutations with a tick of fixation after the end of the burn-in can then be counted. As we shall see below, the actual count is much lower than the expected count; fixation is being highly suppressed in this model compared to the expectations without Hill–Robertson interference.

Without further ado, the recipe:

```
initialize() {
  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.05);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
```

```

sim.addSubpop("p1", 1000);
}
6000 late() {
// Calculate the fixation probability for a beneficial mutation
s = 0.05;
N = 1000;
p_fix = (1 - exp(-2 * s)) / (1 - exp(-4 * N * s));

// Calculate the expected number of fixed mutations
n_gens = 1000; // first 5000 ticks were burn-in
mu = 1e-6;
locus_size = 100000;
expected = mu * locus_size * n_gens * 2 * N * p_fix;

// Figure out the actual number of fixations after burn-in
subs = sim.substitutions;
actual = sum(subs.fixationTick >= 5000);

// Print a summary of our findings
cat("P(fix) = " + p_fix + "\n");
cat("Expected fixations: " + expected + "\n");
cat("Actual fixations: " + actual + "\n");
cat("Ratio, actual/expected: " + (actual/expected) + "\n");
}

```

Running this produces output like this (a typical result):

```

P(fix) = 0.0951626
Expected fixations: 19032.5
Actual fixations: 302
Ratio, actual/expected: 0.0158676

```

This is obviously a very simple model; the point is just to demonstrate that it is straightforward to make models that test theoretical predictions like this – and that Hill–Robertson interference can make a large difference to dynamics! If you run the model in SLiMgui, the way that the competition of different lineages slows progress towards fixation is quite apparent. Of course if the mutational input is constant, and the time to fixation increases, that has to mean (assuming equilibrium) that fewer mutations are fixing. That can also be observed with this model in SLiMgui; you can see many substantial bars, representing beneficial mutations at reasonably high frequencies, getting pushed down to zero by the rise of a competing haplotype. Far fewer beneficial mutations would be lost if they were not competing with each other. Recombination occasionally resolves these conflicts by bringing competing mutations together onto the same chromosome.

A very simple way to check this model is to decrease the mutation rate. The lower the mutation rate, the less competition there should be between different mutations existing simultaneously in the population (to the limiting case of a single extant mutation at any given time, which would experience no Hill–Robertson interference at all). This is very easy to test; just change the mutation rate to  $1e-7$ , both where it is set with `initializeMutationRate()` and where it is assigned to the variable `mu` for the calculations at the end. With both of those set to  $1e-7$ , typical output is something like:

```

P(fix) = 0.0951626
Expected fixations: 1903.25
Actual fixations: 88
Ratio, actual/expected: 0.0462367

```



And if we change the rate to 1e-8, again in both places, we get:

```
P(fix) = 0.0951626
Expected fixations: 190.325
Actual fixations: 16
Ratio, actual/expected: 0.0840667
```

With 1e-9, we get something like this (with a lot of stochasticity in the result, at this point – 1000 ticks post-burn-in is not nearly long enough to get an accurate estimate of the model's behavior when the mutation rate is so low):

```
P(fix) = 0.0951626
Expected fixations: 19.0325
Actual fixations: 6
Ratio, actual/expected: 0.31525
```

So as predicted, the lower the mutation rate, the less Hill–Robertson interference influences the dynamics, and the more closely the model approximates the theoretical ideal of independent mutations without interference. And indeed, if you run the model with the mutation rate of 1e-9 in SLiMgui, you will see that sometimes multiple mutations still interfere, but fairly often, too, single mutations arise and fix individually. Of course a rigorous analysis would want to use a longer burn-in, a longer post-burn-in runtime, multiple runs averaged together for each mutation rate, calculation of a standard error of the mean across each set of multiple runs, statistical tests for significance of differences, and so forth; but even this very simple analysis is sufficient to make the trend quite clear.

## 9.9 Keeping a reference to a sweep mutation

Throughout this chapter, we have followed the technique of looking up the sweep mutation when we need it. In section 9.1, for example, we used `sim.countOfMutationsOfType(m2)` to determine whether the sweep mutation was still segregating (if so, the count would be 1), and we used `sum(sim.substitutions.mutationType == m2)` to determine whether the sweep mutation had fixed or been lost, if it was no longer segregating. Other sections used somewhat different techniques, but always followed the general approach of *looking up* the sweep mutation(s). In this section we'll look at an alternative technique: *keeping a reference* to a sweep mutation.

The general idea here is very straightforward: when we introduce the sweep mutation, we will just store the mutation object itself for future reference, using `defineConstant()`. We could remember it inside the dictionary of an object like `sim`, instead, with `setValue()` / `getValue()`; that also works fine. Let's look at this with a straightforward re-implementation of the recipe from section 9.1:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
1000 late() {
  target = sample(p1.genomes, 1);
  mut = target.addNewDrawnMutation(m2, 10000);
  defineConstant("SWEEP", mut);
}
```

```

1000:100000 late() {
  if (!SWEEP.isSegregating)
  {
    cat(ifelse(SWEEP.isFixed, "FIXED\n", "LOST\n"));
    sim.simulationFinished();
  }
}

```

The first thing to notice is the new call to `defineConstant()`, which sets up a new constant named `SWEEP` that references the newly generated sweep mutation. When the sweep mutation eventually fixes or is lost, the mutation will be removed from the simulation, but `SWEEP` will continue to be defined; it will continue to refer to the sweep mutation, allowing us to work with it.

The second thing to notice is how convenient this makes the remainder of the recipe. We can now find out whether the sweep mutation is still segregating in the simulation by simply asking it, with the `isSegregating` property of `Mutation`. If it isn't, we can find out whether it has fixed with the `isFixed` property; if that is F, the the mutation must have been lost. Not only is this more convenient, it is also faster; we no longer have to do time-consuming lookups to find the sweep mutation, or the `Substitution` object derived from it, because we have the mutation object itself readily at hand.

You might wonder why we haven't been using this technique throughout the chapter. One reason, in truth, is because the ability to do this is new, as of SLiM 3.5. Previously, `defineConstant()` and `setValue()` would not accept a value of type `object`; now they do (although only `Chromosome`, `Mutation`, and `Substitution` objects are presently legal; other classes will still cause an error). In general, SLiM objects have a lifetime that is defined by the dynamics of the model; when an individual dies, or a mutation is lost, those objects are removed from the simulation and cease to exist, and so allowing permanent references to objects to be kept presents difficulties (see section 27.2). But with version 3.5, SLiM has been specifically redesigned to make this possible for `Mutation` (and for `Chromosome` and `Substitution`), because it's so useful.

But there are other reasons we haven't been using this technique, too. In particular, it's a bit tricky to use with models that restart the simulation from a saved point, as used in section 9.2 and 9.3 to enforce "conditionality", for example. The defined constant `SWEEP` would live on past the reload of the model, and would then refer to a mutation that used to exist in an "alternate timeline". This poses no real problems, but it's unlikely to be what you want. To work around this difficulty, you would need to fix the constant each time the model restarts, probably by remembering it with `defineGlobal()` instead of `defineConstant()`, and then calling `defineGlobal()` again when it changes. This is probably a good approach in many cases; but it was deemed too conceptually complex for the introductory selective sweep models in this chapter.

With those caveats, though, it's a very useful technique! You can remember whole vectors of mutations; for example, it's trivial to write a model that remembers every mutation that exists in one tick X, with `defineConstant()`, and then checks in a later tick Y to see which of those mutations have fixed, which have been lost, and which are still segregating (see the next section). You can track all of the sweep mutations you've introduced, or use this facility to track other important mutations in your model without having to look them up from their `id` or their mutation type. This would be useful for tracking a "marker mutation", such as the chromosomal inversion marker used in section 14.4.

## 9.10 Tracking the fate of background mutations

In section 9.9 we introduced the technique of keeping a reference to a mutation such as the sweep mutation. Here we will extend this technique to keep references to all of the mutations that

comprise the background of a hard selective sweep, to see what happens to them all. This turns out to be quite straightforward. The complete recipe:

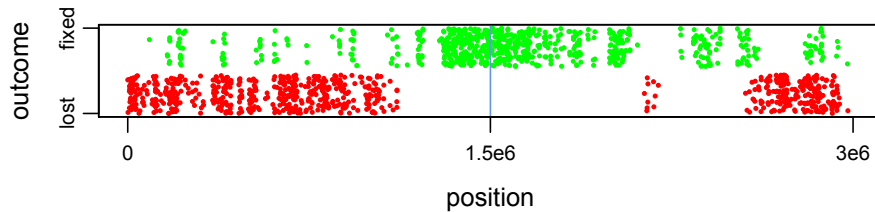
```
initialize() {
  defineConstant("L", 3e6);
  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.8, "f", 0.5);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-7);
}
1 early() {
  sim.addSubpop("p1", 500);
}
1000 late() {
  target = sample(p1.genomes, 1);
  defineConstant("BACKGROUND", target.mutations);
  mut = target.addNewDrawnMutation(m2, asInteger(L/2));
  defineConstant("SWEEP", mut);
}
1000: late() {
  if (!SWEEP.isSegregating & !SWEEP.isFixed)
    stop("LOST");
}
1500 late() {
  nonSeg = BACKGROUND[!BACKGROUND.isSegregating];
  fixed = nonSeg[nonSeg.isFixed];
  lost = nonSeg[!nonSeg.isFixed];
  writeFile("fixed.txt", paste(fixed.position, sep=", "));
  writeFile("lost.txt", paste(lost.position, sep=", "));
}
```

We let neutral diversity build up for 1000 ticks, and then introduce a sweep mutation of type m2 in tick 1000. In the 1000 late() event, we first choose a single genome as the target, and then we define a new constant, BACKGROUND, that contains the entire vector of mutations that target currently contains – the genetic background into which the sweep mutation will be introduced. Then we introduce the sweep mutation with addNewDrawnMutation(), and we remember it as the defined constant SWEEP. With this setup, we’re now ready to track the fate of the sweep itself, as well as all of the background mutations.

We monitor the sweep itself in the 1000: late() event, by just watching for the sweep mutation switching to a state that indicates it has been lost (not segregating and not fixed), in which case we stop with an error. You may therefore need to run the model several times to get a sweep that completes; we could make this model run conditionally on fixation, using the technique of section 9.2, but we would have to be careful that SWEEP and BACKGROUND represented the correct mutations after restarting the sweep (further discussed in section 9.9).

Once we have a run where the sweep mutation is not lost, we stop in tick 1500 and assess the fates of the background mutations. We start by pruning the BACKGROUND vector down to those mutations that are no longer segregating, using the isSegregating property of Mutation. Then we subdivide that vector into those mutations that fixed, versus those that were lost, using the isFixed property. Finally, we write out the positions of the mutations in those two categories to two separate text files.

With a little bit of R code to read in the text files and generate a plot, we end up with this visualization of the background mutation fates:



The x axis is position along the chromosome, with the blue line at the center showing the position of the sweep mutation. The y axis represents just the two outcomes, lost or fixed, with the points (representing individual mutations) shown in corresponding colors of red and green; the vertical spread of the points is just jitter to make the individual points, and thus the density of mutations in different regions, more apparent.

This visualization makes it very clear that the central portion of the chromosome has been carried along to fixation by the hard sweep, while recombination has led the regions further away from the sweep mutation to fix much less often. This is just a snapshot at one instant in time, some ticks after the sweep mutation has fixed; one could use the same technique to look at the pattern of fixation and loss throughout the sweep and then onward in time, as regions that hitchhiked along with the sweep but didn't quite fix drift toward fixation or loss in later ticks. One could also look at the mutations within `BACKGROUND` that are still segregating, to see their distribution along the chromosome and monitor the length of time it takes for them to either fix or be lost, as a function of their distance from the sweep mutation.

The same sort of analysis could be done using tree-sequence recording; sections 17.3, 17.4, and 17.7 illustrate some of the techniques one would use, although a complete example is beyond the scope of this manual. If the ancestry trees are not needed, however, and one just wishes to track mutations directly, the technique shown here may prove simpler. The two approaches can also be used jointly; you could track mutations in `SLiM` directly, write out the mutation `id` values of mutations of interest, and then use those `id` values to look up the mutations in the tree sequence in Python during your downstream analysis.

This technique could also be applied to many other situations. For example, if your model involves a single migrant moving from one subpopulation to another, and you want to track the fate of all of the mutations carried by that migrant, you could remember the migrant's mutations with `defineConstant()` and then monitor them. You could similarly watch mutations from one region in a continuous-space model, to see how they spread across space and the spatial pattern in which they fix or are lost.

Finally, it's worth mentioning again that although we have been using `defineConstant()` to remember mutations, `setValue()` also works, and lets you associate a vector of remembered mutations with a particular `SLiM` object in your simulation; each subpopulation could remember its own vector of mutations of interest, for example.

## 9.11 Effective population size versus census population size

A key concept in population genetics is the *effective population size*, usually represented with the symbol  $N_e$ . We will explore that concept in this section with a simple model that shows the difference between  $N_e$  and the *census population size*, usually symbolized as  $N$ . Conceptually, this exploration might have fit better in chapter 5, since that chapter dealt with demography, but it is in this chapter instead because we will use a selective sweep to show how  $N_e$  and  $N$  can diverge, and how that divergence can be measured.

Let's start at the conceptual level. The census population size,  $N$ , is the actual number of individuals in a population: in empirical terms, it is how many organisms you would count, for

your species of interest, if you went out into the real world and counted every single one. This is the population size we have been using in our models thus far, because these are the terms that SLiM generally thinks in. SLiM is an individual-based simulation framework (see section 1.1); since that means that it explicitly models each individual in a population, the census population size is its natural metric. In Wright–Fisher models, such as we have been working with, the census population size is directly specified to SLiM as a model parameter, such as with the initial population size passed to `addSubpop()`. In non-Wright–Fisher models, which we have not encountered yet in any detail (but see the conceptual overview in section 1.6), the census population size is emergent from the birth–death dynamics of the model, rather than being a fixed parameter, but still, SLiM always thinks in terms of explicitly modeled individuals, and so it knows the exact census population size at all times.

The effective population size,  $N_e$ , is quite different. It is never specified as a parameter to SLiM; even in Wright–Fisher models it is an emergent property of the model, a metric, something that you measure rather than something that you dictate. It is not usually equal to  $N$ , and indeed, it can depart from  $N$  quite substantially. What it represents is a bit subtle:  $N_e$  is the census population size that an abstract, idealized population would need to have in order to match a given value of a particular metric. That given metric value is typically measured in, or estimated from, an actual population – in SLiM, the simulated population.

There are two key concepts here. One is that  $N_e$  involves an abstract, idealized population. This population embodies a bunch of assumptions: random mating, constant size, equal numbers of children per parent, no selection, and so forth. These assumptions need not be true of the actual population you are simulating in SLiM; rather, the point of  $N_e$  is to say “If these assumptions were true, what would we estimate the population size to be, given this particular metric of interest?” And that brings us to the second key point:  $N_e$  depends upon the choice of a particular metric of interest.  $N_e$  is not a single quantity; you can estimate  $N_e$  based upon coalescence time, or the amount of inbreeding in a population, or the amount of genetic variation present, or lots of other things. Estimating  $N_e$  in several different ways, using different metrics, will often produce different answers – sometimes very different! – and that is absolutely fine; different  $N_e$  estimates are not supposed to agree, in general, because each  $N_e$  estimate is an answer to a different question: “What size would an idealized population be if it had a given value of *this particular metric*?”

To make this a bit more concrete: in an idealized population, population genetics tells us that the expected heterozygosity,  $\pi$ , is equal to  $4N\mu$ , where  $N$  is the census size of the idealized population and  $\mu$  is the mutation rate (note that this formula is an approximation, based upon the assumption that  $\pi \ll 1$ ). Suppose we measure the heterozygosity  $\pi$  of our simulated population, and we know the mutation rate; this equation then allows us to *estimate* the population size  $N$ , and this estimate would be  $N_e$ ; in other words,  $N_e = \pi / 4\mu$ . This estimate is based upon the assumptions of the idealized population; the formula  $\pi = 4N\mu$  depends upon those assumptions, and so it is sometimes written instead as  $\pi = 4N_e\mu$  to emphasize that fact. We will call this  $N_e$  estimate the “heterozygosity  $N_e$ ”, in the discussion that follows; estimating  $N_e$  from heterozygosity is also sometimes called the “coalescence  $N_e$ ”, because  $\pi$  is four times the expected time to pairwise coalescence in an idealized diploid population, but there is some complexity surrounding this terminology (see Sjödin et al. 2005), so we will avoid it here.

Why is estimating  $N_e$  useful? After all, in a SLiM simulation you know what the actual census population size,  $N$ , is; why would you care what the size of some idealized population, involving a bunch of assumptions, would hypothetically be? There are several reasons. One is that many analytical models are based upon  $N_e$  rather than  $N$ ; if you want to parameterize such a model to compare it to the results of your simulation, you will therefore need to have an  $N_e$  estimate – if your simulation substantially departs from analytical assumptions, using  $N$  instead of  $N_e$  would



incorrectly parameterize the analytical model. Another is that in many empirical systems you don't know  $N$ ; populations are often too large to comprehensively survey, and estimating the true census population size using mark-recapture methods, etc., can be difficult and time-consuming, whereas estimating  $N_e$  from other data can be relatively straightforward. To make a simulation model of your empirical system, for which  $N$  is unknown, you will then want to model the biology of your system in SLiM, and try to parameterize your model (both  $N$  and perhaps other parameters) such that the  $N_e$  measured on your simulation matches the  $N_e$  that you estimated from your empirical data. This might even allow you to estimate the true value of  $N$  for your system, from the simulated  $N$  that produces the observed empirical  $N_e$ ; you could use a technique like ABC to converge upon a best estimate for  $N$  using simulations (see section 14.6).

That's enough of a digression into effective population size; it is a very complex topic, and the aim here is not to be a population genetics textbook. Reviews of the topic can be found in Wang, Santiago, & Caballero (2016) and Sjödin et al. (2005), for those who want to delve deeper. We now have the conceptual background we need to understand the recipe for this section. Here is the core of the model, which is fairly straightforward:

```
initialize() {
  defineGlobal("N", 1000);
  defineGlobal("L", 1e7);
  defineGlobal("MU", 1e-7);
  defineGlobal("R", 1e-8);
  defineGlobal("S", 2.0);
  initializeSLiMOptions(keepPedigrees=T);
  initializeMutationRate(MU);
  initializeMutationType("m1", 0.5, "f", 0.0);           // neutral
  initializeMutationType("m2", 0.5, "f", S);             // sweep
  m2.convertToSubstitution = F;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(R);
}
1 late() {
  sim.addSubpop("p1", N);
  p1.setValue("previous_N", p1.individualCount);

  defineConstant("LOG", community.createLogFile("Ne_log.csv"));
  LOG.addCycle();
  LOG.addCustomColumn("N(t-1)", "p1.getValue('previous_N')");
  LOG.addCustomColumn("N(t)", "p1.individualCount");
  LOG.addCustomColumn("freq", "mutTypeFrequency(m2)");
  LOG.addCustomColumn("Ne_heterozygosity", "estimateNe_Heterozygosity(p1)");
  LOG.addCustomColumn("Ne_inbreeding", "estimateNe_Inbreeding(p1)");
}
2: late() {
  LOG.logRow();
  p1.setValue("previous_N", p1.individualCount);
}
10000 late() {
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m2, integerDiv(L, 2));
}
20000 late() {
  sim.simulationFinished();
}
```



This recipe performs a trivial selective sweep by introducing a beneficial mutation; see section 9.1 for discussion of this techniques. It also uses the `LogFile` class to log out information about the simulation as it progresses; see section 4.2.5 for an introduction to `LogFile`. It does not condition upon fixation of the sweep mutation (see section 9.2), because that would write the same ticks to the `LogFile` more than once, which would slightly complicate the analysis; it may therefore be necessary to run the model more than once to get a good sweep. (As of this writing, calling `setSeed(1683065704505)` at the start of `initialize()` provides a good run.)

Missing from the script shown above are a handful of user-defined functions, used by `LogFile` to calculate the sweep frequency and two different flavors of  $N_e$  in each tick as configured by the `late()` event. Here are those functions:

```
function (float)mutTypeFrequency(o<MutationType>$ mutType)
{
  muts = sim.mutationsOfType(mutType);
  if (muts.size() > 0)
    return sim.mutationFrequencies(NULL, muts);
  return NULL;
}

function (float)estimateNe_Heterozygosity(o<Subpopulation>$ subpop)
{
  pi = calcHeterozygosity(p1.genomes);
  return pi / (4 * MU);
}

function (integer)tabulateFecundity(o<Subpopulation>$ subpop, i$ previous_N)
{
  parentIDs = subpop.individuals.pedigreeParentIDs;
  rescaledParentIDs = parentIDs - min(parentIDs);
  return tabulate(rescaledParentIDs, previous_N - 1);
}

function (float)estimateNe_Inbreeding(o<Subpopulation>$ subpop)
{
  previous_N = subpop.getValue("previous_N");
  k = tabulateFecundity(subpop, previous_N);
  return (previous_N * mean(k) - 2) / (mean(k) - 1 + var(k) / mean(k));
}
```

We haven't seen user-defined Eidos functions previously, but they are fairly straightforward as you can see above; you declare the function using the same function signature syntax that Eidos uses for its own built-in functions, with a return value type, a name for the function, and a list of typed parameters, and then you implement the function (see the Eidos manual for details).


The `mutTypeFrequency()` function just wraps SLiM's `mutationFrequencies()` method; it looks up the sweep mutation with `mutationsOfType()` given the mutation type passed to it in `mutType`, and then returns its frequency if it exists, or `NULL` if it doesn't. (`LogFile` writes out `NA` when a logger produces a value of `NULL`, which dovetails nicely with plotting in R.)

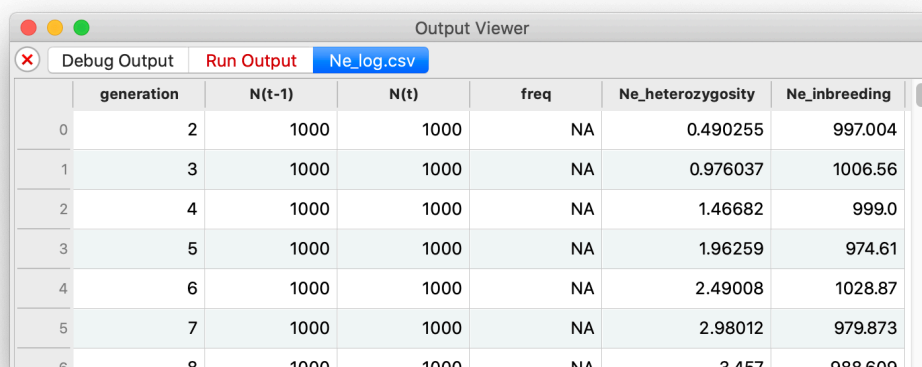
The `estimateNe_Heterozygosity()` function estimates  $N_e$  based upon the observed heterozygosity,  $\pi$ , in a given subpopulation, using exactly the formula that we discussed above. It uses SLiM's built-in `calcHeterozygosity()` function to obtain  $\pi$ ; this implementation uses all of the genomes in the subpopulation for that estimation, but of course you could use `sample()` to obtain just a subset of genomes if you wanted to mirror an empirical sampling procedure.

Next we have a helper function, `tabulateFecundity()`. This function is used by the other  $N_e$  estimator; it has been split out into its own function since it might have broader utility. It looks up the parent pedigree IDs for all individuals in the given subpopulation; these are integer identifiers that uniquely identify every individual in the simulation. This is the reason for the call to `initializeSLiMOptions(keepPedigrees=T)` in `initialize()`; that turns on this “pedigree tracking” feature so that we can obtain these pedigree IDs (see sections 14.1 and 24.1 for further discussion; since this is just an incidental feature of this model, we won't focus on it here). So now we have a vector containing everybody's parental pedigree IDs, and that vector might have many duplicates; every time that a parental individual with a pedigree ID of 17, for example, had an offspring, 17 would appear again in this vector. Pedigree IDs start at 0, at the start of a run, and count upwards; after a little while they can get rather large. We are only interested in the parental pedigree IDs presently in use, so we want to rebase our vector to have a minimum value of 0; that is the purpose of the next line. Finally, we use the `tabulate()` function of Eidos to tabulate the number of occurrences of each value in the vector; this counts the number of times that each parent produced an offspring, and produces a new vector containing those counts. This is the raw material from which the next  $N_e$  estimator does its work.

And then we have the functions `estimateNe_Inbreeding()`; this is the remaining  $N_e$  estimator. We will not discuss the mathematics upon which it is based, except to say that it uses the fecundity tabulation to estimate  $N_e$  from the amount of inbreeding present, which is related to the distribution of fecundity among the parental generation. It was kindly provided by Miguel Navascués for this recipe; he says it is based upon a formula in a book chapter by Waples, in a book titled “Population Viability Analysis” from 2002. Note that its implementation in a nonWF model would need to be adjusted, particularly with overlapping generations.

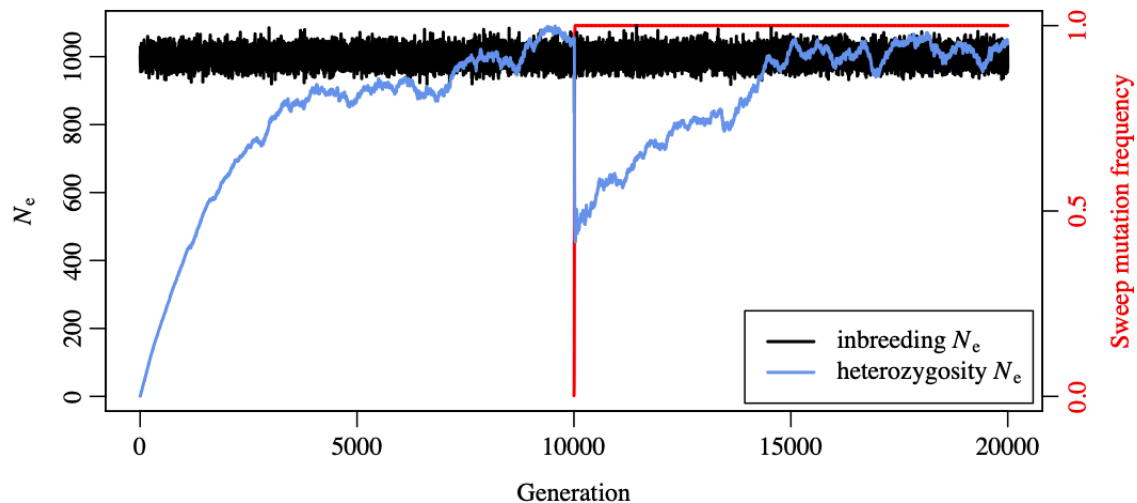
Rather than dwelling further upon the two particular  $N_e$  estimators that we have provided in this recipe, we will just say: there are many estimators for  $N_e$ , and little agreement upon them. As far as we know these estimators are correct, in the sense of not containing mathematical errors, but we have no doubt that there are those who would disagree with their methodology, their philosophy, or their implementation. Such debate may be useful, but it is not our focus here. Again, please consult the reviews cited above for further depth on this topic.

So, let's get to results! First of all, we can look at the LogFile output in SLiMgui, as in section 4.2.5, by clicking the debugging output viewer button  and choosing the LogFile's tab at the top of the output viewer. The beginning of it looks like this, for one run of the model:



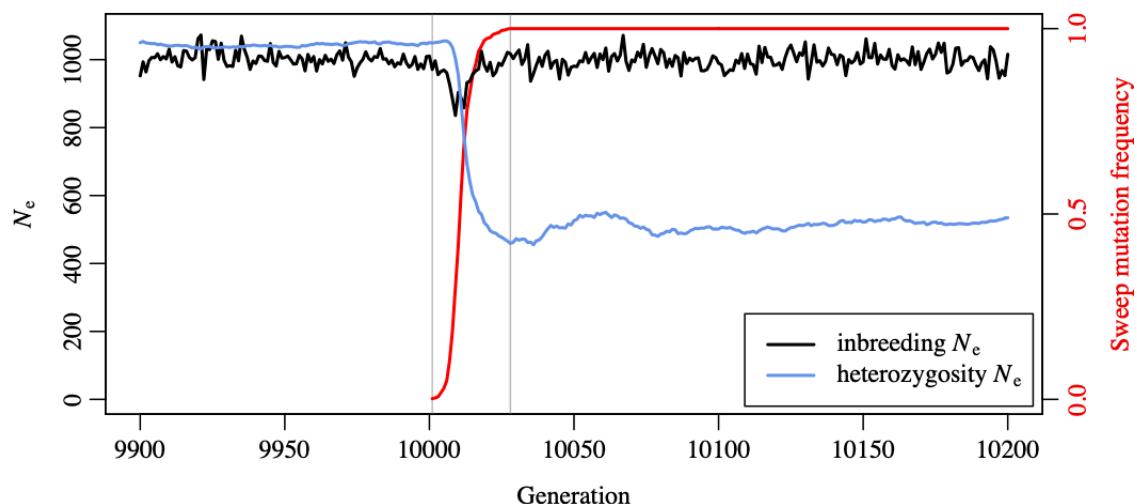
	generation	N(t-1)	N(t)	freq	Ne_heterozygosity	Ne_inbreeding
0	2	1000	1000	NA	0.490255	997.004
1	3	1000	1000	NA	0.976037	1006.56
2	4	1000	1000	NA	1.46682	999.0
3	5	1000	1000	NA	1.96259	974.61
4	6	1000	1000	NA	2.49008	1028.87
5	7	1000	1000	NA	2.98012	979.873

So far so good; all of our custom logging functions appear to be working. (The freq column is NA because the sweep mutation has not yet been introduced.) We can easily plot this CSV output in R; the script for doing so is not shown here, but the resulting plot looks like this:



The x-axis here spans 20,000 generations, so the sweep here is just a vertical red line at generation 10,000. The inbreeding  $N_e$  fluctuates around the true value of  $N$ , 1000, due to stochasticity during reproduction, since it is based upon the distribution of fecundity in each generation; it shows no visible effect from the sweep here (but see below). The heterozygosity  $N_e$ , on the other hand, depends upon the level of genetic variation in the population, which rises from zero rather slowly; this is why we have a burn-in period of 10,000 generations in this model before the sweep mutation is introduced. This implies, of course, that estimating  $N_e$  from heterozygosity, or any metric of genetic variation, may be problematic when a model has not yet completed its burn-in and attained an equilibrium level of variation; but the estimator is not *wrong*, it is simply telling us that during the burn-in period an idealized population with the same amount of genetic variation at equilibrium would be relatively small, and that is true. This estimator is also rather stochastic, since the genetic variation rises and falls as new mutations arise and old mutations fix or are lost. When the sweep occurs, however, the heterozygosity  $N_e$  estimator drops very sharply, to approximately 400, and then rises again to equilibrium. This reflects the temporary loss of genetic diversity in the population due to this rather strong hard sweep.

Let's zoom in on the sweep itself, with another plot of the same dataset using a much narrower x-axis range:



The sweep's frequency over time is now resolved into a red sigmoid curve; the gray vertical lines demarcate the time period during which the sweep mutation was segregating. It can now be seen that the inbreeding  $N_e$  does, in fact, dip down in response to the sweep, reflecting the fact that individuals carrying the sweep mutation had higher reproductive success than individuals that didn't, which generates elevated inbreeding in the population that is detected by the estimator. This dip is only down to perhaps 850 or so, however – much shallower than the dip in the heterozygosity  $N_e$ . Our inbreeding  $N_e$  estimator here underestimates the extent of inbreeding, since it only looks at the distribution of fecundity among the parental generation; once the sweep is underway, the parental generation is itself inbred, which could be taken into account with a more accurate formula. For example, one could perhaps use SLiM's `relatedness()` function to calculate the mean pairwise consanguinity in the population; it uses the grandparental pedigree IDs, so it would observe a higher degree of inbreeding, producing a lower  $N_e$  estimate. As we said above, there are a great many ways to calculate  $N_e$ , and little agreement. There is nothing necessarily *wrong* with our inbreeding  $N_e$  estimator here; if you were trying to match this model up with empirical data that consisted only of parental identifications, with no data on the grandparents, this estimator might be the one you would use, since it would match your available empirical data and the empirical estimate of  $N_e$  that you derived from that data.

One other concept to note is that although the estimators shown here estimate  $N_e$  from a metric taken in a single tick, to get a less noisy estimation of  $N_e$  it can be useful to smooth out the estimate by, e.g., taking the harmonic mean of  $N_e$  estimates across multiple ticks. Of course such smoothing may hide transient changes in  $N_e$  that are actually important, such as the transient dip in the inbreeding  $N_e$  shown above; this is always a trade-off with smoothing of time series data, and so this technique must be used with care.

That just about wraps up this section. It got more deeply into details of population genetics than most of the recipes in this manual, but the difference between  $N$  and  $N_e$  is a common source of confusion, so that seemed worthwhile. The key points to remember from all of this are: (1) SLiM thinks in terms of the census population size,  $N$ , since it is an individual-based simulation framework; (2) analytical models typically think in terms of the effective population size,  $N_e$ ; (3) empirical work also often involves estimation of  $N_e$ , since determining  $N$  for real populations can be very difficult; (4) to bring these different worlds together, estimating  $N_e$  from your SLiM simulation data is often useful for a variety of purposes; and (5) there are many ways to estimate  $N_e$ , and they mean different things and produce different numbers, so think carefully about your choice of estimator and the assumptions upon which it is based. Do not just use the  $N_e$  estimators given in this recipe; they may be inappropriate for your research problem, or it may simply be that their implementation here (based, as it is, upon a WF model) does not dovetail well with the mechanics of your SLiM model.

## 10. Context-dependent selection using `mutationEffect()` callbacks

In this and following chapters, we will show how *Eidos callbacks* can be used to modify SLiM's standard behavior. You have seen one Eidos callback already, the `initialize()` callbacks that are called by SLiM at initialization time to modify the default initialization behavior (which sets up only the `Species` object). At least one `initialize()` callback is required, since SLiM's default initialization is insufficient to produce a working simulation. Other Eidos callbacks are optional, because SLiM's default behavior is sufficient. In this chapter we will look at one particular Eidos callback, the `mutationEffect()` callback, used to modify SLiM's default evaluation of the fitness of individuals as a function of the mutations present in the individuals' genomes.

There are just a few conceptual points to discuss before the first recipe. First of all, `mutationEffect()` callbacks return a relative fitness value for a focal mutation in a focal individual. Neutrality is indicated by a relative fitness of `1.0`; fitness uses a different scale than selection coefficients, for which `0.0` indicates neutrality. A `mutationEffect()` callback is called once for each mutation possessed by each individual; the callback can therefore assign a different fitness value to the same mutation depending upon the focal individual possessing the mutation. If a given individual is homozygous for a mutation, the `mutationEffect()` callback is still called only once; a flag provided to the callback indicates whether the focal mutation is homozygous or heterozygous in the focal individual. This is because `mutationEffect()` callbacks return a relative fitness rather than a selection coefficient: they take all of the information regarding the focal mutation in the focal individual – selection coefficient, dominance coefficient, homozygosity versus heterozygosity, genetic background, subpopulation, sex, etc. – and condense all of it into a determination of the fitness effect of the focal mutation. Each mutation in the focal individual is evaluated separately – by one or more `mutationEffect()` callbacks if any apply, otherwise by SLiM's standard fitness equation – to produce a set of relative fitness values, one for each mutation. SLiM then multiplies these relative fitness values to determine the final fitness of the individual. This process is repeated for each individual in the simulation. This is just a quick summary; see sections 22.6, 23.3, and 25.2 for a fuller explanation.

### 10.1 Temporally varying selection

One way to model temporally varying selection in SLiM is to use the `setSelectionCoeff()` method of `Mutation`; you can find the mutation(s) whose selection coefficient you want to change, and then use that method to make the change at a specific point in time. However, there are several disadvantages to this approach in general. First of all, the change is permanent; it would require additional bookkeeping to later restore the original selection coefficient (if the modified selection regime ends, for example). Second, each mutation that you wish to modify must be changed individually. (To be fair, there are also advantages to this method; the change is done once and then is finished with, and subsequently you pay no speed penalty for the change.)

Here we will examine another solution to this problem, a `mutationEffect()` callback:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0); // neutral
    initializeMutationType("m2", 0.5, "f", 0.1); // beneficial
    initializeGenomicElementType("g1", c(m1,m2), c(0.995,0.005));
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
2000:3999 mutationEffect(m2) { return 1.0; }
10000 early() { sim.simulationFinished(); }
```

The `initialize()` callback sets up two mutation types: a very common neutral mutation type (`m1`), and an uncommon beneficial mutation type (`m2`). This produces distinctive simulation dynamics in which occasional beneficial mutations arise and sweep quickly.

However, from tick 2000 to 3999, the simulation switches to pure neutral dynamics, because mutation type `m2` reverts to neutrality for that period of time due to the `mutationEffect()` callback that is defined. This change in dynamics can be seen quite clearly in SLiMgui when the model is run. Let's look at the callback in more detail:

```
2000:3999 mutationEffect(m2) { return 1.0; }
```

The tick range used, 2000:3999, is expressed in the usual syntax. Following that comes the declaration that this block is a `mutationEffect()` callback, rather than an ordinary Eidos event: `mutationEffect(m2)`. A `mutationEffect()` callback must be declared as applying to one specific mutation type – in this case, `m2`; the callback modifies the fitness effect only of mutations belonging to that mutation type. (This is both for conceptual clarity and for efficiency.) The rest of the callback definition is a compound statement that returns a `float` value, used by SLiM as the fitness effect of the mutation. Here the value 1.0 is returned, which represents neutrality. (Remember that a neutral mutation has a selection coefficient of 0.0 but a multiplicative fitness effect of 1.0). This is the first time we have seen the Eidos keyword `return`; it simply causes the executing script block to return immediately, passing the returned value out to the caller of the block, which in this case is the SLiM engine itself. It can be used in Eidos events too, although a value may not be returned; when no return value is expected, one can simply “`return;`”.

This recipe is trivially simple, but of course the code in the `mutationEffect()` callback could do anything, so the potential power of this mechanism should be apparent. In the context of temporally varying selection, you could make the fitness effect vary sinusoidally through time using an expression based on `sin(sim.cycle)`, or make it be random in each tick with `rnorm()`, or any other effect you wish.

## 10.2 Spatially varying selection

The previous example showed a simple recipe implementing temporally varying selection. In this section we will see how to make selection vary spatially between subpopulations. More specifically, we will examine a model in which mutations have beneficial effects in one subpopulation, but deleterious effects in another subpopulation. Whether such mutations fix or not depends on the strength of the fitness effects, the migration rates between the subpopulations, and various other factors. The recipe:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "e", 0.1); // deleterious in p2
  initializeGenomicElementType("g1", c(m1,m2), c(0.99,0.01));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
  sim.addSubpop("p2", 500);
  p1.setMigrationRates(p2, 0.1); // weak migration p2 -> p1
  p2.setMigrationRates(p1, 0.5); // strong migration p1 -> p2
}
mutationEffect(m2, p2) { return 1/effect; }
10000 early() { sim.simulationFinished(); }
```



Here we set up two subpopulations of equal size, with weak migration from p2 to p1, but with strong migration from p1 to p2 (remember that you can use the population visualization graph in SLiMgui to see a graphical depiction of the population structure, after you Recycle and then Step through the initialization). Mutations are mostly neutral, but occasionally beneficial mutations are drawn from an exponential distribution with mean 0.1. This is all review; the interesting part is the `mutationEffect()` callback:

```
mutationEffect(m2, p2) { return 1/effect; }
```

Here no tick range is specified; this `mutationEffect()` callback is therefore active in every tick. Subpopulation p2 is given as a parameter to the callback definition; this restricts the operation of the callback to only the specified subpopulation, producing spatial variation in selection. The body of the callback returns `1/effect` as the modified fitness effect of each m2 mutation.

We haven't seen `effect` before; it is a variable defined by SLiM when a `mutationEffect()` callback is called, and it contains the mutational effect – the fitness value – that SLiM would normally use for the mutation. By returning `1/effect`, we are telling SLiM to invert the normal fitness effect of all m2 mutations in p2; if they are of high fitness in p1 (2.0, say) they become low fitness in p2 ( $1/2.0 == 0.5$ ), and vice versa. Of course there is nothing magical about `1/effect` in particular; it is just a convenient, simple expression. One could write the `mutationEffect()` callback script to calculate and return whatever fitness effect one wished.

There is a point worth clarifying here. Mutation type m2 draws selection coefficients from an exponential DFE; each mutation of type m2 thus has a different selection coefficient. Because of this, the `mutationEffect()` callback here is not called just once per tick, to calculate a modified fitness effect for all mutations of type m2; it is called *once per individual per m2 mutation*, and it calculates a modified fitness effect for *that* mutation in *that* individual. These calculations can therefore depend upon both the focal mutation and the focal individual. In addition to the `effect` variable that contains the default relative fitness effect for the mutation, SLiM also defines `mut`, the mutation being assessed; `subpop`, the subpopulation containing the individual possessing the mutation; `homozygous`, a flag which is T (true) if the individual is homozygous for the mutation, F (false) otherwise; and a few others as well that we will see in the next sections. The code in a `mutationEffect()` callback can use all of these variables, and any other model state, to calculate its modified fitness effect.

Because `mutationEffect()` callbacks might be called thousands or even millions of times every tick, SLiM and Eidos are highly optimized to make those calls as fast as possible. You should do your part by making your Eidos code as tight as possible; writing an inefficient `mutationEffect()` callback is an excellent way to make your simulation slow to a crawl. See section 21.1 for some tips on how to write fast Eidos code.


If you open this recipe in SLiMgui, Recycle, and Play, you will see the spatial dynamics very clearly; mutations will spread that are beneficial in p1, turning that subpopulation's individuals green, but deleterious in p2, turning that subpopulation's individuals red. With the recipe as written, these mutations will often fix; the strong migration from p1 versus the weak migration from p2 gives p1 an advantage, allowing it to force p2 to fix mutations that are deleterious in that context. If you reverse the migration bias, that will no longer happen; instead p2 will be able to force p1 to lose mutations that are beneficial in that context. One could easily use this model to explore this scenario in detail, with questions such as the effect of subpopulation size, migration rate, selection strength, and so forth. Since neutral mutations are also in the model, one could also ask questions about how these sorts of dynamics affect neutral diversity and divergence. Not bad for sixteen lines of code!

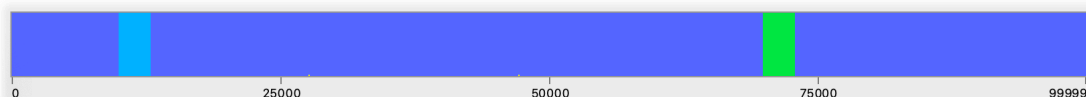
## 10.3 Fitness as a function of genomic background

### 10.3.1 Epistasis

The `mutationEffect()` callback mechanism can also make the fitness effect of a mutation depend upon the genetic background of the individual that possesses the mutation. One example of this is the phenomenon of *epistasis*, in which two loci interact to produce a non-additive fitness effect (Philipps 2008). The recipe here is a bit contrived – a more realistic model would probably use introduced epistatic mutations rather than random mutations – but it serves to illustrate the concept:

```
initialize() {
  initializeMutationRate(1e-8);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.1); // epistatic mut 1
  initializeMutationType("m3", 0.5, "f", 0.1); // epistatic mut 2
  initializeGenomicElementType("g1", m1, 1);
  initializeGenomicElementType("g2", m2, 1); // epistatic locus 1
  initializeGenomicElementType("g3", m3, 1); // epistatic locus 2
  initializeGenomicElement(g1, 0, 10000);
  initializeGenomicElement(g2, 10001, 13000);
  initializeGenomicElement(g1, 13001, 70000);
  initializeGenomicElement(g3, 70001, 73000);
  initializeGenomicElement(g1, 73001, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
mutationEffect(m3) {
  if (individual.genome1.countOfMutationsOfType(m2))
    return 0.5;
  else if (individual.genome2.countOfMutationsOfType(m2))
    return 0.5;
  else
    return effect;
}
```


The `initialize()` callback sets the stage by making three mutation types, three genomic element types, and five genomic elements. Over the majority of the chromosome, genomic element type `g1` is used; it generates only neutral mutations. In a locus spanning `10001:13000`, genomic element type `g2` is used, which generates mutations of type `m2`, which are beneficial with a selection coefficient of `0.1`. In a second locus spanning `70001:73000`, genomic element type `g3` is used, which generates mutations of type `m3`, which are similarly beneficial. If you turn on display of genomic elements in SLiMgui, using the  action button to the right of the chromosome view and choosing the Display Genomic Elements item, the resulting chromosomal setup looks like this:

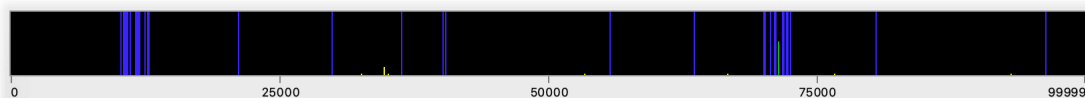


The twist, however, comes in the `mutationEffect(m3)` callback, which makes mutations of types `m2` and `m3` interact epistatically. Any mutation of type `m3` has a fitness effect of `0.5` – strongly deleterious – if it is in the same genome as any mutation of type `m2`. This is achieved in several steps. First of all, `individual` is defined by SLiM for `mutationEffect()` callbacks; it is the focal individual for which the fitness of the focal mutation is being evaluated, and `individual.genome1`

and `individual.genome2` are the two genomes (the two homologous chromosomes) of that focal individual. The call to `individual.genome1.countOfMutationsOfType()` counts the mutations of type `m2` in the first genome; if that count is non-zero, the `if` statement evaluates it as true (T), and 0.5 is returned, making the focal mutation deleterious. The next two lines perform the same check for `individual.genome2` (these two `if` conditions could be combined into a single test using the “logical or” operator, `|`, but the line of code would then be too long to fit without wrapping here). If neither Genome object contains a mutation of type `m2`, the final `else` clause will execute and `effect` will be returned, keeping the normal fitness effect of the focal mutation without modification (see section 25.2 for discussion of `effect`). There are some new concepts here, and this manual is not going to explain them all in detail; for full discussions of the `logical` type, the behavior of true and false values in Eidos, and so forth, please refer to the Eidos manual, which explains these ideas thoroughly.

Note that mutations of type `m2` suffer no direct fitness penalty from sharing an individual with an `m3` mutation; the `mutationEffect()` callback affects only `m3` mutations. Since the fitness of the carrying individual will be severely compromised, however, the net effect is that `m2` and `m3` mutations are rarely found together; collocation harms `m2` mutations too, although indirectly.

This produces some interesting dynamics. Type `m2` and `m3` mutations arise fairly often, and quickly sweep to fixation; however, they never sweep together, so if a mutation at one of the two epistatic loci is sweeping, the other locus will have no active mutations. The two loci thus “take turns” sweeping new mutations to fixation. Since new beneficial loci sweep so often, neutral loci generally fix only if they are linked to a beneficial mutation. Because of recombination, that is most likely to happen close to one of the two epistatic loci, so if we run the full simulation and then turn on display of fixed mutations (“substitutions”) by clicking the  button to the right of the chromosome view and choosing Display Substitutions, we see a pattern of clustering near the epistatic loci:



All of the fixed mutations within the loci are beneficial epistatic mutations, since the two loci generate only those mutation types. All of the bands for fixed mutations outside of those loci, however, are for neutral mutations that hitchhiked.

The astute reader will have noticed a problem with all this. Biologically, this dynamic of “taking turns” sweeping at one or the other locus makes no sense. If a mutation of type `m2` sweeps to fixation at the `g2` locus, that mutation still exists in the genome, and the epistatic interaction between `m2` and `m3` mutations should prevent an `m3` mutation from sweeping, forever after. That objection is correct, and points out a very important issue.

The reason that this recipe behaves in this manner has to do with the way that SLiM handles fixed mutations (see sections 1.5.2 and 22.3). When a mutation fixes, SLiM normally removes it from the simulation, replacing it with a `Substitution` object that provides a permanent record of the fixed mutation. The assumption is that since every individual possesses the fixed mutation, it has the same fitness effect in every individual, and therefore can be ignored. However, epistasis violates that assumption, since the fixed mutation causes a differential fitness effect among individuals based upon the genetic background in which it is found. SLiM’s replacement of fixed `m2` and `m3` mutations in this model thus produces incorrect behavior.

What to do? Happily, SLiM provides a way to handle this situation. The `MutationType` class has a `logical` property named `convertToSubstitution`; by default it is T, indicating to SLiM that fixed mutations of that type should be replaced by `Substitution` objects. We can set it to F instead,

telling SLiM to keep all mutations of that type active in the simulation even once fixed. (See sections 22.3 and 24.11.1 for further discussion of this property). The new recipe:

```
initialize() {
  initializeMutationRate(1e-8);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.1); // epistatic mut 1
  m2.convertToSubstitution = F;
  initializeMutationType("m3", 0.5, "f", 0.1); // epistatic mut 2
  m3.convertToSubstitution = F;
  initializeGenomicElementType("g1", m1, 1);
  initializeGenomicElementType("g2", m2, 1); // epistatic locus 1
  initializeGenomicElementType("g3", m3, 1); // epistatic locus 2
  initializeGenomicElement(g1, 0, 10000);
  initializeGenomicElement(g2, 10001, 13000);
  initializeGenomicElement(g1, 13001, 70000);
  initializeGenomicElement(g3, 70001, 73000);
  initializeGenomicElement(g1, 73001, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
mutationEffect(m3) {
  if (individual.genome1.countOfMutationsOfType(m2))
    return 0.5;
  else if (individual.genome2.countOfMutationsOfType(m2))
    return 0.5;
  else
    return effect;
}
```

This will now produce the correct epistatic dynamics. If you run this model in SLiMgui, you will see that one or the other epistatic locus wins an initial contest by fixing a mutation. Once that happens, mutations at the other locus will never manage to establish.

The `convertToSubstitution` property should be used to prevent fixed mutation substitution whenever the assumption of equal effects in all individuals would be violated, whether by a `mutationEffect()` callback introducing a mechanism like epistasis, or by differential effects of the mutation type on mate choice or other dynamics. SLiM is not able to guess when substitution should be turned off, so you must keep this caveat in mind. However, also keep in mind that turning off substitution will make your models run more slowly, especially if there are many mutations of that type in the model.

## 10.4 Fitness as a function of population composition

### 10.4.1 Frequency-dependent selection

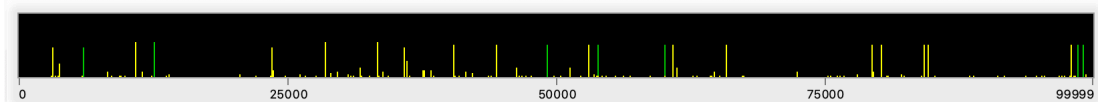
In previous sections we have seen how we can use a `mutationEffect()` callback to modify the fitness effects of mutations in order to model scenarios such as epistasis, polygenic selection, and spatiotemporal variation in selection. Now we will shift to making the fitness of mutations be a function of population composition. First we'll look at frequency-dependent selection, in which the fitness effect of a mutation depends upon the frequency of the mutation in the population (Ayala & Campbell 1974). Let's start with a simple recipe for negative frequency-dependence, in which the fitness of a mutation is inversely correlated with its frequency:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.1); // balanced
  initializeGenomicElementType("g1", c(m1,m2), c(999,1));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
mutationEffect(m2) {
  return 1.5 - sim.mutationFrequencies(p1, mut);
}

```

The idea here should be pretty clear; if the `mutationFrequencies()` method tells us that `mut` is rare in `p1`, then the mutation will be highly beneficial, but if it tells us that `mut` is common, the mutation will be deleterious. These dynamics prevent the mutation from either fixing or being lost; instead we have what is called “balancing selection”, in which selection favors keeping mutations of that type at an intermediate frequency. (See section 10.5 for an adaptation of this recipe using `setSelectionCoeff()` instead; and see section 11.3 for a very different model of balancing selection, using a `modifyChild()` callback instead of a `mutationEffect()` callback.) Running this for a while in SLiMgui gives us a picture something like this (where the yellow lines are neutral mutations and the green lines are the mutations under balancing selection):



There are seven `m2` mutations under balancing selection, all at frequency  $\sim 0.5$  since that is the point at which the fitness effect changes from beneficial to deleterious according to the callback. Next let's look at a model of positive frequency-dependence:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.1); // positive freq. dep.
  initializeGenomicElementType("g1", c(m1,m2), c(999,1));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
mutationEffect(m2) {
  return 1.0 + sim.mutationFrequencies(p1, mut);
}

```

The way this works is probably pretty obvious at this point; at low frequencies (as calculated by `mutationFrequencies()`) mutations of type `m2` are close to neutral, but at higher frequencies they become strongly beneficial. Note that these mutations still do not sweep to fixation as quickly as one might expect. This is because `mutationEffect()` callbacks get called only once per mutation *per individual*. If an individual carries one copy of a mutation (i.e., is a heterozygote), the `mutationEffect()` callback is called once. If an individual carries two copies (i.e., is a homozygote), the `mutationEffect()` callback is *still* called only once. This means that the `mutationEffect()` function effectively defines the fitness effect as involving complete dominance. To differentiate between these cases, moving away from complete dominance, a



mutationEffect() callback may consult a flag named homozygous that is defined by SLiM to indicate whether the focal mutation is homozygous or not (see section 25.2):

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.1); // positive freq. dep.
  initializeGenomicElementType("g1", c(m1,m2), c(999,1));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
mutationEffect(m2) {
  dominance = asInteger(homozygous) * 0.5 + 0.5;
  return 1.0 + sim.mutationFrequencies(p1, mut) * dominance;
}
```

#### 10.4.2 Cultural effects on fitness

Sometimes fitness might be influenced by environmental factors as well as genetic factors. In some cases, these environmental factors would be associated with the subpopulation in which an individual resides; in that case, they can be modeled as spatial variation in selection (see section 10.2). In other cases, however, the environmental factors might be a matter of individual variation that is not correlated with the subpopulation; that possibility is what we will explore in this recipe.

In particular, we will make a simple model of cultural differences between individuals. Each individual will be assigned to one of two cultural groups at birth. If an individual belongs to one cultural group, a particular type of mutation will be beneficial; if the individual belongs to the other cultural group, those mutations will be neutral. An analogue to this in human history, which we will loosely follow here, would be the allele conferring the ability to digest lactose as an adult; if an individual belongs to a cultural group that drinks milk in adulthood, mutations promoting the retention of the lactase enzyme into adulthood are beneficial, whereas if the individual belongs to a cultural group that does not drink milk in adulthood, such mutations are nearly neutral (or perhaps slightly deleterious, due to the energetic investment of producing an unneeded enzyme).

For the initial version of this model, the cultural group into which an individual is assigned will be entirely random. The cultural group of an individual will be tracked using the tag property of Individual, an integer property that is not used at all by SLiM, and is thus free for you to use as you wish. We will use a tag value of 1 to indicate a milk-drinker, and a value of 0 to indicate a non-milk-drinker. The tag value will be set up in a late() event that runs after offspring are generated but before fitness values are calculated.

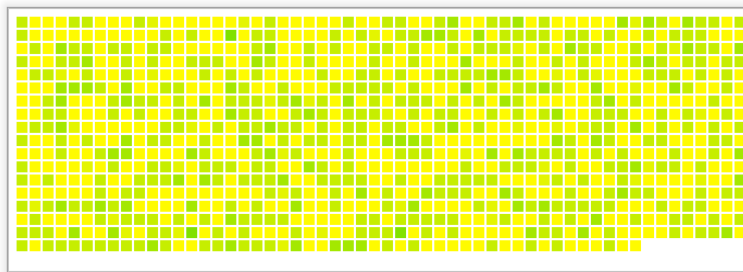
The recipe:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.1); // lactase-promoting
  m2.convertToSubstitution = F;
  initializeGenomicElementType("g1", c(m1,m2), c(0.99,0.01));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 1000); }
10000 early() { sim.simulationFinished(); }
```



```
late() {
  // Assign a cultural group: milk-drinker == 1, non-milk-drinker == 0
  p1.individuals.tag = rbinom(1000, 1, 0.5);
}
mutationEffect(m2) {
  if (individual.tag == 0)
    return 1.0; // neutral for non-milk-drinkers
  else
    return effect; // beneficial for milk-drinkers
}
```

When run, the occasional mutations of type `m2` sweep to fixation, as you might expect, but even when they are close to fixation only about half of the population – the milk-drinkers – are experiencing a fitness benefit from the mutation. This can be seen easily in the pattern of individual fitness values displayed in SLiMgui when an `m2` mutation was at a frequency of 0.95:



The `m2` mutations are prevented from being removed at fixation, by setting the `convertToSubstitution` property of `m2` to `F`. As the model runs, `m2` mutations accumulate, and the fitness benefits of being a milk-drinker become larger and larger. Since that status is assigned randomly, however, this does not affect the frequency of milk-drinking; a randomly chosen half of the population is less likely to pass on its genes to the next generation, but milk-drinking remains a random choice.

The `late()` event assigns cultural `tag` values to all of the individuals in the new offspring generation. We first generate 1000 random values, either 0 or 1, using `rbinom()` to draw from a simple binomial distribution. The resulting vector is assigned directly into `p1.individuals.tag` using the multiplexed assignment semantics of Eidos to put each value in the vector into the `tag` property of the corresponding individual in subpopulation `p1` (see the Eidos manual for further discussion of multiplexed property assignment).

Given those `tag` values, the `mutationEffect()` callback is then trivial: for the lactase-promoting mutations of type `m2`, the fitness is neutral if the `tag` value of a given individual carrying the mutation is 0, indicating a non-milk-drinker, whereas for milk-drinkers, with a `tag` value of 1, `effect` is returned to accept SLiM's default calculated fitness for the mutation (which uses both the selection coefficient of 0.1 and the dominance coefficient of 0.5 defined for `m2`). This could be modified to make the mutations slightly deleterious in non-milk-drinkers, if one wished, of course.

In this model, the cultural group of an individual is assigned randomly. In many cases one would wish this trait to have some degree of heritability, even though it is not genetically based; in humans and some other species, individuals can inherit cultural traits from their parents through social learning. To implement that, a `modifyChild()` callback is needed; we will therefore take this model up again in section 12.1. For the time being, then, this model simply shows how to make the fitness effect of a particular type of mutations depend upon the non-genetic state of the individual.

### 10.4.3 Kin selection and the green-beard effect

According to the theory of kin selection and inclusive fitness (Hamilton 1964ab; Dawkins 1976), a gene that promotes altruism towards kin can spread even if its *direct* fitness effect on its carrier is negative (due to altruistic self-sacrifice), as long as the altruistic behavior of the carrier produces sufficiently large *indirect* benefits for other individuals that carry the same gene. In other words, one cannot look at things solely from the perspective of the individual; one must look from the perspective of the gene, to see whether the average indirect benefits that accrue to carriers of that gene outweigh the direct costs of possessing the gene. In the extreme case, it can make evolutionary sense for the carrier of an altruism gene to sacrifice its own life, even before reaching reproductive age, in order to benefit its kin that carry the same gene. Famously, J.B.S. Haldane declared (perhaps apocryphally) that he was “prepared to lay down his life for eight cousins or two brothers”, because of the degree of kinship they would share with him.

We would like to test whether inclusive fitness theory is really correct that kin-directed altruism can evolve even when the individuals committing the altruistic acts (at a cost to themselves) do not necessarily receive any benefit from altruistic acts by others. Using the `tag` property of the `Individual` class, as shown in the previous section, we can develop a model to answer that question. Here we will explore this question in a closely related evolutionary problem, the modeling of *green-beard alleles* (Hamilton 1964ab; Dawkins 1976). A green-beard allele causes three pleiotropic effects: (1) a phenotypic trait of some kind (the “green beard”), (2) the ability to recognize other individuals possessing this phenotypic trait, and (3) a tendency to direct altruistic acts toward other individuals possessing this phenotypic trait (at some cost to the altruistic individual, and some benefit to the receiving individual). The green-beard allele therefore rolls up kin recognition and altruism towards kin into a single package, making it a straightforward concept to test in a model.

The idea for this recipe is, in effect, to add a new stage to SLiM's default generational life cycle by adding an event to the model. In this new life cycle stage, a finite number of one-on-one interactions between randomly chosen individuals in the population are modeled. If the individuals both have the green-beard allele, an altruistic act occurs and one receives a benefit and the other receives a cost; if either individual does not have the green-beard allele, no altruistic act occurs and the interaction is neutral. The benefits and costs incurred by each individual are tallied up separately, and are taken into account in a `mutationEffect()` callback that models the fitness effect of the green-beard allele for each individual based upon its particular interaction tally. Some individuals with the green-beard allele will, by chance, incur nothing but costs from their interactions, harming themselves to the point where they are unlikely to reproduce. Other individuals with the green-beard allele will, by chance, incur a mixture of costs and benefits, or if they're lucky, nothing but benefits; their outcome will thus be neutral or positive.

Let's get to the recipe. We'll build it one step at a time; the first step is just to set up a single subpopulation with two mutation types:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.0); // green-beard
  m2.convertToSubstitution = F;
  m2.color = "red";
  initializeGenomicElementType("g1", m1, 1);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 early() { sim.simulationFinished(); }
```

Mutation type `m2` will be used for green-beard alleles, but is not used yet. It is set to not convert into a `Substitution` object upon fixation, using the `convertToSubstitution` property of `MutationType`, so that we can easily see whether or not it has fixed at the end of a model run. We also set it to display in red, but in any case `m2` is not used yet; as it stands, then, this is just a model of neutral drift using only `m1`.

Now let's add a system for tagging individuals with costs and benefits:

```
1: late() {  
    p1.individuals.tag = 0;  
}  
mutationEffect(m2) { return 1.0 + individual.tag / 10; }
```

Every individual in SLiM has a `tag` property that can be set to any value. As explained in the previous section, `tag` is not used by SLiM; it's just scratch space for models to use as they please. In this recipe, we use it to hold the cost/benefit tallies for individuals. The `mutationEffect()` callback then adds the `tag` value for the individual it is evaluating to a neutral relative fitness of `1.0`. The `tag` value is an integer, so we divide it by `10` in the callback; each increment or decrement of an individual's `tag` will represent a change to its relative fitness of `0.1` or `-0.1`.

Let's introduce a green-beard allele into the population. It won't be very interesting if it just gets lost due to drift in the first generation or two, as green-beard alleles at low frequency generally drift freely since interactions between the rare green-beards are unlikely – so we'll introduce several copies of it to give it an initial boost. The concept of introducing a mutation is covered in depth in section 9.1, but it should be pretty clear what's going on here:

```
1 late() {  
    target = sample(p1.genomes, 100);  
    target.addNewDrawnMutation(m2, 10000);  
}
```

This code draws `100` target genomes from the population using the `sample()` function (which samples without replacement, by default). It then adds a new mutation of type `m2` to those target genomes with a single call to `addNewDrawnMutation()`. This call adds the same new mutation to all of the target genomes, rather than a different new mutation to each genome, because the `addNewDrawnMutation()` method is a class method of `Genome`, not an instance method, and it is therefore called just once, rather than being multicast out to each instance. This is conceptually similar to writing a static member function in C++ (which is like a class method in Eidos) to perform an operation across a vector of objects. That static member function would take a vector of objects as one of its parameters; in Eidos, you instead call the class method on the target vector, as seen here, just like calling an instance method. If this is not clear, don't worry; it is not an essential point, since the syntax for calling class methods and instance methods is identical in Eidos. You can consult the Eidos manual for more details on class versus instance methods.

Some individuals might receive two copies of the green-beard mutation (one in each of their genomes), and end up being homozygous, but there is no harm in that, and most of the individuals targeted by this code will end up heterozygous for the new green-beard allele. (You could guarantee heterozygosity by sampling individuals instead of genomes, and then getting just one genome from each sampled individual; conversely, you could guarantee homozygosity by sampling individuals and then adding to both of the genomes of each sampled individual.) Again, it should be emphasized that adding `100` copies is in no way “cheating”; this is just a model of whether a green-beard allele can rise from low frequency to fixation, rather than a model of whether a single copy of a green-beard allele can rise all the way to fixation. It would be easy to construct the latter model in SLiM using the tools introduced in section 9.2.

Now we need to make our green-beards interact! We'll do this with an extension to our previous 1: late() event:

```
1: late() {
  p1.individuals.tag = 0;

  for (rep in 1:50) {
    individuals = sample(p1.individuals, 2);
    i0 = individuals[0];
    i1 = individuals[1];
    i0greenbeards = i0.countOfMutationsOfType(m2);
    i1greenbeards = i1.countOfMutationsOfType(m2);

    if (i0greenbeards & i1greenbeards) {
      alleleSum = i0greenbeards + i1greenbeards;
      i0.tag = i0.tag - alleleSum;      // cost to i0
      i1.tag = i1.tag + alleleSum * 2;  // benefit to i1
    }
  }
}
```

This bears some explaining. First, we zero out all of the individual tag values, as before. Then we run a loop 50 times, to produce 50 interactions between pairs of individuals (which might or might not possess green beards). Each time through the loop, we draw two individuals from the population using `sample()`, giving us a vector `individuals` containing two objects of class `Individual` which we extract into `i0` and `i1`.

The next two lines count the number of `m2` mutations contained in the two genomes belonging to each individual. The value of `i0greenbeards` and `i1greenbeards`, then, is 0 if the corresponding individual does not have the green-beard mutation at all, 1 if it is heterozygous, or 2 if it is homozygous. This is equivalent to, but more concise than, writing:

```
i0greenbeards = i0.genomes[0].countOfMutationsOfType(m2)
               + i0.genomes[1].countOfMutationsOfType(m2);
```

(and the same for `i1greenbeards`).

Next comes an `if` statement that will execute if both `i0greenbeards` and `i1greenbeards` are non-zero (since in Eidos, as in many languages, 0 is considered false and any non-0 value is considered true). So this executes if and only if both of the interacting individuals are at least heterozygous for the green-beard allele. In that case, `alleleSum` is computed as the total number of green-beard alleles between the two individuals; this makes the green-beard effect stronger when homozygotes are involved, weaker when heterozygotes are involved. Finally, the tag values of the two interacting individuals are modified to reflect the interaction; individual `i0` acts altruistically and incurs a cost, whereas individual `i1` receives the altruistic act and gets a benefit. The costs and benefits are added to the tag value of each individual. Note that the benefit is larger than the cost; this makes the inclusive fitness effect of the green-beard allele positive, and thus means that the green-beard allele is selected for, on average, in each generation. But as promised, the benefit and the cost are incurred by different individuals in this model. Indeed, the altruistic individual feels a nearly lethal fitness effect, if both interacting individuals are homozygous. From the perspective of the individual, it is a truly selfless altruistic sacrifice. From the perspective of the green-beard allele, it is an entirely selfish act, however – which is why the “selfish gene” perspective has so much explanatory power.

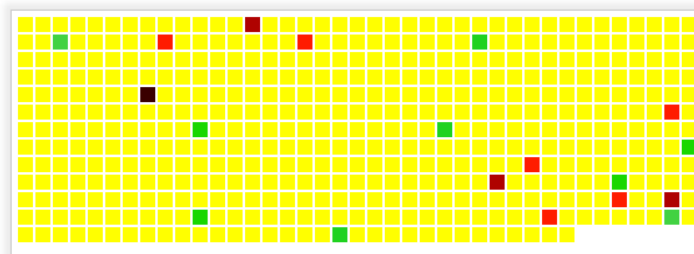
For the record, here is the full recipe for our green-beard model:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.0); // green-beard
  m2.convertToSubstitution = F;
  m2.color = "red";
  initializeGenomicElementType("g1", m1, 1);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
1 late() {
  target = sample(p1.genomes, 100);
  target.addNewDrawnMutation(m2, 10000);
}
1: late() {
  p1.individuals.tag = 0;

  for (rep in 1:50) {
    individuals = sample(p1.individuals, 2);
    i0 = individuals[0];
    i1 = individuals[1];
    i0greenbeards = i0.countOfMutationsOfType(m2);
    i1greenbeards = i1.countOfMutationsOfType(m2);

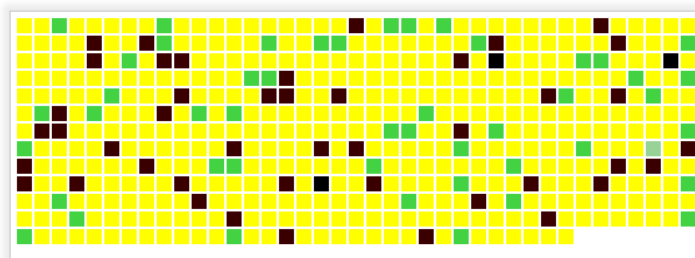
    if (i0greenbeards & i1greenbeards) {
      alleleSum = i0greenbeards + i1greenbeards;
      i0.tag = i0.tag - alleleSum; // cost to i0
      i1.tag = i1.tag + alleleSum * 2; // benefit to i1
    }
  }
}
mutationEffect(m2) { return 1.0 + individual.tag / 10; }
10000 early() { sim.simulationFinished(); }
```

If you Recycle and run this recipe in SLiMgui, it may take several tries before the green-beard mutation “catches” and rises to fixation. Again, this is because its fitness effect is close to neutral when it is at low frequency; early on, it is quite liable to be lost simply due to drift. Once the mutation “catches”, though, the population view will show something like this:



This display, of the model when the green-beard allele is at a frequency of around 0.3, shows the result of the individual interactions between green-beards. Some benefit from those interactions, and are colored in a shade of green or blue (depending on the magnitude of the benefit); some are harmed, and are colored in a shade of orange or red, shading down to black (depending on the magnitude of the harm).

Even once the green-beard mutation has fixed, the majority of individuals will be yellow (indicating neutrality – a relative fitness of exactly 1.0), because only 50 pairs of individuals are chosen to interact in each tick:



The larger the number of interactions, relative to the population size, the stronger the selection will be in favor of the green-beard allele. If you increase the number of interactions from 50 to 5000, for example, there will be so many interactions relative to the population size that the green-beard allele will rise to fixation almost deterministically. In this case, each individual is likely to participate in several interactions, and thus to both benefit from and be harmed by its interactions – and the benefit will, on average, outweigh the harm, even at the level of a single individual. With only 50 interactions, however, some individuals will benefit while others will self-sacrifice and die, as seen above – the individuals colored black have sacrificed themselves for no personal benefit, whereas the green individuals have received a large benefit with no self-sacrifice. With this few interactions the model is quite stochastic in its behavior, and the green-beard allele will sometimes be lost even when it has risen as high as a frequency of 0.5.

This recipe is, from a rather sidelong angle, basically a model of a selective sweep, and as such, it showed how to introduce a new mutation into a population and watch it sweep to fixation. In the next chapter, that subject will be treated comprehensively.

## 10.5 Changing selection coefficients with `setSelectionCoeff()`

The preceding recipes have demonstrated the use of `mutationEffect()` callbacks to implement a wide variety of different types of selection in SLiM. It is worth noting, however, that it is also possible to simply change the selection coefficient of a mutation using the `setSelectionCoeff()` method of `Mutation` (see section 24.10.2). Indeed, this can have large performance advantages, since executing Eidos callbacks is relatively slow. However, this strategy can only be applied in very limited cases.

First of all, if you want the fitness effect of a mutation to vary from individual to individual – whether because of the genetic background of the individual, or the subpopulation the individual is in, or any other individual-specific model state – then a `mutationEffect()` callback is necessary. This is because changing the selection coefficient of a mutation using `setSelectionCoeff()` changes that mutation's selection coefficient for all individuals, in all subpopulations. The recipes in sections 10.2 (spatially varying selection) and 10.3 (genomic background effects) could therefore not be implemented using `setSelectionCoeff()`, nor could those of sections 10.4.2 (cultural effects on fitness) or 10.4.3 (green-beard alleles).

Second, if you want a temporary fitness effect it is often best to use `mutationEffect()`, because when the callback is no longer active mutations will automatically revert to their original effect. To take the recipe in section 10.1 (temporally varying selection) as an example, when the callback expires the m2 mutations revert back to their original selection coefficient of 0.1 without needing to be re-set to 0.1; indeed, their selection coefficients are 0.1 the entire time, the `mutationEffect()` callback just overrides that with its own effect. If this recipe used `setSelectionCoeff()` instead,



the original selection coefficients would need to be restored when the altered fitness regime expired. In this case that would be straightforward – just set them all to 0.1 with another call to `setSelectionCoeff()` – but if the `m2` mutations were drawn from a distribution of fitness effects, and thus all had different selection coefficients, this would be more complicated.

The recipes in section 10.4.1, on the other hand, can in fact be improved by rewriting them to use `setSelectionCoeff()`. Here is a modified version of the first recipe from that section, which used a `mutationEffect()` callback to model a dominant frequency-dependent mutation type:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0); // neutral
    initializeMutationType("m2", 1.0, "f", 0.1); // balanced
    initializeGenomicElementType("g1", c(m1,m2), c(999,1));
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
late() {
    m2mut = sim.mutationsOfType(m2);
    freqs = sim.mutationFrequencies(NULL, m2mut);
    for (index in seqAlong(m2mut))
        m2mut[index].setSelectionCoeff(0.5 - freqs[index]);
}
10000 early() { sim.simulationFinished(); }
```

Note that this recipe uses a dominance coefficient of 1.0, to match the behavior caused by the `mutationEffect()` callback in the section 10.4.1's first recipe; any dominance coefficient could be used here, however. The `late()` event here calculates the frequency of each `m2` mutation, and then uses a loop to set the selection coefficient of each mutation accordingly. The recipe in section 10.4.1 used a fitness formula of 1.5–frequency in its callback, but here we use 0.5–frequency; this is because `mutationEffect()` callbacks return relative fitness values, where 1.0 is neutral, whereas with selection coefficients – the currency in which the present recipe trades – 0.0 is neutral instead. As always, this distinction should be treated with care; it is a common cause of errors.

This recipe runs about two orders of magnitude faster than the original recipe – not a small difference. It might not produce exactly identical results, because of the aggregated effects of tiny differences in floating-point roundoff error due to the different way in which fitness values are juggled in the two models, but it is effectively identical for all practical purposes.

However, this recipe's strategy using `setSelectionCoeff()` only works because this is a single-subpopulation model; with multiple subpopulations, one would presumably want the frequency-dependent effect to be different in each subpopulation, and to depend upon the frequency within that subpopulation. Such a model could no longer be achieved using `setSelectionCoeff()`. Section 10.4.1 thus presented a more generally applicable recipe, albeit a slower one.

## 10.6 Varying the dominance coefficient among mutations

The dominance coefficient is a property of the mutation type, not the mutation (see sections 1.5.3 and 7.1), meaning that all mutations of a given mutation type have the same dominance coefficient. Usually this is fine, but occasionally it proves inconvenient – occasionally one would like dominance coefficients to depend upon some other state, or to be drawn from a distribution the way selection coefficients are, or other such variable behaviors. This is possible to model in SLiM using a `mutationEffect()` callback. In this section we will look at a very simple recipe in which the dominance coefficients for one mutation type are drawn from a uniform distribution.

The recipe, in full:

```
initialize() {
  initializeMutationRate(1e-8);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", 0.05);
  initializeGenomicElementType("g1", c(m1,m2), c(1.0,0.01));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
}
mutation(m2) {
  mut.setValue("dom", runif(1));
  return T;
}
mutationEffect(m2) {
  if (homozygous)
    return 1.0 + mut.selectionCoeff;
  else
    return 1.0 + mut.getValue("dom") * mut.selectionCoeff;
}
100000 late() { sim.outputFixedMutations(); }
```

The `initialize()` callback sets up a uniform chromosome composed of a single genomic element type, `g1`, that draws from two mutation types: `m1`, representing neutral mutations, and `m2`, representing beneficial mutations. In tick 1 we create a new subpopulation with 500 individuals, and in tick 100000 the model terminates with a little output. The interesting bits are in between.

First of all, we have a new type of callback, a `mutation()` callback. This callback type, which was added in SLiM 3.3, is used fairly infrequently in SLiM models, so it will not get its own dedicated chapter of recipes; instead we will just discuss it here and elsewhere as it come up. SLiM calls `mutation()` callbacks whenever a new mutation is created by SLiM, providing the script with an opportunity to modify the new mutation or even to veto its creation altogether (this is strongly parallel to the `modifyChild()` callbacks we will see in chapter 12, but those allow modification or vetoing of proposed offspring, rather than proposed mutations). Section 25.9 provides detailed reference documentation for `mutation()` callbacks, but they are really quite simple. The declaration of the callback here specifies that it applies only to `m2` mutations (just as `mutationEffect()` callbacks specify their focal mutation type). The focal mutation is passed to the callback through the pseudo-parameter `mut`, as in `mutationEffect()` callbacks; that is the mutation that has just been created, to which we will assign a random dominance coefficient. The callback draws a random dominance coefficient in  $[0,1]$  using `runif()`, and assigns that into the named key "dom" on the focal mutation using `setValue()`; this method provides a dictionary-like mechanism for remembering arbitrary values by name (see the Eidos manual). Finally, the callback returns `T` to confirm to SLiM that the proposed mutation should actually be created.

Of course SLiM doesn't know that that stored value is a dominance coefficient; it will continue to use the dominance coefficient stored by the `m2` mutation type. The next step is therefore to redefine how the fitness of `m2` mutations is evaluated, with the `mutationEffect(m2)` callback shown above. This defines fitness in the standard way in SLiM, as  $(1+s)$  when homozygous and  $(1+hs)$  when heterozygous (see section 25.2), looking up the selection coefficient  $s$  from the `selectionCoeff` property of the focal mutation, `mut`. The twist is that instead of fetching the dominance coefficient  $h$  from `mut.mutationType.dominanceCoeff`, as would be standard (see

section 25.2), it uses `mut.getValue()` to fetch the dominance coefficient from the focal mutation using the key "dom" where it was previously stored.

The end result is that each beneficial mutation has its own independent dominance coefficient. The effect of this can be seen easily in SLiMgui, in the different sweep dynamics of different beneficial mutations; those with a dominance coefficient close to 0.0 take off slowly but sweep rapidly when they grow near fixation, whereas those with a dominance coefficient close to 1.0 take off quickly but often fluctuate uncertainly before managing to fix, as expected from theory.

In this recipe the dominance coefficients are drawn from a uniform distribution, but of course any distribution could be used. More generally, the dominance coefficient used by the `mutationEffect()` callback could come from anywhere; it could depend upon other genetic state within the focal individual, and could even vary over time, making it straightforward to model phenomena such as the evolution of dominance. This recipe assigns the dominance coefficients in the `mutation()` callback at the point when each new `m2` mutation is created, but the `mutationEffect()` callback could instead calculate the dominance coefficient from other state.

There are two lessons here, then. First of all, built-in support for variable dominance coefficients is not provided by SLiM, but dominance can easily be varied or determined in script. Second, a `mutation()` callback provides a convenient place to modify new mutations, add state to them, or even control whether or not they are created at all.

The recipe above has a disadvantage, however: `mutationEffect()` callbacks can be slow, since they are called for every mutation in every individual in every tick. If mutations that require these customized dominance coefficients are relatively uncommon, as they are in the recipe above, then it may not be too painful; but if such mutations are common, performance may be a problem. In that event, there is an alternative approach to modeling variable dominance coefficients: use multiple mutation types. Suppose, for example, that you wish to model a scenario in which the dominance coefficient depends upon the selection coefficient: we want to draw selection coefficients from a normal DFE, and neutral mutations should be completely recessive, while strongly beneficial or strongly deleterious mutations should be completely dominant, with gradations in between those extremes. If you need the spectrum of dominance coefficients to be continuously gradated, the approach above would be necessary. If, however, you can accept some discrete quantization of the gradations, then you can represent the discrete dominance levels with different mutation types. Here's a recipe demonstrating this approach:

```
initialize() {
  initializeMutationRate(1e-7);
  for (i in 0:10)
    initializeMutationType(i, i * 0.1, "n", 0.0, 0.02);
  initializeGenomicElementType("g1", m0, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
}
mutation(m0) {
  s = mut.selectionCoeff;
  d = asInteger(min(floor(abs(s) * 100.0), 10.0));
  mut.setMutationType(d);
  return T;
}
100000 late() { }
```

This uses a loop to set up eleven mutation types, `m0` to `m10`, with dominance coefficients of `0.0`, `0.1`, ..., `1.0`. (Note that you can pass an integer value to `initializeMutationType()` instead of a string; passing, e.g., `5` is equivalent to passing `"m5"`.) They are all set up with the same normal DFE, with a mean of `0.0` and standard deviation of `0.02`, but in fact each will be used for a different subrange of that DFE. The `initializeGenomicElementType()` call defines the `g1` genomic element type as using only `m0`; that mutation type is responsible for drawing all selection coefficients in the model. (The other ten mutation types could just as easily be defined as type `"f"` with a selection coefficient of `0.0`; it would make no difference, since SLiM never draws selection coefficients from them.)

The other necessary component of the recipe is the `mutation(m0)` callback, which gets called whenever a new `m0` mutation has been created by SLiM and is about to be added to a gamete during offspring generation. The callback here gets the selection coefficient that has been assigned to the mutation (from the normal DFE defined for `m0`), and does a little math to get its absolute value, scale it as desired, round it down to the nearest integer, clip it to a maximum of `10.0`, and then convert it from `float` to `integer`. The end result is an integer in `[0, 10]` that represents the mutation type we want to use for the mutation – the mutation type that will provide the dominance coefficient we want. We then call `setMutationType()` on the new mutation to set that mutation type on it (passing an integer, like `5` to present `m5`, as we did to `initializeMutationType()`). By changing the mutation type, we change the dominance coefficient, achieving the desired effect.

The number of mutation types defined determines the degree of granularity in the gradations of the dominance coefficients; if finer gradations are needed, simply use more mutation types. Since they are set up with a loop, this requires no additional code at all. The number of mutation types also has no impact on performance; you could use 1001 mutation types instead of 11 without any significant speed penalty.

This recipe runs in about 15 seconds on my machine (in SLiMgui), while the equivalent model using the technique of the previous recipe runs in about 160 seconds, so this change makes a very large difference – more than a 10x speedup. The reason is that a `mutationEffect()` callback is no longer needed, and all fitness calculations can be done in SLiM's core; the only overhead of this model beyond a vanilla SLiM model is the `mutation()` callback. That callback runs only once per mutation, when each mutation is first created, so its overhead is fairly negligible – about 6% of total runtime, according to SLiMgui's profiler.

The final lesson for this chapter on `mutationEffect()` callbacks, then, is that although `mutationEffect()` callbacks are extremely flexible, they can also be quite slow if they have to be called frequently. Alternative designs that avoid their use are often preferable.

## 11. Complex mating schemes

Since SLiM is based on the Wright–Fisher model, biparental mating in SLiM is normally random; for each offspring individual to be generated in a subpopulation, two parents are chosen at random from the appropriate subpopulation (which may not be the same subpopulation as the offspring’s subpopulation, if migration is involved). It is possible to modify this default behavior, however, by writing a special type of Eidos script block called a `mateChoice()` callback. These callbacks are documented comprehensively in the SLiM reference, in section 25.4. Here we will explore recipes that illustrate the use of `mateChoice()` callbacks to implement two different types of non-random mating: assortative mating, and sequential mate search.

Some mating schemes are actually better modeled using a different type of callback, a `modifyChild()` callback. Mostly, `modifyChild()` callbacks are used for purposes other than mating schemes; recipes using them are mostly in chapter 12 (and they are documented in the SLiM reference in section 25.5). However, the last recipe in this chapter will implement an interesting type of non-random mating, gametophytic self-incompatibility based on an S-locus, using a `modifyChild()` callback. See section 16.12 for another type of non-random mating, forcing SLiM to execute a specified pedigree, done by explicitly specifying mating pairs in a nonWF model.

### 11.1 Assortative mating

Assortative mating is the preference of an individual for mates that resemble the individual itself in some way. A species could exhibit assortative mating by size, for example, which would mean that smaller individuals tend to prefer other smaller individuals as mates, whereas larger individuals tend to prefer other larger individuals. Assortative mating is an important topic in evolutionary biology because it is thought to be important to the process of speciation: a population can diverge into genetically distinct lineages if assortative mating becomes strong enough to reproductively isolate phenotypically distinct subsets of the population from each other. Indeed, this mechanism can be so effective that if a single gene provides both adaptive phenotypic divergence and assortative mating based upon the diverging trait, the trait governed by the gene is called a “magic trait” because of its power to facilitate speciation (Servedio et al. 2011).


In this section we’ll look at a simple magic-trait model in SLiM (see sections 13.3 and 15.8 for further investigations of this topic). Since this is a relatively complex model, let’s put it together one piece at a time. The first piece is to set up the genetic and population structure:

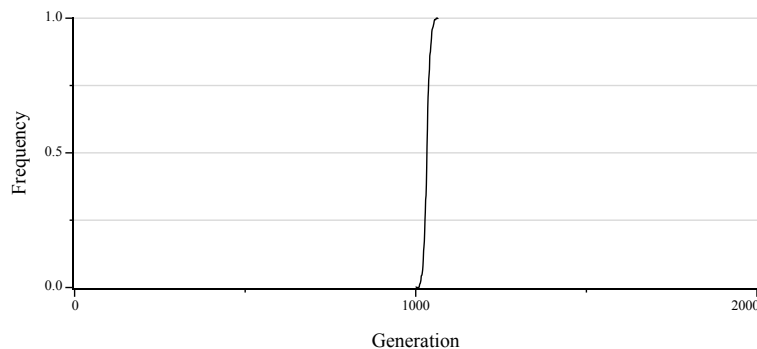
```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
    sim.addSubpop("p2", 500);
    p1.setMigrationRates(p2, 0.1);
    p2.setMigrationRates(p1, 0.1);
}
1000 late() {
    target = sample(p1.genomes, 1);
    target.addNewDrawnMutation(m2, 10000);
}
19999 late() { sim.simulationFinished(); }
```

This is very similar to the “introduced adaptive mutation” recipe of section 9.1: we have two mutation types, one neutral (m1) and the other adaptive (m2), and after running a burn-in period of 1000 ticks using only type m1, we introduce a single mutation of type m2 into a randomly chosen genome. The main difference between section 9.1’s recipe and this model is that here we have two subpopulations connected by migration; the gene flow introduced by the migration rate of 0.1 in both directions between the subpopulations is substantial.

When run, this model provides sweep dynamics similar to those seen in some recipes in the previous chapter. In the snapshot below, most of the neutral diversity has been swept away by the beneficial mutation introduced at position 10000, which is about to fix and is carrying a few neutral mutations along with it:



We can also examine the behavior of this model in SLiMgui using the Mutation Frequency Trajectories graph. First recycle the simulation, then click the Show Graph popup button  and open the Mutation Frequency Trajectories graph window from that menu. At this point, the simulation is not yet initialized, and so the graph is empty. Step forward one tick, over the `initialize()` callback; now the mutation types have been defined, and so you can choose mutation type m2 from the popup in the graph window. Play the simulation forward, and you should end up with a plot something like this, if the introduced mutation is not lost (if it is lost, you can just recycle and try again):



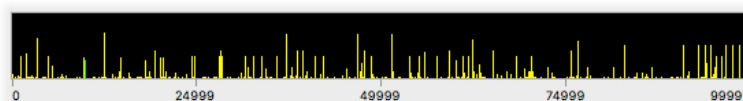
This plot illustrates that from the point at which the mutation was introduced, in tick 1000, it swept very rapidly to fixation.

For our magic-trait model we need divergent ecological selection between the subpopulations, which we introduce by adding a `mutationEffect()` callback that flips the fitness effect of the introduced mutation in subpopulation p2:

```
mutationEffect(m2, p2) { return 0.2; }
```

That’s all it takes. For simplicity, we are modeling a dominant mutation here, but we could easily use a scheme such as that in section 10.4.1 if the mutation were not dominant.

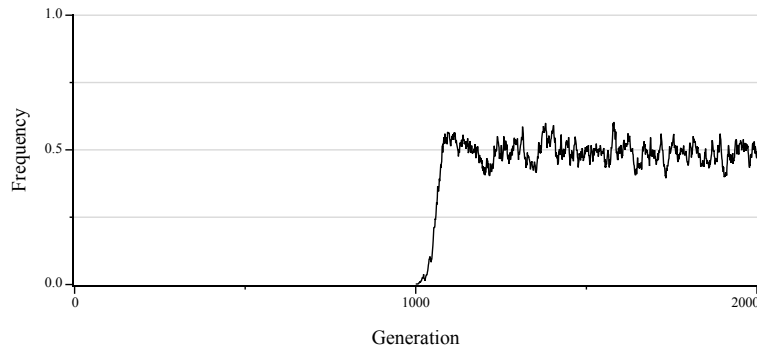
A little while after the introduced mutation arose, this looks like:





The introduced mutation is not fixing in this version of the model; indeed, it is stuck fluctuating around a frequency of 0.5, unable to increase further because it is deleterious in p2. Note that there is lots of neutral variation in this run, in contrast to the previous version of the model.

If we look at the Mutation Frequency Trajectories graph now, we see a very different pattern, illustrating how the spatial variation in selection is preventing the introduced mutation from fixing:



Finally, for a magic-trait model we need assortative mating based upon the same locus that is under divergent selection. In this model, that will be provided by a `mateChoice()` callback:

```
2000: mateChoice() {
    parent1HasMut = (individual.countOfMutationsOfType(m2) > 0);
    parent2HasMut = (sourceSubpop.individuals.countOfMutationsOfType(m2)
        > 0);
    if (parent1HasMut)
        return weights * ifelse(parent2HasMut, 2.0, 1.0);
    else
        return weights * ifelse(parent2HasMut, 0.5, 1.0);
}
```

This callback is a bit complicated, so let's walk through it. The first line determines whether the parent that is choosing a mate possesses the magic-trait mutation. The choosing parent is provided to the callback as an object named `individual`, of class `Individual`; this class is essentially a bag containing the two `Genome` objects belonging to the individual. The `countOfMutationsOfType()` method of `Individual` counts all occurrences of mutations of the given type in both of the individual's genomes; in other words, if the individual is homozygous for a given `m2` mutation, that mutation is represented twice in the count. Comparing the total count to 0 yields a single logical truth value indicating whether any mutation of that type is present in the individual. Note that the use of `> 0` means that we are ignoring dominance; for purposes of mate choice, we are treating the mutation as dominant. If we wanted heterozygotes to prefer heterozygotes and homozygotes to prefer homozygotes, or some such mating scheme, we could use the actual count instead; with a little change to the following logic that would work fine too.

Similarly, we can assess whether the other individuals in the subpopulation – the potential mates – possess the mutation using `countOfMutationsOfType(m2)` for all the individuals in the subpopulation. As previously, we use a `> 0` comparison to get a vector of logical values, `parent2HasMut`, indicating whether each individual in the subpopulation possesses the magic-trait mutation in either of its genomes.

Finally, we have a little logic at the end to determine the mating preferences we want to return. The standard fitness-based mating weights are supplied to the `mateChoice()` callback in the `weights` variable, and we don't want to ignore that; if we didn't use that vector at all, we would be entirely overriding SLiM's built-in fitness calculations, based on the selection coefficients of

mutations (which you may do if you wish, but is usually not desirable). Here, we combine `weights` multiplicatively with an assortative-mating term computed with `ifelse()`. The `ifelse()` function looks at each element of its first parameter (here, `parent2HasMut`) and chooses a corresponding element for the result; if the element from the first parameter is `T`, the element from the second parameter is chosen, whereas if the element from the first parameter is `F`, the element from the third parameter is chosen. The second and third parameters may be non-singleton vectors, too; `ifelse()` is a very powerful vectorized comparison function (see the Eidos documentation). Here, if the parent choosing a mate possesses the magic-trait mutation, candidate mates that also possess it get a multiplier of `2.0`, expressing the preference of carriers for other carriers. If the parent choosing a mate does not possess the magic-trait mutation, then candidate mates that possess it get a multiplier of `0.5`, expressing the dislike of non-carriers for carriers.

That version of the `mateChoice()` callback makes the logic clear, but unfortunately it is quite slow; for each individual choosing a mate, it calls `countOfMutationsOfType(m2)` on every individual in the source subpopulation. The `countOfMutationsOfType(m2)` method is slow, because it has to scan through the genomes of the target individual checking the type of each mutation. Furthermore, this work is almost entirely unnecessary, because the number of `m2` mutations possessed by each individual in a given tick doesn't change – we keep counting the same thing over and over even though we get the same answer every time. We can do much better, by replacing that version of the callback with this new logic:

```
2000: early() {
  // tag all individuals with their m2 mutation count
  inds = sim.subpopulations.individuals;
  inds.tag = inds.countOfMutationsOfType(m2);

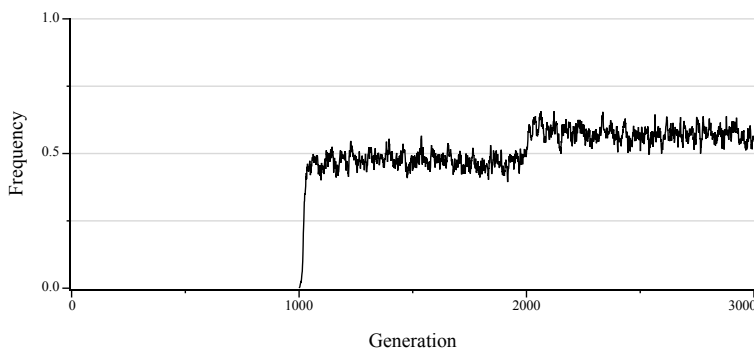
  // precalculate mating weights vectors
  for (subpop in c(p1,p2))
  {
    has_m2 = (subpop.individuals.tag > 0);
    subpop.setValue("weights1", ifelse(has_m2, 2.0, 1.0));
    subpop.setValue("weights2", ifelse(has_m2, 0.5, 1.0));
  }
}
2000: mateChoice() {
  if (individual.tag > 0)
    return weights * sourceSubpop.getValue("weights1");
  else
    return weights * sourceSubpop.getValue("weights2");
}
```

The new `2000: early()` event, which runs before offspring generation, calls `countOfMutationsOfType()` on every individual and caches the result in the `tag` field of the individuals; accessing these stored values will be much faster. Then it loops over the two subpopulations in the model and caches precalculated mating-weight vectors for each of them. Two vectors are cached: `weights1` is for use when the choosing parent has an `m2` mutation, `weights2` is for when it does not. The `mateChoice()` callback now uses those cached values; it checks `individual.tag` to find out whether the focal individual has an `m2` mutation, and uses either `weights1` or `weights2` from the source subpopulation accordingly.

The weights are kept by the subpopulations using a dictionary-like `getValue()` / `setValue()` mechanism (provided by `Dictionary`; see the Eidos manual). This `getValue()` / `setValue()` facility provides a way to attach named state to SLiM objects; it is similar to the `tag` property we have used in various other recipes, but is more flexible (allowing us to keep track of state of whole vectors of type `float` here, for example, whereas `tag` is limited to singleton integer values).

This type of optimization – moving the calculation of constant values out of callbacks into caches constructed in a previously run event – is very important for achieving maximal performance in SLiM models that use `mateChoice()` callbacks. It is also very useful for optimizing `mutationEffect()` callbacks, since they are often called a very large number of times.

If we recycle and run, and then look at the Mutation Frequency Trajectories graph, we can see the effects of this callback kick in at tick 2000, when the callback becomes active:



At tick 2000, when the `mateChoice()` callback becomes active, the frequency of the magic-trait allele jumps upward. If the two subpopulations were able to reach complete reproductive isolation through this mechanism, we would expect the frequency plotted here to equilibrate at 1.0 (because this graph shows the frequencies in subpopulation p1 only). In practice, full divergence is not possible, because WF models in SLiM are based upon juvenile dispersal, and fitness acts during mating (as opposed to causing mortality earlier in the generational life cycle). Given the migration rates declared in the model, approximately 10% of the individuals in each subpopulation will come from matings in the other population, every tick. Those migrants are also mating assortatively – they might be maladapted, but they will nevertheless preferentially mate and produce offspring that are also maladapted. Given the very high migration rate, adaptive divergence is helped only a little by the addition of assortative mating (but see below).

The rest of the chromosome, outside the magic-trait locus, is subject only to neutral mutations in this simple model. These neutral mutations can provide a means of monitoring the degree of divergence between the subpopulations; if the subpopulations become fully reproductively isolated, divergence at neutral sites should be observed, whereas without reproductive isolation divergence at neutral sites should be small or absent. This is often measured with a metric called  $F_{ST}$ : higher  $F_{ST}$  indicates greater genetic divergence among subpopulations. We can add code to start calculating the mean  $F_{ST}$  between p1 and p2 at tick 3000:

```
10000: late() {
    FST = calcFST(p1.genomes, p2.genomes);
    sim.setValue("FST", sim.getValue("FST") + FST);
}
19999 late() {
    cat("Mean FST at equilibrium: " + (sim.getValue("FST") / 500));
    sim.simulationFinished();
}
```

This uses a SLiM function named `calcFST()` that calculates the  $F_{ST}$  between two `Genome` vectors (see section 24.17.2; if you want to see the Eidos implementation of that function, just type `functionSource("calcFST")` in the Eidos console). The events above just call that function, record the results, and print out a summary at the end of the run (discussed below).

The mean  $F_{ST}$  for the tick is added to a value kept by the simulation using the same `getValue()` / `setValue()` mechanism we saw above, to keep a running total; that total is then used in tick 3499 to calculate the average  $F_{ST}$  over the ticks 3000:3499.

We also need to add a line to the tick 1 event, to zero out the running  $F_{ST}$  total:

```
sim.setValue("FST", 0.0);
```

If we run the model with the `mateChoice()` callback commented out, we get this output at the end of the run (for a run in which the focal mutation is not lost):

```
Mean FST at equilibrium: 0.0656018
```

Running it with the `mateChoice()` callback active, on the other hand, produces this output at the end of the run:

```
Mean FST at equilibrium: 0.107475
```

The magic trait therefore appears to have increased the level of divergence between the subpopulations at neutral loci, relative to the divergence for the same trait under natural selection but without the “magic” effect on mate choice, due to a decrease in gene flow. Single runs are of course not sufficient to show this convincingly; in practice, you would want to do many replications of both models, and do some statistics to show that the difference between the outcomes of the two models is significant. You might also wish to run for more ticks before starting to gather the  $F_{ST}$  statistics, and to gather them for a longer period of time, to assure that equilibrium had been reached and that transient dynamics did not weigh too heavily in the  $F_{ST}$  measurement. You might verify that the magic trait was not lost from the simulation; that happens occasionally, which of course defeats the purpose of the model. Finally, you might look separately at the  $F_{ST}$  at different positions along the chromosome, since it might be much higher near the magic trait locus than at more distant locations, due to linkage.

For the record, the complete model with all of the above components is:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 1.0, "f", 0.5); // introduced mutation
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.setValue("FST", 0.0);
  sim.addSubpop("p1", 500);
  sim.addSubpop("p2", 500);
  p1.setMigrationRates(p2, 0.1);
  p2.setMigrationRates(p1, 0.1);
}
1000 late() {
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m2, 10000);
}
mutationEffect(m2, p2) { return 0.2; }
2000: early() {
  // tag all individuals with their m2 mutation count
  inds = sim.subpopulations.individuals;
  inds.tag = inds.countOfMutationsOfType(m2);
```

```

// precalculate the mating weights vectors
for (subpop in c(p1,p2))
{
  has_m2 = (subpop.individuals.tag > 0);
  subpop.setValue("weights1", ifelse(has_m2, 2.0, 1.0));
  subpop.setValue("weights2", ifelse(has_m2, 0.5, 1.0));
}
}
2000: mateChoice() {
  if (individual.tag > 0)
    return weights * sourceSubpop.getValue("weights1");
  else
    return weights * sourceSubpop.getValue("weights2");
}
10000: late() {
  FST = calcFST(p1.genomes, p2.genomes);
  sim.setValue("FST", sim.getValue("FST") + FST);
}
19999 late() {
  cat("Mean FST at equilibrium: " + (sim.getValue("FST") / 10000));
  sim.simulationFinished();
}

```

## 11.2 Sequential mate search

In the previous section we saw how to implement assortative mating with a `mateChoice()` callback that modified the standard mating weights vector, `weights`, by multiplying it with an additional term derived from the genetic match between the focal individual and the other individuals in the subpopulation. In other words, the ultimate choice of mate was left to SLiM's machinery; the `mateChoice()` callback just modified the mating weights.

In this section we will explore a completely different kind of `mateChoice()` callback, one which makes the mate choice determination itself and tells SLiM's machinery which mate was chosen (if any). The standard weights vector will be used internally, within the `mateChoice()` callback; but the callback will simply return the chosen mate (as a singleton object vector of class `Individual`). In certain circumstances, it will instead return a zero-length vector as a signal to SLiM that no suitable mate could be found, initiating a new mate search with a new choosing parent.

The biological scenario here has to do with sequential mate search, the search for a mate by examining candidates sequentially until a suitable candidate is found or the breeding season ends. Conceptually, we will model a species like peacocks, in which the choosy parent is looking for a mate with a large ornament that is attractive but costly in fitness; however, to keep the model simple we will model hermaphrodites: all individuals can play both the chooser and the chosen.

Let's build the model one step at a time, first constructing the genetic and population structure:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", -0.025); // ornamental
  initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.01));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10001 early() {
  sim.simulationFinished();
}

```

This is straightforward: one population of 500 individuals, and a genetic structure that allows both neutral mutations (m1) and mutations that influence the ornament size of the individuals (m2). The ornamental mutations occur only about 1% as often as the neutral mutations, but may occur anywhere in the genome; this could be thought of as a “many genes of small effect”, quantitative-genetics sort of scenario. These ornamental mutations are all slightly deleterious, as genes that increase the size of a peacock’s tail presumably would be (apart from their positive effect on mating). If run, this model behaves as you might expect: neutral mutations accumulate, but no ornamental mutations fix, or even reach appreciable frequency, because of their deleterious effect.

Now we want to introduce the effect of the ornamental mutations on mating; to do so, we add a `mateChoice()` callback. This callback should implement sequential choosy mate choice by searching for a mate, preferring potential mates with more ornamental mutations:

```
mateChoice() {
  fixedMuts = sum(sim.substitutions.mutationType == m2);
  for (attempt in 1:5)
  {
    mate = sample(p1.individuals, 1, T, weights);
    osize = fixedMuts * 2 + mate.countOfMutationsOfType(m2);

    if (runif(1) < log(osize + 1) * 0.1 + attempt * 0.1)
      return mate;
  }
  return float(0);
}
```

The first line of this callback just totals up the number of ornamental mutations that have fixed. Once a mutation fixes, SLiM removes it from the active simulation, and from fitness calculations; since all individuals possess the fixed mutation, it has no differential effect on fitness or dynamics, in general. In this model, however, we want such fixed mutations to continue to influence mate choice, as described below. This could also be done by setting the `convertToSubstitution` property of m2 to F, as we have seen in some previous recipes; we’re using a different strategy here just to show a different angle on this common issue.

Next, the callback uses a for loop to make up to five attempts at finding a mate. Each attempt is a little less picky than the previous attempt, reflecting declining standards as breeding season proceeds. If all five attempts fail, `float(0)` is returned to indicate that the individual failed to find a mate. Within each attempt, a candidate mate is chosen using the `sample()` function, using the standard fitness-based weights vector as the weights for sampling; the deleterious effect of the ornamental mutations is thus still taken into account, reducing the likelihood that highly ornamented individuals will be chosen as mates for survival- or growth-based reasons. The total ornament size of the candidate mate is calculated, using both fixed and unfixed ornamental mutations. Finally, `runif()` generates a random uniform draw (between 0 and 1, by default), and that drawn value is compared to a threshold value determined by the candidate mate’s ornament size as well as the attempt number; if the candidate gets sufficiently lucky, it ends up as the chosen mate. When a mate is chosen, it is simply returned. It would be possible instead to construct a new weights vector, with 1.0 for the chosen mate and 0.0 for all other entries, and return that to force SLiM to choose that individual; simply returning the individual is a shorthand that can be handled by SLiM far more efficiently than can a returned weights vector (see section 25.4).

If you run this model, you can now see the ornamental mutations increasing in frequency and fixing despite their deleterious fitness effect, because they are favored by sexual selection. We might be interested in the final ornament size attained, so we can flesh out the final event:



```

1:10001 early() {
  if (sim.cycle % 1000 == 1) {
    fixedMuts = sum(sim.substitutions.mutationType == m2);
    osize = fixedMuts * 2 + p1.individuals.countOfMutationsOfType(m2);
    catn(sim.cycle + ": Mean ornament size == " + osize);
  }
}

```

This prints the mean ornament size in the population every thousand cycles. A typical run of the model would produce output like:

```

1: Mean ornament size == 0
1001: Mean ornament size == 4.642
2001: Mean ornament size == 7.086
3001: Mean ornament size == 7.932
4001: Mean ornament size == 8.004
5001: Mean ornament size == 8.036
6001: Mean ornament size == 8.048
7001: Mean ornament size == 8.654
8001: Mean ornament size == 8.234
9001: Mean ornament size == 8.408
10001: Mean ornament size == 8

```

Individuals have evolved to possess, on average, about eight ornamental mutations. It looks like an equilibrium has been reached at that point; if increasing ornament size any further would provide a fitness advantage, the advantage is clearly not large.

The equilibrium ornament size is a result of a balance between the direct deleterious fitness effect of a large ornament and the indirect beneficial fitness effect due to being chosen more frequently as a mate. The indirect benefit from mate choice is governed by the expression for the threshold probability of being chosen as a mate:

```
log(osize + 1) * 0.1 + attempt * 0.1
```

This function is designed to provide diminishing returns; as the ornament size increases, the marginal benefit of increasing ornament size diminishes. Meanwhile, the marginal deleterious effect of increasing ornament size remains constant. At some point – around an ornament size of 8, apparently – the benefit is no longer worth the cost. The model therefore reaches an equilibrium ornament size, just as happens in reality with ornamented species subject to both natural selection against the ornament and sexual selection for the ornament, such as peacocks.

The decreasing choosiness over the course of the five mate-choice trials perhaps influences the equilibrium point, although the direction of this influence is not entirely clear; perhaps decreasing choosiness means that one can afford to have a smaller ornament size and just wait to be chosen in a later choice round, or perhaps it means that one needs to have a larger ornament so as to try to guarantee that one is chosen in the first round, before one's competitors become more viable choices due to decreased choosiness in the later rounds.

One point worth noting is the fact that  $\log(\text{osize} + 1)$  is used, rather than  $\log(\text{osize})$ . At the beginning of the run, every individual has an ornament size of zero, and so  $\log(\text{osize})$  would be  $-\text{INF}$ . If we allowed that to occur, SLiM would be unable to find any mates at all in the first generation; it would try a large number of times, and then would terminate with an error. There are lots of ways one could prevent that from happening; using  $\log(\text{osize} + 1)$  was the strategy chosen here.

Here is the full model with all of the above components:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", -0.025); // ornamental
  initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.01));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
1:10001 early() {
  if (sim.cycle % 1000 == 1) {
    fixedMuts = sum(sim.substitutions.mutationType == m2);
    osize = fixedMuts * 2 + p1.individuals.countOfMutationsOfType(m2);
    catn(sim.cycle + ": Mean ornament size == " + mean(osize));
  }
}
mateChoice() {
  fixedMuts = sum(sim.substitutions.mutationType == m2);
  for (attempt in 1:5)
  {
    mate = sample(p1.individuals, 1, T, weights);
    osize = fixedMuts * 2 + mate.countOfMutationsOfType(m2);

    if (runif(1) < log(osize + 1) * 0.1 + attempt * 0.1)
      return mate;
  }
  return float(0);
}

```

This recipe is perhaps a good case study for how to optimize callbacks in SLiM, since the large majority of runtime is spent in the `mateChoice()` callback – about 90%, according to SLiMgui’s built-in profiling feature (see section 21.5). We can redesign the `1:10001 early()` event and the `mateChoice()` callback (leaving the `initialize()` callback and the subpopulation creation event the same) to achieve about a one-third speedup:

```

1:10001 early() {
  fixedMuts = sum(sim.substitutions.mutationType == m2);
  inds = p1.individuals;
  osize = fixedMuts * 2 + inds.countOfMutationsOfType(m2);
  inds.tagF = log(osize + 1) * 0.1;

  if (sim.cycle % 1000 == 1)
    catn(sim.cycle + ": Mean ornament size == " + mean(osize));
}
mateChoice() {
  for (attempt in 1:5)
  {
    mate = sample(p1.individuals, 1, T, weights);

    if (runif(1) < mate.tagF + attempt * 0.1)
      return mate;
  }
  return float(0);
}

```

The difference is that the `early()` event now precalculates the base mating appeal of every individual, with a vectorized method call and some other vectorized calculations, and puts the results into the `tagF` property of the individuals with a vectorized property assignment (see the

Eidos manual for review of these vectorized features of the language). The `mateChoice()` callback can then be much simpler, using the saved `tagF` values instead of doing individual, non-vectorized calculations. This design is faster in part because vectorized operations are intrinsically faster than doing the same calculations one at a time (see section 21.1). It is also faster because the mating appeal of every individual is calculated just once; given the sequential-search design of this model, the mating appeal of many individuals might be used more than once. With the first design of the `mateChoice()` callback, each such use of a given individual's mating appeal would require it to be recalculated; with the second design, the precalculated value merely needs to be re-fetched from `tagF`. The model is still not terribly fast – the `sample()` function is intrinsically slow when passed a vector of weights, which we still have to do at least once for every `mateChoice()` callback – but a one-third speedup is not bad.

Since the `sample()` calls are the large majority of the time spent in this revised recipe, optimizing that aspect of the code would be the next step if more speed were needed. The strategy would be similar to what we have already done: vectorize the `sample()` calls, rather than sampling just one mate at a time. This would be simplest to do in a nonWF model (see section 1.6); we could draw all the first parents with a single `sample()` call, then draw all of their first mate choices with another single `sample()` call, then assess which mates were deemed acceptable with a single vectorized `runif()` call and a vectorized comparison, and so forth. This design works much better in a nonWF model because the `reproduction()` callback of a nonWF model can handle all mating events in a single call, potentially in a vectorized fashion (see, e.g., section 16.15), whereas `mateChoice()` callbacks in WF models are fundamentally and irredeemably unvectorized – they are called once per first parent, and the only way to vectorize them is to move calculations out of them entirely (as we did above with `tagF`). Switching to a nonWF design would also get rid of the need to use a weights vector with `sample()`, since the deleterious fitness effects of ornament size would then be taken care of by mortality rather than reduced mating success – some individuals would die as a result of selection against their ornament, and then the survivors would mate influenced only by the positive effects of ornament size on mating appeal. This seems closer to the typical biological reality, for species like peacocks, while also providing a large performance benefit by eliminating the need for the sampling weights. We will not show such an optimized nonWF model here, since we won't even get to nonWF models until chapter 16; but it ought to be reasonably straightforward to implement, and would likely be significantly faster than even the optimized design shown above.

### 11.3 Gametophytic self-incompatibility

In the previous sections we have seen the use of `mateChoice()` callbacks to implement assortative mating and sequential mate choice, two types of non-random mating. In this section, we will see the use of a different type of callback, a `modifyChild()` callback, to model an interesting type of non-random mating, gametophytic self-incompatibility based on S-locus alleles.

Gametophytic self-incompatibility is quite a common mating scheme in plants (Newbigin et al., 1993). In a nutshell, the pollen grain (the male gamete, which is haploid) expresses the particular allele that it possesses at a special locus called the S-locus. When a pollen grain lands on the stigma of a female flower and begins to grow a pollen tube down the style towards the ovaries of the flower, the pollen tube expresses the S-locus allele of the pollen grain as it grows. Female flowers, in their styles, do some sort of check of the S-locus allele being expressed by the pollen tube, and if it exactly matches an S-locus allele possessed by the female plant (on either of its two copies of that locus, since the female flower is part of a diploid plant), it halts the growth of the pollen tube, preventing fertilization. The reasons why this mechanism might be beneficial are thought to be related to promotion of outcrossing and prevention of selfing.

Why use a `modifyChild()` callback instead of a `mateChoice()` callback to model a mate choice scheme such as gametophytic self-incompatibility? There are several considerations. Conceptually, a `mateChoice()` callback allows control over how likely every possible mating pair in a population is to form, whereas a `modifyChild()` callback is about controlling the outcome of a specific mating – governing whether that mating is fertile and what the genetic outcome of the mating is. Gametophytic self-incompatibility is not really about the choice of mates across the population (pollen lands indiscriminately on all female flowers, at least without complications such as heterostyly or enantiostyly); instead, it is about whether the combination of a specific pollen grain and a specific flower is fertile or infertile, making `modifyChild()` a natural choice.

Another consideration is that `mateChoice()` callbacks are about pre-mating reproductive isolation, whereas `modifyChild()` callbacks are about post-mating reproductive isolation (among other uses). Gametophytic self-incompatibility depends upon the S-locus allele expressed by the haploid pollen grain; that information is simply not available in a `mateChoice()` callback, since gametes have not yet been produced at that stage. From this perspective, then, `modifyChild()` is actually a forced choice for this recipe.

A third consideration is that if a `mateChoice()` callback rejects a first parent completely, SLiM's mating algorithm goes back to try a different first parent within the same source subpopulation, whereas if a `modifyChild()` callback suppresses a child completely, SLiM's mating algorithm re-chooses the source subpopulation as well, based upon the migration rates set for the target subpopulation (see section 22.2). This difference reflects the pre-mating versus post-mating semantics of the two callbacks, and makes good sense here; if a particular source subpopulation's pollen grains have a high probability of being incompatible with the female flowers of the target subpopulation, that source subpopulation should be underrepresented in gene flow into the target subpopulation, compared to the expected gene flow based upon pollen flow alone.

A final consideration, if none of the previous issues decides the question, might be speed. Unfortunately, `mateChoice()` callbacks tend to be quite slow. In computer science parlance, they are generally  $O(N)$ , meaning that the computation time taken by a `mateChoice()` callback is proportional to the number of individuals in the subpopulation; this is because each individual must be evaluated to produce a mating weight. On the other hand, `modifyChild()` callbacks are generally  $O(1)$ , meaning they take a constant amount of time regardless of population size; this is because only the product of one mating event is considered by the callback. In practice, this algorithmic speed difference can result in a very large difference to the running speed of a model. Usually, however, the previous considerations will dictate the choice of implementation.

Let's build the model in two steps, beginning without the `modifyChild()` callback:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", 0.0); // S-locus mutations
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElementType("g2", m2, 1.0);
  initializeGenomicElement(g1, 0, 20000);
  initializeGenomicElement(g2, 20001, 21000);
  initializeGenomicElement(g1, 21001, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 late() {
  cat("m1 mutation count: " + sim.countOfMutationsOfType(m1) + "\n");
  cat("m2 mutation count: " + sim.countOfMutationsOfType(m2) + "\n");
}
```

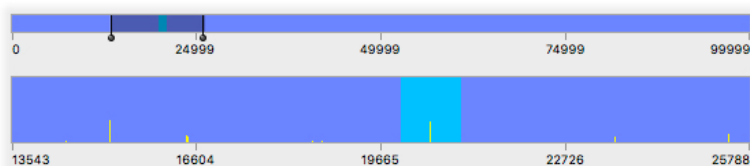
We have two mutation types, both neutral, and a chromosome that uses mutation type m1 over most of its length (with g1) but mutation type m2 within a small (1000-bp) locus (with g2). That small locus is of course the S-locus, but without the `mateChoice()` callback it acts identically to the rest of the chromosome. Since there is not yet any code to enforce a difference between the S-locus and the rest of the chromosome, this recipe is, at this point, simply a model of neutral drift.


In the tick 10000 event, it outputs two metrics prior to termination: the number of mutations of type m1 and of type m2. These serve as a quick-and-dirty measure of genetic diversity, both across the bulk of the chromosome (the m1 count) and within the S-locus (the m2 count). A typical test run of this recipe produces something like:

```
m1 mutation count: 124
m2 mutation count: 1
```

You can run the model a bunch of times and confirm that there is not a whole lot of variation around these numbers; the m1 count is typically 100–200, the m2 count typically 0–5.

It is also interesting to look at the graphical representation of the chromosome in SLiMgui. Selecting a subrange of the chromosome, so as to zoom in on just one section, a typical run of the recipe so far looks like:



This view has display of genomic elements turned on, using the  action button, so that the location of the S-locus is clear. The two things to note here are that the distribution of neutral mutations is fairly sparse, and that the distribution of neutral mutations inside versus outside the S-locus is comparable.

Now let's add in the `modifyChild()` callback that implements the gametophytic self-incompatibility system:

```
modifyChild(p1) {
  pollenSMuts = child.genome2.mutationsOfType(m2);
  styleSMuts1 = parent1.genome1.mutationsOfType(m2);
  styleSMuts2 = parent1.genome2.mutationsOfType(m2);
  if (identical(pollenSMuts, styleSMuts1))
    if (runif(1) < 0.99)
      return F;
  if (identical(pollenSMuts, styleSMuts2))
    if (runif(1) < 0.99)
      return F;
  return T;
}
```

First, the callback gets a vector of all of the S-locus mutations present in the pollen grain. The `child` variable is defined by SLiM within `modifyChild()` callbacks, and `child.genome2` represents the gamete produced by the second parent in the mating; see section 25.5 for details. The result, `pollenSMuts`, represents the S-locus allele expressed by the pollen grain. All mutations at the S-locus are considered, in this model, to change the expressed S-allele; it would be trivial to add in the possibility of neutral mutations at the S-locus as well, by adding in a probability of type m1 mutations in genomic element type g2.

Next, the callback gets the two S-locus alleles of the female flower by fetching the vector of mutations of type `m2` from `parent1.genome1` and `parent1.genome2`, the two homologous genomes of the first parent (again, `parent1` being defined by SLiM in these callbacks; see section 25.5).

Next comes the crucial step at which the growth of the pollen tube is stopped if there is an incompatibility between the S-allele of the pollen and either of the S-alleles of the female flower. This is checked using the Eidos function `identical()`, which checks whether its two arguments are exactly identical. We don't need to worry about non-identity due to mutations being listed in a different order in the vectors, because `Genome` keeps its list of mutations in sorted order, and `mutationsOfType()` returns a sorted subset of that list.

If the pollen S-allele is identical to either of the female flower's S-alleles, there is a 99% probability (in this model) of the pollen tube being blocked, as implemented by the test `(runif(1) < 0.99)`. Returning `F` in these cases tells SLiM to suppress generation of the proposed child altogether; this is, conceptually, the stoppage of the pollen tube. It is important to guarantee that a `modifyChild()` callback will never suppress 100% of all proposed children, for any state that your model might reach; if that ever happens, SLiM will get stuck generating an infinite succession of proposed children that all get suppressed (and will eventually detect this condition and terminate with an error message). In this case, the model would not quite hang without the `runif()` test, since eventually SLiM would manage to generate pollen grains that all happened to have a mutation within the S-locus that made them compatible with the available female flowers; but it would take an awfully long time. Indeed, even at 99% the first generation can take a while to finish, and many of the individuals in the second generation will have an S-locus mutation because of the effective imposition of gametophytic self-incompatibility in a single generation. A more realistic model might perhaps "phase in" the gametophytic self-incompatibility slowly over some thousands of generations, reflecting the gradual evolution of an increasingly strong mechanism, by making the threshold against which `runif()` is compared depend upon `sim.cycle`. This is left as an exercise for the reader; it should not change the final state of the model at equilibrium (although equilibrium might take a lot more than the 10000 ticks we use here).

The final line simply returns `T`, indicating that since the pollen grain was compatible with the female flower, the proposed child can be generated.

The full recipe, for the record:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", 0.0); // S-locus mutations
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElementType("g2", m2, 1.0);
  initializeGenomicElement(g1, 0, 20000);
  initializeGenomicElement(g2, 20001, 21000);
  initializeGenomicElement(g1, 21001, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 late() {
  cat("m1 mutation count: " + sim.countOfMutationsOfType(m1) + "\n");
  cat("m2 mutation count: " + sim.countOfMutationsOfType(m2) + "\n");
}
modifyChild(p1) {
  pollenSMuts = child.genome2.mutationsOfType(m2);
  styleSMuts1 = parent1.genome1.mutationsOfType(m2);
  styleSMuts2 = parent1.genome2.mutationsOfType(m2);
  if (identical(pollenSMuts, styleSMuts1))
```



```

    if (runif(1) < 0.99)
      return F;
  if (identical(pollenSMuts, styleSMuts2))
    if (runif(1) < 0.99)
      return F;
  return T;
}

```

If you run the full recipe, you should get output something like:

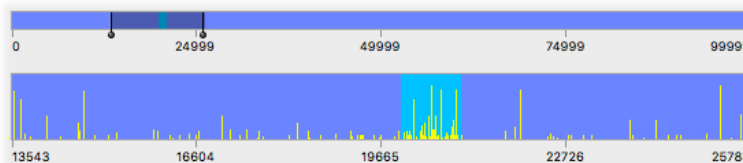
```

m1 mutation count: 582
m2 mutation count: 55

```

There is vastly more genetic diversity now, both within the S-locus (mutation type m2) and across the whole chromosome (type m1). Gametophytic self-incompatibility basically imposes a regime of balancing selection (i.e., negative frequency-dependent selection) at the S-locus, and that regime tends to preserve allelic diversity at the locus (see section 10.4.1 for a different model of negative frequency-dependent selection). It also increases the outcrossing rate in the population, particularly when there are relatively few S-alleles (when there are many S-alleles, it would mostly tend to diminish selfing, since most non-selfing mating pairs would possess different S-alleles anyway; it would be trivial, of course, to add selfing to this model to experiment with that additional nuance, as seen in section 6.3.1).

Examining the same subsection of the chromosome as before, in SLiMgui, it now looks like this:



Compared to the previous snapshot, the increase in genetic diversity across the chromosome is immediately apparent. It is also striking how much more diversity is contained within the S-locus itself (due to the balancing selection there) compared to the rest of the chromosome. Note that in the implementation of this model, the number of S-alleles is not just the number of different mutations within the S-locus that are circulating in the population. Instead, every unique haplotype within the S-locus as a whole represents a different S-allele, and new S-alleles can be generated in this model by recombination as well as by mutation. Other schemes for evaluating what an S-allele “really” is, based upon the mutations present at the S-locus, could be implemented using a different test than `identical()` in the `modifyChild()` callback.

## 12. Direct child modifications

Thus far we have seen two ways of modifying SLiM's basic Wright–Fisher model: `mutationEffect()` callbacks that modify the default fitness effect of mutations (chapter 10), and `mateChoice()` callbacks that alter the default behavior of random mating (chapter 11). In this chapter we will look at a third type of modification, `modifyChild()` callbacks. These allow you to modify children generated by SLiM (by adding or removing mutations, for example), or to suppress particular children entirely (see section 25.5 for a full technical discussion). Here, we will use `modifyChild()` callbacks to solve a variety of problems, from social learning of cultural traits to simulation of a “gene drive” based upon CRISPR/Cas9. Note that section 11.3 also used a `modifyChild()` callback, to implement a gametophytic self-incompatibility system.

### 12.1 Social learning of cultural traits

In section 10.4.2 we explored a simple model of a cultural trait; the `tag` value of `Individual` was used to track whether each individual was a milk-drinker or not, and a `mutationEffect()` callback was used to make mutations promoting the production of lactase into adulthood beneficial for milk-drinkers but neutral for non-milk-drinkers. In that model, whether a given individual was a milk-drinker or not was determined at random, in a `late()` callback that tagged all offspring with their assigned cultural group. We will now extend that model to include social learning: individuals will tend to inherit milk-drinking from their parents, although a substantial random factor in the cultural assignment will be retained. To achieve this, instead of assigning the cultural group in a `late()` event, we will do it in a `modifyChild()` callback, so that we have the information we need regarding the culture of the parents. The recipe:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0); // neutral
    initializeMutationType("m2", 0.5, "f", 0.1); // lactase-promoting
    m2.convertToSubstitution = F;
    initializeGenomicElementType("g1", c(m1,m2), c(0.99,0.01));
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 1000);
    p1.individuals.tag = rbinom(1000, 1, 0.5);
}
modifyChild() {
    parentCulture = (parent1.tag + parent2.tag) / 2;
    childCulture = rbinom(1, 1, 0.1 + 0.8 * parentCulture);
    child.tag = childCulture;
    return T;
}
mutationEffect(m2) {
    if (individual.tag == 0)
        return 1.0; // neutral for non-milk-drinkers
    else
        return effect; // beneficial for milk-drinkers
}
10000 early() { sim.simulationFinished(); }
```

The setup of the model is similar to its predecessor in section 10.4.2: mutation type `m2` is used to represent alleles that promote retention of lactase production (and removal of fixed `m2` mutations is prevented with `convertToSubstitution`), tag values of 1 are used to indicate milk-drinkers, and

the same `mutationEffect()` callback as in section 10.4.2 produces a differential fitness effect of `m2` mutations depending upon the culture of the individual. What has changed is the way that the `tag` values are maintained. The `late()` event has been removed. We now assign initial `tag` values in tick 1, immediately after creating subpopulation `p1`. Thenceforth, the `tag` values of new individuals are assigned in the `modifyChild()` callback:

```
modifyChild() {
  parentCulture = (parent1.tag + parent2.tag) / 2;
  childCulture = rbinom(1, 1, 0.1 + 0.8 * parentCulture);
  child.tag = childCulture;
  return T;
}
```

This callback is called once for every new offspring individual created, throughout the run of the model. Its first line gets the `tag` values of the two parents (using `parent1` and `parent2` variables that are set up for all `modifyChild()` callbacks; see section 25.5 for a complete list of these variables), and averages them to get a `parentCulture` value that will be 0, 0.5, or 1. The next line determines what the child's culture will be (0 or 1) using `rbinom()` to draw from a binomial distribution; the trial success probability used for the draw depends on `parentCulture` in such a way as to make children tend to follow the culture of their parents, but with a 10% chance of deviating even if the parents share the same culture. To make this cultural determination take effect, `childCulture` is assigned into the `tag` property of the child (using the `child` variable defined for the callback by SLiM). Finally, a value of `T` is returned to indicate that the child should be generated; it is possible for a `modifyChild()` callback to suppress the generation of some children by returning `F`.

In the original model of section 10.4.2, the fraction of milk-drinkers remained about 0.5, because assignment into a cultural group was random. In this model, in contrast, the fraction of milk-drinkers can increase over time as individuals learn milk-drinking from their parents; the fitness benefit of milk-drinking increases as more `m2` mutations sweep. It is easy to observe this in SLiMgui by adding a custom output event:

```
late() {
  if (sim.cycle % 100 == 0)
    cat(sim.cycle + ": " + mean(p1.individuals.tag) + "\n");
}
```

This prints the fraction of milk-drinkers in the population, every 100 generations. Running this model produces output that shows the progressive increase in milk-drinking:

```
100: 0.465
200: 0.567
300: 0.552
400: 0.681
500: 0.735
...
```

However, there will always be a minimum of about 10% non-milk-drinkers in the population, because of the element of chance provided by the binomial draw in the `modifyChild()` callback. It would be possible to modify that to allow a specific culture to fix in the population. On the other hand, one could also model the fact that milk-drinkers who do not retain lactase production into adulthood will typically suffer from symptoms of lactose intolerance, making milk-drinking disadvantageous in some circumstances. One could also model a slight deleterious effect of lactase retention genes among non-milk-drinkers, since they are devoting energy to producing an enzyme that they do not use. There is always more complexity to be added; but as it stands, this

model shows the coevolution of both genetic and cultural factors related to milk-drinking, using tag values in combination with a `modifyChild()` callback to model social learning in SLiM.

## 12.2 Lethal epistasis

In section 10.3.1 we saw a model of epistasis that influenced the fitness of the epistatic alleles using a `mutationEffect()` callback. Sometimes the situation is simpler than the scenario presented in that model: sometimes the fitness of individuals carrying both epistatic alleles is zero, making the epistatic interaction lethal. In this case, a `modifyChild()` callback is well-suited to the task, since it can suppress the generation of particular offspring depending upon their genetics (or other factors). In this case, we will model two introduced mutations, A and B, which are normally beneficial, but are lethal when they occur in the same offspring:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 0.5, "f", 0.5); // mutation A
    m2.convertToSubstitution = F;
    initializeMutationType("m3", 0.5, "f", 0.5); // mutation B
    m3.convertToSubstitution = F;
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
1 late() {
    sample(p1.genomes, 20).addNewDrawnMutation(m2, 10000); // add A
    sample(p1.genomes, 20).addNewDrawnMutation(m3, 20000); // add B
}
modifyChild() {
    hasMutA = any(child.genomes.countOfMutationsOfType(m2) > 0);
    hasMutB = any(child.genomes.countOfMutationsOfType(m3) > 0);
    if (hasMutA & hasMutB)
        return F;
    return T;
}
10000 early() { sim.simulationFinished(); }
```

The mechanics here for the introduction of mutations A and B follow the pattern we have seen in other recipes: a target vector of genomes is chosen with `sample()`, and then a mutation is added to the target genomes with `addNewDrawnMutation()`. Unlike most previous recipes, however, here we sample 20 genomes, not just one, in order to add the new mutation to multiple individuals. Because `addNewDrawnMutation()` is a class method of `Genome`, not an instance method, a single new mutation is added to all of the target genomes, rather than a different new mutation being added to each target genome as a result of multicasting; see section 10.4.3 for discussion of the mechanics of this. No effort is made here to avoid adding A and B to the same individuals, but that would be trivial to add by refining the choice of targets for B. The mutation types for A and B, `m2` and `m3`, are set not to convert fixed mutations to substitutions, so that their epistatic interaction persists even after fixation; see section 10.3.1 for extensive discussion of this.

The new and interesting behavior is in the `modifyChild()` callback. First it determines whether the proposed child possesses mutation A and mutation B, setting up logical flags `hasMutA` and `hasMutB` by checking whether the count of mutations of the relevant type is greater than zero in

either of the child's genomes. Then, if the child has both A and B, it returns F, indicating that this child should be suppressed. New parents will be chosen and a new child will be generated instead (also subject to the approval of the `modifyChild()` callback). Otherwise it returns T, indicating the generation of the child should proceed.

When this model is run, either A or B quickly “wins”, fixing while its epistatic competitor is lost. If the `modifyChild()` callback is removed, on the other hand, both A and B will typically fix. We can try an interesting variant by making the epistatic interaction lethal only if A and B are both homozygous, by replacing the previous `modifyChild()` callback with a new version:

```
modifyChild() {
    mutACount = sum(child.genomes.countOfMutationsOfType(m2));
    mutBCount = sum(child.genomes.countOfMutationsOfType(m3));
    if ((mutACount == 2) & (mutBCount == 2))
        return F;
    return T;
}
```

This results in a form of balancing selection between A and B, since now homozygosity is disadvantageous but heterozygosity is advantageous. Both will tend to fluctuate around a frequency of 0.5, although one may stochastically manage to fix eventually (in which case the other will almost immediately be lost).

This type of model could also be used to represent an epistatic interaction during development that is lethal in some fraction of cases, but is otherwise harmless – either the epistatic interaction causes a lethal anomaly during development, or it doesn't, in which case the offspring is normal. This could be modeled simply by adding a random factor to the `if` statement in the `modifyChild()` callback.

### 12.3 Simulating gene drive

There has recently been a lot of buzz about a new genetic-engineering technology called CRISPR/Cas9 that allows genetic modifications to be performed much more quickly and easily than previous methods (Doudna & Charpentier 2014). One potential application of the CRISPR/Cas9 machinery is to use it for the construction of a so-called *gene drive*, which can quickly drive genetically modified alleles to high frequency in a population even if they carry a fitness cost. We will here refer to a CRISPR/Cas9-based gene drive with the term mutagenic chain reaction, or MCR.

The basic idea of MCR is that the CRISPR/Cas9 machinery embedded into the organism's genome could cause the machinery itself to be spliced into any homologous chromosome that does not already contain it. If a fertilized egg ends up with one copy of the MCR machinery, inherited from one parent, the machinery could then splice itself into the homologous chromosome in the egg, changing the egg from being heterozygous to homozygous for the MCR locus. This is in some ways similar to the idea of meiotic drive (see section 16.17) and related concepts in evolutionary biology, and it clearly underlines the fundamental truth of the “selfish gene” perspective: an MCR gene of this type could, as we shall see, spread to fixation in a population even if it has a deleterious fitness effect at the level of the individual organism. Such selfish genes can even drive a species to extinction, in extreme cases.

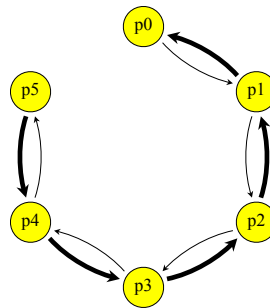
Just for fun, we're going to make this model relatively complex in terms of population structure, and we're going to introduce spatial variation in selection using a `mutationEffect()` callback as well. We'll build this model one step at a time, starting with the demographic structure:

```

initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", -0.1); // MCR complex
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  for (i in 0:5)
    sim.addSubpop(i, 500);
  for (i in 1:5)
    sim.subpopulations[i].setMigrationRates(i-1, 0.001);
  for (i in 0:4)
    sim.subpopulations[i].setMigrationRates(i+1, 0.1);
}
10000 late() { sim.simulationFinished(); }

```

This sets up a six-subpopulation stepping-stone model, similar to that previously discussed in section 5.3.1. Note that migration in this model is primarily from p5 down to p0; the migration rate in that direction is a hundred times higher than in the direction from p0 up to p5:



The code above sets up a mutation type m2 for the MCR complex, but doesn't use it, so at present this is just a simulation of neutral drift in a stepping-stone model. Let's start using mutation type m2, although not yet with its planned MCR capabilities, by adding a little code to introduce an m2 mutation and track its fate (replacing the final script block in the previous model):

```

100 late() {
  p0.genomes[0:49].addNewDrawnMutation(m2, 10000);
}
100:10000 late() {
  if (sim.countOfMutationsOfType(m2) == 0)
  {
    fixed = any(sim.substitutions.mutationType == m2);
    cat(ifelse(fixed, "FIXED\n", "LOST\n"));
    sim.simulationFinished();
  }
}

```

The introduction code here simply introduces the mutation into the first 50 genomes of subpopulation p0, without bothering to randomly select target individuals using `sample()`. Introducing the mutation into many genomes at once is a quick-and-dirty trick to help an introduced mutation avoid being lost due to genetic drift at the earliest stages of establishment, without putting in the machinery to make the simulation truly conditional on establishment as we did in section 9.3. Biologically, it could be viewed as a large-scale immigration event, or as a

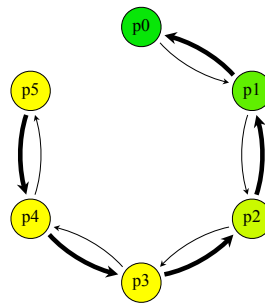


planned introduction of mutant individuals orchestrated by humans. See section 10.4.3 for discussion of the Eidos mechanics underlying this mutation introduction code.

This model will generally terminate, perhaps around tick 140, with the message "LOST". This is because mutation type m2 is strongly deleterious, as presently defined, and so gets eliminated by selection fairly quickly even though it is initially introduced in fifty copies. Since this is not very interesting, let's move on to the next step: making mutation type m2 strongly beneficial in subpopulation p0 and strongly deleterious in subpopulation p5, with gradations in between, using a `mutationEffect()` callback:

```
mutationEffect(m2) {
    return 1.5 - subpop.id * 0.15;
}
```

This `mutationEffect()` callback uses `subpop.id`, the identifier of the subpopulation in which the mutation is presently being evaluated, to generate a fitness of 1.5 for p0, but a fitness of 0.75 for p5. If we run the model now, we see that it soon reaches an equilibrium state of migration-selection balance:



It is at high frequency in p0 – nearly fixed, but prevented from fixing completely by gene flow from p1. It is essentially absent from p5, on the other hand, since it is weeded out by selection, and since gene flow in the p0 to p5 direction is so weak. The model will tend to maintain this dynamic equilibrium.

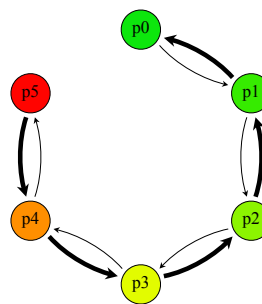
Let's add the `modifyChild()` callback that implements the MCR behavior of m2:

```
100:10000 modifyChild() {
    mut = sim.mutationsOfType(m2);
    if (size(mut) == 1)
    {
        hasMutOnChromosome1 = child.genome1.containsMutations(mut);
        hasMutOnChromosome2 = child.genome2.containsMutations(mut);
        if (hasMutOnChromosome1 & !hasMutOnChromosome2)
            child.genome2.addMutations(mut);
        else if (hasMutOnChromosome2 & !hasMutOnChromosome1)
            child.genome1.addMutations(mut);
    }
    return T;
}
```

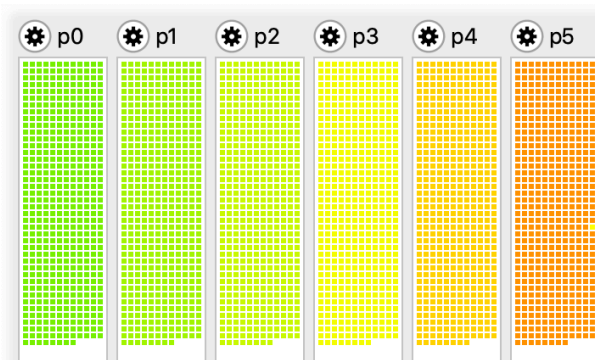
The first line just finds the introduced mutation by searching for it in the list of mutations kept by the simulation; we could have instead kept a reference to the mutation, following the strategy of section 9.9 (see section 27.2 for further discussion). The `if` statement tests whether we found the MCR mutation; in the final tick of the simulation, when the mutation has either fixed or been lost, we won't find it.

Assuming we do find the MCR mutation, we then check whether the particular child that we are working with – the target of this `modifyChild()` operation – has the MCR mutation in either of its genomes, using the `containsMutations()` method of `Genome`. With that information, the rest is simple: if it is in one genome but not the other, we add the MCR mutation to the genome that doesn't already contain it, just as the CRISPR/Cas9 gene drive machinery for MCR would do in an actual organism. If neither genome of the target contains the MCR gene, or if both genomes do, we leave the child as it is. Finally, we return `T`, indicating that the child in question should in fact be generated (a return of `F` would suppress generation; we saw an example of this in section 12.2).

If we recycle and run this model, it shows a consistent behavior of rapid fixation, even in the subpopulations where it is deleterious, and even despite having to “swim upstream” against the predominant direction of gene flow. At the moment just prior to complete fixation, the model looks something like this (note that once the mutation actually fixes, it is removed from the simulation and replaced by a substitution object, so the populations revert to yellow):



Incidentally, in snapshots like the one above, we've been using the Population Visualization graph to see the state of the model, because it is a bit more informative than the default population view, which doesn't show population connectivity:



This display shows every individual in the model, colored according to its fitness, but it lacks the connectivity arrows in the Population Visualization plot.

For reference, here is the full gene drive model with all of the components introduced above:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", -0.1); // MCR complex
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
```

```

1 early() {
    for (i in 0:5)
        sim.addSubpop(i, 500);
    for (i in 1:5)
        sim.subpopulations[i].setMigrationRates(i-1, 0.001);
    for (i in 0:4)
        sim.subpopulations[i].setMigrationRates(i+1, 0.1);
}
100 late() {
    p0.genomes[0:49].addNewDrawnMutation(m2, 10000);
}
100:10000 late() {
    if (sim.countOfMutationsOfType(m2) == 0) {
        fixed = any(sim.substitutions.mutationType == m2);
        cat(ifelse(fixed, "FIXED\n", "LOST\n"));
        sim.simulationFinished();
    }
}
mutationEffect(m2) {
    return 1.5 - subpop.id * 0.15;
}
100:10000 modifyChild() {
    mut = sim.mutationsOfType(m2);
    if (size(mut) == 1) {
        hasMutOnChromosome1 = child.genome1.containsMutations(mut);
        hasMutOnChromosome2 = child.genome2.containsMutations(mut);
        if (hasMutOnChromosome1 & !hasMutOnChromosome2)
            child.genome2.addMutations(mut);
        else if (hasMutOnChromosome2 & !hasMutOnChromosome1)
            child.genome1.addMutations(mut);
    }
    return T;
}

```

It has been proposed that MCR could have many compelling applications, such as driving particular mosquito species – those that carry important human diseases such as malaria – toward extinction, or at least driving a potentially deleterious gene for resistance to diseases like malaria toward fixation in those mosquito species. To know whether such proposals are realistic, however, we need to model them. In this section we developed a simple toy model of MCR; a model useful for real-world prediction would need many additional ramifications, of course, but this is a starting point for further exploration.

## 12.4 Suppressing hermaphroditic selfing

In section 6.3.1, it was noted that a low rate of selfing will normally be observed in SLiM in hermaphroditic models, even when the selfing rate is explicitly set to zero. This occurs because SLiM chooses each of the parents in a biparental mating randomly (weighted according to fitness), and does not explicitly prevent the same individual from being chosen as both parents. Normally this does not present a problem; it is typically a very small effect, and indeed it is sometimes desirable since the model will then better match the predictions from some simple analytical models. Sometimes, however, this selfing does prove to be an issue (particularly with small effective population sizes or high variance in fitness). In such cases, it can be prevented with a call at the beginning of the `initialize()` callback of your script:

```
initializeSLiMOptions(preventIncidentalSelfing=T);
```

Before this option was added to SLiM, preventing incidental selfing required a simple `modifyChild()` callback. **This recipe is probably mainly of historical interest**, since it has been superseded by the configuration flag shown above; but it has been retained in the cookbook as an illustration of how `modifyChild()` callbacks can be used to perform simple changes to mating behavior of this sort. The original, now-obsolete recipe used this callback:

```
modifyChild()
{
    // prevent hermaphroditic selfing
    if (parent1 == parent2)
        return F;
    return T;
}
```

This can simply be dropped in to more or less any hermaphroditic model, such as this:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
2000 late() { sim.outputFixedMutations(); }
```

It will then suppress the selfing events, by returning F (and thus suppressing the proposed child) whenever the two parents of the proposed child are the same. This could be implemented as a `mateChoice()` callback instead, by changing the weight for the already-chosen first parent to 0, but that would be much slower since a modified mating-weights vector would have to be built for each mating event. Often, as here, suppressing specific mating combinations is most easily and efficiently done with a `modifyChild()` callback instead.

There are two caveats. The first is that if you turn selfing on in your model this `modifyChild()` callback will then cause your model to hang, because SLiM will keep trying to satisfy your requested selfing rate, whereas the callback will keep preventing it from doing so. If you need to attain exactly the requested selfing rate, however, while suppressing the background selfing events generated randomly by SLiM, you can just add a check of the `isSelfing` flag provided to the `modifyChild()` callback (see section 25.5). For selfing events that are intended by SLiM to satisfy the requested selfing rate, this flag will be T; for any additional background selfing events caused by random mate choice, this flag will be F. Suppressing only the selfing events in which `isSelfing` is F ought to produce the desired effect. (You could also do a bit of math and set an adjusted selfing rate that accounts for the background selfing rate, but that is more error-prone.)

The other caveat is that if you define other `modifyChild()` callbacks as well, you might want to think about how the multiple callbacks will stack together. Typically you would want this selfing-suppression callback to occur first in the chain, and thus you would want to define it earliest in your script. See section 25.11 for discussion of this issue.

## 12.5 Tracking separate sexes in script

SLiM can model separate sexes automatically, and can even model sex chromosomes rather than autosomes, as shown in sections 6.2.1–6.2.3. Sometimes, however, it proves useful to track the sex of individuals yourself in script, rather than using SLiM's built-in facility for this. This can come up when modeling unusual mating systems, or unusual patterns of inheritance, for example.

In general, if you are trying to develop a sexual model in SLiM and you are running into limitations, consider whether tracking the sex of individuals yourself in script could provide the additional flexibility needed.

The basic technique is extremely simple:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
    p1.individuals.tag = repEach(0:1, 250); // F==0, M==1
}
modifyChild() {
    if (parent1.tag + parent2.tag != 1)
        return F;
    child.tag = rdunif(1);
    return T;
}
1: late() {
    catn("Sex ratio (M:M+F): " + mean(p1.individuals.tag));
}
2000 late() { sim.outputFixedMutations(); }
```

The `initialize()` callback is boilerplate; there is nothing to discuss there.

After creating subpopulation `p1`, we initialize `p1.individuals.tag` with values of 0 for females and 1 for males using a vectorized property assignment; the 500 values generated by `repEach()` are assigned, one-to-one, into the `tag` properties of the 500 individuals in `p1` (see the Eidos value for further discussion of vectorized property assignment). The `tag` property is a scratch space provided by SLiM that models can use for any purpose; here we will use it to store the sex of individuals. The first 250 individuals are female, the second 250 are male, as set up with `repEach()`; this particular configuration is of no consequence in this model, but it's good to start with an even sex ratio, at least. The values of 0 and 1 are purely conventional, but they make it very easy to calculate the sex ratio with the standard meaning of M:M+F, using `mean()`; we leverage this to print the sex ratio every tick in the `1: late()` event.

The heart of this recipe is the `modifyChild()` callback:

```
modifyChild() {
    if (parent1.tag + parent2.tag != 1)
        return F;
    child.tag = rdunif(1);
    return T;
}
```

This does two things. First, it enforces that one parent must be female and one must be male, by checking that the sum of their `tag` values is exactly 1; if not, it returns `F`, which rejects the proposed child and draws new parents. Second, it draws the sex of the offspring using `rdunif()` and assigns it into the child's `tag` property, and then returns `T` to accept the proposed child.

When SLiM is managing sexes for you, it always ensures that the first parent is the female and the second parent is the male. If you want to do the same, you can replace the `if()` test above with a more stringent test:

```

if (parent1.tag == 1)
  return F;
if (parent2.tag == 0)
  return F;

```

That doesn't matter in this model, but when doing more complex things it can be convenient to know which parent is which.

When run, this model prints the sex ratio in every tick, to ensure that things are going smoothly. The ratio will fluctuate around 0.5 stochastically, since the sex of every offspring is determined by `rdunif()`:

```

Sex ratio (M:M+F): 0.468
Sex ratio (M:M+F): 0.434
Sex ratio (M:M+F): 0.496
Sex ratio (M:M+F): 0.512
Sex ratio (M:M+F): 0.48
...

```

If an exact sex ratio is desired, one could simply set `p1.individuals.tag` to `repEach(0:1, 250)` in a `late()` event in every tick, as we did here for the initial subpopulation; or one could modify the logic of the `modifyChild()` callback, probably using an externally initialized counter.

This is the whole model; tracking sex in script is really quite trivial. The greater flexibility of this approach should be apparent: all of the rules governing sex are explicitly written in the script, such as (1) the initial configuration of sex in the population, (2) the rules for who can mate with whom, and (3) the rules for how the sex of the offspring is determined. Modeling a system like *Wolbachia* infection, which has various effects upon sex, would be a straightforward extension.

This section's recipe is a WF model, but to fully utilize the additional flexibility provided by tracking sex yourself in script, a nonWF model is often best; section 16.23 provides an example.



### 13. Phenotypes, fitness functions, quantitative traits, and QTLs

Thus far we have focused on models in which the fitness effect of a mutation depends upon its selection coefficient. The fitness effect might vary depending on time, or on the subpopulation in which it is found; but still, a mutation's effect on the fitness of its carrier could be expressed in the form of a selection coefficient (at a given time, in a given environment). This idea of a “selection coefficient” assumes that a locus directly determines a discrete Mendelian trait (like peas that are wrinkled or smooth, yellow or green), and that a given mutation at that locus therefore has a predictable fitness effect – beneficial, neutral, or deleterious – in a given environment.

In section 10.3.1, however, we looked at a model of epistasis that violated this simple assumption. A mutation subject to epistasis has different fitness effects depending upon the genetic background in which it is found – sometimes beneficial on one genetic background, but deleterious on a different background. Epistasis is a special case of *quantitative genetics*. In this chapter we will explore how to build quantitative genetics models in SLiM.

Quantitative genetics is concerned with phenotypic traits that are *polygenic*: governed by multiple loci, called quantitative trait loci or QTLs. Quantitative genetics partitions the effects of QTLs into *additive* and *non-additive* effects. The effects of QTLs are additive if the phenotypic effects of the QTLs can be added together to predict the final phenotype. For example, suppose SNP A increases human height by 2 cm, SNP B by 6.5 cm, and SNP C by 0.3 cm (these SNPs being located in different QTLs, in different areas of the chromosome). If these *effect sizes* are additive, an individual possessing all three SNPs would be expected to be 8.8 cm taller than an individual possessing none of them. We will focus upon additive effects here, for simplicity, but non-additive effects can also be modeled in SLiM (as, indeed, we saw in section 10.3.1).

Continuous-valued phenotypic traits like height, lifespan, fertility, intelligence, dispersal distance, and elevational preference are likely to be quantitative traits – they are probably governed by many QTLs of relatively small effect size, rather than just one Mendelian locus. Such phenotypes may also be influenced by other factors, such as the environment (through phenotypic plasticity), random chance (often called “environmental variance” or “developmental noise”), maternal effects, and so forth; the degree to which they are determined by the additive effects of QTLs is, roughly speaking, the heritability of the trait. (A good textbook on quantitative genetics would, of course, clarify many of these concepts, which are being defined rather loosely here.) Heritability and non-genetic effects can also be modeled in SLiM, as we will see in later sections.

In short, then: in this chapter we will model quantitative phenotypic traits that are governed by multiple QTLs. Selection will act upon the phenotype, not upon the individual QTLs; we will therefore deal not with selection coefficients, but with QTL effect sizes. These effect sizes will additively determine the *phenotypic trait value*, expressed as a number (like height, expressed in cm). The fitness of a particular phenotypic trait value will then be determined by a *fitness function* that expresses the relationship between phenotype and fitness. A particular QTL mutation might be beneficial against one genetic background (if, added together with all other QTL effect sizes, it produces a fitter phenotype), but deleterious against a different genetic background (if, added together with all other QTL effect sizes, it produces a less-fit phenotype). This is epistasis writ large; but as long as we focus on additive effects the models will remain quite simple.

The recipes we will look at in this chapter will use a variety of strategies to implement quantitative traits, but a common element across all of them is that the effect size of each QTL mutation will need to be kept somewhere; each QTL mutation will have a particular additive effect size that will be determined when the mutation is created, and will be fixed thereafter. Since these QTL mutations have no meaningful selection coefficient, we will often re-purpose the `selectionCoeff` property of `Mutation` as a convenient place to store the effect size, as we will first see in section 13.2. However, there is nothing magical about using the `selectionCoeff` property

for this purpose, and the recipe of section 13.5 will store mutation effect sizes using the `setValue()` / `getValue()` mechanism of `Dictionary` instead (see the Eidos manual for details).

Another common element is that the fitness function will need to be implemented in script, and will need to influence the fitness of individuals somehow. There are two good ways to do this in SLiM, neither of which we have seen before now (since fitness has generally been based upon selection coefficients): `fitnessEffect()` callbacks, and the `fitnessScaling` property.

A `fitnessEffect()` callback is not specific to a particular mutation type like a normal `mutationEffect()` callback (see chapter 10), and it is not called on a per-mutation basis. Instead, `fitnessEffect()` callbacks are called *once per individual* per cycle, and can provide a per-individual fitness effect that gets multiplied together with all of the other fitness effects in the model to produce the fitness of an individual (see section 25.3). In section 13.1, we will use a `fitnessEffect()` callback to implement the fitness function for a simple polygenic trait.

The `fitnessScaling` property is actually quite similar, but it is a property of `Individual` rather than being a callback. This property has a default value of `1.0`, and it is reset to `1.0` in every cycle, but script can set it to whatever fitness effect is desired for an individual. Its value will be multiplied together with all of the other fitness effects in the model, just like the result of a `fitnessEffect()` callback, to produce the fitness of an individual. Using `fitnessScaling` can be faster and more convenient than a `fitnessEffect()` callback because it can be used in a vectorized fashion, allowing fitness effects to be determined across whole subpopulations with a single vectorized assignment. We will use `fitnessScaling` in most of the recipes in this chapter.

Chapter 15 provides QTL-based continuous-space models, and sections 16.7 and 16.9 provide QTL-based nonWF models, building upon the conceptual foundation established here.

### 13.1 Polygenic selection

Our first quantitative trait model will be a model of polygenic selection in which QTLs of fixed effect size produce a phenotypic trait value that depends simply upon the total number of QTL mutations that exist in an individual's genomes. Since QTLs need to continue to affect phenotype even after they fix, we will need to use the `convertToSubstitution` property of `MutationType` to suppress the substitution of the QTL mutations, as in the epistasis example of section 10.3.1. Here is the complete model, which is quite short:

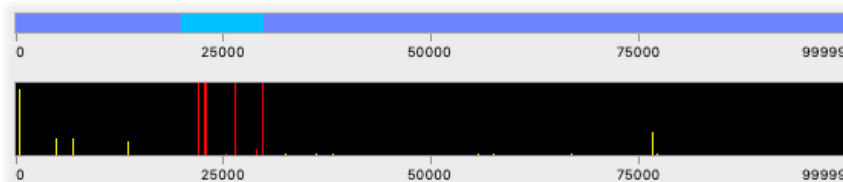
```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.0); // QTLs
  m2.convertToSubstitution = F;
  m2.color = "red";
  initializeGenomicElementType("g1", m1, 1);
  initializeGenomicElementType("g2", m2, 1);
  initializeGenomicElement(g1, 0, 20000);
  initializeGenomicElement(g2, 20001, 30000);
  initializeGenomicElement(g1, 30001, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
fitnessEffect() {
  phenotype = sum(individual.genomes.countOfMutationsOfType(m2));
  return 1.5 - (phenotype - 10.0)^2 * 0.005;
}
5000 late() {
  print(sim.mutationFrequencies(NULL, sim.mutationsOfType(m2)));
}
```

The `initialize()` callback here sets up a chromosome with a genomic element of type `g2` in a central locus, generating QTL mutations of mutation type `m2` that are under polygenic selection; the rest of the chromosome uses `g1` and `m1`, generating neutral mutations. We set `m2` mutations not to convert to substitutions when it fixes, and to display in red in SLiMgui.

The `fitnessEffect()` callback sets up the conditions for polygenic selection. It is called once per individual per cycle, and implements the fitness function that converts individual phenotypic trait values into fitness effects. Since `m1` and `m2` mutations are intrinsically neutral in this model (i.e., have a fixed selection coefficient of `0.0`), the `fitnessEffect()` callback is the sole determinant of individual fitness here. It implements a quadratic fitness function, with fitness being highest when exactly ten `m2` mutations are possessed, falling away for counts of `m2` mutations both lower and higher than that optimum. With this particular fitness function the initial fitness, with no `m2` mutations, is `1.0`; at the optimum, with ten `m2` mutations, it is `1.5`.

Let's look more closely at how this `fitnessEffect()` callback works. First it counts the total number of mutations of type `m2` on both chromosomes of the focal individual, using the `countOfMutationsOfType()` method we saw in the previous section, and assigns that value to the variable `phenotype`. To do this, it first gets `individual.genomes`, which is a vector containing both of the focal individual's genomes, then calls `countOfMutationsOfType()` on that vector to produce a result vector with the counts for each of the two genomes, and then uses `sum()` to add the two values together to get the phenotypic trait value, `phenotype`. (As with `mutationEffect()` callbacks, `individual` is a pseudo-parameter to the callback, automatically set up by SLiM to refer to the focal individual for which the callback is being called.) Then the callback does a little arithmetic to compute a fitness effect from `phenotype`, and returns the result. SLiM will multiply this together with any other fitness effects in the model to produce the fitness of the focal individual. The fitness function itself is a simple quadratic function; the distance of `phenotype` from the phenotypic optimum, `10.0`, is squared and scaled, and then subtracted from `1.5`. The result is that an individual at the phenotypic optimum has a scaled squared deviation of `0.0` and thus a fitness of `1.5`, whereas an individual with a phenotype of `0.0` has a scaled squared deviation of `0.5` and thus a fitness of `1.0`.

Running this recipe in SLiMgui shows the expected dynamics: within the QTL (the genomic element where `m2` mutations are generated), five `m2` mutations arise and drive to fixation fairly rapidly, and the model then stabilizes at equilibrium. Many new `m2` mutations continue to flicker in and out of existence in that locus, but once five have fixed (producing a phenotype value of `10.0` for homozygous individuals), further `m2` mutations will produce a lower fitness value, and will therefore be very unlikely to fix. The final state:



The `50000 late()` event produces a line of output at the end of the model run. First it fetches a vector of the `m2` mutations present in the model (including any fixed mutations, since `m2.convertToSubstitution` is `F`). Then it calls the `Species` method `mutationFrequencies()` to get the frequency of each mutation as a new `float` vector, which it prints. (The `NULL` value passed to `mutationFrequencies()` indicates that population-wide frequencies are desired, rather than frequencies within a particular subpopulation.) The result, for the run shown above, is:

```
1 0.08 1 1 1 1 0.001
```

This shows that five m2 mutations have reached fixation (a frequency of 1), producing the optimum phenotype value of 10, while two other m2 mutations are segregating at low frequencies.

It is important to understand why this model used a `fitnessEffect()` callback instead of a `mutationEffect(m2)` callback. In essence, a `mutationEffect(m2)` callback would tell SLiM the fitness effect of a particular focal m2 mutation – but that is a bit complicated, in this model. As explained at the beginning of this chapter, QTL mutations don't have a well-defined selection coefficient or fitness effect; their fitness effect is only well-defined when they are considered in aggregate, because their relationship to fitness is fundamentally epistatic. It would be possible, of course, for each m2 mutation to calculate a *de facto* fitness effect, by counting the number of other m2 mutations present in the focal individual's genomes, calculating the resulting overall fitness effect, and then allocating a proportional share of that overall fitness effect back to itself; but that would be both slow and roundabout. Conceptually, the fitness of each individual depends, with a simple formula, upon the number of m2 mutations it possesses, and so it makes sense to implement the model that way.

### 13.2 A simple model of variable QTL effect sizes

The previous section's recipe was based upon the simplifying assumption of a fixed effect size for all of the QTL mutations. In this model we remove that assumption: different QTL mutations will now have different effect sizes, drawn from a normal distribution. These effect sizes will be kept, for convenience, in the `selectionCoeff` property of the QTL mutations, but they will not be used as selection coefficients. We will also abandon the `fitnessEffect()` callback that we used in section 13.1, and instead use the `fitnessScaling` property of `Individual` to impose fitness effects upon the individuals in the model, as discussed in the chapter introduction. The full model:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 0.5); // QTLs
  m2.convertToSubstitution = F;
  initializeGenomicElementType("g1", m1, 1);
  initializeGenomicElementType("g2", m2, 1);
  initializeGenomicElement(g1, 0, 20000);
  initializeGenomicElement(g2, 20001, 30000);
  initializeGenomicElement(g1, 30001, 99999);
  initializeRecombinationRate(1e-8);
}
mutationEffect(m2) { return 1.0; }
1 early() { sim.addSubpop("p1", 500); }
1: late() {
  inds = sim.subpopulations.individuals;
  phenotypes = inds.sumOfMutationsOfType(m2);
  inds.fitnessScaling = 1.5 - (phenotypes - 10.0)^2 * 0.005;

  if (sim.cycle % 100 == 0)
    catn(sim.cycle + ": Mean phenotype == " + mean(phenotypes));
}
5000 late() {
  m2mut = sim.mutationsOfType(m2);
  freqs = sim.mutationFrequencies(NULL, m2mut);
  effects = m2mut.selectionCoeff;
  catn();
  print(cbind(freqs, effects));
}
```

Many aspects of this model, such as the genetic structure and the fitness function, have been kept the same as in the previous section so that the aspects of the design that have changed can be seen more clearly. The `initialize()` callback is the same except that mutation type `m2` is now configured to draw its selection coefficients from a normal distribution with mean `0.0` and standard deviation `0.5` (and `m2` is no longer colored red).

SLiM will not use these selection coefficient values; `m2` mutations will be neutral as far as SLiM is concerned, because of the new `mutationEffect(m2)` callback we defined above:

```
mutationEffect(m2) { return 1.0; }
```

This tells SLiM, in a nutshell, “pay no attention to the selection coefficients for `m2`; `m2` mutations are always neutral, period.” So now the `selectionCoeff` property on the `m2` mutations is ours to play with, but it is conveniently pre-filled by SLiM with values drawn from the normal distribution we specified in `initializeMutationType()`. (Sneaky, eh?) We will use these values as effect sizes when we calculate phenotypic trait values in the 1: `late()` event we defined above. Let’s review that event’s code:

```
1: late() {
    inds = sim.subpopulations.individuals;
    phenotypes = inds.sumOfMutationsOfType(m2);
    inds.fitnessScaling = 1.5 - (phenotypes - 10.0)^2 * 0.005;

    if (sim.cycle % 100 == 0)
        catn(sim.cycle + ": Mean phenotype == " + mean(phenotypes));
}
```

We set up `inds` to be a vector of all the individuals. Next, `inds.sumOfMutationsOfType(m2)` calculates, for each individual in `inds`, the sum of the selection coefficients of all `m2` mutations in the individual’s genomes. This method is tailor-made for QTL-based models where selection coefficients are actually effect sizes; the sum of the selection coefficients is then the phenotypic trait value (or more precisely, the additive component of it; we are ignoring non-additive effects for now). So this line computes the phenotypic trait values of all of the individuals in the model, and stores that vector in `phenotypes`. The next line calculates the fitness effects for those phenotypes, again in a single vectorized calculation. It is the same fitness function equation as in section 13.1, but here `phenotypes` is a vector, and so the result is a vector. That result is assigned back into `inds.fitnessScaling`; this is a vectorized property assignment, in which each element of the result is assigned into the `fitnessScaling` property of the corresponding element of `inds`. This line, then, calculates the fitness effect of each individual’s phenotype and puts the result into that individual’s `fitnessScaling` property. (See the Eidos manual for further discussion of vectorized method calls, calculations, and assignments.) These `fitnessScaling` property values will have a multiplicative effect upon the fitness of the individuals, as desired, similarly to how the `fitnessEffect()` callbacks influenced individual fitness in section 13.1’s model.

The last two lines of the event just produce output; every 100 ticks the model prints a summary line that gives the cycle number and the mean phenotypic trait value, such as:

```
100: Mean phenotype == 0.0533152
200: Mean phenotype == 1.05143
300: Mean phenotype == 1.59688
...
4800: Mean phenotype == 10.0042
4900: Mean phenotype == 10.014
5000: Mean phenotype == 9.94105
```



The phenotypic optimum at 10.0 is reached by tick 3100 or so, in fact; after that the model just wobbles around on the peak of the fitness function. At the end of the model run, the 5000 `late()` event produces some final output, in the form of a table of frequencies and effect sizes:

	[,0]	[,1]
[0,]	0.001	0.767606
[1,]	1	0.477175
[2,]	1	0.906239
[3,]	0.034	-0.0489351
[4,]	1	0.884906
[5,]	1	0.433817
[6,]	0.359	-0.1595
[7,]	1	0.348028
[8,]	0.004	0.440343
[9,]	1	0.165444
[10,]	1	0.561426
[11,]	1	0.231208
[12,]	0.002	0.0837105
[13,]	0.007	0.787386
[14,]	1	0.472853
[15,]	1	0.541729
[16,]	0.022	0.524584
[17,]	0.359	-0.0856977
[18,]	0.115	-0.531236
[19,]	0.006	0.300258
[20,]	0.006	-0.0756474
[21,]	0.039	-0.167648
[22,]	0.359	0.238666
[23,]	0.045	-0.00518997
[24,]	0.017	-0.433331
[25,]	0.068	0.0859256

This is a two-column matrix, constructed by `cbind()`; we haven't used matrices much in this manual, but they are supported by Eidos and work similarly to how they work in R (see the Eidos manual for details). Here it is just a convenient way to produce the two-column output we want, with frequencies of `m2` mutations in the left column and effect sizes in the right column. Ten of the `m2` mutations have a frequency of 1 (i.e., have fixed). The sum of the effect sizes of these ten mutations is 5.022825; when present in two copies (since these are diploid individuals), the additive phenotypic effect of these mutations is 10.04565. These mutations, then, get the population almost exactly to the fitness peak. The remaining mutations are mostly segregating at low frequencies; probably most of them will not fix, since most of them would take population further away from the fitness peak, but a few of them are either slightly beneficial (on the background of fixed mutations) or are only very slightly deleterious; those mutations have chances.

This recipe has two main points. One is to introduce the idea of keeping QTL effect sizes in the `selectionCoeff` property of the mutations, while rendering those mutations neutral despite their non-zero selection coefficients by writing a `mutationEffect()` callback that declares them neutral. There are other places we could keep these effect size values, but using `selectionCoeff` for this purpose is immensely convenient. The other main point here is to introduce the use of `fitnessScaling` as a way to modify the fitness of individuals. The `fitnessScaling` property can be used for all sorts of purposes besides implementing QTL models, as we will see in later chapters; but it is a very good way to handle QTL-based fitness effects because it enables extremely efficient vectorized fitness calculations, as we saw in this recipe.



### 13.3 A model of discrete QTL effects with a structured chromosome

This recipe is an extension of the recipe in section 13.2. Like that recipe, it models a phenotypic trait based upon multiple loci with additive effects. However, rather than just having a genomic element within which new QTL mutations can arise at random, this recipe will model exactly 10 QTLs, each spanning a single base position, and each of which can have an effect size of either  $-1$  or  $+1$ . Such a design, with a fixed number of loci of discrete effect sizes, is common for models of quantitative traits, so it is worthwhile to explore how to simulate them in SLiM.

There are several other extensions in this recipe, compared to section 13.2; we will discuss them as they arise. For now, we will start with the initialization portion of the recipe:

```
initialize() {
    initializeMutationRate(1e-6);

    // neutral mutations in non-coding regions
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);

    // mutations representing alleles in QTLs
    scriptForQTLs = "if (runif(1) < 0.5) -1; else 1;";
    initializeMutationType("m2", 0.5, "s", scriptForQTLs);
    initializeGenomicElementType("g2", m2, 1.0);
    m2.convertToSubstitution = F;
    m2.mutationStackPolicy = "L";

    // set up our chromosome: 10 QTLs, surrounded by neutral regions
    defineConstant("C", 10); // number of QTLs
    defineConstant("W", 1000); // size of neutral buffer on each side
    pos = 0;
    q = NULL;

    for (i in 1:C)
    {
        initializeGenomicElement(g1, pos, pos + W-1);
        pos = pos + W;

        initializeGenomicElement(g2, pos, pos);
        q = c(q, pos);
        pos = pos + 1;

        initializeGenomicElement(g1, pos, pos + W-1);
        pos = pos + W;
    }

    defineConstant("Q", q); // remember our QTL positions

    // we want the QTLs to be unlinked; build a recombination map for that
    rates = c(rep(c(1e-8, 0.5), C-1), 1e-8);
    ends = (repEach(Q + W, 2) + rep(c(0,1), C))[0:(C*2 - 2)];
    initializeRecombinationRate(rates, ends);
}
```

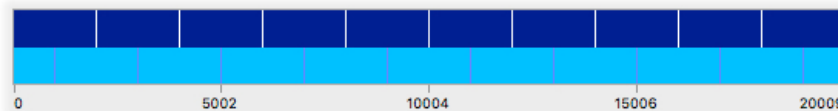
This sets up two mutation types: `m1`, representing neutral mutations, and `m2`, representing mutations at QTLs. The `m2` mutation type draws its mutational effects from a user-specified distribution, rather than from one of SLiM's built-in mutational distributions. It does this by specifying the mutation type as "s", for "script", and then supplying a short Eidos snippet as a string parameter. That snippet is run as a lambda (see section 20.5) by SLiM whenever it needs to

draw a new selection coefficient. The m2 mutation type is also set not to be removed when it fixes, because QTLs should continue to influence phenotype even after fixation (see section 1.5.2). Finally, it is set to use a “last mutation” stacking policy, so that when a new mutation occurs at a given QTL it replaces the allele that was previously at that site, rather than “stacking” with it as is SLiM’s default behavior (see section 1.5.3).

Two genomic element types are also set up by this code: g1, representing neutral buffer zones around the QTLs, and g2, representing the QTLs themselves. The rest of this initialize() callback sets up the chromosome by tiling these two genomic element types. The genetic structure used here simulates ten separate chromosomes, with one QTL on each, using a recombination rate of 0.5 between specific pairs of bases to subdivide the chromosome object into separate “effective chromosomes” that have no linkage between them (see section 6.1.4, which introduced this technique). This model places each QTL at the center of its effective chromosome, with 1000 bases on each side experiencing neutral mutations, but those choices are easily changed.

This model makes somewhat liberal use of the defineConstant() function of Eidos to set up constants related to the genomic structure of the simulation. Defined constants are much like variables, except that they cannot be redefined (except by removing them with rm() first), and they persist for the lifetime of the simulation rather than disappearing at the end of the callback in which they are defined. They are thus very useful for symbolically representing model parameters; this model also defines a constant Q to remember the positions of all of the QTLs being simulated.

After this initialization() callback has run, the genetic structure of the simulation looks like this in SLiMgui (with display of genomic elements and recombination rates enabled):



This shows the pattern of non-coding regions and QTLs (bottom), and the recombination breakpoints that define the breaks between effective chromosomes in the model (top).

Next we will set up the initial population state:

```
1 late() {
  sim.addSubpop("p1", 500);
  sim.addSubpop("p2", 500);

  // set up migration; comment these out for zero gene flow
  p1.setMigrationRates(p2, 0.01);
  p2.setMigrationRates(p1, 0.01);

  // replicate the s1 event to output in tick 2 also
  community.registerEarlyEvent("s2", s1.source, 2, 2);

  // optional: give m2 mutations to everyone, as standing variation
  g = sim.subpopulations.genomes;

  for (q in Q)
  {
    isPlus = asLogical(rbinom(size(g), 1, 0.5));
    g[isPlus].addNewMutation(m2, 1.0, q);
    g[!isPlus].addNewMutation(m2, -1.0, q);
  }
}
```

This model incorporates a two-subpopulation structure, with limited gene flow between the subpopulations. The two subpopulations will experience different environments that are selecting for different optima; unlike the “spatially varying selection” model of section 10.2, however, the two environments are selecting for different phenotypes, as determined by the additive effects of the underlying QTLs, rather than just exerting differential selection on each individual locus. That differential selection will be implemented below; for now we just set up the subpopulations and migration rates. This event also contains a call to `registerEarlyEvent()` that will not run right now; if you want to run the model at this stage you will need to comment that out. It will be explained below.

The latter half of the event is optional; it sets up the initial state of the QTLs in the model by placing initial mutations in them. It does this by looping over all of the QTL positions (as stored in `Q`). For each QTL, it decides whether each genome in the simulation will start with a + or a – allele, by drawing from a binomial distribution. It then distributes the + and – alleles to all of the genomes by calling `addNewMutation()` to add a + or – allele to each genome. For a given QTL, this code results in just two mutation objects being created, by calling `addNewMutation()` just once for the + allele and once for the – allele (rather than once per genome), using vectorized method calls; section 10.4.3 discusses this distinction further. Here this is a non-essential detail, but it improves the efficiency of the model since fewer mutation objects need to be tracked by SLiM. This whole initialization loop can be removed, in which case the model starts with no QTL mutations, producing an initial value of zero for each QTL until new mutations arise. That would be a model of emerging genetic diversity from a clonal population, rather than this model of selection beginning with a high degree of random standing genetic variation.

Next comes some script to implement the quantitative trait machinery:

```
mutationEffect(m2) { return 1.0; }
1: late() {
    // evaluate and save the additive effects of QTLs
    for (subpop in c(p1,p2))
    {
        inds = subpop.individuals;
        phenotype = inds.sumOfMutationsOfType(m2);
        optimum = (subpop == p1 ? 10.0 else -10.0);
        inds.fitnessScaling = 1.0 + dnorm(optimum - phenotype, 0.0, 5.0);
        inds.tagF = phenotype;
    }
}
```

The single-line `mutationEffect()` callback just renders `m2` mutations neutral, since we are using their `selectionCoeff` property to hold an additive effect size instead of a selection coefficient (see section 13.2, which introduced this technique).

The `1: late()` callback is more interesting. As in section 13.2, this event calculates the phenotype of every individual using `sumOfMutationsOfType(m2)`, but here we do it one subpopulation at a time. That way, in the next line, we can select a different phenotypic optimum value for `p1` versus `p2`. We use the “ternary conditional” operator of Eidos, `?else`, which is essentially a compact `if–else` statement within a single expression; in short, `optimum` is set to `10.0` if we are presently handling `p1` in the loop, or `-10.0` if we’re on `p2`. The next line, as in section 13.2, encapsulates the fitness function; here we have switched to a vectorized call to `dnorm()`, providing us with a Gaussian fitness function instead of a quadratic. This change is of no great importance; Gaussian fitness functions are more common in the literature, but the quadratic fitness function was perhaps conceptually simpler. Note that a baseline of `1.0` is added to the Gaussian value; this decreases the fitness difference between the most-fit and least-fit individuals.

Finally, we save the phenotypes of all the individuals in their `tagF` property for future reference, since we will want to use them for various purposes later on. The `tagF` property is very similar to the `tag` property that we have seen in previous models, except that it stores a `float` value instead of an integer. One could also use the `getValue()` and `setValue()` methods of `Individual` to keep this phenotype state (provided by Eidos class `Dictionary`; see the Eidos manual), rather than `tagF`; that would, however, be substantially slower than using `tagF`.

There is one more element we would like this model to contain: assortative mating. Unlike the model of section 11.1, where mating was assortative based upon possession of a single introduced mutation, here we want mating to be assortative based upon the phenotype produced by the additive effects of the underlying QTLs; in other words, we want individuals to prefer to mate with other phenotypically similar individuals, regardless of their underlying genetics. This means that mating can actually be disassortative at the genetic level, because individuals with the same phenotype might achieve that phenotype through entirely different QTL alleles. We implement this with a simple `mateChoice()` callback (see chapter 11):

```
mateChoice() {
  phenotype = individual.tagF;
  others = sourceSubpop.individuals.tagF;
  return weights * dnorm(others, phenotype, 5.0);
}
```

This looks up the phenotype of the focal individual and all other individuals from the `tagF` property where it was stored, and uses `dnorm()` to implement a Gaussian mating preference function. This mating preference is combined multiplicatively with SLiM's default fitness-based mating weight vector, `weights`, to produce final mating weights. This means that individuals with a low fitness (due to being maladapted to their environment) will be less likely to be chosen as mates, but that individuals phenotypically similar to the focal individual that is choosing a mate will be preferred. Any particular relative weighting of these different priorities could be expressed by modifying the fitness function and/or the mating weight function in the script; every aspect of fitness evaluation and mate choice is under complete script control here.

Incidentally, it might be of interest to note that the quantitative trait is now something like a magic trait, since it is under divergent ecological selection and also influences mate choice. However, according to the technical definition (Servedio et al. 2011) it is not actually a magic trait, I think, since it is based upon multiple loci. The individual QTLs might be considered “magic genes” since they perhaps satisfy the criteria of the formal definition, but since there is epistasis between them (because of the way they combine additively to determine the quantitative trait that is actually under selection, as discussed at the beginning of this chapter), it is not entirely clear how they relate to the usual intuitive meaning of “magic trait”.

In any case, we now end the model with some custom output by tallying up fitnesses, phenotypes, and frequencies:

```
s1 2001 early() {
  cat("-----\n");
  cat("Output for end of cycle " + (sim.cycle - 1) + ":\n\n");

  // Output population fitness values
  cat("p1 mean fitness = " + mean(p1.cachedFitness(NULL)) + "\n");
  cat("p2 mean fitness = " + mean(p2.cachedFitness(NULL)) + "\n");

  // Output population additive QTL-based phenotypes
  cat("p1 mean phenotype = " + mean(p1.individuals.tagF) + "\n");
  cat("p2 mean phenotype = " + mean(p2.individuals.tagF) + "\n");
}
```

```

// Output frequencies of +1/-1 alleles at the QTLs
muts = sim.mutationsOfType(m2);
plus = muts[muts.selectionCoeff == 1.0];
minus = muts[muts.selectionCoeff == -1.0];

cat("\nOverall frequencies:\n\n");
for (q in Q)
{
  qPlus = plus[plus.position == q];
  qMinus = minus[minus.position == q];
  pf = sum(sim.mutationFrequencies(NULL, qPlus));
  mf = sum(sim.mutationFrequencies(NULL, qMinus));
  pf1 = sum(sim.mutationFrequencies(p1, qPlus));
  mf1 = sum(sim.mutationFrequencies(p1, qMinus));
  pf2 = sum(sim.mutationFrequencies(p2, qPlus));
  mf2 = sum(sim.mutationFrequencies(p2, qMinus));

  cat("    QTL " + q + ": f(+) == " + pf + ", f(-) == " + mf + "\n");
  cat("        in p1: f(+) == " + pf1 + ", f(-) == " + mf1 + "\n");
  cat("        in p2: f(+) == " + pf2 + ", f(-) == " + mf2 + "\n\n");
}
}

```

This relatively complex output code assesses and prints information about the genetic structure and fitness of each subpopulation at the beginning and end of the simulation; this is a good illustration of how SLiM's output can be customized in Eidos. This code runs at the beginning of tick 2001, so as to produce output regarding the very end of tick 2000, after fitness values have been calculated; calling `cachedFitness()` in a `late()` event in tick 2000 would raise an error, since fitness values would not yet be available.

Note that the purpose is now clear of the line we saw before:

```
community.registerEarlyEvent("s2", s1.source, 2, 2);
```

The output callback is named as script block `s1`; that line of code replicates it as an `early()` event in tick 2, so that it produces output at both the beginning and the end of the run. If SLiM's syntax for specifying the ticks in which a callback runs were more general, this trick would not be needed; at present, however, this is a simple way to achieve what we want.

Running this model produces a bunch of output. Let's look first at this block of output produced at the beginning of the run:

Output for end of cycle 1:

```

p1 mean fitness = 1.02079
p2 mean fitness = 1.02007
p1 mean phenotype = 0.244
p2 mean phenotype = -0.2

```

and this corresponding block produced at the end of the run:

Output for end of cycle 2000:

```

p1 mean fitness = 1.06187
p2 mean fitness = 1.06732
p1 mean phenotype = 8.16
p2 mean phenotype = -9.936

```

The initial output shows us that the individuals in p1 and p2 began fairly maladapted, with a mean phenotype near 0.0 and a fitness just over 1.0. They ended quite a bit better adapted; p2 is almost smack on the optimum of -10.0, and p1 is much closer to its optimum of 10.0 than it was. The resulting fitness values are also higher, at approximately 1.06; the difference is not large, but apparently it is enough, in combination with assortative mating, to drive this local adaptation.

There is also output about “Overall frequencies”; for the initial setup, it generally looks something like this:

```
QTL 1000: f(+) == 0.489, f(-) == 0.511
         in p1: f(+) == 0.508, f(-) == 0.492
         in p2: f(+) == 0.47, f(-) == 0.53
```

This shows that the QTL at position 1000 started at a frequency of close to 0.5, both in the population as a whole and in p1 and p2 individually. The other QTLs start in a similar state, thanks to the random initialization they receive in this model. This provides lots of standing genetic variation at each QTL, but one could use a different initial state if desired.

The situation at the end of the run looks quite different, with some QTL states like this:

```
QTL 1000: f(+) == 0.463, f(-) == 0.537
         in p1: f(+) == 0.84, f(-) == 0.16
         in p2: f(+) == 0.086, f(-) == 0.914
```

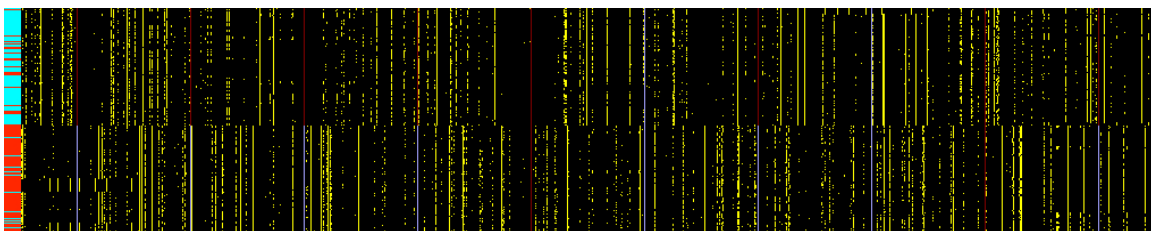
This QTL has clearly been selected toward the + allele in p1 and toward the - allele in p2, and is therefore contributing to the local adaptation achieved by the population.

On the other hand, some other QTLs look quite different:

```
QTL 3001: f(+) == 0, f(-) == 1
         in p1: f(+) == 0, f(-) == 1
         in p2: f(+) == 0, f(-) == 1
```

This QTL has fixed for the - state in both populations. The particular “solution” found by the model, with some QTLs fixing and some divergent, might be of interest; of even more interest, perhaps, would be the degree of repeatability of these results across many runs, and whether that is driven by the particular QTL effect sizes allowed, and so forth. The solution space might be constrained by the particular genetic configuration we modeled here; it may not be possible for both subpopulations to reach their optima, for example, because of the constraints imposed by having exactly ten QTLs with only simple +/- effects.

In SLiMgui, the command Create Haplotype Plot, accessible from the Simulation menu or the Show Graph pop-up menu button, provides an interesting view of the final state for such a run:



The population has diverged into two very different overall haplotype states, not only at the QTLs (the thin red and blue vertical lines), but also at neutral sites across the genome (the yellow vertical lines). The color strip on the left edge shows the population to which each individual belongs; the two overall haplotype states clearly correspond to the two subpopulations, although there is a small community of migrants and descendants of migrants that have carried each



subpopulation's haplotypes over to the opposite subpopulation. They will tend to mate only among themselves, which apparently prevents the migration from resulting in very much actual gene flow. (See section 14.4 for a more extensive discussion of haplotype plots.)

Following up on that observation, assortative mating does appear to be helping divergence considerably; a sample run with the `mateChoice()` callback commented out results in considerably less divergence:

```
p1 mean fitness = 1.0394
p2 mean fitness = 1.04998
p1 mean phenotype = 3.796
p2 mean phenotype = -5.624
```

This is true even if the model is run for much longer; indeed, without assortative mating, long runs of this model can result in all ten QTLs fixing, allowing one population to become fixed near its optimum while the other population has been dragged to a highly maladapted position. The model may show various behaviors depending upon the strength of assortative mating and other details of the formula with which the final mating weights are calculated.

This model includes neutral diversity at loci surrounding each QTL; no attempt has been made to analyze that here, apart from the superficial haplotype analysis above, but presumably there might be interesting patterns there related to gene flow, hitchhiking, and so forth. Increasing the length of the neutral buffers around the QTLs (with the defined constant `W`), and/or increasing the recombination rate, would probably be helpful for making these patterns more clear.

### 13.4 A quantitative genetics model with heritability

In previous sections in this chapter we have seen recipes for quantitative genetics models in which phenotype was calculated as the additive effect of a set of QTLs, and fitness was determined by the phenotypic trait value of individuals compared to an environmental phenotypic optimum. In this section we will extend this type of quantitative genetics model to incorporate heritability, simulated by adding random deviations (representing “environmental variance” or “developmental noise” or similar non-heritable effects) to the additive genetic variation of the QTLs. This model is based much more upon section 13.2 than 13.3; it utilizes QTLs with effect sizes drawn from a Gaussian distribution, rather than the discrete effect size distribution of section 13.3, and it allows new QTL mutations to arise spontaneously at any location. Also, like section 13.2, this recipe has only a single subpopulation, and does not contain assortative mating; we are keeping things simple here.

The mechanics of this model are quite simple. Here is the entire model except the final output event and the implementation of heritability:

```
initialize() {
    initializeMutationRate(1e-6);

    initializeMutationType("m1", 0.5, "f", 0.0);    // neutral
    initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
    m2.convertToSubstitution = F;

    initializeGenomicElementType("g1", c(m1, m2), c(1, 0.01));
    initializeGenomicElement(g1, 0, 1e5 - 1);
    initializeRecombinationRate(1e-8);
}
1 late() {
    sim.addSubpop("p1", 1000);
}
```

```

1: late() {
    // evaluate QTLs to get phenotypes and fitness
    inds = sim.subpopulations.individuals;
    phenotypes = inds.sumOfMutationsOfType(m2);
    inds.fitnessScaling = 1.0 + dnorm(10.0 - phenotypes, 0.0, 5.0);
    inds.tagF = phenotypes;
}
mutationEffect(m2) {
    return 1.0; // QTLs are neutral; fitness effects are handled below
}

```

We have seen this sort of QTL-based mechanics before. In brief, QTLs are here represented by mutation type `m2`, and the selection coefficient of `m2` mutations is used as their additive effect size. A `mutationEffect(m2)` callback makes all `m2` mutations be neutral regardless of these selection coefficients. A `late()` event computes phenotypes using `sumOfMutationsOfType()`, and then computes fitness values using a Gaussian fitness function based upon the deviation of phenotypes of the optimum. The computed phenotypes are also stored in the `tagF` property for later use.

Note that, as in section 13.3, a baseline value of `1.0` is added to the Gaussian function value returned by `dnorm()`. Section 13.3 didn't really discuss that choice in any detail, however; let's embark on that digression now. Fundamentally, the problem is that of trying to choose a function that provides a reasonable phenotype-to-fitness map. There is not a lot of empirical data to guide this choice in most systems, so it is a difficult and somewhat arbitrary decision. In a hard selection model, where low mean population fitness results in a decrease in population size, an appropriate fitness function might have a minimum value of zero, allowing the modeling of phenomena such as lethal mutations and population extinction. In such a model, if most individuals have a fitness of `0.001` and one has a fitness of `0.1` (having just received a beneficial mutation, for example), most of the individuals will produce no offspring at all, while the individual with fitness of `0.1` might produce just one or two, leading to a very small population in the next generation. This seems quite reasonable. The same situation in a soft selection model, where population size does not depend upon mean population fitness, makes considerably less sense, however. In this scenario, the child generation must be filled with the requisite number of individuals, so the individual with fitness `0.1` will end up generating most of the offspring – perhaps hundreds or even thousands of offspring – while all the rest of the individuals survive but generate few or no offspring. For some organisms this might be realistic, but for many it would be nonsensical – no matter how much more fit one elephant is than all of the other elephants in Africa, it is not going to generate thousands of offspring all across Africa all by itself! In short, fitness in soft selection models works differently than in hard selection models, and is often modeled more appropriately with a fitness function that does not allow such large differences in relative fitness to exist, so that one individual does not completely dominate the reproductive output of the population. Adding a constant value to a Gaussian function is a simple way to achieve this; the larger the constant added, the more the relative fitness values in the population are effectively homogenized. Using a constant of `1.0` makes some superficial sense, since `1.0` represents neutrality and the Gaussian fitness function can then be thought of as representing some marginal increase in fitness beyond neutrality. However, this constant is actually arbitrary; after the rescaling that is implicit in the concept of relative fitness, `1.0` will not be neutral anyway. The constant added should probably therefore be considered to be a free parameter of the model, if a fitness function of this form is used.

To continue this digression a bit longer, it is also worth noting that if large differences in relative fitness are allowed in a model, as with the situation above where most individuals have fitness `0.001` and one has fitness `0.1`, this can lead to undesirable mate-choice dynamics, too. By default, SLiM models are hermaphroditic, and while they employ biparental mating by default, they do not

prevent the same parent from being chosen as *both* parents in a biparental mating event, resulting in hermaphroditic selfing. Normally this is not a problem, since for typical population sizes it is unlikely to occur often enough to make a significant difference to the model's results. In some sense it is even desirable, since this behavior means that SLiM's default configuration mirrors simple analytical population genetics models more closely; this is the reason why SLiM does not prevent it by default. However, when there is large variance in fitness or when the effective population size is very small, the chance of hermaphroditic selfing can become quite high; the fitness 0.1 individual may be quite likely to be chosen as both parents of a given offspring, and it may therefore generate most of its offspring through selfing. In that case it may be desirable to suppress it. See section 12.4 for further discussion of this issue and how to deal with it in SLiM.

QTLs are about 1% of all new mutations, and thus new QTLs arise spontaneously throughout the run. If they tend to improve the phenotypes of individuals in the population, then they tend to be retained; if not, they tend to be lost. The population as a whole begins with a mean phenotype of 0.0, since no QTLs exist. Over the model run, the population will execute an adaptive walk towards the fitness peak at +10.0. The only other component we need for the base model is an event that checks for the termination condition (arrival at the fitness peak) and prints output:

```
1:100000 late() {
  if (sim.cycle == 1)
    cat("Mean phenotype:\n");

  meanPhenotype = mean(p1.individuals.tagF);
  cat(format("%.2f", meanPhenotype));

  // Run until we reach the fitness peak
  if (abs(meanPhenotype - 10.0) > 0.1)
  {
    cat(", ");
    return;
  }

  cat("\n\n-----\n");
  cat("QTLs at cycle " + sim.cycle + ":\n\n");

  qtls = sim.mutationsOfType(m2);
  f = sim.mutationFrequencies(NULL, qtls);
  s = qtls.selectionCoeff;
  p = qtls.position;
  o = qtls.originTick;
  indices = order(f, F);

  for (i in indices)
    cat("    " + p[i] + ": s = " + s[i] + ", f == " + f[i] +
      ", o == " + o[i] + "\n");

  sim.simulationFinished();
}
```

This event runs in every tick. It prints the mean phenotype in each tick, forming a comma-separated list that can be easily copied into an environment such as R for plotting. When the fitness peak is reached, within a small tolerance, it prints a list of all of the QTLs in the population and terminates. The QTL list contains the position, effect size, frequency, and origin tick of each QTL, and is sorted by frequency using the `order()` function (see the Eidos manual for documentation on `order()`, of course).

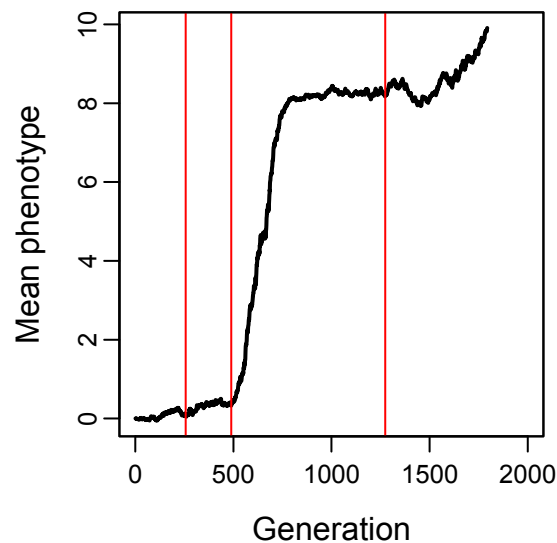
A run of the model produces output like this:

```
Mean phenotype:
0.00, -0.00, -0.00, -0.01, -0.00, -0.00, -0.01, -0.01, -0.01, -0.01, ...,
9.77, 9.77, 9.82, 9.81, 9.83, 9.83, 9.84, 9.85, 9.85, 9.82, 9.89, 9.91
```

-----  
QTLs at cycle 1793:

```
45907: s = 1.28919, f == 1, o == 489
98721: s = 2.78947, f == 1, o == 257
53961: s = 1.59969, f == 0.592, o == 1274
21414: s = -1.09245, f == 0.0515, o == 1639
93840: s = -0.338536, f == 0.0345, o == 1657
74095: s = 1.35625, f == 0.026, o == 1762
...
```

Plotting the mean phenotype time series in R produces a visualization of the adaptive walk:



The black curve shows the mean phenotype over time; the three red lines show the ticks of origin of the three high-frequency QTLs listed in the output above. This visualization shows that the adaptive walk involved a joint sweep by the first two QTLs, followed by a second sweep by the third QTL, which arose later. At the end of the run, the population has reached the adaptive peak only on average, since the third QTL is present at a frequency of only 0.592. Individuals that do not possess this QTL at all have a phenotype of ~8.16; those with only one copy have a phenotype of ~9.76; and those with two copies have a phenotype of ~11.36. In the final state of the model, heterozygote advantage thus produces balancing selection upon the third QTL, an interesting state of affairs that would presumably resolve eventually, when a QTL of a more convenient effect size arose and displaced the QTL under balancing selection by allowing the entire population to sit close to the adaptive peak.

So far so good; but earlier it was promised that this model would incorporate heritability as well. Let's add that now, beginning with the addition of a line at the top of the `initialize()` callback that sets up a constant representing the desired heritability,  $h^2$ :

```
defineConstant("h2", 0.1);
```

And then, here is a complete replacement for the 1: `late()` event that calculates phenotypes and fitness effects:

```
1: late() {  
  // sum the additive effects of QTLs  
  inds = sim.subpopulations.individuals;  
  additive = inds.sumOfMutationsOfType(m2);  
  
  // model environmental variance, according to the target heritability  
  V_A = sd(additive)^2;  
  V_E = (V_A - h2 * V_A) / h2;    // from h2 == V_A / (V_A + V_E)  
  env = rnorm(size(inds), 0.0, sqrt(V_E));  
  
  // set fitness effects and remember phenotypes  
  phenotypes = additive + env;  
  inds.fitnessScaling = 1.0 + dnorm(10.0 - phenotypes, 0.0, 5.0);  
  inds.tagF = phenotypes;  
}
```

This replacement event calculates phenotypes in much the same way as the original, but with the addition of random noise representing environmental variance. The desired variance is calculated based upon the additive genetic variance and the heritability, following standard quantitative genetics, and `rnorm()` generates random values with (approximately) that variance. Note that a built-in function exists, `calcVA()`, that calculates the additive genetic variance for a quantitative trait, but since the calculation is very simple and we already have the additive effects summed up, we don't bother to use that function here.

The model above produces somewhat different results than the original model without heritability. The most obvious difference is that the plot of mean phenotype over time is noisier, because the mean phenotype itself is noisier. Other effects of the heritability on the evolutionary trajectory are more difficult to see in a single run; statistical analysis of a large number of runs would be needed to draw firm conclusions.

We have often used the term “phenotype” here, rather than just referring to, e.g., the breeding value of the quantitative trait. This is deliberate, because this model really is a model of selection on the organism phenotype, including the effects of environmental noise; the variable above named `additive` represents the breeding value, if I have got my quantitative genetics right. Here the phenotype is determined by just a single quantitative trait, but it would be quite simple to extend the model to encompass multiple traits that all influenced the organism's phenotype in different ways. The 1: `late()` event here encapsulates both the genotype-to-phenotype map and the fitness function, and can be broadened to any such functions desired, for any number of traits.

It should be noted that this model demonstrates just one possible way of adding environmental variance to influence heritability. The method used here does not change the magnitude of the additive genetic variance, and so the overall phenotypic variance will be larger with lower heritability. Alternatively, one could scale the additive genetic variance down so that the phenotypic variance remains constant regardless of the heritability value chosen. Both methods would achieve the same target heritability value, but with different overall phenotypic variance.

In fact, trying to attain a target heritability value, as done here, is rather artificial to begin with; it is common in analytical quantitative genetics models, but from an individual-based perspective such as that of SLiM, the heritability is properly regarded as an emergent property of the model, not a value the model ought to impose. From that perspective, the better approach would be to simply add in environmental variation of a particular magnitude (determined from empirical data, perhaps). The magnitude of the environmental variance, rather than the heritability, would be the

model parameter, and the heritability would be a consequence of the model's dynamics. The code above can of course be easily modified to implement such a scheme instead.

It should also be noted that in this model the heritability being modeled is both the narrow-sense heritability  $h^2$  and the broad-sense heritability  $H^2$ ; they are the same here, because  $V_A$  is equal to  $V_G$ , which is true because the only source of genetic variance is the additive effects of the QTLs. If that were no longer the case (due to epistasis, maternal effects, dominance, etc.), then the model might need to be modified to correctly implement the particular type of heritability desired; that gets into quantitative genetics theory that we will not explore further here.

The heritability algorithm used in this recipe is adapted from code kindly provided by Mikhail Matz.

### 13.5 A QTL-based model with two quantitative phenotypic traits and pleiotropy

We have now seen a variety of quantitative trait models. All of these models shared their basic approach: QTL mutations are intrinsically neutral (enforced with a `mutationEffect()` callback), but have an additive effect upon a phenotypic trait value possessed by each individual. Individual fitness is then modeled using some form of fitness function (often Gaussian), based upon the deviation of the individual's phenotype from the optimum phenotype in its environment. All of these recipes model a single phenotypic trait, however, with the QTL mutations influencing only that one trait. In this section we will look at an extension of the same basic approach, modeling two phenotypic traits. QTL mutations in this model will influence *both* phenotypic traits, pleiotropically, with effect sizes drawn from a multivariate normal distribution (thus allowing the effects on the phenotypic traits to be either independent or correlated). This recipe can be trivially extended to encompass any number of QTL-based phenotypic traits, with any type of pleiotropy. Finally, at the end, we will add in live R-based plotting (introduced more comprehensively in section 14.8), to plot the adaptive trajectory of the population.

Let's begin with the `initialize()` callback as usual:

```
initialize() {
  initializeMutationRate(1e-7);

  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "f", 0.0); // QTLs
  m2.convertToSubstitution = F;
  m2.color = "red";

  // g1 is a neutral region, g2 is a QTL
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElementType("g2", c(m1,m2), c(1.0, 0.1));

  // chromosome of length 100 kb with two QTL regions
  initializeGenomicElement(g1, 0, 39999);
  initializeGenomicElement(g2, 40000, 49999);
  initializeGenomicElement(g1, 50000, 79999);
  initializeGenomicElement(g2, 80000, 89999);
  initializeGenomicElement(g1, 90000, 99999);
  initializeRecombinationRate(1e-8);

  // QTL-related constants used below
  defineConstant("QTL_mu", c(0, 0));
  defineConstant("QTL_cov", 0.25);
  defineConstant("QTL_sigma", matrix(c(1,QTL_cov,QTL_cov,1), nrow=2));
  defineConstant("QTL_optima", c(20, -20));
```



```

catn("\nQTL DFE means: ");
print(QTL_mu);
catn("\nQTL DFE variance-covariance matrix: ");
print(QTL_sigma);

// find Rscript
if (fileExists("/usr/bin/Rscript"))
  defineConstant("RSCRIPT", "/usr/bin/Rscript");
else if (fileExists("/usr/local/bin/Rscript"))
  defineConstant("RSCRIPT", "/usr/local/bin/Rscript");
else
  stop("Couldn't find Rscript.");
}

```

QTLs will be represented by `m2`; it is declared to be neutral here, so a `mutationEffect(m2)` callback making it neutral will not be needed. In other QTL recipes the `selectionCoeff` property of the mutations was used to store the additive effect of each QTL mutation, but we will take a different approach here. The `m2` mutations are colored red in SLiMgui, and are not converted to substitutions when they fix, since their phenotypic effect will continue to be important. The chromosome is a mixture of `g1` neutral regions and `g2` regions that can give rise to QTL mutations. Next, we define some QTL-related constants: `QTL_mu` gives the mean effect of new QTL mutations on the two phenotypes, `QTL_cov` gives the covariance between the two effects, and `QTL_sigma` is a variance-covariance matrix derived from `QTL_cov` that will govern new mutational effects (this matrix is sometimes called an M-matrix). `QTL_optima` gives the optimal phenotypic values for the two quantitative traits; note that the optima have different signs, but the M-matrix encodes a positive mutational correlation, so adaptation in this model will be contrary to the pleiotropically preferred direction of evolution. The means and M-matrix are printed out; the M-matrix looks like this:

```

QTL DFE variance-covariance matrix:
      [,0] [,1]
[0,]      1 0.25
[1,] 0.25      1

```

Support for matrices was added to Eidos in SLiM 2.6, so the `matrix()` function and other aspects of working with matrices in Eidos are relatively new; see the Eidos manual.

The last thing the `initialize()` callback does is find the location of the `Rscript` command-line tool, part of the standard R installation, which we will use later to run our R script; it is commonly found in one of two locations. If it is somewhere else on your system, you will need to fix this code to provide the correct location. You can execute `which Rscript` in a terminal window to find out the path on your system; if that does not find `Rscript`, you might need to install R on your system first.

Next we start a new subpopulation:

```

1 late() {
  sim.addSubpop("p1", 500);
}

```

Then we draw the effects of new mutations in each tick. In previous recipes this was done for us automatically by SLiM, by drawing a new “selection coefficient” (which was actually an effect size) from the DFE of the QTL mutation type. Here we do it ourselves instead, in a `mutation()` callback:

```

mutation(m2) {
  // draw mutational effects for the new m2 mutation
  effects = rmvnorm(1, QTL_mu, QTL_sigma);
  mut.setValue("e0", effects[0]);
  mut.setValue("e1", effects[1]);

  // remember all drawn effects, for our final output
  old_effects = sim.getValue("all_effects");
  sim.setValue("all_effects", rbind(old_effects, effects));

  return T;
}

```

This type of callback was added in SLiM 3.3; we saw it previously in section 10.6 (where we used it to attach a dominance coefficient to every new mutation). Section 25.9 provides the reference documentation on `mutation()` callbacks, but they are really quite straightforward. The callback here is called by SLiM whenever a new mutation of type `m2` is created (the declaration of it as a `mutation(m2)` callback specifies that focus). The callback can modify that focal mutation (passed to it as a pseudo-parameter named `mut`), or can log information about it, or can even veto its creation by SLiM altogether (by returning `F` rather than `T`). In many ways this is similar to a `mutationEffect()` callback, but whereas `mutationEffect()` callbacks are called to evaluate the fitness effect of a mutation in every tick, `mutation()` callbacks are called just once for a given mutation, at the moment it is created by SLiM.

Here, we call `rmvnorm()` to draw the effect sizes for the new mutation. This function draws from the multivariate normal distribution defined by `QTL_mu` and `QTL_sigma`; we ask it to generate one multivariate draw for the new mutation, and it returns a matrix with two columns, one for each effect size. Those effect sizes are placed into keys named `"e0"` and `"e1"` on the new mutation, for later reference. The callback also keeps a record of all drawn effect sizes in the `"all_effects"` key of the simulation object, for purposes of output. Finally, it returns `T` to tell SLiM to proceed with creating the proposed mutation.

Now that mutations have effect sizes assigned to them, we can add code to calculate the phenotype and fitness of each individual as the sum of those additive effects:

```

late() {
  for (ind in sim.subpopulations.individuals)
  {
    // construct phenotypes from additive effects of QTL mutations
    muts = ind.genomes.mutationsOfType(m2);
    phenotype0 = size(muts) ? sum(muts.getValue("e0")) else 0.0;
    phenotype1 = size(muts) ? sum(muts.getValue("e1")) else 0.0;
    ind.setValue("phenotype0", phenotype0);
    ind.setValue("phenotype1", phenotype1);

    // calculate fitness effects
    effect0 = 1.0 + dnorm(QTL_optima[0] - phenotype0, 0.0, 20.0) * 10.0;
    effect1 = 1.0 + dnorm(QTL_optima[1] - phenotype1, 0.0, 20.0) * 10.0;
    ind.fitnessScaling = effect0 * effect1;
  }
}

```

This event loops through the individuals in the simulation, and for each individual it gets all of the `m2` mutations possessed in both genomes, adds up their effects, and stores the result as a phenotypic trait value. Then it calculates the fitness effect for each of the two phenotypes, and `fitnessScaling` gets the multiplicative combination of these fitness effects.

There are only two important differences between this and previous QTL recipes. First, this recipe gets the mutational effect sizes from the "e0" and "e1" keys of the mutations, rather than from the selectionCoeff property (via sumOfMutationsOfType()). Second, it stores the phenotypes in "phenotype0" and "phenotype1" keys on the individuals, rather than in the tagF property. Using tagF works well when you have only a single value to store; it is simple to use, and is fast to get and set. Using key-value pairs as in this recipe is a bit more complex, and a bit slower, but more general and extensible; any number of keys can be defined on an object, and so this recipe can be easily extended to any number of phenotypic traits, each influenced by separate mutational effects. (Note that the only reason this recipe bothers to store the individual phenotype values at all is for output; however, other recipes have used them for purposes such as assortative mating.)

The fitness effect of each phenotypic trait is drawn here from a Gaussian function based on the difference between the phenotype and the optimum, as we have seen before. If one wished the two traits to be subject to a single multivariate Gaussian fitness function, instead of having independent fitness effects, a dmnorm() function is available in Eidos to facilitate such scenarios, but we shall not do that here. A width of 20.0 is hard-coded here for the fitness functions, but the optima are taken from the constant we defined earlier. The 10.0 multiplier makes it so an individual precisely at the phenotypic optimum has a fitness of 10.0 relative to an individual infinitely far from the optimum; strong selection, but perhaps not unrealistically so.

All that remains is output and a termination condition. For this model, this is quite complex, since we have a bunch of things that we want to look at:

```
1:1000000 late() {
  // output, run every 1000 cycles
  if (sim.cycle % 1000 != 0)
    return;

  // print final phenotypes versus their optima
  inds = sim.subpopulations.individuals;
  p0_mean = mean(inds.getValue("phenotype0"));
  p1_mean = mean(inds.getValue("phenotype1"));

  catn();
  catn("Cycle: " + sim.cycle);
  catn("Mean phenotype 0: " + p0_mean + " (" + QTL_optima[0] + ")");
  catn("Mean phenotype 1: " + p1_mean + " (" + QTL_optima[1] + ")");

  // keep running until we get within 10% of both optima
  if ((abs(p0_mean - QTL_optima[0]) > abs(0.1 * QTL_optima[0])) |
      (abs(p1_mean - QTL_optima[1]) > abs(0.1 * QTL_optima[1])))
    return;

  // we are done with the main adaptive walk; print final output

  // get the QTL mutations and their frequencies
  m2mut = sim.mutationsOfType(m2);
  m2freqs = sim.mutationFrequencies(NULL, m2mut);

  // sort those vectors by frequency
  o = order(m2freqs, ascending=F);
  m2mut = m2mut[o];
  m2freqs = m2freqs[o];
```

```

// get the effect sizes
m2e0 = m2mutts.getValue("e0");
m2e1 = m2mutts.getValue("e1");

// now output a list of the QTL mutations and their effect sizes
catn("\nQTL mutations (f: e0, e1):");
for (i in seqAlong(m2mutts))
  catn(m2freqs[i] + ": " + m2e0[i] + ", " + m2e1[i]);

// output covariances
fixed_m2 = m2mutts[m2freqs == 1.0];
cov_fixed = cov(fixed_m2.getValue("e0"), fixed_m2.getValue("e1"));
effects = sim.getValue("all_effects");
cov_drawn = cov(drop(effects[,0]), drop(effects[,1]));

catn("\nCovariance of effects among fixed QTLs: " + cov_fixed);
catn("\nCovariance of effects specified by the QTL DFE: " + QTL_cov);
catn("\nCovariance of effects across all QTL draws: " + cov_drawn);

sim.simulationFinished();
}

```

This should be fairly self-explanatory, so we won't go through it in any great detail. Every thousand cycles this 1:1000000 late() event prints a summary of the adaptation so far, like this:

```

Cycle: 1000
Mean phenotype 0: 0 (20)
Mean phenotype 1: 0 (-20)

Cycle: 2000
Mean phenotype 0: 2.89204 (20)
Mean phenotype 1: -3.42491 (-20)

Cycle: 3000
Mean phenotype 0: 2.95127 (20)
Mean phenotype 1: -5.91266 (-20)

...

```

The mean trait value is printed for each phenotypic trait, with the optimum in parentheses; the population didn't manage to adapt at all by tick 1000 (no QTL mutations had arisen without being lost), but by tick 3000 things are moving along better, especially for the second trait. The model continues running until both phenotypic means get within 10% of their optima. That can take a while, since evolution has to run counter to the M-matrix, and since selection gets weaker as the optima are approached. When it gets the peak, the model produces some final output:

```

...

Cycle: 31000
Mean phenotype 0: 21.2388 (20)
Mean phenotype 1: -17.9747 (-20)

Cycle: 32000
Mean phenotype 0: 20.7437 (20)
Mean phenotype 1: -18.6298 (-20)

QTL mutations (f: e0, e1):
1: 2.70148, -0.0621418

```

```

1: -0.291556, -1.16253
1: 0.923384, -1.24329
1: -0.431645, 0.170891
1: 1.82426, 0.911538
1: 0.0652766, -1.76526
1: 0.444558, -1.22771
1: 1.62655, -1.33047
1: 1.47864, -0.957393
1: 1.44353, -1.7145
1: 0.496692, -0.441125
1: 1.25745, 1.08043
1: -0.707134, -1.06546
0.801: -0.566786, -0.611955
0.017: -0.308758, -1.02767
0.001: -0.408555, -0.222152

```

Covariance of effects among fixed QTLs: 0.26061

Covariance of effects specified by the QTL DFE: 0.25

Covariance of effects across all QTL draws: 0.236848

After the output of the mean phenotypes, we get a dump of all of the segregating QTL mutations in the population, sorted by frequency. Most of the QTLs have fixed, one is approaching fixation, and two are at very low frequency. Their effect sizes on the two phenotypic traits are shown in the next two columns. Finally, we get a summary of the observed covariance in effects among the QTL mutations that fixed, the requested covariance, and the observed covariance across all effects drawn during the run (including many QTL mutations that were lost). You might expect that the covariance among the fixed QTLs would have to be negative, in order for adaptation to reach the two optima with different signs, but that is not the case; often it is true, but sometimes, as in this run, the optima can be reached even with a positive covariance among the mutational effects.

So we have a two-trait QTL-based adaptive walk model with pleiotropy and correlated mutational effects! As an aside, for advanced users planning to implement QTL models of their own: it is not, in fact, necessary to prevent the QTL mutations from fixing by setting `convertToSubstitution=F`. Instead, you can allow them to fix, and then add in the effects of the `Substitution` objects on the calculated phenotypes. This will be faster, if implemented carefully (tip: add newly fixed substitutions to an accumulator total), since the SLiM core won't get bogged down managing the bookkeeping on an ever-growing set of QTL mutations.

Now let's add a little extra code to plot the trajectory of the adaptive walk in SLiMgui, using the live R plotting technique of section 14.8. First let's open a PNG-based plot window in SLiMgui:

```

1 late() {
  sim.setValue("history", matrix(c(0.0, 0.0), nrow=1));
  defineConstant("pngPath", writeTempFile("plot_", ".png", ""));

  // If we're running in SLiMgui, open a plot window
  if (exists("slimgui"))
    slimgui.openDocument(pngPath);
}

```

This starts a history of the population's adaptive walk in a new key named "history" on the simulation object. It also makes a temporary PNG file with `writeTempFile()`, and tells SLiMgui to open that file; see section 14.8 for details.

Now we just need to update that plot with periodic callouts to R. To do so, insert the following code inside the 1:1000000 final output event, immediately above the comment “keep running until we get within 10% of both optima”:

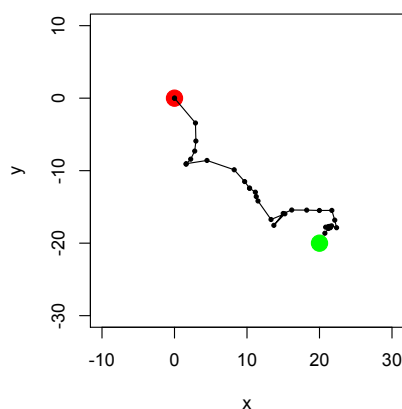
```
// update our plot
history = sim.getValue("history");
history = rbind(history, c(p0_mean, p1_mean));
sim.setValue("history", history);

rstr = paste('{',
  'x <- c(' + paste(history[,0], sep=", " + ')',
  'y <- c(' + paste(history[,1], sep=", " + ')',
  'png(width=4, height=4, units="in", res=72, file="' + pngPath + '"',
  'par(mar=c(4.0, 4.0, 1.5, 1.5))',
  'plot(x=c(-10,30), y=c(-30,10), type="n", xlab="x", ylab="y")',
  'points(x=0,y=0,col="red", pch=19, cex=2)',
  'points(x=20,y=-20,col="green", pch=19, cex=2)',
  'points(x=x, y=y, col="black", pch=19, cex=0.5)',
  'lines(x=x, y=y)',
  'dev.off()',
  '}', sep="\n");

scriptPath = writeTempFile("plot_", ".R", rstr);
system(RSCRIPT, args=scriptPath);
```

This updates the "history" key with the latest data, and then generates a string contain R plotting code and sends that code to R to run. The path to Rscript was looked up earlier, by code in our initialize() callback. The R code uses the png() function of R to open a new plotting device, generating a PNG file from the plot data when the device is closed.

When run in SLiMgui, a plot window will open and update every 1000 ticks to show the adaptive trajectory thus far. For the same run of the model as shown in the output above, the resulting plot looks like this (the walk begins at the red disc, and the green disc is the optimum):



This sort of live visualization of model dynamics can be extremely helpful for both model design and graphical debugging, and is quite simple to set up, as we have seen here.

In closing, it is perhaps worth emphasizing once more that although this recipe models two phenotypic traits, it is designed to be extensible. It uses key-value pairs set on the mutations and individuals to track the QTL mutational effects and phenotypic trait values; any number of such key-value pairs can be maintained. The rmvnorm() function can also draw from a multivariate normal distribution of any dimensionality, with any (positive-definite) M-matrix.



### 13.6 A variety of fitness functions

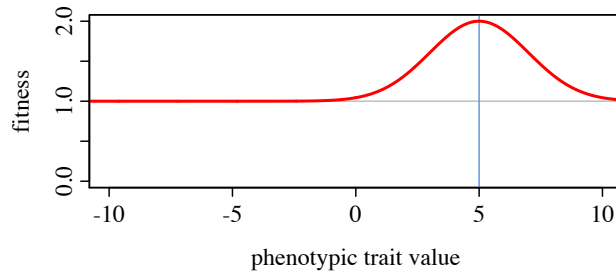
In previous recipes we have looked at successively more complex QTL-based models, introducing possibilities such as heritability, multiple phenotypic traits, and pleiotropy. However, the preceding models have all had one thing in common: they have all entailed a stabilizing fitness function, in which some particular phenotypic trait value is optimal (producing the highest fitness possible in the model), and as phenotype diverges from that optimum, fitness falls to progressively lower values. There are, however, many other fitness functions that one might wish to employ, and we will look at several here, including stabilizing selection (revisited as our starting point), directional selection, disruptive selection, truncation selection, and “squashed stabilizing selection”, a combination of stabilizing selection and negative frequency-dependent selection (Haller & Hendry, 2013).

As a baseline, let's start with *stabilizing selection*. For simplicity, we will mostly follow the recipe of section 13.2: randomly arising QTL mutations across the chromosome with effect sizes drawn from a normal distribution, influencing a single phenotypic trait through purely additive effects, using the `fitnessScaling` property of `Individual` to implement the effect of phenotype on fitness. However, rather than using the quadratic fitness function of recipe 13.2, let's use the `dnorm()` function that we used in the later recipes of this chapter, since it is a bit more standard. Our baseline model looks like this:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);      // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 0.15); // QTLs
  m2.convertToSubstitution = F;
  initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.1));
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
mutationEffect(m2) { return 1.0; }
1 early() {
  sim.addSubpop("p1", 500);
  cat("Phenotypes: 0");
}
1: late() {
  inds = sim.subpopulations.individuals;
  phenotypes = inds.sumOfMutationsOfType(m2);
  scale = dnorm(5.0, 5.0, 2.0);
  inds.fitnessScaling = 1.0 + dnorm(phenotypes, 5.0, 2.0) / scale;

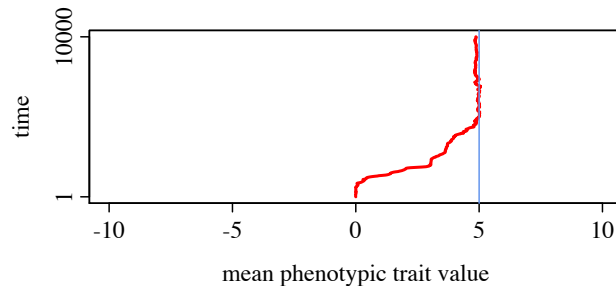
  if (sim.cycle % 10 == 0)
    cat(", " + mean(phenotypes));
}
5000 late() {
  m2mut = sim.mutationsOfType(m2);
  freqs = sim.mutationFrequencies(NULL, m2mut);
  effects = m2mut.selectionCoeff;
  catn();
  print(cbind(freqs, effects));
}
```

We rescale the result of `dnorm()` by dividing by the value of `dnorm()` at the optimum, making the optimum phenotype have a fitness of 2.0; this is an arbitrary choice, of course, and for a more realistic model would depend upon empirical measurements of the fitness surface governing the organisms of interest. This fitness function looks like this:



The blue line represents the fitness optimum, at a phenotypic trait value of 5.0. The red curve shows the fitness function itself: fitness as a function of phenotypic trait value. Since this is a model of stabilizing selection, fitness falls off as phenotype diverges from the optimum.

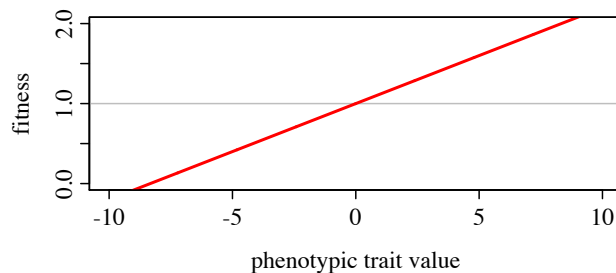
The model logs mean phenotypic trait values every tenth tick as it runs, and plotting those values allows us to see the effect of this fitness function. Note this in order to keep the same x-axis scale as the previous plot, for comparison, time (the independent variable) is shown here on the y-axis; apologies if that is confusing. We can very clearly see evolution toward the optimum phenotype over time, from the initial phenotypic trait value of 0.0:



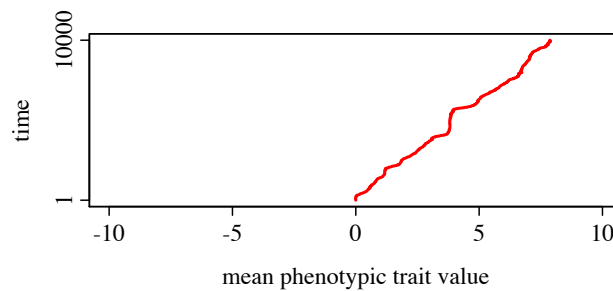
So far so good; this is all essentially review. Now let's look at *directional selection*, in which there is no optimum phenotype; instead, there is a linear relationship between phenotype and fitness, such that fitness increases (or decreases) monotonically as phenotype increases. To implement directional selection, let's replace the first four lines of the 1: `late()` event (while preserving the final two lines, which log out the mean phenotypes) with the following code:

```
inds = sim.subpopulations.individuals;
phenotypes = inds.sumOfMutationsOfType(m2);
inds.fitnessScaling = 1.0 + phenotypes * 0.12;
```

These lines implement the directional selection desired, with a linear relationship between phenotype and fitness. The constant 0.12 is the slope of that linear relationship; again, this is arbitrary, and would come from empirical data in a more realistic model. This produces the following fitness function:



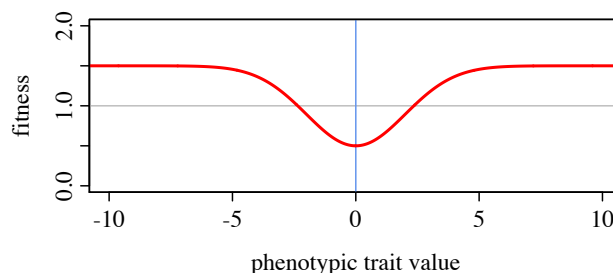
Fitness increases without bound with increasing phenotypic trait value, and so we would expect phenotype to evolve toward ever-higher values as the model runs, and indeed, that is what we see when the phenotype history is plotted:



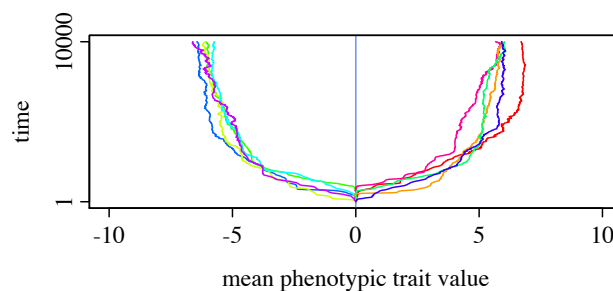
Next let's look at *disruptive selection*. This type of selection is, in a sense, the opposite of stabilizing selection: instead of a phenotypic optimum, there is what one might call a phenotypic *dysoptimum* – the phenotypic trait value that produces the lowest fitness possible. As phenotype diverges from that dysoptimum, fitness increases, both for phenotypic trait values higher than the dysoptimum, and for those lower than the dysoptimum. We can implement this by replacing the same lines in the 1: late() event with the following code:

```
inds = sim.subpopulations.individuals;
phenotypes = inds.sumOfMutationsOfType(m2);
scale = dnorm(0.0, 0.0, 2.0);
inds.fitnessScaling = 1.5 - dnorm(phenotypes, 0.0, 2.0) / scale;
```

This provides a fitness dysoptimum at 0.0, the initial phenotypic trait value for the population, with increasing fitness both below and above. Here is the resulting fitness function:



The dysoptimum is shown with the blue line. How will the population evolve in response to this fitness function? Let's answer that empirically, by plotting results from ten separate runs:



The population escapes the dysoptimum in either direction. If speciation were possible in the model, then perhaps the population might manage to diverge in *both* directions, separating into two distinct and reproductively isolated phenotypes; but since this model is panmictic, the

population “chooses” one direction or the other to evolve in, based upon the QTLs that happen to randomly arise early in the run. The population then runs in the chosen direction, slowing down as it reaches one of the flatter regions of the fitness function where selection for further directional change is weak.

Now let's look at *truncation selection*. With this type of selection, there is a threshold phenotypic trait value, on one side of which fitness is zero – having a phenotype beyond the threshold is lethal. On the other side of the threshold, the fitness might be constant, or some other type of selection might operate within that regime. To implement simple truncation selection, let's make all phenotypic trait values lower than 0.0 lethal with the following code, which again replaces the same lines of our baseline recipe:

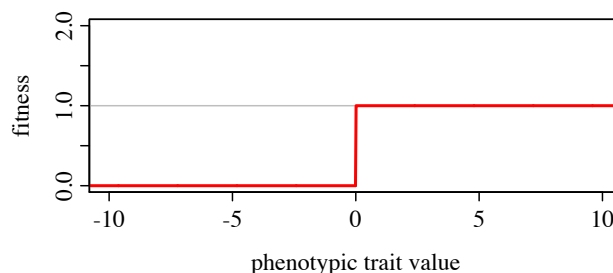
```
inds = sim.subpopulations.individuals;
phenotypes = inds.sumOfMutationsOfType(m2);
inds.fitnessScaling = ifelse(phenotypes < 0.0, 0.0, 1.0);
```

To make this interesting, we need to build up some genetic diversity in QTLs first, so let's run for 10000 ticks under neutral conditions first, with a population size of 5000, and then impose truncation selection for the next 5000 ticks. The output code needs to be tweaked a bit also. Since several modifications were needed, here is the complete model except for the `initialize()` callback, which remains unchanged from the baseline recipe:

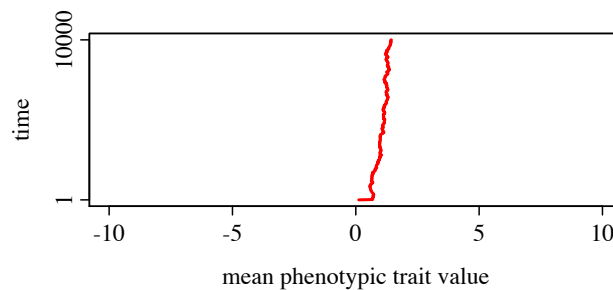
```
mutationEffect(m2) { return 1.0; }
1 early() { sim.addSubpop("p1", 5000); }
10000 late() {
  inds = sim.subpopulations.individuals;
  phenotypes = inds.sumOfMutationsOfType(m2);
  cat("Phenotypes: " + mean(phenotypes));
}
10001: late() {
  inds = sim.subpopulations.individuals;
  phenotypes = inds.sumOfMutationsOfType(m2);
  inds.fitnessScaling = ifelse(phenotypes < 0.0, 0.0, 1.0);

  if (sim.cycle % 10 == 0)
    cat(", " + mean(phenotypes));
}
15000 late() {
  m2mut = sim.mutationsOfType(m2);
  freqs = sim.mutationFrequencies(NULL, m2mut);
  effects = m2mut.selectionCoeff;
  catn();
  print(cbind(freqs, effects));
}
```

The fitness function is now a simple step function:



And here is the pattern of evolution in response to this fitness function:



This plots only the post-burnin period when truncation selection is operating. The initial phenotypic trait value is close to zero, although not exactly zero due to drift, but there would be considerable variance around that mean since the QTL mutations are neutral at that point. Then truncation selection kicks in, and the mean phenotype immediately jumps to the right; every individual with a phenotypic trait value less than 0.0 is being killed, so QTL mutations with a large negative effect size are weeded out very rapidly. After that initial jump, the population evolves more slowly to the right. One reason for this might be ongoing selection against any remaining QTL mutations with small negative effect sizes. A second reason might be that the truncation selection introduces a bias for new mutations; in effect, the DFE for new mutations excludes all new mutations with effect sizes that are sufficiently negative that their carrier's phenotype is less than 0.0. A third reason, of course, is that drift might happen to push the mean rightward in this particular run. Further investigation would be needed to tease apart the relative effects of these three mechanisms.

The last type of selection we'll look at here is *squashed stabilizing selection*, a term coined by Haller & Hendry (2013) to refer to a combination of stabilizing selection and negative frequency-dependent selection, which, they argue, there is reason to believe are commonly found together in nature due to ecological effects such as competition that disadvantage common phenotypes. We have seen negative frequency-dependent selection before at the level of individual mutations, as in sections 10.4.1 and 11.3, where the fitness effect of a mutation depends upon its frequency in the population, leading to effects such as balancing selection. Here, we are instead talking about negative frequency-dependent selection at the phenotypic level, where common phenotypes receive a fitness penalty compared to rare phenotypes. This frequency-dependent effect is combined multiplicatively with the standard effect of stabilizing selection.

The code for this recipe is based upon the baseline recipe (for stabilizing selection) given at the beginning of this section, with the usual lines in the 1: `late()` event replaced by this code:

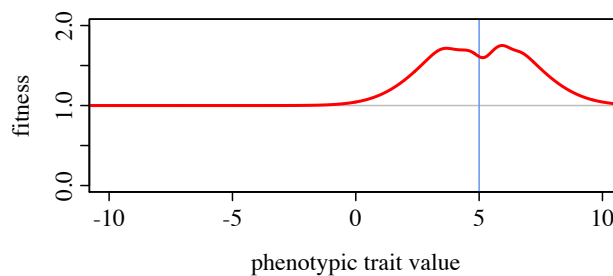
```
inds = sim.subpopulations.individuals;
phenotypes = inds.sumOfMutationsOfType(m2);
competition = sapply(phenotypes,
  "sum(dnorm(phenotypes, applyValue, 0.3));");
competition = (competition / size(inds)) / dnorm(0.0, 0.0, 0.3);
scale = dnorm(5.0, 5.0, 2.0);
inds.fitnessScaling = 1.0 +
  (dnorm(phenotypes, 5.0, 2.0) / scale) * (1.0 - competition);
```

This is by far the most complex fitness function we've seen, so let's tease it apart step by step. First of all, we calculate phenotypes from additive QTL effects as usual, yielding the `phenotypes` variable. Next, we want to know how much competition each individual is feeling from other individuals due to their similarity in phenotype; if many other individuals have similar phenotypes, the competitive effect felt will be strong, if few, it will be weak. We calculate this for each

individual using `sapply()` to loop over all of the phenotypes present in the population, and for each phenotype present, the lambda `"sum(dnorm(phenotypes, applyValue, 0.3))"` calculates the sum of the competitive effects from all individuals, using a Gaussian competition function with a width of `0.3`. The result of the `sapply()` call is therefore a vector of the strength of competition felt by each individual. The next line normalizes those competition strengths by the number of individuals in the model and the maximum competition strength possible, yielding final competition strengths usable in calculating fitness.

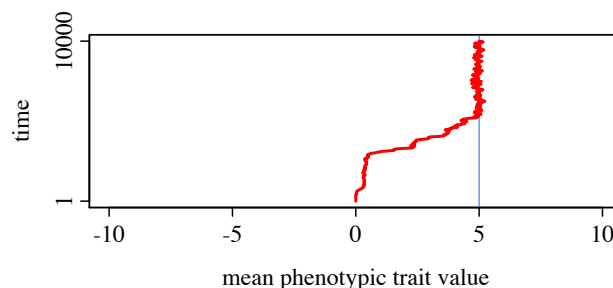
The last two lines then calculate fitness values. This follows the fitness calculation code for stabilizing selection, except that for each individual the increase in fitness due to proximity to the fitness optimum is scaled downward by the strength of competition felt by that individual. Individuals close to the optimum might nevertheless have a low fitness (close to neutral), if there are many other individuals with similar phenotype. This implements a type of squashed stabilizing selection where competition only decreases the benefits of proximity to the optimum, but never decreases individual fitness below `1.0`; of course other types of fitness function would also be possible, and the choice of model should depend upon the biology of the system being modeled.

For one particular run of the model, here is the fitness function at the end of the run:



It is now clear where the term “squashed stabilizing selection” comes from; the fitness peak at the optimum (shown with the blue line) has been squashed downward into a dimpled shape due to the effects of competition. The shape of this fitness function is dynamic, and changes in every tick based upon the distribution of phenotypes in the population. In effect, the population is under a stable sort of disruptive selection: in every tick, selection disfavors individuals near the mean and favors evolution away from the mean, and yet, because of the underlying stabilizing fitness function, that disruptive selection never causes the population to evolve very far away from the optimum. This state of affairs will persist indefinitely, unless speciation allows the population to split into two or more separate lineages that can pursue independent evolutionary trajectories.

The phenotypic history for this run of the model looks like this:



It is instructive to compare this plot to the earlier plot for the simple stabilizing selection model. Both models move toward the phenotypic optimum in very similar ways initially. Once they have arrived at the optimum, however, the squashed stabilizing selection model fluctuates around the optimum much more rapidly, and perhaps travels further from the optimum than the stabilizing



selection model does; it is under constant pressure, in every tick, to try to move away from the position it presently occupies, while simultaneously being constrained to stay within the vicinity of the optimum. This produces various interesting effects and dynamics as explored in Haller & Hendry (2013); that paper also provides a video of the squashed stabilizing fitness function over time, showing how its shape fluctuates in response to the changing phenotypic distribution. We will see squashed stabilizing selection again in sections 15.6 and 15.7, where the strength of competition will be evaluated for us by SLiM's `InteractionType` facility.

The recipes in this section have really provided just a taste of the variety of possible fitness functions; one can delve into multi-peak fitness functions, multidimensional fitness functions, and so forth, but that is beyond the scope of this manual. The approach shown here is general, however: map phenotype to fitness according to the environment and mechanisms present in your system, and use the `fitnessScaling` property to impose those fitness effects upon individuals.

## 14. Advanced models

This chapter will present some advanced models that draw upon many of the concepts covered in previous chapters, while also using relatively advanced features of Eidos and SLiM that may not have been covered in previous recipes. A knowledge of the topics covered in previous chapters will be assumed, and simple details will not be explained in depth, to keep things short.

### 14.1 Relatedness, inbreeding, and heterozygosity

Inbreeding is an important concept in evolutionary biology, but it is not very precisely defined. It can result from a variety of processes, from small population size to assortative mating, and it can manifest in a variety of genetic patterns, such as decreased heterozygosity, loss of genetic diversity, and a decreased time to the most recent common ancestor of pairs of individuals. The particular effects of inbreeding observed in a system may depend on the process generating the inbreeding. So before embarking on a study of inbreeding, one should define one's terms clearly.

Here, we will construct a model of inbreeding that results from a tendency of individuals to mate with their close kin, as defined by their pedigree-based relatedness. SLiM (beginning in version 2.1) has a built-in facility for tracking this type of relatedness, which can be turned on with the call `initializeSLiMOptions(keepPedigrees=T)` in the `initialize()` callback of a script (see section 24.1). When this call is made, individuals in a simulation will keep track of the identities of their parents and grandparents, and the relatedness between individuals can then be assessed using the `relatedness()` method of `Individual` (see section 24.7.2). The pedigree information is also available through properties on the `Individual` class (section 24.7.1), for purposes such as locating “trios” (two parents and an offspring that they generated) for analysis.

It should be emphasized that the relatedness metric available through this mechanism is purely pedigree-based. This can be quite different from genetic relatedness, which depends not only on pedigree but also on factors such as assortment and recombination (see sections 1.7 and 14.7). The inbreeding generated by this model will be based upon this relatedness metric.

With that as preamble, here is the first stage of the model:

```
initialize() {
  initializeSLiMOptions(keepPedigrees = T);
  initializeMutationRate(1e-5);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-7);
}
1 early() {
  sim.addSubpop("p1", 100);
}
1000 late() {
  // Print mean heterozygosity across the population
  heterozygosity = calcHeterozygosity(p1.genomes);
  cat("Mean heterozygosity = " + heterozygosity + "\n");
}
```

This is a simple neutral model, similar to those we have seen before. It turns on pedigree tracking in its `initialize()` callback, and then runs for 1000 ticks. At the end of the run, a `late()` callback computes the mean heterozygosity across the population, using the built-in function `calcHeterozygosity()`. That function was added to SLiM in version 3.5, and is actually written in Eidos under the hood; you can see its source code with the `functionSource()` function if you're

curious, by executing `functionSource("calcHeterozygosity")` in the Eidos console. Various population-genetics utility functions like this are now available; see section 24.17.2.

So now we have a baseline model that has only whatever inbreeding results from its small population size of 100 individuals. Now let's add a `mateChoice()` callback that uses the pedigree tracking information to create a mating preference for kin:

```
mateChoice() {  
    // Prefer relatives as mates  
    return weights * (individual.relatedness(sourceSubpop.individuals) +  
        0.01);  
}
```

This uses the `relatedness()` method of `Individual` to calculate the relatedness between the focal individual that is choosing a mate (`individual`) and all of its potential mates (`sourceSubpop.individuals`). A constant of 0.01 is added to those values to guard against the possibility that all of the values would be exactly zero; that would result in a weights vector of all zeros being returned to SLiM, which would reject the proposed mating (see the discussion in section 25.4), which we don't want to happen. In fact, this particular model is safe from that problem, because the relatedness of an individual to itself is 1.0, and so individuals will always be able to mate with themselves (since individuals here are hermaphroditic); but the safeguard is shown here to raise awareness of the issue, since in many other types of models this issue will need to be considered – sexual models, models with multiple subpopulations and migration, etc. Finally, the relatedness values are multiplied by `weights`, the vector of default mating weights supplied to the callback by SLiM. Again, this is not important for this model (as a neutral, non-sexual model, `weights` will be a vector of all 1 values), but it is important for models more generally, so that the fitness-based mating weights are taken into account even when a `mateChoice()` callback is implemented. Without this, the fitness-based mating weights computed by SLiM would be completely replaced by the `mateChoice()` callback, making all selection coefficients and `mutationEffect()` callbacks irrelevant – rarely what is wanted.

This particular callback makes the mating weights be (almost) proportional to relatedness. Individuals that share no grandparents will have a mating weight of only 0.01, whereas full siblings will have a weight of 0.51 and an individual will have a weight of 1.01 for hermaphroditic selfing. This should generate fairly strong inbreeding. Of course a `mateChoice()` callback could rescale or otherwise modify these values to produce whatever mating preferences are desired.

That's all of the code needed for the model; the relatedness-based mating preference requires just a one-line `mateChoice()` callback. If we run this model ten times with the `mateChoice()` callback, the average of the reported heterozygosity values across those runs is 0.001638, whereas the average across ten runs without the mate choice callback is 0.002827. Clearly the `mateChoice()` callback is having a pronounced effect on the heterozygosity observed in the model, as intended (with a two-sample independent *t*-test *p*-value of 0.002, if you're skeptical).

Incidentally, it would also be possible to implement the tendency towards mating with kin using a `modifyChild()` callback instead. That callback would evaluate the relatedness of the two parents of the proposed child, and would tell SLiM to short-circuit generation of the child, with some probability, if the relatedness of the parents was not sufficiently high. If only a weak inbreeding effect is desired, this might be much faster than the `mateChoice()` scheme shown above, since for the generation of a typical child only one or a few relatedness values would need to be calculated, and the relatively large overhead of running the `mateChoice()` callback would be avoided. This should be very simple, so it is left as an exercise for the reader.

## 14.2 Mortality-based fitness

Normally, in WF models, SLiM uses fitness values as mating weights: high-fitness individuals are more likely to be chosen as mates than low-fitness individuals (see section 22.2.2). By default, SLiM does not model mortality; there is no concept of individuals dying before reaching reproductive age (except for the suppression of child generation with a `modifyChild()` callback, which can be viewed as a type of mortality-based fitness; see chapter 12). However, it is straightforward to add mortality to a model: if model mechanics dictate that an individual dies, then it can be given a fitness of zero, which means that it cannot possibly be chosen as a mate; effectively, it has been removed from the population (although it will remain as an individual in the population until the next generation starts; there is no way to actually remove dead individuals from the population in WF models).

(Note that this section is aimed primarily towards WF models; in nonWF models, as discussed in section 1.6, fitness translates into mortality anyway, rather than influencing mating, so every nonWF model is a model of mortality-based fitness. However, in nonWF models it can still be useful to explicitly kill off a specific individual, either by making its fitness zero or by calling the nonWF-only method `killIndividuals()`; see discussion in section 16.5.)

Here we will look at three different models of how mortality might be implemented in WF models. The first model converts the fitness effects of mutations directly into mortality using a `mutationEffect()` callback. This might be a good way to model death due to deleterious genetic factors. The second model is a tag-based model; if individuals die, their `tag` value is set to zero, and then a special `mutationEffect()` callback is used to reduce the fitness of tagged individuals to zero. This might be a good way to model death due to non-genetic factors, such as predation. The third model uses the `fitnessScaling` property of `Individual`, introduced in SLiM 3.0.

One question regarding mortality-based fitness might be: what difference does this make? Why would one wish to model mortality-based fitness rather than (or in addition to) mating-based fitness? The answer is that the two types of fitness have different effects on the distribution of the number of offspring generated by individuals. With mating-based fitness, all individuals of a given low fitness value are equal in the offspring-production game; the expected number of offspring is the same for all, and is lower than the expected number of offspring for a high-fitness individual. With mortality-based fitness, however, all individuals of a given low fitness value are *not* equal, by the time mating season arrives: some are alive, and some are dead. Those that are alive have an expected number of offspring that is just as high as a high-fitness individual; they survived to mating season, and now the playing field is even. Those that are dead, on the other hand, have an expected number of offspring of zero. What the evolutionary consequences of this difference might be is not the focus here; but there clearly is a difference, and so we want to be able to model mortality-based fitness.

So, with no further ado, let's look at our first model:

```
initialize() {
    initializeMutationRate(1e-7);

    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 0.5, "f", -0.005);    // deleterious
    m2.convertToSubstitution = F;

    initializeGenomicElementType("g1", c(m1,m2), c(1.0,0.1));
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
```

```

1 early() {
    sim.addSubpop("p1", 500);
}
mutationEffect(m2)
{
    // convert fecundity-based selection to survival-based selection
    if (runif(1) < effect)
        return 1.0;
    else
        return 0.0;
}
10000 late() {
    sim.outputMutations(sim.mutationsOfType(m2));
}

```

Mutations are mostly neutral (m1) but occasionally slightly deleterious (m2), and the deleterious mutations are not converted to substitutions when they fix (using `m2.convertToSubstitution = F`), since we want them to continue to cause mortality. At the end of a run, the model uses `outputMutations()` to print information about all of the m2 mutations existing in the population (include those that have fixed, since they do not get substituted).

The interesting part of the model is the `mutationEffect()` callback. It converts a fitness effect for an m2 mutation (which will be 0.995 if homozygous and 0.9975 if heterozygous, given the dominance coefficient of 0.5 used for m2) into either 0 or 1, with the probability of a fitness of 1 being equal to the fitness effect of the focal mutation in the individual. If the new fitness value is 0, the focal m2 mutation has resulted in mortality; the individual will not mate, and is effectively dead. If the new fitness is 1, on the other hand, the individual has survived the deleterious effects of the focal m2 mutation, and that mutation will have no further deleterious effect; it will not influence the individual's expected number of offspring, since its fitness effect is now 1.

There are a couple of things to note here. First of all, the mortality effects of multiple m2 mutations in a single individual will combine multiplicatively, just as their non-mortality-based fitness effects would combine multiplicatively in SLiM without the `mutationEffect()` callback. This is because the `mutationEffect()` callback will be called for each m2 mutation, and the probabilities of mortality that these callbacks cause will combine multiplicatively (as independent probabilities generally do).

Second, the way that this `mutationEffect()` callback works depends on the m2 mutations being deleterious, not beneficial. This is because for a beneficial mutation, effect would be greater than 1, and so the comparison with `runif(1)` would always be T. On a more conceptual level, you can't be more likely to survive than a survival probability of 1. If you wish to model beneficial mutations with a mortality-based effect, you would need to provide some baseline probability of mortality in the model, such that an individual with no mutations would still have, say, a probability of 0.5 of dying before reaching reproductive age. The presence of beneficial mutations could then reduce this probability of dying, down to the limit of a probability of 0.0. (Section 16.4 discusses this in depth, in the context of nonWF models.)

Third, there is no obstacle to combining this sort of mortality-based fitness effect with the usual mating-based fitness effects of SLiM, using other mutation types. There is nothing magical about this model; the `mutationEffect()` callback here is just another `mutationEffect()` callback, albeit one that (somewhat unusually) models a stochastic fitness effect that varies from individual to individual.

Now let's look at the second recipe for mortality-based fitness, this one driven by tags:

```

initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
late() {
    // initially, everybody lives
    sim.subpopulations.individuals.tag = 1;

    // here be dragons
    sample(sim.subpopulations.individuals, 100).tag = 0;
}
fitnessEffect() {
    // individuals tagged for death die here
    if (individual.tag == 1)
        return 1.0;
    else
        return 0.0;
}
10000 late() { sim.outputFull(); }

```

This model involves only neutral `m1` mutations. It has a `fitnessEffect()` callback that evaluates phenotypic fitness (in this case, mortality) that is called once per individual per cycle, without reference to any focal mutation, allowing a fitness effect to be generated that depends upon the overall state of each individual. This technique was previously used in section 13.1, where it was used to define the phenotypic fitness effect of additive QTLs.

In this case, the “overall state” that the `fitnessEffect()` callback models is mortality due to having previously been tagged as dead. The `fitnessEffect()` callback simply returns `1.0` (lived) for individuals with a tag of 1, and `0.0` (dead) for all others. The tag values are assigned in a `late()` event that runs near the end of every tick. In this model, 100 individuals out of the 500 in the population are chosen for death at random using the `sample()` function, but that is just an arbitrary placeholder for whatever sort of mortality-generating logic you might wish to implement in your model – predation, social interactions, developmental disorders, or anything else. Indeed, the logic could depend on the genetics of individuals in some way, combining the approach of this recipe with that of the previous recipe.

One note here is that the tagging logic occurs in a `late()` event. That is because tag values for newly generated offspring need to be assigned before fitness values for the new offspring generation are calculated, and a `late()` event is the right time to do that (see the tick cycle diagram in chapter 22). This has the side effect that mortality does not occur in the first generation at all; tag values have not been assigned, and so the whole set of machinery that generates mortality is not yet active. This might sound unfortunate, but in fact it is the way the previous recipe worked as well, and indeed it is the way that models in SLiM generally work: since the first parental generation starts out with empty chromosomes, and since new mutations are added to genomes during offspring generation, the first generation generally is not subject to any selection. If the initial population were created in a `late()` event instead of an `early()` event – which is probably better practice, in fact – this issue goes away, since the initial generation will then be treated like every other generation in the model.



OK, with that discussion out of the way it is time for the third and final recipe for mortality-based fitness:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
late() {
    // here be dragons
    sample(sim.subpopulations.individuals, 100).fitnessScaling = 0;
}
10000 late() { sim.outputFull(); }
```

This is equivalent to the second recipe, but is much simpler and runs faster. In this recipe, we use the `fitnessScaling` property of `Individual`, which was added in SLiM 3.0, to directly kill off individuals in the `late()` event, rather than using tag values and a `fitnessEffect()` callback that translates those tag values into fitness effects. The `fitnessScaling` values get multiplied into each individual's calculated fitness value by SLiM, just as the value returned by the `fitnessEffect()` callback did in the second recipe. Note that it is not necessary to initialize the `fitnessScaling` values to 1.0 in each new generation; SLiM does that automatically.

The second recipe, then, is generally inferior, but is included for a couple of reasons. One is simply the historical reason that, prior to SLiM 3.0, it was a good way to implement mortality-based fitness, and might still be of interest for that reason alone. It also illustrates the use of tag values in a very simple model, which is perhaps useful. Finally, since it is strictly equivalent to the third recipe, it might expose the underlying logic of the third recipe more clearly; using the `fitnessScaling` property is a bit “magical”, with a lot of what is happening hidden behind the scenes, whereas the second recipe shows the mechanism more explicitly.

Note that something like the first recipe could also be implemented using `fitnessScaling`, in fact. To achieve this, you would make the selection coefficient of the `m2` mutations be 0.0, rather than -0.005, so that they are neutral as far as SLiM's fitness-calculation machinery is concerned. Then, you would loop through individuals in a `late()` event, and for each individual you would count `m2` mutations, determine a survival probability based on that count ( $0.995^{(\text{count}/2)}$ , perhaps), draw a random number to determine survival, and set `fitnessScaling` to 0 for the individuals that did not survive. This would not be quite the same as the first recipe, though, since it would not account for homozygosity versus heterozygosity (i.e., dominance effects) in the same way. Which strategy is preferable would depend upon the biology you were trying to model.

### 14.3 Reading initial simulation state from an MS file

At the beginning of execution of a SLiM model, the genomes of all individuals are empty; they contain no mutations (see discussion in section 1.5.1). Mutations can be introduced explicitly in script (see section 9.1), or the saved state of a SLiM simulation can be read in to provide a non-empty initial state (see section 24.14.2, and section 9.2 for an example). Sometimes, however, information about the desired initial state of the model will be in a file that is in a non-SLiM format, and you will want to read that file in and create that initial state in the model. In this section, we will examine a recipe for reading in a file that is in the popular “MS” format. This can

be useful, since MS uses a very fast coalescent method to generate genetic diversity in a neutral model; it can be used to generate an initial “burn-in” state, saved in MS format, that a SLiM model can then use as a starting state simulations.

**This recipe is probably of mainly historical interest.** One reason is that a `readFromMS()` method on `Genome` is available in SLiM 3.3 and later, which does what this recipe does except much faster (see section 24.4.2). A second reason is that nowadays, if one wishes to start a simulation using “burn-in” state from a coalescent simulation, one probably wants to use tree-sequence recording (see section 1.7). In particular, the `msprime` coalescent simulator can be used to run the burn-in and save the result as a `.trees` file that SLiM can read directly (see sections 17.8 and 17.9); or even better, `msprime` can be used to “recapitate” a SLiM simulation with a coalescent burn-in after the fact (see section 17.10). Nevertheless, we’ll keep this recipe since it provides a good illustration of how to create initial mutations in a model based upon data from a file.

As a first step, we need a file in MS format to read in. We could use MS to generate that, of course, but instead, for illustration purposes here, we will use a simple neutral SLiM model:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 1000);
}
20000 late() {
    p1.outputMSSample(2000, replace=F, filePath="ms.txt");
}
```

Notice that the `outputMSSample()` call samples without replacement, using `replace=F`, and it requests 2000 samples – twice the size of the population, because there are two genomes per diploid individual. This means that the call will output not just a sample, but the full state of the entire population. The result of this model is a standard MS-format file, saved out to `ms.txt`:

```
//
segsites: 345
positions: 0.0047300 0.0048000 0.0053201 0.0063001 0.0068701 0.0072201
0.0078301 0.0098501 0.0140401 0.0144601 0.0162002 0.0196102 0.0236902...
0010000000001000000000000000100100100100000000100001010000000000000000
00000000000000000001000000010011000100000000111000100000000000000001100
10000001111000100000000000000001001000001001000000101100100000001000010001
001000000000000000000000000000010000000001000000000000000000000000010000000000
00000000000000000100000000000000000000000000000000000000000100
011000000000000000000000000000000000000000000000000000000000000000010
00000000000001000000000000000000000000000000000000000000000000000001
0000010000000000000000000000000000000000000000000000000000000000000000
0000000000011000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
...
```

Ellipses have been used here to abbreviate the lengthy content of the file. It begins with a `//` line that marks the beginning of a sample block. A `segsites:` line then gives the number of segregating sites per sample, and a `positions:` line gives the positions, in the interval `[0,1]`, of each segregating site. The remainder is a series of samples, one per line, with 1 and 0 values indicating whether each corresponding mutation is (1) or is not (0) present in that genome.

Now let's look at a recipe for reading this file back in and instantiating it into neutral mutations. Here we use the same chromosome length, population size, and other parameters, so that the loaded state corresponds to the model's defined dynamics. This is not required, but be careful that what you are doing makes sense. Here is the recipe:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 late() {
  sim.addSubpop("p1", 1000);

  // READ MS FORMAT INITIAL STATE
  lines = readFile("ms.txt");
  index = 0;

  // skip lines until reaching the // line, then skip that line
  while (lines[index] != "//")
    index = index + 1;
  index = index + 1;

  if (index + 2 + p1.individualCount * 2 > size(lines))
    stop("File is too short; terminating.");

  // next line should be segsites:
  segsitesLine = lines[index];
  index = index + 1;
  parts = strsplit(segsitesLine);
  if (size(parts) != 2) stop("Malformed segsites.");
  if (parts[0] != "segsites:") stop("Missing segsites.");
  segsites = asInteger(parts[1]);

  // and next is positions:
  positionsLine = lines[index];
  index = index + 1;
  parts = strsplit(positionsLine);
  if (size(parts) != segsites + 1) stop("Malformed positions.");
  if (parts[0] != "positions:") stop("Missing positions.");
  positions = asFloat(parts[1:(size(parts)-1)]);

  // create all mutations in a genome in a dummy subpopulation
  sim.addSubpop("p2", 1);
  g = p2.genomes[0];
  L = sim.chromosome.lastPosition;
  intPositions = asInteger(round(positions * L));
  muts = g.addNewMutation(m1, 0.0, intPositions);

  // add the appropriate mutations to each genome
  for (g in p1.genomes)
  {
    f = asLogical(asInteger(strsplit(lines[index], "")));
    index = index + 1;
    g.addMutations(muts[f]);
  }
}
```

```

// remove the dummy subpopulation
p2.setSubpopulationSize(0);

// (optional) set the tick and cycle to match the save point
community.tick = 20000;
sim.cycle = 20000;
}
30000 late() { sim.outputFixedMutations(); }

```

We will focus on a few salient points in the tick 1 `late()` event, which first creates the `p1` subpopulation and then reads in the MS file and adds mutations to the individuals in `p1` as needed.

First of all, the `readFile()` function is used to read in the MS data. This function returns a string vector, with one string element per line in the file. The rest of the code then processes these lines. First, a scan through the lines is conducted to find the `//` line that indicates the start of the actual sample data; MS files can have various information above that point that this code does not attempt to parse. Below that line should be a `segsites:` line and then a `positions:` line, as shown in the snippet above; the code scans for those lines and does some minimal error-checking.

Next, the recipe creates all of the mutations referenced by the MS data. This recipe assumes that these mutations are all neutral, since that is how MS would typically be used as input to a SLiM script. One twist here is that mutations cannot be created in isolation; according to SLiM's design, mutations must always reside in a `Genome` object. This recipe therefore creates a dummy subpopulation with a single individual, and throws all of the MS mutations into the first genome of that individual. This design may seem a bit odd, but it is harmless. Also, note that MS positions in `[0,1]` are converted to discrete base positions by multiplying by the last base position (`L`) and rounding; this is the opposite of what `outputMSSample()` does, so it recovers the original base positions. For MS positions that do not originate in SLiM, however, this formula results in half as much “density” at positions 0 and `L` as at other base positions; `floor(positions * (L+1))` might be better in that case, but note that a position of exactly 1.0 would then produce `L+1`, which is out of range, so that edge case would need to be checked for and clamped down to `L`.

Having created all of the mutations, the recipe then returns to processing input lines, which now are each a representation of one genome, as explained above. The `strsplit()` function is used to separate the 0 and 1 values into separate string elements, which are then converted to integer and then to logical. This results in a logical vector that indicates whether each corresponding mutation is or is not referenced by the focal genome. A call to `addMutations()` adds the selected mutations to the focal genome, and then the code moves to the next input line.

At the end, the dummy `p2` subpopulation is removed. Finally, the tick and cycle are set to 20000, to dovetail with the fact that the simulation that generated the MS data ended at tick 20000.

A similar strategy could be used for other purposes, such as to read in empirical SNP data (but see `readFromVCF()`, section 24.4.2, which may provide a better solution).

## 14.4 Modeling chromosomal inversions with a `recombination()` callback

In previous chapters we have seen five types of callbacks: `initialize()`, `mutationEffect()`, `fitnessEffect()`, `mateChoice()`, and `modifyChild()`. There are actually a few other callback types that are less commonly used; here we will focus on `recombination()` callbacks (see section 25.6). This type of callback allows the script to modify the recombination breakpoints used by SLiM when generating a gamete to produce an offspring individual. In most models, the standard recombination map set by `initializeRecombinationRate()` suffices, since usually all individuals use the same map (or perhaps different maps for males and females, which is supported by that call as well). In some cases, however, recombination behavior needs to vary at the individual level. That would be true in a model of the evolution of recombination itself, for example; one

would want individual-level variation in recombination, controlled by the genetics of individuals, to evolve in response to natural selection. It is also true in the model we will explore here: a model of chromosomal inversions.

We'll build this model in two steps. Here's our starting point:

```
initialize() {
  defineConstant("L", 1000000);
  defineConstant("INV_LENGTH", 500000);
  defineConstant("INV_START", asInteger(L/2 - INV_LENGTH/2));
  defineConstant("INV_END", INV_START + INV_LENGTH - 1);
  defineConstant("N", 500);

  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral sites
  initializeMutationType("m2", 0.5, "f", 0.0); // start marker
  initializeMutationType("m3", 0.5, "f", 0.0); // end marker
  c(m2,m3).convertToSubstitution = T;
  c(m2,m3).color = "red";

  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);
}
1 late() {
  sim.addSubpop("p1", N);

  // give some genomes an inversion
  inverted = sample(p1.genomes, 100);
  inverted.addNewDrawnMutation(m2, INV_START);
  inverted.addNewDrawnMutation(m3, INV_END);
}
mutationEffect(m2) {
  // fitness of the inversion is frequency-dependent
  f = sim.mutationFrequencies(NULL, mut);
  return 1.0 - (f - 0.5) * 0.2;
}
5000 late() {
  sim.outputFixedMutations();

  // Assess fixation inside vs. outside the inversion
  pos = sim.substitutions.position;
  catn(sum((pos >= INV_START) & (pos <= INV_END)) + " inside.");
  catn(sum((pos < INV_START) | (pos > INV_END)) + " outside.");
}
```

This initial model is just a model of neutral background mutations with balancing selection on an introduced mutation. The tick 1 `late()` event gives some genomes “marker mutations”, of types `m2` and `m3`, that indicate the presence of an inversion; however, the machinery to implement the inversion behavior is not yet present. The first base position inside the inversion will be located at `INV_START`; the last will be located at `INV_END`.

In this model we want the `m2` inversion marker to be subject to balancing selection strong enough to keep the inversion near intermediate frequency. (The `m3` inversion marker will be subject to the same dynamics, since it will always travel with the `m2` marker; if it doesn't, that would be a bug in the model. The `m3` marker is actually just present for visualization in SLiMgui; it is unused in the model.) This is a proxy for the inversion itself (or more realistically, an unmodeled mutation within the inversion) having a phenotypic effect that is under balancing selection in the

environment. The `mutationEffect()` callback in the code above provides this functionality (see section 10.4.1 for more discussion of modeling frequency-dependent selection).

The tick 5000 `late()` event produces final output for the model. First it prints a list of the fixed mutations using `outputFixedMutations()`. Then it looks at all of the mutations that have fixed during the simulation (kept by the `sim` object in its `substitutions` property), and extracts their positions. Finally, it tallies and prints the number of fixed mutations that occurred inside versus outside the inversion region. If we run this model now, we'll see final output like this:

```
79 inside.  
101 outside.
```

Or this:

```
219 inside.  
129 outside.
```

Or this:

```
58 inside.  
94 outside.
```

The point being that the number of fixed mutations inside versus outside the inversion are comparable (since those regions are equal in length), albeit with a lot of variation. This makes sense, because so far our marker mutations are still just ordinary mutations under balancing selection; we have no machinery to make them model a chromosomal inversion in particular. Now we will add the core of this recipe – a `recombination()` callback that modifies recombination when an inversion is present, to reflect how inversions work biologically.

At this point, however, we need to go on a little digression. Prior to SLiM 3.7, this recipe implemented the effect of the inversion with the following `recombination()` callback (modified slightly to update it to the current recipe framework used here). The code below is **not** the final recipe, and the implementation here is seriously flawed, as we will discuss below, but it is useful to understand it as a first step to understanding the final recipe:

```
recombination() {  
    // THIS CODE DOES NOT CORRECTLY MODEL INVERSIONS!  
    if (genome1.containsMarkerMutation(m2, INV_START) ==  
        genome2.containsMarkerMutation(m2, INV_START))  
        return F;  
  
    inInv = (breakpoints > INV_START) & (breakpoints <= INV_END);  
    if (!any(inInv))  
        return F;  
  
    breakpoints = breakpoints[!inInv];  
    return T;  
}
```

Let's walk through this in some detail. First, the callback determines whether the parent individual that is generating the focal gamete is heterozygous for the inversion. The inversion, as modeled here, only affects recombination if the parent is heterozygous, so in other cases the callback returns F immediately. A return of F is a flag value indicating that no change to the proposed breakpoints is needed, allowing SLiM to handle this case very efficiently.

Next – handling the heterozygous case – a logical vector, `inInv`, is constructed that has T for proposed breakpoints that are within the inversion region, F otherwise. The positions of proposed




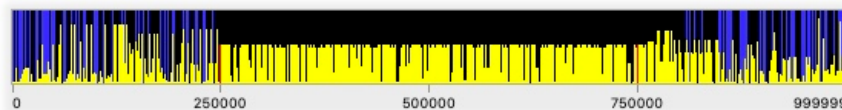
breakpoints are supplied to the callback by SLiM in the `breakpoints` variable; that variable can also be set by the callback to change the proposed breakpoints as we will see momentarily. The inversion region is defined by the code here to stretch from base position `INV_START` to `INV_END` inclusive. Recombination positions fall immediately to the *left* of the given base position; in other words, crossover occurs between the specified base and the *preceding* base. For this reason, the logic here considers a breakpoint exactly at position `INV_START` to be outside the inversion, while a breakpoint exactly at position `INV_END` is inside the inversion.

If there are no proposed breakpoints inside the inversion region, the callback returns `F` immediately; it does not modify the breakpoints in that case. Otherwise – now we are handling the case of a parent that is heterozygous for the inversion *and* drew breakpoint positions inside the inversion – the callback removes all proposed breakpoints inside the inversion by subsetting `breakpoints` with the negation of `inInv`, and then it returns `T` to indicate that the proposed breakpoints were changed. This achieves the desired goal of suppressing all recombination within the inversion region for gametes generated by heterozygote parents.

With this `recombination()` callback active, the results are very different from before:

```
0 inside inversion.
43 outside inversion.
```

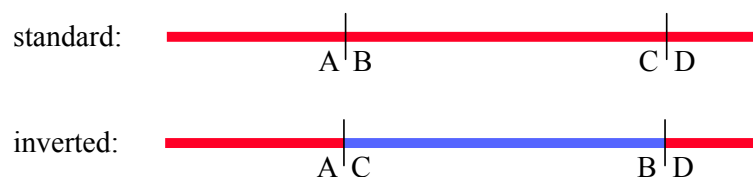
This can be seen graphically in SLiMgui. If display of fixed mutations is turned on with the  button to the right of the chromosome view, this is what things look like at the end of the run:



The red marker mutations at `INV_START` and `INV_END` (250000 and 749999, as parameterized here) are visible if you look closely. Many mutations outside the inversion have drifted to fixation. Inside the inversion, however, there are two major haplotypes, and mutations within the inversion can't fix because they can't cross from one haplotype to the other. This is the result of suppression of recombination within the inversion; chromosomes containing the inversion accumulate one set of mutations through drift, while chromosomes not containing the inversion accumulate a different set of mutations.

So far, so good – except that this model is wrong. The problems were kindly pointed out and rectified by Sara Schaal, Vince Buffalo, Peter Ralph, and Andy Kern; the issue posted on GitHub at <https://github.com/MesserLab/SLiM/issues/203> might be of interest to those who wish to dig further into this. The SLiM-Extras repository at <https://github.com/MesserLab/SLiM-Extras> now has a subfolder named `inversions` that contains their model and results. Their model uses tree-sequence recording, which is beyond the scope of this chapter (see section 1.7 and chapter 17), but I have integrated their changes into this recipe without using tree-sequence recording, and will summarize both the problems they found and how they fixed them.

There are two problems with the `recombination()` callback above. To understand the first problem, consider what an inversion actually looks like biologically:



Here A is a point just to the left of INV\_START, B is a point just to the right of it, etc. Since SLiM does not actually understand inversions, the genome is laid out ABCD in SLiM for both inverted and uninverted individuals, however; it is up to the script to make recombination in SLiM do what is biologically correct given the conceptual picture above.

Consider what happens, with the `recombination()` callback above, for an individual that is *homozygous for the inversion*, when a single breakpoint is drawn inside the inversion. Since the individual is homozygous for the inversion, the callback simply returns F and recombination proceeds as usual; since the genome in SLiM is laid out ABCD, this will mean that the crossover somewhere between B and C will result in AB being inherited together from one parental genome, and CD being inherited together from the other parental genome. Looking at the diagrams above, however, this is clearly incorrect; with a crossover somewhere between C and B, AC should have traveled together, and BD should have traveled together.

So that is one problem. The other problem involves individuals that are *heterozygous for the inversion*. In such individuals, the `recombination()` callback above forces exactly zero recombination breakpoints to occur inside the inversion, by simply removing all breakpoints that fall inside the inversion. This is, again, not biologically correct. In fact, for heterozygous individuals any *even* number of recombination breakpoints is permitted (see Guerrero, Rousset & Kirkpatrick 2012).

How can these problems be fixed? Here is a better `recombination()` callback, from Schaal, Buffalo, Ralph & Kern:

```
recombination() {
  gm1 = genome1.containsMarkerMutation(m2, INV_START);
  gm2 = genome2.containsMarkerMutation(m2, INV_START);
  if (!(gm1 | gm2)) {
    // homozygote non-inverted
    return F;
  }
  inInv = (breakpoints > INV_START) & (breakpoints <= INV_END);
  if (sum(inInv) % 2 == 0) {
    return F;
  }
  if (gm1 & gm2) {
    // homozygote inverted
    left = (breakpoints == INV_START);
    right = (breakpoints == INV_END + 1);
    breakpoints = sort(c(breakpoints[!(left | right)],
      c(INV_START, INV_END + 1)[c(sum(left) == 0, sum(right) == 0)]));
    return T;
  } else {
    // heterozygote inverted: resample to get an even # of breakpoints
    // this is *recursive*: it calls this recombination callback again!
    breakpoints = sim.chromosome.drawBreakpoints(individual);
  }
  return T;
}
```

As before, if the parent is homozygous non-inverted the callback simply returns F; but if the parent is homozygous inverted it no longer does that since that case needs to be treated more carefully. Next we get the breakpoints that are inside the inversion, as before, and if there is an even number of them we again return F (fixing the second problem described above).

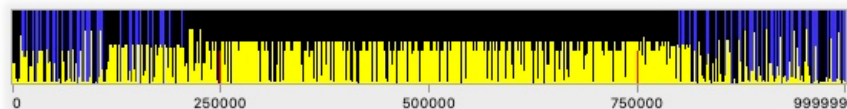
Third, we handle the case of a parent that is homozygous inverted; the logic here is a bit tricky in order to correctly handle breakpoints that fall exactly at the ends of the inversion (at INV\_START

or at `INV_END + 1`), but essentially this code adds new breakpoints at `INV_START` and `INV_END + 1` to the breakpoints vector. This makes AC and BD travel together as they should (fixing the first problem described above). It then returns `T`, telling SLiM that the breakpoints vector has been changed. To see how this works for inheritance in homozygous inverted individuals, notice that A and C will be inherited together if there are an even number of breakpoints between the two; the same goes for B and D. We ensure that this is the case here by adding breakpoints to both ends of the inversion, since we know that an odd number of breakpoints have been drawn within in the inversion. And, as long as A and C are inherited together correctly, and similarly for B and D, then we are guaranteed that the remainder of the chromosome will be as well.

Finally, we handle the case of the parent being heterozygous inverted. If the number of breakpoints inside the inversion is even, we already returned `T` above, so here the number of breakpoints inside the inversion must be odd, and that is not permitted, biologically. What we want to do is to simply redraw breakpoints from scratch until we get an even number instead. This adopts a “no-cost infinite gamete assumption”, which seems reasonable biologically. This strategy is implemented by simply calling the Chromosome method `drawBreakpoints()` to generate a new set of breakpoints for us. That works because `drawBreakpoints()` calls our `recombination()` callback to vet the breakpoints again; this `recombination()` callback is therefore called *recursively*, meaning that it calls itself (indirectly, in this case, as a side effect of calling `drawBreakpoints()`). If this recursive call to the `recombination()` callback again sees an odd number of breakpoints inside the inversion, it will again call `drawBreakpoints()`, which will again call the `recombination()` callback, and the recursion will go another level deeper. In principle this recursion could go arbitrarily deep, drawing an odd number of breakpoints inside the inversion again and again, but in practice that is so unlikely that we don’t need to worry about it. When an even number of breakpoints inside the inversion is obtained, the recursion unwinds and the correct breakpoint vector is returned to SLiM.

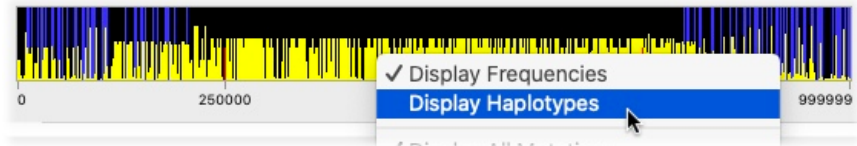
Remarkably, this model will even correctly handle variation in the recombination rate along the chromosome (i.e., a recombination rate map as in section 6.1.1), as long as one does not wish the inversion to actually *change* the recombination rate map (to have a different pattern of hotspots versus coldspots when the inversion is present). This works because breakpoints are drawn and executed according to SLiM’s ABCD conception of the genome, whether the inversion is present or not.

In this way, all the cases are handled correctly and the recipe models inversions in a way that is much closer to the biological reality. Here’s the result of a run in SLiMgui:



The appearance is much the same; recombination is still being suppressed quite effectively inside the inversion, and so fixation occurs primarily outside the inversion, and two distinct haplotypes segregate inside the inversion region, as before. This is true because the recombination rate of  $1e-8$  is low enough that getting two (or four, or six...) recombination breakpoints inside the inversion is highly unlikely; with a higher recombination rate (or a longer inversion), that would occur more often and recombination would be more likely to act inside the inversion (fixing the second problem above). Also (although it is also not readily visible in SLiMgui), linkage patterns at the boundary of the inversion will now be correct; mutations just outside and just inside the inversion should tend to travel together correctly (fixing the first problem above). Results showing the effects of these improvements are in the SLiM-Extras repository linked to above.

While we're on the subject of haplotypes, this model is a particularly good testbed for looking at some advanced features of SLiMgui that facilitate the examination of haplotypes and linkage disequilibrium in SLiM (also discussed in chapter 8). Let's control-click on the chromosome view to get a context menu that allows us to configure its display:

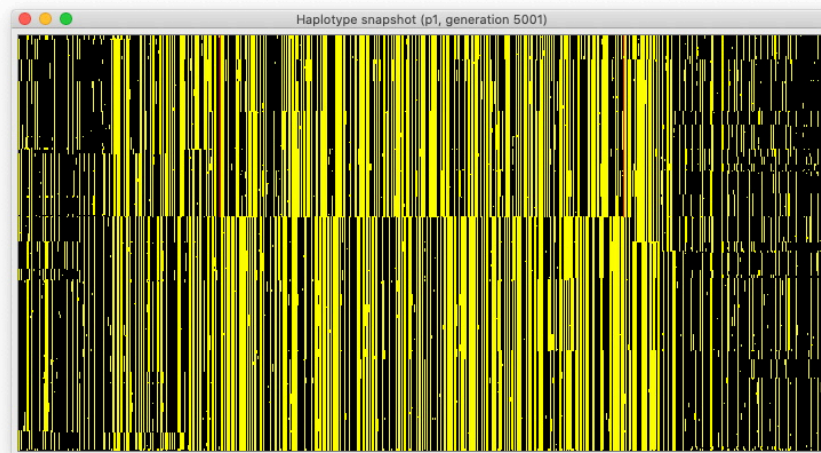


Select "Display Haplotypes" from the context menu, and the chromosome view changes to show the haplotypes that are in the population, clustered according to their genetic similarity:



Here the effect of the inversion is even clearer; outside of the inversion there are few mutations at high frequency and not much consistent population structure, but inside the inversion the population has clearly differentiated into two completely different haplotypes with little or no admixture. The marker mutation at the start of the inversion can be seen, associated with one of the two haplotypes; the one at the end of the inversion is lost amid the noise here.

This alternate display mode for the chromosome view is based upon just a small sample of the genomes from the selected subpopulation(s), allowing genetic clustering and display to be done in real time. It can also be useful to get a more comprehensive haplotype plot, based upon a larger sample or upon the entire population. To obtain that, choose Create Haplotype Plot from the Show Graph button's pop-up menu (or from the Simulation menu). This shows a panel that allows us to choose plot options; we can use the default options here, so click OK. After a progress panel (since the analysis can be quite lengthy with a large sample), a new plot window opens:



This shows essentially the same information as the chromosome view did above, but it is based upon all 1000 genomes in the population, and thus provides some additional detail – both inversion markers are visible, and the linkage between regions just outside versus just inside the inversion is much more readily apparent. A context menu on the plot window, obtained by control-clicking or right-clicking within the window, allows the appearance of the plot to be

adjusted, and also allows the plot's image to be copied or saved as a file. Note that section 13.3 also used a haplotype plot, to look at genome-wide divergence due to assortative mating, and section 15.9 uses one to look at speciation in a spatial model.

That completes this model of chromosomal inversions using a `recombination()` callback, but as usual there is much more that could be done. Rather than using balancing selection, for example, spatially varying selection among subpopulations could allow adaptive ecological divergence between subpopulations to arise as a result of the protection from recombination afforded by the inversion, thereby preserving the inversion as a result of divergent selection and local adaptation. It would also be interesting to model the rise of an inversion to high local frequency, in a model like that, to explore how inversions can facilitate divergence and speciation.

Thanks to Sara, Vince, Peter, and Andy for contributing this new and improved recipe.

## 14.5 Modeling both X and Y Chromosomes with a Pseudo-Autosomal Region (PAR)

SLiM has built-in support for modeling either the X or Y chromosome when sex is enabled (see section 6.2.3). However, some models need to go beyond this built-in support. You might wish to model *both* the X and Y chromosomes, and you might even wish to model a pseudo-autosomal region (PAR) – a region in which recombination between the X and Y occurs freely, giving the region evolutionary dynamics similar to those of an autosome. Both males and females are diploid for genes in the PAR; females have two copies of the PAR on their two X chromosomes, whereas males have the same PAR on their X, and a homologous PAR on their Y. Because crossing over occurs freely between the X and Y within the PAR, genes in the PAR exhibit an autosomal pattern of inheritance rather than sex-linked inheritance.

Modeling this in SLiM is possible by implementing your own sex-chromosome mechanics, which is fairly straightforward. In this section we'll explore a simple model of neutral X and Y chromosome evolution with a single PAR connecting them. This model was provided by Melissa Jane Hubisz, and has been adapted for publication as a recipe here. The recipe:

```
initialize()
{
    initializeMutationRate(1.5e-8);
    initializeMutationType("m1", 0.5, "f", 0.0); // PAR
    initializeMutationType("m2", 0.5, "f", 0.0); // non-PAR
    initializeMutationType("m3", 1.0, "f", 0.0); // Y marker

    // 6 Mb chromosome; the PAR is 2.7 Mb at the start
    initializeGenomicElementType("g1", m1, 1.0); // PAR: m1 only
    initializeGenomicElementType("g2", m2, 1.0); // non-PAR: m2 only
    initializeGenomicElement(g1, 0, 2699999); // PAR
    initializeGenomicElement(g2, 2700000, 5999999); // non-PAR

    // turn on sex and model as an autosome
    initializeSex("A");

    // no recombination in males outside PAR
    initializeRecombinationRate(c(1e-8, 0), c(2699999, 5999999), sex="M");
    initializeRecombinationRate(1e-8, sex="F");
}
1 late() {
    // initialize the pop, with a Y marker for each male
    sim.addSubpop("p1", 1000);
    i = p1.individuals;
    i[i.sex == "M"].genome2.addNewMutation(m3, 0.0, 5999999);
}
```



```

modifyChild() {
    numY = sum(child.genomes.containsMarkerMutation(m3, 5999999));

    // no individual should have more than one Y
    if (numY > 1)
        stop("### ERROR: got too many Ys");

    // females should have 0 Y's
    if (child.sex == "F" & numY > 0)
        return F;

    // males should have 1 Y
    if (child.sex == "M" & numY == 0)
        return F;

    return T;
}
10000 late() {
    p1.outputMSSample(10, replace=F, requestedSex="F");
}

```

The `initialize()` callback sets up the genetic structure. This model turns on sex, because we want SLiM to track males and females for us, but it requests modeling of an autosome with `initializeSex("A")`; we will handle the tracking of the X versus Y chromosomes ourselves. To do that, we set up a special “marker” mutation type, `m3`, that will be used to tag Y chromosomes, as we will see shortly. We set up a 6 Mb chromosome with a 2.7 Mb PAR at the beginning; the PAR uses mutation type `m1` (through genomic element type `g1`), while the rest of the chromosome uses mutation type `m2` (through genomic element type `g2`), so that we can easily tell sex-linked mutations from pseudo-autosomal mutations later on. The only wrinkle during initialization is that we set up a recombination map in males that prevents recombination outside the PAR; females, which have two X chromosomes, are allowed to recombine freely.

Next we have a tick 1 `late()` event that sets up the initial population. After making a new subpopulation in the usual way with `addSubpop()`, it gets the male individuals, selects only their second `Genome` objects using the `genome2` property, and adds an `m3` mutation to them to mark them as Y chromosomes. These marker mutations will be handled in the usual way by SLiM, so they will be inherited and will continue to mark Y chromosomes in future generations. Because recombination is prevented in males outside the PAR, they will stay associated with the non-PAR Y chromosome genetic information.

There is one problem with this scheme, however. When SLiM generates offspring, it has its own ideas about whether a given child ought to be male or female. We need to follow SLiM’s guidance on this, otherwise our model will end up with individuals that SLiM considers female but that possess a Y chromosome, and individuals SLiM thinks are male but that have no Y. This is the purpose of the `modifyChild()` callback. It simply compares what SLiM expects for the sex of the child (`child.sex`) with the genetics that SLiM is proposing that the child will inherit (`child.genomes`). If they don’t match, then it returns `F` to indicate that SLiM needs to choose new parents and try again. Note the `containsMarkerMutation()` method; this just checks for a mutation of a given type at a given position, which can be done quite quickly by SLiM. It is thus optimal when the position of a mutation (if it exists) is already known, as it is here.

Because `modifyChild()` callbacks get called quite frequently (once for each new offspring generated by SLiM, or even more in this case since the callback rejects some proposed children), the speed of callback code is essential. The callback above has very clear logic, but it is slow in several ways. It does a safety check that is never, in fact, hit (since the model works properly); it



assigns a value into a variable, numY, which is relatively slow because it requires Eidos to set up a symbol table entry; and it performs some unnecessary logical operations and tests (if child.sex is not "F" then it *must* be "M", for example). Worst of all, it calls containsMarkerMutation() for both child genomes, but this is completely unnecessary because SLiM always guarantees (unless you change it in script somehow) that an individual's first genome is inherited from the maternal parent and its second genome is inherited from the paternal parent. If a Y chromosome is present, it will therefore always be in the second genome; there is no need to check the first chromosome for having a Y chromosome. With an optimized version of the callback, the model runs about twice as fast, a very significant difference. The optimized callback:

```
modifyChild() {
    // females should not have a Y, males should have a Y
    if (child.sex == "F")
        return !child.genome2.containsMarkerMutation(m3, 5999999);
    else
        return child.genome2.containsMarkerMutation(m3, 5999999);
}
```

This will produce the same result in all cases (unless the model has already broken, such as by an individual having two Y chromosomes, or by a Y chromosome being present in the first genome of a proposed child somehow).

Finally, we have a late() event that outputs an MS-format sample from the females in the population; among other things, this establishes the end of the model as tick 10000.

There is one remaining issue. Addressing it is optional, in a sense, but if we don't address it the model will run slower and slower until it grinds nearly to a halt. The problem is that while PAR mutations will fix and be converted into Substitution objects automatically by SLiM as usual, non-PAR mutations will not. This is because their threshold for fixation is lower than SLiM realizes; only a quarter of Genome objects are Y chromosomes, and only three-quarters are X chromosomes, so sex-linked mutations need to fix when they reach those frequencies, not a frequency of 1.0 as SLiM expects. Fixing this is not difficult, but it does require a bit of code:

```
1:10000 late() {
    // periodically remove m2 (non-PAR) mutations that are fixed in X or Y
    // m1 (PAR) mutations will be automatically removed when fixed
    if (sim.cycle % 1000 == 0) {
        numY = sum(p1.individuals.sex == "M");
        numX = 2 * size(p1.individuals) - numY;

        // look at the mutations in a single Y chromosome
        // to find mutations that are fixed in all Y's
        firstMale = p1.individuals[p1.individuals.sex == "M"][0];
        fMG = firstMale.genomes;

        if (fMG[0].containsMarkerMutation(m3, 5999999)) {
            firstY = fMG[0];
            firstX = fMG[1];
        } else if (fMG[1].containsMarkerMutation(m3, 5999999)) {
            firstY = fMG[1];
            firstX = fMG[0];
        } else
            stop("### ERROR: no Ys in first male");

        ymut = firstY.mutationsOfType(m2);
        ycounts = sim.mutationCounts(NULL, ymut);
        removeY = ymut[ycounts == numY];
```

```


// now do the same for the X
xmutts = firstX.mutationsOfType(m2);
xcounts = sim.mutationCounts(NULL, xmutts);
removeX = xmutts[xcounts == numX];

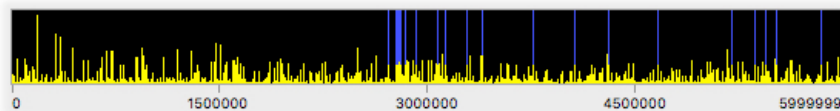
cat("Cycle " + sim.cycle + ": Removing ");
cat(removeX.size() + "/" + removeY.size() + " on X/Y\n");

removes = c(removeY, removeX);
sim.subpopulations.genomes.removeMutations(removes, T);
}
}

```

This event should ideally be inserted before the tick 10000 `late()` event that produces final output. It runs every 1000 ticks, since the operation it performs is somewhat time-consuming; doing it every tick would be wasteful. It finds a male individual, and uses that individual as a template for finding and removing fixed sex-linked mutations. Any mutation that has fixed must be possessed by the template male (by the definition of “fixation”), so the code just gets all the mutations of type `m2` (non-PAR mutations) from the male’s X and Y and checks them all for fixation. The fixation check itself is done using `mutationCounts()`, which returns the number of occurrences of the mutations – population-wide, in this case, because of the `NULL` passed for the first parameter. The event prints the number of X-linked and Y-linked mutations that it intends to remove, and then it removes them with `removeMutations()`. The optional `T` value passed for the second argument to `removeMutations()` (which is named `substitute`) indicates that SLiM should create `Substitution` objects for the removed mutations; we are notifying SLiM that those mutations are in fact fixed, even though SLiM doesn’t realize it. If you don’t care about that record-keeping, you can omit that optional `T` value, and the mutations will simply be removed.

And that’s it. We’ve implemented tracking of Y chromosomes with marker mutations, we’ve guaranteed that those markers stay correctly synchronized with offspring sex, and we’ve added machinery to detect fixed sex-linked mutations and turn them into `Substitutions` just as SLiM does for autosomal mutations. If we run this model in SLiMgui with display of fixed mutations turned on (with the  button to the right), the behavior of the PAR versus the non-PAR regions is easy to see as soon as the first pass of the fixation check has run in tick 1000:



The PAR, on the left, behaves as an autosome, so it takes a while for mutations to fix; one is close to fixation, but none has made it there yet. The non-PAR region, on the right, behaves as separate sex chromosomes that cannot recombine, and each sex chromosome is present in fewer copies than the PAR; mutations in that region thus have a smaller effective population size, and fix more rapidly. The Y, present in only a quarter as many copies as the PAR, fixes particularly quickly; all 20 of the fixations here are in fact on the Y. After tick 3000, the situation is similar:



There have now been 19 fixations on the X, 120 on the Y, and only 11 in the PAR. The trend is clear, and it appears that our model of a pseudo-autosomal region is functioning as expected.

## 14.6 Estimating model parameters with ABC

One major use of simulation models is to try to better understand an empirical system by comparing simulated results with empirical data. For example, one might have an observation from an empirical system, and have a guess as to a model that approximates that empirical system well; one might then want to know the value of a particular parameter of that model that produces the best possible fit of the model to the data. In this recipe we'll explore doing this using a technique called Approximate Bayesian Computation (ABC). This is quite a complex topic, and we will not attempt to provide a thorough introduction to it here; please use the internet to inform yourself further regarding the assumptions, limitations, and caveats involved in this method, as well as about the underlying theoretical framework upon which it rests.

In this recipe we're going to branch out a bit and present R code as well as Eidos code. The R code will run the ABC process, while the Eidos code will run the SLiM simulation that provides the ABC process with the information it needs. Let's start with the Eidos code:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 999999);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 100); }
1000 late() { cat(sim.mutations.size() + "\n"); }
```

This is a trivial model, obviously. We start with a population of size 100, and model neutral mutations in that population. The model is allowed to equilibrate until tick 1000, at which point the number of segregating mutations detected in the population is printed. Simple and fast, which is important since ABC is going to run it a whole bunch of times; the drawback of ABC-based methods is that they can be quite computationally expensive.

To tie this back to an empirical question, this model would be a reasonable starting place if you said: "I have a population of size 100 that I have sampled comprehensively, and the analysis I've done tells me there are 262 segregating mutations in the population. I think the population has been about size 100 for a long time, so it's at equilibrium (to the extent that a small population evolving under drift is ever at equilibrium). I know the recombination rate and the chromosome length, but I don't know the mutation rate. What's my best guess, and what's the posterior distribution around that guess?" So now we've got a SLiM model of that scenario, with a guess of  $1e-7$  hard-coded into it. Let's tweak the model by replacing the mutation rate with a symbolic constant:

```
initializeMutationRate(mu);
```

We could define the constant mu at initialization time with a call to the Eidos function `defineConstant()`, like:

```
defineConstant("mu", 1e-7);
```

But let's not do that; instead, we will pass a value for it in to the simulation on the command line. To do this, let's first save the model to a file called "abc.slim" in our home directory (we will assume that the file is then at the path `~/abc.slim`, as it is on most Unix systems including macOS). If we run this model at the command line, we get an error regarding the undefined constant:

```

darwin:~ bhallar $ slim ~/abc.slim
...
ERROR (EidosSymbolTable::_GetValue): undefined identifier mu.

Error on script line 2, character 24:

    initializeMutationRate(mu);
                        ^^

```

But we can pass a value for the constant in, using the `-d` (`-define`) command-line option (see section 20.2):

```

darwin:~ bhallar $ slim -d mu=1e-7 ~/abc.slim
...
262

```

This defines `mu` to be  $1e-7$  at the very beginning of the model, before the first `initialize()` callback is called. This run of SLiM is actually where the value of 262 came from, above; it's the value from our "empirical system" (i.e., this first run of our model), for which we're going to try to recover an estimate of `mu` using ABC. In practice, such observed values would probably come from field or lab data.

So now we have a working model at `~/abc.slim` that expects a value for a constant `mu` to be supplied to it on the command line, and the last line of the output it generates is the outcome of the model – the number of segregating mutations observed in the population at equilibrium. The next step is to write an R function to run our model, given a random number seed and a mutation rate `mu`. This function can use the `system2()` function of R to run SLiM with the desired command-line arguments:

```

runSLiM <- function(x)
{
  seed <- x[1]
  mu <- x[2]
  #cat("Running SLiM with seed ", seed, ", mu = ", mu, "\n");
  output <- system2("/usr/local/bin/slim", c("-d", paste0("mu=", mu),
    "-s", seed, " ~/abc.slim"), stdout=T)
  as.numeric(output[length(output)])
}

```

The output is collected as a character vector of output lines. Note that this function takes the seed and `mu` values as a single vector, `x`; this is because of the design of the R package we will use to run the ABC in a moment. First, let's test this function:

```

> runSLiM(c(100030, 1e-7))
[1] 281

```

We get a different result than before, since a different random number seed was used, but it seems to work fine. So now we have R running our SLiM model for us.

The next step is to actually run an ABC analysis. For this, we will use the package `EasyABC`, available through CRAN:

```
library(EasyABC)

# Set up and run our ABC
prior <- list(c("unif", 1e-9, 1e-6))
observed <- 262

ABC_SLiM <- ABC_sequential(method="Lenormand", use_seed=TRUE,
  model=runSLiM, prior=prior, summary_stat_target=observed,
  nb_simul=1000)
```

The EasyABC package has many different options for running ABC analyses, including several “sequential” ABC methods that try to arrive at an answer more quickly through successive rounds of ABC, and several ways of running the ABC inside an MCMC (Markov Chain Monte Carlo) process, a particularly powerful way of converging rapidly on the posterior distribution of the ABC. We have chosen the “Lenormand” method, since it is simple to set up, is happy to run with only one unknown parameter (which some of the other methods don’t seem to be, strangely), and converges fairly quickly. Since ABC is a Bayesian method, we also need a prior for  $\mu$ ; we have chosen a uniform prior from  $1e-9$  to  $1e-6$ , since that encompasses the range of mutation rates we think would be likely in our empirical system, and since we have no information about which values within this range are more or less likely. We also need to tell EasyABC how many runs we want to perform; the `nb_simul` parameter does that (sort of – see the documentation), and the value of **1000** here should give us quite a good posterior distribution at the expense of longer runtime (a value of **100** actually works pretty well already).

The final line, which actually runs the ABC analysis, will probably take several minutes, depending upon the speed of your machine. When it finishes, `ABC_SLiM` contains information about the ABC run. Details about that information can be found in EasyABC’s documentation; we will not explain it here, but we will use it to extract the results we want. First of all, to get the best fit estimate from a one-parameter ABC like this, one typically wants the sum of the weighted values chosen by the ABC:

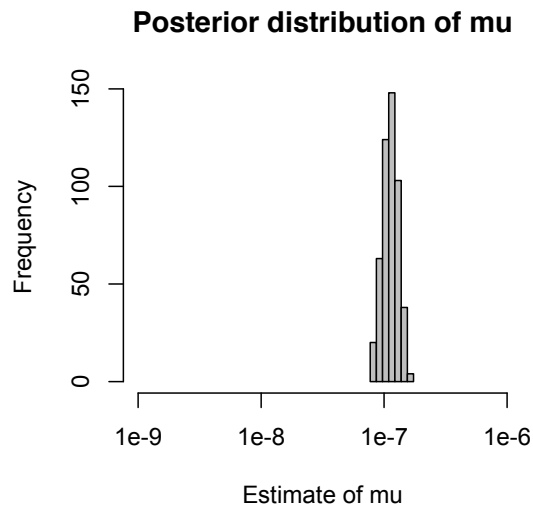
```
> sum(ABC_SLiM$param * ABC_SLiM$weights)
[1] 0.0000001134854
```

The ABC did quite a good job of recovering the actual value of  $\mu$ ,  $1e-7$ , that was used to generate the target value of 262. We can also plot the posterior distribution for  $\mu$  that was arrived at by the ABC analysis, using this R code:

```
log_param <- log(ABC_SLiM$param, 10)
breaks <- seq(from=min(log_param), to=max(log_param), length.out=8)

quartz(width=4, height=4)
hist(log_param, xlim=c(-9, -6), breaks=breaks, col="gray",
  main="Posterior distribution of mu", xlab="Estimate of mu", xaxt="n")
axis(side=1, at=-6:-9, labels=c("1e-6", "1e-7", "1e-8", "1e-9"))
```

This code converts the posterior distribution data to a log scale manually and plots it on that scale, for clarity; there are probably better ways to do that. The `quartz()` call just opens a graphics device on macOS; if you’re on a different platform you will probably have to change that to your platform-specific graphics device call. Apart from those details, the code is pretty standard. The resulting plot looks like this:



We started with a uniform prior from  $1e-9$  through to  $1e-6$ , providing the ABC analysis with no information as to which values within that range were more likely. By doing quite a few runs of SLiM (12500, as it happens), the ABC narrowed that range down considerably, effectively ruling out a large part of it completely, and it gave us a posterior distribution showing which values of  $\mu$  would be most likely to produce the observed number of segregating mutations, 262, that was observed empirically. All in just a few lines of Eidos and a few lines of R; not bad!

This completes our foray into Approximate Bayesian Computation in SLiM and R. This topic can be pursued much further: estimation of the joint distribution of multiple parameters, usage of a Markov Chain Monte Carlo (MCMC) method for running the ABC (which is also supported by the EasyABC package), delving into the often difficult problem of priors, and so forth. But this recipe should provide a starting point for such explorations.

## 14.7 Tracking true local ancestry along the chromosome

Ancestry, relatedness, pedigree, and similar concepts are often important in forward genetic simulations such as those run by SLiM. Depending upon exactly what you need, there are several different approaches to these sorts of questions in SLiM:

- The `subpopID` property of mutations (section 24.10.1) keeps track of the subpopulation in which a given mutation originated. In an admixture model in which you want to determine whether a given mutation originally arose in one subpopulation or another, this is often all you need. Note also that if you are adding mutations yourself using `addNewMutation()` or `addNewDrawnMutation()`, you can set the `subpopID` in those calls to whatever value you wish. The `subpopID` property is writeable, so you can also change it later to any integer value (including some type of ancestry information).
- Mutation types can also be useful for tracking ancestry information. If different mutation types are used for mutations with different origins in your model, they can then be used to determine the origin of each mutation later, and SLiM's methods for retrieving and counting the mutations of a given mutation type can



be used to separate out mutations of different origins. This approach will work even in non-admixture models in which all mutations originate in a single subpopulation, as long as you can classify mutations according to their ancestry at the point when they originate (in a `modifyChild()` callback that looks at the parents to determine an ancestry group, for example). The mutation type of mutations can be changed with the `setMutationType()` method.

- By enabling an optional feature, SLiM can do pedigree-based tracking of the ancestry of individuals for the purposes of calculating relatedness, finding “trios” of parents and an associated offspring, and so forth (see section 14.1). This provides information about relatedness and ancestry at the level of individuals, rather than the level of mutations. Pedigree-based arranged matings can also be implemented (see section 16.12).
- If tree-sequence recording is turned on (see section 1.7 and chapter 17, and section 17.5 in particular), it provides a type of true local ancestry tracking for every position along the chromosome. This is similar in some ways to this recipe, but much more efficient, and was added in SLiM 3.

Sometimes it is desirable to track relatedness not at the individual level with a pedigree, or at a mutational level with `subpopID` or mutation types, but instead at the level of chromosomal regions – perhaps down to the ancestry of each individual base position along the chromosome. In this way, the ways that assortment, recombination, selection, and drift determine the ancestry at each position can be analyzed. This type of relatedness tracking is also possible in SLiM, even when not using tree-sequence recording; we will explore an example in this section.

Models can implement “true local ancestry” tracking themselves using marker mutations – mutations which have no selective effect, and are not meant to represent actual mutational changes to genomes, but instead simply mark particular positions on particular genomes for future reference. The recipe in section 14.4 used a marker mutation to track individuals possessing a chromosomal inversion, for example, and in section 14.5 a marker mutation was used to mark Y chromosomes in a model of a pseudo-autosomal region shared between sex chromosomes. Here marker mutations will be used at every position, to indicate ancestry. At the end, the model will assess the average ancestry, across the subpopulation, at every chromosome position.

Here is the complete recipe:

```
initialize() {
    defineConstant("L", 1e4); // chromosome length

    initializeMutationRate(0);
    initializeMutationType("m1", 0.5, "f", 0.1); // beneficial
    initializeMutationType("m2", 0.5, "f", 0.0); // p1 marker
    m2.convertToSubstitution = F;
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-7);
}
1 early() {
    sim.addSubpop("p1", 500);
    sim.addSubpop("p2", 500);
}
```

```

1 late() {
    // p1 and p2 are each fixed for one beneficial mutation
    p1.genomes.addNewDrawnMutation(m1, asInteger(L * 0.2));
    p2.genomes.addNewDrawnMutation(m1, asInteger(L * 0.8));

    // p1 has marker mutations at every position, to track ancestry
    p1.genomes.addNewMutation(m2, 0.0, 0:(L-1));

    // make p3 be an admixture of p1 and p2 in the next cycle
    sim.addSubpop("p3", 1000);
    p3.setMigrationRates(c(p1, p2), c(0.5, 0.5));
}
2 late() {
    // get rid of p1 and p2
    p3.setMigrationRates(c(p1, p2), c(0.0, 0.0));
    p1.setSubpopulationSize(0);
    p2.setSubpopulationSize(0);
}
2: late() {
    if (sim.mutationsOfType(m1).size() == 0)
    {
        p3g = p3.genomes;

        p1Total = sum(p3g.countOfMutationsOfType(m2));
        maxTotal = p3g.size() * L;
        p1TotalFraction = p1Total / maxTotal;
        catn("Fraction with p1 ancestry: " + p1TotalFraction);

        p1Counts = integer(L);
        for (g in p3g)
            p1Counts = p1Counts + integer(L, 0, 1,
                g.positionsOfMutationsOfType(m2));
        maxCount = p3g.size();
        p1Fractions = p1Counts / maxCount;
        catn("Fraction with p1 ancestry, by position:");
        catn(format("%.3f", p1Fractions));

        sim.simulationFinished();
    }
}
100000 late() {
    stop("Did not reach fixation of beneficial alleles.");
}

```

The broad outline here is straightforward. We model two subpopulations, p1 and p2, which are created in the tick 1 event and configured in the tick 1 late() event. That late() event also sets up a new subpopulation, p3, that will be produced by admixing migrants from p1 and p2 in tick 2. At the end of tick 2, p1 and p2 are removed, and p3 is allowed to evolve thenceforth until the termination of the model. This model is therefore a very simple model of the admixture of two subpopulations; more complex population dynamics could easily be used with the same ancestry-tracking mechanism shown here.

When p1 and p2 were configured, they were each set up to contain a beneficial mutation of type m1, fixed within the respective subpopulation. The new p3 subpopulation therefore contains some genomes that originated in p1 and contain that subpopulation's beneficial mutation (located at  $L * 0.2$  on the chromosome), and some genomes that originated in p2 and contain that subpopulation's beneficial mutation (located at  $L * 0.8$  on the chromosome). The expectation is

that recombination between those points will eventually produce at least one new genome that contains both beneficial mutations, and that subsequently both beneficial mutations will sweep to fixation in p3. The question we wish to investigate is the precise pattern of true local ancestry along the chromosome at the point when fixation of both beneficial mutations has just occurred.

To do this, the model adds a marker mutation of type m2 at every position along every genome in p1, in the 1:late() event. This is done using a single vectorized call to addNewMutation(), for greater efficiency; adding the mutations one by one would take a very long time, especially with a longer chromosome. These added markers indicate that a given chromosome position traces its ancestry to p1. In models with more complex demography involving more than two “parental” subpopulations, additional mutation types could be employed to indicate derivation from each of the possible ancestral subpopulations. An important point to understand is that because mutations in SLiM “stack” by default, these marker mutations have no effect whatsoever on the other mutations that might occur during a simulation; they are simply carried along, almost as if they were epigenetic marks of some sort on the genomes being simulated. For simplicity, this model does not include neutral mutations along the chromosome, and indeed it uses a mutation rate of 0; but the ancestry-tracking mechanism shown here is compatible with any other model dynamics.

Thus configured, the model runs until both beneficial mutations either fix or are lost. This termination condition is checked in the 2:late() event. When there are no longer any circulating m1 mutations, the model outputs some final analysis and terminates. The final analysis here is in two parts. The first part simply calculates the fraction of all chromosome positions in all genomes that derives from p1. In one run of this model, the output happens to be:

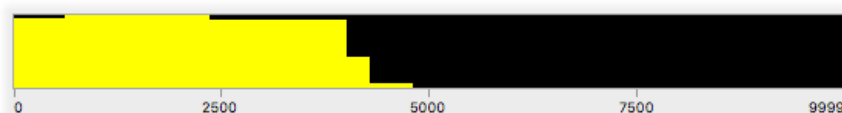
Fraction with p1 ancestry: 0.398137

The second part calculates the fraction of all genomes derived from p1 at each individual position along the chromosome, and outputs those fractions as a big vector:

Fraction with p1 ancestry at each chromosome position:  
0.946 0.946 0.946 0.946 0.946 0.946 0.946 0.946 0.946 0.946 0.946 0.946...

It does that by looping through the genomes of p3, and for each genome getting the positions at which m2 mutations exist using positionsOfMutationsOfType(). Those vectors of positions can be converted into integer vectors that have a 1 at the positions where m2 mutations occurred (and a 0 everywhere else), using the integer() function of Eidos. Summing those vectors across all of the p3 genomes gives a count of the number of m2 mutations at each position; dividing that by the maximum possible count gives a frequency, which indicates the pattern of ancestry.

This pattern can also be seen in SLiMgui, with the neutral marker mutations shown in yellow:

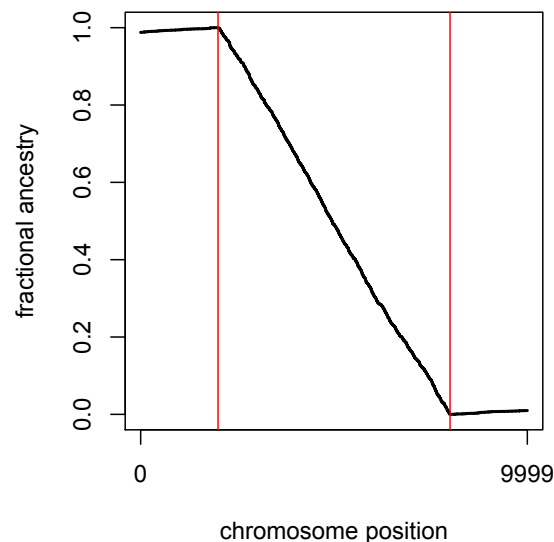


This indicates that the left-hand half of the chromosome mostly derives from p1 – albeit with some variation due to multiple recombination events that created some haplotype variation – whereas the right-hand half of the chromosome derives from p2 in all individuals.

The recipe here tracks true local ancestry down to the level of individual base positions. This does involve some overhead, in terms of both memory usage and processor power, so if this level of granularity is not needed, it would be much more efficient to place marker mutations every 10, 100, or even 1000 positions along the chromosome. This could be done with a few trivial modifications to the recipe presented here. However, this recipe can be scaled up to  $L = 1$  Mbp and still run in only a couple of minutes, so the overhead is really not too bad. For even larger

models, however, both the runtime and the memory usage of this recipe can become prohibitive; at  $L = 10^8$ , the memory usage of this recipe is estimated to be over 8 TB (that is not a typo).

A final note: having done one run of the recipe and seen the pattern of ancestry in SLiMgui, it would be natural to wonder what the average pattern of ancestry would be across many runs. That is straightforward to do, by simply running the model many times, collecting the data from each run, and averaging the ancestry pattern across the runs. A simple R script to do this is provided in the online SLiM-Extras repository, in the “sublaunching” folder since it illustrates how to sublaunch runs of SLiM from a script (in this case, an R script). The script is named `aggregateLocalAncestry.R` (<https://github.com/MesserLab/SLiM-Extras/blob/master/sublaunching/aggregateLocalAncestry.R>). If you have R installed you should be able to run it easily, after modifying the file paths in the script to point to your installed `slim` and your copy of this recipe. If you do that, you will get a plot:



The locations where the beneficial mutations were placed in `p1` and `p2` are shown with red lines. The mean true local ancestry (the fraction of the genomes in `p3` that derive from `p1` at a given chromosome position) is exactly **1.0** at the first beneficial mutation’s location, and exactly **0.0** at the second’s; this makes sense, since the model did not terminate until both beneficial mutations had fixed, which would imply pure `p1` or `p2` ancestry at those positions. In between, the local ancestry changes linearly from `p1` to `p2`; it might seem plausible that the pattern would be sigmoid instead, but apparently it is not. Toward the ends of the chromosomes, outside the beneficial mutations, the ancestry falls toward **0.5**; it’s hard to see here, but that fall-off is not linear, in fact. This pattern presumably recapitulates known results from population genetics, but a modified version of this model could explore much more complex scenarios.

This recipe is interesting from a historical perspective, and the approach it shows might prove useful in some circumstances, but it has largely been eclipsed by tree-sequence recording. Section 17.5 gives a recipe to solve essentially the same problem, using the ancestry information provided by the recorded tree sequence. As that section shows, assessing the true local ancestry for a chromosome of length  $1e8$ , which would take this section’s recipe over 7.2 days to complete, can be done in slightly over one minute with tree-sequence recording.

## 14.8 Live plotting with R using `system()`

The previous section presented a final analysis of the results from its recipe using a plot generated in R after the model had finished. SLiMgui provides some built-in plots, as we saw in

chapter 8, but it would be awfully nice to be able to make custom plots through R's extensive plotting facilities within a running SLiM model – and even better, to have those plots update live as the model runs in SLiMgui. That is the goal of this section's recipe. (This technique is so useful, however, that we already used it in section 13.5; here we will introduce it more thoroughly.)

To achieve this goal, we will use several tools we haven't encountered before now. Most important is the `Eidos system()` function, which allows any Un\*x command to be executed; we will use it to sublaunch R processes that will generate plots for us. We will also use the `Eidos writeTempFile()` function, which generates a temporary file with a unique, non-colliding filename; we will use it to create the R script we will execute and the PNG plot file we will display. (Note that the version of the model provided in SLiMguiLegacy generates a PDF plot file instead, since the SLiMguiLegacy app can display PDF files but not PNG files.)

Let's launch into it:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);

  if (fileExists("/usr/bin/Rscript"))
    defineConstant("RSCRIPT", "/usr/bin/Rscript");
  else if (fileExists("/usr/local/bin/Rscript"))
    defineConstant("RSCRIPT", "/usr/local/bin/Rscript");
  else
    stop("Couldn't find Rscript.");
}
```

The first part is just a vanilla neutral model setup, nothing surprising. The second part looks up the location of the `Rscript` utility, which is a command-line tool that executes a given R script; it is typically part of a standard install of R. We will use this tool later on, so here we figure out where it is, by just looking at two likely locations. If `Rscript` is located somewhere else, you can tailor this code as needed. Executing `which Rscript` in a terminal window will probably tell you where `Rscript` is located for you. If it seems to be missing, you probably need to install R on your machine. On macOS you can download a double-click installer from CRAN; on Linux, depending on your distro, executing `sudo apt-get install r-base-core` might install it for you.

Next is the tick 1 setup:

```
1 early() {
  sim.addSubpop("p1", 5000);
  sim.setValue("fixed", NULL);

  defineConstant("pngPath", writeTempFile("plot_", ".png", ""));

  // If we're running in SLiMgui, open a plot window
  if (exists("slimgui"))
    slimgui.openDocument(pngPath);
}
50000 late() { sim.outputFixedMutations(); }
```

In tick 1, we create a new subpopulation as usual. We also start a new value kept by the simulation, named `fixed`; this will keep a record of the number of fixed mutations over time as the simulation runs, and that is what we will plot.

Next we call `writeTempFile()` to make a temporary file with a filename that begins with "plot\_" and ends with ".png"; several characters in between will be randomly generated by `writeTempFile()` to create a unique filename that is not presently used in the temporary directory where the file will be placed (/tmp or a subdirectory thereof, on all Unix platforms including macOS). This call returns the filesystem path to the file it creates, and we define that as a constant named `pngPath` for future use.

The last couple of lines execute only if the model is running in SLiMgui; when running at the command line, the `slimgui` object does not exist. When running in SLiMgui, on the other hand, `slimgui` is a global singleton object (of Eidos class `SLiMgui`) that represents the SLiMgui application itself, and can be used to control SLiMgui from a model's script (see section 24.13). If the `slimgui` object exists, as tested by the Eidos function `exists()`, then we are indeed running in SLiMgui, in which case we call the `openDocument()` method of `slimgui` with the path to the PNG file we have just created. A new window should open in SLiMgui as a result; initially it will be blank since the PNG file is empty, but it will update automatically whenever the PNG file changes.

Since we are running in SLiMgui, we can rely upon it for the automatically updating PNG display facility used here. When running SLiM at the command line, you can do something very similar to open the plot file in the PNG preview app of your choice, and get live plotting even without running SLiMgui. However, many image display programs do not notice changes to the file they are displaying, and will not automatically redisplay it. Graham Gower, a SLiM user on Linux, has kindly supplied two modified versions of this recipe that demonstrate approaches to this on Linux (based on PDF rather than PNG, for historical reasons); they are available in the SLiM-Extras repository on GitHub (<https://github.com/MesserLab/SLiM-Extras>). One of his recipes uses `xdg-open` to open the plot image in the user's preferred viewer (which may or may not handle the redisplay issue). The other opens the plot image in a specific viewer app called `mupdf`, and sends it HUP signals to cause it to redisplay with each replot in SLiM. Their code should be fairly self-explanatory.

Finally, we have a tick `50000` event that ends the simulation. All we need in addition to that code is an event to tabulate results and generate plots:

```
1: early() {
  if (sim.cycle % 10 == 0)
  {
    count = sim.substitutions.size();
    sim.setValue("fixed", c(sim.getValue("fixed"), count));
  }

  if (sim.cycle % 1000 != 0)
    return;

  y = sim.getValue("fixed");

  rstr = paste('{',
    'x <- (1:' + size(y) + ') * 10',
    'y <- c(' + paste(y, sep=", " + ')',
    'png(width=4, height=4, units="in", res=72, file="' + pngPath + '")',
    'par(mar=c(4.0, 4.0, 1.5, 1.5))',
    'plot(x=x, y=y, xlim=c(0, 50000), ylim=c(0, 500), type="l",',
    'xlab="Generation", ylab="Fixed mutations", cex.axis=0.95,',
    'cex.lab=1.2, mgp=c(2.5, 0.7, 0), col="red", lwd=2,',
    'xaxp=c(0, 50000, 2))',
    'box()',
    'dev.off()',
    '}', sep="\n");
```



```

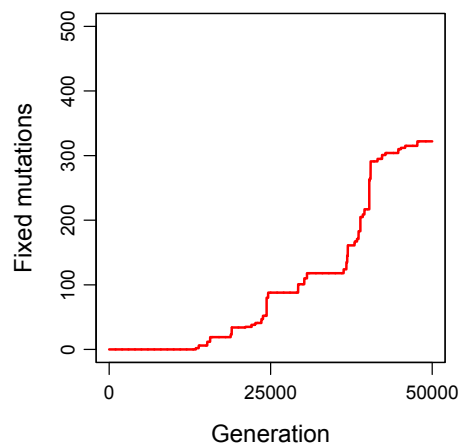
scriptPath = writeTempFile("plot_", ".R", rstr);
system(RSCRIPT, args=scriptPath);
deleteFile(scriptPath);
}

```

First of all, in the event code above, every tenth cycle we add the current count of fixed mutations to the value we're using to tabulate those results, using `getValue()` and `setValue()` to update the saved value.

Second, every 1000<sup>th</sup> cycle we generate a new plot. First, we get the current tabulation of counts using `getValue()`. Then we assemble an R script into `rstr` using a rather complicated command that spans twelve lines. (Note that if there is *any* typo in that code, the `Rscript` call will fail silently and no plot will be produced; retyping this code by hand is not recommended, use copy/paste or simply open this recipe from the Open Recipe submenu in SLiMgui.) A call to `c()` is used to piece together a vector of script lines, some of which are simple strings, others of which are themselves assembled using the `+` operator to concatenate strings and so forth. The vector of script lines is then pasted together using a newline character as the separator, to make a more or less readable R script (you could add a `print()` call to see the final result if you wish). We write that script to a temporary file using `writeTempFile()`, and then we call `system()` to request that Rscript should run our script. Once Rscript is done we clean up the script file.

That is all that is needed. SLiMgui will automatically notice that the PNG file has been overwritten with new data, and will redisplay it more or less continuously (there is a short lag because of delays involved in filesystem notifications and other factors, but typically less than a second). At the end of a typical run, the plot window in SLiMgui shows something like this:



The plotting code that we sent to R included niceties like axis labels, font sizes, and line colors, so the final plot is reasonably nice-looking. Beyond aesthetics, we can see a few interesting things in this plot. First of all, there is a long delay – perhaps 15000 generations – before any mutations fix at all. This is because the chromosome starts empty, in this model, and it takes some time to accumulate neutral diversity and have it drift to fixation. This is why it is usually important to run models for a burn-in period before collecting data; the early state of a model is usually far from equilibrium. Second, even after mutations start to fix the line is quite jagged; there are long periods without any fixation, and then sudden jumps when many mutations fix simultaneously. This is because whole haplotypes representing many individual mutations often fix as a single unit, followed by periods in which competing haplotypes are drifting up and down in frequency without fixation; these evolutionary dynamics are also very visible in SLiMgui's chromosome view.

Apart from a little bit of hassle involved in assembling an R script as an Eidos string, this recipe is quite short and simple considering that it is continuously writing new script files and then sublaunching R processes to generate PNG plots! This example is quite straightforward, but there is no limit to the complexity possible here; arbitrary plots could be made, multiple plots could be generated simultaneously, and other processing, such as statistical analysis, could also be done in R while a SLiM simulation is still running. Of course this model could also be adapted to sublaunch Python instead, for those who prefer it. Sublaunching is very general and powerful.

## 14.9 Modeling haploid organisms

SLiM models diploid individuals that contain two haploid genomes; this is, at present, a design constraint in SLiM that cannot be modified. However, it is still possible to model haploids in SLiM quite easily with scripting, by effectively suppressing one of the two haploid genomes of each diploid individual. Section 14.5's recipe, which shows how to model X and Y chromosomes with a pseudo-autosomal region, shares some aspects of its design with the recipe shown here, but modeling haploids is actually much simpler. Similar techniques could be used to model mitochondrial DNA, or alternative mating systems (but nonWF models provide better methods; see section 16.16 and 16.24 for nonWF models of haplodiploidy and alternation of generations).

In any case, here we will stick with a simple model of haploids that reproduce clonally. This model is short enough that we will just present it in its entirety:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 1.0, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(0);
}
1 early() {
  sim.addSubpop("p1", 500);
  p1.setCloningRate(1.0);
}
late() {
  // remove any new mutations added to the disabled diploid genomes
  sim.subpopulations.individuals.genome2.removeMutations();

  // remove mutations in the haploid genomes that have fixed
  muts = sim.mutationsOfType(m1);
  freqs = sim.mutationFrequencies(NULL, muts);
  if (any(freqs == 0.5))
    sim.subpopulations.genomes.removeMutations(muts[freqs == 0.5], T);
}
200000 late() {
  sim.outputFixedMutations();
}
```

We begin with a typical `initialize()` callback, except that the recombination rate is set to zero since there is no recombination in this clonal model. Note also that although this is a neutral model for simplicity, the dominance coefficient is set to 1.0 on mutation type `m1` as a reminder; since mutations will never be homozygous in this model, dominance coefficients should always be 1.0 for conceptual clarity. Next we add a new subpopulation in tick 1 as usual; but in addition, we set the subpopulation to reproduce clonally, as befits haploids (but see section 16.14). At the end of the model we output fixed mutations, as a placeholder for whatever output would be desired. The interesting model mechanics occur in between, in the `late()` callback.

The first part of the `late()` callback removes mutations in the second haploid genome of each new child generated. SLiM doesn't know that we're modeling haploids, and thus not using the second genomes, so it adds new mutations to them as usual during offspring generation. This call removes them again in order to keep all of the second genomes of individuals empty.

The second part of the `late()` callback solves another problem: removing fixed mutations. SLiM automatically converts fixed mutations into `Substitution` objects and removes them from the simulation; however, it defines fixation as occurring when the frequency of a mutation reaches 1.0. In this haploid model, mutations fix when they reach a frequency of 0.5, because of the empty second genomes. We therefore need to remove mutations manually, rather than relying on SLiM's built-in machinery. This callback achieves that, passing `T` for the `substitute` parameter of `removeMutations()` so that `Substitution` objects are created for the fixed mutations as usual.

Those are the only overrides needed in script to produce a model of haploids. The mechanisms used here to enforce haploidy, such as removal of mutations on the unused chromosome and substitution of fixed mutations at a frequency of 0.5, could also be used to make just one segment of the simulated chromosome haploid. This would allow mitochondrial DNA and autosomal DNA to be jointly simulated in SLiM, for example. Some modifications to the recipe above would be needed to achieve this; you would probably want to use a sexual autosomal model with biparental mating, add code to the `modifyChild()` callback to enforce maternal inheritance of the mitochondrial DNA (i.e., to block any proposed child that received its mitochondrial DNA from the father, or both parents, or neither), and use a recombination map to prevent recombination in the mitochondrial portion of the genome, for example. Section 14.5 provides an example of a scenario that is in many ways parallel to that. Other ploidy schemes, such as haplodiploidy and alternation of generations, should also be implementable with modifications of these basic ideas, although a nonWF approach is usually better. Even tetraploidy and higher ploidy levels can be modeled, but that is more complex; a recipe for that may be found in the SLiM-Extras repository.

For an alternative model of haploid organisms that is more compatible with tree-sequence recording, but which requires the nonWF model type, see sections 16.13 and 16.14.

## 14.10 Using mutation rate variation to model varying functional density

Beginning with SLiM 2.5, it is possible to set up a mutation-rate map that varies the mutation rate along the length of the chromosome. This is similar to setting up a recombination-rate map, as already supported by SLiM; see sections 6.1.1 and 6.1.2. A variable mutation-rate map can, of course, be used for the obvious purpose of configuring mutational “hot” and “cold” spots along the chromosome, and the code for doing so would look much like the code for those recipes. Here, we will explore a less obvious use: modeling positional variation in functional density.

We know that different regions of chromosomes often have higher or lower functional density, as a consequence of variation in the density of genes and the importance of those genes. Regardless of the literal appropriateness of the term “junk DNA”, it is clear that large chromosomal regions exist that are non-coding, and mutations in these regions generally appear to have relatively little effect on fitness compared to mutations in coding regions. This is quite orthogonal to mutational “hot” and “cold” spots; the variation we are concerned with here is not in the mutation rate *per se*, but in the rate at which mutations actually influence fitness.


Prior to SLiM 2.5, modeling this would have required that one define a complex map of genomic elements along the chromosome, all experiencing the same mutation rate (since that could not be varied), but using a different fraction of neutral mutations to deleterious mutations (and/or beneficial mutations, but for simplicity we will here focus on deleterious mutations). This design would have been necessary in order to achieve the desired variation in the rate of

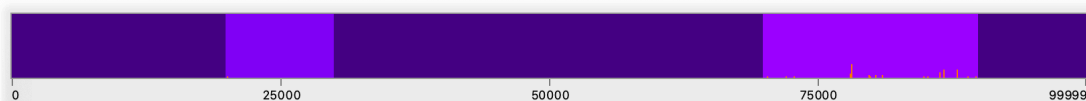
deleterious mutations, even if one was not actually interested in the neutral mutations at all. Such a model would run much more slowly than necessary because of the neutral mutation overhead.

By defining a mutation-rate map, however, it is now straightforward to build a model in which functional density varies along the chromosome, without having to model the neutral mutations. A proof-of-concept model for this is so simple as to be trivial:

```
initialize() {
  initializeMutationType("m1", 0.5, "f", -0.01); // deleterious
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);

  // Use the mutation rate map to vary functional density
  ends = c(20000, 30000, 70000, 90000, 99999);
  densities = c(1e-9, 2e-8, 1e-9, 5e-8, 1e-9);
  initializeMutationRate(densities, ends);
}
1 early() {
  sim.addSubpop("p1", 500);
}
200000 late() { sim.outputFixedMutations(); }
```

The variables `ends` and `densities` are set up to encode the desired mutation-rate map, with the end position for each chromosomal range and the effective functional density – the rate of mutations having a deleterious effect, in this model – of that chromosomal range. As the recipes of sections 6.1.1 and 6.1.2 illustrate, such maps can easily be generated randomly based upon empirical metrics, or can even be read in from a file with empirical map data; in this model, to keep things simple, we instead just specify a very simple map with five regions of different functional density. After the model has been initialized, clicking the  button to the right of the chromosome view in SLiMgui and choosing Display Rate Maps will show the mutation-rate map (we have seen that option show the recombination-rate map before; it shows whichever map has been defined, or both if both have been defined):



The highest rate is in the region from base position 70000 to 89999, and indeed quite a few deleterious mutations can be seen in that region (but none at very high frequency; the parameters used in this model mean that the deleterious mutations are eliminated fairly efficiently). There is a less active region from 20000 to 30000, with one mutation in this snapshot; the rest of the chromosome has a lower functional density, and receives deleterious mutations relatively rarely.

In this recipe we work with only one mutation type, modeling deleterious mutations. It would be possible to extend it to include several types of functional mutations, each with a different rate of occurrence along the chromosome. In that case, the mutation-rate map would be set to encode the sum of the rates for each of the mutation types in each chromosomal region, and genomic elements would be used to partition mutations in each region into the correct fraction for each of the functional mutation types. This would be somewhat more complex than this recipe, but manageably so; and it would again be much more efficient than using a constant mutation rate along the chromosome together with a varying fraction of neutral mutations, since modeling all of the neutral mutations could again be avoided.

## 14.11 Modeling microsatellites

A microsatellite (also known as a *short tandem repeat* or *simple sequence repeat*) is a chromosomal region in which a specific nucleotide sequence repeats, often many times. Microsatellites are important in many areas of applied genetics, from kinship analysis to forensics, and are also often used to assess the similarity among individuals or subpopulations in ecology and evolutionary biology. It is thus useful to be able to include them in evolutionary models. Since explicitly modeling mutations that change the number of repeats in a microsatellite would change the length of the chromosome, which is not allowed in SLiM, explicitly modeling the repeated nucleotide sequence of a microsatellite doesn't fit well into SLiM's conceptual model. Nevertheless, it is possible to model microsats more abstractly in SLiM, using mutations that conceptually represent a microsat with a particular number of repeats at a given locus. This recipe will illustrate one simple approach to this.

We will build this model step by step. First, the model initialization:

```
initialize() {
  defineConstant("L", 1e6);           // chromosome length
  defineConstant("msatCount", 10);    // number of microsats
  defineConstant("msatMu", 0.0001);   // mutation rate per microsat
  defineConstant("msatUnique", F);    // T = unique msats, F = lineages

  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);

  // microsatellite mutation type; also neutral, but magenta
  initializeMutationType("m2", 0.5, "f", 0.0);
  m2.convertToSubstitution = F;
  m2.color = "#900090";
}
```

We define the length of the chromosome (L), and several constants related to our microsats: the number of microsats we will model, the mutation rate per microsat (per gamete), and a flag that indicates whether to unique the microsats as the model runs (more on this below). The rest sets up a simple neutral model structure, plus a mutation type, m2, that will represent our microsatellites. We prevent microsatellites from fixing, with `convertToSubstitution`, since they are a permanent feature of the chromosomal structure. We also set them to display in SLiMgui using a shade of magenta, to set them apart from the other neutral mutations in the model.

Next we create a subpopulation with microsatellites:

```
1 late() {
  sim.addSubpop("p1", 500);

  // create some microsatellites at random positions
  genomes = sim.subpopulations.genomes;
  positions = rdunif(msatCount, 0, L-1);
  repeats = rpois(msatCount, 20) + 5;

  for (msatIndex in 0:(msatCount-1))
  {
    pos = positions[msatIndex];
    mut = genomes.addNewDrawnMutation(m2, pos);
    mut.tag = repeats[msatIndex];
  }
}
```

```

        // remember the microsat positions for later
        defineConstant("msatPositions", positions);
    }

```

We call `addSubpop()` to create the subpopulation, and then we create our microsatellites. Each microsat is simply an `m2` mutation at a given position; the number of repeats in a given microsat is represented using the `tag` property of the mutation. This code draws the positions and repeats for all of the microsats first, and then loops to create each microsat using that information. Finally, we remember the positions of the microsats in a defined constant for later use.

This code makes the initial population homogenous: the number of repeats of a given microsat is the same across all genomes. It would of course be possible to start with a non-homogenous state instead. To avoid creating a new `Mutation` for every microsat in every genome, however, it would be best to determine all of the genomes containing a given repeat count at a given position, and then call `addNewDrawnMutation()` just once on that entire genome vector so as to create a single `Mutation` object that is shared among all of those genomes. Alternatively, a uniquing strategy could be employed, similar to what we will see below, such that the first microsatellite created at a given position with a given number of repeats is instantiated with a new `Mutation`, and then all subsequent microsats with the same number of repeats at the same position look up and use that existing `Mutation` object. Such extensions are left as an exercise for the reader.

With the initial subpopulation state set up, let's define an endpoint for the model:

```

10000 late() {
    // print frequency information for each microsatellite site
    all_msats = sim.mutationsOfType(m2);

    for (pos in sort(msatPositions))
    {
        catn("Microsatellite at " + pos + ":");

        msatsAtPos = all_msats[all_msats.position == pos];

        for (msat in sortBy(msatsAtPos, "tag"))
            catn("    variant with " + msat.tag + " repeats: " +
                sim.mutationFrequencies(NULL, msat));
    }
}

```

This output callback loops over the (sorted) positions of the microsats. For each microsat position, it looks up all of the microsats that exist at that position, and outputs frequency counts for each variant (sorted by repeat count).

We'll see some example output below, but the model isn't finished yet; let's finish it first. The remaining piece in the puzzle is for our microsatellites to mutate. Microsats mutate in an interesting way: they add or remove repeats. Furthermore, the probability of this happening is much higher than the usual mutation rate in most organisms – often as high as one in ten thousand. We model all this with a special `modifyChild()` callback:

```

modifyChild() {
    // mutate microsatellites with rate msatMu
    for (genome in child.genomes)
    {
        mutCount = rpois(1, msatMu * msatCount);
    }
}

```



```

    if (mutCount)
    {
        mutSites = sample(msatPositions, mutCount);
        msats = genome.mutationsOfType(m2);

        for (mutSite in mutSites)
        {
            msat = msats[msats.position == mutSite];
            repeats = msat.tag;

            // modify the number of repeats by adding -1 or +1
            repeats = repeats + (rdunif(1, 0, 1) * 2 - 1);

            if (repeats < 5)
                next;

            // if we're uniquing microsats, do so now
            if (msatUnique)
            {
                all_msats = sim.mutationsOfType(m2);
                msatsAtSite = all_msats[all_msats.position == mutSite];
                matchingMut = msatsAtSite[msatsAtSite.tag == repeats];

                if (matchingMut.size() == 1)
                {
                    genome.removeMutations(msat);
                    genome.addMutations(matchingMut);
                    next;
                }
            }

            // make a new mutation with the new repeat count
            genome.removeMutations(msat);
            msat = genome.addNewDrawnMutation(m2, mutSite);
            msat.tag = repeats;
        }
    }
}

return T;
}

```

There's a lot to parse there; let's take it step by step. First of all, the microsats in each genome in the proposed child mutate independently, so we loop over the microsats and mutate them each separately. For each genome, we draw the number of mutations that occur from a Poisson distribution; this is faster than doing a separate random draw to determine the fate of each individual microsat, but is otherwise essentially equivalent. If the number of microsat mutations is zero, we're done; we loop back to handle the other genome, and then we're finished and return T. When the number of microsat mutations is greater than zero, though, we have work to do.

In that case, the next thing we do is to draw the positions of the microsats that mutated, using `sample()` to draw from the vector of positions we defined when we set up the simulation. Each microsat thus has an equal probability of mutating (see below for discussion of this). We loop over those positions and mutate each chosen microsat in turn. To mutate a microsat, we first look up the existing mutation by position and find out how many repeats it has. We then decide what the new, mutated repeat count will be; here we just add or subtract 1, and limit the repeat count to a minimum of 5, but more sophisticated and realistic dynamics could be introduced. Finally,

ignoring the “if (msatUnique)” section for a moment (since msatUnique is presently defined as F anyway), we effect the mutation event by removing the old microsat mutation from the genome and adding a newly created microsat mutation at the same position, with the new repeat count for its tag. (We can’t simply set the tag of the existing microsat mutation, because it is potentially shared among many genomes; changing its tag would change the repeat count in all of those genomes!)

Running this model produces output like:

```
Microsatellite at 60933:
  variant with 21 repeats: 0.001
  variant with 21 repeats: 0.026
  variant with 21 repeats: 0.001
  variant with 22 repeats: 0.741
  variant with 23 repeats: 0.105
  variant with 23 repeats: 0.126
Microsatellite at 98509:
  variant with 34 repeats: 1
Microsatellite at 123123:
  variant with 20 repeats: 0.964
  variant with 21 repeats: 0.001
  variant with 21 repeats: 0.035
Microsatellite at 142781:
  variant with 23 repeats: 0.795
  variant with 24 repeats: 0.205
...
```

This looks reasonable, except that more than one variant with the same repeat count sometimes exists for a given microsatellite. Each such variant represents an independent mutational lineage; each time a microsat mutation occurs, it is represented by a new `Mutation` object that SLiM tracks forevermore. In some cases, keeping track of each mutational lineage may be desirable; it could provide a sort of ancestry tracking, for example, that goes beyond what the repeat counts alone would provide. Often, however, one would like such replicate mutational lineages to be merged, such that just a single microsat mutation exists to represent a microsat with a given number of repeats at a given position. This is what that “if (msatUnique)” code, which we skipped over above, is for: it “uniques” the microsatellite lineages. To see the effect of this, let’s change that flag by modifying its definition in the `initialize()` callback:

```
defineConstant("msatUnique", T);    // T = unique msats, F = lineages
```

If we run the model again (using the same random number seed, so as to reproduce the same evolutionary dynamics), the output now looks like this:

```
Microsatellite at 60933:
  variant with 21 repeats: 0.028
  variant with 22 repeats: 0.741
  variant with 23 repeats: 0.231
Microsatellite at 98509:
  variant with 34 repeats: 1
Microsatellite at 123123:
  variant with 20 repeats: 0.964
  variant with 21 repeats: 0.036
Microsatellite at 142781:
  variant with 23 repeats: 0.795
  variant with 24 repeats: 0.205
...
```

The microsats have been uniqued, as desired. The “if (msatUnique)” code in the callback, which implements this feature, is actually quite straightforward. It gets a vector of all existing microsat mutations from the simulation, and then narrows that down to those at the desired position, and then finally down to those with the desired number of repeats. If it found an exact match, then it removes the existing microsat mutation and adds the match; the next statement then loops forward to the next mutation site, if any. If it did not find a match, the code drops through to add a newly created mutation instead; that is the case we saw before.

That’s all that’s needed: a little initialization code, a little output code, and a `modifyChild()` callback to handle mutating the microsats. Although that `modifyChild()` callback is not particularly short, conceptually it is really quite simple: decide on how many microsat mutation events there will be in each genome and where they will be, and then change the mutations in the genome to reflect those mutation events with new (or uniqued) mutations that have the appropriate repeat count in their `tag` values.

There are a few ways in which greater realism could be added to this model. We already discussed the possibility of starting with a heterogenous subpopulation state; any initial state could be set up, although care should be taken to unique the microsat mutations, to keep memory usage down and to provide a single mutational lineage for each unique microsatellite state.

Another way in which this model oversimplifies the biology is in its mutational model. In reality, microsats can sometimes mutate by adding or subtracting more than a single repeat, and their probability of mutating at all can depend upon the number of repeats present, as well as upon things like the sex of the parent. The mutation rate for microsats can even depend upon the similarity or dissimilarity in repeat counts between the two copies of the microsat in the parent that generated the gamete, due to effects during meiosis. All of these effects could be modeled with extensions to the `modifyChild()` callback presented here. To implement this, it would probably be necessary (or at least simpler) to get rid of the `rpois()` call that draws the number of microsat mutations for each genome, and instead simply loop over all of the existing microsat positions and determine whether a mutation occurred at each position with a `runif()` draw that is compared to the calculated probability of mutation for that microsat in that genome.

A third oversimplification is that this recipe does not explicitly track variation in microsats due to point mutations rather than repeat-count mutations. Perhaps a good approach for this would involve modeling repeat-count changes just as in this model, while adding in similar code that would model nucleotide changes as well (using an appropriate mutation probability based on the length of the microsat). If such a model was run *without* uniquing, each mutational lineage would then be tracked separately, and so point mutations would create new mutational lineages that would be considered distinct from other mutation lineages with the same repeat count. However, this design would overcount the number of genetically distinct lineages, since repeat-count mutations as well as point mutations would generate distinct mutational lineages. To account for things better would require additional state information to be attached to each `Mutation` object, using the `setValue()/getValue()` mechanism; in this way, two mutational lineages with the same repeat count and no nucleotide differences could be merged together by the uniquing code, while two mutational lineages with the same repeat count but *with* nucleotide differences could be kept separate. In the extreme of maximal realism, one could actually store generated string representations of nucleotide sequences inside the `Mutation` objects, but that would be overkill for most purposes. Usually, a single token attached to each mutation, such as a random number generated with `runif()`, could probably provide sufficient realism. Whenever a point mutation occurred, a new mutation object would be created (with no uniquing necessary) and would be given a new random token representing its unique new nucleotide sequence. Whenever a repeat-count mutation occurred, on the other hand, the token value would be inherited from the old microsat mutation at that position, indicating that the nucleotide difference, whatever it might be,

was (assumed to be) preserved across the repeat count change. If the uniquing code then re-used an existing mutation only when the token values of the desired mutation and the existing mutation were the same, a closer approximation of the correct pattern of microsatellite diversity might emerge. Implementing this goes beyond the scope of this recipe, and so is left as an exercise for the reader, but in essence it is actually quite simple; the design simply adds another piece of state, tracked with `setValue()/getValue()`, that is handled very similarly to the `tag` value in the existing recipe. That token value is inherited (just as `tag` is), changes its value upon mutation (just like `tag`, but for point mutation events instead of repeat-count mutation events), and must be equal in order for uniquing to find a match (just as `tag` is used by the existing uniquing code). Token values would be set up when the subpopulation is initialized, just as `tag` values are, either with zero values (representing a homogenous initial state) or with some sort of population structure indicated by different random token values on different genomes to represent standing variation in microsatellite diversity.

Depending upon the level of realism desired, therefore, this recipe might provide a complete solution, or it might merely point the way. In either case, the basic strategy outlined here of using `tag` values to indicate repeat counts on mutations that represent a microsatellite at a given locus is the recommended approach in SLiM. In general, using mutations to represent some conceptual difference between genomes, rather than necessarily representing a literal nucleotide difference, can be a useful strategy for advanced models in SLiM, as similarly shown by section 14.4's recipe that tracks genomes that contain an inversion using a marker mutation.

## 14.12 Modeling transposable elements

A transposable element, or *transposon*, is a chromosomal region which is capable of replication or positional change within the genome, either on its own or with the assistance of an enzyme such as reverse transcriptase or transposase. Transposons constitute a substantial fraction of the genome of many species, and can have evolutionary effects through side effects such as disabling genes into which they “jump”, altering the regulation of nearby genes, and copying genetic material within the genome. Given their evolutionary importance, incorporating transposons into evolutionary models may be useful; in this recipe we will thus explore a simple transposon model.

There are various subtypes and classifications of transposons; here we will explore autonomous Class I transposons, which “copy and paste” themselves to new locations in the genome under their own power. Given the diversity of evolutionary effects transposons can have, and the diversity of ways in which they can function, this recipe cannot provide a general formula for modeling transposons and all of their effects; all this recipe will do is point the way. At the end of this section, we will discuss ways in which this recipe could be extended to model further aspects of the behavior of transposons.

We will not model the actual nucleotide sequence of transposons here. Furthermore, since copying a transposon to a new location changes the length of the chromosome (which is not possible in SLiM), we will model transposons conceptually rather than literally, as loci in the genome that are capable of copying themselves. This approach is similar to the approach taken in section 14.11 for modeling microsatellites.

We will also model the vulnerability of transposons to mutations that disable them by deactivating their ability to jump; we will track disabled transposons, and assess the fraction of each transposon that has been disabled. This is important, since even disabled transposons may have evolutionary effects such as changes in gene regulation, and since most of the transposons in typical organisms appear to be disabled.

We will start with the model's initialization:

```
initialize() {
  defineConstant("L", 1e6);           // chromosome length
  defineConstant("teInitialCount", 100); // initial number of TEs
  defineConstant("teJumpP", 0.0001);   // TE jump probability
  defineConstant("teDisableP", 0.00005); // disabling mut probability

  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);

  // transposon mutation type; also neutral, but red
  initializeMutationType("m2", 0.5, "f", 0.0);
  m2.convertToSubstitution = F;
  m2.color = "#FF0000";

  // disabled transposon mutation type; dark red
  initializeMutationType("m3", 0.5, "f", 0.0);
  m3.convertToSubstitution = F;
  m3.color = "#700000";
}
```

This defines the chromosome length and a few constants governing the behavior of TEs in the model. It then sets up a simple neutral simulation, and defines two additional mutation types: one for active TEs, m2, which displays as red in SLiMgui, and one for TEs that have been disabled by mutation, m3, which display as a darker red.

Next we set up the initial state of our subpopulation. In this recipe, the tag values on the m2 and m3 mutations are used as identifiers; each TE gets its own unique tag value, which is used for both its active (m2) and disabled (m3) forms, allowing the mutations representing the two forms to be matched up. The tag value on the simulation itself, sim.tag, is used to keep track of the next unused tag value; it starts at 0 and counts up. The setup thus looks like this:

```
1 late() {
  sim.addSubpop("p1", 500);

  sim.tag = 0; // the next unique tag value to use for TEs

  // create some transposons at random positions
  genomes = sim.subpopulations.genomes;
  positions = rdunif(teInitialCount, 0, L-1);

  for (teIndex in 0:(teInitialCount-1))
  {
    pos = positions[teIndex];
    mut = genomes.addNewDrawnMutation(m2, pos);
    mut.tag = sim.tag;
    sim.tag = sim.tag + 1;
  }
}
```

We make a new subpop, start sim.tag at 0, and then create new TEs that are fixed across the whole population. (As in the recipe of section 14.11, creating an initial state that involves heterogeneity in the TEs possessed by individuals would also be possible but is more complex; see

that recipe for discussion.) The positions for the TEs are drawn randomly across the chromosome, and each TE is tagged with a sequential value from `sim.tag`.

Before implementing any more of the TE dynamics, let's implement the final output event. We want this to generate information on both active and inactive TEs, so the code is somewhat involved:

```
5000 late() {
    // print information on each TE, including the fraction of it disabled
    all_tes = sortBy(sim.mutationsOfType(m2), "position");
    all_disabledTEs = sortBy(sim.mutationsOfType(m3), "position");
    genomeCount = size(sim.subpopulations.genomes);

    catn("Active TEs:");
    for (te in all_tes)
    {
        cat("    TE at " + te.position + ": ");

        active = sim.mutationCounts(NULL, te);
        disabledTE = all_disabledTEs[all_disabledTEs.tag == te.tag];

        if (size(disabledTE) == 0)
        {
            disabled = 0;
        }
        else
        {
            disabled = sim.mutationCounts(NULL, disabledTE);
            all_disabledTEs = all_disabledTEs[all_disabledTEs != disabledTE];
        }

        total = active + disabled;

        cat("frequency " + format("%0.3f", total / genomeCount) + ", ");
        catn(round(active / total * 100) + "% active");
    }

    catn("\nCompletely disabled TEs: ");
    for (te in all_disabledTEs)
    {
        freq = sim.mutationFrequencies(NULL, te);
        cat("    TE at " + te.position + ": ");
        catn("frequency " + format("%0.3f", freq));
    }
}
```

This prints all of the active TEs in the model (sorted by position), with information on their overall frequency in the population and on the fraction of their occurrences that have been disabled by mutations. After that, it prints a list of the completely disabled TEs (sorted as well); frequencies are also given for those, since TEs could conceivably be disabled before fixing. The logic of this code is quite straightforward, so there is no need to belabor it here.

We want our TEs to copy themselves; this occurs during the life of the organism, not during meiosis, so we model it with a `late()` event:



```

late() {
  // make active transposons copy themselves with rate teJumpP
  for (individual in sim.subpopulations.individuals)
  {
    for (genome in individual.genomes)
    {
      tes = genome.mutationsOfType(m2);
      teCount = tes.size();
      jumpCount = teCount ? rpois(1, teCount * teJumpP) else 0;

      if (jumpCount)
      {
        jumpTEs = sample(tes, jumpCount);

        for (te in jumpTEs)
        {
          // make a new TE mutation
          pos = rdunif(1, 0, L-1);
          jumpTE = genome.addNewDrawnMutation(m2, pos);
          jumpTE.tag = sim.tag;
          sim.tag = sim.tag + 1;
        }
      }
    }
  }
}

```

This loops through the genomes of all individuals in the simulation. For each genome, it gets all of the TEs present, and decides how many (if any) will “jump” according to the probability `teJumpP` for each TE, using a draw from a Poisson distribution. (This is faster than doing a separate random draw for each TE, but is otherwise essentially equivalent.) If any TEs did jump, the ones that jumped are selected at random using `sample()`. Each jump is simulated by creating a new TE at a new, randomly chosen location. The new TEs get new `tag` values assigned sequentially from `sim.tag`, so that their corresponding disabled versions can be looked up. And that is it for jumping; its logic is straightforward since disabled TEs are not involved in the jumping process.

Finally, we need to implement the disabling of TEs by random point mutations. Since TEs are simulated here as point mutations themselves, we need to simulate this disabling process ourselves; SLiM’s automatic mutation generation will never modify our TEs for us. We model disabling mutations in a `modifyChild()` callback. Its logic is similar to that of the jumping code above, except that when a TE is disabled, a mutation of type `m3` needs to be substituted in place of the existing `m2` mutation that represents the TE. The `m3` mutation corresponding to any given `m2` TE is created just once, and then the same `m3` mutation is looked up and reused every time that same `m2` TE is disabled in another genome. This “uniquing” of the `m3` mutations makes the model much more memory-efficient, and makes the output code much simpler. The main purpose of the `tag` values we have been managing is, in fact, to facilitate this uniquing process. The disabling callback looks like this:

```

modifyChild() {
  // disable transposons with rate teDisableP
  for (genome in child.genomes)
  {
    tes = genome.mutationsOfType(m2);
    teCount = tes.size();
    mutatedCount = teCount ? rpois(1, teCount * teDisableP) else 0;

```

```

    if (mutatedCount)
    {
        mutatedTEs = sample(tes, mutatedCount);

        for (te in mutatedTEs)
        {
            all_disabledTEs = sim.mutationsOfType(m3);
            disabledTE = all_disabledTEs[all_disabledTEs.tag == te.tag];

            if (size(disabledTE))
            {
                // use the existing disabled TE mutation
                genome.removeMutations(te);
                genome.addMutations(disabledTE);
                next;
            }

            // make a new disabled TE mutation with the right tag
            genome.removeMutations(te);
            disabledTE = genome.addNewDrawnMutation(m3, te.position);
            disabledTE.tag = te.tag;
        }
    }

    return T;
}

```

For each genome in the proposed child, we get the TEs contained, and calculate the number that will be disabled using a Poisson draw as before. We select the TEs that actually mutated, and for each, attempt to look up a corresponding m3 mutation using the tag value of the m2 mutation. If we find one, we substitute that. If not, we create a new m3 mutation to represent the disabled state, marking it with the TE's tag value so we can look it up next time. That's it for the TE disabling code; a bit lengthy, but the logic is simple.

If this model is run, typical output might look like this:

```

Active TEs:
  TE at 1793: frequency 0.011, 100% active
  TE at 2435: frequency 0.208, 100% active
  TE at 3629: frequency 0.002, 100% active
  TE at 6339: frequency 0.208, 94% active
  TE at 7081: frequency 0.020, 100% active
  TE at 7728: frequency 1.000, 79% active
...
Completely disabled TEs:
  TE at 24821: frequency 1.000
  TE at 83627: frequency 1.000
  TE at 98286: frequency 1.000
...

```

That output shows a mix of TEs at high frequency (perhaps present already in the initial population setup, since not that many ticks have elapsed) and TEs at low frequency (which must be copies due to jumping). Some TEs are completely active, whereas others have been partially or fully disabled by mutations. (Once even a single copy of a TE is disabled by mutation, that copy might drift to fixation; the fact that TEs have been completely disabled does not mean that TE-disabling mutations are particularly common.)

Note that this model is never at equilibrium; the number of TEs is likely to grow without bound, since the probability of jumping is greater than the probability of TEs being disabled by mutations. That means that the model runs ever more slowly, since the TE mutations are not allowed to fix. If a model didn't care about disabled TEs at all, it would be much simpler and faster to simply delete TEs from a genome when they are disabled.

Note also that the probabilities of jumping and of being disabled in this recipe are arbitrary and have almost no empirical basis; they just worked well for testing the code. Please do not use these values in any production code of your own.

With that, we have wrapped up this recipe, but obviously there are a great many aspects of the biology of transposons that we haven't covered here:

- Effects of transposons on fitness could be modeled through `mutationEffect()` callbacks, either modifying the fitness effects of other mutations due to the proximity of transposons, or modeling a direct fitness effect for the transposons themselves in their own `mutationEffect()` callback. Alternatively, at the moment of a transposon's jump a genetic effect could be modeled by examining and altering other mutations in the vicinity of the jump destination, to simulate up/down-regulation of those mutations (taking care to make a private copy of any modified mutations first, so that other genomes sharing the same mutations do not receive the same modifications as an unintended side effect).
- Transposons that relocate themselves, rather than copying themselves, could be implemented by adding a call to `genome.removeMutations(te)` in the jump code.
- Attempts by the organism to suppress TEs *en masse*, for example through epigenetic controls such as methylation, could be simulated by converting TEs in a given individual into a suppressed version (perhaps using a new mutation type to represent suppression); such suppression could be temporary, since one could write code to convert suppressed TEs back into active TEs again, too.
- Non-autonomous TEs that can jump only in the presence of a separate enabling gene could easily be modeled simply by checking for the presence of the enabling gene in an individual prior to allowing any TEs in that individual to jump.
- It is not clear exactly what, if anything, restricts the proliferation of TEs in real organisms, but one could certainly model some sort of balancing factor that would limit the proliferation of TEs in this model. The probability of jumping could decrease as the number of active TEs increases (for some unclear reason), or the fitness of organisms could start to decrease sharply as their TE load increases above some bound (with perhaps a bit more biological justification), or whatever other mechanism one wished.
- One of the most interesting aspects of transposons is that they sometimes jump more frequently when an individual is subject to environmental stress; that could be modeled by calculating a jump probability for the TEs in an individual that depends upon that individual's fitness, or upon the mismatch between its phenotype and its environment in some specific trait, if desired.

The point is that although this recipe is fairly rudimentary, many aspects of the evolutionary dynamics of transposons could be easily modeled in SLiM by implementing whatever extensions to this recipe are desired, as outlined above.

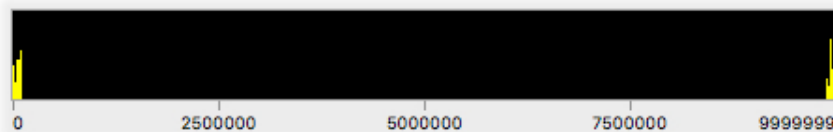
### 14.13 Modeling opposite ends of a chromosome

This recipe is not about how to model something complicated; instead, it is intended to illuminate something conceptual about using SLiM in certain situations.

Consider the following recipe:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeGenomicElement(g1, 9900000, 9999999);
  initializeRecombinationRate(1.5e-7);
}
1 early() { sim.addSubpop("p1", 500); }
200000 late() { sim.outputFixedMutations(); }
```

This is a simple neutral model of a chromosome of length  $1e7$  bases. The only unusual thing about it is that we are actually only modeling the two ends of the chromosome; we have defined a genomic element spanning the first  $1e5$  bases, and another spanning the last  $1e5$  bases, and the intervening region, which is 98% of the length of the chromosome, is not modeled. In SLiMgui, the model looks like this after a little while:



This is perfectly fine; all that it means is that SLiM will not automatically generate mutations within the region that is not covered by any genomic element, and so that central region remains empty. The region does not *have* to remain empty; it would be legal to call `addNewMutation()` to create a new mutation within that region, and SLiM would then track that added mutation normally. The only effect that genomic elements have, in SLiM, is of causing the automatic generation of new mutations by the SLiM core. But let's keep the model as it is, and ask: what is the behavior of the two end regions, with respect to recombination between them? Do they assort independently, as if they were separate chromosomes, since they are separated by a rather long stretch of chromosome? Or if not, then what is the effective recombination rate between them?

First of all, let's make sure we understand exactly what "recombination rate" really means in SLiM. As section 24.1 states, the recombination rate is the probability that a crossover will occur between two adjacent bases. This is binomial; conceptually, a coin is flipped, and the coin lands "heads" (crossover) with probability  $p$ , and "tails" (no crossover) with probability  $1-p$ , and that is done at every position between adjacent bases along the whole chromosome. There are  $L-1$  positions between bases in a chromosome of length  $L$ . In this model,  $2e5-2$  of them occur between the bases spanned by the two genomic elements, and the remainder,  $9800002$ , occur between those two regions. Those are the positions we're interested in, and at each position a crossover occurs with probability  $r=1.5e-7$ .

To get the probability that the two genomic elements will assort apart, rather than together, we cannot simply multiply the number of positions ( $9800002$ ) by the probability per position ( $1.5e-7$ ),

of course; that would give us 1.47, a nonsensical result that isn't even a valid probability. Instead, we need to observe that the key question is actually whether the number of crossovers that occur between the two regions is *even* or *odd*. An even number of crossovers will cancel out; we will cross over and then cross back, perhaps repeatedly, with no net effect. The two genomic elements will then assort together. An odd number of crossovers, on the other hand, will result in all but one crossover canceling out, and one crossover will remain; the two genomic elements will then assort apart.

So the question then becomes: for a binomial draw with 9800002 trials and a probability per trial of  $1.5 \times 10^{-7}$ , what is the probability that the result of the draw will be odd? There is, of course, established mathematical theory on this point, but let's answer the question through simulation. In an Eidos console, such as the one we can open inside SLiMgui, we can do a large number of draws from that binomial distribution:

```
> draws = rbinom(1e8, 9800002, 1.5e-7)
```

That will take a moment, and then we have 1e8 draws from the requisite binomial distribution. Next let's ask what fraction of them is odd:

```
> sum(draws % 2 == 1) / 1e8
0.473642
```

So, the two genomic elements will assort apart about 47.4% of the time, and the remaining time will assort together.

So now suppose we want to construct a SLiM model that just elides that central region (since we weren't using it anyway) and places the two chromosome ends immediately next to each other. There is no particular obstacle to doing this, except that we need to know what recombination rate to use for the join point between the two halves – which we have just figured out. So we could now rewrite our model as:

```
initialize() {
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeGenomicElement(g1, 100000, 199999);

  rates = c(1.5e-7, 0.473642, 1.5e-7);
  ends = c(99999, 100000, 199999);
  initializeRecombinationRate(rates, ends);
}
1 early() { sim.addSubpop("p1", 500); }
200000 late() { sim.outputFixedMutations(); }
```

This model ought to behave identically to the previous model, and ought to be somewhat more efficient, too (although the difference may not be large enough to be noticeable). This is because when running the first model SLiM would potentially be generating two or more breakpoints between the two ends, and then resolving whether there were an even or odd number as it ran, whereas when running the second model SLiM either generates zero breakpoints or one breakpoint, with the correct calculated probability.

As turns out, we could get the same probability by using the rate-rescaling formula of section 5.5, since what we are effectively doing is rescaling a region of length 9800002 down to a length of 1 (length in terms of potential recombination positions, that is). That formula, executed in the Eidos console, gives us:

```
> 0.5*(1-(1-2*1.5e-7)^9800002)
0.473567
```

Which is very close to the number we got before (which, remember, was inexact since it came from simulation). So that's pleasing.

The point of this recipe, then, is twofold. One point is to provide another angle on the rate-rescaling formula of section 5.5, to illustrate that it applies in situations other than the rescaling of an entire model. The second, more important, point is to illustrate once again that the recombination rate as specified by SLiM is the probability of a crossover occurring between two adjacent bases, and the consequences that that has for reasoning about how recombination works on larger scales.

This is as good a place as any for a digression that I think this manual ought to contain somewhere: how SLiM actually generates recombination breakpoint positions under the hood. This is worth demystifying because for models that use very high recombination rates the subtleties of this process can be important. Mostly, though, this digression is just for those who are curious, and beginning users of SLiM should feel free to skip it.

Overall, SLiM does this in several steps: (1) it decides upon the number of breakpoints it will generate, (2) it chooses the location for each breakpoint based upon a weighted uniform draw along the chromosome (where the weights are equal to the recombination rates for each individual base, as defined by the recombination map), and then (3) it sorts and uniques the list of breakpoints to provide a final list. The uniquing of the list means that there is, at most, one crossover (i.e., breakpoint) between any two base positions; this is biologically realistic, and also computationally simpler.

So the big question is: how to decide upon the number of breakpoints to generate? One way would be to start from the fact that between each pair of bases the question is one of a binomial draw, with a single trial of probability equal to the recombination rate (by SLiM's definition). One could do one such draw per pair of bases, add up the results, and that's the number of crossovers that occurred. The immediate problem with that approach is simply that it is immensely computationally expensive, and if the user has supplied a complex recombination map with different rates at every position it can't be reduced to a single binomial draw with a larger number of trials.

So perhaps we might wish to model recombination as a Poisson process instead, leveraging the fact that Poisson draws can be added together:  $\text{Poisson}(\lambda_1) + \text{Poisson}(\lambda_2) = \text{Poisson}(\lambda_1 + \lambda_2)$ . If two adjacent base positions have a probability of a crossover between them of  $\lambda$ , then the mean of the binomial draw can be used as the mean of a Poisson draw instead (the Poisson distribution being well-known as a viable approximation for the binomial distribution with small  $\lambda$  and largish  $n$ ). Then we can add up the  $\lambda$  values across the chromosome and get the number of recombination events with a single Poisson draw that uses that total  $\lambda$  value. And indeed, this is precisely what SLiM 2.x and earlier did.

Unfortunately, as SLiM users started moving towards new areas of modeling in SLiM such as multiple chromosomes, it became clear that there was a problem with this approach. When modeling multiple chromosomes, you want, intuitively, to connect them together in SLiM with a recombination rate of 0.5 to represent perfect independence of assortment (see the recipe in section 6.1.4, for example). Unfortunately, this value is large enough that the Poisson approximation of the binomial distribution breaks down. The Poisson draw can not only be 0 or 1 (as with the binomial), but also larger values, and with a rate of 0.5 this happens often enough to matter. In SLiM's case, if the Poisson draw indicated that (for example) two crossovers occurred,



their positions would be sorted and uniqued (to prevent multiple crossovers at the same exact location, as described above), resulting in just one crossover where the Poisson draw had specified two. The net effect of this would be that the probability of a crossover at the given location would be somewhat lower than desired – about 0.39, in fact, rather than 0.5. (It is lower than 0.5 because more than half of the Poisson draws for  $\lambda=0.5$  will be zero, compensating for the fact that some of the draws are greater than one and making the mean of the Poisson distribution come out to 0.5).

So at first blush it looks like we can't use a binomial draw (too complicated in the case of complex recombination maps) and we can't use a Poisson draw (inaccurate with large recombination rates such as 0.5). What to do?

The solution we arrived at, thanks to Peter Ralph, is to reparameterize the Poisson draws that we're doing. If the user has requested a recombination rate of  $p$  (probability of crossover between two adjacent bases), but we want to use a Poisson draw and have it generate a crossover, after sorting and uniquing, with probability  $p$ , then we can transform the requested  $p$  into a reparameterized value  $\lambda$  such that  $\text{Prob}[(\text{Poisson}(\lambda) > 0) = p]$ . The formula for this reparameterization is:

$$\lambda = -\log(1 - p)$$

With this reparameterization, we end up with the correct probability of crossover after using a  $\text{Poisson}(\lambda)$  draw, accounting for SLiM's sorting and uniquing of the breakpoint vector. This means that we can add up the  $\lambda$  values for each region along a whole chromosome, even with a complex recombination map, and draw a preliminary number of crossovers from a Poisson distribution with the total  $\lambda$ , and then after we select locations with the usual weighted uniform draws, and sort and unique them, the probability of crossover at every specific site will be as the user requested. It will work even when some positions have a rate of 0.5; and even though the Poisson distribution is only an approximate estimation for the binomial distribution, this solution is in fact exact, since it is based upon the probability that the Poisson draw for a specific position is greater than zero, not upon the mean of the Poisson distribution.

If that was all gibberish, never mind. The point is that, as of SLiM 3, recombination rates should be accurate (within numerical precision limits) even for large recombination rates like 0.5, with no sacrifice in speed.

## 14.14 Visualizing ancestry and admixture with mutation() callbacks

Beginning with version 3.3, SLiM provides the ability to examine, modify, or even veto every new mutation that it auto-generates, using a `mutation()` callback (see section 25.9). We previously saw the use of this callback in section 10.6 (where we used a `mutation()` callback to give an independent dominance coefficient to each mutation of a particular mutation type), and in section 13.5 (where we used a `mutation()` callback to tag new QTL mutations with effect sizes influencing two phenotypic traits). Both of those recipes, then, used the `mutation()` callback to associate some additional state information with every new auto-generated mutation of a given mutation type.

There are many other uses for `mutation()` callbacks, however; notably, they can be used to modify the mutation type used by new mutations. This can be useful for keeping track of particular mutations in a model, by giving them a different mutation type from the rest. Using otherwise identical mutation types as a way of categorizing mutations can be a powerful tool; one could use a distinct mutation type to separately track all mutations originating in a particular subpopulation, or in a particular tick, or within the gametes of a particular individual, or just all of the mutations in a randomly chosen sample.

In this recipe we will look at a two-subpopulation model that evolves separately for 10,000 generations, after which the subpopulations come into secondary contact and admix for another 5,000 generations. We're interested in the pattern of ancestry along the chromosome. Each subpopulation undergoes a mixture of common neutral mutations and rare beneficial mutations, and so when admixture occurs recombination leads to some kind of alternation of ancestry along the chromosome, as selection favors individuals that combine as many beneficial mutations as possible into their genome. The neutral background mutations are swept along by this process, and their pattern of ancestry reveals the pattern of admixture that occurred. Sections 14.7 and 17.5 present similar recipes along this theme (one using marker mutations, the other using tree-sequence recording); this section provides a different angle, and a different type of solution, for this problem. The focus here is not upon determining the true local ancestry at every position along the chromosome, but upon tracking and visualizing the fates of individual mutations, using different mutation types depending upon their subpopulation of origin.

Let's start with the `initialize()` callback and subpopulation setup:

```
initialize() {
    initializeMutationRate(1e-8);

    // neutral and beneficial for p1
    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.color = "yellow";
    m1.colorSubstitution = "yellow";

    initializeMutationType("m2", 0.5, "f", 0.1);
    m2.color = "red";
    m2.colorSubstitution = "red";

    // neutral and beneficial for p2
    initializeMutationType("m3", 0.5, "f", 0.0);
    m3.color = "blue";
    m3.colorSubstitution = "blue";

    initializeMutationType("m4", 0.5, "f", 0.1);
    m4.color = "green";
    m4.colorSubstitution = "green";

    initializeGenomicElementType("g1", c(m1, m2), c(1, 0.0001));
    initializeGenomicElement(g1, 0, 9999999);
    initializeRecombinationRate(1e-7);
}
1 early() {
    sim.addSubpop("p1", 1000);
    sim.addSubpop("p2", 1000);
}
```

This is straightforward enough; we make four mutation types (m1 and m3 being neutral, m2 and m4 being beneficial), and we set up custom colors for all of them in SLiMgui (see section 7.4). We then create two subpopulations, p1 and p2, with no migration initially.

As the comments indicate, we want new mutations in p1 to use m1 and m2, whereas new mutations in p2 should use m3 and m4. There is no way to tell SLiM to do that intrinsically, however; the genetic structure of genomic elements and genomic element types in SLiM is fixed across the whole model (see section 1.5.4). The genetic structure we set up here therefore just uses m1 and m2, which is fine for p1 but wrong for p2.

We will fix that problem using `mutation()` callbacks:

```

mutation(m1, p2)
{
    // use m3 instead of m1, in p2
    mut.setMutationType(m3);
    return T;
}
mutation(m2, p2)
{
    // use m4 instead of m2, in p2
    mut.setMutationType(m4);
    return T;
}

```

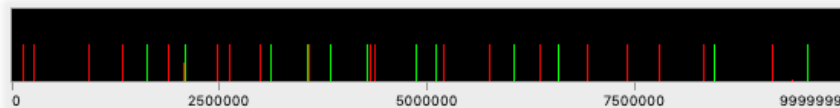
The first callback is declared as applying to m1 mutations generated in p2, and it changes the mutation type of all such mutations to m3, and then returns T to tell SLiM to proceed with generating the proposed mutation. The second callback similarly applies to m2 mutations in p2, and changes their type to m4. That is the crux of the recipe; all that is left is to begin admixture at tick 10000 (turning off mutation at the same time, for simplicity, so that we can see the effect on the standing genetic variation more clearly), and to end at tick 15000:

```

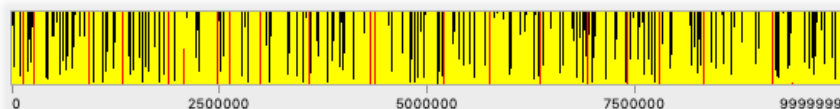
10000 early() {
    sim.chromosome.setMutationRate(0.0);
    p1.setMigrationRates(p2, 0.01);
    p2.setMigrationRates(p1, 0.01);
}
15000 late() { sim.outputFixedMutations(); }

```

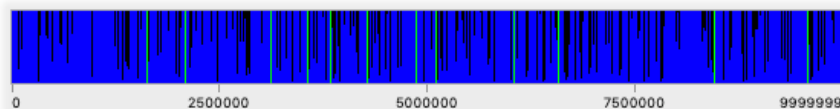
Running this model to tick 10000, we see this in SLiMgui if we focus in on just the beneficial mutations (by control-clicking or right-clicking on the chromosome view):



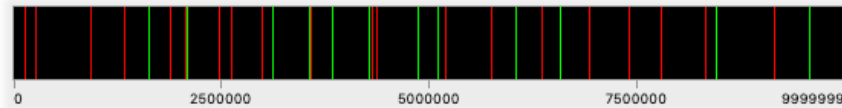
The beneficial mutations have generally reached a frequency of 0.5, but cannot spread further because of the lack of migration between the subpopulations. We can look at all mutation types, but just in p1 (select p1 from the subpopulation table at the upper left of the window):



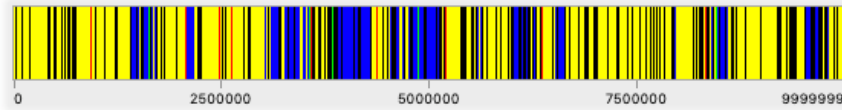
Here we see that within p1 the beneficial m2 mutations (red) have generally fixed, and there are many neutral m1 mutations (yellow) as well, at various frequencies. The pattern in p2 is similar, but the colors are different because the mutation types used are different, thanks to the `mutation()` callbacks; in p2 the beneficial mutations are green (m4) and the neutral mutations are blue (m3):



At this point, the model turns on admixture, and after 272 more ticks all of the beneficial mutations have either fixed or been lost, population-wide:



A lot of neutral diversity was carried along, but it has not necessarily fixed yet. Once we run out to 15000 ticks, however, the pattern in the neutral mutations is quite visible:



Comparing the last two snapshots above, it is apparent that regions where the beneficial mutations from p1 won (red bars) mostly contain neutral mutations from p1 as well (yellow bars), whereas regions where the beneficial mutations from p2 won (green bars) mostly contain neutral mutations from p2 as well (blue bars).

The recipe of section 17.5 does this even better, in a sense; as we will see there, using tree-sequence recording allows the true local ancestry at every position in every genome to be known very efficiently. However, that information is only available after the fact, from analysis in Python. The recipe here provides similar information, based upon the mutations in the model, in a way that can be visualized interactively in SLiMgui. This information could also be used to actually influence the dynamics of the model as it runs – perhaps individuals with a higher degree of p1 ancestry turn out to be more susceptible to an introduced disease, for example – in a way that would be difficult to do with tree-sequence recording.

The point of this recipe is not really about tracking true local ancestry, however; for most purposes tree-sequence recording does that better and faster. The point here is that `mutation()` callbacks can be used to tweak the mutation type of new mutations in *any way desired*, to separately track particular sets of mutations based upon *any* metric of interest – their subpopulation of origin, their tick of origin, the state of the individual in which they originated, or anything else. Once separate mutation types are being used to track different sets of mutations, they can be visualized differently in SLiMgui, they can be located separately with methods like `mutationsOfType()`, they can influence model dynamics differently, and they can be distinguished in the output of methods like `outputFixedMutations()`. This is a very general technique with a wide variety of uses.

### 14.15 Modeling biallelic loci with a `mutation()` callback

A question that occasionally arises is: how do I model a biallelic locus in SLiM? A biallelic locus is a locus that has only two possible states; we will call them A and a. A new mutation at a site that has an A will produce an a; a mutation at a site that has an a will produce an A. This is a very common thing to model mathematically, because it is simple and tractable. However, it takes a bit of doing in SLiM, since SLiM is geared towards other types of models: either infinite-alleles models (the default), or nucleotide-based models. In this section we'll construct a recipe of 100 unlinked biallelic loci – *two* recipes, in fact, to look at two different approaches to the problem.

There are a couple of issues to be handled. One is that normally in SLiM each new mutation event creates a new mutation object, representing a new “mutational lineage”; SLiM is normally an “identity by descent” style of model wherein each mutational lineage is tracked separately (see section 1.5.2). Since we want our model to be strictly biallelic, however, we want “identity by state” instead; we want only two mutation objects in the model for a given position, A and a. We will achieve this by “uniquing down” new mutations to a canonical set of mutations.

Another issue is that normally in SLiM, mutations “stack”; a new mutation does not replace an existing mutation at a given position, but instead stacks on top of it (see section 1.5.3). We want A to replace a, and a to replace A, though; we don't want our biallelic mutations to stack. We will achieve this using SLiM's “stacking policy” facility in one recipe, and in the other recipe we will script our own solution.

Finally, there is the question of how exactly the A and a states are modeled. In SLiM, we typically use the *lack* of a mutation at a position to represent the “ancestral” or “wild-type” state at that position, and that empty state has no fitness effect; only when a mutation exists is there a fitness effect (see section 1.5.2). We will follow that convention in one recipe; allele a will be represented by the empty state, and allele A by the presence of a mutation. In the other recipe, we will instead represent both a and A with mutations, and will not allow the empty state to occur.

Let's start with the version that uses mutation objects for both states; it is a bit simpler in its implementation. Here's the initialization:

```
initialize() {
  // an m2 mutation is "A"
  initializeMutationType("m2", 0.5, "f", 0.0);
  m2.convertToSubstitution = F;
  m2.color = "red";

  // an m3 mutation is "a"
  initializeMutationType("m3", 0.5, "f", 0.0);
  m3.convertToSubstitution = F;
  m3.color = "cornflowerblue";

  // enforce mutually exclusive replacement
  c(m2,m3).mutationStackGroup = 1;
  c(m2,m3).mutationStackPolicy = 'l';

  initializeGenomicElementType("g1", c(m2,m3), c(1.0,1.0));
  initializeGenomicElement(g1, 0, 99);
  initializeMutationRate(1e-4);
  initializeRecombinationRate(0.5);
}
```

We define two mutation types, m2 and m3 (I usually follow the convention of reserving m1 for ordinary neutral mutations, but that is just a habit). Both types are set up to be neutral; these recipes will be neutral models for simplicity. We prevent both types from being converted to substitutions when they fix; for one thing, being “fixed” is just a temporary state in this model since back-mutations are constantly occurring, and for another, we want to prevent the empty state from ever occurring in this model. We also give the mutation types different colors for easy viewing in SLiMgui.

Next we set up the stacking policy for these mutation types. We want them both to be in the same “stacking group”; a new mutation of either type will interact with an existing mutation of either type, therefore they belong in the same stacking group. The “interaction” in question is replacement – we want a new mutation of either type to replace any existing mutation of either type, therefore both mutation types should use the type “l” stacking policy, where “l” is for “last” meaning that the last mutation wins.

Finally, we set up the chromosome to have 100 unlinked loci (using a recombination rate 0.5), with a mutation rate of 1e-4 (quite high, for testing purposes) and equal probabilities of m2 and m3 mutations.

With this genetic configuration, SLiM will do much of the work for us automatically: m2 and m3 mutations will replace each other. Next we should set up the initial state of the model:

```
1 early() {
    sim.addSubpop("p1", 100);

    // create the canonical mutation objects
    target = p1.genomes[0];
    target.addNewDrawnMutation(m2, 0:99);
    defineConstant("MUT2", target.mutations);
    target.removeMutations();
    target.addNewDrawnMutation(m3, 0:99);
    defineConstant("MUT3", target.mutations);
    target.removeMutations();

    // start homozygous "aa" at every position
    p1.genomes.addMutations(MUT3);

    // log results
    log = community.createLogFile("freq.csv", logInterval=10);
    log.addTick();
    log.addMeanSDColumns("freq", "sim.mutationFrequencies(NULL, MUT2);");
}
```

After creating a new subpopulation, this prepares the model for action. As mentioned above, we want to “unique down” new mutations to a set of canonical mutation objects so that we represent identity by state rather than identity by descent; we don’t want multiple mutational lineages for one mutational state, as SLiM would normally provide. In this `early()` event we actually create the canonical mutation objects that we will use throughout the simulation; this is a technique I don’t think we have seen before. We use one genome (it doesn’t matter which) as a temporary “target”, and add new m2 mutations at every position in it. Then we fetch those new mutations out of the target, and remember them in a defined constant for the remainder of the run: these are our canonical mutations for the m2 mutation type. We finish by removing them all from the temporary target genome. We do the same with m3, obtaining our canonical m3 mutations.

Recall that we never want to allow the empty state to occur at any position; every position in every genome must have a mutation, either m2 or m3. Arbitrarily, we therefore initialize the model to be m3 at every position by adding the canonical m3 mutations to every genome, providing a legal initial state according to our rules.

Finally, we set up a `LogFile` object to log out the mean (and standard deviation, which we will not use here) of the frequencies of m2 mutations, every tenth tick. We will use this later to generate a plot.

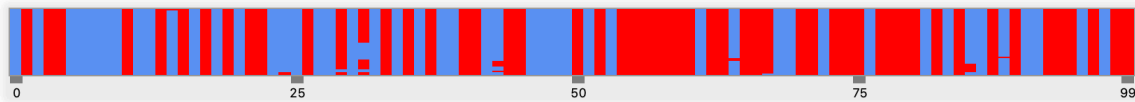
We are almost done; we just need to “unique down” to the canonical mutation whenever a new mutation occurs. We do that with two trivial `mutation()` callbacks (and here’s the termination event, too):

```
mutation(m2) {
    // unique down to the canonical m2 mutation
    return MUT2[mut.position];
}
mutation(m3) {
    // unique down to the canonical m3 mutation
    return MUT3[mut.position];
}
50000 late() { }
```

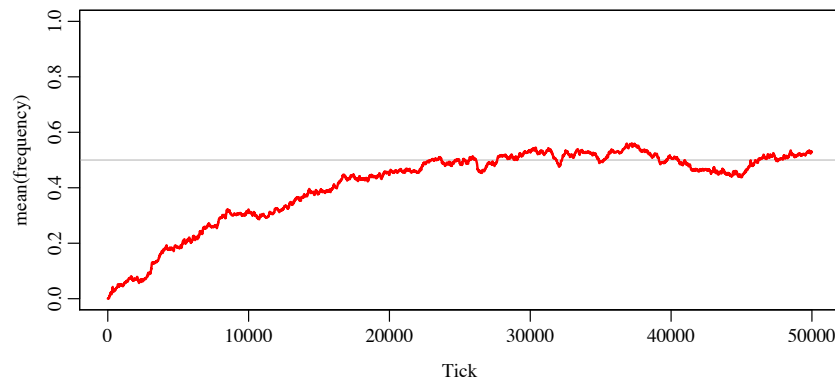


These simply look up the corresponding canonical mutation, by position, from the appropriate global constant and return it. SLiM will add that mutation rather than the mutation it originally proposed.

If we run this in SLiMgui, the “Display Haplotypes” mode of the chromosome view proves very useful since every position has either an m2 or m3 mutation. At the end of a run, the chromosome view looks like this:



Some positions are (temporarily) fixed for allele A (red), some positions for allele a (blue), and a few positions are at an intermediate frequency with both alleles segregating. If we plot the logged output in R, we can see that the mean frequency of A alleles in the model evolves towards 50%, with some stochasticity:



The model appears to be working correctly, although one could perform additional statistical tests to confirm that it matches theoretical expectations from analytical models.

Let's move on to the second recipe for this section; recall that for this one, we will use the empty state to represent a, and a mutation at a given position will represent A. We will again want to unique those mutations down to provide identity by state, and we will need to ensure that mutations do not stack, and that allele a replaces A and vice versa. Here's the initialization:

```
initialize() {
  // m2 models a biallelic locus; an m2 mutation is "A",
  // absence of an m2 mutation is "a"; "aa" is neutral
  initializeMutationType("m2", 0.5, "f", 0.0);
  m2.convertToSubstitution = F;
  m2.color = "red";

  // m3 is used for new mutations; new m3 mutations get
  // unique down to the correct biallelic m2 state
  initializeMutationType("m3", 0.5, "f", 0.0);
  m3.convertToSubstitution = F;
  m3.color = "cornflowerblue";

  initializeGenomicElementType("g1", m3, 1.0);
  initializeGenomicElement(g1, 0, 99);
  initializeMutationRate(1e-4);
  initializeRecombinationRate(0.5);
}
```

We create mutation type m2 to represent A, and tell it not to fix. We also create a mutation type m3 which is used only for the internal implementation; we will never have m3 mutations segregating in the population (so if we ever see a mutation in SLiMgui that is colored blue, we know we have a bug). New mutations are always of type m3, and our script will sort out what to do with them. We do not alter SLiM's default stacking policy; this recipe will actually leverage stacking to do its work, as shown below. In other respects this mirrors the previous recipe.

Next let's look at the initial population setup:

```
1 early() {
    sim.addSubpop("p1", 100);

    // create the permanent m2 mutation objects we will use
    target = p1.genomes[0];
    target.addNewDrawnMutation(m2, 0:99);
    defineConstant("MUT", target.mutations);

    // then remove them; start with "aa" for all individuals
    target.removeMutations();

    // log results
    log = community.createLogFile("freq.csv", logInterval=10);
    log.addTick();
    log.addMeanSDColumns("freq", "sim.mutationFrequencies(NULL, MUT);");
}
```

We create a defined constant containing the canonical m2 mutations for the model, as we did before. After removing them from the temporary target genome, every genome is empty at every position; this represents the homozygous aa state in this model. We end by creating a LogFile instance, as before.

Now we define a mutation() callback that influences the way in which new m3 mutations are generated:

```
mutation(m3) {
    // if we already have an m2 mutation at the site, allow
    // the new m3 mutation; we will remove the stack below
    if (genome.containsMarkerMutation(m2, mut.position))
        return T;

    // no m2 mutation is present, so unique down
    return MUT[mut.position];
}
```

This first checks whether an m2 mutation already exists in the target genome at the position that is mutating. If one does, then what we need is a back-mutation to the empty state – we want to remove the existing m2 mutation and replace it with nothing. At present, mutation() callbacks don't support doing that, however (it would be a nice feature to add some day). So instead, we allow the proposed m3 mutation to be added, by returning T. This creates a stacked setup, with the new m3 mutation on top of the existing m2 mutation. We will clean up that mess in the late() event shown below.

Otherwise, there is no previously existing m2 mutation, so this is a new mutation at an empty position. We return the canonical m2 mutation for this position, to “unique down” to it, as in the previous recipe.

OK, so far so good – but we’ve got that mess to clean up. Wherever a stacked m2/m3 pair exists, we want to remove both of them, thereby achieving the empty a allelic state that we want. The code for this is a bit complicated:

```
late() {
  // implement back-mutations from A to a
  m3mut = sim.mutationsOfTypes(m3);

  // do we have any m3 mutations segregating?
  // if so, we have m2/m3 stacked mutations to remove
  if (m3mut.length() > 0)
  {
    genomes = sim.subpopulations.genomes;
    counts = genomes.countOfMutationsOfTypes(m3);
    hasStacked = genomes[counts > 0];

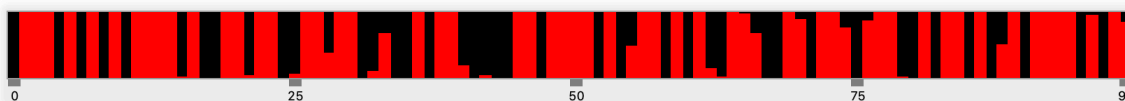
    for (genome in hasStacked)
    {
      stacked_m3 = genome.mutationsOfTypes(m3);
      stackPositions = stacked_m3.position;
      all_m2 = genome.mutationsOfTypes(m2);
      s = (match(all_m2.position, stackPositions) >= 0);
      stacked_m2 = all_m2[s];
      genome.removeMutations(c(stacked_m3, stacked_m2));
    }
  }
}
```

First we ask `sim` for all m3 mutations. If any exist, we have stacks to clean up; if none exist, then we can skip the rest of the work. Assuming some exist, we fetch all the genomes in the model and narrow down to those that contain at least one m3 mutation; these are the genomes that we need to clean up. We loop through them with a `for` loop (this code cannot be vectorized, since each genome needs to be processed differently). For each focal genome, we get the m3 mutations it contains – there could be more than one – and the positions at which they occur. Then we use `match()` to find the m2 mutations in the genome that are at the same positions; these are the ones that are stacked with the new m3 mutations. Finally, we remove them all from the focal genome. When the `for` loop is done, all of the stacks have been cleaned up, all back-mutations to allele a have been handled, and no m3 mutations are left in the model – until next tick.

And then we have a termination event as before, and we’re done:

```
50000 late() { }
```

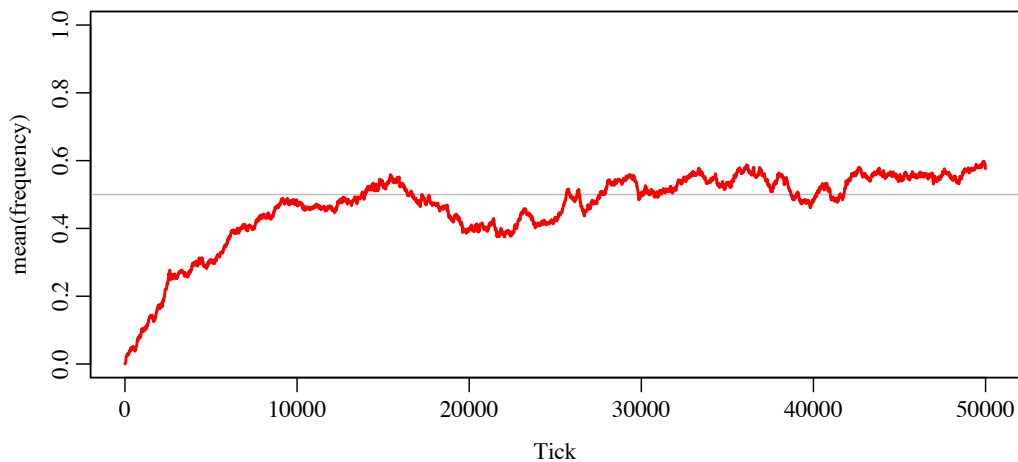
The default chromosome view display mode works well for this model, since it uses the empty state for a; we can see the frequency of A at each position:



The complementary black bars coming down from the top (if you shift your perspective or stand on your head) are the frequencies of a at each position. As before, most sites are fixed for a or A, and a few are at intermediate frequencies. (It might look like there are more at intermediate frequencies here, compared to the previous recipe; but actually, the Display Haplotypes mode that we used to view the previous recipe will tend to obscure low-frequency heterozygosity because it is based upon a subsample of the population’s genomes, and alleles at low frequency will often

not be visible due to sampling error; generate a full-size haplotype plot based on all genomes to see the full reality of the haplotype structure in the population.)

The R plot from the logged data also looks fine:



It looks similar to the results from the other recipe, and indeed, the two models ought to behave identically apart from the effects of stochasticity.

Which recipe is “better”? It’s hard to say; that depends upon your larger goals. The first recipe is conceptually simpler and is also shorter. They run at about the same speed; the first recipe has twice as many segregating mutations, which makes for some extra work for SLiM, but the second recipe has to deal with removing the m2/m3 stacks. The first recipe may produce cleaner tree-sequence recording output (see section 1.7), if you are interested in that, because it won’t have transient m3 mutations flickering in and out each tick; but neither recipe will really produce a very sensible tree sequence, since the tree sequence is inherently designed to represent identity by descent, not identity by state.

Perhaps the biggest difference is how fitness is handled. Both of these recipes are neutral models, for simplicity, which has kind of hidden that aspect of their design. In the first recipe, with both m2 and m3 mutations segregating to represent A and a, heterozygous individuals contain an m2 mutation in one genome and an m3 mutation in the other. SLiM does not know that these are, in a sense, two sides of the same coin; instead, it considers these individuals to be heterozygous for the m2 mutation and heterozygous for the m3 mutation, and it will calculate the fitness effect of each and multiply them together. Probably, to avoid confusion, you ought to keep the fitness effect of a (m3) neutral, and allow only A (m2) to be non-neutral. In the second recipe, this is true automatically and there is no potential for confusion; the empty state is a, and only A has an associated mutation object that has a fitness effect.

Both recipes are really pretty reasonable and usable. Probably the factor that makes one strongly preferable to the other will arise when you try to extend them to make a more complex model, of which these biallelic dynamics are only a component. At that point you might find that one or the other recipe proves to be more easily adaptable to your final design. Experiment with both! Also note that modeling identity by state is not essential here; these recipes do it to make the visualization and analysis easier, but if you want to have multiple mutation lineages in a biallelic model, you certainly can (and then tree-sequence recording will produce more coherent results). The first recipe probably lends itself better to that approach.

In closing, it’s worth noting that we could have also constructed a biallelic model using a nucleotide-based SLiM model. To do this, just restrict the initial state of the model to have only

(for example) C and G in the ancestral sequence, and then use a custom mutation matrix to force C to mutate only to G, and G to mutate only to C. That will give you biallelic behavior, although if you want to actually have only one mutation of each type at each position you will still have to “unique down” as well. But we haven’t seen nucleotide-based models yet (that’s chapter 18), and in any case it is a bit odd to use nucleotides to represent states that are not, conceptually, actually nucleotides at all. The recipes shown here are therefore probably preferable.

Overall, these two recipes are not intended to represent the one right way (or two right ways!) to write a biallelic model in SLiM. Instead, the goal here has been to illustrate different techniques for influencing things like identity by state versus identity by descent, mutation stacking and stacking policy, and “uniquing down” of new mutations to alternative mutational states. Once you are comfortable with these techniques you can use them to achieve a wide variety of goals, and you will have a better understanding of how SLiM really works under the hood.

## 15. Continuous-space models and interactions

This chapter introduces two new features added in SLiM 2.3: continuous space, and the `InteractionType` class. These two features will be treated together in this chapter, since many of the uses of `InteractionType` involve modeling spatial interactions, but in fact each feature may be used independently of the other. Note that these are advanced, optional features.

Continuous-space support in SLiM is enabled by supplying a `dimensionality` parameter to the `initializeSLiMOptions()` call during model initialization. This parameter may be "x", "xy", or "xyz"; these set up SLiM to support 1D, 2D, or 3D continuous space, respectively, within each subpopulation using the coordinate axes specified. (Setting up spaces using different coordinate axes than these, such as 2D space using *y* and *z*, is not presently supported.) The corresponding *x*, *y*, and *z* properties of `Individual` will then be interpreted by SLiM as spatial coordinates, rather than simply as type `float` tag values. Model code can then set the positions of individuals as desired; this is typically done in a `modifyChild()` callback, so that the positions of new offspring are immediately set, but it can also be done at any other point in the tick cycle. Individual positions can also be used by model code in any way desired; for example, spatially continuous variation in selection could be implemented in a `mutationEffect()` callback by computing an environmental value at the spatial location of the focal individual, and then comparing the individual's phenotype to that environmental value (see section 15.9). When continuous space is enabled, SLiMgui will display subpopulations graphically using the spatial positions of individuals.

Interactions between individuals can be implemented in pure Eidos, as seen previously in the frequency-dependent model of section 10.4.1 and the green-beard model of section 10.4.3. This can be cumbersome, however, since all of the interaction mechanics must be written by the user in Eidos, which – as an interpreted language – is not nearly as fast as the C++ in which the SLiM engine is written. The `InteractionType` class provides a solution to this problem, by providing built-in support for fast interaction evaluation while still allowing scripted customization of the interactions through a new type of callback, the `interaction()` callback. `InteractionType` can be used to model non-spatial interactions, as we will see in section 15.6; but it is particularly well-suited to modeling interactions in continuous space, since it is able to translate distances into interaction strengths automatically using any of several different spatial kernels (Gaussian, negative exponential, etc.) called “interaction functions”. In addition, `InteractionType` is optimized for spatial searches (such as nearest-neighbor searches) through the use of two specialized data structures, a “*k*-d tree” and a “sparse array” (really, in SLiM 4, a collection of “sparse vectors”). Together, these features allow complex spatial interactions to be implemented in just a few lines of Eidos code, with performance that can be orders of magnitude better than would otherwise be possible.

When continuous space is enabled in a model, the `Subpopulation` class then contains several features to support that functionality. First, `Subpopulation` is then aware of its spatial boundaries (the spatial coordinates of the edges of the subpopulation). Second, it can help to enforce those boundaries through various boundary conditions. Third, it then supports “landscape maps”, grid-based maps that define the values of particular environmental variables across the spatial extent of the subpopulation. Landscape maps allow simulations to incorporate spatial variation in properties such as elevation or temperature – any variable relevant to the model's dynamics.

With that introduction, let's delve into some recipes that exemplify these new features.

### 15.1 A simple 2D continuous-space model

In this recipe we will explore a very simple model utilizing continuous space. In this model, individuals will live on a two-dimensional landscape. Individuals will not interact spatially in this model; that will be introduced in section 15.2. The model:



```

initialize() {
  initializeSLiMOptions(dimensionality="xy");
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 late() {
  sim.addSubpop("p1", 500);

  // initial positions are random in ([0,1], [0,1])
  p1.individuals.x = runif(p1.individualCount);
  p1.individuals.y = runif(p1.individualCount);
}
modifyChild() {
  // draw a child position near the first parent, within bounds
  do child.x = parent1.x + rnorm(1, 0, 0.02);
  while ((child.x < 0.0) | (child.x > 1.0));

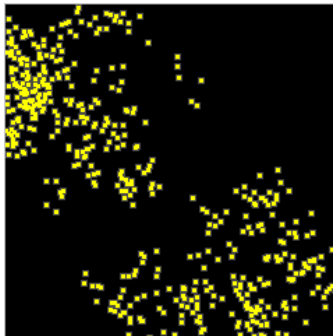
  do child.y = parent1.y + rnorm(1, 0, 0.02);
  while ((child.y < 0.0) | (child.y > 1.0));

  return T;
}
2000 late() { sim.outputFixedMutations(); }

```

There are only a few new things about this model. First of all, continuous 2-D space is enabled in the model with the call to `initializeSLiMOptions()`. Second, when the subpopulation is created in the 1 `late()` event initial positions for all individuals are generated using `runif()`. Note that by default the spatial extent of the subpopulation spans the interval [0,1] in *x* and *y*, so the default `min` and `max` values for `runif()` suffice. Third, a `modifyChild()` callback generates a spatial position for the offspring individual. In this recipe, the offspring position is based upon the position of the first (i.e. maternal) parent, as given by `parent1.x` and `parent1.y`, with random deviations drawn from a normal distribution using `rnorm()`. The new positions could fall outside of the subpopulation's boundaries, so they are redrawn until they fall inside using `while` loops.

When run in SLiMgui, a typical snapshot of this model looks like this:



Here, SLiMgui is displaying each individual at its corresponding spatial position, as a small square (colored according to fitness, as usual). Notably, spatial structure has emerged already in this simple model, because of the way that child positions are based upon maternal positions; this tends to encourage spatial clustering. Since spatial position is of no consequence in the model, however, this makes no difference to the evolutionary dynamics observed.

## 15.2 Spatial competition

The recipe in the previous section set up spatiality but did not use it. In this section we will extend that recipe to include spatial competition between individuals. The strength of competition between two individuals will depend upon the spatial distance between them, falling off with increasing distance according to a Gaussian kernel with a characteristic width. This could represent, for example, competitive interference due to overlap in foraging areas. The recipe:

```
initialize() {
  initializeSLiMOptions(dimensionality="xy");
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);

  // Set up an interaction for spatial competition
  initializeInteractionType(1, "xy", reciprocal=T, maxDistance=0.3);
  i1.setInteractionFunction("n", 3.0, 0.1);
}
1 late() {
  sim.addSubpop("p1", 500);

  // initial positions are random in ([0,1], [0,1])
  p1.individuals.x = runif(p1.individualCount);
  p1.individuals.y = runif(p1.individualCount);
}
1: late() {
  // evaluate interactions before fitness calculations
  i1.evaluate(p1);
}
fitnessEffect() {
  // spatial competition
  totalStrength = i1.totalOfNeighborStrengths(individual);
  return 1.1 - totalStrength / p1.individualCount;
}
modifyChild() {
  // draw a child position near the first parent, within bounds
  do child.x = parent1.x + rnorm(1, 0, 0.02);
  while ((child.x < 0.0) | (child.x > 1.0));

  do child.y = parent1.y + rnorm(1, 0, 0.02);
  while ((child.y < 0.0) | (child.y > 1.0));

  return T;
}
2000 late() { sim.outputFixedMutations(); }
```

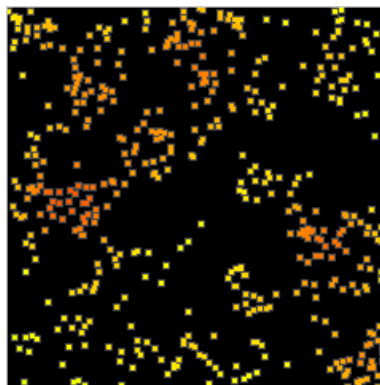
Here we have added a few new elements. First of all, a call to `initializeInteractionType()` creates a new interaction type with identifier 1, which is therefore referred to as `i1`, in much the same way that mutation types, genomic element types, and subpopulations are numbered and named in SLiM. This call also defines the interaction type as using both the x and y coordinates of the model, and sets the maximum range for the interaction as 0.3; beyond that limit interaction strengths are always assumed to be zero, allowing better performance. The interaction is also configured with `reciprocal=T`; this guarantees that the interaction strength exerted upon A by B is equal to the interaction strength exerted upon B by A, allowing computational optimizations. (Whether such optimizations are actually done depends on the version of SLiM.)

Second, a call to `setInteractionFunction()` tells `i1` that it should convert spatial distances into interaction strengths using a Gaussian function (represented by "n" for "normal", to avoid confusion with the "g" for "gamma" used elsewhere in SLiM). This Gaussian function is scaled to have a maximum value of 3.0 and a standard deviation of 0.1, representing fairly local interactions. It can now be seen how the maximum distance set for `i1` was chosen; it is 3× the standard deviation of the interaction kernel. Interaction strengths beyond that distance would be extremely small, and are neglected by this model for efficiency (see section 15.12 for discussion).

Next, interaction type `i1` needs to actually be used. A precondition for this is that `i1` be "evaluated" in each tick. This is done in the 1: `late()` event, with a call to its `evaluate()` method. This takes a snapshot of the model's spatial state at that moment in time, and `i1` will then calculate all interactions based upon that snapshot. This is necessary because under the hood, `InteractionType` performs a lot of time-consuming analysis to set up spatial data structures representing the state of the model, allowing it to respond quickly to spatial queries. It does that analysis just once, at the point of evaluation, and the results are cached and used for all queries until the interaction is evaluated again. (In fact, this is not quite accurate; `interaction()` callbacks may be called in a deferred fashion, as discussed when we introduce `interaction()` callbacks.) We pass `p1` to `evaluate()` to indicate that the interaction will be used within that subpopulation. It is required to `evaluate()` subpopulations for both receivers and exerts of the interaction type.

Having evaluated `i1` in the `late()` event, just before fitness calculations in the tick cycle, the model then uses `i1` in a `fitnessEffect()` callback to calculate fitness values that represent the effects of spatial competition. We have seen `fitnessEffect()` callbacks only a few times before (see sections 13.1 and 14.2). They are called once per individual per cycle, to evaluate a fitness effect for a given focal individual (see section 25.3). The returned fitness effect is combined, multiplicatively, with all other fitness effects that apply to that focal individual. In the `fitnessEffect()` callback here, the call to `totalOfNeighborStrengths()` totals up the interaction strengths between `individual` (the focal individual, called the *receiver*) and all individuals interacting with it, called *exerters*, in its subpopulation. These interaction strengths, as we saw above, were configured to be calculated using a particular Gaussian kernel, and a maximum distance of 0.3 was chosen. Those settings are now used to find the interaction strength between `individual` and every *exerter*, and the sum of those strengths is returned. The `fitnessEffect()` callback divides that total by the number of individuals in the subpopulation to get a mean interaction strength, and subtracts that mean value from 1.1 to get a fitness value. Those details are fairly arbitrary; the overall intent, however, is that the stronger the competition felt by a receiver, the lower the receiver's fitness. This `fitnessEffect()` callback will be called once for each individual in each cycle, resulting in a fitness effect due to spatial competition between individuals.

If we run this model in SLiMgui, it looks markedly different from the previous model:



Perhaps the most obvious difference is that individuals are no longer all yellow (which indicated neutrality). Now, individuals in relatively tight spatial clusters are orange or even red, indicating they have fitness values below 1.0 due to the effects of competition. Those individuals will be less likely to reproduce, whereas the individuals enjoying a lack of competition in isolated areas will be more likely to reproduce. That leads to the other obvious difference: the space is more completely and uniformly occupied. This model uses the same offspring positioning algorithm as the previous recipe, which promotes spatial clustering; however, excessive clustering is now opposed by the effects of competition, producing that more uniform distribution.

As discussed above, this model implements the fitness effect of spatial competition using a `fitnessEffect()` callback. This could easily be implemented using the `fitnessScaling` property of `Individual` instead, as we will see in section 15.4; the choice of which method to use is somewhat arbitrary, so we will use both in this chapter in order to illustrate both methods.

### 15.3 Boundaries and boundary conditions

Here we will pause to examine the question of boundaries and boundary conditions in spatial models. When a new offspring individual is given a position that is based upon the parental position(s) plus some random deviation, as is commonly done, the question arises of how to constrain those new positions; we have already confronted that question, in fact, in the `modifyChild()` callbacks we have used to set offspring positions.

One option is simply to impose no constraint; space is then considered to be infinite in extent in all directions. This is the default in SLiM, simply because SLiM imposes no default constraint; however, it is rarely desirable in individual-based models since a finite population on an infinite landscape has zero density – rarely an interesting or biologically relevant case. Instead, models generally use one of several standard boundary conditions. In this section we will break from our usual conventions and will simply assume all of the code from the previous recipe (section 15.2); here we will present only replacement `modifyChild()` callbacks to implement each of four standard boundary conditions. The literature contains some discussion of these different boundary conditions and the effects they may have on evolutionary dynamics (notably, Mazzucco, Doebeli & Dieckmann 2018). We will focus on implementation here.

First of all, with “stopping” boundaries, new positions are simply clamped to the spatial boundary; points outside the boundary are forced to the nearest point inside bounds. In SLiM, this can be implemented as:

```
modifyChild() {
  // Stopping boundary conditions
  pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
  child.setSpatialPosition(p1.pointStopped(pos));

  return T;
}
```

This introduces several new concepts. The `spatialPosition` property of `Individual` provides a float vector of the spatial coordinates of the individual; since this model has “xy” dimensionality, this vector has two elements, with the same values as the `x` and `y` properties that we have been using. The `setSpatialPosition()` is just the reciprocal of this, taking a float vector of coordinates and setting them into the `x` and `y` properties of the `Individual` in one operation, as a convenient shorthand. Finally, the `pointStopped()` method of `Subpopulation` implements the spatial boundary condition by clamping the coordinates in the float vector it is passed to the boundaries of the subpopulation and returning the clamped vector. So all in all, this callback gets the coordinates of the parent, adds a normal draw to each (thus deviating the offspring’s `x` and `y`

coordinates from those of the parent), asks the subpopulation to clamp the new position into bounds, and sets the final position into the child. All of this is just convenient shorthand; it could easily be implemented by using the `x` and `y` properties of the `Individual` directly, as well as the spatial boundaries of the `Subpopulation` (available through its `spatialBounds` property).

Second, with “reflecting” boundaries, new positions outside the spatial boundary are reflected to fall inside it; the extent to which a point lies *outside* an edge is translated into the extent to which the point lies *inside* that edge instead. This requires just a trivial modification of the previous recipe, substituting the `pointReflected()` method in place of `pointStopped()`:

```
modifyChild() {
  // Reflecting boundary conditions
  pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
  child.setSpatialPosition(p1.pointReflected(pos));

  return T;
}
```

Third, with “absorbing” boundaries, new offspring whose proposed positions lie outside bounds are absorbed – they are nipped in the bud, as it were, and are not generated. This is accomplished in SLiM by returning `F` from the `modifyChild()` callback, which tells SLiM to start over from scratch by choosing new parents and generating a completely new offspring:

```
modifyChild() {
  // Absorbing boundary conditions
  pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
  if (!p1.pointInBounds(pos))
    return F;

  child.setSpatialPosition(pos);
  return T;
}
```

This uses the `pointInBounds()` method of `Subpopulation` to test whether the new position lies inside the spatial boundaries. If not, `F` is returned, terminating generation of the proposed child.

Fourth, with “reprising” boundaries, if a proposed position falls outside bounds a new position is generated until a position within bounds is obtained. This is the boundary condition we implemented the hard way in sections 15.1 and 15.2, but it can be done a little more easily (and more generally, since the spatial boundaries are now not hard-coded):

```
modifyChild() {
  // Reprising boundary conditions
  do pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
  while (!p1.pointInBounds(pos));
  child.setSpatialPosition(pos);

  return T;
}
```

This again uses `pointInBounds()`, but this time if the point is not inside bounds the code loops back to generate a new point, until a point within bounds is obtained.

Finally, with “periodic” boundaries space is constructed in such a manner that boundaries do not exist but the spatial extent is nevertheless finite; the spatial topology is instead cylindrical or toroidal, wrapping around at some or all edges. This boundary condition is a more complex topic than the other boundary conditions, since it actually changes the way in which distances are

calculated; we will therefore defer a presentation of it until section 15.12. It's an important topic, though, since periodic boundaries can avoid problematic “edge effects” as discussed there.

We have been using Subpopulation methods to check/enforce the boundary conditions for us. Subpopulation is the class responsible for landscape-level properties such as the coordinates of the spatial boundaries, as well as other state we will see later. By default the spatial boundaries used by Subpopulation span the interval [0,1] in each dimension, and we have allowed that default to stand in the models shown here. This can be changed, using the `setSpatialBounds()` method of Subpopulation, but we will not pursue that here (see section 15.7 for an example). In any case, the Subpopulation methods we have seen will check and enforce whatever boundaries are set.

One final point worth mentioning is that Subpopulation provides a more general way to set up initial random positions. These recipes will use this code to do so:

```
1 late() {
    sim.addSubpop("p1", 500);

    // Initial positions are random within spatialBounds
    p1.individuals.setSpatialPosition(p1.pointUniform(500));
}
```

The `pointUniform()` method generates points drawn at random from within the spatial bounds of the subpopulation (drawing each coordinate from a uniform distribution spanning the range within bounds). Here it generates 500 such points – one for each individual – and returns them as one big vector. That vector is passed to `setSpatialPosition()`, which operates in a vectorized fashion, assigning one point to each corresponding individual (see section 24.7.2). This is equivalent to the code presented in earlier recipes, but it will also work properly if the spatial bounds of the subpopulation have been changed from the default, and it is simpler and faster.

## 15.4 Mate choice with a spatial kernel

Now we will work with the “reprising” boundary condition model of section 15.3, and we will add the element of spatial mate choice. The likelihood that one individual will choose another individual as a mate will depend upon the spatial distance between them, falling off with increasing distance according to a Gaussian kernel (but a different one from the competition kernel). This could represent, for example, mate-finding based upon auditory cues such as birdsong which become progressively less perceptible as distance increases.

Doing this is remarkably easy. We just need to add a second interaction type, evaluate it, and utilize it in a `mateChoice()` callback:

```
initialize() {
    initializeSLiMOptions(dimensionality="xy");
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);

    // spatial competition
    initializeInteractionType(1, "xy", reciprocal=T, maxDistance=0.3);
    i1.setInteractionFunction("n", 3.0, 0.1);

    // spatial mate choice
    initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.1);
    i2.setInteractionFunction("n", 1.0, 0.02);
}
```



```

1 late() {
    sim.addSubpop("p1", 500);
    p1.individuals.setSpatialPosition(p1.pointUniform(500));
}
1: late() {
    i1.evaluate(p1);
    inds = sim.subpopulations.individuals;
    competition = i1.totalOfNeighborStrengths(inds);
    inds.fitnessScaling = 1.1 - competition / size(inds);
}
2: first() {
    i2.evaluate(p1);
}
mateChoice() {
    // spatial mate choice
    return i2.strength(individual);
}
modifyChild() {
    do pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
    while (!p1.pointInBounds(pos));
    child.setSpatialPosition(pos);

    return T;
}
2000 late() { sim.outputFixedMutations(); }

```

Interaction type `i1` is used for competitive interactions, as before. We now declare `i2` as well, which we will use for mate choice. It also uses a Gaussian kernel, this time with a maximum value of `1.0`, a standard deviation of `0.02`, and a maximum distance of `0.1`. This is quite a narrow, short-range mate-choice function that will promote local mating quite strongly.

Having declared `i2`, we then need to evaluate it. We do this in a `first()` event. We haven't seen `first()` events much yet; they are similar to `early()` events, but execute even earlier in the tick cycle (see section 22.0). In this model the distinction makes no difference; we could just as well use an `early()` event. This seems like a good opportunity to introduce `first()` events, though, particularly since in a nonWF model it would be essential for the evaluation to be in a `first()` event rather than an `early()` event, since `early()` events run *after* reproduction in nonWF models (see chapter 23). Putting evaluation of interactions used during reproduction into a `first()` event is a good habit to get into, for that reason.

In any case, having evaluated the interaction we are prepared to use it in the `mateChoice()` callback, which simply returns the result of the call `i2.strength(individual)`. This call asks `i2` to evaluate the strength of interaction between the first parent (`individual`) and all other individuals in its subpopulation. The result is returned as a vector. Happily, that is precisely what `mateChoice()` callbacks are expected to return – a vector of mating weights between the first parent and all other individuals. The result can therefore simply be passed on to SLiM without further processing.

We made one other change from section 15.3: the `fitnessEffect()` callback is gone, and instead we implemented the fitness effect of spatial competition with a couple of lines in the `1: late()` event. After evaluating `i1`, we get a vector of all of the individuals in the simulation, and then call `i1.totalOfNeighborStrengths()` with that vector. This returns a vector with the total interaction strength felt by each individual, evaluated in bulk. We then do a vectorized calculation of the resulting fitness effects for the individuals, and do a vectorized assignment of those fitness effects into the `fitnessScaling` property of the individuals vector. This property simply represents a fitness effect, set by script, that gets multiplied together with all other fitness effects in the model

to produce the final fitness of an individual (see section 14.2 and chapter 13 for further discussion). This is equivalent to the `fitnessEffect()` callback we used in previous models, except that it is a bit more efficient – we can avoid the overhead of all those `fitnessEffect()` callbacks, and the calculations are all vectorized so there is less overhead from Eidos script interpretation.

That is all that is needed; we now have a model that includes both spatial competition and spatial mate choice, using different kernels. If we run this model in SLiMgui, it looks essentially identical to the model of section 15.2, but the evolutionary dynamics under the hood are quite different. We can explore that with a quick modification of these recipes, by dropping in the heterozygosity calculator used in section 14.1. The code from the final output event of that recipe can be used without modification. In a quick experiment, we can do 25 runs of the models, run to tick 10000 to allow equilibration, and output the mean heterozygosity at the end of each run. Let's also do runs of the equivalent non-spatial model (constructed by simply removing all of the interaction code, the `mateChoice()` callback, and all references to spatial positions). Simple summary statistics across the mean heterozygosity values obtained from the 25 runs of each of the three models look like this:

<u>Model</u>	<u>Mean</u>	<u>Std. Deviation</u>
Non-spatial	0.0002148	0.000092
Competition only (15.2)	0.0001934	0.000071
Competition and mate choice (15.4)	0.0001005	0.000067

The first thing to note is that the outcomes of the non-spatial and competition-only models are quite similar. Indeed, even with 25 samples each, a *t*-test finds that they do not differ significantly ( $p = 0.3651$ ). Apparently local offspring generation and spatial competition do not suffice to drive much, if any, genetic divergence among spatial clusters. This is not surprising, since with non-spatial mating in each generation the model is then *de facto* panmictic.

The second thing to note, however, is that the model with spatial mate choice – this section's recipe – is quite different from the other two. This is confirmed by *t*-test; it is highly significantly different from both the non-spatial model ( $p = 0.0000104$ ) and the competition-only model ( $p = 0.0000204$ ). The addition of spatial mate choice (especially with the narrow kernel used here) has driven substantial genetic divergence among spatial clusters. We have ourselves a real spatial model.

## 15.5 Mate choice with a nearest-neighbor search

In this section we will modify the previous recipe to implement spatial mate choice using a nearest-neighbor search instead of a spatial kernel. Each individual choosing a mate will make a choice from among its three nearest neighbors, without further consideration of distance. In this recipe the selection will be random, but it would be simple to make the mate choice depend upon some genetic or non-genetic trait of the prospective mates, reflecting choosy mate selection from within a small pool of nearby candidates.

We will begin with the recipe of section 15.4. To modify it for the present recipe, we need only to replace the `mateChoice()` callback with the following:

```
mateChoice() {
    // nearest-neighbor mate choice
    neighbors = i2.nearestNeighbors(individual, 3);
    return (size(neighbors) ? sample(neighbors, 1) else float(0));
}
```

The `nearestNeighbors()` method returns the three (3) nearest neighbors of the first parent (`individual`). In SLiM, a “neighbor” is an individual within the maximum interaction distance of the focal individual (other than the focal individual itself); individuals beyond that maximum cannot be “neighbors”, even if they are the nearest individual to the focal individual. It is therefore possible for this method to return fewer than three individuals; indeed, if `individual` is spatially isolated this method might return an empty vector. We would like to simply return one individual, chosen at random from the neighbors vector using `sample()`, as the chosen mate; to guard against the possibility of a zero-length result from `nearestNeighbors()`, however, we check the length of neighbors and return `float(0)` if it is zero-length, indicating to SLiM that no mate could be found (see section 25.4). This code uses the Eidos “ternary conditional” operator, `?else`, for brevity; it is like an `if-else` statement, but compacted into a single expression (see the Eidos manual for further discussion).

To digress for a moment: a previous version of this recipe used this callback code instead:

```
mateChoice() {
  // nearest-neighbor mate choice
  neighbors = i2.nearestNeighbors(individual, 3);
  mates = rep(0.0, p1.individualCount);
  mates[neighbors.index] = 1.0;

  return mates;
}
```

This constructs a mating-weight vector with 0 for all potential mates, and then changes the values for the neighbors found (if any) to 1, using the subpopulation indices of the neighbors as indices into the weights vector. This behaves identically to the new version, except that it is *much* slower because SLiM has to scan through the entire returned weights vector with each mating, assessing its meaning and then using the specified weights to choose a mate, which is very inefficient. Returning a singleton `Individual` – a specific chosen mate – from a `mateChoice()` callback is much faster, when possible.

Note that the `nearestNeighbors()` call is purely a spatial query; it does not involve the calculation of interaction strengths at all. In this recipe, the parameters that configure `i2` – the Gaussian function’s maximum value and standard deviation, and indeed the fact that a Gaussian function is to be used at all – are therefore irrelevant and can be removed. The complete recipe:

```
initialize() {
  initializeSLiMOptions(dimensionality="xy");
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);

  // spatial competition
  initializeInteractionType(1, "xy", reciprocal=T, maxDistance=0.3);
  i1.setInteractionFunction("n", 3.0, 0.1);

  // spatial mate choice
  initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.1);
}
1 late() {
  sim.addSubpop("p1", 500);
  p1.individuals.setSpatialPosition(p1.pointUniform(500));
}
```

```

1: late() {
    i1.evaluate(p1);
    inds = sim.subpopulations.individuals;
    competition = i1.totalOfNeighborStrengths(inds);
    inds.fitnessScaling = 1.1 - competition / size(inds);
}
2: first() {
    i2.evaluate(p1);
}
mateChoice() {
    // nearest-neighbor mate choice
    neighbors = i2.nearestNeighbors(individual, 3);
    return (size(neighbors) ? sample(neighbors, 1) else float(0));
}
modifyChild() {
    do pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
    while (!p1.pointInBounds(pos));
    child.setSpatialPosition(pos);

    return T;
}
2000 late() { sim.outputFixedMutations(); }

```

We have now seen three different types of spatial query using `InteractionType`. The first is the use of `totalOfNeighborStrengths()` to add up all of the interactions felt by a focal individual from every other interacting individual; we used this to build spatial competition. The second is `strength()`, which we used to obtain a vector of interaction strengths between the focal individual and all other individuals (it can also calculate the interaction strengths with specific other individuals). And the third is the `nearestNeighbors()` method used here, which returns a vector with (up to) a specified number of the nearest neighbors of a focal individual. `IndividualType` supports several other queries, such as `distance()` to get distances between the focal individual and others, `distanceToPoint()` and `nearestNeighborsOfPoint()` to do those queries using an arbitrary spatial point rather than a focal individual, and `drawByStrength()` to draw neighbors of a focal individual weighted by interaction strength (a fast combination of `nearestNeighbors()`, `strength()`, and `sample()`, conceptually). Finally, there is a variant of `nearestNeighbors()`, called `nearestInteractingNeighbors()`, that limits the nearest-neighbor search to individuals that actually exert an interaction upon the focal individual. This is useful when working with “sex-segregated” interactions that are only exerted by/upon individuals of a particular sex, which `InteractionType` also supports; for example, in a sexual model one might want a mating interaction that is exerted only by males and felt only by females, so that `nearestInteractingNeighbors()` would only return males as potential mates. Instead of providing examples of the use of all these methods – which are all documented in section 24.8.2 – we will now change direction to start building – gradually – towards models of landscape heterogeneity using landscape maps.

## 15.6 Divergence due to phenotypic competition with an `interaction()` callback

Here we will depart from the previous recipes, which have all used neutral spatial models, to explore a different topic: a non-neutral, non-spatial model. This recipe will involve a quantitative trait, using a strategy similar to that in the recipe of section 13.4. It will use an `InteractionType` to model competition among individuals based upon their phenotype: individuals with a more similar phenotype will compete more strongly (since they utilize similar resources). The goal here is to demonstrate the use of an `interaction()` callback to influence the interaction strengths

calculated by SLiM. It is most straightforward to introduce this first in a non-spatial model; in a later section we will see the use of an `interaction()` callback in a spatial model.

This recipe is a first step toward something like the model of Dieckmann & Doebeli (1999): a model demonstrating that phenotypic competition and assortative mating can produce speciation in a non-spatial (i.e. sympatric) model. More specifically, we will be working toward the sexual, “ecological trait” model described in that paper, although our model will be different in many ways – it will be a generational model rather than a continuous-time model, and we will not include the “mating trait” used in that paper, for example. We will continue the development of this model in the next several sections.

Here’s the starting point for the model, without any interaction code yet:

```
initialize() {
  defineConstant("optimum", 5.0);
  defineConstant("sigma_K", 1.0);

  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
  m2.convertToSubstitution = F;

  initializeGenomicElementType("g1", c(m1, m2), c(1, 0.01));
  initializeGenomicElement(g1, 0, 1e5 - 1);
  initializeRecombinationRate(1e-8);
}
mutationEffect(m2) { return 1.0; }
1 late() {
  sim.addSubpop("p1", 500);
}
1: late() {
  inds = sim.subpopulations.individuals;

  // construct phenotypes and fitness effects from QTLs
  phenotypes = inds.sumOfMutationsOfType(m2);
  inds.fitnessScaling = 1.0 + dnorm(phenotypes, optimum, sigma_K);
  inds.tagF = phenotypes;
}
1:2001 late() {
  if (sim.cycle == 1)
    cat("   cyc      mean      sd\n");

  if (sim.cycle % 100 == 1)
  {
    phenotypes = p1.individuals.tagF;
    cat(format("%5d ", sim.cycle));
    cat(format("%6.2f ", mean(phenotypes)));
    cat(format("%6.2f\n", sd(phenotypes)));
  }
}
```

This is a simple model of randomly arising QTLs with additive effects that produce a phenotype; we have seen this sort of model before in sections 13.2 and 13.4 (see those recipes for further discussion). A phenotypic optimum is defined at 5.0, and stabilizing selection around that optimum is produced by the 1: `late()` event, which implements a Gaussian fitness function based on divergence from the optimum. The fitness values are placed in the `fitnessScaling` property of the individuals, and the phenotypes themselves are saved in the `tagF` property of the individuals

for future reference. The individual QTL mutations, of mutation type `m2`, are made effectively neutral by the `mutationEffect(m2)` callback, so that fitness is determined only by the overall phenotype they produce additively. QTL mutations are prevented from being replaced by `Substitution` objects when they fix using `convertToSubstitution`, since they should continue to influence phenotypes. All of this should be familiar from chapter 13; we will be using QTL-based models a lot in this chapter, so it is worth reviewing them now if any of this does not make sense.

At this point, let's pause for a moment. We have stabilizing selection toward an optimum at `5.0`, and we start with a mean phenotype of `0.0` since the model starts with no QTLs defined in the initial population. At the end of every hundredth cycle, the output event prints out the cycle counter with the mean and standard deviation of the population's phenotypic values. A typical run looks something like this:

cyc	mean	sd
1	0.00	0.00
101	-0.00	0.36
201	0.00	0.12
301	0.02	0.26
401	-0.06	0.38
501	-0.21	1.16
601	-0.27	0.91
701	0.12	0.46
801	0.21	0.38
901	0.17	0.30
1001	1.26	1.40
1101	4.52	0.46
1201	4.86	0.50
1301	4.87	0.43
1401	4.93	0.48
1501	5.12	0.39
1601	4.96	0.41
1701	4.97	0.39
1801	5.01	0.35
1901	4.84	0.30
2001	4.92	0.27

It took a little while for the population to develop enough useful variance to be able to reach the fitness peak, but it got there in the end. The population often contains an appreciable amount of variance in the middle of the evolutionary trajectory, but as it settles onto the optimum the variance decreases and tends toward zero, since any deviation from the phenotypic optimum is punished by the fitness function. In practice, as seen above, the standard deviation fluctuates around `0.3` or `0.4` for quite a while after the population reaches the optimum.

Now let's finish the model by implementing competition. We have seen interaction types several times before in this chapter, but here we will use them somewhat differently: to govern competition based upon phenotypic similarity, rather than competition based upon spatial proximity. First, we create an `InteractionType` by adding these lines to the end of the `initialize()` callback:

```
initializeInteractionType(1, "", reciprocal=T);    // competition
i1.setInteractionFunction("f", 1.0);
```

Now we have an interaction type, `i1`, that will evaluate our phenotypic competition interaction. It is a reciprocal interaction, as in previous recipes. It has a spatiality of `""` because this is a non-spatial model. This means that it has no concept of distance, so the only interaction formula we are allowed to use is a fixed value, type `"f"`, as specified here. In its present form this interaction



is not terribly useful; every individual interacts with every other individual with a strength of 1.0. We will fix that momentarily. First, however, let's add code to evaluate and use the interaction, at the end of the 1: late() event:

```
// evaluate phenotypic competition
i1.evaluate(p1);
competition = sapply(inds, "sum(i1.strength(applyValue));");
effects = 1.0 - competition / size(inds);
inds.fitnessScaling = inds.fitnessScaling * effects;
```

This code evaluates `i1`, and then uses `sapply()` to calculate, for each individual, the sum of the strengths of all interactions felt by that individual. The `sapply()` function is a very useful tool, documented in the Eidos manual; in short, it loops over its first parameter, executes the code given (as a string) in its second parameter each time through the loop, and returns a vector that contains the concatenation of all results from the loop. Its effect here is thus very much like running a `for (ind in inds) loop`, with a loop body that executes `sum(i1.strength(ind))` and pastes its results together into a single vector. Using `sapply()` is both more concise and faster than such a loop, however; it is a function worth learning.

In short, `sapply()` provides us with the strength of competition felt by every individual. We then convert that into a fitness effect for each individual, much as we did in sections 15.4 and 15.5. The only twist is that here we need to multiply those fitness effects into the existing values in `inds.fitnessScaling`, since we already used the `fitnessScaling` property for fitness effects due to proximity to the phenotypic optimum. Fitness effects combine multiplicatively, in general, so we multiply the new phenotypic competition fitness effects into the previously calculated phenotypic selection fitness effects to produce final `fitnessScaling` values.

The only hitch is that our interaction is still rather useless; it still evaluates to a fixed interaction strength of 1.0 between every pair of individuals. Now we will make the interaction `i1` more useful by adding an `interaction()` callback:

```
interaction(i1) {
  return dnorm(exerter.tagF, receiver.tagF, sigma_C) / NORM;
}
```

This uses two defined constants, `sigma_C` and `NORM`, that we need to add at the top of the `initialize()` callback:

```
defineConstant("sigma_C", 0.4);
defineConstant("NORM", dnorm(0.0, mean=0, sd=sigma_C));
```

Since this is the first time we've seen an `interaction()` callback, let's examine it closely. It defines the interaction strength of interaction type `i1`, according to its declaration. It uses two variables, `exerter` and `receiver`, that are defined by SLiM; they are pseudo-parameters of the `interaction()` callback, similar to those we have seen with other types of callbacks. The `exerter` is the individual exerting the interaction, and the `receiver` is the individual receiving the interaction; since this interaction is reciprocal, the distinction between the two is arbitrary. The callback looks up the phenotypic values of the two individuals (from their `tagF` properties where we put them before) and uses `dnorm()` to calculate the degree of competition between the two, based upon a Gaussian competition kernel with width `sigma_C`. Finally, the interaction strength is normalized to have a maximum value of 1.0 here by dividing by `NORM` (defined to be the value returned by `dnorm()` for two individuals with identical phenotypes), so that the maximum strength of competition is independent of the width of the competition kernel.

SLiM will now automatically use this `interaction()` callback to determine the strength of `i1` between every pair of individuals. Note that this callback calculates the strength from scratch; it is also possible for an `interaction()` callback to modify the default strength calculated by the interaction function, which is supplied to the callback in the pseudo-parameter `strength`. (Section 25.7 has full documentation regarding `interaction()` callbacks, including all of the pseudo-parameters available to them.)

That completes the model. If we run this full model now in SLiMgui, we should see something like:

cyc	mean	sd
1	0.00	0.00
101	1.62	2.71
201	4.57	1.38
301	4.76	1.41
401	4.87	1.22
501	4.75	1.44
...	...	...
1301	4.98	1.38
1401	5.04	1.60
1501	5.43	1.61
1601	5.01	1.32
1701	5.20	1.30
1801	5.50	1.78
1901	5.22	1.39
2001	5.54	1.75

Note that the phenotypic variance rises to higher levels now, and stays high for the remainder of the run. The theoretical expectation is that the variance will remain high indefinitely, because the presence of phenotypic competition rewards divergence from the mean phenotype.

A key point to understand regarding this model is that the `interaction()` callbacks will be called while fitness values are being calculated, as a side effect of the `i1.strength()` call in the `apply()` call in the `1: late()` event. When `evaluate()` is called on an `InteractionType`, the positions of all individuals are saved in a snapshot, and distances and interaction strengths are calculated based upon that snapshot. When `interaction()` callbacks are involved, however, they are called in a deferred fashion because they are slow; calling them may not be necessary at all, if particular interaction strengths are never queried, so deferring the calls can provide a large performance gain. This means that `interaction()` callbacks are called at a later time than the “official” evaluation of the interaction; indeed, they can be called up until mating begins in the following tick, depending upon when the model queries the interaction. If this proves difficult to manage in a model, you may wish to run the interaction queries you need up front, immediately after calling `evaluate()`, and cache their results for later use; or alternatively, you may wish to defer your call to `evaluate()` until you are ready to actually run your interaction queries. That is not an issue in this model, however – the `interaction()` callbacks are called just after `i1.evaluate()` anyway, as it happens.

The full model looks like this:

```
initialize() {
    defineConstant("optimum", 5.0);
    defineConstant("sigma_K", 1.0);
    defineConstant("sigma_C", 0.4);
    defineConstant("NORM", dnorm(0.0, mean=0, sd=sigma_C));

    initializeMutationRate(1e-6);
    initializeMutationType("m1", 0.5, "f", 0.0);           // neutral
```

```

initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
m2.convertToSubstitution = F;

initializeGenomicElementType("g1", c(m1, m2), c(1, 0.01));
initializeGenomicElement(g1, 0, 1e5 - 1);
initializeRecombinationRate(1e-8);

initializeInteractionType(1, "", reciprocal=T); // competition
i1.setInteractionFunction("f", 1.0);
}
mutationEffect(m2) { return 1.0; }
1 late() {
  sim.addSubpop("p1", 500);
}
1: late() {
  inds = sim.subpopulations.individuals;

  // construct phenotypes and fitness effects from QTLs
  phenotypes = inds.sumOfMutationsOfType(m2);
  inds.fitnessScaling = 1.0 + dnorm(phenotypes, optimum, sigma_K);
  inds.tagF = phenotypes;

  // evaluate phenotypic competition
  i1.evaluate(p1);
  competition = sapply(inds, "sum(i1.strength(applyValue));");
  effects = 1.0 - competition / size(inds);
  inds.fitnessScaling = inds.fitnessScaling * effects;
}
interaction(i1) {
  return dnorm(exerter.tagF, receiver.tagF, sigma_C) / NORM;
}
1:2001 late() {
  if (sim.cycle == 1)
    cat(" cyc      mean      sd\n");

  if (sim.cycle % 100 == 1)
  {
    phenotypes = p1.individuals.tagF;
    cat(format("%5d ", sim.cycle));
    cat(format("%6.2f ", mean(phenotypes)));
    cat(format("%6.2f\n", sd(phenotypes)));
  }
}

```

Since this model does not include assortative mating, it is not expected to produce speciation (and we will see in the next section that it does not). At present, it is merely a model of what Haller & Hendry (2013) called “squashed stabilizing selection”: selection based upon a fitness function that is stabilizing around an optimum, but has been squashed downward at its center due to negative frequency-dependent selection, producing disruptive selection that nevertheless keeps the population in the vicinity of the fitness peak. We will add assortative mating to this model in the recipe of section 15.8, but first, let’s examine a way to improve upon the model as it stands.

## 15.7 Modeling phenotype as a spatial dimension

In the previous section, we built a model of a quantitative trait constructed from randomly arising QTLs that combined additively to determine the phenotype. We then built an interaction to model competition based upon similarity in phenotype. In this section we will modify that recipe

slightly, to better utilize the power of `InteractionType`. Although this model will remain a non-spatial model for the time being, we will now model the phenotype in SLiM as if it were a spatial dimension. This will bring us two benefits: speed, and additional visualization power.

This change is quite straightforward. Beginning with the model of section 15.6, we first need to enable spatiality at the beginning of the `initialize()` callback:

```
initializeSLiMOptions(dimensionality="x");
```

We will use the `x` property of individuals to store their phenotypes now, instead of `tagF`:

```
inds.x = phenotypes;
```

The `interaction()` callback that evaluates phenotypic competition should change to use `x`:

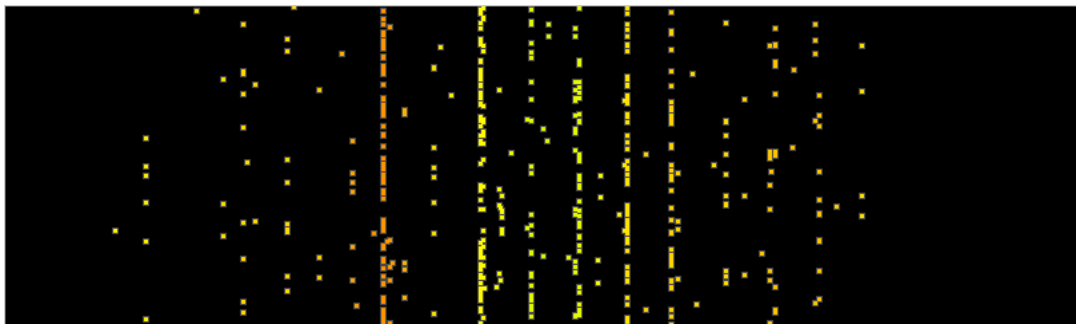
```
interaction(i1) {  
    return dnorm(exerter.x, receiver.x, sigma_C) / NORM;  
}
```

Finally, the output code should also use `x` instead of `tagF` (see the full model below).

Let's make one other change. Previously, we've seen two-dimensional spatial models that used the default boundaries of `[0, 1]` in `x` and `y`. In section 15.3 we mentioned that this default could be changed; let's change it now, because in this model phenotypic values often go outside that range. We can add a line immediately after `p1` is defined, telling `p1` its spatial boundaries:

```
p1.setSpatialBounds(c(0.0, 10.0));
```

That's it; we now have a model in which phenotype is a pseudo-spatial dimension. Why is this better? One reason is that it provides better visualization capabilities in SLiMgui. If we run the model now, we get a display that shows individuals in a one-dimensional phenotypic space (with random `y` coordinates chosen by SLiM to spread the individuals out for greater visibility):



Since we told `p1` that its spatial boundaries were `[0.0, 10.0]`, that is the spatial extent displayed by SLiMgui here, so the phenotypic optimum is at the horizontal center of the display here. This snapshot, taken at about tick 300, shows that the population has already reached the optimum; the mean phenotype here is 4.91, in fact, with a standard deviation of 1.40 – typical of the model's equilibrium state. Colors correspond to fitness as usual; the band of individuals on the left is particularly low-fitness because it is far off of the optimum, and yet is also quite crowded – there are a lot of individuals with that phenotype at this instant in time. All of the discrete banding here is the result of variation in QTLs of relatively large effect. Note that individuals quite far off the fitness peak can still be fairly high-fitness, as long as their phenotype is rare.

So this is one benefit of treating phenotype as a spatial dimension: the dynamic evolution of the phenotypic distribution can be watched in real time, color-coded by fitness as the fitness function resulting from the squashed stabilizing selection fluctuates from tick to tick.

The other benefit is that the model is much faster, because `InteractionType`'s optimizations for spatial interactions are brought to bear. First, if we realize that the Gaussian phenotypic competition function we're presently simulating with an `interaction()` callback is equivalent to one of SLiM's built-in interaction functions, then we can completely remove the `interaction()` callback. We then need to change the definition of the `i1` interaction like so:

```
initializeInteractionType(1, "x", reciprocal=T, maxDistance=sigma_C * 3);
i1.setInteractionFunction("n", 1.0, sigma_C);
```

This defines `i1` as a spatial interaction using `x`, which is our phenotypic "dimension". It then tells `i1` to use a Gaussian function with a maximum value of `1.0` and a width of `sigma_C` – precisely what the `interaction()` callback used to do. Indeed, this model is exactly equivalent; running it with the same random number seed produces the same result as shown in the snapshot above. The difference is that it runs much faster – it completes about 4750 ticks in 30 seconds (on my machine), whereas the version using an `interaction()` callback completes about 420 ticks in 30 seconds. Not a bad speed improvement! The key is that we have avoided having to make a call to an Eidos `interaction()` callback for every interaction; having to run a callback a bunch of times in every tick is always slow.

Indeed, even a little more speed can be squeezed out, at the price of accuracy. If a maximum distance of `0.8` is set for `i1`, and `totalOfNeighborStrengths()` is used instead of `strength()` in the code that calculates total phenotypic competition, about 5030 ticks can be completed in 30 seconds. Since a maximum distance of `0.8` is two standard deviations of the competition kernel, the effects of this change should be relatively small in terms of the overall behavior of the model, but it is nevertheless a sacrifice in accuracy, and the performance gain in this case is not large, so we will not consider this change to be a part of the official recipe for this section. In other cases, however, the performance gain can be very large; a maximum distance should always be considered for spatial interactions, particularly if it can safely be set to a short enough distance to exclude most other individuals from interacting with the focal individual at all.

The full recipe for this section, for posterity, is:

```
initialize() {
  defineConstant("optimum", 5.0);
  defineConstant("sigma_K", 1.0);
  defineConstant("sigma_C", 0.4);
  defineConstant("NORM", dnorm(0.0, mean=0, sd=sigma_C));

  initializeSLiMOptions(dimensionality="x");
  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
  m2.convertToSubstitution = F;

  initializeGenomicElementType("g1", c(m1, m2), c(1, 0.01));
  initializeGenomicElement(g1, 0, 1e5 - 1);
  initializeRecombinationRate(1e-8);

  initializeInteractionType(1, "x", reciprocal=T,
    maxDistance=sigma_C * 3);
  i1.setInteractionFunction("n", 1.0, sigma_C);
}
```

```

mutationEffect(m2) { return 1.0; }
1 late() {
  sim.addSubpop("p1", 500);
  p1.setSpatialBounds(c(0.0, 10.0));
}
1: late() {
  inds = sim.subpopulations.individuals;

  // construct phenotypes and fitness effects from QTLs
  phenotypes = inds.sumOfMutationsOfType(m2);
  inds.fitnessScaling = 1.0 + dnorm(phenotypes, optimum, sigma_K);
  inds.x = phenotypes;

  // evaluate phenotypic competition
  i1.evaluate(p1);
  competition = sapply(inds, "sum(i1.strength(applyValue));");
  effects = 1.0 - competition / size(inds);
  inds.fitnessScaling = inds.fitnessScaling * effects;
}
1:2001 late() {
  if (sim.cycle == 1)
    cat("   cyc      mean      sd\n");

  if (sim.cycle % 100 == 1)
  {
    phenotypes = p1.individuals.x;
    cat(format("%5d   ", sim.cycle));
    cat(format("%6.2f   ", mean(phenotypes)));
    cat(format("%6.2f\n", sd(phenotypes)));
  }
}

```

## 15.8 Sympatric speciation facilitated by assortative mating

In the previous section we refined our non-spatial model of phenotypic competition, by treating phenotype as though it were a spatial dimension so that SLiM could run the model more quickly and with better visualization. It is now time to take another step toward the model of Dieckmann & Doebeli (1999) by changing mating in the model to be assortative by phenotype. This will in some ways be similar to what we did in section 15.4; but there mating was spatially assortative, whereas here mating will be phenotypically assortative.

Beginning with the model of section 15.7, the changes needed are quite small. First, we need to add code in `initialize()` to define a new interaction type, `i2`, for mate evaluation:

```

initializeInteractionType(2, "x", reciprocal=T, maxDistance=sigma_M * 3);
i2.setInteractionFunction("n", 1.0, sigma_M);

```

We have to define the `sigma_M` kernel width as well, at the beginning of `initialize()`:

```

defineConstant("sigma_M", 0.5);

```

Since this interaction is based on "x", which is our pseudo-spatial phenotype dimension, it represents phenotypic proximity just as `i1`, our phenotypic competition interaction, does. Indeed, the only reason not to use `i1` to govern mate choice as well is that we want to be able to make it use a different interaction function than competition, so that we can play with the width of the mate-choice kernel independently of the width of the competition kernel.

Next, we need to evaluate that interaction in a `first()` event:



```

1: first() {
    // evaluate mate choice in preparation for reproduction
    i2.evaluate(p1);
}

```

And finally, we need a mate choice callback that uses that interaction:

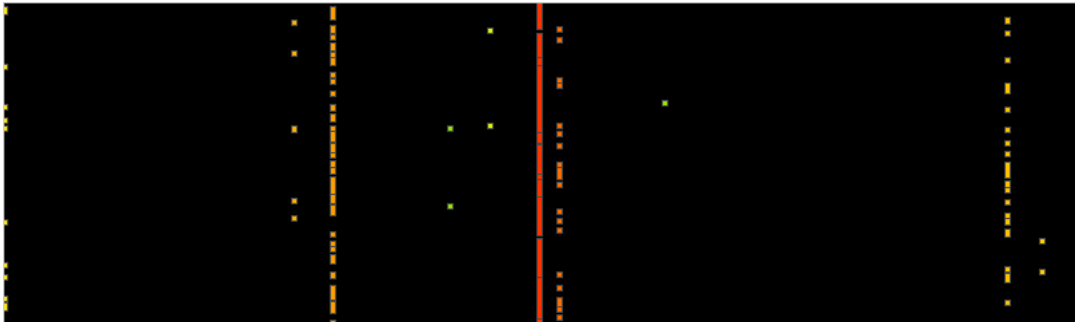
```

mateChoice() {
    // spatial mate choice
    return i2.strength(individual);
}

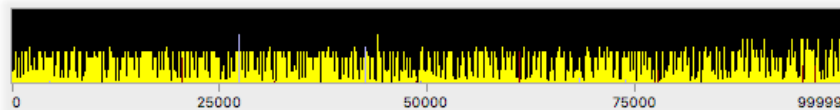
```

And that's it. We now have a model in which phenotype, as determined by randomly arising additive QTLs, is something like a “magic trait” (Servedio et al. 2011) – a trait that simultaneously determines both fitness (because of the phenotypic optimum) and assortative mating (because of the `mateChoice()` callback) (although it is perhaps not strictly a magic trait since it is governed by underlying QTLs; see sections 11.1 and 13.3 for more discussion of this topic). When exposed to the disruptive selection caused by phenotypic competition in the model, the magic-ish trait here readily produces speciation. (Note that this is true speciation, not evolutionary branching, because this is a sexual model in the sense that matters – biparental mating with assortment and recombination of gametes – even though it models hermaphrodites, not separate sexes).

We can see evidence for speciation in two different ways in SLiMgui. First of all, instead of the cloud of different phenotypes around the optimum that the previous model generated, we now see a set of discrete phenotypic clusters:



Second, we now see strong genetic separation between these clusters in the pattern of neutral variation exhibited by the population:



Note that there are no neutral mutations at high frequency at all; instead, there is a large amount of neutral diversity that is pinned at a couple of intermediate frequencies. These are blocs of neutral mutations that have fixed within one of the species in the model, but are unable to spread more widely because of the reproductive isolation between the species. Hybridization is not impossible; it does occur occasionally, as can be seen in the snapshot above. But it is rare enough that gene flow between the species is insufficient to allow that neutral diversity to spread.

It would be fair to argue that although this is a model of speciation given a pre-existing magic trait, it is not a model of the emergence of the magic trait itself, because it lacks the “mating trait” of the Dieckmann & Doebeli (1999) model. Adding in such a trait would be a simple extension of

the present model, and would presumably support the result found by Dieckmann & Doebeli (1999) – that in an ecological scenario such as this, assortative mating will readily emerge, transforming an ordinary trait into a magic trait that then facilitates speciation. We will leave that extension of the model as an exercise for the reader, since pursuing it would not serve the aims of this chapter. Instead, in the next section we will turn back toward spatial models, while incorporating what we have done here with QTLs, phenotypic competition, and phenotype-based assortative mate choice.

Here's the full model, for the record:

```
initialize() {
  defineConstant("optimum", 5.0);
  defineConstant("sigma_K", 1.0);
  defineConstant("sigma_C", 0.4);
  defineConstant("sigma_M", 0.5);
  defineConstant("NORM", dnorm(0.0, mean=0, sd=sigma_C));

  initializeSLiMOptions(dimensionality="x");
  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
  m2.convertToSubstitution = F;

  initializeGenomicElementType("g1", c(m1, m2), c(1, 0.01));
  initializeGenomicElement(g1, 0, 1e5 - 1);
  initializeRecombinationRate(1e-8);

  initializeInteractionType(1, "x", reciprocal=T,
    maxDistance=sigma_C * 3); // competition
  i1.setInteractionFunction("n", 1.0, sigma_C);

  initializeInteractionType(2, "x", reciprocal=T,
    maxDistance=sigma_M * 3); // mate choice
  i2.setInteractionFunction("n", 1.0, sigma_M);
}
mutationEffect(m2) { return 1.0; }
1 late() {
  sim.addSubpop("p1", 500);
  p1.setSpatialBounds(c(0.0, 10.0));
}
1: late() {
  inds = sim.subpopulations.individuals;

  // construct phenotypes and fitness effects from QTLs
  phenotypes = inds.sumOfMutationsOfType(m2);
  inds.fitnessScaling = 1.0 + dnorm(phenotypes, optimum, sigma_K);
  inds.x = phenotypes;

  // evaluate phenotypic competition
  i1.evaluate(p1);
  competition = sapply(inds, "sum(i1.strength(applyValue));");
  effects = 1.0 - competition / size(inds);
  inds.fitnessScaling = inds.fitnessScaling * effects;
}
2: first() {
  // evaluate mate choice in preparation for reproduction
  i2.evaluate(p1);
}
```

```

mateChoice() {
  // spatial mate choice
  return i2.strength(individual);
}
1:2001 late() {
  if (sim.cycle == 1)
    cat("   cyc   mean       sd\n");

  if (sim.cycle % 100 == 1)
  {
    phenotypes = p1.individuals.x;
    cat(format("%5d ", sim.cycle));
    cat(format("%6.2f ", mean(phenotypes)));
    cat(format("%6.2f\n", sd(phenotypes)));
  }
}

```

## 15.9 Speciation due to spatial variation in selection

In the previous section, we observed that speciation can occur as a result of phenotypic competition and assortative mate choice in a non-spatial model. In this section we will return to spatial modeling, while preserving many aspects of the previous recipe. We will now introduce spatial variation in selection, and will observe adaptive speciation among spatial groups as a result of local selection pressures in combination with phenotypic competition. The optimum trait value for the quantitative trait – the phenotypic optimum, in other words – will vary according to the x position in space, producing a linear environmental gradient. This model is broadly inspired by the model of Doebeli & Dieckmann (2003), although again there are many differences.

This recipe has enough differences from the previous recipe that we will build it here from scratch, rather than just giving changes relative to the previous recipe. Let's start with the initialization and setup:

```

initialize() {
  defineConstant("sigma_C", 0.1);
  defineConstant("sigma_K", 0.5);
  defineConstant("sigma_M", 0.1);
  defineConstant("slope", 1.0);
  defineConstant("N", 500);

  initializeSLiMOptions(dimensionality="xyz");
  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
  m2.convertToSubstitution = F;

  initializeGenomicElementType("g1", c(m1, m2), c(1, 0.1));
  initializeGenomicElement(g1, 0, 1e5 - 1);
  initializeRecombinationRate(1e-8);

  initializeInteractionType(1, "xyz", reciprocal=T,
    maxDistance=sigma_C * 3); // competition
  i1.setInteractionFunction("n", 1.0, sigma_C);

  initializeInteractionType(2, "xyz", reciprocal=T,
    maxDistance=sigma_M * 3); // mate choice
  i2.setInteractionFunction("n", 1.0, sigma_M);
}

```

```

mutationEffect(m2) { return 1.0; }
1 late() {
  sim.addSubpop("p1", N);
  p1.setSpatialBounds(c(0.0, 0.0, -slope, 1.0, 1.0, slope));
  p1.individuals.setSpatialPosition(p1.pointUniform(N));
  p1.individuals.z = 0.0;
}

```

This model is our first to use dimensionality "xyz". The x and y dimensions are true spatial dimensions; this is a 2-D model. The z dimension will be used here for phenotype, just as we used x as a pseudo-spatial phenotypic dimension in previous recipes. We set up QTL machinery for the phenotype much as we did before; the fraction of mutations that are QTLs is higher in this model just so that we don't have to wait as long to get interesting behavior. We create two interaction types, i1 for competition and i2 for mating, as before. Note that both of these interaction types have spatiality "xyz"; i1 therefore encompasses both spatial and phenotypic competition in a single interaction, and i2 encompasses both spatially and phenotypically assortative mate choice. For our purposes here this is sufficient, but more interaction types could be defined as desired.

Then we set up the population, in the 1 late() event. We set spatial boundaries of [0.0, 1.0] for x and y, and of [-slope, slope] for z. The optimum phenotype will actually vary from -slope/2 to slope/2, from the left edge to the right edge of the environment; the wider interval here is used just to allow for some phenotypic variation beyond that interval. (In fact, the spatial boundary for z is not used in this model, nor by SLiMgui, so it is irrelevant anyway). Finally, we set random spatial positions for all of the initial individuals, and zero out their phenotypes (z).

Next, let's implement the machinery to manage spatial positions and phenotypes:

```

modifyChild() {
  // set offspring position based on parental position
  do pos = c(parent1.spatialPosition[0:1] + rnorm(2, 0, 0.005), 0.0);
  while (!p1.pointInBounds(pos));
  child.setSpatialPosition(pos);

  return T;
}
1: late() {
  inds = sim.subpopulations.individuals;

  // construct phenotypes and fitness effects from QTLs
  phenotypes = inds.sumOfMutationsOfType(m2);
  optima = (inds.x - 0.5) * slope;
  inds.fitnessScaling = 1.0 + dnorm(phenotypes, optima, sigma_K);
  inds.z = phenotypes;

  // color individuals according to phenotype
  for (ind in inds)
  {
    hue = ((ind.z + slope) / (slope * 2)) * 0.66;
    ind.color = rgb2color(hsv2rgb(c(hue, 1.0, 1.0)));
  }

  // evaluate phenotypic competition
  i1.evaluate(p1);
  competition = sapply(inds, "sum(i1.strength(applyValue));");
  effects = 1.0 - competition / size(inds);
  inds.fitnessScaling = inds.fitnessScaling * effects;
}

```

```

2: first(p1) {
  // evaluate mate choice in preparation for reproduction
  i2.evaluate();
}
mateChoice() {
  // spatial mate choice
  return i2.strength(individual);
}

```

As in previous recipes, we use a `modifyChild()` callback to place offspring near their first parent, with a small random deviation. We ensure that the offspring phenotype is `0.0` here, so that it doesn't cause the `pointInBounds()` call to return `F`; we don't want SLiM to bounds-check offspring phenotypes for us. The phenotype of `0.0` set here is overwritten a moment later, in the 1: `late()` event above, when phenotypes are calculated for all individuals and placed into their `z` coordinate. That code also calculates fitness effects due to phenotypic proximity to the environmental optimum; in previous recipes the optimum was a constant value, but now the optimum depends upon the individual's position in space, since we are now modeling a linear spatial environmental gradient, so we compute a vector of optima that we can pass to `dnorm()` to do a vectorized assessment of local adaptation on the gradient.

Next, a bit of extra code here calculates color values for individuals; in this model individuals are colored according to their phenotype, rather than their fitness, so that spatial adaptive divergence is directly visible in SLiMgui, as we'll see below. The code here calculates a hue in the HSV (hue/saturation/value) color system, then converts that to the RGB (red/green/blue) color system and then to a hexadecimal color string which it sets as the color of the individual (see the Eidos manual for details on these functions).

Next, we evaluate competition, both spatial and phenotypic; it is actually unchanged from the previous recipe. Similarly, the `mateChoice()` callback above is unchanged from the previous section, although it now scores mates based upon both spatial and phenotypic similarity. The `strength()` method simply evaluates interaction strength; here that is based upon both spatial position and phenotype, since the interaction now encompasses both, but the code using `strength()` does not need to change at all.

Finally, let's use the same output event that we used before:

```

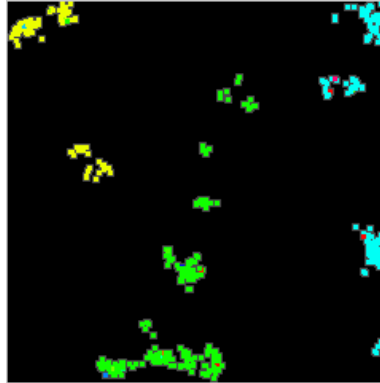
1:5001 late() {
  if (sim.cycle == 1)
    cat("   cyc   mean      sd\n");

  if (sim.cycle % 100 == 1)
  {
    phenotypes = p1.individuals.z;
    cat(format("%5d ", sim.cycle));
    cat(format("%6.2f ", mean(phenotypes)));
    cat(format("%6.2f\n", sd(phenotypes)));
  }
}

```

This prints out a history of the phenotypic mean and standard deviation every 100 cycles, with a header emitted in cycle 1.

Running this model does indeed produce speciation. One sign of that is that we can see discrete clusters of individuals of different colors, representing the fact that they have adapted to different parts of the environmental gradient:



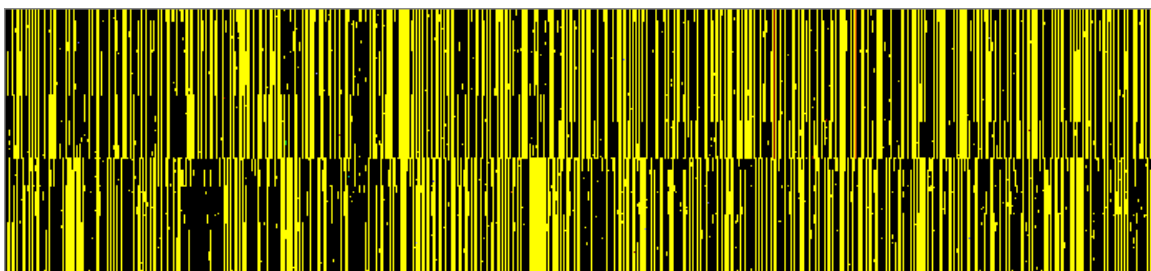
There are three species present here, colored yellow, green, and cyan, adapted to the conditions at the left, center, and right of the environmental gradient, respectively. (There are also a few mutants of other colors sprinkled in.)

Another piece of evidence for speciation is the presence of large amounts of neutral diversity that is not mixing between the different phenotypic clusters, as we saw before in section 15.8:



Many mutations are at a frequency of approximately 2/3 because they are shared between two species; the yellow species split from the green species only about 1000 ticks before these snapshots were taken, and so the two species share a great deal of their genetic background. Many other mutations are at a frequency of approximately 1/3 because they are shared only within the cyan species, which split from the green species more than 8000 ticks earlier. Remarkably, not a single mutation has fixed across the whole population after 9000 ticks of runtime; the onset of speciation is quite early (in this run of the model, at least), and the reproductive barrier is quite strong with the parameter values used here.

One could bring in other machinery to further assess the extent of adaptive divergence and reproductive isolation, such as the  $F_{ST}$  calculation function, `calcFST()`, that we used in section 11.1. One could also use SLiMgui's "Create Haplotype Plot" feature to see whether the groups that appear to be species fall out as genetic clusters naturally (see sections 13.3 and 14.4 for previous uses of this feature). For a different run of the model than shown above, with only two species present, a haplotype plot looks like this:



The two species are clearly visible in the haplotype plot, with completely different neutral variation along the whole chromosome. Within each species some haplotype structure is also visible; this is divergence among the separate spatial clusters within each species.



The full model, for reference:

```
initialize() {
  defineConstant("sigma_C", 0.1);
  defineConstant("sigma_K", 0.5);
  defineConstant("sigma_M", 0.1);
  defineConstant("slope", 1.0);
  defineConstant("N", 500);

  initializeSLiMOptions(dimensionality="xyz");
  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
  m2.convertToSubstitution = F;

  initializeGenomicElementType("g1", c(m1, m2), c(1, 0.1));
  initializeGenomicElement(g1, 0, 1e5 - 1);
  initializeRecombinationRate(1e-8);

  initializeInteractionType(1, "xyz", reciprocal=T,
    maxDistance=sigma_C * 3); // competition
  i1.setInteractionFunction("n", 1.0, sigma_C);

  initializeInteractionType(2, "xyz", reciprocal=T,
    maxDistance=sigma_M * 3); // mate choice
  i2.setInteractionFunction("n", 1.0, sigma_M);
}
mutationEffect(m2) { return 1.0; }
1 late() {
  sim.addSubpop("p1", N);
  p1.setSpatialBounds(c(0.0, 0.0, -slope, 1.0, 1.0, slope));
  p1.individuals.setSpatialPosition(p1.pointUniform(N));
  p1.individuals.z = 0.0;
}
modifyChild() {
  // set offspring position based on parental position
  do pos = c(parent1.spatialPosition[0:1] + rnorm(2, 0, 0.005), 0.0);
  while (!p1.pointInBounds(pos));
  child.setSpatialPosition(pos);

  return T;
}
1: late() {
  inds = sim.subpopulations.individuals;

  // construct phenotypes and fitness effects from QTLs
  phenotypes = inds.sumOfMutationsOfType(m2);
  optima = (inds.x - 0.5) * slope;
  inds.fitnessScaling = 1.0 + dnorm(phenotypes, optima, sigma_K);
  inds.z = phenotypes;

  // color individuals according to phenotype
  for (ind in inds)
  {
    hue = ((ind.z + slope) / (slope * 2)) * 0.66;
    ind.color = rgb2color(hsv2rgb(c(hue, 1.0, 1.0)));
  }
}
```

```

    // evaluate phenotypic competition
    i1.evaluate(p1);
    competition = sapply(inds, "sum(i1.strength(applyValue));");
    effects = 1.0 - competition / size(inds);
    inds.fitnessScaling = inds.fitnessScaling * effects;
}
2: first() {
    // evaluate mate choice in preparation for reproduction
    i2.evaluate(p1);
}
mateChoice() {
    // spatial mate choice
    return i2.strength(individual);
}
1:5001 late() {
    if (sim.cycle == 1)
        cat("   cyc      mean      sd\n");

    if (sim.cycle % 100 == 1)
    {
        phenotypes = p1.individuals.z;
        cat(format("%5d ", sim.cycle));
        cat(format("%6.2f ", mean(phenotypes)));
        cat(format("%6.2f\n", sd(phenotypes)));
    }
}
}

```

## 15.10 A simple biogeographic landscape model

The time has come to introduce a new topic: landscape maps. For this, we will temporarily abandon the spatial quantitative-trait local adaptation model we have been building in previous recipes, and switch to a much simpler model. Our goal in this section is to build a biogeographic model that uses a landscape map. Just for fun, the map we will use is a map of the world. There are no end of issues with this – the map projection distorts the geography, Russia and Alaska are not connected across the Bering Strait because the spatial boundary intervenes, and so forth. But as a simple toy model to illustrate the concept and the potential of landscape maps, it should serve our purposes.

Let's start with the model, and then delve into the concepts:

```

initialize() {
    initializeSLiMOptions(dimensionality="xy");
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);

    // spatial competition
    initializeInteractionType(1, "xy", reciprocal=T, maxDistance=30.0);
    i1.setInteractionFunction("n", 5.0, 10.0);

    // spatial mate choice
    initializeInteractionType(2, "xy", reciprocal=T, maxDistance=30.0);
    i2.setInteractionFunction("n", 1.0, 10.0);
}

```

```

1 late() {
  sim.addSubpop("p1", 1000);

  p1.setSpatialBounds(c(0.0, 0.0, 539.0, 216.0));

  mapImage = Image("world_map_540x217.png");
  p1.defineSpatialMap("world", "xy", 1.0 - mapImage.floatK,
    valueRange=c(0.0, 1.0), colors=c("#0000CC", "#55FF22"));

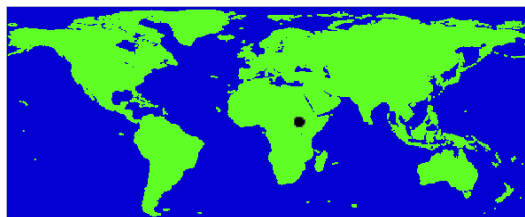
  // start near a specific map location
  for (ind in p1.individuals) {
    ind.x = rnorm(1, 300.0, 1.0);
    ind.y = rnorm(1, 100.0, 1.0);
  }
}
1: late() {
  i1.evaluate(p1);
  inds = sim.subpopulations.individuals;
  competition = i1.totalOfNeighborStrengths(inds) / size(inds);
  competition = pmin(competition, 0.99);
  inds.fitnessScaling = 1.0 - competition;
}
2: first() {
  i2.evaluate(p1);
}
mateChoice() {
  return i2.strength(individual);
}
modifyChild() {
  do pos = parent1.spatialPosition + rnorm(2, 0, 2.0);
  while (!p1.pointInBounds(pos));

  // prevent dispersal into water
  if (p1.spatialMapValue("world", pos) == 0.0)
    return F;

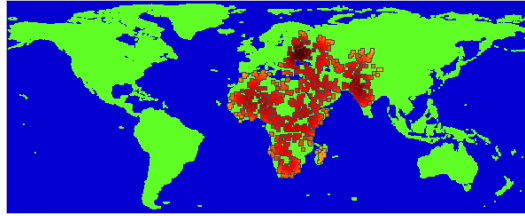
  child.setSpatialPosition(pos);
  return T;
}
20000 late() { sim.outputFixedMutations(); }

```

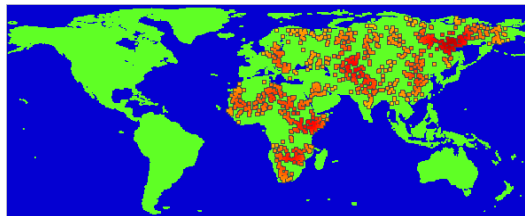
Here's what the model looks like in SLiMgui at the end of tick 1, after population setup:



The black dot in eastern Africa is the initial population; this model seeds the population at a specific location, perhaps simulating – not realistically, obviously! – the origin of *Homo sapiens* in Africa. This model has spatial competition and spatial mate choice, as we have seen in previous recipes, so the population rapidly spreads to occupy new territory in order to escape the local competition from other individuals. Here is the population at the end of tick 63:



Notice that the individuals are constrained to occupy only land locations. It is also interesting that they have managed to reach Madagascar; the dispersal kernel used can jump small distances, so the population can sometimes bridge the Mozambique Channel. Some water gaps are too large to bridge, however. Here is the population at the end of tick 1000:



The population has spread across all of mainland Asia, but has not been able to enter Indonesia, and has thus not reached Australia; the land area provided by the Indonesian islands is probably too small to support a subpopulation (and similarly, the subpopulation that colonized Madagascar has died out). Reaching the New World would be even more difficult, given the fact that the Bering Strait is unavailable (a periodic boundary condition in the x direction would be needed to make that work; see section 15.12); individuals would have to make like the Vikings and jump from mainland Europe to Iceland, Greenland, and thence to North America.

The fact that this particular biogeographic pattern is observed is not necessarily a problem, of course; a great many organisms are unable to disperse from Eurasia to Australia or the Americas. But it is a consequence of the details of this model – the particular dispersal kernel chosen, the particular way in which dispersal is constrained to land, the details of the map used (the fact that it does not include elevation, rivers, etc.), the way in which competition and mating are implemented, and so forth. There is no obstacle to changing these details to match the biological details of a particular species, to produce a more empirically based biogeographic model.

So, how was this recipe constructed? Let's now delve into a few of the details. Most of the model is familiar boilerplate: the setup in `initialize()`, the `1: late()` event that evaluates the spatial interactions, and the `mateChoice()` callbacks, for example. Let's start, then, by looking at the `1 late()` event that initializes the population:

```
1 late() {
  sim.addSubpop("p1", 1000);

  p1.setSpatialBounds(c(0.0, 0.0, 539.0, 216.0));

  mapImage = Image("world_map_540x217.png");
  p1.defineSpatialMap("world", "xy", 1.0 - mapImage.floatK,
    valueRange=c(0.0, 1.0), colors=c("#0000CC", "#55FF22"));

  // start near a specific map location
  for (ind in p1.individuals) {
    ind.x = rnorm(1, 300.0, 1.0);
    ind.y = rnorm(1, 100.0, 1.0);
  }
}
```

It begins by creating a subpopulation, `p1`, as usual. It then sets up spatial boundaries for `p1`, from (0, 0) to (539, 216); this is dictated by the size of the map that we are about to load, which is 540x217 pixels in size. Note that the bounds are made *one smaller* than the image size, in fact, in each dimension; this is because the spatial map's grid is aligned with the corners of the bounds, and so only half of the area of edge pixels (and one-quarter of the area of corner pixels) lies inside the spatial bounds, as will be illustrated in the next section. The spatial bounds of the subpopulation do not need to derive exactly from the size of the spatial map in this way, but it is usually desirable for them to at least match the aspect ratio of the map, so that the map is not stretched.

The map is then read in from a file; note that the path to this file will probably be different on your system (it can be found inside the Recipes folder that can be downloaded from SLiM's home page). This is a simple black-and-white PNG image file, which we load using the Eidos class `Image` (see the Eidos manual for documentation on that class). We get the black channel from the image, as `float` values in `[0,1]`, with the `floatK` property (for a color image, RGB channels would be defined, but for a grayscale image only a single channel is defined, named "K", as in the standard color space "CMYK"). We don't need to do any coordinate transformations; `Image` and `defineSpatialMap()` take care of that for us. We do, however, need to flip the sense of the pixels in the image – it depicts land as black pixels (0), but we want land to be 1 and water to be 0, so we transform the values with `1.0 - mapImage.floatK`.

Now we call the `defineSpatialMap()` method of `Subpopulation` to give the map to SLiM. The first parameter, "world", is a name for the map; you can define as many spatial maps as you wish, and you refer to them by name. The "xy" parameter indicates the spatiality of the map, which must be a subset of the dimensionality of the model as a whole; here our map covers dimensions `x` and `y`. Next we provide the values for the map as a matrix of `float` values (if we were creating a 1D spatial map we would pass a vector; if a 3D spatial map, an array). (Note that the Eidos manual discusses how matrices and arrays work in more detail.)

The last two parameters here are optional, and are for SLiMgui's benefit. The first parameter, `valueRange`, gives the expected range of values for the map data; this need not match the actual range of the data. The purpose of this is to tell SLiMgui what value range should be displayed using distinct colors; values outside this range will be clamped to be within the range for display. The second parameter, `colors`, gives a vector of color strings that specify how particular values should be displayed (see the Eidos manual for details on color strings). The lowest value in `valueRange` will be displayed using the first color in `colors`; the highest value in `valueRange` will be displayed using the last color in `colors`. Color values in between will be evenly distributed across `valueRange`, and intermediate values – those not corresponding exactly to a given color – will be displayed using an interpolation between the two nearest color values. Here, the values supplied for these parameters indicate that a value of 0.0 should be displayed with the color "#0000CC" (dark blue), whereas 1.0 should use color "#55FF22" (medium green). Values between 0.0 and 1.0 would be interpolated, but in fact our map data is binary, so that does not arise.

With that, we have defined a spatial map and told SLiMgui how to display it. We now just need to use it to govern the model dynamics. At the end of the `1 late()` event the positions of all individuals are initialized to be near a particular spot in East Africa, using `rnorm()` to draw the coordinates instead of using the `pointUniform()` method we have used in previous recipes. (See the SLiM-Extras GitHub repository for a modified recipe that draws initial positions anywhere on land, using `pointUniform()` with a vectorized iteration strategy for efficiency.)

Then in our `modifyChild()` callback we enforce that individuals can only disperse to land, by consulting the spatial map:

```

modifyChild() {
  do pos = parent1.spatialPosition + rnorm(2, 0, 2.0);
  while (!p1.pointInBounds(pos));

  // prevent dispersal into water
  if (p1.spatialMapValue("world", pos) == 0.0)
    return F;

  child.setSpatialPosition(pos);
  return T;
}

```

The first couple of lines generate a random offspring position based upon the position of the first parent, as we have seen before. That code loops until a point inside the spatial bounds of the subpopulation is generated. Next, the callback consults the spatial map to find out its value at the candidate point using the `spatialMapValue()` method, giving it map name and the candidate point. This method simply looks up the value nearest to the given point in the map data. If it is `0.0` – water – then the callback returns `F`, rejecting the proposed child. Otherwise, the location is on land, so it is set as the new child's position and `T` is returned to accept the proposed child.

Notice, then, that a reprising boundary condition is used for the edges of the map, but an absorbing boundary condition is used for the coastlines. This means that individuals close to a coastline suffer a fitness penalty; their children are relatively likely to land in water, and when they do, they don't get another chance to generate a different child location. This is part of the reason that colonizing areas like Indonesia is difficult in this model (the fact that locations in places like Indonesia are relatively unlikely to be chosen as child locations in the first place is also important). We can change the `modifyChild()` callback's code a bit to change that:

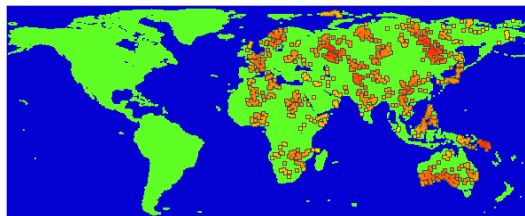
```

modifyChild() {
  do {
    do pos = parent1.spatialPosition + rnorm(2, 0, 2.0);
    while (!p1.pointInBounds(pos));
  } while (p1.spatialMapValue("world", pos) == 0.0);

  child.setSpatialPosition(pos);
  return T;
}

```

Now the boundary condition on coastlines is reprising also, and a quick run of the model shows that that allows much more dispersal into small land areas:



Indonesia and Australia are soon reached, and the UK and Madagascar often also support populations. Reaching North America is still quite difficult, however.

This is obviously just a fun toy model, but it would be easy to extend. A map using more values, to indicate things like mountain ranges, could be used, and the map could be much higher resolution (SLiM places no limit on the size of spatial maps, as long as your computer has enough memory). Dispersal on a high-resolution map could be much more short-range, making it so that even small barriers like rivers would present obstacles to dispersal. A much larger population size



would probably be appropriate, for most organisms. Moreover, instead of using a constant population size as this model does, which is clearly unrealistic, the population size could be related to fitness in such a way that when the population discovers a new area and expands into it, thereby increasing in fitness since the effects of competition are diminished, the population size would increase to reflect the new higher carrying capacity; that would happen naturally in a nonWF model (see section 16.10), so that would probably be a better design than the WF model shown here. Mate choice could be based not only on spatial distance, but also upon, for example, the map value at a point intermediate between the two proposed parents, so that a mountain range would produce vicariance between even closely adjacent subpopulations. All of these sorts of features could be added with just a few more lines of code. Note also that although this model reads its spatial map in from an image file, it is also straightforward to construct a map at runtime in script; see `twoHabitatMap.txt` in the online SLiM-Extras repository.

One particularly interesting feature for a biogeographic model like this would be to include spatial heterogeneity that affects the selection on individuals, such that there is selection pressure towards local adaptation. For instance, with this world map the more polar regions might exert selection toward more cold-adapted phenotypes whereas the more tropical regions might select for more warm-adapted phenotypes. In section 15.9 we saw a very primitive stab at this sort of model, with the introduction of a simple linear environmental gradient. Real landscapes are more complicated than that, though – mountaintops are similar to polar environments, whereas coastlines tend to have more moderate climates, for example. In the next section, we'll look at a model of local adaptation to spatial heterogeneity that goes beyond a simple linear environmental gradient by using a spatial map.

### 15.11 Local adaptation on a heterogeneous landscape map

The previous section introduced the ability to define an arbitrary landscape map, whereas in section 15.9 we explored a model of adaptation to a linear environmental gradient. One could try combining those two approaches, to see how complex environmental heterogeneity influences evolutionary outcomes like divergence and speciation. This recipe will demonstrate such an approach, inspired by the model of Haller, Mazzucco & Dieckmann (2013), although the landscapes generated here will be fairly different from those used in that paper. This recipe will generate a random “landscape map” representing the local phenotypic optimum across the landscape, and will then simulate the local adaptation of a spatial population to the conditions of that landscape. Here the landscape is generated algorithmically, but it would be trivial to instead read a landscape map from a file as was done in the previous recipe.

The model of section 15.9 provides us with a quantitative trait based upon underlying randomly arising QTLs, with phenotypic competition and assortative mating based upon that quantitative trait. Let's assume that machinery for now (the full recipe will be presented at the end), and look at the parts that are new. Most importantly, we have gotten rid of the slope constant of section 15.9, and instead here generate a heterogeneous landscape:

```
1 late() {
  sim.addSubpop("p1", N);
  p1.setSpatialBounds(c(0.0, 0.0, 0.0, 1.0, 1.0, 1.0));
  p1.individuals.setSpatialPosition(p1.pointUniform(N));
  p1.individuals.z = 0.0;

  defineConstant("mapValues", matrix(runif(25, 0, 1), ncol=5));
  p1.defineSpatialMap("map1", "xy", mapValues, interpolate=T,
    valueRange=c(0.0, 1.0), colors=c("red", "yellow"));
}
```

We define spatial bounds of [0, 1] for each dimension, including the z dimension that is used for phenotype, and set up the initial population at random positions on the landscape with phenotypic values of 0.0, as in section 15.9. A map is then generated simply as a set of 25 draws from a uniform distribution between 0 and 1, transformed into a 5×5 matrix. This map, saved off in the defined constant `mapValues` (we will use it later), is set as a spatial map on subpopulation `p1` with a call to `defineSpatialMap()`. We specify that it corresponds to dimensions "xy", that values are expected to range between 0.0 and 1.0, and that colors for that range should ramp from red to yellow. This is basically familiar from the biogeographic model of section 15.10, although the spatial map there came from a file.

We'll look at what this map ends up looking like in a moment, but first let's finish building the model. In the 1: `late()` event, we will replace the code from section 15.9 that colors individuals according to phenotype with new code that uses the same color scheme as the landscape coloring:

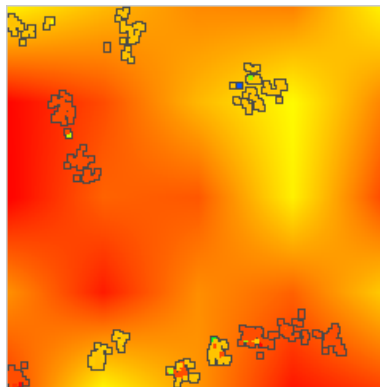
```
// color individuals according to phenotype
inds.color = p1.spatialMapColor("map1", phenotype);
```

This uses a vectorized call to the `spatialMapColor()` method of `Subpopulation`, which translates a value into a color using the mapping established by a particular spatial map. In this way, the colors of individuals adapted to a given phenotypic optimum will exactly match the colors of map areas where that phenotypic optimum exists. Individuals that are perfectly adapted to their environment will therefore be displayed in SLiMgui in a color that matches their environment, whereas a mismatch in color between an individual and its environment will indicate maladaptation.

Finally, for evaluating the fitness effects due to individual phenotype compared to the phenotypic optimum, we need to use `spatialMapValue()` to find the phenotypic optimum for the point in space occupied by each individual. The map is looked up by name, as we saw in the previous section, and the location of each individual is obtained using its `spatialPosition()` method. We select just the x and y coordinates of the individual locations, since those are the dimensions used by the spatial map. We then call `spatialMapValue()` in vectorized fashion, to get the optima for all individuals in one call. The final code looks like this:

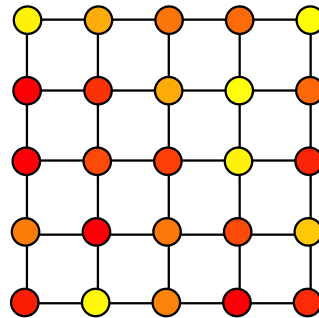
```
phenotype = inds.sumOfMutationsOfType(m2);
location = inds.spatialPosition[rep(c(T,T,F), inds.size())];
optimum = p1.spatialMapValue("map1", location);
inds.fitnessScaling = 1.0 + dnorm(phenotype, optimum, sigma_K);
inds.z = phenotype;
```

The rest of the recipe is as it was before. An example run looks like this:

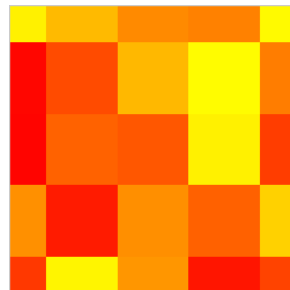


We have a nice randomly heterogeneous landscape, and the population has clearly adapted to it; we have reddish individuals in two of the more reddish areas, and orange individuals in three more orange areas. As before, the structure of neutral diversity visible in the chromosome view provides clear evidence of reproductive isolation and speciation.

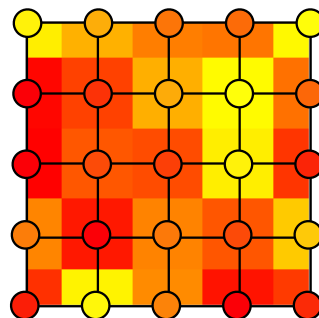
There is one thing that may be surprising, however. We supplied a 5x5 map to `defineSpatialMap()`, but the snapshot above shows what appears to be a continuous landscape. This is a result of the `interpolation=T` parameter we supplied to `defineSpatialMap()`, which instructed it to use interpolation (bilinear interpolation, in this case, to be precise) to make the landscape continuous. To understand better how that works, let's look in a little more detail at how SLiM builds landscape maps. First of all, here is the 5x5 grid of values that we supplied to `defineSpatialMap()`, colored according to the given color map:



Given this particular pixel grid, if we were to supply `interpolate=F` to `defineSpatialMap()` instead, the landscape displayed by SLiMgui would look like this:

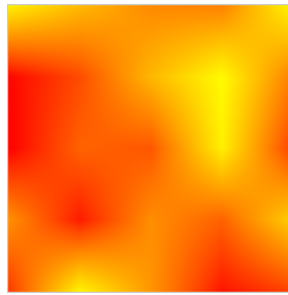


Note that because the pixel grid is aligned with the corners of the spatial bounds of the subpopulation, only  $\frac{1}{2}$  or  $\frac{1}{4}$  of the area of the outer pixels is contained within bounds. The reason for this is clear if we look at the pixel grid superimposed on that landscape:

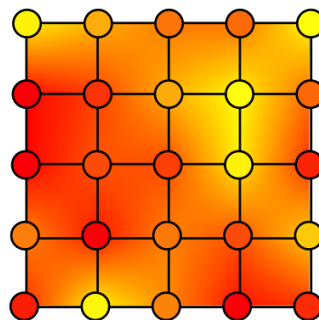


This is by design; it is the most natural way to handle such spatial maps when interpolation is involved (try thinking through the alternative to see why), and for consistency it is also how SLiM handles spatial maps when interpolation is not used.

So now when interpolation is turned on, the landscape looks like this:



This is the result of bilinear interpolation, which shades continuously between the defined points to produce a continuous map. The values defined by the pixel grid remain fixed, however. To illustrate that, here is the pixel grid superimposed on the interpolated landscape:

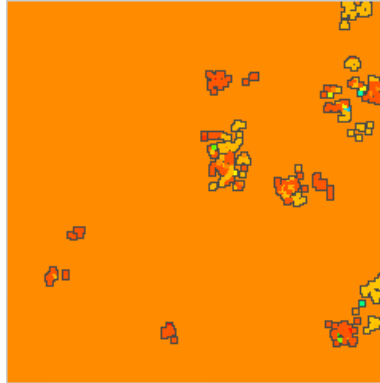


SLiM does not interpolate statically; it does not generate and store an interpolated map of some large but fixed size. Instead, when interpolation is enabled for a given spatial map, it calculates the exact interpolated value for a given point upon request. Interpolated maps are therefore completely continuous and effectively infinite-resolution (within the precision limits of floating-point numbers). With only 25 defined values, we therefore have an infinitely detailed landscape. In the previous recipe, using the world map, we did not enable interpolation, because we actually wanted the pixel grid; we wanted a world of binary pixels, either land or water, without allowing any shading between the two.

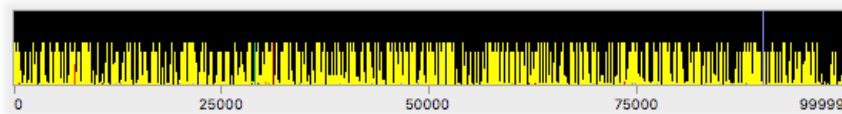
A logical step following the work of Haller, Mazzucco & Dieckmann (2013) would be to introduce temporal change in the landscape as well. We will not delve into that idea in any detail; but it is worth noting that that, too, is quite easy in SLiM. This is not part of the official recipe for this section – but try adding this code:

```
1: late()  
{  
  weight = (cos((sim.cycle - 1) / 1000.0) + 1.0) / 2.0;  
  newMap = weight * mapValues + (1 - weight) * 0.5;  
  p1.defineSpatialMap("map1", "xy", newMap, interpolate=T,  
    valueRange=c(0.0, 1.0), colors=c("red", "yellow"));  
}
```

This will cause the landscape to slowly cycle between heterogeneity and homogeneity. During the heterogeneous phases, speciation generally occurs as seen above. As the landscape fades into homogeneity, these species can persist, preserved by assortative mating and by the negative frequency-dependence of the competition function, such that speciation continues even when the landscape heterogeneity is completely gone:



Although the two species appear to have become sympatric in some areas, considerable reproductive isolation remains:



And when the heterogeneity returns to the landscape, the species can find their way back to areas where they are well-adapted – albeit with some evidence of genetic intermingling with the other species, such as some partially introgressed haplotypes.

In any case, here is the full model (without the temporal-change code discussed just above):

```
initialize() {
  defineConstant("sigma_C", 0.1);
  defineConstant("sigma_K", 0.5);
  defineConstant("sigma_M", 0.1);
  defineConstant("N", 500);

  initializeSLiMOptions(dimensionality="xyz");
  initializeMutationRate(1e-6);
  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  initializeMutationType("m2", 0.5, "n", 0.0, 1.0); // QTL
  m2.convertToSubstitution = F;

  initializeGenomicElementType("g1", c(m1, m2), c(1, 0.1));
  initializeGenomicElement(g1, 0, 1e5 - 1);
  initializeRecombinationRate(1e-8);

  initializeInteractionType(1, "xyz", reciprocal=T,
    maxDistance=sigma_C * 3); // competition
  i1.setInteractionFunction("n", 1.0, sigma_C);

  initializeInteractionType(2, "xyz", reciprocal=T,
    maxDistance=sigma_M * 3); // mate choice
  i2.setInteractionFunction("n", 1.0, sigma_M);
}
mutationEffect(m2) { return 1.0; }
1 late() {
  sim.addSubpop("p1", N);
  p1.setSpatialBounds(c(0.0, 0.0, 0.0, 1.0, 1.0, 1.0));
  p1.individuals.setSpatialPosition(p1.pointUniform(N));
  p1.individuals.z = 0.0;
```

```

    defineConstant("mapValues", matrix(runif(25, 0, 1), ncol=5));
    p1.defineSpatialMap("map1", "xy", mapValues, interpolate=T,
        valueRange=c(0.0, 1.0), colors=c("red", "yellow"));
}
modifyChild() {
    // set offspring position based on parental position
    do pos = c(parent1.spatialPosition[0:1] + rnorm(2, 0, 0.005), 0.0);
    while (!p1.pointInBounds(pos));
    child.setSpatialPosition(pos);

    return T;
}
1: late() {
    // construct phenotypes and fitness effects from QTLs
    inds = sim.subpopulations.individuals;
    phenotype = inds.sumOfMutationsOfType(m2);
    location = inds.spatialPosition[rep(c(T,T,F), inds.size())];
    optimum = p1.spatialMapValue("map1", location);
    inds.fitnessScaling = 1.0 + dnorm(phenotype, optimum, sigma_K);
    inds.z = phenotype;

    // color individuals according to phenotype
    inds.color = p1.spatialMapColor("map1", phenotype);

    // evaluate phenotypic competition
    i1.evaluate(p1);
    competition = sapply(inds, "sum(i1.strength(applyValue));");
    effects = 1.0 - competition / size(inds);
    inds.fitnessScaling = inds.fitnessScaling * effects;
}
2: first() {
    // evaluate mate choice in preparation for reproduction
    i2.evaluate(p1);
}
mateChoice() {
    // spatial mate choice
    return i2.strength(individual);
}
10000 late() {
    sim.simulationFinished();
}

```

This is the most complex recipe we will build involving spatiality, interactions, and landscape maps, but there is so much more that could be done. Interactions could be based upon genetics in ways beyond the QTL-based models we have explored; a spatial green-beard model would be interesting, for example. SLiM allows multiple landscape maps to be defined; it would be interesting to bring in empirical data on elevation, rainfall, mean temperature, and a host of other variables, and allow a population to evolve in a complex, multidimensional landscape to see whether realistic patterns of spatial biodiversity might be realized. One could even create a model in which the behavior of the organisms on the landscape modify the landscape itself – a model of desertification driven by overgrazing, for example.

The landscape in this recipe was created with `runif()` draws that provided spatially uncorrelated white noise. That is fine for some applications, particularly given the smoothing provided by interpolation, but real environments usually exhibit more complex patterns of spatial autocorrelation in environmental variables, from linear environmental gradients to patchy heterogeneity of a particular spatial scale. The SLiM-Extras repository on GitHub has a model,



landscape\_ac.slim, with a more sophisticated random landscape generation method that produces random landscapes that exhibit spatial autocorrelation; if you want a more realistic pattern of spatial variation, but do not have environmental data, that model might prove useful.

This recipe used the generated landscape map to determine the phenotypic optimum across the landscape, allowing local adaptation to occur. It is often desirable to instead use a landscape map to determine habitability. This makes sense when the environmental variable in question is something to which your simulated organisms cannot adapt; instead, some areas simply have a higher carrying capacity than others. This is quite simple; **in the context of a nonWF spatial model like that of section 16.10 or 16.11**, just implement density-dependent competition with a local scaling for the carrying capacity density taken from the habitability map, something like this:

```
K_local = K * p1.spatialMapValue("habitability", inds.spatialPosition);
inds.fitnessScaling = K_local / competition;
```

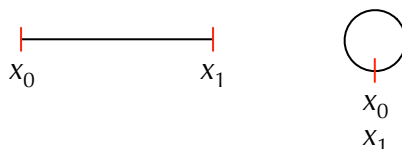
See those sections for discussion of how carrying capacity density works, and so forth. The WF model framework is not well-suited to variation in population density across the landscape, so a nonWF model really should be used, as discussed in that chapter.

Before we wrap up our discussion of spatial models, there is one topic that we deferred.

## 15.12 Periodic spatial boundaries

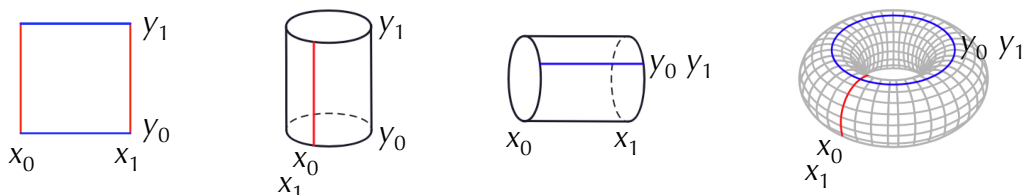
In section 15.3, various options for spatial boundary conditions were introduced: stopping, absorbing, reflecting, and reprising boundaries. The possibility of periodic spatial boundaries was also mentioned, but was deferred as an advanced topic. The time has come to explore this.

A periodic spatial boundary is one which wraps around: one edge of a spatial dimension is connected seamlessly to the opposite edge. A non-periodic one-dimensional bounded space is a line segment: points in this space fall between  $x_0$  and  $x_1$ . An individual might travel from  $x_0$  to  $x_1$ , at which point the end of the line segment is reached and motion must stop or reverse. The periodic version of this is a circle:  $x_0$  and  $x_1$  have been joined together to form a closed curve:



Here, an individual might travel from  $x_0$  to  $x_1$  and continue onwards; at the moment it reaches  $x_1$  it has also, simultaneously, returned to  $x_0$ , and may continue towards  $x_1$  again (and again, and again). Note that there is no “seam” and no “privileged point” in this space; the spatial topology at every point is identical, and an individual moving within the space cannot tell where it is located with respect to the point where  $x_0$  meets  $x_1$ .

A non-periodic two-dimensional bounded space is a rectangle with extents  $[x_0, x_1]$  and  $[y_0, y_1]$ . There are three periodic versions of this:  $x_0$  and  $x_1$  may be joined,  $y_0$  and  $y_1$  may be joined, or both may be joined. The first two options produce a topology like the surface of a cylinder without end caps; the third option produces not a sphere, but a torus, like the surface of a doughnut:



Again, movement along the periodic dimension(s) may continue indefinitely, wrapping around at the boundary; and again, there is no “seam” or “privileged point” in these spaces (along the periodic axis or axes), and an individual moving within the space cannot tell where it is located with respect to the points where the original coordinates meet.

Finally, a non-periodic three-dimensional bounded space is a cube; it may be made periodic in  $x$ , in  $y$ , in  $z$ , in  $x$  and  $y$ , in  $x$  and  $z$ , in  $y$  and  $z$ , or in  $x$  and  $y$  and  $z$ , yielding seven different periodic versions of three-dimensional space. The topology of these is harder to visualize than the one-dimensional and two-dimensional cases, but the principle is the same: movement along the periodic axis or axes may continue indefinitely because it wraps around, along the periodic axis or axes there is no “seam”, and an individual moving within the space cannot tell where it is located with respect to the points where the original coordinates meet.

This is all rather abstract, but periodic boundary conditions are very useful in modeling, especially when doing theoretical work rather than trying to simulate a real landscape. The reason is simple: periodic spatial boundaries eliminate edge effects, which are otherwise a source of bias in spatial models. In effect, periodic boundaries allow you to model an infinite space with no edges at all. Of course the space is not really infinite, but instead repeats periodically; but if the spatial scale of important interactions and dynamics in the model is small compared to the size of the periodic space, this is often an acceptable approximation.

Setting up periodic spatial boundaries in SLiM is a little more complex than implementing other boundary conditions because periodic spatial boundaries fundamentally change many aspects of SLiM’s spatial engine; enforcing the boundary condition is no longer just a question of modifying generated offspring positions in a particular way (although that is one part of it). For instance, consider two individuals that occupy positions (0.0, 0.1) and (0.0, 0.9) in a two-dimensional space with extent [0.0, 1.0] in both  $x$  and  $y$ . With any other boundary condition, the distance between these two individuals is 0.8 (0.9 – 0.1). With periodic boundaries, however, the distance between them is 0.2, because a line of length 0.2 may be drawn between them that wraps around the boundary in the  $y$  dimension. By definition, the distance between two points in a periodic space is always the *shortest* distance possible out of the infinitely many different distances that could be calculated. This means that distances, interaction strengths, and indeed all of the underlying mechanics of `InteractionType`’s spatial queries must take the spatial periodicity into account.

For this reason, periodic spatial boundaries must be declared up front, and cannot be changed subsequently. This declaration is done with a parameter to `initializeSLiMOptions()` named `periodicity`, which specifies the periodic spatial dimensions as a string. In this recipe, we will work with a two-dimensional model that is periodic in both dimensions – a toroidal model, as pictured above. This can be set up as follows:

```
initialize() {
    initializeSLiMOptions(dimensionality="xy", periodicity="xy");
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
    initializeInteractionType("i1", "xy", reciprocal=T, maxDistance=0.2);
    i1.setInteractionFunction("n", 1.0, 0.1);
}
```

The `initializeSLiMOptions()` call establishes a 2-D space (with `dimensionality="xy"`) and then declares both of those dimensions to be periodic (with `periodicity="xy"`). We also set up a spatial interaction here, involving both spatial dimensions; it uses a Gaussian interaction function with a standard deviation of 0.1, so it falls off at well under the spatial scale of the model as a

whole. We declare it to have a maximum distance of 0.2, for efficiency; the interaction strength will be very low further than two standard deviations out anyway.

Next, let's set up a subpopulation with random initial positions:

```
1 late() {  
  sim.addSubpop("p1", 2000);  
  p1.individuals.setSpatialPosition(p1.pointUniform(2000));  
}
```

Nothing surprising. Let's implement a modifyChild() callback to set up offspring positions:

```
modifyChild() {  
  pos = parent1.spatialPosition + rnorm(2, 0, 0.02);  
  child.setSpatialPosition(p1.pointPeriodic(pos));  
  return T;  
}
```

This uses a function named pointPeriodic() that is similar to the pointStopped() and pointReflected() functions we saw before; it translates the point it is passed so that it falls within the spatial boundaries, while implementing the periodic boundary conditions requested. In this case, pointPeriodic() wraps a point that lies beyond the periodic spatial boundaries, just as if the offspring had walked off of one edge of the space and re-appeared at the opposite edge.

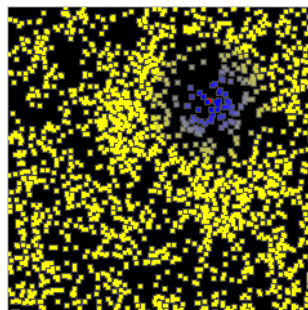
We need an event to define the end of the simulation, as usual:

```
1000 late() { sim.outputFixedMutations(); }
```

And now let's do something a bit more interesting: let's use the interaction type that we defined above to show, visually in SLiMgui, how interactions work in periodic space:

```
late()  
{  
  i1.evaluate(p1);  
  focus = sample(p1.individuals, 1);  
  s = i1.strength(focus);  
  inds = p1.individuals;  
  for (i in seqAlong(s))  
    inds[i].color = rgb2color(c(1.0 - s[i], 1.0 - s[i], s[i]));  
  focus.color = "red";  
}
```

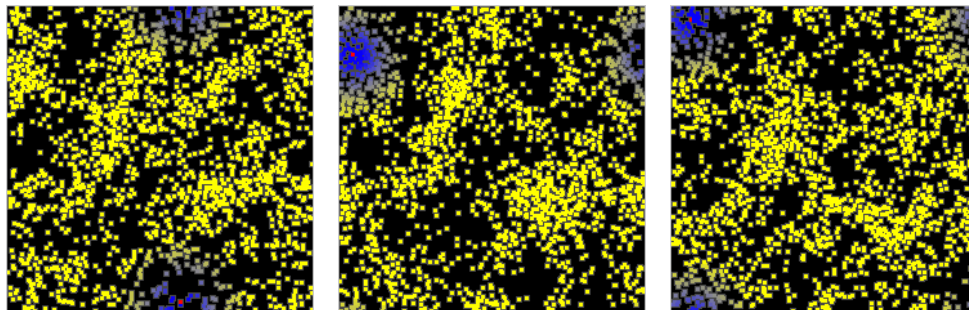
This event runs in every tick. It evaluates the interaction, then chooses a focal individual randomly. It asks the interaction type to calculate the interaction strength between the focal individual and all other individuals; then it loops over the individuals (by index) and sets each one's color in SLiMgui using a particular formula (see the Eidos manual for discussion of the rgb2color() function). Finally, it sets the color of the focal individual to red. The result:



The focal individual can be seen in red. The individuals closest to it are blue, while those farthest away are yellow. Because of the particular RGB (red, green, blue) color values passed to `rgb2color()`, the color of individuals at intermediate distances fades smoothly from blue to yellow. This is a nice illustration of the shape of the interaction kernel, and in fact this sort of coloration strategy can be quite useful in testing and visualizing spatial models.

There's an interesting digression to be had here. The interaction kernel we're using here is of type "n", meaning "normal" – more properly called "Gaussian", in this context. The interaction strength is given by the height of the interaction kernel, a 2D Gaussian density function. It's common knowledge that a normal distribution – a 1D Gaussian function – contains 68.2% of its area within  $1\sigma$  of the mean, 95.4% of its area within  $2\sigma$ , and 99.7% within  $3\sigma$ . Do those numbers then reflect the volumes contained within  $1\sigma$ ,  $2\sigma$ , and  $3\sigma$  of the center of our 2D Gaussian interaction kernel, which would correspond to the expected total interaction strength originating within those radii (assuming uniform spatial distribution of individuals)? The answer is no. The radius of a draw from a 2D Gaussian function – and therefore the volume under the surface of the 2D Gaussian interaction kernel – follows what is known as the Rayleigh distribution. According to this distribution, only 39.3% of the volume under the curve is within  $1\sigma$  of the center; 86.5% is within  $2\sigma$  and 98.9% is within  $3\sigma$ . So an interaction kernel with a maximum distance of only  $2\sigma$ , such as we used here, will capture only 86.5% of the total interactions around a focal individual (again, given a uniform spatial distribution of individuals). This might be acceptable for quick prototyping, but it is pretty rough – much rougher than a  $2\sigma$  width in the 1D case. Probably at least  $3\sigma$  is a good idea for all important work, unless you actually want a truncated kernel.

Coming back to earth after that digression, we can see that the snapshot above involved a focal individual that was far from any of the spatial boundaries. When a different focal individual is chosen, we can see that the spatial interaction wraps around the edges of the periodic space:



The third snapshot illustrates that wrapping occurs in both spatial dimensions, not just one at a time. Due to the toroidal geometry of the space, the four corners of the space as displayed in SLiMgui are actually all the same point! In the diagram above of the toroidal geometry of this space, the four corners as displayed in SLiMgui correspond to the intersection of the blue and red lines drawn on the torus. Note that topologically, there is nothing special about the blue and red lines in particular; every point on the torus lies at the intersection between a curve that circles around the major circumference of the torus (like the blue line) and a curve that circles around the minor circumference of the torus (like the red line). An individual living in this space cannot tell whether it is at the periodic boundary or not; as was emphasized earlier, there is no seam.

If you wish, you can play with this model by making only the x dimension periodic, or only the y dimension, to see how that affects the spatial interaction and what the resulting cylindrical topology feels like in practice. If you do so, note that `pointPeriodic()` will enforce only the periodic boundary condition; it will leave coordinates that are not periodic unmodified. To keep the modeled individuals inside bounds, some boundary condition – whether stopping, reflecting, absorbing, or reprising – must be enforced for the non-periodic coordinates. This can be done just

as it was in section 15.3; in particular, it is useful to note that a call to `pointPeriodic()` can be wrapped inside a call to `pointStopped()` or `pointReflected()` to achieve the desired effect. This works because `pointPeriodic()` has already brought the periodic coordinates into bounds, so they will not be modified by `pointStopped()` or `pointReflected()`. So a model that used periodic boundaries for only one of the two axes, and that enforced reflecting boundaries on the non-periodic axis, could use a `modifyChild()` callback like this:

```
modifyChild() {
  pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
  child.setSpatialPosition(p1.pointReflected(p1.pointPeriodic(pos)));
  return T;
}
```

All of the other elements of spatial modeling that were introduced in earlier recipes in this chapter – spatial competition, spatial mate choice, landscape maps – will work with periodic spatial boundaries. You can still model phenotype as a spatial dimension, too, as in section 15.7 for example; you will just (probably) not want that dimension to be periodic, since phenotypic traits are usually linear – being very short is not the same thing as being very tall!

The concept of periodic space can take some getting used to; cylindrical or toroidal space may seem rather artificial at first. But in fact, because of the absence of edge effects, periodic space is actually *less* artificial than other arbitrarily-chosen boundary conditions; it will not exhibit biases in the spatial density of individuals in one part of the space versus another, individuals everywhere will feel the same average interaction strength if they are randomly distributed, and motion in particular directions from particular locations will not be artificially impeded.

Because of the greater underlying complexity, models using periodic space will be a little slower than non-periodic models, but unless population size is large the difference should be slight. Unless you actually *want* a particular edge effect in your model, perhaps reflecting some real-world landscape's dynamics, periodic boundaries may be the best option.

### 15.13 Density-dependent fecundity with `summarizeIndividuals()`

In previous sections, we have looked at a variety of ways to use `InteractionType` to evaluate spatial interactions, modeling phenomena such as spatial resource competition and spatial mate choice. In this section we're going to look at another type of spatial interaction: local density-dependent fecundity. This could easily be implemented using `InteractionType`: calculate the interaction strength felt by a prospective parent and scale the probability of its reproduction accordingly. Instead of doing that, we will look at a completely new technique: using a function called `summarizeIndividuals()` to generate a spatial map of local density, and using that map to scale individual fecundity. The pros and cons of this approach will be discussed at the end.

We'll build this on top of a very straightforward model:

```
initialize() {
  initializeSLiMOptions(dimensionality="xy");
  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 late() {
  sim.addSubpop("p1", 1000);
  p1.individuals.setSpatialPosition(p1.pointUniform(1000));
}
```



```

modifyChild() {
  pos = parent1.spatialPosition + rnorm(2, 0, 0.01);
  pos = p1.pointReflected(pos);
  child.setSpatialPosition(pos);
  return T;
}
10000 late() { sim.outputFixedMutations(); }

```

Nothing here is new so far; we make a subpopulation of 500 individuals in a 2D space, and have offspring land near their parents. For simplicity we do not include spatial competition or mate choice here, so this is almost a return to the recipe of section 15.1.

Next, let's introduce the `summarizeIndividuals()` function. Our ultimate goal is to use it to assess local density, but let's start with a simpler problem: assessing local presence/absence. We will do this in a new `late()` event that will run in every tick:

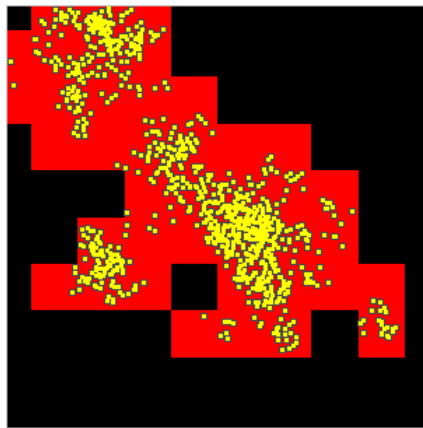
```

late() {
  inds = p1.individuals;
  bounds = p1.spatialBounds;

  // make a presence/absence map: 1 is present, 0 is absent
  presence = summarizeIndividuals(inds, c(10, 10), bounds,
    operation="1;", empty=0.0, perUnitArea=F);
  p1.defineSpatialMap("presence", "xy", presence, F,
    range(presence), c("black", "orange", "red"));
}

```

Before even explaining this code, let's look at the result in SLiMgui:



The background colors show the spatial map named "presence". Red squares indicate cells in the 10×10 spatial map where individuals are present; black squares indicate absence. Notice that the cells on the edges and corners are smaller, because part of their extent lies outside the spatial bounds; this is the standard behavior of spatial maps, as discussed in section 15.11 in detail.

So how does this work? The operation of the `summarizeIndividuals()` function is somewhat complex (see section 24.17.3 for complete documentation), but essentially it builds the presence/absence map for us by *binning* the individuals in the model (passed to it as `inds`) into cells in a 10×10 grid (as requested by `c(10, 10)`) that spans the spatial bounds of the subpopulation (passed to it as `bounds`), and then calculating a value for each grid cell based upon the contents (the individuals) in each corresponding bin.

That per-cell calculation is highly configurable, which is where much of the complication of this function comes in. The `operation` parameter is an Eidos *lambda*, a singleton string value that



contains executable code; we have seen lambdas before in contexts such as `sapply()` and `LogFile` (see also section 20.5). In this case, `operation` is `"1;"`, perhaps the simplest possible lambda; it simply evaluates to 1. So with this lambda, each grid cell evaluates to 1: presence! What if no individuals are present in a given grid cell? In that case, `summarizeIndividuals()` actually takes the value to be used as a separate parameter, `empty`. Here `empty` is `0.0`, so grid cells that contain no individuals receive a value of 0: absence! The last parameter, `perUnitArea`, provides a final processing option that we will explore in a moment.

Perhaps the power of `summarizeIndividuals()` is already somewhat apparent; since we can supply any lambda we wish for `operation`, we can do a great many things. We could assess the mean age of individuals across space, for example, by passing `"mean(individuals.age);" for operation` (the focal individuals being evaluated are passed to the lambda in the pseudo-parameter `individuals`); or if individual phenotypic values for a quantitative trait are stored in the `tagF` property (see chapter 13) we could get the minimum phenotypic value in grid cells across space with `"min(individuals.tagF);" . The operation lambda is not limited to simple property accesses, either; any Eidos code is allowed, including calls to methods and user-defined functions.`

Here, however, we have a simpler goal: a map of local density. To produce that, change the above `late()` event to the following code:

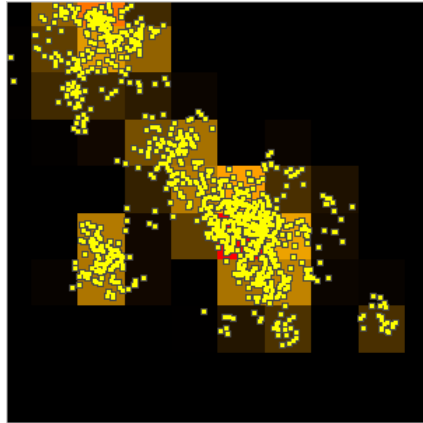
```
late() {
  inds = p1.individuals;
  bounds = p1.spatialBounds;

  // make a density map: 0 is empty, 1 is maximum density
  density = summarizeIndividuals(inds, c(10, 10), bounds,
    operation="individuals.size();", empty=0.0, perUnitArea=T);
  density = density / max(density);
  p1.defineSpatialMap("density", "xy", density, F,
    range(density), c("black", "orange", "red"));
}
```

A couple of things have changed. We now use an operation of `"individuals.size();" , so we are getting the size of the individuals vector in each grid cell to produce a count. As noted above, the grid cells in spatial maps are not all the same size; since we want to assess local density, which is in units of “individuals per unit area”, we now pass T for perUnitArea, and can now see what it does: it divides the result of operation for a given grid cell by the spatial area of that grid cell (in the units of the spatial bounds). This means that the count in edge cells will be multiplied by two (because half their area is outside bounds), and the count in corner cells will similarly be multiplied by four. This perUnitArea=T option also rescales the counts downward by a constant factor because the unit area of one grid cell is less than one, but that doesn’t matter here because our next step is to normalize the resulting matrix, named density, to a range of [0, 1]. This normalization is not needed to define the spatial map, but doing this now will make our density-dependence code later on more efficient. Finally, we use a three-color scale in SLiMgui to improve the display.`

An aside: here we normalized to `[0, 1]`, but of course you can do whatever calculations you wish with the result from `summarizeIndividuals()`, before making a spatial map from it; you might rescale the values, transform them with `log()` or `sqrt()`, combine them with values from another call to `summarizeIndividuals()`, or whatever else is desired. It’s also worth noting here that the vector of individuals passed to `summarizeIndividuals()` can be any set of individuals; it need not correspond to a whole subpopulation. The spatial bounds can also be whatever area you wish; it can be a sub-area within the simulated landscape, and then only individuals inside that sub-area will be incorporated into the summary.

With the above changes to the `summarizeIndividuals()` call, here's the new appearance in SLiM (using the same seed value, so it is directly comparable to the previous screenshot):

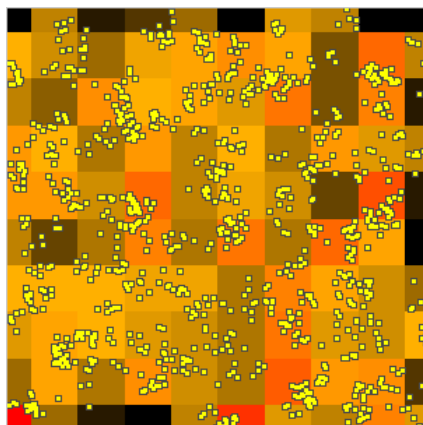


The highest density is in the red square near the center; shades of orange to black indicate lower densities. We have successfully shifted from local presence/absence to local density.

Finally, we want to actually use this spatial map to influence local fecundity. We will do this by adding a bit of code to the `modifyChild()` callback. Here's the final version of that callback:

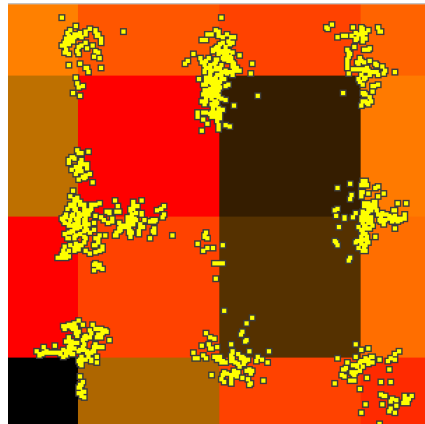
```
modifyChild() {  
    pos = parent1.spatialPosition + rnorm(2, 0, 0.01);  
    pos = p1.pointReflected(pos);  
  
    if (runif(1) < p1.spatialMapValue("density", pos))  
        return F;  
  
    child.setSpatialPosition(pos);  
    return T;  
}
```

Inserted into the middle is a little new logic that gets the map value for the candidate child position (rather than the parent position, as it happens; perhaps this models juvenile mortality at high densities). Since we normalized the map values to `[0, 1]` above, we can now conveniently use this value as a probability threshold; the callback returns `F`, suppressing the generation of this proposed child, if a random uniform draw is less than the normalized density value. This achieves the desired density-dependent fecundity, which smooths out the spatial distribution of the individuals tremendously, as seen in this screenshot:



Offspring are still produced close to their parent, which normally produces a clumped spatial distribution as we saw before; now, however, offspring are preferentially generated into grid squares that had a low density in the parental generation, driving the spatial distribution outward into unoccupied areas.

The quantization of the density information into discrete grid squares is not without disadvantages. It's not so easy to see with the 10×10 grid above, but the individuals are actually playing little bet-hedging games with their spatial locations; they are not randomly distributed. This is more clear with a lower-resolution grid. Here's the same model with the `dim` parameter changed from `c(10, 10)` to `c(4, 4)`:



Now the problem is quite apparent; the individuals are clustering around the vertices between grid cells! The payoff of this strategy is simple: if you're in such a location and any of the four grid cells you're near happens to become low-density, you've got a one-in-four chance that an offspring you generate will land in that good habitat. If, on the other hand, you're far from such a vertex, your offspring will all land in the same grid cell as you; all your eggs are literally in one basket. If that basket – that grid cell – happens to become high-density, you and your whole lineage may die out. Worse, you and your offspring are competing against each other; if your lineage is successful and grows, then that means that the density of your grid cell is increasing – so your growth is self-limiting and ultimately doomed. If you were near a vertex, you and your offspring could occupy four different grid cells, hedging against extinction and limiting competition.

This is not necessarily a fatal flaw. For one thing, it's “biologically realistic”, in the sense that it is an emergent strategic response to the incentives presented by the model, and real-world organisms might well respond to the same incentives in the same way. (Here's a fun example of such patterns in nature, although it isn't clear exactly what “incentives” are being responded to in this case: <https://www.nature.com/articles/s42003-020-01431-0>). Of course these patterns might still be undesirable; you might not really want to be setting up these particular incentives for your digital organisms. Happily, the extent of this problem also depends upon parameterization; it is more visible at 4×4 than at 10×10 because the scale of dispersal relative to the scale of the spatial grid is smaller at 4×4, making it easier for the organisms to stay close to the favored positions, and making the benefit of the favored positions larger. At 10×10 the effect is weaker, because dispersal is larger, relative to the grid size, and so (a) it is more difficult for an individual and its offspring to stay close to the favored locations, and (b) the favored locations are less favored in the first place, because even an individual at the center of a grid cell might generate an offspring into an adjacent cell. At 20×20 the artifact seems to be virtually gone; a statistical test might still reveal some degree of bias, but depending upon what you're trying to achieve that might be acceptable (after all, other spatial modeling techniques can also produce artifacts, clustering, etc., as we have seen in previous sections). There is a limit to this, though; at 100×100 the grid cells are so small that

almost all of them are empty, and so they no longer really serve to prevent the spatial clustering that was observed in the model before the density map was implemented. The upshot, then, is that it is important to really understand what you are modeling (working in a graphical modeling environment helps a lot with this!), to look carefully for artifacts and biases introduced by your modeling choices, and to choose arbitrary parameters (like the resolution of the density map here) carefully, with an eye to how they might influence model dynamics.

This might seem like an abrupt change of subject, but it really isn't: As mentioned at the outset, this recipe could have been implemented using `InteractionType` instead. Why might we choose `summarizeIndividuals()` instead? What are the pros and cons of this approach?

An advantage of `InteractionType` is that it is *exact*. Evaluating an interaction goes out and finds all of the neighbors that exist within the maximum interaction distance, calculates their strength of interaction with the focal individual or point, and returns the precise interaction strength felt. If desired, the interaction strength exerted by nearby individuals can depend upon their exact distance to the receiver, as expressed by the interaction function. Parallels can easily be drawn between this approach and analytical formulas of spatial interactions. The results from `summarizeIndividuals()`, on the other hand, are in some sense *approximate*. Individuals get binned together based on their spatial position, and the gathered value for a grid cell is representative of a sort of average across that whole cell; precise interaction strengths based upon distances between pairs of interacting individuals are never calculated. Every individual inside a given grid cell will see the same "interaction strength" (here, density), while an individual just barely across the border into an adjacent grid cell might see a completely different value. This can lead to artifacts, as we just saw, if it is not handled carefully.

As is often the case, however, strengths are also weaknesses, and weaknesses are also strengths. Because of the exact calculations it does for every focal individual or point, `InteractionType` is relatively slow. With a large number of individuals, and a relatively large maximum interaction distance, the overhead of its calculations can be very large, scaling *quadratically* with the number of individuals in the worst case – every individual interacting with every other individual. The `summarizeIndividuals()` approach provides a much more efficient alternative. Since it simply bins individuals spatially (a kind of radix sort, really) and then evaluates the value contributed by each individual independent of the other individuals, it scales *linearly* with the number of individuals – in computer science parlance, it is  $O(N)$  instead of  $O(N^2)$ . For large  $N$  (the number of individuals), the difference can be the difference between a simulation that runs in hours and a simulation that would not finish in months.

So the use of `summarizeIndividuals()` can be an optimization technique, then. Its approximate nature can be somewhat unfortunate; that two nearby parents might feel very different densities because they fall into different grid squares is clearly not optimal. How much of a problem this is will depend upon circumstances. For large population sizes, these quantization biases might (or might not) become fairly unimportant; they might tend to "average out" as  $N$  approaches infinity. They might be managed by *convolving* the resulting matrix – blurring it, essentially, so that some non-locality of density gets spread between adjacent cells of the spatial map. That is not hard to do, although it is beyond the scope of this section. You could also consider taking more than one sample from the spatial map, at a cluster of points surrounding an individual, and averaging the results. For sufficiently large  $N$ , such techniques should still be much faster than using `InteractionType`, but there's nothing like benchmarking.

The other reason to use `summarizeIndividuals()` is that you actually *want* a quantized spatial map of some metric. These can be particularly useful visualization in SLiMgui and for output as image files. For the former, note that you can define a spatial map in your model and never actually use it at all; you can define it purely as a visualization for SLiMgui. For the latter, note that you can construct an `Image` object directly from a matrix returned by `summarizeIndividuals()`,

and then save that `Image` to disk as a PNG with the `write()` method. This works even when you are not running under SLiMgui.

This section's recipe was a WF model, since we have not yet introduced nonWF models; it seemed worthwhile to introduce `summarizeIndividuals()` in this simpler context. As chapter 16 will show in detail, however, this sort of spatial model is probably better as a nonWF model. In this recipe, the population size is fixed and only the *relative* probability of reproduction varies across the landscape with density; the overall population size is set as a top-down parameter, and so although density-dependent fecundity is present, it is not regulating the population size, which is a bit unnatural biologically. In a nonWF model, this density-dependent fecundity could actually regulate the population size, such that making the effect of density on fecundity more strongly negative would result in a smaller equilibrium population size, while making it weaker would lead to a larger equilibrium size. This design would also mean that if the spatial distribution was highly clumped at a given point in time, fewer offspring would be produced in the next generation because of the high local density near the parental individuals; the WF model will generate the same number of offspring regardless of the spatial distribution of the parents, which is again biologically unnatural. The next chapter will go into these concepts in depth; but `summarizeIndividuals()` can be used in nonWF models just as easily as it was used here.

## 16. Going beyond Wright–Fisher models: nonWF model recipes

Beginning with SLiM 3.0, two fairly different types of models are supported in SLiM: Wright–Fisher or WF models, and non-Wright–Fisher or nonWF models. See section 1.6 for a discussion of the differences between these two types of models. All of the recipes presented so far have been WF models, since that is the default model type in SLiM. In this chapter, we will go beyond the assumptions and constraints of Wright–Fisher models, and will examine recipes for a variety of nonWF models.

A great deal of the overall design of SLiM is shared between WF and nonWF models. All of the Eidos classes that embody SLiM are the same (Community, Species, Subpopulation, Individual, etc.), and the way that SLiM models the chromosome, genomes, mutations, and so forth is unchanged. Since that foundation is all shared, a good understanding of the concepts in the preceding chapters will be assumed. Many of the techniques presented in the preceding recipes will also work in nonWF models. We will not re-cast all of those techniques in a nonWF context, since that would mostly just be repetitive and uninteresting; instead, we will focus on the important ways in which nonWF models are different from WF models.

As discussed in more detail in section 1.6, the main differences between WF and nonWF models fall into a few major categories. First of all, in nonWF models generations may be overlapping and individuals can live for more than one tick; for this reason, in nonWF models the model’s script is responsible for creating new offspring as needed, rather than that happening automatically every tick as it does in WF models. Second, for the same reason, in nonWF models the parental generation does not die off automatically after offspring are generated; instead, fitness governs mortality (rather than governing mating success as in WF models). Third – as a consequence of the previous two differences, really – in nonWF models population regulation is a consequence of the balance between individual reproduction and individual mortality, just as it is in natural populations, rather than being enforced through a set population size as in WF models, and so the model typically needs to treat population regulation explicitly, with some individual-level effect such as density-dependence. Fourth, migration in nonWF models is similarly managed on an individual basis in the model’s script, rather than being done automatically by the SLiM engine based upon set migration rates.

All of this points to two basic observations. One observation is that the tick cycle is quite different between WF and nonWF models. Chapter 22 discusses the tick cycle for WF models, whereas chapter 23 discusses the tick cycle for nonWF models and the important conceptual ways in which it differs from the WF tick cycle. An understanding of those differences, such as the way in which the semantics of `early()` and `late()` events have changed and the way that the meaning of fitness has shifted, will be important to understand the recipes that follow, so chapter 23 should be consulted for further information as needed. The other observation is that nonWF models are generally more individual-based and more complex than WF models, because more responsibilities like offspring generation and migration have been pushed from SLiM onto the model’s script. With this additional complexity comes considerable additional power and flexibility, however, as we will see.

### 16.1 A minimal nonWF model

Let’s begin with a minimal nonWF model, similar to the minimal WF model presented in section 4.1. This will illustrate several of the fundamentals of nonWF models: how to switch SLiM into nonWF “mode”, how to implement individual-based reproduction and density-dependent population regulation, and how to work with overlapping generations.

With no further ado, here is the recipe:



```

initialize() {
    initializeSLiMModelType("nonWF");
    defineConstant("K", 500);

    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;

    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);
}
reproduction() {
    subpop.addCrossed(individual, subpop.sampleIndividuals(1));
}
1 early() {
    sim.addSubpop("p1", 10);
}
early() {
    p1.fitnessScaling = K / p1.individualCount;
}
late() {
    inds = p1.individuals;
    catn(sim.cycle + ": " + size(inds) + " (" + max(inds.age) + ")");
}
2000 late() {
    sim.outputFull(ages=T);
}

```

The first line of the `initialize()` callback is a call to `initializeSLiMModelType("nonWF")`; this tells SLiM that we are building a nonWF model. That has various consequences; it activates the nonWF tick cycle shown in chapter 23, for example, and it enables some properties and methods on SLiM's objects while disabling others. It is possible, when writing a WF model, to include a call to `initializeSLiMModelType("WF")`, but unnecessary, since that is the default.

The next line sets up a defined constant, `K`. This will be the carrying capacity for the model's population regulation; we'll cover that below.

The rest of the `initialize()` callback is much as we have seen before, but you might notice one surprising thing: we set the `convertToSubstitution` property of mutation type `m1` to `T`. In WF models, mutations convert to substitutions automatically; the default value of that property is `T`, so it would be redundant to set it to `T` again. In nonWF models, however, mutations do not convert to substitutions automatically; the `convertToSubstitution` property is `F` by default and must be set to `T` when desired. The reason is that in nonWF models fitness is absolute, not relative, and so only completely neutral mutations with no side effects are safe to convert to `Substitution` objects (see section 23.4 for further discussion). Since `m1` mutations are completely neutral in this model, we tell SLiM to allow them to fix; this will make the model run much faster.

The next script block is a new type of callback, a `reproduction()` callback, that may be used only in nonWF models. Such `reproduction()` callbacks are called once per individual, at the beginning of each cycle; this provides an opportunity for that focal individual to generate offspring (which it might or might not do); see section 23.1 for further discussion. This callback calls `subpop.sampleIndividuals(1)` to draw one random individual from `subpop` (the subpopulation of the focal individual) as a mate, and then calls `subpop.addCrossed()` to add a new offspring individual that is the result of crossing – biparental sexual reproduction – between the focal individual for the callback (`individual`) and the chosen mate. The new individual is created, and is in fact returned to the caller, but is not actually added to the subpopulation until offspring

generation is finished. Note that although each individual will generate exactly one offspring of its own here, as the focal individual or “first parent”, it might also be chosen as a mate by another individual (perhaps more than once, or perhaps not at all).

The next script block returns us to more familiar territory, since it simply calls `addSubpop()` to create subpopulation `p1` with ten initial individuals. The only point to be made here is that while in a WF model this would set the subpopulation’s size forever (until `setSubpopulationSize()` was called, at least), in a nonWF model this only sets the initial subpopulation size; the size of the subpopulation will henceforth be governed by birth and death events, not by its initial size.

Speaking of death, the next `early()` event governs that. It calculates an absolute fitness value,  $K / p1.individualCount$ , and sets that into the `fitnessScaling` property of `p1`. We have seen the `fitnessScaling` property a little before, chiefly in chapters 13 and 15 as a property on `Individual`. It also exists as a property on `Subpopulation`, where it scales the absolute fitness of the whole subpopulation because the calculated fitness for every individual in the subpopulation is multiplied by the subpopulation’s `fitnessScaling` value.

This `fitnessScaling` property of `Subpopulation` may be used in WF models too, in fact, but is not generally useful in that context; since WF models use relative fitness, scaling the fitness of all individuals by the same constant has no effect. In nonWF models, however, it has a very important effect! This `fitnessScaling` factor is what regulates the population size in the model; if this line were commented out, the population would grow exponentially forever (until SLiM crashed, or got so slow as to be effectively halted). Instead, when the subpopulation size is less than  $K$ , `fitnessScaling` will be greater than  $1.0$  (and so no mortality will occur and the subpopulation size will grow); but if it is larger than  $K$ , `fitnessScaling` will be less than  $1.0$  and mortality will bring it back toward  $K$ . Since new individuals get generated at the beginning of each cycle by the `reproduction()` callback, once the model reaches equilibrium the population size will double during offspring generation to around  $1000$ , then a `fitnessScaling` value of approximately  $0.5$  will be set, and then approximately half of the individuals will die during the viability/survival tick cycle stage, bringing the population size down to approximately  $500$  (i.e.,  $K$ ).

Note that there is nothing magical about this particular formula; any formula or model design that influences individual fitness in such a manner as to regulate the population size will work (see section 16.20). This particular formula produces exponential growth until  $K$  is reached (assuming an otherwise neutral model), and then stochastic population size fluctuation around  $K$  thenceforth. The population size will be stochastic around  $K$  because in nonWF models fitness is the probability of death, but of course sometimes more individuals will die than expected, sometimes less; the fate of each individual is in the hands of SLiM’s random number generator.

A *very important* point to note, however, is that at low population density this equation will produce very high absolute fitness; when at 1/100th of carrying capacity, every individual’s absolute fitness will be multiplied by 100 to reflect the benefits of low density – decreased competition, more resources, etc. This is biologically unrealistic for almost all species; almost always, there would be some maximum benefit that an individual could gain from being at low density, and so you would want to use a more sophisticated density-dependence equation that would cap the fitness scaling at that maximum value. (Indeed, in many species Allee effects would exist, such that below some threshold density individual fitness would actually *decrease* due to other biological considerations – difficulty of finding a mate, lack of benefits from flocking/schooling, lack of cooperative interactions, etc.) In this model the lack of a cap to the benefits of low density doesn’t matter; absolute fitness values above  $1.0$  are all equivalent anyway here, since survival probability cannot be greater than 100%. In a model that includes deleterious mutations or other fitness-decreasing effects, however, this density-dependence equation’s oversimplification of biological reality *will* matter, masking or completely eliminating those other fitness effects when at low density and thus making it virtually impossible for extinction to occur. In short: this is a toy

model, and in a more realistic model you would want to think carefully about how you enforce density-dependence to ensure that the fitness effects of density are appropriate at all population sizes. We will often use this same toy-model density-dependence equation in this manual, since it is simple and general, but that does not mean it is sufficiently realistic for real-world modeling. Section 16.4 discusses this problem from another angle: the problem of beneficial mutations having no effect in individuals whose absolute fitness is already 1.0 or greater.

Next we have a `late()` event that outputs two pieces of information in each tick: the population size, and the age of the oldest individual. Here's the initial output from one run:

```
1: 10 (0)
2: 20 (1)
3: 40 (2)
4: 80 (3)
5: 160 (4)
6: 320 (5)
7: 496 (6)
8: 485 (7)
9: 500 (6)
10: 522 (7)
...
```

You can see the exponential growth at the start settling in around 500 once the model reaches carrying capacity. You can also see that we already have overlapping generations in this model; the individuals at the end of tick 1 are of age 0 (newly generated juveniles), and that initial cohort ages without mortality during exponential growth (since we have not implemented a maximum age or any sort of age-dependent reduction in fitness). By tick 7, though, the carrying capacity has filled up and individuals start to die. Since every individual in this model has the same fitness, mortality is purely random, and sometimes you will get a Methuselah that lives to be 20 or even older; but most individuals will die through sheer bad luck before then, and the maximum age in this model will tend to fluctuate around 10 or 15. Later recipes will explore how to control the population age in biologically realistic ways, but for now let's just bask in the glory of the fact that we have already modeled something that can't be modeled in WF SLiM: overlapping generations.

The final event produces output from the model with `outputFull()`, as we have seen many times before. The only new element is the `ages=T` parameter, which requests that age information be added to the output (the details of the output format are given in section 26.1.1).

That's it; that completes our first nonWF recipe! In subsequent sections we will explore the greater power the nonWF paradigm affords us, because we can now control the mating, fecundity, migration, fitness, and survival of each individual.

## 16.2 Age structure (a life table model)

In the previous recipe, the probability of survival was the same regardless of age, and that produced a particular emergent age structure in the population. In most biological systems, however, the probability of survival is age-dependent. Commonly, this is modeled with a life table that gives the probability that an individual of a given age will die within the next time period (the next year, often).

To model non-overlapping generations in a nonWF model, one might use a life table that gives a probability of survival of 0.0 for all individuals of age 1 or older (newly generated juveniles having an age of 0); with such a life table, offspring would be generated and then the parental individuals (all having an age of 1) would all immediately die (see section 16.15). In this model we will implement a slightly more complex life table, for an imaginary species that has high

juvenile mortality, low adult mortality, and a maximum age of seven ticks. We will also implement age-dependent fertility and density-dependent population regulation.

Let's look at this model one piece at a time, beginning with initialization:

```
initialize() {
  initializeSLIMModelType("nonWF");
  defineConstant("K", 30);
  defineConstant("L", c(0.7, 0.0, 0.0, 0.0, 0.25, 0.5, 0.75, 1.0));

  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;

  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
```

This is identical to the previous recipe except for the addition of the defined constant L, which is our life table. It gives the probability of mortality for each age; newly generated juveniles have a mortality of 0.7 (i.e., 70%), then the mortality drops to zero for several years, and then it ramps gradually upward with increasing age until it reaches 1.0 for age 7; all individuals of age 7 will die. Note that this is only the *age-related* mortality; density-dependence will also cause mortality, as we will see below, but that will be additional to this age-related mortality, which would occur even in a population that was not limited by its density.

Next, let's implement reproduction:

```
reproduction() {
  if (individual.age > 2) {
    mate = subpop.sampleIndividuals(1, minAge=3);
    subpop.addCrossed(individual, mate);
  }
}
```

This is essentially the same as the reproduction() callback in the previous recipe, except that here we have prevented reproduction by individuals of age 2 or below (with the same limitation on the chosen mate's age). One could similarly restrict reproduction in a nonWF model based upon genetics, individual state, resource acquisition, mate compatibility, or any other factor.

Population initialization looks like this:

```
1 early() {
  sim.addSubpop("p1", 10);
  p1.individuals.age = rdunif(10, min=0, max=7);
}
```

We again start at a population size of 10 and allow the population to grow upward to the carrying capacity (only 30 in this model, to make reading the output of the model easier). Since addSubpop() sets the age of all new individuals to 0, that would provide our model with a bit of an artificial start, and it might also present difficulties since juvenile mortality in this model is so high – sometimes the population might go extinct before it reaches reproductive age. We therefore draw random ages from a discrete uniform distribution from 0 to 7. If one had empirical data about the age distribution in one's system, that might of course be an even better starting point.

And here we manage both age-related and density-dependent mortality:

```

early() {
  // life table based individual mortality
  inds = p1.individuals;
  ages = inds.age;
  mortality = L[ages];
  survival = 1 - mortality;
  inds.fitnessScaling = survival;

  // density-dependence, factoring in individual mortality
  p1.fitnessScaling = K / (p1.individualCount * mean(survival));
}

```

We calculate the age-related mortality by getting all of the individuals in `p1`, getting their ages, and then looking up those ages in `L` to get a vector of the mortality rates for the individuals. Survival rates are the opposite of mortality rates, so we subtract from 1; if an individual has a mortality rate of 1 it has a survival rate of 0, and *vice versa*. Finally, we set those survival rates into the `fitnessScaling` properties of the individuals. We saw the `fitnessScaling` property of `Subpopulation` in the previous recipe, where it scaled the fitness of all individuals in the subpopulation by the same constant factor. The `fitnessScaling` property of `Individual` has much the same effect, but on an individual basis; each individual can have a different `fitnessScaling` value, which is multiplied into that individual's calculated fitness. This is equivalent to implementing a `fitnessEffect()` callback (see section 25.3) that returns a survival-based fitness effect for the focal individual based upon that individual's age; but this way is much faster since it is done in a vectorized fashion without `fitnessEffect()` callbacks.

Next, the callback above calculates density-dependent mortality based upon `K` and the current population size, as before, but also factors in the mean survival rate in the population due to age-related mortality. Without that correction, the population would equilibrate around a lower size than `K`, because age-related mortality would occur *in addition to* the density-dependent mortality necessary to bring the population down to `K`. With the correction, the population size should fluctuate stochastically around `K`, as desired. One could of course get fancier, and come up with equations that made the probability of density-dependent mortality depend upon age (or any other individual state) in some manner; perhaps older individuals would be weaker and more vulnerable to diseases and parasites that are common when population density is high, for example.

```

late() {
  // print our age distribution after mortality
  catn(sim.cycle + ": " + paste(sort(p1.individuals.age)));
}
2000 late() { sim.outputFixedMutations(); }

```

These are the last components of our model: output and termination. The `late()` output event prints the population's age distribution in each tick; this is post-mortality, since `late()` events run after the viability/survival tick cycle stage in nonWF models. A typical run:

```

1: 0 0 0 1 2 4 5 6 6
2: 0 0 0 0 0 1 1 1 2 3 5 6
3: 0 0 0 1 1 1 1 1 2 2 2 3 4
4: 0 0 0 1 1 1 2 2 2 2 2 3 3 3 4 5
5: 0 0 0 0 0 0 1 1 1 2 2 2 3 3 3 3 3 4 4 4 5
...
1996: 0 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 3 3 4 4 4 5 5 5 5
1997: 0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 4 5 6 6
1998: 0 0 0 0 1 1 1 1 2 2 2 2 2 3 3 3 3 4 4 4 4 5 5 6
1999: 0 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 3 4 4 4 4 5 5 5 6
2000: 0 0 0 0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6

```

The beginning of the output shows growth toward the carrying capacity; even before the carrying capacity is reached some age-related mortality occurs. The end of the output shows the (somewhat stochastic) equilibrium population size and age structure. With this life table, the population is dominated by middle-aged individuals; most juveniles die, and few individuals make it to age 7 since the mortality rate ramps upward beginning at age 4. Note that no individuals of age 7 are visible here because this output is post-mortality; no individual of age 7 should ever exist at this point in the tick cycle. If this output event were an `early()` event instead, however, we would expect to see the occasional age 7 individual.

This model uses a life table, but the larger point is that nonWF models allow one to model any individual-based mortality effects, whether due to genetics (which would typically occur through SLiM's built-in fitness calculations), age (as with a life table or similar scheme), density (as also modeled here), individual state, environmental effects, or anything else. Fitness effects influencing survival can be expressed through `fitnessEffect()` callbacks, or using the `fitnessScaling` properties of `Subpopulation` and `Individual` that we used here.

### 16.3 Monogamous mating and variation in litter size

In the previous recipe we explored how to manipulate individual fitness values in order to implement age-dependent mortality using a life table. In this recipe we will look at the other end of the circle of life: mate choice and fertility. The previous nonWF models we have built have used an extremely simplistic model of reproduction in which each individual of reproductive age produces exactly one offspring per tick itself with a randomly selected mate (and may be chosen as a mate by another individual, too). Here we will implement a very different model of reproduction: monogamy (within a single breeding season), and generation of a litter of offspring of non-deterministic size.

This model will look very similar to section 16.1's recipe, so let's just see the whole model all at once:

```
initialize() {
  initializeSLiMModelType("nonWF");
  defineConstant("K", 500);

  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;

  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
reproduction() {
  // randomize the order of p1.individuals
  parents = sample(p1.individuals, p1.individualCount);

  // draw monogamous pairs and generate litters
  for (i in seq(0, p1.individualCount - 2, by=2))
  {
    parent1 = parents[i];
    parent2 = parents[i + 1];
    litterSize = rpois(1, 2.7);

    for (j in seqLen(litterSize))
      p1.addCrossed(parent1, parent2);
  }
}
```



```

        // disable this callback for this cycle
        self.active = 0;
    }
    1 early() {
        sim.addSubpop("p1", 10);
    }
    early() {
        p1.fitnessScaling = K / p1.individualCount;
    }
    late() {
        inds = p1.individuals;
        catn(sim.cycle + ": " + size(inds) + " (" + max(inds.age) + ")");
    }
    2000 late() {
        sim.outputFull(ages=T);
    }
}

```

This is, in fact, identical to section 16.1 except for the `reproduction()` callback, so let's look at that in detail:

```

reproduction() {
    // randomize the order of p1.individuals
    parents = sample(p1.individuals, p1.individualCount);

    // draw monogamous pairs and generate litters
    for (i in seq(0, p1.individualCount - 2, by=2))
    {
        parent1 = parents[i];
        parent2 = parents[i + 1];
        litterSize = rpois(1, 2.7);

        for (j in seqLen(litterSize))
            p1.addCrossed(parent1, parent2);
    }

    // disable this callback for this cycle
    self.active = 0;
}

```

We do something quite different here: this `reproduction()` callback runs only once per cycle, not once per individual! It disables itself at the end of its own execution by setting its active flag to 0 (see section 25.11 for discussion of this feature, which we haven't seen much before now). The `reproduction()` callback is called by SLiM for some particular focal individual, but the callback ignores that individual and instead generates offspring for all of the individuals in the whole population all at once. This is perfectly fine, and can be a useful strategy when the reproduction behavior of individuals is non-independent; with monogamy, for example, once two individuals have formed a mating pair those individuals are not available to be chosen as mates by any other individuals, so mating behavior in this model is non-independent, and it is thus more convenient to handle the reproduction of the whole population *en masse*.

The first thing we want to do is choose all of the monogamous mating pairs. The order of individuals in SLiM is not guaranteed to be random in SLiM, so it would be unwise to simply pair individuals 0 and 1, 2 and 3, etc.; that could lead to biases in mate choice that could manifest in strangely skewed genetics over time. Instead, we use the `sample()` function to draw a complete sample from the population, without replacement, effectively randomizing its order. Then we pair individuals 0 and 1, 2 and 3, etc., from *that*, which is safe. We do that pairing with a `for` loop over

the even values up to `p1.individualCount - 2` (guaranteeing that a pair of individuals remains the last time through the loop, not just an odd one out at the end). Individuals `i` and `i+1` are then taken to be a monogamous mating pair. Of course one could implement any mating scheme at all; females could choose males assortatively, or based upon their physical condition, or in any other manner, and monogamy does not need to be enforced (as we saw in the previous two recipes).

Next, we want to generate a litter for that mating pair. We draw the size of the litter from a Poisson distribution with a mean of 2.7, arbitrarily. At the risk of sounding like a broken record, of course litter size could depend upon anything at all – the genetics of the two parents, their genetic compatibility, their respective conditions, their ages, their fecundity the previous year, their phenotypic match with their environment, etc. Here, we happen to use a Poisson distribution with a mean of 2.7. That gives us an integer litter size; we then generate the litter by calling `addCrossed()` that many times with the same parents.

The only thing left is for the `reproduction()` callback to deactivate itself, as explained above. This model has the same output code as section 16.1's recipe; a typical run produces:

```
...
1995: 497 (6)
1996: 531 (7)
1997: 524 (6)
1998: 497 (7)
1999: 529 (8)
2000: 511 (6)
```

Note that the equilibrium age here is around 6 to 7, where in section 16.1 it was more around 10 to 15. That is because section 16.1's recipe produced one offspring per individual per cycle (not counting being chosen as a mate by another reproducing individual), whereas this recipe produces about 1.35 (half of 2.7). That floods this model with young individuals, relative to the earlier model. Density-dependent mortality and carrying capacity remain the same, however, so skewing the age distribution towards juveniles in that manner inevitably means fewer old individuals and a shorter expected lifespan. That happens because with more offspring generated, the pre-mortality population size is larger, and so (given that the carrying capacity is the same) density-dependent fitness effects are stronger, individual fitness is lower, and the probability of mortality per individual is higher, reducing the expected lifespan.

## 16.4 Beneficial mutations and absolute fitness

Thus far, we have only looked at neutral nonWF models. Fitness in nonWF models is absolute and affects survival, where in WF models it is relative and affects mating success; this makes fitness dynamics a bit different between nonWF and WF models. In particular, since one cannot survive with more than a 100% probability, fitness values above 1.0 in nonWF models do not benefit the individual at all; fitness values above 1.0 are interpreted as being 1.0. However, fitness is also multiplicative (in both nonWF and WF models), and the important thing is the final fitness value for an individual. The way this works out in practice can be a bit counterintuitive, so in this section we will explore a simple model of an introduced beneficial mutation that sweeps to fixation in a population, and we will look at the population dynamics as it does so.

The recipe is again based on that of section 16.1:

```
initialize() {
  initializeSLIMModelType("nonWF");
  defineConstant("K", 500);
```

```

        initializeMutationType("m1", 0.5, "f", 0.0);
        m1.convertToSubstitution = T;
        initializeMutationType("m2", 1.0, "f", 0.5);    // dominant beneficial

        initializeGenomicElementType("g1", m1, 1.0);
        initializeGenomicElement(g1, 0, 99999);
        initializeMutationRate(1e-7);
        initializeRecombinationRate(1e-8);
    }
    reproduction() {
        for (i in 1:5)
            subpop.addCrossed(individual, subpop.sampleIndividuals(1));
    }
    1 early() {
        sim.addSubpop("p1", 10);
    }
    100 early() {
        mutant = sample(p1.individuals.genomes, 10);
        mutant.addNewDrawnMutation(m2, 10000);
    }
    early() {
        p1.fitnessScaling = K / p1.individualCount;
    }
    late() {
        inds = p1.individuals;
        catn(sim.cycle + ": " + size(inds) + " (" + max(inds.age) + ")");
    }
    2000 late() {
        sim.outputFull(ages=T);
    }
}

```

The changes from section 16.1 are minor. In the `initialize()` callback we create a mutation type `m2`, for beneficial mutations; this has quite a strong selection coefficient and is dominant, to make it less likely to be lost to drift at the very beginning, but that is unimportant to the point of this recipe; we just don't want to have to get into making the recipe conditional on fixation since that is an extra complication (see section 9.2). In the `reproduction()` callback each individual now reproduces five times per cycle, creating very strong density-dependent fitness effects; this is a highly fecund species. That is not essential to the point of this recipe, but it will make the effect more obvious. In a new `100 early()` event we now select one genome from the population at random, and add an `m2` mutation to it. The rest of the model is the same. The `m2` mutation will (usually) sweep to fixation, and we will look at the resulting population size and age structure.

At the beginning, the model rapidly grows to the carrying capacity and stays there (each output line, remember, shows the cycle counter followed by the population size and the age of the oldest individual, post-mortality):

```

1: 10 (0)
2: 60 (1)
3: 360 (2)
4: 513 (3)
5: 473 (4)
6: 460 (4)
7: 480 (3)
8: 504 (3)
9: 505 (4)

```

The population size is around 500, the oldest individual usually 3 or 4. So far so good. Now let's look at the output at the end of the run, in a run in which the beneficial mutation fixes:

```
1996: 768 (4)
1997: 771 (3)
1998: 781 (4)
1999: 763 (3)
2000: 802 (4)
```

The population size is now fluctuating around 760 to 800, although the typical age of the oldest individual is still 3 to 4. What happened to our carrying capacity of 500?

The answer lies in the fitness values calculated by SLiM. Two things affect fitness in this model. One is density-dependent mortality, as embodied by the `fitnessScaling` value set in the `early()` callback. When the population is above carrying capacity (as it is in every tick from 4 onwards, in this model, due to the many juveniles), this scaling value will be less than 1.0; given how many juveniles this model makes, it is probably usually 0.2 or lower, in fact. The other thing affecting fitness is the beneficial mutation; since it is dominant, it gives any individual possessing it a fitness effect of 1.5 since its selection coefficient is 0.5. These fitness effects are multiplicative, so once the beneficial mutation has fixed, every individual will have a fitness value of approximately  $0.2 * 1.5$ , or 0.3. This means, in effect, that fewer individuals will die, and the carrying capacity will increase. Which is precisely what we see.

This may be unexpected for those who are used to the world of Wright-Fisher models, but it makes good biological sense. If every individual possesses a mutation that makes them more fit – more likely to survive – then the population size *ought* to increase. If there is some reason why that shouldn't happen, such as a hard limit on the amount of available food, then you ought to add that biological detail to your model explicitly. This sort of thing is precisely why the individual-based nature of nonWF models, with emergent dynamics for things like population size and age structure, has the potential to be more biologically realistic.

This point becomes even more pointed if you write a model in which new beneficial mutations can arise spontaneously. In such a nonWF model, absolute fitness would increase a little bit more with every new beneficial mutation, and the population would evolve toward a “Darwinian demon” with infinite absolute fitness and infinite population size. Fundamentally, that is just a more extreme case of the same situation as in this recipe: if your model tells SLiM that absolute fitness has increased, population size will increase concomitantly. If you want some mechanism to hold that tendency in check, you need to add that mechanism to your model yourself.

In this recipe, for example, it would certainly be possible to force the model to maintain the same carrying capacity throughout, but trying to do so might just expose how biologically unrealistic that constraint really is. If the carrying capacity is the same before and after the beneficial mutation fixes, that means that the survival probability is the same (assuming we don't alter reproductive output). So once the beneficial mutation has fixed, it apparently no longer confers any benefit to the carrier – even though it *did* confer a benefit (relative to the non-carrier individuals) earlier on, before the beneficial mutation had fixed. What has changed? Nothing about the environment, and the population size has not changed (since we are making the carrying capacity fixed) – and yet somehow, almost magically, the fitness benefit that used to exist has evaporated. The only thing that has changed, in fact, is that now *other* individuals also possess the mutation, whereas early on in the sweep few or none did. And that suggests one way that we could make the carrying capacity fixed in this model: add in negative frequency-dependent selection (see section 10.4.1). Maybe there's a good biological reason for that: limited resources, for example, such that a mutation that makes an individual better at obtaining those resources confers less and less benefit as the mutation gets more and more common. If that's the biology,

then great, model that. But if one is tempted to hold the population size fixed simply because one is used to Wright–Fisher models that hold the population size fixed – not for any biological reason – then it would probably be best to think twice. The lesson of this recipe, in other words, might be: Model the biology, not your own modeling assumptions. Let emergent dynamics emerge.

Or if you really want to stay in the world of Wright–Fisher assumptions, then write a WF model; there’s nothing wrong with that, as long as you understand the assumptions you’re making.

## 16.5 A metapopulation extinction-colonization model

Our models so far have been of a single subpopulation, but nonWF models have many advantages for modeling migration that we will explore in this recipe and the next. Here, we will model a metapopulation undergoing local extinctions and then re-colonizations from other subpopulations. This would be difficult to implement in a WF model, because population size is not allowed to go to zero (that signifies removal of the subpopulation from the model), and because re-colonization by migrants does not happen naturally (it would have to be done as re-creation of the extinct subpopulation using `addSubpopSplit()`, and the founders could come only from a single source subpopulation). In a nonWF model, however, this is quite straightforward. For simplicity, we will model a non-spatial metapopulation in which every subpopulation is connected to every other by migration of equal strength:

```
initialize() {
  initializeSLiMModelType("nonWF");
  defineConstant("K", 50);      // carrying capacity per subpop
  defineConstant("N", 10);      // number of subpopulations
  defineConstant("m", 0.01);    // migration rate
  defineConstant("e", 0.1);     // subpopulation extinction rate

  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;

  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
reproduction() {
  subpop.addCrossed(individual, subpop.sampleIndividuals(1));
}
1 early() {
  for (i in 1:N)
    sim.addSubpop(i, (i == 1) ? 10 else 0);
}
early() {
  // random migration
  nIndividuals = sum(sim.subpopulations.individualCount);
  nMigrants = rpois(1, nIndividuals * m);
  migrants = sample(sim.subpopulations.individuals, nMigrants);

  for (migrant in migrants)
  {
    do dest = sample(sim.subpopulations, 1);
    while (dest == migrant.subpopulation);

    dest.takeMigrants(migrant);
  }
}
```

```

// density-dependence and random extinctions
for (subpop in sim.subpopulations)
{
  if (runif(1) < e)
    sim.killIndividuals(subpop.individuals);
  else
    subpop.fitnessScaling = K / subpop.individualCount;
}
}
late() {
  if (sum(sim.subpopulations.individualCount) == 0)
    stop("Global extinction in cycle " + sim.cycle + ".");
}
2000 late() {
  sim.outputFixedMutations();
}

```

We start off by defining constants for the carrying capacity  $K$ , the subpopulation count  $N$ , the per-individual migration rate  $m$ , and the per-tick probability  $e$  of a local extinction event in a given subpopulation, such as might occur due to a forest fire or a flood. We then set up a neutral nonWF model with simple offspring generation (one offspring per individual per tick), and create  $N$  subpopulations. The first subpopulation begins with 10 individuals; the rest begin empty, awaiting migrants (which is legal in nonWF models; subpopulations can be empty). At the end of the model, we have a `late()` event that halts the model with a message if all subpopulations have gone extinct; if we make it to tick 2000 we output fixed mutations and stop.

The interesting code is in the middle, in the large `early()` event. This first implements random migration by drawing the number of migrants from a Poisson distribution and then sampling migrants at random from the full population; this gives each individual the same probability  $m$  of migrating, and is more efficient than doing a random draw for each individual. It then loops through the chosen migrants, finds their destination subpopulation (ensuring that it is not the subpopulation the migrant already occupies), and finally calls `takeMigrants()` to move the individual to its new home. The `takeMigrants()` call removes the individual from its old subpopulation and adds it immediately to the target subpopulation; it is the way that migration is typically implemented in nonWF models. Note that the design of this code avoids choosing and moving a migrant, and then accidentally choosing that same individual as a migrant again and moving it again; all migrants are selected, and then all migrants are moved. This design is generally a good idea, to avoid accidentally skewing the migration rates for subpopulations away from their intended rates. (The `migrant` property of `Individual` could also be used to prevent this, together with the ability of `sampleIndividuals()` to select individuals that have not already migrated.)

The second half of the `early()` event implements both density-dependence and random local extinction events. For each subpopulation it draws from a random uniform distribution, and if the draw is less than the probability of local extinction  $e$ , it calls `sim.killIndividuals()` to kill all individuals in the subpopulation:

```
sim.killIndividuals(subpop.individuals);
```

We haven't seen the `killIndividuals()` method before, but it is very straightforward: the individuals are simply removed from their subpopulation(s) immediately. This method can only be called in nonWF models, since WF models have a fixed subpopulation size.



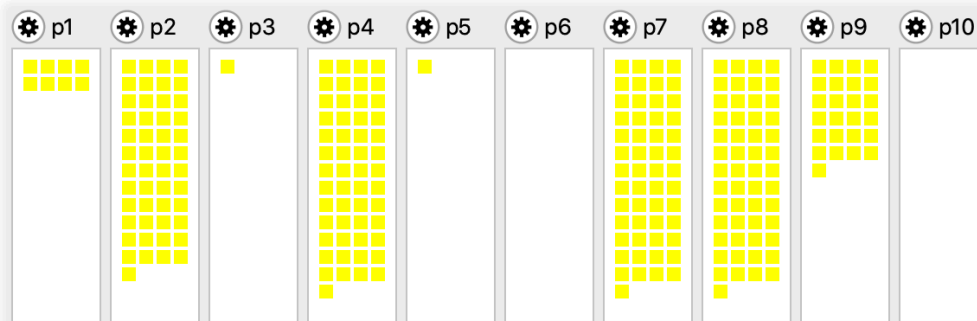
In an older version of this recipe, this line was instead:

```
subpop.fitnessScaling = 0.0;
```

This sets the subpopulation's `fitnessScaling` property to zero, effectively reducing the fitness of all individuals in the subpopulation to zero and thus killing them all. The two approaches are almost identical; the only difference, really, is that `killIndividuals()` kills the individuals immediately, whereas the `fitnessScaling` approach just sets them up to be killed when the viability/survival tick cycle stage rolls around. In particular, if the script were to access `subpop.individuals` against after the `killIndividuals()` call it would be empty, whereas after setting `fitnessScaling`, `subpop.individuals` would still contain all the same individuals, since they have not yet died in that version of the script. For our purposes here, the two approaches work equally well; the recipe has switched to use `killIndividuals()` just to provide an example of its use.

So that takes care of local extinction events. The rest of the time – when the `runif()` draw is not below the local extinction threshold – `fitnessScaling` is set based on subpopulation density as usual, producing growth up to the carrying capacity *K* for each subpopulation.

When run, this model produces fairly realistic extinction-colonization dynamics; after a subpopulation is hit by an extinction event, it will eventually be recolonized and then undergo rapid growth until reaching carrying capacity again. Here's a snapshot from SLiMgui mid-run:

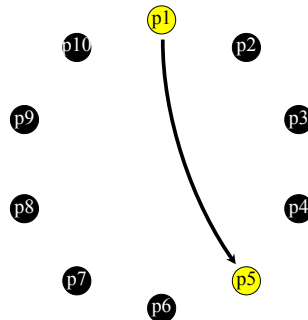


Two subpopulations are presently empty, two have just been recolonized by a single migrant, two others are at intermediate stages of growth, and four are roughly at carrying capacity (which is, as usual in nonWF models, not a hard limit). Note that all individuals are displayed here in yellow, because SLiMgui normalizes away subpopulation-level `fitnessScaling` constants in order to display individual fitness prior to density-dependent scaling, since that is generally what one is interested in seeing; it does the same thing, in a sense, in WF models too. In fact, though, the populations below carrying capacity have a fitness greater than or equal to 1.0 that is driving their growth (by suppressing density-dependent mortality), and the populations at carrying capacity have a fitness less than 1.0 that compensates for their births to keep them at equilibrium.

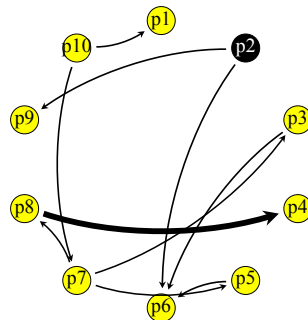
If the migration rate is too low, or the extinction rate is too high, the whole population will often go extinct; but with less apocalyptic parameter values recolonization will keep up with extinction and the population will persist (for a while, anyway; the design of the extinction events in this model means that sudden global extinction of all subpopulations will happen with probability  $e^{-N}$ , so eventually the model will always go extinct as long as  $e > 0$ ).

A nice feature in SLiMgui that is worth pointing out is that it keeps metrics regarding the behavior of your model, and can display those metrics for you. You might recall the Population Visualization graph that SLiMgui can display to show population sizes, fitnesses, and migration patterns (see, e.g., sections 5.1.3 and 5.2.1). In nonWF models the migration rates between

subpopulations are not set ahead of time, but are instead an emergent property of the model, as we have seen in this recipe. SLiMgui will monitor the actual migration generated in the running nonWF model and display it in the Population Visualization graph. For example, here's the pattern of migration in tick 3 of a run of this recipe:



This shows that p5 has just received a migrant from p1; this is the initial colonization of p5, in fact. The other subpopulations are black because they are empty at this point. Later in the run, it might look like this instead:



Lots is going on here; p8 sent a relatively large proportion of its population to p4, by chance (thus the thicker arrow) – maybe two or three or four migrants. Migrants were sent from p2 to both p6 and p9, and then, immediately after, p2 got hit by an extinction event. And so forth. This facility can be quite useful for debugging migration code, or for seeing how the pattern of migration changes over time when it depends upon other model state (as in the next recipe).

In fact, SLiMgui monitors and displays other emergent metrics in nonWF models, too. In the subpopulation table, for example, where things like the cloning rate, selfing rate, and sex ratio are displayed for WF models, the actual, observed metrics for those properties will be displayed by SLiMgui for nonWF models. This particular recipe is not a good showcase for that feature, however, since it is an asexual model involving only biparental mating.

## 16.6 Habitat choice

In the previous section migration in nonWF models using the `takeMigrants()` method was introduced. Here we will further explore migration in nonWF models. In WF models, as you may recall, migration occurs during offspring generation: parents from one subpopulation mate, but their offspring gets added to a different subpopulation if it migrates. This type of juvenile migration is the only possibility in WF models; but nonWF models are not restricted in that way, and here we will construct a nonWF model of migration that can occur at any age. Each tick, individuals will choose which environment they will live in, a phenomenon commonly called “habitat choice”. All else equal, they will prefer the environment they are already in; but if the other environment is

better for them, then with a non-zero probability they will decide to move. This model will also include variation in the individual propensity to migrate, and emergent variation in the total number of migrants in each tick – both difficult to capture in WF models.

The basic design of this model is patterned after the recipe in section 10.2: we have two subpopulations, p1 and p2, and a mutation type, m2, that represents relatively rare mutations that are beneficial in p1 but deleterious in p2. For balance, let's also have a mutation type m3 for mutations that are deleterious in p1 but beneficial in p2. Finally, let's throw a spanner into the works by making offspring initially go to a random subpopulation, not always to the subpopulation of their parents (perhaps representing some sort of shared spawning environment from which juveniles initially disperse randomly).

Here is the model except for the habitat choice code:

```
initialize() {
  initializeSLIMModelType("nonWF");
  defineConstant("K", 500);

  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;
  initializeMutationType("m2", 0.5, "e", 0.1); // deleterious in p2
  m2.color = "red";
  initializeMutationType("m3", 0.5, "e", 0.1); // deleterious in p1
  m3.color = "green";

  initializeGenomicElementType("g1", c(m1,m2,m3), c(0.98,0.01,0.01));
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
reproduction() {
  dest = sample(sim.subpopulations, 1);
  dest.addCrossed(individual, subpop.sampleIndividuals(1));
}
1 early() {
  sim.addSubpop("p1", 10);
  sim.addSubpop("p2", 10);
}
early() {
  p1.fitnessScaling = K / p1.individualCount;
  p2.fitnessScaling = K / p2.individualCount;
}
mutationEffect(m2, p2) { return 1/effect; }
mutationEffect(m3, p1) { return 1/effect; }

1000 late() {
  for (id in 1:2)
  {
    subpop = sim.subpopulations[sim.subpopulations.id == id];
    s = subpop.individualCount;
    inds = subpop.individuals;
    c2 = sum(inds.countOfMutationsOfType(m2));
    c3 = sum(inds.countOfMutationsOfType(m3));
    catn("subpop " + id + " (" + s + "): " + c2 + " m2, " + c3 + " m3");
  }
}
```

The structure of this model is quite predictable. The `initialize()` callback sets up the local-adaptation `m2` and `m3` mutation types, and the two `mutationEffect()` callbacks render mutations of those types deleterious in one or the other subpopulation. The `reproduction()` callback generates each new offspring in a randomly chosen subpopulation, representing random juvenile dispersal from a spawning environment as mentioned above. We also have the usual nonWF density-dependent fitness scaling, and we have an output event that prints information about the two subpopulations (as we will discuss below).

The interesting part, then, is the large `early()` event for habitat choice, which should be inserted just above the density-dependence `early()` event above (so that density-dependence is based upon post-migration subpopulation sizes, not the pre-migration sizes):

```
early() {
  // habitat choice
  inds = sim.subpopulations.individuals;
  inds_m2 = inds.countOfMutationsOfType(m2);
  inds_m3 = inds.countOfMutationsOfType(m3);
  pref_p1 = 0.5 + (inds_m2 - inds_m3) * 0.1;
  pref_p1 = pmax(pmin(pref_p1, 1.0), 0.0);
  inertia = ifelse(inds.subpopulation.id == 1, 1.0, 0.0);
  pref_p1 = pref_p1 * 0.75 + inertia * 0.25;
  choice = ifelse(runif(inds.size()) < pref_p1, 1, 2);
  moving = inds[choice != inds.subpopulation.id];
  from_p1 = moving[moving.subpopulation == p1];
  from_p2 = moving[moving.subpopulation == p2];
  p2.takeMigrants(from_p1);
  p1.takeMigrants(from_p2);
}
```

The logic of this event goes through several steps, but is not complicated. First it gets a vector of all individuals, and derives vectors of the number of `m2` and `m3` mutations possessed by each individual; each of these is a vector of counts corresponding to the original vector of individuals. It then computes a vector of habitat preferences for each individual: starting from a neutral preference of `0.5`, the more `m2` mutations an individual has, and the fewer `m3` mutations it has, the more that individual prefers `p1` over `p2`. Each `m2` or `m3` mutation shifts its preference by only 10%, however, so these preferences are not initially very strong; and this preference is clamped to the range `[0.0, 1.0]` so that even once many `m2` and `m3` mutations exist this preference is still bounded. Next the script computes an “inertial” habitat preference for `p1` over `p2`, expressing a simple desire not to move; for individuals presently in `p1` this is `1.0`, whereas for those in `p2` it is `0.0`. The final habitat preference is computed as a weighted average of these two considerations; in this recipe the genetically-based preference is given a weight of `0.75` and the inertial preference is given a weight of `0.25`, making individuals fairly strongly inclined to move, but this balance is a free parameter in the model. Next, we actually decide which subpopulation each individual chooses, by comparing a random uniform draw to the individual’s weighted preference. (Note that these calculations continue to be vectorized; this event handles all migration with a single sequence of calculations.) The `moving` vector is the subset of individuals that chose a different subpopulation than they currently occupy; these individuals will migrate, while the rest stay put. The script then finds which migrants are currently in `p1` (moving to `p2`) and which are in `p2` (moving to `p1`), and then, finally, it makes `takeMigrants()` calls that actually move those individuals. This method simply removes the given individuals from their current subpopulation and inserts them into the target subpopulation. One could implement habitat choice in many different ways, embodying different decision-making processes for the individuals involved, different degrees of knowledge

about the available habitat options, different approaches to the stochasticity of the choice, and so forth; this is just one simple algorithm for demonstration purposes.

Without any migration (if the `habitat-choice early()` event is commented out, in other words), this model will “flip” to favor one subpopulation based upon the `m2` and `m3` mutations that happen to do well early on; the successful subpopulation will grow very large (as its carrying capacity increases, because so many individuals carry mutations that are beneficial in that environment; see section 16.4), whereas the unsuccessful subpopulation will shrink toward zero (swamped by vast numbers of offspring from the other subpopulation that are massively maladapted and immediately die). Output from that version of the model typically looks like this:

```
subpop 1 (191): 0 m2, 1180 m3
subpop 2 (1314): 0 m2, 8072 m3
```

The model has “flipped” toward subpop `p2`, which is now 1314 individuals to `p1`’s 191 individuals. There are quite a large number of copies of `m3`-type alleles in play, whereas there are no `m2`-type alleles. Subpop `p1` will never be able to dig itself out of this hole, since it gets swamped with new offspring from `p2` every tick that carry `m3` alleles and not `m2` alleles, and any `m2` alleles it manages to scrape together get diluted into `p2` and selected out. If the model runs longer, `p1` will effectively go extinct except for whatever cohort of confused migrants arrives to repopulate it in each tick.

But with the `early()` event that implements habitat choice, the story is very different. Now, if the probability of correct habitat choice is sufficiently high, the subpopulations can diverge; even though they still sabotage each other with maladapted offspring, those offspring tend to migrate over to the subpopulation where they are more fit. One subpopulation often grows significantly larger than the other, but the model no longer “flips”; the smaller subpopulation is much more able to persist. Output from the model with habitat choice:

```
subpop 1 (620): 2838 m2, 173 m3
subpop 2 (1065): 149 m2, 7177 m3
```

The subpopulations are now much closer in size, and show clear divergence in their `m2` and `m3` profiles. In SLiMgui the different haplotypes of the two subpopulations are immediately visible, since the `m2` and `m3` mutations have been given different colors. Subpopulation `p2` is larger here, since there are more `m3` alleles segregating, but it is no longer swamping `p1` or diluting away the `m2` alleles to such an extent that `p1` can’t also thrive, and if new `m2` mutations arise they will often be able to establish themselves in `p1` so the current imbalance may not even persist. Divergence here is much more successful, and in fact it is effective enough that neutral sites will diverge as well, indicating that this mechanism is sufficient to generate substantial reproductive isolation. The degree of isolation will depend upon parameters such as the strength of habitat choice versus the “inertial” preference for the current habitat, and the strength of the divergent selection on the `m2` and `m3` alleles, of course.

This recipe goes beyond the capabilities of WF models in several important ways. One way is that individuals of all ages migrate in this model, not just newly created juveniles as in WF models. Indeed, in this model the same individual may jump back and forth between `p1` and `p2` several times, if it has no strong preference. A second way is that the migratory behavior here is based upon individual genetic state. In principle this is possible to implement in WF models, using a `modifyChild()` callback that accepts or rejects proposed migrant offspring based upon their genetics, but in practice it would be difficult and problematic. A third way is that the amount of migration in this model is itself condition-dependent; if there are many maladapted individuals, there will be many migrants, if fewer, fewer migrants. This would be difficult to do in a WF model since SLiM then fulfills a pre-set migration rate; that pre-set rate would have to be carefully

predicted and altered in every tick in order to try to foretell the desired result from offspring generation, which would be very clumsy if it worked at all. Migration is an individual choice, so it is much simpler and more natural to model it as such, as nonWF models do.

Many interesting extensions to this model could be made. For example, one could impose a fitness cost upon migration, and then allow the propensity for migration to itself evolve by making the weighting between habitat choice and “inertia” depend upon a quantitative trait. Individuals that chose to migrate too often would suffer a high cost, but those that did not migrate at all, even when strongly maladapted, would also be penalized, and so some intermediate, optimal migration tendency would perhaps tend to evolve. Such a model would rely heavily upon the advantages of the individual-based migration of nonWF models; it would be difficult to do as a WF model.

## 16.7 Evolutionary rescue after environmental change

The evolutionary response of a population to environmental change, and the probability of extinction if that response is insufficient, is the subject of an important class of models called “evolutionary rescue” models – particularly relevant in this era of anthropogenic climate change. Evolutionary rescue can be based upon standing genetic variation, new mutations that provide new adaptive potential, or genetic variation brought in by migrants. Here we will look at a QTL-based model of evolutionary rescue that may (or may not) occur as a result of both standing genetic variation and new mutations. This model is based heavily upon other QTL models in this manual (see chapter 13), adapted here to illustrate that QTL-based approaches are entirely compatible with nonWF models.

Let’s look at this model piece by piece, beginning with `initialize()`:

```
initialize() {
  initializeSLIMModelType("nonWF");
  defineConstant("K", 500);
  defineConstant("opt1", 0.0);
  defineConstant("opt2", 10.0);
  defineConstant("Tdelta", 10000);

  initializeMutationType("m1", 0.5, "n", 0.0, 1.0); // QTL
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
```

For simplicity and speed, we define only a QTL mutation type, `m1`; this model has no neutral mutations in it (but of course they would be trivial to add). Each QTL is drawn from a normal distribution centered on `0.0` with a standard deviation of `1.0` (which are important parameters for this model, since the exact nature of the standing genetic variation and mutational variance will be important). Besides this, the initialization is quite standard. We set up defined constants for the carrying capacity (`K`), for the phenotypic optimum before and after environmental change (`opt1` and `opt2`), and for the time when the environmental change will occur (`Tdelta`).

Next we set up our reproduction and our initial population:

```
reproduction() {
  subpop.addCrossed(individual, subpop.sampleIndividuals(1));
}
1 early() {
  sim.addSubpop("p1", 500);
}
```



This is boilerplate, except that here we start the population already at its carrying capacity so that if we configure the environmental change to occur immediately at the beginning of the model (as we will try below), the population is not at a disadvantage due to not yet having grown to capacity.

We need our QTL machinery, which is quite simple in this model:

```
early() {
  // QTL-based fitness
  inds = sim.subpopulations.individuals;
  phenotypes = inds.sumOfMutationsOfType(m1);
  optimum = (sim.cycle < Tdelta) ? opt1 else opt2;
  deviations = optimum - phenotypes;
  fitnessFunctionMax = dnorm(0.0, 0.0, 5.0);
  adaptation = dnorm(deviations, 0.0, 5.0) / fitnessFunctionMax;
  inds.fitnessScaling = 0.1 + adaptation * 0.9;
  inds.tagF = phenotypes; // just for output below

  // density-dependence with a maximum benefit at low density
  p1.fitnessScaling = min(K / p1.individualCount, 1.5);
}
mutationEffect(m1) { return 1.0; }
```

The `mutationEffect(m1)` callback makes the direct fitness effect of `m1` mutations neutral, as usual in such QTL models; the only effect of QTL mutations on fitness is indirect, through their effect on individual phenotypic values (see chapter 13 for an introduction to the design of QTL models).

The `early()` event calculates individual phenotypes as the sum of the effects of all QTLs possessed by the individual. It then decides which phenotypic optimum is in effect, calculates the deviation of each individual from that optimum, and calculates the degree of adaptation of each individual from that using `dnorm()` (normalizing the adaptation values to the range (0,1] with `fitnessFunctionMax`). Finally, `fitnessScaling` values for individuals are set based upon their adaptation; a perfectly adapted individual will have an adaptation value of 1.0 and thus a `fitnessScaling` value of 1.0, whereas an infinitely maladapted individual will have an adaptation value of 0.0 and thus a `fitnessScaling` value of 0.1, the “floor” in this model (a crucial parameter that will influence the probability of evolutionary rescue).

The `early()` event also, at the end, implements density-dependent population regulation. This is done in the usual way, except that a maximum of 1.5 is imposed with `min()`. In principle, this sort of correction ought to have been imposed on all our other nonWF models, but in models that do not include deleterious mutations, and which are not expected to spend time at low density, it is unimportant. The rationale for the correction is that being at low population density does convey *some* benefit (assuming the absence of Allee effects, as we have been doing), allowing individuals to survive and reproduce at their full capacity even when they carry some minor deleterious mutations that would negatively impact them if they were in a population at full carrying capacity – but this benefit is surely limited! Typically, a population will not thrive if it carries many large-effect deleterious mutations, even if it is released from all density-dependent pressures. The maximum of 1.5 here embodies that intuitive fact, by stating that the fitness benefit of low density can never be more than a multiplicative fitness effect of 1.5. (This is another important model parameter, of course.)

Finally, we have our output and termination events:

```

late() {
  if (p1.individualCount == 0)
  {
    // stop at extinction
    catn("Extinction in cycle " + sim.cycle + ".");
    sim.simulationFinished();
  }
  else
  {
    // output the phenotypic mean and pop size
    phenotypes = p1.individuals.tagF;

    cat(sim.cycle + ": " + p1.individualCount + " individuals");
    cat(", phenotype mean " + mean(phenotypes));
    if (size(phenotypes) > 1)
      cat(" (sd " + sd(phenotypes) + ")");
    catn();
  }
}
20000 late() { sim.simulationFinished(); }

```

The simulation checks for extinction in every tick, and stops with a termination message. Otherwise, it prints a summary with the current population size, and the mean and standard deviation of the distribution of phenotypes. If extinction is avoided, the model stops after tick 20000.

Running this recipe as configured, we begin with no genetic diversity at all:

```

1: 500 individuals, phenotype mean 0 (sd 0)
2: 529 individuals, phenotype mean 0.000628046 (sd 0.107673)
3: 499 individuals, phenotype mean -0.00650893 (sd 0.152125)
4: 470 individuals, phenotype mean -0.00802766 (sd 0.188649)
5: 497 individuals, phenotype mean 0.00997383 (sd 0.225336)
...

```

By tick 10000, when the environment changes, we have built up some standing genetic variation – although really not much more than we had just a few ticks in:

```

...
9995: 493 individuals, phenotype mean -0.101779 (sd 0.616717)
9996: 498 individuals, phenotype mean -0.139806 (sd 0.592579)
9997: 518 individuals, phenotype mean -0.146021 (sd 0.55228)
9998: 493 individuals, phenotype mean -0.127487 (sd 0.537018)
9999: 500 individuals, phenotype mean -0.139612 (sd 0.531529)

```

Then we hit tick 10000, the optimum suddenly changes to 10.0, and things get ugly fast:

```

10000: 123 individuals, phenotype mean -0.095242 (sd 0.560048)
10001: 77 individuals, phenotype mean 0.0100077 (sd 0.628869)
10002: 54 individuals, phenotype mean -0.191957 (sd 0.624277)
10003: 35 individuals, phenotype mean -0.0215701 (sd 0.55838)
10004: 18 individuals, phenotype mean 0.211882 (sd 0.862124)
10005: 7 individuals, phenotype mean 0.196655 (sd 1.07714)
10006: 5 individuals, phenotype mean 0.200521 (sd 0.144637)
10007: 3 individuals, phenotype mean 0.123028 (sd 0.0531924)
10008: 2 individuals, phenotype mean 0.153739 (sd 0)
10009: 1 individuals, phenotype mean 0.153739
Extinction in cycle 10010.

```

The population evolved toward the new optimum, but not quickly enough to overcome its crash in population size, and it went extinct only eleven ticks after the environment shifted. However, this is actually not the typical outcome for the model. Here's another run just before the environmental change:

```
...
9995: 494 individuals, phenotype mean 0.152236 (sd 0.576086)
9996: 511 individuals, phenotype mean 0.144287 (sd 0.590623)
9997: 503 individuals, phenotype mean 0.151616 (sd 0.616162)
9998: 489 individuals, phenotype mean 0.115655 (sd 0.617752)
9999: 508 individuals, phenotype mean 0.11127 (sd 0.611983)
```

And here's what happened after:

```
10000: 112 individuals, phenotype mean 0.373886 (sd 0.714775)
10001: 84 individuals, phenotype mean 0.394201 (sd 0.768464)
10002: 58 individuals, phenotype mean 0.447692 (sd 0.855927)
10003: 48 individuals, phenotype mean 0.689259 (sd 0.925998)
10004: 31 individuals, phenotype mean 0.877411 (sd 1.12496)
10005: 29 individuals, phenotype mean 1.12907 (sd 1.10892)
10006: 25 individuals, phenotype mean 1.50111 (sd 1.29306)
10007: 20 individuals, phenotype mean 1.45553 (sd 1.30944)
10008: 17 individuals, phenotype mean 1.88997 (sd 1.42636)
10009: 15 individuals, phenotype mean 2.67739 (sd 1.60781)
10010: 16 individuals, phenotype mean 2.92828 (sd 1.58097)
10011: 26 individuals, phenotype mean 3.32725 (sd 1.53169)
10012: 35 individuals, phenotype mean 3.87526 (sd 1.24333)
10013: 56 individuals, phenotype mean 4.13832 (sd 1.10771)
10014: 98 individuals, phenotype mean 4.27727 (sd 1.12148)
10015: 158 individuals, phenotype mean 4.54129 (sd 0.940975)
10016: 285 individuals, phenotype mean 4.66029 (sd 0.881386)
10017: 321 individuals, phenotype mean 4.84315 (sd 0.807692)
10018: 314 individuals, phenotype mean 4.98733 (sd 0.726859)
10019: 355 individuals, phenotype mean 5.08917 (sd 0.712417)
10020: 337 individuals, phenotype mean 5.18797 (sd 0.735515)
...
```

The population size dips down as low as 15 individuals, and the phenotypic optimum still seems quite distant, but it manages to stage a full recovery; it's back up to carrying capacity within about a hundred ticks.

In twenty runs of the model, extinction occurred nine times and evolutionary rescue occurred eleven times. We can test the importance of standing genetic variation for rescue by setting `Tdelta` to 0, making the optimum be 10.0 from the start of the model with no chance for standing genetic variation to build up; in this variant of the model, extinction occurred sixteen out of twenty times, rescue only four times. So: probably important. Which might seem a bit surprising, since the variance in phenotype is really not large in tick 9999; but there are, nevertheless, a lot of useful QTLs that can be brought together by recombination.

We can also test the importance of new mutations to evolutionary rescue, by setting the mutation rate to 0.0 when the environment changes; the population will then have nothing but standing variation at its disposal. A proper test of this would require many runs of the model, to see how often it occurs and so forth, but I can state that evolutionary rescue does sometimes occur from the standing variation alone. Here's the population just before the change, in one run of that variant of the model:

```

...
9995: 462 individuals, phenotype mean -0.020969 (sd 0.842111)
9996: 489 individuals, phenotype mean -0.0413428 (sd 0.843401)
9997: 488 individuals, phenotype mean 0.0205512 (sd 0.805816)
9998: 496 individuals, phenotype mean -0.0103366 (sd 0.744229)
9999: 488 individuals, phenotype mean -0.0287371 (sd 0.829185)

```

And here's the recovery, in all its glory:

```

10000: 110 individuals, phenotype mean 0.0798129 (sd 0.941083)
10001: 86 individuals, phenotype mean 0.199899 (sd 0.936995)
10002: 70 individuals, phenotype mean 0.369471 (sd 0.986702)
10003: 52 individuals, phenotype mean 0.643418 (sd 1.1836)
10004: 46 individuals, phenotype mean 0.780035 (sd 1.25709)
10005: 42 individuals, phenotype mean 1.23978 (sd 1.55453)
10006: 43 individuals, phenotype mean 1.86605 (sd 1.7264)
10007: 39 individuals, phenotype mean 2.62644 (sd 1.805)
10008: 48 individuals, phenotype mean 3.24882 (sd 1.82018)
10009: 67 individuals, phenotype mean 3.83253 (sd 1.69954)
10010: 101 individuals, phenotype mean 4.43066 (sd 1.53875)
10011: 165 individuals, phenotype mean 4.76209 (sd 1.42616)
10012: 289 individuals, phenotype mean 5.14574 (sd 1.27371)
10013: 325 individuals, phenotype mean 5.48173 (sd 1.19875)
10014: 360 individuals, phenotype mean 5.79835 (sd 1.02552)
10015: 369 individuals, phenotype mean 5.93227 (sd 0.973104)
10016: 386 individuals, phenotype mean 5.96707 (sd 0.959739)
10017: 372 individuals, phenotype mean 6.00515 (sd 0.953603)
10018: 371 individuals, phenotype mean 6.2132 (sd 0.813674)
10019: 364 individuals, phenotype mean 6.31304 (sd 0.698918)
10020: 380 individuals, phenotype mean 6.37598 (sd 0.604686)
10021: 427 individuals, phenotype mean 6.45539 (sd 0.471641)
10022: 370 individuals, phenotype mean 6.51224 (sd 0.363968)
10023: 412 individuals, phenotype mean 6.53184 (sd 0.305041)
10024: 415 individuals, phenotype mean 6.54332 (sd 0.277311)
10025: 404 individuals, phenotype mean 6.53433 (sd 0.299239)
10026: 380 individuals, phenotype mean 6.55845 (sd 0.234658)
10027: 386 individuals, phenotype mean 6.55116 (sd 0.256157)
10028: 421 individuals, phenotype mean 6.53684 (sd 0.293376)
10029: 390 individuals, phenotype mean 6.53791 (sd 0.29318)
10030: 401 individuals, phenotype mean 6.53193 (sd 0.307162)
10031: 403 individuals, phenotype mean 6.55492 (sd 0.248401)
10032: 418 individuals, phenotype mean 6.57826 (sd 0.165723)
10033: 428 individuals, phenotype mean 6.57869 (sd 0.163795)
10034: 399 individuals, phenotype mean 6.58121 (sd 0.151873)
10035: 420 individuals, phenotype mean 6.5856 (sd 0.128374)
10036: 402 individuals, phenotype mean 6.58132 (sd 0.15131)
10037: 422 individuals, phenotype mean 6.58565 (sd 0.128071)
10038: 419 individuals, phenotype mean 6.59647 (sd 0)

```

This is somewhat remarkable, since the new optimum is more than twelve standard deviations away from the population's phenotypic mean at the moment of the environmental change. The population fixes for a single QTL haplotype by the end (thus, a standard deviation of 0), and that haplotype provides a phenotype of 6.59647, which is almost exactly eight standard deviations away from where it started – quite impressive. So rescue appears to be possible from standing variation alone (sometimes), and from new mutations alone (sometimes), and most often from both together (but still, only sometimes).

These outcomes will depend – perhaps quite sensitively – on the various parameters of the model, such as the carrying capacity, the distance from the old optimum to the new one, the mutational distribution and rate, the “floor” of the fitness function, and the maximum fitness benefit from low population density. There are other implicit parameters here too, such as the level of individual fecundity and the variance in that fecundity (here, zero). This is also a hermaphroditic model, and hermaphroditic selfing is not prevented; switching to a sexual model would make evolutionary rescue that much more difficult, since a non-zero number of both males and females would need to be present in every tick. In short, gaining a proper understanding of the dynamics of even this rather simple model would require some real work, which is beyond the scope of this manual.

Nevertheless, the population dynamics of the model seem fairly realistic, and adding in even more realism – sex, Allee effects, gradual environmental change instead of a sudden shift, etc. – would not be difficult. Simulating this in a WF model would be more difficult to do with this level of realism, since the population size would have to be set explicitly in every tick (rather than being emergent from the birth/death dynamics), and would be fulfilled deterministically by SLiM rather than exhibiting the natural stochastic variation around the carrying capacity that this nonWF model exhibits.

Another interesting direction to take this model would be to use it to investigate the advantages of sexual versus clonal reproduction. It has long been theorized that one of the disadvantages of clonal reproduction is the difficulty of responding to environmental changes without the ability to recombine parental genomes to bring adaptive alleles together onto the same chromosome. One could experiment, in this model, with the effect of sexual versus clonal reproduction on evolutionary rescue – and even the evolution of the reproductive mode in response to environmental change. One could add a second QTL-based trait (see section 13.5) that governed the probability that an individual would clone or reproduce sexually, and see whether environmental change – perhaps cyclical or unpredictable – would provide enough of an advantage to sexual reproduction to prevent clonal reproduction from taking over the population. This would be straightforward to simulate in a nonWF model, since each individual generates its own offspring and can choose its own reproductive mode, based upon genetics or anything else. It would be considerably harder to implement in a WF model since the reproductive mode is controlled only by the subpopulation-wide cloning rate and cannot easily be influenced by individual genetics or other state.

## 16.8 Pollen flow

Plants reproduce sexually when a pollen grain from a flower reaches another (or perhaps the same) flower and fertilizes an ovule. The pollen might be transmitted by a pollinator, or by wind or water or other vectors. An important aspect of plant reproductive biology, then, is that pollen from a flower in one subpopulation might end up fertilizing a flower in a different subpopulation. In animals, gene flow between subpopulations generally results from the migration of individuals, such as we modeled in sections 16.5 and 16.6 (and in many WF recipes as well). In plants, in contrast, gene flow between subpopulations often results from the migration of gametes, especially in the angiosperms.

Pollen flow between subpopulations is a very important aspect of plant reproduction, then, but it is quite difficult to model with a WF model in SLiM since offspring in WF models always come from the mating of two individuals in the same subpopulation. Another advantage of nonWF models, then, is that they can easily simulate pollen flow, because sexual reproduction can involve any two individuals in the model.

This will be a very quick model since the concept is very simple and we have no complicated analysis to do with the results:

```
initialize() {
  initializeSLiMModelType("nonWF");
  defineConstant("K", 200);

  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;

  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
reproduction() {
  // determine how many ovules were fertilized, out of the total
  fertilizedOvules = rbinom(1, 30, 0.5);

  // determine the pollen source for each fertilized ovule
  other = (subpop == p1) ? p2 else p1;
  pollenSources = ifelse(runif(fertilizedOvules) < 0.99, subpop, other);

  // generate seeds from each fertilized ovule
  // the ovule belongs to individual, the pollen comes from source
  for (source in pollenSources)
    subpop.addCrossed(individual, source.sampleIndividuals(1));
}
1 early() {
  sim.addSubpop("p1", 10);
  sim.addSubpop("p2", 10);
}
early() {
  for (subpop in sim.subpopulations)
    subpop.fitnessScaling = K / subpop.individualCount;
}
10000 late() {
  sim.outputFixedMutations();
}
```

Most of this model is boilerplate that should be familiar by now. The interesting part is the `reproduction()` callback. Here we model hermaphroditic (or perhaps monoecious) flowering plants, so we do not model separate sexes, but we assume that selfing is no more common than would be expected by chance (when an individual happens to choose itself as a pollen source in this `reproduction()` code, which we do not prevent here). Each flower has 30 ovules, each with a probability of 0.5 of being fertilized, so the total number of fertilized ovules is drawn from a binomial distribution. We then determine the subpopulation that supplied the pollen for each of those ovules (assuming independence), with a 99% chance that the pollen came from the local subpopulation and a 1% chance that it was carried from the other subpopulation. Finally, we loop to generate seeds for the fertilized ovules, using the proper pollen sources.

This model doesn't show any particularly exciting behavior; it's just a two-subpopulation neutral model with a little gene flow. But it models pollen flow correctly, and thus provides a good foundation for building a more complex model of plant evolution.

Those interested in modeling plants might also wish to look at the recipe in section 11.3, which shows how to model gametophytic self-incompatibility. That recipe enforces self-incompatibility



with a `modifyChild()` callback, which ought to be compatible with this nonWF model. Note, however, that in nonWF models suppression of a proposed child by a `modifyChild()` callback is the end of that proposed child; in WF models, SLiM loops until the set subpopulation size is filled, but in nonWF models SLiM simply attempts to generate each requested offspring, and those that are rejected by a `modifyChild()` callback are abandoned. That behavior might be realistic and desirable (if pollen limitation is severe, or if stigmas are getting clogged by a large amount of incompatible pollen that prevents fertilization); if not, it is easy enough to make the `reproduction()` callback here loop until offspring generation succeeds. (The `addCrossed()` method will return `NULL` if the requested offspring is not generated.) Alternatively, one could implement the self-incompatibility system directly in the `reproduction()` callback, rather than in a `modifyChild()` callback, ensuring that the flower chosen as a pollen source for each fertilized ovule is compatible. The latter approach is perhaps simpler, and should be faster.

## 16.9 Litter size and parental investment

Litter size (clutch size, brood size) is often involved in evolutionary trade-offs. All else being equal, a larger litter is obviously better; the more offspring, the higher will be the fraction of one's own genes in the next generation. But all else is never equal, because each offspring requires an investment of some sort – at least the energy required to make the egg and sperm, and often quite a bit more beyond that. Offspring that receive insufficient parental investment will suffer lower fitness, and at some point the disadvantages of that, in higher offspring mortality and/or lower offspring mating success, will outweigh the advantages of having the extra offspring. In some species, particularly those in harsh and extreme environments, scraping together enough resources to produce even a single offspring is difficult, and the optimum may lie around one offspring per breeding season or even less; other species pursue a strategy of extremely low parental investment and produce as many offspring as they can. These different life history strategies can sometimes be simplified (or oversimplified) into “*K* strategists” and “*r* strategists”; more broadly, life-history tradeoffs are clearly of central importance in evolutionary biology.

In this recipe we will simulate a species with a quantitative trait that governs its litter size. Section 16.3's recipe included litter size variation, but here it will be governed by genetics, not just chance. We will also account for parental investment and the resulting impact of larger litter size on offspring fitness due to limited parental resources. The `initialize()` callback:

```
initialize() {
    initializeSLiMModelType("nonWF");
    initializeSex("A");
    defineConstant("K", 500);

    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;
    initializeMutationType("m2", 0.5, "n", 0.0, 0.3); // QTL

    initializeGenomicElementType("g1", c(m1,m2), c(1.0,0.1));
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);
}
```

This is a sexual nonWF model, with both neutral mutations (m1) and QTL mutations (m2). This initialization is probably rote by now (see chapter 13 for an introduction to QTL-based models).

Moving on to the reproduction() callback:

```
reproduction(NULL, "F") {
  mate = subpop.sampleIndividuals(1, sex="M");

  if (mate.size())
  {
    qtlValue = individual.tagF;
    expectedLitterSize = max(0.0, qtlValue + 3);
    litterSize = rpois(1, expectedLitterSize);
    penalty = 3.0 / litterSize;

    for (i in seqLen(litterSize))
    {
      offspring = subpop.addCrossed(individual, mate);
      offspring.setValue("penalty", rgamma(1, penalty, 20));
    }
  }
}
```

This callback applies only to females (the "F" in its declaration). The focal female chooses a male mate using `sampleIndividuals()`, and assuming that succeeds (i.e., there is at least one male in the population), the female will generate a litter with that mate. The script gets the female's litter-size phenotype from the `tagF` property where it will be stored (as we will see below) and derives an expected litter size from its value such that the new individuals at the beginning of the simulation, with no QTL mutations, have an expected litter size of 3. We then calculate the actual litter size by doing a Poisson draw with the expectation as mean, and calculate a fitness penalty for the offspring based upon the litter size they come from. The initial litter size of 3 entails no fitness penalty, but larger litter sizes will have correspondingly larger penalties because of the decreased amount of parental investment per offspring.

Having determined the litter size and the fitness penalty, we then make the litter's offspring in a loop. (Note the use of `seqLen(litterSize)`, which will produce the correct number of loops even if the litter size is zero; using the sequence operator with `1:litterSize` would be incorrect, since a litter size of 0 would generate the sequence `1 0` and the loop would then run twice instead of zero times. Be careful using the sequence operator in such cases!) Each offspring is generated with a call to `addCrossed()`, and then the fitness penalty due to parental underinvestment is set into a key named "penalty" on the offspring for later use. Note that the actual penalty for each individual is drawn from a gamma distribution with a mean of the expected penalty; this is a bit gratuitous, probably, but makes the actual penalty somewhat non-deterministic.

Next we create our initial population:

```
1 early() {
  sim.addSubpop("p1", 500);
  p1.individuals.setValue("penalty", 1.0);
}
```

We set the initial fitness penalty on the new subpopulation's individuals to 1.0 (i.e., no penalty, since this is a multiplicative fitness effect).

Now comes an `early()` event that, together with the `reproduction()` callback, really does the bulk of the work in this model:

```

early() {
  // non-overlapping generations
  inds = sim.subpopulations.individuals;
  inds[inds.age > 0].fitnessScaling = 0.0;
  inds = inds[inds.age == 0]; // focus on juveniles

  // QTL calculations
  inds.tagF = inds.sumOfMutationsOfType(m2);

  // parental investment fitness penalties
  inds.fitnessScaling = inds.getValue("penalty");

  // density-dependence for juveniles
  p1.fitnessScaling = K / size(inds);
}

```

This callback has four parts, as commented. The first part makes generations non-overlapping in this model, by setting the fitness of all individuals to 0.0 unless they are juveniles; of course the same life-history tradeoffs apply with overlapping generations too, but having discrete generations makes the model's operation easier to observe. Having set the fitness of the adults to 0.0, it then narrows the focus of the code to the juvenile population for the rest of the event by redefining the `inds` variable to only those individuals of age 0.

The second part totals the effects of all QTL (`m2`) mutations for juveniles and puts those totals into the `tagF` property of the juveniles; the `reproduction()` callback expects to find the total there, as we already saw, and the output code below uses it as well.

The third part fetches individual fitness penalty values using the "penalty" key and sets them into the `fitnessScaling` property of the individuals to create the desired fitness effect; note that a large "penalty" value is actually beneficial, since these values are fitness effects.

Finally, the fourth part implements density-dependence with the usual use of the `fitnessScaling` property of the subpopulation; here, however, we use only the number of juveniles in the calculation, to account for the fact that the adults are already marked for death and thus (we choose) should not participate in density-dependence effects.

Note that with this density-dependence formula the population size will generally be well under `K` at equilibrium – in the ballpark of 200 individuals – because the average individual will be subject to a substantial fitness penalty due to parental underinvestment. If we wanted the model to stay close to `K` regardless of the mean fitness penalty due to parental underinvestment, we could compensate for the mean penalty by scaling by the total individual fitness instead of the number of individuals, with code like this:

```

p1.fitnessScaling = K / sum(inds.fitnessScaling);

```

However, this would still not produce exactly the right result. In particular, some individuals come from a small litter will have "penalty" values well over 1.0, but that large positive fitness won't make a difference to their survival probability (because you can't survive more than 100% of the time), so the calculation here will overestimate the number of juveniles that are likely to survive, and the density-dependence will then be a bit too strong. Still, the formula above will only be substantially off when the mean litter size is small; once the model has equilibrated the mean population size will be close to `K`, given the parameters used here.

But in any case, it is perfectly reasonable – perhaps even *more* reasonable – for a large mean fitness penalty due to low parental investment to lead to a depressed mean population size; we have shown the alternative density-dependence calculation above just to illustrate the choices that

can go into deciding upon the implementation of density-dependence. We will stick to the first version of the density-dependence code for the “definitive” version of this recipe.

OK, since this is a QTL model we need to zero out the fitness effect of the QTL mutations as usual, so that their only fitness effects are indirect:

```
mutationEffect(m2) { return 1.0; }
```

Then we do a little output in each tick, and provide a termination event:

```
late() {  
  // output the phenotypic mean and pop size  
  qtlValues = p1.individuals.tagF;  
  expectedSizes = pmax(0.0, qtlValues + 3);  
  
  cat(sim.cycle + ": " + p1.individualCount + " individuals");  
  cat(", mean litter size " + mean(expectedSizes));  
  catn();  
}  
20000 late() { sim.simulationFinished(); }
```

The model starts off like this:

```
1: 500 individuals, mean litter size 3  
2: 430 individuals, mean litter size 3.00075  
3: 408 individuals, mean litter size 3.00007  
...
```

The mean litter sizes printed in the output are calculated in the same way as in the `reproduction()` callback. The model starts with the default litter size of 3, and then QTLs start to arise that modify that value. By the end of the model, we’re in a fairly different place:

```
19998: 182 individuals, mean litter size 8.18836  
19999: 174 individuals, mean litter size 8.21173  
20000: 183 individuals, mean litter size 8.20294
```

With these parameter values and configuration, the model tends to equilibrate at a litter size around 6.5 to 7.5, but the range of outcomes is fairly broad and values as high as 9 or 10 are often seen; apparently the selection on litter size imposed by this model is not terribly strong, so the fitness peak is pretty broad. In any case, somewhere in this vicinity there seems to be a crossover point where the benefit of more offspring is counterbalanced by the decrease in parental investment. In this simple model, that result could probably be calculated analytically, but of course that would be impossible in a more complex model involving other biological realism as well.

The population size at the end is much smaller than it was at the beginning. This is because every individual at the end is suffering from a lack of parental investment, and that depresses the population size below the carrying capacity, as discussed above. We don’t even need to think about that; it just happens automatically, as an emergent property of the model. If the optimal litter size were sufficiently large (with sufficiently low parental investment as a result), evolution toward the optimum litter size could probably drive the population extinct.

In this recipe, the penalty for each offspring depends upon the size of the litter to which the offspring belonged. This makes sense when parental investment is, in fact, per offspring, such as feeding newly hatched juveniles in a nest. In other cases, investment might depend upon the expected litter size, not the actual litter size; if a bird adds body fat before the breeding season and uses that energy to generate a predetermined number of eggs, but only a subset of those eggs

hatch, the investment is per egg, not per hatched chick. We can modify the recipe for that case by changing the penalty calculation to be:

```
penalty = 3.0 / expectedLitterSize;
```

Interestingly, even though `litterSize` is drawn from a distribution with a mean of `expectedLitterSize`, this produces fairly different outcomes. The equilibrium litter size reached is now typically around 3.5 to 4.0 – much smaller.

This model has a lesson that goes far beyond litter size and parental investment: nonWF models can include genetic variation, and evolution, of traits that it is difficult or impossible to model in such a way in WF models. One could easily write a nonWF model of the evolution of the sex ratio, or of the selfing or cloning rate, or of migration or dispersal behavior, or of – as here – litter size or parental investment. WF models can include all of those phenomena, but since they are handled by SLiM's core engine in WF models, it is quite difficult to include genetically-based individual variation in them.

## 16.10 Spatial competition and spatial mate choice in a nonWF model

In chapter 15 a variety of spatial models were explored, all of which were WF models. Those spatial modeling techniques work just as well in nonWF models – indeed, better in some ways, as we will see. The model here is derived from the recipe of section 15.5, and includes both spatial competition and spatial mate choice.

Let's begin with the `initialize()` callback as usual:

```
initialize() {
    initializeSLiMModelType("nonWF");
    initializeSLiMOptions(dimensionality="xy", periodicity="xy");
    defineConstant("K", 300); // carrying capacity
    defineConstant("S", 0.1); // spatial competition distance

    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;

    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);

    // spatial competition
    initializeInteractionType(1, "xy", reciprocal=T, maxDistance=S);

    // spatial mate choice
    initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.1);
}
```

We set up "xy" dimensionality with periodic boundary conditions for both dimensions, in order to get a toroidal space (see section 15.12). We will still need density-dependent population regulation of some kind, but since this is a continuous-space model our concept of density ought to depend upon *local* density, not overall population size; K is the carrying capacity that we will aim for by calibrating our local density function. We also define the maximum spatial competition distance with the symbol S. We set up a neutral model with the usual boilerplate code, and initialize spatial interactions for competition and mate choice.

Next let's look at reproduction:

```

2: first() {
    // look for mates
    i2.evaluate(p1);
}
reproduction() {
    // choose our nearest neighbor as a mate, within the max distance
    mate = i2.nearestNeighbors(individual, 1);

    for (i in seqLen(rpois(1, 0.1)))
    {
        if (mate.size())
            offspring = subpop.addCrossed(individual, mate);
        else
            offspring = subpop.addSelfed(individual);

        // set offspring position
        pos = individual.spatialPosition + rnorm(2, 0, 0.02);
        offspring.setSpatialPosition(p1.pointPeriodic(pos));
    }
}

```

In a `first()` event, before `reproduction()` callbacks are called, we evaluate interaction `i2`; we will use it to find a mate within the maximum mating distance. We haven't seen `first()` events much, but they are a convenient place to prepare for reproduction – particularly in nonWF models, since `early()` events run after reproduction has completed.

If a mate is found, `addCrossed()` is used for biparental mating, otherwise `addSelfed()` is used for selfing (note this variation in individual mating behavior based on spatial dynamics, which would be quite difficult to achieve in a WF model). In either case, we draw a litter size using `rpois()` with an expected mean size of `0.1`, so individuals will reproduce relatively infrequently and generations will be highly overlapping; if we're at equilibrium and a new individual is born from a given parent 10% of the time, then individuals ought to have an average lifespan of `10` ticks. We loop over our litter size (note the use of `seqLen()`, so that if the litter size is zero we get the correct behavior; using `1:rpois()` instead would yield the sequence `1 0` when the litter size is zero, erroneously producing two offspring). Each offspring is positioned a small distance from the first parent, with accounting for the periodic boundaries.

Population initialization is very similar to the WF model:

```

1 early() {
    sim.addSubpop("p1", 1);

    // random initial positions
    p1.individuals.setSpatialPosition(p1.pointUniform(1));
}

```

Note that because individuals that can't find mates turn to selfing, we can start the model with just a single individual and let it grow to capacity. Next we have spatial competition:

```

early() {
    i1.evaluate(p1);

    // spatial competition provides density-dependent selection
    inds = p1.individuals;
    competition = i1.totalOfNeighborStrengths(inds);
    competition = (competition + 1) / (PI * S^2);
    inds.fitnessScaling = K / competition;
}

```



We evaluate `i1`, and then evaluate the effect of competition for all individuals in a vectorized fashion. The call to `totalOfNeighborStrengths()` returns a vector of the interaction strength felt by each individual, and we rescale these values to normalize them (discussed below). The fitness effect on each individual is then calculated as the carrying capacity  $K$  divided by the rescaled strength of competition felt by the individual. If that strength is equal to  $K$ , we are at equilibrium; the individual will feel no fitness effect, positive or negative, from spatial competition. If the local population density around the individual is higher than that equilibrium density, its fitness will be lower, and vice versa; *local* density is what is actually enforced by this formula, not total population size. If the population happens to be uniformly distributed in space, then its equilibrium size will be equal to  $K$ ; but if there are areas of space that are uninhabitable, or if individuals tend to cluster for any reason, then the equilibrium population size may differ from  $K$ .

To understand how the rescaling works, let's look at this in more detail. The maximum competition distance,  $S$ , is set to  $0.1$  here, which means that the circle covered by the interaction radius around a focal individual has a radius of  $0.1$ . The term  $(\pi * S^2)$  is the area of this interaction circle ( $\pi r^2$ ), so in this case it would be  $0.01\pi$  or about  $0.03141593$ . The focal individual's interaction circle would then, on average, include about 3.1% of the individuals in the whole model (since the area of the space defined by `p1` is  $1.0$ , since we didn't change its dimensions from a unit square; if that did change, the calculations here would have to be tweaked). The initial value of `competition` will be the population size inside that interaction circle, minus one because the focal individual is not itself included in the total interaction strength; we compensate by using  $(\text{competition} + 1)$ . For our situation in which the interaction circle has an area of  $\sim 0.031$  and contains, on average, 3.1% of the population, it can thus be seen that the fitness scaling value for every individual will be approximately  $1.0$  when the local population density is at carrying capacity. The same logic applies for other values of  $S$ . Note that this is only an argument about *local* population density; a non-uniform spatial distribution might cause some regions to experience a fitness greater than or less than  $1.0$  even when the population size as a whole is exactly at carrying capacity. That is a *good* thing, and biologically realistic, as we will see.

Finally, note that periodic boundary conditions can be important when modeling competition in this sort of way, because they provide a uniform strength of competition across space, without edge effects. With a non-periodic boundary condition, the strength of spatial competition felt by individuals at the edge of the space would be lower, and that would encourage the population density to be higher at the edges than in the center (because, in effect, the local population density being regulated by the competition function includes the empty areas beyond the edges of the space). We will see a way to compensate for this undesirable edge effect without the use of periodic boundaries in the next section, but for now we will use periodic boundaries to simplify the situation.

So far so good. Since we have overlapping generations, it would be nice to model some movement by the individuals in the model, rather than just representing them as motionless points. To do that, we have a `late()` event that moves everybody around:

```
late()
{
    // move around a bit
    for (ind in p1.individuals)
    {
        newPos = ind.spatialPosition + runif(2, -0.01, 0.01);
        ind.setSpatialPosition(p1.pointPeriodic(newPos));
    }
}
```

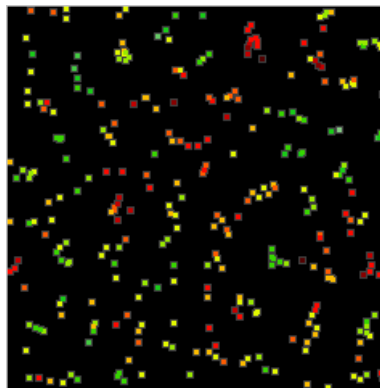
This just deviates individual positions by a small random factor, with accounting for periodic boundaries using `pointPeriodic()` (see section 15.12 for discussion of periodic boundaries).

Finally, we have a termination event:

```
10000 late() {  
    sim.outputFixedMutations();  
}
```

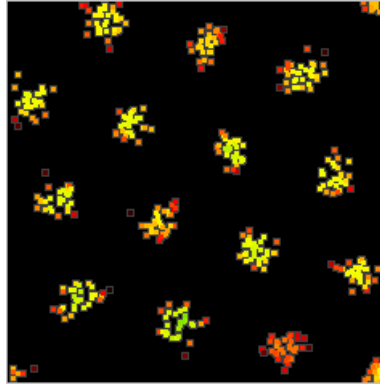
Something worth noting about this model is that it contains no `modifyChild()` callback. Because nonWF models create their own offspring, we can fix the spatial positions of offspring directly in the `reproduction()` callback. It would also work to do it in a separate `modifyChild()` callback, as in section 15.5, but that would be a bit more complex and a bit slower. Often, nonWF models can get away with fewer callbacks, or even no callbacks except `initialize()` and `reproduction()`, as shown here.

When this recipe is run, it looks a lot like the recipe from section 15.5:



There are differences from section 15.5's recipe, though: individuals are moving around in space during their lifetimes, and generations are clearly overlapping; there is much more continuity from tick to tick. The population size is no longer fixed at  $K$ , either; it starts at 1 and grows organically, with a natural pattern of population growth that spreads across space. Even once it reaches carrying capacity it is not fixed at exactly  $K$ , as the WF model was, but fluctuates around the carrying capacity stochastically, depending in part upon how non-uniform the spatial clustering at any given moment happens to be. This added realism could be very important in some models. For example, if something were to suddenly perturb the population – if a disease outbreak in one corner of the space suddenly killed off all the individuals within a small radius, say – the WF model would nevertheless force the total population size to be  $K$  in every tick, and so the population density would increase in all the other areas of the space, which clearly makes no sense. This model's more robust implementation of population regulation based upon local population density prevents that aberration; the population size would initially be depressed by the perturbation, and the hole created by the disease outbreak would fill back in from its edges naturally, over time.

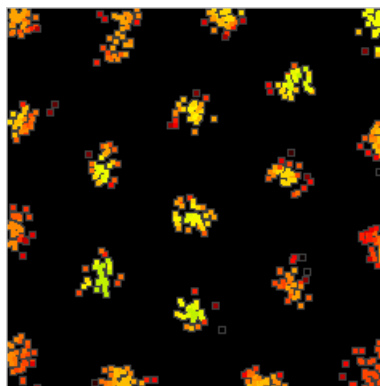
It has been mentioned several times now that spatial clustering might cause this model to reach an equilibrium population size different from  $K$ . In the snapshot above, it is not immediately obvious that this is happening, but in fact it is, and the population size of this recipe fluctuates around roughly 310, slightly above the intended carrying capacity of 300. This can be made much more obvious by increasing  $S$  to 0.2; the model at equilibrium then looks like this:



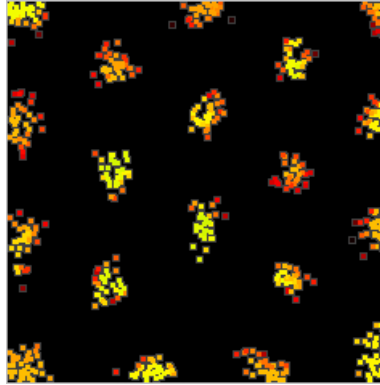
Rather remarkably, the population has naturally clustered itself into fifteen clumps, equidistantly arranged across the periodic space in a hexagonal pattern; this seems to be the optimal arrangement for this value of  $S$ , and the model finds it fairly quickly every time it is run. The equilibrium population size is now about 500 individuals. In the next section, we will discuss this phenomenon further, and look at a way of preventing it if it is undesirable.

For now, let's just note that this is not a bug; this is the emergent behavior of the model, and it is not necessarily unbiological. For one thing, some species are territorial, and what this model is doing could be seen as a sort of territoriality; if the carrying capacity and other parameters were adjusted appropriately, each cluster could contain the appropriate number of individuals for family groups of the modeled species. For another, complex spatial clustering has been observed in various natural systems; Vincenot et al. (2016) provide an overview of some examples in plants, for instance, and discuss the modeling of such clustering. Of course these clusters might not behave precisely as one might wish; they are probably almost completely genetically isolated from each other, for one thing, so one might wish to model dispersal that would provide gene flow (such as the way that young male lions leave their natal pride and head out to form their own pride). A sufficiently large dispersal kernel would disrupt the clustering seen here enough to produce gene flow; different dispersal kernel shapes (exponential, Cauchy) might also have interesting effects.

There is one more interesting aside to be explored here. We can modify this recipe a bit more, by removing the periodic boundaries and instead implementing reprising boundaries following the appropriate recipe from section 15.3. With  $S$  still set to 0.2, if we run this modified model we will see a fairly different pattern of spatial clustering:



Twelve clusters are now arranged around the edges, and another eight clusters are packed into the interior in an asymmetrical pattern. This is one way that the model can satisfy the constraints it has been given, but it is not entirely stable, and there are several other configurations with either nineteen or twenty clusters that are commonly observed. Here is another, with nineteen:



The equilibrium population size here is about 680 in the top configuration, and about 660 in the bottom configuration. This is higher than it was with periodic boundaries, and of course the nineteen or twenty clusters observed now is more than the fifteen observed with periodic boundaries too. This difference occurs because reprising boundaries provide the model with more elbow room; the clusters can now take advantage of the fact that they feel no competition from the empty space beyond the edges, whereas periodic boundaries did not afford this luxury.

The overall point, then, is that the dynamics of spatial models are considerably more subtle than one might initially appreciate, and that choices such as boundary conditions can have large consequences for dynamics that might not be immediately obvious. In the next section we shall explore this point.

### 16.11 A spatial model with carrying-capacity density

The previous section provided a recipe for a nonWF model with both spatial competition and spatial mate choice. Spatial competition provided population regulation, since more individuals would produce more competition, reducing absolute fitness. However, as we saw at the end of the section, that model tends to produce spatial clustering that might often be undesirable. This occurs because of the shape of the competition function. The competitive interaction strength felt between two individuals in that recipe was constant out to the maximum interaction distance, and then fell off abruptly to zero. This produces an incentive for clustering, for two reasons: (1) being tightly clustered implies no more fitness penalty than being loosely clustered, since the interaction strength is constant, and (2) clustering allows the empty spaces between clusters to be shared by the clusters; if the clusters arrange themselves into a regularly spaced configuration, they can efficiently share the empty spaces between them. That effect allows the mean interaction strength felt by individuals to decrease, in turn allowing the model to exceed its nominal carrying capacity.

For this reason (and others), a Gaussian competition kernel is often preferred in this sort of spatial model; it seems to produce fewer artifacts of this type. In chapter 15 we constructed a series of models inspired by classic papers such as Dieckmann & Doebeli (1999) and Doebeli & Dieckmann (2003), and here we return to that thread. Doebeli & Dieckmann (2003) introduced a concept that they called “carrying-capacity density”, based upon a Gaussian competition kernel; with the proper scaling, as set out in their paper, the population will equilibrate at the intended carrying capacity if the environment is homogeneous, just as it did in the previous recipe, but clustering artifacts driven by the shape of the competition kernel will be minimized. Here, we will adjust the model of section 16.10 to implement this concept of carrying-capacity density. We will make small changes and slowly evolve the section 16.10 recipe, one step at a time (apologies if this is confusing; the full, final recipe is available in SLiMgui and online).

One part that requires changes is the `initialize()` callback:

```

initialize() {
    initializeSLiMModelType("nonWF");
    initializeSLiMOptions(dimensionality="xy", periodicity="xy");
    defineConstant("K", 300); // carrying-capacity density
    defineConstant("S", 0.1); // sigma_S, the spatial competition width

    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;

    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);

    // spatial competition
    initializeInteractionType(1, "xy", reciprocal=T, maxDistance=S * 3);
    i1.setInteractionFunction("n", 1.0, S);

    // spatial mate choice
    initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.1);
}

```

We now call  $K$  the carrying-capacity density, and  $S$  the spatial competition width, which is referred to with the symbol  $\sigma_s$  in Doebeli & Dieckmann (2003). The spatial competition function now uses a Gaussian kernel (type "n"), with a maximum distance of three standard deviations (so by the time it cuts off it should be very close to zero anyway). Otherwise this `initialize()` callback is the same as in section 16.10.

Another element that changes is the population regulation event:

```

early() {
    i1.evaluate(p1);

    // spatial competition provides density-dependent selection
    inds = p1.individuals;
    competition = i1.totalOfNeighborStrengths(inds);
    competition = (competition + 1) / (2 * PI * S^2);
    inds.fitnessScaling = K / competition;
}

```

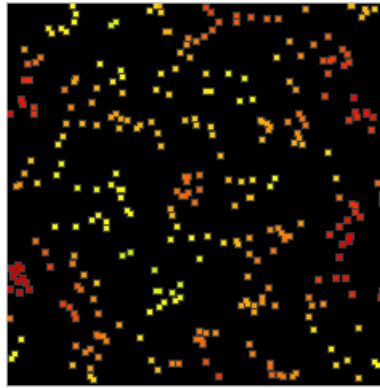
This is similar to section 16.10's code, but the rescaling of the competition strength has changed in accord with the new Gaussian competition kernel shape. The new rescaling is parallel to the rescaling that is applied in the standard formula for the normal distribution, usually written as:

$$f(x | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

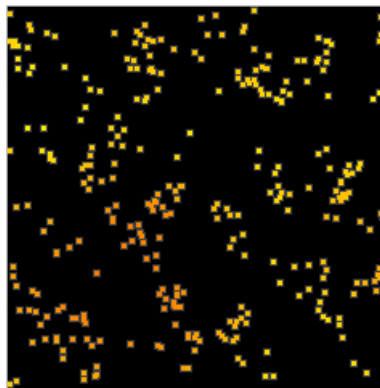
Without the rescaling constant, that formula would produce a maximum value of 1.0; rescaled, it produces a lower maximum density, such that the total probability density – the integral under the curve – is exactly 1.0 (as required by the definition of a probability density function). Now, note that the Gaussian (type "n") interaction function we're using here utilizes the same formula, but without that rescaling factor; the correction factor in the code really just introduces that rescaling factor, in order to normalize the Gaussian interaction function. This is parallel to the reasoning followed in section 16.10 that led us to rescale using the formula for the area of a circle of radius  $S$ . The rescaling factor is squared in this recipe (because there is no square root) because we have two spatial dimensions; the spatial competition function is really the product of two

Gaussian functions, one for  $x$  and one for  $y$ . See Doebeli & Dieckmann (2003) and related papers for further discussion, since a full derivation is beyond the scope of this manual. Note that, in terms of the implementation, we could just rescale the maximum strength for `i1` instead, supplying the same rescaling constant directly to `setInteractionFunction()`, and that would run faster, too; the design shown here is more explicit for pedagogical purposes.

When this script is run, the spatial distribution looks a lot like the recipe from section 15.10:



Close examination, however, shows that this spatial distribution is less regularly clustered than that previous recipe's distribution. Changing `S` to `0.2` shows that the clustering previously observed at that interaction scale is all but gone too:



A little bit of clustering still occurs, partly because offspring land near their first parent, and partly just as a result of stochasticity (one would not expect a stochastic process to produce a perfectly uniform spatial distribution, after all). But the competition function is no longer contributing heavily to that clustering as it was before; indeed, it may be working to smooth it away by favoring new offspring that land in relatively unoccupied areas.

In the section 16.10 we briefly tried replacing the periodic boundaries of the model with reprising boundaries, and found that it strongly promoted clusters around the edge of the space due to an “edge effect”: the uninhabited area beyond the edge of the spatial bounds artificially lowered their local density. Let's try that again now that we have replaced the fixed interaction function with the Gaussian kernel; and this time, let's actually switch the recipe over to using reprising boundaries. To do that we need to replace the `initializeSLiMOptions()` call:

```
initializeSLiMOptions(dimensionality="xy");
```

and replace the new offspring positioning in the `reproduction()` callback:

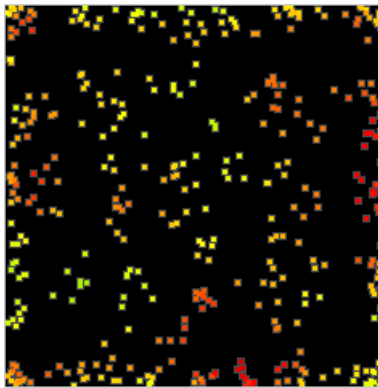


```
// set offspring position
do pos = individual.spatialPosition + rnorm(2, 0, 0.02);
while (!p1.pointInBounds(pos));
offspring.setSpatialPosition(pos);
```

and finally, replace the movement code in the late() event:

```
// move around a bit
for (ind in p1.individuals)
{
    do newPos = ind.spatialPosition + runif(2, -0.01, 0.01);
    while (!p1.pointInBounds(newPos));
    ind.setSpatialPosition(newPos);
}
```

If we run this new model in SLiMgui, here's the result:



The edge effect remains; we are no longer seeing clusters around the edge, but we still see a border area with higher-than-average density, and within that a ring of lower-than-average density, because the edge effect promotes survival at the edge and the resulting high density at the edge decreases survival just inside that border region.

We are not quite done, then. The recipe thus far revealed the inner workings of how carrying-capacity density is calculated, but we would very much like to get rid of this edge effect, since periodic boundaries are not always desirable. To correct for this edge effect, we would like to remove the area beyond the edge of the spatial bounds from the area considered “habitable” when we calculate local density. Happily, there is a simple way to do this (as of SLiM 3.7).

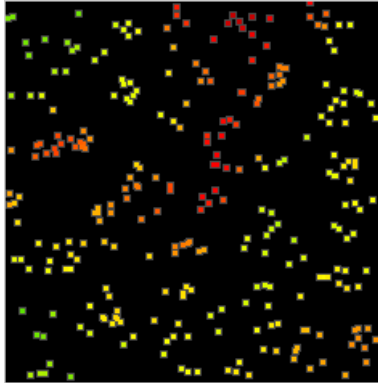
Recall that the competition strength calculations in the early() event now look like this:

```
// spatial competition provides density-dependent selection
inds = p1.individuals;
competition = i1.totalOfNeighborStrengths(inds);
competition = (competition + 1) / (2 * PI * S^2);
inds.fitnessScaling = K / competition;
```

We will now replace that code with this new script:

```
// spatial competition provides density-dependent selection
inds = p1.individuals;
competition = i1.localPopulationDensity(inds);
inds.fitnessScaling = K / competition;
```

Before discussing this change in detail, let's just look at what our model – which is now the final recipe for this section – looks like in SLiMgui:



We have gotten rid of the edge effect, even though we no longer have periodic boundaries. This is therefore a complete spatial nonWF model using carrying-capacity density for local population regulation without edge effects. But how does the code above work?

Essentially, the `localPopulationDensity()` method replaces the logic we had before and improves upon it. Internally, it does the exact same calculation that `totalOfNeighborStrengths()` does, tallying up the interaction strengths of all neighbors of each individual within the maximum interaction distance. It also does the same thing that the `+ 1` in the previous version achieved: it adds in the focal individual itself, since local density ought to include the individual itself. (Note that this was `+ 1` because the maximum value of the interaction functions used has been `1.0`, but in general the maximum interaction function value should be added in to the total.) Finally, it does the same thing as the division by  $(2 * \pi * S^2)$  did: it divides by the integral of the interaction function to get local population density. However, `localPopulationDensity()` is very smart about this: it actually clips that integral to both the maximum interaction distance and the edges of the spatial bounds, tailoring the integral of the interaction function to each individual's local circumstances so that the total interaction strength is divided by the correct estimate of the "interaction field" present, correcting for the edge effect. We are done!

Incidentally, these effects of interaction kernel shape that we have been looking at have been explored in various papers; see Payne et al. (2011) for a model of this based upon the same Dieckmann and Doebeli models that we have been exploring, although that paper concerns itself more with clustering in phenotypic space than in the regular spatial dimensions (as we saw in chapter 15, phenotype can in some ways be treated similarly to a spatial dimension). Payne et al. (2011) found that a box-shaped (i.e., platykurtic) kernel encouraged clustering, just as did the cut-off kernel we used in section 16.10, which was of course also platykurtic. Leimar et al. (2008) explored competition kernel shape in more detail, and found that a characteristic of the Fourier transform of the competition kernel was predictive of its effects upon clustering. A Gaussian competition kernel does not promote clustering, according to their findings, and this is one reason why this kernel shape is popular (although mathematical convenience also plays a role, as they explain).

## 16.12 Forcing a specific pedigree in a nonWF model

It is sometimes desirable to force a SLiM model to follow a specific pedigree through arranged matings between individuals. This is quite easy to do with a nonWF model, since in nonWF models matings are arranged by the model's script, not by SLiM's core engine. We will use `tag` values to identify each individual, with a unique `tag` value for every individual.

We will actually look at two recipes here; the first tracks the pedigree realized by a model run and produces output files recording that pedigree, while the second reads those output files and exactly reproduces the pedigree.

Let's begin with the first recipe:

```
initialize() {
  initializeSLiMModelType("nonWF");
  defineConstant("K", 10);

  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);

  // delete any existing pedigree log files
  deleteFile("mating.txt");
  deleteFile("death.txt");
}
reproduction() {
  // choose a mate and generate an offspring
  mate = subpop.sampleIndividuals(1);
  child = subpop.addCrossed(individual, mate);
  child.tag = sim.tag;
  sim.tag = sim.tag + 1;

  // log the mating
  line = paste(community.tick, individual.tag, mate.tag, child.tag);
  writeFile("mating.txt", line, append=T);
}
1 early() {
  sim.addSubpop("p1", 10);

  // provide initial tags and remember the next tag value
  p1.individuals.tag = 1:10;
  sim.tag = 11;
}
early() {
  // density-dependence
  p1.fitnessScaling = K / p1.individualCount;
}
survival() {
  if (!surviving) {
    // log the death
    line = community.tick + " " + individual.tag;
    writeFile("death.txt", line, append=T);
  }
  return NULL;
}
100 late() { sim.simulationFinished(); }
```

This recipe is quite straightforward; in most respects it follows the typical skeleton of a nonWF model, but with some logging code added in. In the `initialize()` callback we delete any pre-existing log files. The `reproduction()` callback chooses a random mate and calls `addCrossed()` to generate a child, as usual; it also sets that child up with a unique identifying tag value, and appends a line summarizing the mating event to the `mating.txt` log file. The `1 early()` event sets up the subpopulation as usual, and also sets up initial tag values for the new individuals; the value of `sim.tag` is used to track the next unused tag value in the run. The `early()` event provides

density-dependent population regulation, as we have seen in previous nonWF models. Finally, a `survival()` callback appends death information to the `death.txt` log file, as discussed next. The model ends at the end of tick 100.

We haven't seen `survival()` callbacks before; these callbacks, which are available only in nonWF models, allow the model to override SLiM's default decisions during the nonWF tick cycle's viability/survival selection stage, regarding which individuals will survive and which will die (see section 25.10). Normally, SLiM's decisions are based on fitness; fitness in nonWF models is interpreted as a probability of survival, as we have seen, and for each individual SLiM draws a random uniform deviate from [0,1] and compares it to the individual's fitness to see whether it lives or dies. With a `survival()` callback, called for each individual in each tick, the model's script can override this decision in order to force a focal individual to live (by returning T) or to die (by returning F). This recipe does not intervene in survival decisions in this way, however; it simply observes them, logging out the tick and individual tag whenever an individual is about to die. It returns NULL, indicating that SLiM should follow its default course of action. (See sections 16.15, 16.19, and 16.22 for more active uses of `survival()` callbacks).

A run of this model produces `death.txt` and `mating.txt` files on the user's desktop (on macOS, at least; the output paths in the recipe may need to be modified on other platforms). The `death.txt` file is simply a series of lines, each of which has a tick and then the tag value of an individual that died in that tick, like this:

```
2 2
2 4
2 6
...
3 7
3 19
...
4 20
4 21
...
```

This file records that in tick 2 the individuals with tag values 2, 4, 6, etc., died; in tick 3, individuals with tag values 7, 19, etc.; in tick 4, individuals with tag values 20, 21, etc.; and so forth, through to the end of the run.

The `mating.txt` file is almost as simple; here, each line records a tick, the tag values of the first and second parent, and then the tag value of the generated offspring:

```
2 1 3 11
2 2 10 12
2 3 2 13
...
3 1 13 21
3 3 20 22
...
4 1 13 30
4 3 22 31
...
```

The first line in this example, for example, specifies that in tick 2 the individuals with tag values 1 and 3 mated to produce an offspring individual with tag value 11.

These files, then, provide all the information we need to reproduce the full pedigree and population history of the model run. So much for the first recipe; now let's move on to the second recipe and see how it uses these files to force a replication of the logged pedigree:

```

function (+)readMatrix(s$ path, [string$ sep = ","])
{
  if (!fileExists(path))
    stop("readMatrix(): File not found at path " + path);
  df = readCSV(path, colNames=F, sep=sep);
  m = df.asMatrix();
  return m;
}
initialize() {
  initializeSLiMModelType("nonWF");
  defineConstant("K", 10);

  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);

  // read in the pedigree log files
  defineConstant("M", readMatrix("mating.txt", sep=""));
  defineConstant("D", readMatrix("death.txt", sep=""));

  // extract the ticks for quick lookup
  defineConstant("Mt", drop(M[,0]));
  defineConstant("Dt", drop(D[,0]));
}
reproduction() {
  // generate all offspring for the tick
  m = M[Mt == community.tick,];

  for (index in seqLen(nrow(m))) {
    row = m[index,];
    ind = subpop.subsetIndividuals(tag=row[,1]);
    mate = subpop.subsetIndividuals(tag=row[,2]);
    child = subpop.addCrossed(ind, mate);
    child.tag = row[,3];
  }

  self.active = 0;
}
1 early() {
  sim.addSubpop("p1", 10);

  // provide initial tags matching the original model
  p1.individuals.tag = 1:10;
}
early() {
  // execute the predetermined mortality
  inds = p1.individuals;
  inds.fitnessScaling = 1.0;

  d = drop(D[Dt == community.tick, 1]);
  indices = match(d, inds.tag);
  inds[indices].fitnessScaling = 0.0;
}
100 late() { sim.simulationFinished(); }

```

Let's walk through how it works. First of all, we define a new function named `readMatrix()` that reads in a text file from a given filesystem path, assumed to be composed of lines of values, and returns a matrix representing the contents of the file. (Note that this function is in the online SLiM-Extras repository, too.) We haven't used matrices much in previous recipes; they are a relatively new feature of Eidos, and work in much the same way as matrices in R do (see the Eidos manual for discussion). Indeed, we haven't seen many examples of user-defined functions either (again, see the Eidos manual). Without worrying too much about such details, however, if we treat this function as a black box and call it with the path of our `death.txt` file, we get this matrix:

```

      [,0] [,1]
[0,]    2    2
[1,]    2    4
[2,]    2    6
...

```

And for our `mating.txt` file we get this:

```

      [,0] [,1] [,2] [,3]
[0,]    2    1    3   11
[1,]    2    2   10   12
[2,]    2    3    2   13
...

```

The `initialize()` callback of this recipe sets up the model in the usual way (mirroring the setup of the first recipe, which is what one would probably want in most cases). It then calls `readIntTable()` to read in the log files, and retains the resulting matrices as defined constants, `M` and `D`. Finally, it extracts the first column from `M` and `D` (which contains the ticks for each logged event) and retains those as constants `Mg` and `Dg`, for simplicity and speed later. So far so good; this is the information the recipe will use to reproduce the logged pedigree.

Next, the `reproduction()` callback executes the pedigree's mating events. It does this by fetching the rows of `M` that refer to the current tick, and looping through those rows to generate each mating event (again, you may wish to refer to the Eidos manual for information on the syntax involved in working with matrices). The `tag` value of the offspring is set according to the logged pedigree as well, so that the new individual will match up with the `tag` values used in the log files. The callback triggers all of the mating events for the tick in a single call, rather than working with the individual supplied to the `reproduction()` callback, so it then disables itself, by setting `self.active` to `0`, ensuring that it is not called again in the current tick (as we saw before in section 16.3).

The `1 early()` event creates the initial subpopulation, as in the first recipe, and sets the `tag` values of individuals in the same way so that both models get the same setup.

Finally, we have the `early()` event. This does not cause density-dependent population regulation in the usual way of nonWF models; instead, it uses the `fitnessScaling` property values of individuals to weed out specifically the individuals that died in each tick in the original model run. It begins by setting `fitnessScaling` to `1.0` on all individuals. Then it looks up the mortality events for the current tick, similarly to how the `reproduction()` callback looked up mating events. It then uses `match()` to find the indices of the individuals in question, and sets `fitnessScaling` to `0.0` for those individuals. Those individuals will be killed by SLiM during the viability/survival tick cycle stage that follows the `early()` event. The model terminates at the end of tick `100`, matching the behavior of the original model.

In this way, this recipe reproduces the saved pedigree, even when a different random number seed is used. If you run the second recipe repeatedly, you will therefore get replicates of the saved pedigree, but with different mutational and recombinational histories and thus different segregating



mutations at the end of the runs. Of course you may not wish to take on faith that the pedigree is replicated exactly, so if you wish you can add calls to `catn()` in the appropriate spots to log out each mating and mortality event, to confirm that they follow the pedigree as intended.

This basic scheme could be extended in various ways, as needed. For example, the log files presently support only biparental matings; if you wanted to force a pedigree that involved cloning in some cases, you could extend the `mating.txt` file format to have an extra column with a value indicating whether the offspring was generated by cloning or not, or you could add a third log file, named something like `cloning.txt`, to record those events; either strategy would work. You could also log out, and then reproduce, other model events if you wished, such as migration events; since you are in complete control of such events in nonWF models, doing this would be quite a straightforward extension of these recipes. You could probably even record, and then reproduce, the recombination breakpoints used in the generation of each offspring individual, using a `recombination()` callback, so as to make the replicate run duplicate exactly the same pattern of local ancestry along the chromosome as the original model run, if you wanted to. This strategy, of driving model dynamics from file-based data, is quite general and can be applied to many tasks.

The second recipe can of course be used on its own, to make SLiM follow any pedigree specified by its input files; that pedigree information could come from empirical data rather than from recording a previous SLiM model run as we have done here.

### 16.13 Modeling clonal haploids in a nonWF model with `addRecombinant()`

In section 14.9 we saw a recipe for modeling clonal haploids in SLiM by keeping the second genome of each individual empty. New mutations arising in those second genomes were removed in each tick, and a recombination rate of zero was used to prevent any recombination with those empty genomes. That recipe is general and flexible; however, it does have some drawbacks, particularly when tree-sequence recording is being used (see section 1.7 and chapter 17). With tree-sequence recording, the recipe of section 14.9 records rather odd dynamics, with mutations appearing and then disappearing in each generation, and the empty second genomes of individuals inheriting from each other. These bizarre inheritance records are probably harmless, but may complicate post-simulation analysis, and in any case constitute something of an offense against clean design. Even more problematic for section 14.9's strategy, when doing tree-sequence recording, is the question of how to model horizontal gene transfer, as we discuss in section 16.14.

Here, then, we will look at an alternative strategy for modeling clonal haploids. This strategy can be employed only in nonWF models, but integrates much more smoothly with tree-sequence recording and is conceptually more straightforward as well. The recipe, in full:

```
initialize() {
    initializeSLiMModelType("nonWF");
    defineConstant("K", 500); // carrying capacity
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    m1.convertToSubstitution = T;
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);
}
reproduction() {
    subpop.addRecombinant(individual.genome1, NULL, NULL, NULL, NULL, NULL);
}
1 early() {
    sim.addSubpop("p1", 500, haploid=T);
}
```

```

early() {
    p1.fitnessScaling = K / p1.individualCount;
}
50000 late() { sim.outputFixedMutations(); }

```

This mostly has the typical nonWF setup and population regulation, as we have seen before. It sets `m1.convertToSubstitution` to `T`; SLiM will be able to automatically detect and remove fixed mutations for us, unlike in section 14.9. The one notable difference from typical nonWF setup is that we pass `haploid=T` to `addSubpop()`. This tells SLiM that the 500 individuals created for `p1` should be true haploids, with the second genome of each individual being a null genome, not just an empty genome. Null genomes (see section 1.5.1) are essentially placeholders used by SLiM to represent the conceptual absence of a genome; they are used when modeling sex chromosomes (see section 6.2.3), and they can also be used when modeling haploids in nonWF models. So we start with 500 individuals that are haploid, with null second genomes. Any attempt to modify or manipulate these null genomes in SLiM will produce an error, for code safety.

The most interesting and new part of this model is the `reproduction()` callback, which uses a method we have not seen before, `addRecombinant()`. Like `addCrossed()`, `addCloned()`, `addSelfed()`, and `addEmpty()`, this adds a new offspring individual to the target subpopulation, but it provides greater control over precisely how that new individual is generated. Section 24.15.2 provides full documentation on this complex method, but in short, it allows you to supply two parent genomes for each genome in the new offspring, with a list of recombination breakpoints to be used in stitching together those parent genomes through crossovers. The first three parameters to its call here specify that `genome1` from the focal parent should be used to generate the offspring's first genome, with no second recombinant strand (`NULL`), and no recombination breakpoints (`NULL`); this provides clonal reproduction of the parent's first genome. New mutations will be added by SLiM as usual. The second group of three parameters to `addRecombinant()` are all `NULL` here; that specifies that the second offspring genome should be a null genome, so the offspring is haploid.

This approach, using `addRecombinant()`, allows the model to tell SLiM precisely how to generate offspring. This has several benefits. One is that, unlike the recipe of section 14.9, we don't have to go back and remove mutations added by SLiM to the second genomes of individuals; SLiM knows not to add mutations to those genomes in the first place. Another is that tree-sequence recording, if enabled, is able to better understand and record what is going on; it does not record the addition and removal of spurious mutations, and all of the second genomes of individuals in this model will be recorded as being null genomes (which you can test for with `pyslim`), producing a cleaner recorded tree sequence that better reflects the fact that those second genomes do not actually exist at all, conceptually. The third benefit of using `addRecombinant()` is that it allows more complex modes of offspring generation to be expressed as well; as an example, in the next section we will see how to model horizontal gene transfer in bacteria using `addRecombinant()` to express the horizontal gene transfer to SLiM.

## 16.14 Modeling clonal haploid bacteria with horizontal gene transfer

In section 16.13 we looked at a model of clonal haploids using `addRecombinant()`, as an alternative to the original haploid clonal model presented in section 14.9. The use of `addRecombinant()` allowed the details of child generation to be expressed precisely to SLiM, facilitating a simpler model design and more accurate recording of ancestry in the tree sequence (if tree-sequence recording were enabled; see section 1.7). In this section we'll explore those benefits in more detail in a model of horizontal gene transfer in bacteria.

This is a more complex model than that of section 16.13, so let's take things in two steps. First, here is all of the code except the `reproduction()` callback:

```

initialize() {
  initializeSLIMModelType("nonWF");
  defineConstant("K", 1e5); // carrying capacity
  defineConstant("L", 1e5); // chromosome length
  defineConstant("H", 0.001); // HGT probability
  initializeMutationType("m1", 1.0, "f", 0.0); // neutral (unused)
  initializeMutationType("m2", 1.0, "f", 0.1); // beneficial
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, L-1);
  initializeMutationRate(0); // no mutations
  initializeRecombinationRate(0); // no recombination
}
1 early() {
  // start from two bacteria with different beneficial mutations
  sim.addSubpop("p1", 2, haploid=T);

  // add beneficial mutations to each bacterium, but at different loci
  g = p1.individuals.genome1;
  g[0].addNewDrawnMutation(m2, asInteger(L * 0.25));
  g[1].addNewDrawnMutation(m2, asInteger(L * 0.75));
}
early() {
  // density-dependent population regulation
  p1.fitnessScaling = K / p1.individualCount;
}
late() {
  // detect fixation/loss of the beneficial mutations
  muts = sim.mutations;
  freqs = sim.mutationFrequencies(NULL, muts);

  if (all(freqs == 1.0))
  {
    catn(sim.cycle + ": " + sum(freqs == 1.0) + " fixed.");
    sim.simulationFinished();
  }
}
1e6 late() { catn(sim.cycle + ": no result."); }

```

The `initialize()` code defines a few constants: the carrying capacity, the chromosome length, and the probability that horizontal gene transfer will occur during a given mitosis event. Note that we will model horizontal gene transfer as occurring during reproduction, rather than as a discrete event occurring later in a bacterium's lifetime. This design is much simpler, particularly if tree-sequence recording is enabled; horizontal gene transfer changes the genealogical relationships among individuals, and tree-sequence recording is not designed to accommodate such changes in the middle of an individual's lifespan. The approximation seems unlikely to matter.

Note also that although we define a neutral mutation type here, `m1`, we do not model neutral mutations, and indeed, we use a mutation rate of `0.0`. This is because a model of this sort is likely to use tree-sequence recording to overlay neutral mutations after the fact for much greater speed; since we haven't gotten into tree-sequence recording yet, however, we will defer that topic until sections 17.1 and 17.2. For now, it suffices to say that we do not model neutral mutations here.

The initial population here consists of just two bacteria, which are set up to carry different beneficial mutations at different locations in the genome. The population will expand exponentially until reaching the carrying capacity of `1e5`. We have used a fairly large population size since we are modeling bacteria, but a carrying capacity of `1e6` or even higher might be

desirable for some purposes. Apart from taking more time and memory, this model should scale up without difficulties; the fact that neutral mutations are not included makes it scale much better.

In the `late()` event we detect the fixation or loss of the beneficial mutations; if both mutations have fixed or been lost, the model prints a message indicating how many mutations fixed, and then stops. If the model runs for 1e6 ticks without fixation or loss, it stops with a message.

All of that is routine. Now here's the `reproduction()` callback, where the interesting action is:

```
reproduction() {
  if (runif(1) < H)
  {
    // horizontal gene transfer from a randomly chosen individual
    HGTsource = p1.sampleIndividuals(1, exclude=individual).genome1;

    // draw two distinct locations; redraw if we get a duplicate
    do breaks = rdunif(2, max=L-1);
    while (breaks[0] == breaks[1]);

    // HGT from breaks[0] forward to breaks[1] on a circular chromosome
    if (breaks[0] > breaks[1])
      breaks = c(0, breaks[1], breaks[0]);

    subpop.addRecombinant(individual.genome1, HGTsource, breaks,
      NULL, NULL, NULL);
  }
  else
  {
    // no horizontal gene transfer; clonal replication
    subpop.addRecombinant(individual.genome1, NULL, NULL,
      NULL, NULL, NULL);
  }
}
```

Each bacterium reproduces exactly once each tick, producing two bacteria from one, which makes sense from the perspective of reproduction by mitosis. This reproduction can happen in two different ways, depending upon a random draw from `runif()`. If the draw is greater than or equal to `H`, reproduction is purely clonal as in the model of section 16.13; that is the `else` clause here. If the draw is less than `H`, horizontal gene transfer occurs, which needs some explanation.

In that case, we first draw a random individual (other than the focal individual) to act as the source for the transfer, and get its first genome. Next we use a `do-while` loop to draw two distinct locations along the genome; these will be the endpoints of the transfer. Specifically, the transfer will start at `breaks[0]` and go forward to `breaks[1]`. For a bit of extra biological realism, we will model a circular chromosome here, so if `breaks[0]` is greater than `breaks[1]` the transfer will wrap around from the end of the genome to the start, as modelled in SLiM; we check for that case and patch up the `breaks` vector to reflect what we want to happen in that situation. Finally, we call `addRecombinant()` to generate the offspring bacterium including the horizontal gene transfer. We pass it to the parent genomes – that of the reproducing bacterium, and that of the horizontal gene transfer source – with the breakpoint vector that describes when SLiM should switch between those strands as it produces the offspring genome by recombination. As before, we pass `NULL` for the next three parameters to indicate that the second offspring genome should be a null genome (conceptually, nonexistent). (A diploid model that wanted to generate its own recombination breakpoints might use those parameters, for example.)

Without horizontal gene transfer, this would be a model of clonal competition: one lineage would end up “winning” and the other would go extinct, although it might take a long time for

that outcome to be reached since it would depend on drift. This behavior can be seen by setting `H` to `0.0`. With horizontal gene transfer, however, the bacteria will often stumble upon a lineage (or perhaps more than one lineage) that combines both mutations in the same genome, providing an advantage similar to that provided by recombination in sexual reproduction. Once such a lineage arises it will almost always win, and we will get output like this from the model:

```
197: 2 fixed.
```

Both beneficial mutations fixed in tick 197, thanks to horizontal gene transfer.

The details of the breakpoint generation here might need to be modified in a more realistic model. Here we draw the start and end positions of the transfer region independently, but perhaps it would be better to draw the start location randomly and then draw a transfer length from a geometric distribution or some other distribution. This would constrain the horizontal gene transfer to generally be a small minority of the genome, as is typical in the transfer of a plasmid or a transposon. The location and length of the transfer could also be constrained by some sort of genetic structure to explicitly model the transfer of a plasmid that spans a given range of the genome, of course. The `reproduction()` callback could also base the choice of whether or not horizontal gene transfer occurs upon the contents of the two genomes in question, not just upon a random probability; one could model a selfish gene in the transfer donor that makes horizontal gene transfer more likely to occur, for example. Since all of the logic governing the horizontal gene transfer is in the model's script, it can include whatever biological realism is of interest.

Note that prior to the addition of `addRecombinant()` in SLiM, it would have been possible to model horizontal gene transfer by actually getting all of the mutations from the transfer region out of the source's genome, and then adding them into the target's genome with `addMutations()` (removing any existing mutations from the target region first). This would work fine except that it obscures what is actually going on in terms of genealogy and inheritance. If tree-sequence recording were used with such a model, the transferred region would not be recorded as originating in the source genome; instead, the mutations would just magically appear in the target genome, with no genealogical relationship between source and target recorded in the tree sequence. The method presented here, using `addRecombinant()`, is therefore preferable. (If one needed to model even more complex patterns of inheritance – offspring genomes that consist of a mosaic of genetic material from more than two parental genomes, for example – using the `addMutations()` technique might still be necessary, however, since `addRecombinant()` is designed to record at most two parental genomes for each offspring genome. Tree-sequence recording would not work well in such a model, however.)

To make a relatively realistic model of bacterial evolution with SLiM, this sort of realistic inheritance and horizontal gene transfer is one important ingredient. The other important ingredient is the ability to model a sufficiently large population size, which is often made possible by the performance benefits provided by tree-sequence recording as we will see in chapter 17. For a related bacterial HGT model that incorporates tree-sequence recording, and also manipulates recombination breakpoints to simulate a circular chromosome, see Cury et al. (2022), which inspired this simplified recipe. Thanks to Jean Cury for his collaboration on these models.

## 16.15 Implementing a Wright–Fisher model with a nonWF model

In this chapter, we have focused on aspects of nonWF models that go beyond the Wright–Fisher model, such as overlapping generations, age structure, and individual-level control over reproduction and migration. Sometimes, however, it can be useful to implement a Wright–Fisher model as a nonWF model in SLiM – or at least some aspects of a Wright–Fisher model. You might want to have non-overlapping generations, for example; or you might want panmictic offspring



generation as in the Wright–Fisher model, with each offspring generated from an independent, randomly drawn pair of parents. Implementing such a model using the nonWF model type might be desirable because you might also want some non-Wright–Fisher dynamics in your model that would be difficult to implement in a WF model, or you might want to take advantage of features of SLiM that are only available in nonWF models. In this section, we will look at two nonWF models that incorporate aspects of the Wright–Fisher model. Both models will include deleterious mutations in addition to neutral mutations, to show how each model treats fitness.

The first recipe here is quite simple, so let’s look at it in full:

```
initialize() {
  initializeSLiMModelType("nonWF");
  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;
  initializeMutationType("m2", 0.0, "f", -0.5);
  initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.05));
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
reproduction() {
  K = sim.getValue("K");

  // parents are chosen randomly, irrespective of fitness
  parents1 = p1.sampleIndividuals(K, replace=T);
  parents2 = p1.sampleIndividuals(K, replace=T);

  for (i in seqLen(K))
    p1.addCrossed(parents1[i], parents2[i]);

  self.active = 0;
}
1 early() {
  sim.setValue("K", 500);
  sim.addSubpop("p1", sim.getValue("K"));
}
early()
{
  // parents die; offspring survive proportional to fitness
  inds = sim.subpopulations.individuals;
  inds[inds.age > 0].fitnessScaling = 0.0;
}
10000 late() { sim.outputFixedMutations(); }
```

In the `initialize()` callback, we set up this simple nonWF model with neutral and deleterious mutations and a uniform chromosome. Note that we do not set up a constant `K` for the population carrying capacity there; instead, in the `1 early()` event, we set `K` up as a value kept by the simulation using `setValue()`, making it easier to vary `K` over time (although we don’t do so in this simple recipe). The `p1` subpopulation begins with a size equal to the value of `K` just defined.

The `reproduction()` callback implements a Wright–Fisher style of reproduction. It gets the current subpopulation size using `getValue()`, and then generates that many offspring into `p1` with randomly drawn parents for each offspring. It then deactivates itself by setting `self.active` to `0`, to prevent SLiM from calling it again, since it has reproduced the entire subpopulation; we first saw this “big bang reproduction” strategy in section 16.3. Note that this code does not prevent the first and second parents of a given offspring from being the same individual, so a certain amount of “incidental selfing” will occur (as is typical in the Wright–Fisher model). Changing that would



require that the second parents be drawn inside the for loop, one by one, using the `exclude` parameter to `sampleIndividuals()`, which would look something like this:

```
secondParent = p1.sampleIndividuals(1, exclude=firstParent);
```

This draws the second parent from `p1` while explicitly excluding `firstParent` as a choice; the `sampleIndividuals()` method has many useful options of this sort. We won't make that change in this recipe, though, since incidental selfing is often desirable in Wright–Fisher models (to match the results of analytical models).

Changing this model to be sexual instead of hermaphroditic would also be easy, since `sampleIndividuals()` can similarly be told to draw only females (for the first parents) or only males (for the second parents). One could also easily implement the Wright–Fisher style of migration between subpopulations by generating some fraction  $m$  of the new offspring in `p1` from parents drawn from a different subpopulation instead.

The `early()` event implements non-overlapping generations by simply killing off all non-juvenile individuals, by setting their `fitnessScaling` to `0.0`. This is, in effect, an extremely simple version of the sort of life table we saw in section 16.2. Note that unlike most nonWF models, this model has no other population regulation; the usual density-dependence code is absent. Since the `reproduction()` callback always generates exactly  $K$  offspring, and all non-juveniles are killed off each tick, the size of the population (before the viability/survival tick cycle stage) is deterministic and the population does not require further regulation.

This model remains non-Wright–Fisher in one key way: fitness is still expressed through pre-mating mortality, not through an individual's probability of mating. This has three important consequences. First of all, although the population size is  $K$  after offspring generation, it can drop below  $K$  during the viability/survival tick cycle stage due to fitness-based mortality. The model is thus a “hard selection” model, not a “soft selection” model as is typical for Wright–Fisher models. Second, fitness values over 1.0, due to beneficial mutations or other effects, have no effect since the probability of survival can be at most 100% (see section 16.4 for discussion of this issue.) Third, the distribution of fecundity among individuals here is different from that of a Wright–Fisher model; in this model, as in most nonWF models, either an individual survives (in which case it has the same expected fecundity as any other surviving individual) or it dies (in which case it has an expected fecundity of zero, being dead). In the Wright–Fisher model, on the other hand, fitness modifies the probability that an individual will be chosen as a mate, and so the distribution of fecundity is continuous rather than bimodal. In many cases this difference will have very little effect upon the outcome of the model, so this simple pseudo-Wright–Fisher model design might suffice. However, in SLiM 3.7 and later it is straightforward to modify this model design to make fitness influence mating success rather than survival, using a `survival()` callback. Here is the second recipe for this section, showing that modified design:

```
initialize() {
  initializeSLiMModelType("nonWF");
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.0, "f", -0.5);
  c(m1,m2).convertToSubstitution = T;
  initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.05));
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
```

```

reproduction() {
    K = sim.getValue("K");

    // parents are chosen proportional to fitness
    inds = p1.individuals;
    fitness = p1.cachedFitness(NULL);
    parents1 = sample(inds, K, replace=T, weights=fitness);
    parents2 = sample(inds, K, replace=T, weights=fitness);

    for (i in seqLen(K))
        p1.addCrossed(parents1[i], parents2[i]);

    self.active = 0;
}
1 early() {
    sim.setValue("K", 500);
    sim.addSubpop("p1", sim.getValue("K"));
}
survival() {
    // survival is independent of fitness; parents die, offspring live
    return (individual.age == 0);
}
10000 late() { sim.outputFixedMutations(); }

```

The 1 early(), and 10000 late() callbacks are identical. There are three important differences to discuss: the survival() callback, the reproduction() callback, and the initialize() callback.

First of all, the early() event of the previous recipe, which set the fitnessScaling of the parental generation to 0.0 to provide non-overlapping generations, has been removed; in its place is a survival() callback. These callbacks are called during the viability/survival tick cycle stage in nonWF models, and govern the decisions SLiM makes regarding which individuals live and which die (see section 25.10 for details). Here, we return F for individuals in the parental generation, telling SLiM that those individuals should die regardless of their fitness, and we return T for juveniles, telling SLiM that they should survive regardless of their fitness. The key insight here, then, is that we are still allowing SLiM to calculate fitness values for all individuals based upon the deleterious mutations they possess (and any other fitness effects that might be present in the model), but we are completely decoupling fitness from mortality with the survival() callback. If we left things there, the model would actually become a neutral model; the deleterious mutations would be prevented from having any effect on the actual fitness of individuals, since the fitness values calculated by SLiM would no longer affect actual fitness in the simulation.

Of course that is not what we want; we want fitness values to influence mating success instead. We achieve that with the modified reproduction() callback. The key lines are these:

```

fitness = p1.cachedFitness(NULL);
parents1 = sample(inds, K, replace=T, weights=fitness);
parents2 = sample(inds, K, replace=T, weights=fitness);

```

With these lines, we replace the unbiased selection of parents in the first recipe with a biased selection of parents, proportional to fitness. We use the cachedFitness() method to get SLiM's calculated fitness values for all individuals; they will be in the same order as in p1.individuals, so the elements in the inds and fitness vectors correspond one-to-one. Then we use the sample() function to draw parents with weights=fitness; we no longer use sampleIndividuals() since it does not have a weights parameter.

Finally, the `initialize()` callback now has this line:

```
c(m1,m2).convertToSubstitution = T;
```

Now that fitness is relative and selection is soft, we can allow the non-neutral mutations in the model to be substituted by SLiM when they fix; just as in a WF model, this is safe because a deleterious or beneficial mutation that is fixed across the population has no effect on relative fitness. In this particular model this makes no real difference, since the deleterious mutations modeled here have such a large negative selection coefficient that they will probably never fix anyway; but this is nevertheless an important point for models that involve mutations that are nearly neutral, or even beneficial.

With these changes, we now have a true Wright–Fisher model, and the three caveats discussed above regarding the first recipe no longer apply. Specifically, (1) the population size will remain at *K* regardless of mean fitness, so fitness is now *relative* fitness and this model is now a soft-selection model; (2) fitness values over 1.0 will now influence mating success as one would expect, since fitness is now interpreted as relative fitness rather than absolute fitness; and (3) the distribution of fecundity will no longer be bimodal, since it will no longer be based upon the black-and-white fact of which individuals survive. The only drawback of this second recipe is that it will run a little more slowly than the first, since drawing parents weighted by fitness is more computationally intensive than drawing them without weighting. The difference is not terribly large – a little over a 2× slowdown, perhaps – so in most cases it will make sense to choose the model design that is best suited to the problem at hand.

Even the first recipe shown here is substantially more complex than the equivalent WF model, which looks like this:

```
initialize() {
  initializeMutationType("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.0, "f", -0.5);
  initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.05));
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
10000 late() { sim.outputFixedMutations(); }
```

And it runs several times slower – about 3×, in an informal test. The speed penalty for switching to a nonWF model is large here because most of what this model really does is reproduce, and the reproduction for the nonWF model is handled in script rather than in SLiM's core as it is for the WF model. The penalty associated with switching to nonWF is therefore almost maximized here; if the model had other time-consuming things to do besides reproduce, the penalty would be smaller. For example, if the chromosome length in this model were  $10^7$  instead of  $10^5$ , SLiM would spend much more time handling the genetics of the model – doing mutation and recombination, as well as checking for fixation or loss of mutations – and in that case, this recipe would take only  $\sim 1.35\times$  as long as the equivalent WF model. With a chromosome length of  $10^8$ , the slowdown is only a factor of  $\sim 1.07\times$ . This underlines that when making the choice between a WF and a nonWF model, that choice should almost always be made based upon which model type is a better fit for the scenario being simulated, rather than upon performance. If a model is big and complex enough that its runtime is problematic – i.e., is measured in hours to days – the overhead due to choosing the nonWF model type will probably be small.

## 16.16 Alternation of generations

SLiM is generally a framework for modeling diploid organisms, but with some creative scripting that assumption can be modified. We have seen some recipes for modeling haploids, as in sections 14.9, 16.13, and 16.14. In nonWF models a similar strategy can be used to fully model the phenomenon of *alternation of generations*, the way that diploid and haploid life cycle stages generally alternate in organisms that are often thought of simply as “diploids”. Many sexual animals, for example, have a multicellular diploid phase that produces a unicellular haploid phase – sperm and eggs – that then fuse, in fertilization, to produce the next diploid generation. In plants this situation is generally even more pronounced, often with a multicellular haploid phase, the gametophyte, that can be free-living and large – often larger and more obvious than the diploid sporophyte, which is often reduced. For many organisms, then, it may be important to model both the haploid and diploid phases explicitly; mutations may be expressed differently between them, selection may act differently upon them, they may migrate or disperse differently, and so forth. SLiM does not have intrinsic support for modeling this alternation of generations, but it is straightforward to implement in script in a nonWF model, as we will see in this section.

This model will be somewhat complicated, so let’s start with the setup:

```
initialize()
{
    defineConstant("K", 500);    // carrying capacity (diploid)
    defineConstant("MU", 1e-7);  // mutation rate
    defineConstant("R", 1e-7);   // recombination rate
    defineConstant("L1", 1e5-1); // chromosome end (length - 1)

    initializeSLiMModelType("nonWF");
    initializeSex("A");
    initializeMutationRate(MU);
    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, L1);
    initializeRecombinationRate(R);
}
1 early()
{
    sim.addSubpop("p1", K);
    sim.addSubpop("p2", 0);
}
```

We use defined constants for several of the model parameters. The recipe here involves only neutral mutations, but extending it to other types of mutations should present no difficulties.

This is a sexual model, so we set up separate sexes with `initializeSex()`. We are not modeling sex chromosomes, but we will track the sex of individuals in both the diploid and haploid phase; sperm will be considered “male”, and eggs “female”, in this model.

A key point in the design of this model is that although we are modeling only a single subpopulation, we use two subpopulations in the model, p1 and p2. The first, p1, is used to hold diploids; the second, p2, is used to hold the haploid sperm and eggs. This separation is not strictly necessary, but it makes the design of the model simpler, because this way we can define a `reproduction()` callback for p1 that reproduces the diploids (producing sperm and eggs), and a separate `reproduction()` callback for p2 that reproduces the haploids (producing fertilized eggs that develop into diploids). For other processing of the individuals in the model, such as `mutationEffect()` callbacks, this partitioning will also prove useful, as we will see below.

The next step is to define the `reproduction()` callbacks. Let's start with the one for `p1`:

```
reproduction(p1)
{
  g_1 = individual.genome1;
  g_2 = individual.genome2;

  for (meiosisCount in 1:5)
  {
    if (individual.sex == "M")
    {
      breaks = sim.chromosome.drawBreakpoints(individual);
      s_1 = p2.addRecombinant(g_1, g_2, breaks, NULL, NULL, NULL, "M");
      s_2 = p2.addRecombinant(g_2, g_1, breaks, NULL, NULL, NULL, "M");

      breaks = sim.chromosome.drawBreakpoints(individual);
      s_3 = p2.addRecombinant(g_1, g_2, breaks, NULL, NULL, NULL, "M");
      s_4 = p2.addRecombinant(g_2, g_1, breaks, NULL, NULL, NULL, "M");
    }
    else if (individual.sex == "F")
    {
      breaks = sim.chromosome.drawBreakpoints(individual);
      if (runif(1) <= 0.5)
        e = p2.addRecombinant(g_1, g_2, breaks, NULL, NULL, NULL, "F");
      else
        e = p2.addRecombinant(g_2, g_1, breaks, NULL, NULL, NULL, "F");
    }
  }
}
```

The definitions of `g_1` and `g_2` at the beginning are just shorthand, to keep the lines later in the callback from being so long that they wrap when shown here. As usual in nonWF models, this callback is called by SLiM once per individual in `p1`, giving the individual an opportunity to reproduce – in this model, an opportunity to produce gametes. The top-level loop causes the focal diploid individual to undergo meiosis exactly five times; this is an oversimplification, obviously, but there is no need, in most models, to generate millions of sperm. Within the loop, male individuals undergo meiosis by producing four sperm, whereas females produce just a single egg (plus three “polar bodies” that are discarded by meiosis in most sexual species, due to anisogamy; the polar bodies are not modeled here).

Gametes are produced by the `addRecombinant()` method, adding the resulting haploid individuals to `p2`. The calls to `addRecombinant()` here pass `NULL` for the genomes and breakpoints that generate the second genome of the offspring; this results in an empty second genome in the offspring, as is typical when modeling haploids in SLiM. The genomes used to generate the first genome of the offspring can be supplied as either (`g_1, g_2`) or (`g_2, g_1`); the first of the two genomes supplied is the copy strand at the beginning of recombination. Since the sperm generated use all of the genetic material from meiosis, both of those options are used (twice, because of the homologous chromosomes involved in meiosis); since egg generation produces only a single gamete, the choice of initial copy strand is randomized with a call to `runif()`.

Particularly for the sperm, since we want to generate the gametes in a realistic fashion following the rules of meiosis, we generate the recombination breakpoints ourselves and use them to generate complementary gametes. To generate breakpoints in the standard SLiM fashion, we call the `drawBreakpoints()` method of `Chromosome`; by default this produces a set of breakpoints identical to what SLiM would generate for its own internal use in reproduction. The number of

recombination breakpoints generated is chosen by `drawBreakpoints()`, by default, using the overall recombination rate defined by the model.

Now let's look at the `reproduction()` callback for `p2`, which contains haploid gametes:

```
reproduction(p2, "F")
{
    mate = p2.sampleIndividuals(1, sex="M", tag=0);
    mate.tag = 1;

    child = p1.addRecombinant(individual.genome1, NULL, NULL,
                             mate.genome1, NULL, NULL);
}
```

This callback is defined only for females – i.e., eggs. Each egg gets to “reproduce” – be fertilized – to produce a new diploid organism in `p1`. In this model a random sperm is chosen to fertilize each egg, but one could easily implement phenomena such as sperm competition here to make the choice non-random. We mark sperm that have been used to fertilize an egg with a `tag` value of 1, so that they will not be used again; when we draw a random sperm, we specify in the call to `sampleIndividuals()` that the sperm chosen must have a `tag` value of 0, indicating that it has not already been used. Once the fertilizing sperm has been selected, it is tagged with a value of 1, and the diploid zygote is generated with a call to `addRecombinant()`. The call to `addRecombinant()` here supplies only a single genome for each of the offspring genomes, with `NULL` for the breakpoint vectors; this makes the offspring's genomes a clonal copy of the corresponding genomes from the gametes. Normally, new mutations would be generated and added by SLiM during this clonal replication; we will fix that momentarily.

These callbacks implement the generation of gametes and then the fusion of gametes to produce diploid zygotes; but a little additional machinery is needed, which we implement in an `early()` callback that cleans up after reproduction and sets up for the next reproduction event:

```
early()
{
    if (sim.cycle % 2 == 0)
    {
        p1.fitnessScaling = 0.0;
        p2.individuals.tag = 0;
        sim.chromosome.setMutationRate(0.0);
    }
    else
    {
        p2.fitnessScaling = 0.0;
        p1.fitnessScaling = K / p1.individualCount;
        sim.chromosome.setMutationRate(MU);
    }
}
```

In even-numbered generations the top half of this event will execute; in odd-numbered generations the bottom half will execute. In an even-numbered generation, at the point that `early()` events are called, `p1` will have just generated gametes. This recipe assumes non-overlapping generations, so here we kill off the diploids by setting their `fitnessScaling` to 0.0; `p1` will be emptied out completely. Next, we set the `tag` values of all of the gametes in `p2` to 0; this marks all of the sperm as unused, in preparation for the way the `reproduction(p2)` callback uses the `tag` field. Finally, we set the mutation rate to 0.0; we do not want new mutations to be generated by SLiM during fertilization, so we need to disable mutation temporarily.



In odd-numbered generations, gametes have just undergone fertilization, filling p1 up with new diploid offspring. We therefore kill off the haploid gametes by setting their `fitnessScaling` to `0.0`; p2 will be emptied out completely. Since each egg produces a zygote, we will have way too many diploids; we will be far above carrying capacity. The next line thus implements density-dependent fitness for p1, as usual in nonWF models; note that no such density-dependence was imposed upon the gametes in this model. Finally, we set the mutation rate back up to the defined constant MU, since we want new mutations to arise during gamete production.

With this design, the population will flip back and forth between p1 and p2 as it flips between diploidy and haploidy, and the mutation rate will flip on and off as well. It would be straightforward to implement overlapping generations of diploids in this model; one could even delve into more esoteric ideas such as sperm storage. Fitness effects could differ in the haploid and diploid phases, by implementing `mutationEffect()` callbacks that apply only to p1 or p2.

All that is left to finish off the model is a termination event, which here is trivial:

```
1000 late()
{
    sim.simulationFinished();
}
```

This approach to modeling the alternation of generations may be overkill in many practical situations. This model runs much more slowly than the equivalent model of only the diploid phase; for one thing, it is generating a population of gametes that is more than ten times larger than the population of diploids, every generation. Other strategies for modeling life cycle complexity may be usable instead; section 16.8, for example, presents a model of pollen flow between subpopulations of plants, which is simple to model without getting down to the details of modeling individual pollen grains and the sperm cells they produce as separate entities. Additional biological realism should generally be incorporated into a model only when there is reason to believe that it matters – that it would affect the results of the model. In some cases, however – such as when one wishes to have selection operate in the haploid phase – the additional biological realism of modeling alternation of generations may be useful. If you do want to model alternation of generations, you might find the *shadie* Python package useful; it provides a Python front end (with SLiM as a back end) for many such models (see section 1.10).

One could use the same basic concepts to model of phenomena such as haplodiploidy (see section 16.24); `addRecombinant()` allows many different mating systems to be modeled. Partitioning the population according to genetics – here, diploids versus haploids – is also a useful trick in many scenarios. Indeed, such artificial partitioning can be very useful in other contexts too, such as storing non-reproducing juveniles separately from reproductive adults, or storing sympatric but reproductively isolated groups separately. Since the spatial interaction engine of *InteractionType* evaluates interactions on a per-subpopulation basis (see chapter 15), it can also be useful to partition the population according to “interaction groups” – sets of individuals that interact with each other, but not with the individuals in other interaction groups.

## 16.17 Meiotic drive

In section 12.3 we saw a model of a gene drive, a genetic construct that drives itself higher in frequency by copying itself from one chromosome to the other. We implemented that mechanism with a `modifyChild()` callback that copied the drive allele from one child genome to the other in each newly generated offspring. In this section, we will look at a different type of drive: *meiotic drive*. Meiotic drive involves an allele that, when heterozygous in a parent, somehow ensures that it is inherited by more than 50% of offspring. Various physical mechanisms might underlie such an

inheritance bias, but the mechanism is unimportant for the model here; we will simply model an introduced allele that exhibits meiotic drive.

There is more than one way that a meiotic drive could be implemented in SLiM. One could imagine a `modifyChild()` callback that would check whether a parent had the drive allele, and if so, and if it failed to be inherited by the proposed child, would return `F` to prevent generation of that child. That approach would have the advantage of being compatible with WF models, but it would entail significant overhead since many proposed offspring might be vetoed; and in a nonWF model, the vetoing of proposed children would decrease the population size unless the model took additional steps to ensure that vetoed crosses were redone. Here we will model a more intuitive nonWF-based approach in which we control the recombination breakpoints that are used for offspring generation, making breakpoints that lead to inheritance of the drive allele more likely than breakpoints that do not. Offspring generation therefore always succeeds (unlike the `modifyChild()` design), but inheritance of the meiotic drive allele is biased, as desired.

The bulk of the model is straightforward:

```
initialize() {
  initializeSLiMModelType("nonWF");
  defineConstant("K", 500);           // carrying capacity
  defineConstant("D_pos", 20000);     // meiotic drive allele position
  defineConstant("D_prob", 0.8);     // meiotic drive probability

  initializeMutationType("m1", 0.5, "f", 0.0); // neutral
  m1.convertToSubstitution = T;

  initializeMutationType("m2", 0.1, "f", -0.1); // drive allele
  m2.color = "red";
  m2.convertToSubstitution = F;

  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 10);
}
early() {
  p1.fitnessScaling = K / p1.individualCount;
}
100 early() {
  target = sample(p1.genomes, 10);
  target.addNewDrawnMutation(m2, D_pos);
}
100:1000 late() {
  mut = sim.mutationsOfType(m2);
  if (size(mut) == 0) {
    catn(sim.cycle + ": LOST");
    sim.simulationFinished();
  } else if (sim.mutationFrequencies(NULL, mut) == 1.0) {
    catn(sim.cycle + ": FIXED");
    sim.simulationFinished();
  }
}
```

Similar to the selective-sweep models of chapter 9, we introduce a drive allele in tick 100 and monitor it for fixation or loss in every tick thereafter. Unlike those selective-sweep models,

however, the drive allele is strongly deleterious, with a selection coefficient of  $-0.1$ , so if left to its own devices it would virtually always be lost quickly. However, the meiotic drive machinery will take care of that problem. Here is the `reproduction()` callback for the recipe:

```
reproduction() {
  m = subpop.sampleIndividuals(1);
  b1 = driveBreakpoints(individual.genome1, individual.genome2);
  b2 = driveBreakpoints(m.genome1, m.genome2);
  subpop.addRecombinant(individual.genome1, individual.genome2, b1,
    m.genome1, m.genome2, b2);
}
```

This callback uses `addRecombinant()` to produce its offspring. We used this method before in section 16.13 to implement clonal haploid reproduction, in 16.14 to implement horizontal gene transfer in bacteria, and in 16.16 to implement alternation of generations. As those diverse uses show, it is quite a flexible tool for precisely controlling offspring generation in unusual models, and here we use it to control gamete production in such a manner as to enforce the operation of the meiotic drive. The code here is very simple, however; we just choose a mate, draw recombination breakpoints to generate the two gametes, and call `addRecombinant()` to generate the offspring with those breakpoints. The real meiotic drive magic is in the `driveBreakpoints()` function, which we define as follows:

```
function (i)driveBreakpoints(o<Genome>$ gen1, o<Genome>$ gen2)
{
  // start with default breakpoints generated by the chromosome
  breaks = sim.chromosome.drawBreakpoints();

  // if both genomes have the drive, or neither, then just return
  gen1has = gen1.containsMarkerMutation(m2, D_pos);
  gen2has = gen2.containsMarkerMutation(m2, D_pos);
  if (gen1has == gen2has)
    return breaks;

  // will the drive be inherited? do we want it to be?
  polarity = sum(breaks <= D_pos) % 2; // 0 for gen1, 1 for gen2
  polarityI = (gen1has ? 0 else 1);
  desiredPolarity = (runif(1) < D_prob) ? polarityI else !polarityI;

  // intervene to produce the outcome we want
  if (desiredPolarity != polarity)
    return c(0, breaks);
  return breaks;
}
```

This user-defined function begins by calling `sim.chromosome.drawBreakpoints()` to draw recombination breakpoints following SLiM's standard procedure (using whatever recombination rate map has been defined, etc.). Remember that during meiosis, these breakpoints result in crossovers that recombine the parental genomes, and biologically speaking, either result of this crossing-over could end up as the gamete used in fertilization; if one possible gamete is AABABBB (A and B representing genomic regions from the two parental genomes), there will be another possible gamete that is BBABAAA, the reciprocal result from the crossing-over. What we do in this function is simply bias the choice of which of these possible gametes actually gets used to produce the offspring generated by SLiM. From the perspective of the meiotic drive allele, this choice only matters if the parent is heterozygous, such that one gamete would contain the drive allele and the other would not, so we check for that up front. If the parent is indeed heterozygous, then we look

at the proposed breakpoints to determine which parental genome *would be* the copy strand at the meiotic drive location (represented by the variable `polarity`), and which parental genome *ought to be* the copy strand at the meiotic drive location to produce the result we want (represented by the variable `desiredPolarity`), whether that result is inheritance or non-inheritance of the drive allele (as determined by a coin flip with probability `D_prob`). If those two are not the same – if it *would be* inherited but it *oughtn't* to be, or it *wouldn't* be inherited but it *ought* to be – we simply add a new breakpoint at position 0 at the start of the breakpoint vector before returning it, which has the effect of flipping which genome is used as the initial copy strand.

When this recipe is run, the meiotic drive allele introduced in tick 100 usually sweeps to fixation, although occasionally it is lost early on due to the combination of drift and the selection against it. The probability that it will sweep rather than being lost will depend upon how deleterious it is, what the dominance coefficient for its fitness effect is, and the probability that it will drive during meiosis (`D_prob`), among other things. Indeed, for some values of these parameters the drive will tend to neither fix nor be lost, but instead will often equilibrate at an intermediate frequency.

This recipe could also have been written with a `recombination()` callback that did essentially the same work as the `driveBreakpoints()` function does here. That design would also be fine; depending on what else needs to happen in the model, one or the other might be preferable. The `recombination()` callback design would be usable in WF as well as nonWF models; on the other hand, the design shown in this recipe might work better if, for example, the meiotic drive should operate only in certain circumstances, such as environmental context, which might be easier to determine at the level of the `reproduction()` callback than inside a `recombination()` callback. For many purposes, both designs might be equally good.

This recipe could also be considered to represent other types of inheritance bias and intragenomic conflict, such as an allele that is favored during competition among sperm from a single mate, or an allele that causes lethal maternal effects in offspring from a heterozygous mother that do not inherit the allele. (The latter case might be better implemented as a `modifyChild()` callback, however, as discussed at the beginning of this recipe, since one might want such mortality to actually depress the population size in a nonWF model.)

## 16.18 A spatial epidemiological S-I-R model

Epidemiological models are concerned with the spread of disease through a population, and the effects of various public health interventions and behavioral changes on the course of that spread. There is a large literature on one particular type of epidemiological model, the S-I-R (Susceptible, Infectious, Recovered) model. Susceptible individuals have not previously encountered the disease, and can contract it if exposed. Infectious individuals have contracted the disease and can spread it to susceptible individuals they encounter. Recovered individuals have had the disease and recovered, so they are no longer infectious, nor are they susceptible (since their immune system is no longer naive). This model is very simple, but it provides a conceptual foundation and can be extended in many ways to increase realism.

Simple non-spatial S-I-R models are quite trivial to put together, but spatial dynamics are immensely important to epidemiology. Different countries or regions will have different healthcare systems, different laws, and different cultures that may affect dynamics. Geographical barriers can be important (such as oceans), as can movement patterns (such as air travel). Many diseases act differently depending upon the climate, causing seasonal patterns of spread. Population density matters immensely; disease spreads much more readily in urban areas, but zoonotic diseases often start in rural areas. And so forth – many other types of spatial variation in epidemiologically relevant variables might be important to model. Spatial epidemiological models are not so simple

to put together, so SLiM can be a useful tool in this domain. A SLiM model of S-I-R dynamics should be a nonWF model, since population size needs to be emergent. Although we won't delve into it here, a nonWF model is also appropriate because age might matter a great deal – individuals of different ages might differ in their susceptibility, their mortality, their behaviors, etc.

In this section, then, we will look at a very simple spatial S-I-R model in SLiM. Although this model is being added while the COVID-19 pandemic is raging, please note that writing it has been on my to-do list for months. It is not a model of COVID-19; indeed, it is not even a model of humans, or any other specific organism.

We will use periodic boundary conditions in a 2-D space to avoid edge effects (see section 15.12), and we'll use the tag property of individuals to keep track of the S-I-R state of each individual. Ticks will represent a fairly short amount of time compared to the lifespan of individuals; individuals will have a maximum age of 100 ticks (with increasing mortality with age, independent of infection). The model will be based strongly on the recipe of section 16.11, with spatial competition using carrying-capacity density, and nearest-neighbor mate choice. Here's the initialization:

```
initialize() {
    initializeSLiMModelType("nonWF");
    initializeSLiMOptions(dimensionality="xy", periodicity="xy");

    defineConstant("K", 10000); // carrying-capacity density
    defineConstant("S", 0.01); // sigma_S, the competition width

    defineConstant("HEALTH_S", 0); // susceptible
    defineConstant("HEALTH_I", 1); // infectious
    defineConstant("HEALTH_R", 2); // recovered

    defineConstant("FERTILITY", 0.05);
    defineConstant("INFECTIVITY", 4);
    defineConstant("RATE_DEATH", 0.3);
    defineConstant("RATE_CLEAR", 0.05);
    defineConstant("MAX_AGE", 100.0);

    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;

    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);

    // spatial competition
    initializeInteractionType(1, "xy", reciprocal=T, maxDistance=S * 3);
    i1.setInteractionFunction("n", 1.0, S);

    // spatial mate choice
    initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.05);
}
```

The population size is relatively large for this manual (a carrying-capacity density of 10000) so that we can get some interesting and fine-grained spatial dynamics, and the interaction distances defined are therefore relatively short-range. We define constants for the S-I-R states that we will keep in the tag property, to make the code more self-documenting. We also define a set of constants governing things like fertility, infectivity, and other dynamics.

Notably, the model includes mutations, but only neutral mutations that don't influence the dynamics. For all practical purposes, then, this model is therefore non-genetic and non-evolutionary; we are just using SLiM as a spatial agent-based modeling framework. However, all the genetic machinery in SLiM is still available to us, so a natural extension would be to look at the effects of resistance alleles or immunological genetic traits on the outcome of the model.

Reproduction in this model is quite straightforward:

```
1: first() {
    // look for mates
    i2.evaluate(p1);
}
reproduction() {
    litterSize = rpois(1, FERTILITY);

    if (litterSize)
    {
        mate = i2.nearestNeighbors(individual, 1);

        if (mate.size())
            for (i in seqLen(litterSize))
            {
                offspring = subpop.addCrossed(individual, mate);

                // set offspring position and state
                pos = individual.spatialPosition + rnorm(2, 0, 0.005);
                offspring.setSpatialPosition(p1.pointPeriodic(pos));
                offspring.tag = HEALTH_S;
            }
    }
}
```

As in sections 16.10 and 16.11, we begin by evaluating the mating interaction in a `first()` event, prior to reproduction. Then we determine whether the focal individual is even a candidate to reproduce this tick, since the fertility rate is so low. If it is, we choose the nearest mate within the mating interaction distance, generate the litter of offspring (probably a single offspring, given the low fertility), and handle dispersal for the new offspring, which begin as Susceptible.

The population setup sets individuals to be Susceptible, in addition to initializing positions:

```
1 early() {
    sim.addSubpop("p1", K);
    p1.individuals.setSpatialPosition(p1.pointUniform(K));
    p1.individuals.tag = HEALTH_S;
}
```

Seeding the initial infection is a matter of choosing a target and changing its state to Infectious:

```
100 early() {
    // seed the infection in a susceptible individual
    target = p1.sampleIndividuals(1, tag=HEALTH_S);
    target.tag = HEALTH_I;
}
```

We want to do some basic logging (for later display of the epidemic history in R), and we want to detect when the infection has been lost:



```

1:1000 late() {
  tags = p1.individuals.tag;

  cat(sum(tags == HEALTH_S) + ", " + sum(tags == HEALTH_I) + ", " +
      sum(tags == HEALTH_R) + ", ");

  if ((sum(tags == HEALTH_I) == 0) & (sim.cycle >= 100)) {
    catn("\nLOST in cycle " + sim.cycle);
    sim.simulationFinished();
  }
}

```

We want several forms of population regulation and mortality: spatial competition providing basic density-dependence, age-related mortality providing a cap of the age of individuals, and – most saliently – infection dynamics and death due to disease. We do all this in an `early()` event:

```

early() {
  i1.evaluate(p1);

  // spatial competition provides density-dependent selection
  inds = p1.individuals;
  competition = i1.totalOfNeighborStrengths(inds);
  competition = (competition + 1) / (2 * PI * S^2);
  inds.fitnessScaling = K / competition;

  // age-based mortality; at age 100 mortality is 100%
  age_mortality = sqrt((MAX_AGE - inds.age) / MAX_AGE);
  inds.fitnessScaling = inds.fitnessScaling * age_mortality;

  // SIR model
  infected = inds[inds.tag == HEALTH_I];

  for (ind in infected)
  {
    // make contact with random neighbors each cycle
    contacts = i1.drawByStrength(ind, rpois(1, INFECTIVITY));

    for (contact in contacts)
    {
      // if the contact is susceptible, they might get infected
      if (contact.tag == HEALTH_S)
      {
        strength = i1.strength(ind, contact);

        if (runif(1) < strength)
          contact.tag = HEALTH_I;
      }
    }

    // die with some probability each cycle
    if (runif(1) < RATE_DEATH)
      ind.fitnessScaling = 0.0;

    // recover with some probability each cycle
    if (runif(1) < RATE_CLEAR)
      ind.tag = HEALTH_R;
  }
}

```

The age-based mortality is implemented with a simple formula that makes the probability of death 0% at age 0 and 100% at age 100; the `sqrt()` call transforms that probability scale so that, within those boundary conditions, mortality rises more than linearly with age. (One could implement an explicit life table instead, of course, as in section 16.2.)

The S-I-R model dynamics are implemented in the above code by getting a vector of all infectious individuals and looping over them. For each infectious individual, nearby susceptible individuals might become infected (with a probability that depends on their spatial proximity to the focal individual, using `strength()`). Here the `INFECTIVITY` parameter governs how many nearby individuals can be infected; one might want to scale the probability of infection instead. After infecting nearby individuals, the focal infectious individual might die or recover, using the defined probabilities for those state transitions.

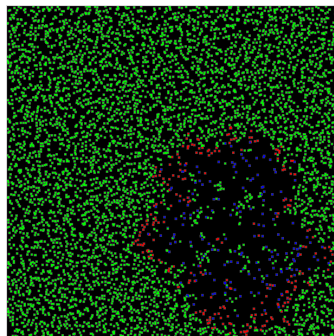
We have a few final behaviors to define. For more realistic spatial dynamics we want individuals to move around the landscape in each tick, changing which other individuals they are interacting with (this actually makes a substantial difference to the dynamics). We also want to color individuals according to their health status, so we can see the epidemic's state visually in SLiMgui. We do these things in a `late()` event:

```
late()
{
    inds = p1.individuals;

    // move around a bit
    for (ind in inds)
    {
        newPos = ind.spatialPosition + runif(2, -0.005, 0.005);
        ind.setSpatialPosition(p1.pointPeriodic(newPos));
    }

    // color according to health status; S=green, I=red, R=blue
    inds_tags = inds.tag;
    inds[inds_tags == HEALTH_S].color = "green";
    inds[inds_tags == HEALTH_I].color = "red";
    inds[inds_tags == HEALTH_R].color = "blue";
}
```

That's the whole model; it's a bit longer than most of the recipes in this manual, but really not so bad considering its complexity. When run in SLiMgui, we can see the strong spatial dynamics of the epidemic:

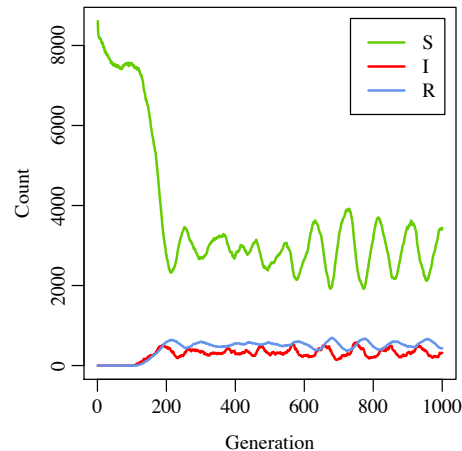


This snapshot is near the start of a run, as a wave of red infectious individuals spreads outward and blue recovered individuals exist behind that wavefront.

As the model runs, it produces a time series of S-I-R values like this:

```
..., 2470, 263, 607, 2501, 254, 611, 2539, 244, 614, 2585, 221, 610, 2603,  
216, 603, 2653, 207, 589, 2699, 206, 596, 2752, 189, 585, 2815, 185, 577,  
2860, 188, 572, 2910, 199, 568, 2956, 198, 558, ...
```

A little R scripting using these values gives us a plot of the epidemic's dynamics:



In this run the dynamics are strongly cyclical, likely as a consequence of the spatiality of the model (a conclusion that could be verified with a non-spatial version of the same model). The dynamics turn out to be pretty sensitive to the parameter values used; it is easy to find regimes where the epidemic sputters out after a short time due to insufficient infectivity, or where the population goes extinct.

Again I want to emphasize that this is quite a simple toy model; it has no effects of genetics, no spatial heterogeneity, only three individual states (S-I-R), no effects of age on infectivity or mortality (or even reproduction – individuals here are fertile from birth), and so forth. One could imagine extending it in all sorts of different ways; but with this foundation, such extensions ought to be straightforward to implement using the techniques shown in other sections.

A final note: one particularly interesting way in which this model could be extended is by making the infectious agent (let's call it a virus) evolve, rather than just the host individuals. It would not be easy to explicitly model the virus as individuals in SLiM, with explicit genetics. However, one could certainly track *properties* of the strain of virus carried by any given infectious individual – the strain's infectivity, virulence, etc. – using a value or values associated with the individual by `setValue()`. When that individual infected another individual, those strain properties would be carried over to the new individual, with random modifications representing mutation and evolution in the host. The strain values associated with an individual's virus could, with a little more scripting, actually influence the properties of the virus in the infected individual – modifying the probabilities of infection, mortality, and recovery, for example. In this way, one could easily model the evolution of the disease in response to different milieu in which it finds itself. Furthermore, if the properties of the virus were affected by the host's genetics as well – evolution toward decreased sociality when infected, say – one could model the co-evolution of host and disease. Since the virus's evolution would not be based upon explicit genetics, but rather just upon heritable and mutable strain properties, this would be less than fully realistic, but it would probably still be quite interesting.

## 16.19 Range expansion in a stepping-stone model in two recipes

Section 5.3.1 demonstrated a linear stepping-stone model using the WF framework. As a WF model, the population sizes were dictated as a top-down parameter, and so the model started with all of the “stepping stones” already at carrying capacity. It is often desirable to have a more dynamic population structure, which the nonWF model allows. Section 16.5 demonstrated this with a non-spatial metapopulation, with extinction and recolonization of subpopulations. In this section we’ll see how to implement a spatial stepping-stone model with that sort of dynamic structure. More specifically, we will look at range expansion through a series of initially empty subpopulations, seeded from a single subpopulation at the end of the “stepping stone” chain. This model will also differ from section 16.5 in using non-overlapping generations, just for variety, employing more or less the strategy of section 16.9. The most interesting contrast with section 16.5, however, is that that recipe moved migrants one at a time, rather inefficiently, whereas this recipe shows how to pre-plan and then execute a complex pattern of migration in bulk with the `takeMigrants()` method. An alternative recipe will then be presented, showing how to implement the same migration behavior using a `survival()` callback instead of `takeMigrants()`.

Here is the model except for the `early()` event that handles migration and population regulation, and an additional output event we will see later:

```
initialize() {
  defineConstant("K", 1000); // carrying capacity per subpop
  defineConstant("N", 10);   // number of subpopulations
  defineConstant("M", 0.01); // migration probability
  defineConstant("R", 1.04); // mean reproduction (as first parent)

  initializeSLIMModelType("nonWF");
  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
reproduction() {
  // individuals reproduce locally, without dispersal
  litterSize = rpois(1, R);

  for (i in seqLen(litterSize))
  {
    // generate each offspring with an independently drawn mate
    mate = subpop.sampleIndividuals(1, exclude=individual);
    if (mate.size())
      subpop.addCrossed(individual, mate);
  }
}
1 early() {
  // create an initial population of 100 individuals, the rest empty
  for (i in seqLen(N))
    sim.addSubpop(i, (i == 0) ? 100 else 0);
}
1001 late() { sim.outputFixedMutations(); }
```

The `1 early()` event sets up `N` subpopulations, all of which are empty except the first; setting them all up ahead of time simplifies the logic in the rest of the model, since we can just assume that all `N` subpopulations exist. We set up the numbering to start with `p0`, for simplicity when using the zero-based indexing of Eidos. The `reproduction()` callback reproduces the focal individual by

choosing a litter size, based upon a Poisson draw with a mean of  $R$ ; many individuals will have one offspring, but many will have zero or two, and occasionally an individual will get lucky and have three or more offspring. Note that the mean reproductive output  $R$  is from the perspective of the focal individual, the first parent; individuals will also often be chosen as a mate, and generate additional offspring in this way, but that is above and beyond their “own” reproduction. In this model, each offspring is generated with a (probably) different mate, independently drawn, rather than implementing monogamy. Selfing is not allowed, using the `exclude=` parameter of `sampleIndividuals()`, even though this is a hermaphroditic model, which makes colonization significantly more difficult; a new colony must be seeded with at least two individuals to have any chance of establishment.

Before this model can run, it needs some population regulation, as well as migration:

```
early() {
  inds = sim.subpopulations.individuals;

  // non-overlapping generations; kill off the parental generation
  ages = inds.age;
  inds[ages > 0].fitnessScaling = 0.0;
  inds = inds[ages == 0];

  // pre-plan migration of individuals to adjacent subpops
  numMigrants = rbinom(1, inds.size(), M);

  if (numMigrants)
  {
    migrants = sample(inds, numMigrants);
    currentSubpopID = migrants.subpopulation.id;
    displacement = -1 + rbinom(migrants.size(), 1, 0.5) * 2; // -1 or +1
    newSubpopID = currentSubpopID + displacement;
    actuallyMoving = (newSubpopID >= 0) & (newSubpopID < N);

    if (sum(actuallyMoving))
    {
      migrants = migrants[actuallyMoving];
      newSubpopID = newSubpopID[actuallyMoving];

      // do the pre-planned moves into each subpop in bulk
      for (subpop in sim.subpopulations)
        subpop.takeMigrants(migrants[newSubpopID == subpop.id]);
    }
  }

  // post-migration density-dependent fitness for each subpop
  for (subpop in sim.subpopulations)
  {
    juvenileCount = sum(subpop.individuals.age == 0);
    subpop.fitnessScaling = K / juvenileCount;
  }
}
```

This has three parts. The first part just implements non-overlapping generations by killing every individual with an age greater than zero, by setting their `fitnessScaling` to zero. At the end of this part, the script refocuses the variable `inds` down to just the survivors of that cull: the juveniles.

The second part handles migration. The logic here is vectorized for efficiency, so it is a bit subtle. First, we decide how many individuals will attempt to migrate; this is a single binomial

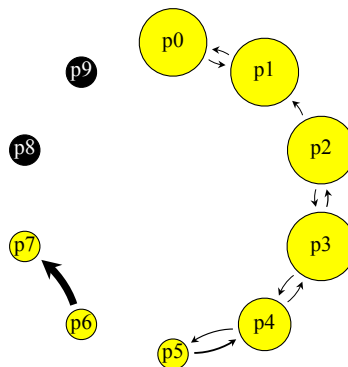
draw with a probability of  $M$  for each “trial” – each individual. If at least one individual is migrating, we proceed by drawing the actual migrants using `sample()`, without replacement. The subpopulation of all the migrants is obtained from the `subpopulation` property of `Individual`, and then we get the `id` of those subpopulations so that we can work with the subpopulations as integer values. Next we decide where each migrant wants to move, by drawing a  $-1$  or  $+1$  displacement for each migrant using `rbinom()`, and adding that displacement to each migrant’s current subpopulation `id`.

The next step is probably the trickiest bit. If an individual is in `p0` and wants to move “leftward”, to an `id` of  $-1$ , or if an individual is in `p9` and wants to move “rightward”, to an `id` of `10`, we want to prohibit those moves; those individuals just won’t migrate after all. We calculate, for each migrant, whether they are actually moving or not, based on this heuristic. If we’re left with any individuals that are actually moving, we want to refocus our efforts down onto that subset, and forget about the individuals that aren’t moving after all. We do that by redefining `migrants` and `newSubpopID` as being the subset of their old values where `actuallyMoving` is `T`. (If this is unclear, note that this sort of vectorized logic is covered in more detail in the Eidos manual, and in the online SLiM workshop.)

Now we know exactly who is moving, and where they are moving to. We implement our pre-planned migration with a `for` loop through the subpopulations, moving all individuals with a destination of `subpop` at one time with a single call to `takeMigrants()`. This is one virtue of pre-planning the migration before executing it; it allows much more efficient migration, since a single vectorized call to `takeMigrants()` is much faster than separate calls for each individual being moved. (The overhead for each separate call to `takeMigrants()` is actually fairly large, for internal implementation reasons; moving ten individuals one at a time actually takes much more than ten times as long as moving all ten at once.) The other virtue of pre-planning the migration is that individuals can only move one “step” per tick; if, instead, we chose and moved migrants from `p0`, then chose and moved migrants from `p1`, etc., a lucky individual might manage to be chosen from `p0` and moved to `p1`, then chosen from `p1` and moved to `p2`, all in a single tick – violating our intended migration dynamics. In general, pre-planning is almost always the best design for migration in nonWF models, to avoid such issues.

Finally, the last part of the `early()` event implements density-dependent fitness. This should be familiar; the only thing to note is that the density-dependence implemented here depends upon the final number of individuals in each subpopulation, after migration, and it is based only on the number of juveniles; the parental generation is, effectively, already gone.

The recipe is not quite done yet, but we can recycle and run it, and the range expansion dynamics will proceed as intended. Since  $R$  and  $M$  are both fairly small, and two individuals are needed to colonize, expansion is fairly slow. Here’s a plot from SLiMgui’s “Population Visualization” graph, showing the population structure a little after 200 ticks in:





The expansion has reached as far as p7, but several of the subpopulations at the end of the chain are still relatively small in size. By chance, p6 is delivering an unusually large fraction of migrants into p7 in this tick.

This visualization helps to reassure that the model is working correctly, but for analysis it would be helpful to have quantitative metrics regarding the progress of the range expansion over time. To do this, we will leverage the `LogFile` class, a new feature in SLiM 3.5 that was introduced in section 4.2.5. We can add a new `late()` event to create and configure a new `LogFile`:

```
1 late() {
  // set up a log file
  log = community.createLogFile("sim_log.txt", sep="\t",
    logInterval=10);
  log.addCycle();
  log.addPopulationSize();
  log.addMeanSDColumns("size", "sim.subpopulations.individualCount;");
  log.addCustomColumn("pop_migrants",
    "sum(sim.subpopulations.individuals.migrant);");
  log.addMeanSDColumns("migrants",
    "apply(sim.subpopulations, 'sum(applyValue.individuals.migrant);')");
}
```

The log file will be a TSV (tab-separated values) file, since we requested a tab separator with `sep="\t"`, and new rows will be appended every 10 ticks automatically. Columns will be in the order configured here: first the cycle number, then the overall population size, and then several custom columns. The `addMeanSDColumns()` method takes a lambda – a string containing Eidos source code – that is expected to return a vector of values, and it appends columns containing the mean and standard deviation of that vector. Our first use of it here will provide the mean and standard deviation of the sizes of the subpopulations in the model. Then we use `addCustomColumn()` to add a single column containing the total number of migrants in the population, calculated with another lambda. Finally, we add columns with the mean and standard deviation of the number of migrants in each subpopulation, so we can get a sense of how the migrant fraction of the subpopulations varies. With a few lines of code, we now have automatic logging of some fairly sophisticated simulation metrics.

Running the model with this logging produces an output file like this:

cycle	num_individuals	size_mean	size_sd	pop_migrants	migrants_mean	migrants_sd
1	100	10	31.273	1	0.1	0.316228
11	149	14.9	46.4195	2	0.2	0.632456
21	207	20.7	64.066	1	0.1	0.316228
...						
381	9218	921.8	238.048	96	9.6	4.24788
391	9400	940	176.237	82	8.2	2.78089
401	9515	951.5	142.741	72	7.2	3.01109
411	9897	989.7	31.4503	108	10.8	3.58391
421	9995	999.5	4.08928	78	7.8	2.25093
431	10023	1002.3	9.60382	93	9.3	2.90784
441	9984	998.4	10.0907	89	8.9	3.95671
451	10024	1002.4	7.74884	87	8.7	3.59166
...						
1001	9991	999.1	8.5173	97	9.7	3.23351

These metrics make it clear that the range expansion has essentially ended by tick 421. At the end, subpopulations are receiving an average of about 9 migrants per tick, and their size has equilibrated at our  $K$  of 1000 as intended. We could now add in some non-neutral dynamics, with perhaps some additional `LogFile` columns measuring things like mutation frequencies, mean fitness, or  $F_{ST}$  between different population pairs, perhaps leveraging SLiM's population-genetics utility functions documented in section 24.17.2 – also new in SLiM 3.5.

With SLiM 3.7 there is an alternative way to implement migration in nonWF models, using a `survival()` callback (see section 25.10). These callbacks, as their name suggests, control survival during the mortality phase of the tick cycle; this use of `survival()` callbacks was shown in section 16.15, for example. However, they can *also* be used to move individuals between subpopulations during the mortality phase. This extension makes sense, philosophically, if death is regarded as simply the movement of an individual to a different place – the “land of the dead” in mythology, such as Niflheim, Mictlan, etc. Section 16.22 will present a model that does this quite literally, storing dead individuals in a “cold storage” subpopulation for later reference. If death is just a special type of migration, then allowing other migration to also occur during SLiM’s mortality phase makes eminent sense! In any case, more pragmatically, SLiM offers this feature because it provides a convenient alternative mechanism for implementing migration that can be simpler than the pre-planning strategy necessary with `takeMigrants()` shown above. A `survival()` callback is called once for each focal individual in the model, so there is no need for pre-planning to prevent an individual from being moved more than once in a given tick.

Most of the script for this alternative recipe is the same as presented above, and will not be shown again here (the complete recipe is available in SLiMgui’s File menu as usual). The `early()` event no longer has any `takeMigrants()` migration code:

```
early() {
    // non-overlapping generations; kill off the parental generation
    inds = sim.subpopulations.individuals;
    sim.killIndividuals(inds[inds.age > 0]);

    // pre-migration density-dependent fitness for each subpop
    for (subpop in sim.subpopulations)
        subpop.fitnessScaling = K / subpop.individualCount;
}
```

This alternative recipe also takes a different approach to implementing non-overlapping generations, using `killIndividuals()` instead of setting the `fitnessScaling` of the parental generation to 0.0. We last saw `killIndividuals()` in section 16.5; it can be a convenient way to immediately kill individuals without waiting for the viability/survival tick cycle stage to happen. Notice how the immediacy of `killIndividuals()` results in a simplification of the density-dependence code, too; compare it to the same code in the previous version of the recipe, which had to carefully exclude the parental generation from the density-dependence.

And then we add a `survival()` callback that implements the new migration scheme:

```
survival() {
    // honor SLiM's survival decision
    if (!surviving)
        return NULL;

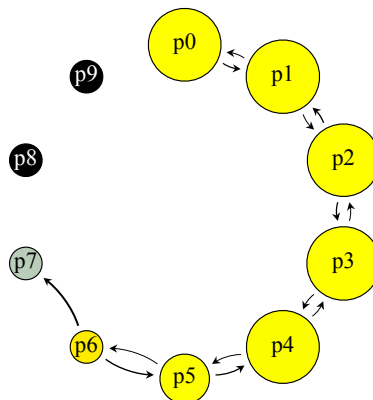
    // migrate with probability M
    if (runif(1) >= M)
        return NULL;

    // migrate the focal individual to an adjacent subpop
    subpops = sim.subpopulations;
    newSubpopID = subpop.id + (-1 + rbinom(1, 1, 0.5) * 2); // -1 or +1
    newSubpop = subpops[subpops.id == newSubpopID];
    if (newSubpop.size())
        return newSubpop;
    return NULL;
}
```

The first couple of lines simply return `NULL`, telling SLiM to go ahead with its default survival decision for the focal individual, if the focal individual is slated to die (we don't want to interfere with that outcome). The next couple of lines similarly return `NULL` if a random draw indicates that the focal individual is not migrating (migration occurring with probability `M`). If we do not return `NULL` at either of these points, it means that the focal individual is attempting to migrate, so the rest of the `survival()` logic handles the migration behavior. It begins by calculating the new subpopulation `id` that the migrant will move to (+1 or -1 from its current position) and looking up the subpopulation with that `id`. That lookup can fail, if an individual in subpop 0 tries to move in the -1 direction or an individual in subpop `N-1` tries to move in the +1 direction; in those cases, `NULL` is returned to leave the focal individual where it is. Otherwise, the callback returns the new subpopulation for the focal individual, and SLiM will move the individual for us.

There is one key difference between these two recipes that has been glossed over until now: the timing of migration with respect to selection. In the first version of the recipe, the `early()` event moved individuals first, and then imposed density-dependent fitness upon individuals in their new subpopulations; a comment in the `early()` event thus read "post-migration density-dependent fitness". In this new version of the model, the `early()` event imposes density-dependent fitness upon individuals in their original subpopulations, and then the `survival()` callback moves individuals to their new subpopulations after the selection decisions have, in effect, already been made (in actual fact the two happen simultaneously, during the mortality phase, but the probability of survival for each individual is based upon the `fitnessScaling` value calculated by the `early()` event prior to migration). For this reason, the comment in the `early()` event was changed to read "pre-migration density-dependent fitness" above. It would be easy to modify the first recipe to do pre-migration density-dependence; `takeMigrants()` can be called at almost any time, so the timing of migration is very flexible with that approach. It would be difficult to modify the second recipe to do post-migration density-dependence, however, since the migration is occurring at the same time as mortality decisions are being made, and so the final post-migration density of each subpopulation is not yet known when the survival decision for a given individual needs to be made. The `survival()` callback approach to migration is therefore less flexible than the `takeMigrants()` approach; however, it can still be very useful!

We can confirm that the model still works as intended with another plot:



This snapshot, again taken a little after tick 200, shows that the migrational dynamics of the model are working much as they did before; the timing of migration in the tick cycle did change, as discussed above, but that makes little difference to this particular model.

## 16.20 Logistic population growth with the Beverton–Holt model

In this section we will look at three different methods of regulating population size, using three different subpopulations in a single model without migration. For simplicity and comparability, all three methods will use non-overlapping generations. We will compare their results and discuss their differences.

The non-spatial nonWF models presented in this manual (such as in section 16.1) generally use a very simple density-dependence equation for the regulation of population size, with a form like this (we will call this the “simple model” in our discussion below):

```
p1.fitnessScaling = K / p1.individualCount;
```

This scales the fitness of the population so that if there are at least  $K$  individuals *before* viability selection, the population size *after* viability selection will be  $K$  (on average, with stochastic variation). In other words, it simply enforces a carrying capacity of  $K$ . If there are fewer than  $K$  individuals, then the `fitnessScaling` value for the population will be greater than  $1.0$ , and so every individual will survive (for otherwise neutral models), which allows the population size to grow exponentially as long as the reproduction rate is above zero. As discussed in section 16.4,  $K$  will be the actual equilibrium population size for the model only if the model is neutral; deleterious mutations will decrease mean fitness and cause equilibration below  $K$ , whereas beneficial mutations will increase mean fitness and cause equilibration above  $K$ , which often makes sense from a biological perspective. The “simple model” is therefore a nicely individual-based and emergent model that encapsulates absolute fitness and hard selection, in general.

The “simple model” can easily be adapted to non-overlapping generations, as for example in section 16.9, by killing off the parental generation and then applying the “simple model” just to the offspring:

```
inds = p1.individuals;
inds[inds.age > 0].fitnessScaling = 0;
n_t_plus_pt5 = sum(inds.age == 0);
p1.fitnessScaling = K / n_t_plus_pt5;
```

Here we use the variable name `n_t_plus_pt5` for the number of juveniles produced by reproduction, for reasons that will become more clear below. The `fitnessScaling` for the juveniles depends only upon the number of juveniles, `n_t_plus_pt5`, since the parental generation has, conceptually, already died. This will be the **first** of the three population-regulation models we will examine in this section.

However, in some respects the “simple model” is not ideal. In particular, it generates exponential growth in population size until  $K$  is reached, whereas for many biological systems logistic growth may be more realistic. For continuous-time models such as the Lotka–Volterra model, the logistic equation itself is often used; for discrete-time models such as those in SLiM, the Beverton–Holt model is often used to provide similar logistic population growth. The Beverton–Holt equation looks like this:

$$n_{t+1} = \frac{R n_t}{1 + n_t / M}$$

The population size at the next timestep,  $n_{t+1}$ , is a function of its size at the current time,  $n_t$ , the proliferation rate per generation,  $R$  (often called  $R_0$ , but we will ignore that distinction here for simplicity) and the model parameter  $M$ , where the carrying capacity is  $K = (R - 1)M$ .

One key point to notice here is that, philosophically, the Beverton–Holt model is fundamentally different from the “simple model”, because it includes both birth and death in a single equation; the new population size is expressed as a function of the old population size times the proliferation rate (birth), limited by the carrying capacity (death). In the “simple model”, on the other hand, we assume that reproduction has already happened, and we are interested solely in the probability of survival for the individuals now alive. In this sense, the Beverton–Holt model could be viewed as a model of density-dependent *fecundity*, rather than a model of density-dependent *survival*; it is, in a sense, scaling the reproduction rate of the population according to population density. That can be biologically realistic; there are certainly species that vary their fecundity with density in such a manner. One could write this perspective into a SLiM nonWF model, too; one could write a `reproduction()` callback that would generate offspring at an appropriate rate to obey the Beverton–Holt equation (with non-overlapping generations, for our purposes here), and then not have any density-dependence at the viability/survival tick cycle stage at all. This will be the **second** of the three population-regulation models we will examine. (Section 16.15 provides a similar approach for the “simple model” as an expression of fecundity rather than survival; this is similar to the Wright–Fisher model, with reproduction up to  $K$  regardless of any intrinsic reproduction rate  $R$ .)

But one *can* also view the Beverton–Holt equation as an expression of density-dependent survival; it’s just that from this perspective it is (I think, and will argue) a bit odd. Viewed in this way, there are three population sizes that we need to be concerned with, not just two: we have  $n_t$ , the parental population size at time  $t$  before reproduction, we have what I will call  $n_{t+0.5}$ , the number of juveniles produced by reproduction (equal to  $Rn_t$ ), and we have  $n_{t+1}$ , the number of surviving juveniles after mortality, which will be the pre-reproduction parental individuals for the next generation. In SLiM, we will move from  $n_t$  to  $n_{t+0.5}$  by reproduction at rate  $R$ , just as in the first model. We will then move from  $n_{t+0.5}$  to  $n_{t+1}$  by mortality due to some mean population fitness, calculated based upon the Beverton–Holt model. To figure out how to do the latter move, let’s rewrite the standard Beverton–Holt equation above, substituting  $n_{t+0.5}$  in place of  $n_t$  using the fact that  $n_{t+0.5} = Rn_t$ . The  $n_t$  in the denominator of the original equation becomes  $n_{t+0.5}/R$  as a result; note that by dividing  $n_{t+0.5}$  by  $R$  we are effectively *back-estimating* the original, pre-reproduction population size  $n_t$  from the post-reproduction size  $n_{t+0.5}$  and  $R$ . This is necessary because conceptually, once we are at the juvenile-mortality phase of the tick cycle we no longer “remember” what the original pre-reproduction population size was; all we know (or all we ought to pretend we know, for biological realism) is the current population size after the parental generation has died, which is  $n_{t+0.5}$ . With that substitution, we then have:

$$n_{t+1} = \frac{n_{t+0.5}}{1 + n_{t+0.5}/RM}$$

This is the form of the equation that makes sense for use in SLiM, because we can now calculate the proper `fitnessScaling` value as  $n_{t+1}/n_{t+0.5}$  by dividing both sides by  $n_{t+0.5}$ . This will be the **third** of the three population-regulation models we will examine.

The oddity of this is that we can’t really excise  $R$  from the model by redefining our terms in this way and thinking only in terms of the post-reproduction number of juveniles; when we do so, we then have to use  $R$  to back-estimate the pre-reproduction size  $n_t$  from  $n_{t+0.5}$ . What this expresses is that the logistic growth modeled by the Beverton–Holt equation depends intrinsically upon the reproduction rate, even when it is modeled as mortality. For example, suppose the carrying capacity  $K$  is 1000, and the post-reproduction population size  $n_{t+0.5}$  is 500, and we want to know what the post-selection population size  $n_{t+1}$  ought to be. The answer, according to the Beverton–



Holt model, depends upon  $R$ ; the survival rate of the juveniles is not solely governed by the number of juveniles and the carrying capacity, as it is in the “simple model”, but also by the reproduction rate. If you think about what logistic population growth entails, you will see that it cannot be otherwise; and yet it is quite strange from an individual-based perspective in which one wishes to draw a clear conceptual separation between reproduction and survival, as SLiM does. This means that when we calculate a `fitnessScaling` value using the Beverton–Holt equation, we must include  $R$  (or  $M$ ) in our equation. We can certainly do that, as shown above; we just should not be surprised that we have to do so. This is another angle on the idea that the Beverton–Holt model is more naturally thought of as expressing density-dependent *fecundity* rather than density-dependent *survival*. This is important to understand, because if you want to use the Beverton–Holt equation in a model with more complex, individual-based reproduction, such as age-structured reproduction rates, you will need to think about this explicitly; you will either have to algebraically derive the theoretical reproduction rate  $R$  for your model, or you will have to use the pre-reproduction population size  $n_t$  and the current size  $n_{t+0.5}$  to calculate the realized reproduction rate  $R$ , even though the parental individuals, conceptually, no longer exist. One way or another, the Beverton–Holt model needs to know  $R$ . Note that if the rather philosophical discussion here is impenetrable – or if you disagree with my philosophical points! – that isn’t really important, as long as the derivation of the `fitnessScaling` value  $n_{t+1}/n_{t+0.5}$  that we will use to govern mortality is clear.

With those caveats, we have now described the three population regulation approaches that we would like to compare. Here’s the setup portion of the corresponding recipe:

```
initialize() {
    defineConstant("K", 50000);
    defineConstant("R", 1.1);
    defineConstant("M", K / (R - 1));

    initializeSLiMModelType("nonWF");
    initializeMutationType("m1", 0.5, "f", 0.0);
    m1.convertToSubstitution = T;
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 50); // the "simple model"
    sim.addSubpop("p2", 50); // Beverton–Holt influencing fecundity
    sim.addSubpop("p3", 50); // Beverton–Holt influencing survival

    log = community.createLogFile("sim_log.txt", logInterval=1);
    log.addCycle();
    log.addSubpopulationSize(p1);
    log.addSubpopulationSize(p2);
    log.addSubpopulationSize(p3);
}
```

The initialization is boilerplate, except that we define the constants  $K$ ,  $R$ , and  $M$  that we will need for our equations. The `1 early()` event sets up our three subpopulations, each with an initial size of 50. It also creates and configures a new `LogFile` object that will log out the size of each subpopulation in each tick; see section 4.2.6 for an introduction to the `LogFile` class.

Next, we have three separate `reproduction()` callbacks, one for each of our three subpopulations:



```

reproduction(p1) {
  // p1 simply reproduces with a mean litter size of R, like p3
  litterSize = rpois(1, R);
  for (i in seqLen(litterSize))
    subpop.addCrossed(individual, subpop.sampleIndividuals(1));
}
reproduction(p2) {
  // p2 reproduces up to the Beverton–Holt equation's target
  n_t = subpop.individualCount;
  n_t_plus_1 = (R * n_t) / (1 + n_t / M);
  mean_litter_size = n_t_plus_1 / n_t;
  litterSize = rpois(1, mean_litter_size);

  for (i in seqLen(litterSize))
    subpop.addCrossed(individual, subpop.sampleIndividuals(1));
}
reproduction(p3) {
  // p3 simply reproduces with a mean litter size of R, like p1
  litterSize = rpois(1, R);
  for (i in seqLen(litterSize))
    subpop.addCrossed(individual, subpop.sampleIndividuals(1));
}

```

The p1 and p3 callbacks implement identical individual-based reproduction logic: choosing a litter size from a Poisson distribution with mean  $R$ , and generating that litter size with a randomly chosen mate for each offspring. This provides population growth at (approximately) the desired proliferation rate  $R$ . For p2, we adopt a different strategy, because this is the subpopulation using the Beverton–Holt equation as a model of fecundity. Here, reproduction is density-dependent; individuals have fewer offspring as density increases, and  $R$  expresses the *maximum* proliferation rate, attained only at low density. To model this, we use the Beverton–Holt equation to calculate the number of offspring,  $n_{t+1}$ , that we ought to generate given the parental generation size  $n_t$ . Then we calculate what our *true* reproduction rate,  $n_{t+1}/n_t$ , ought to be, and we use that as the mean litter size to reproduce the focal individual. We draw the actual litter size from a Poisson distribution with mean `mean_litter_size`, and the rest of the reproduction code is the same as for the other subpopulations. This is a sort of stochastic, individual-based version of the Beverton–Holt model; the population doesn't always produce exactly  $n_{t+1}$  offspring, but each individual's reproductive behavior is such that the average number of offspring in the population will be  $n_{t+1}$ .

Finally, we implement non-overlapping generations and density-dependence:

```

:200 early() {
  // p1 uses the "simple model" with non-overlapping generations
  inds = p1.individuals;
  inds[inds.age > 0].fitnessScaling = 0;
  n_t_plus_pt5 = sum(inds.age == 0);
  p1.fitnessScaling = K / n_t_plus_pt5;

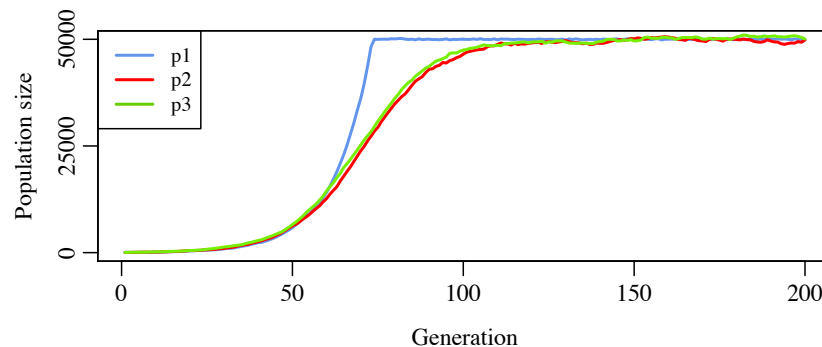
  // p2 has selection only to achieve non-overlapping generations
  inds = p2.individuals;
  inds[inds.age > 0].fitnessScaling = 0;

  // p3 uses the Beverton–Holt equation for survival
  inds = p3.individuals;
  inds[inds.age > 0].fitnessScaling = 0;
  n_t_plus_pt5 = sum(inds.age == 0);
  p3.fitnessScaling = 1 / (1 + (n_t_plus_pt5 / R) / M);
}

```

For p1, we kill the parental generation and then implement the “simple model” for density-dependence, just as shown at the beginning of this section (and now it should be clear why we used the symbol  $n_{t\_plus\_pt5}$  for the number of juveniles, since we called that quantity  $n_{t+0.5}$  in the discussion above). For p2, all we need is to kill the parental generation; we implemented the Beverton–Holt equation in p2’s `reproduction()` callback. For p3, we kill the parental generation and then apply a `fitnessScaling` for the Beverton–Holt model,  $n_{t+1}/n_{t+0.5}$ , as derived above for this third population model.

The output from `LogFile` allows us to easily plot the behavior of the three models in R:



The logistic curves of the two Beverton–Holt models are clearly different from the exponential curve of the “simple model”. The two Beverton–Holt models, on the other hand, perform quite similarly. All three models are subject to stochastic variation, and a particularly low reproduction rate early on can have a long-lasting effect of population growth; indeed, extinction is even possible for all three models. Here, however, we avoided such unlucky outcomes.

## 16.21 Dynamic population structure in nonWF models

Section 5.2 showed how to handle dynamic population structure in WF models: adding and removing subpopulations, and also the more complex operations of splitting and joining subpopulations. We have seen some types of dynamic population structure in previous nonWF recipes, such as the metapopulation extinction-colonization dynamics of section 16.5 and the range expansion scenario of section 16.19. Those models essentially involved moving individuals around between pre-existing subpopulations, using the `takeMigrants()` method. However, we haven’t yet seen how to do those more complex operations: splitting and joining subpopulations. We’ll cover that here; however, in a nonWF model, since we are responsible for explicitly making all decisions for our modeled individuals, we have to decide exactly how those operations work. The first thing we have to do, then, is figure out *exactly* what we want our model to do.

For splitting a subpopulation, there are two obvious choices. One option would be to create a new zero-size subpopulation and move individuals into it from an existing subpopulation using `takeMigrants()`. It’s pretty obvious how to implement this – just call `addSubpop()` with a size of 0, choose target individuals to move, and move them. We’ve seen a similar approach in those previous sections, so we won’t dwell on that here. The other option would be to create a new zero-size subpopulation and generate new individuals for it in the next generation, filling it up with new juvenile individuals that were generated from parents in the other subpopulations. It is less obvious how to implement that, so that is what we will do here. The two are conceptually somewhat different; the first might resemble some kind of vicariance event that splits an existing subpopulation in two, whereas the second might represent a species that always has excess juvenile production, and in some kind of chance long-distance dispersal event some of those excess juveniles land in a new, unoccupied habitat and colonize it. Which approach makes sense

for your simulation is a question of the real-world biology you are trying to model, but as I wrote above, in this section we will implement the second approach.

For joining a subpopulation, there are similarly two obvious choices. One option would be to simply move all of the individuals from one subpopulation into the other with `takeMigrants()`, and then remove the now-empty subpopulation from the model. All of the individuals, whatever their ages, sexes, habitat preferences, etc., will now be together, and the next time reproduction occurs you will start to generate hybrids. The original subpopulations will be lost; only the joined subpopulation will now exist. Again, this approach should be very easy to implement from what we saw in previous recipes using `takeMigrants()`, so we will not dwell on it here. The other option would be to create a new subpopulation with `addSubpop()` of size 0, and generate new juveniles into it that are the result of hybridization between the two original subpopulations. This gives us much greater control. We can make the original subpopulations dwindle away over time as the new hybrid subpopulation grows, or we can keep them around in parallel with the new hybrid subpopulation. We can also control which individuals mate with which during the production of hybrids, allowing any process of hybridization we want (assortative or disassortative mating, age or sex biases in the choice of parents, and so forth). It is again less obvious how to implement this type of population admixture, so we will focus on it here.

To keep things simple, we will implement a very basic scenario for these events: starting with two subpopulations, joining them in a single tick, then splitting off a new subpopulation in a single tick. We will model only neutral mutations, no migration except for the split and join events, and emergent overlapping generations and age structure. The techniques seen in other recipes for age structure, migration, etc., could be added.

Here's the start of the recipe:

```
initialize() {
  initializeSLiMModelType("nonWF");
  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;

  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
reproduction() {
  // each subpopulation reproduces within itself
  subpop.addCrossed(individual, subpop.sampleIndividuals(1));
}
1 early() {
  // start with two subpops that grow to different sizes
  sim.addSubpop("p1", 10).setValue("K", 500);
  sim.addSubpop("p2", 10).setValue("K", 600);
}
early() {
  // density-dependent regulation for each subpop
  for (subpop in sim.subpopulations) {
    K = subpop.getValue("K");
    subpop.fitnessScaling = K / subpop.individualCount;
  }
}
5000 late() { }
```

In most respects this is unsurprising. We create two subpopulations, p1 and p2; a single `reproduction()` callback is used to reproduce both of them (and any other subpopulations that

might exist later on); and we implement density-dependent population regulation for each subpopulation. The only unusual thing here is that we keep the carrying capacity value for each subpopulation, *K*, attached to the subpopulation itself using `setValue()` and `getValue()`; this makes it easy to write the density-dependence code as a simple loop over all existing subpopulations, regulating each in turn. Note that we give *p1* and *p2* different sizes, just because we can. The model runs to tick 5000.

Next, let's implement the join event. We want *p1* and *p2* to merge together to form *p3* in tick 1000, not by moving individuals with `takeMigrants()` – which we certainly could do – but by generating new hybrid offspring and placing them into *p3*. To make things a bit more interesting, we will choose the *first* parent of each offspring randomly from the pool of available individuals (slightly favoring *p2*, note, since it has a larger number of individuals due to its larger carrying capacity), but we will choose the *second* parent of each offspring with a large bias toward *p2*. This is a hermaphroditic model, but the same basic technique could be used to create a bias toward one source subpopulation specifically for the male parents in each mating, for example. The implementation looks like this:

```

999 late() {
    // create a zero-size subpop for the join
    sim.addSubpop("p3", 0).setValue("K", 750);
}
1000 reproduction() {
    // generate juveniles to seed p3
    founderCount = rdunif(1, 10, 20);
    p1_inds = p1.individuals;
    p2_inds = p2.individuals;
    all_inds = c(p1_inds, p2_inds);

    for (i in seqLen(founderCount))
    {
        // select a first parent with equal probabilities
        parent1 = sample(all_inds, 1);

        // select a second parent with a bias toward p2
        if (rdunif(1) < 0.2)
            parent2 = sample(p1_inds, 1);
        else
            parent2 = sample(p2_inds, 1);

        // generate the offspring into p3
        p3.addCrossed(parent1, parent2);
    }

    // we're done, don't run again this tick
    self.active = 0;
}
1000 early() {
    // get rid of p1 and p2 now
    c(p1,p2).fitnessScaling = 0.0;
}

```

We start off in tick 999 by creating *p3* ahead of time. This is necessary because SLiM does not allow us to call `addSubpop()` from inside a `reproduction()` callback; new subpopulations, into which we want to generate offspring, must be created in advance.

Then, in tick 1000, we define a new `reproduction()` callback. This creates all of the new offspring to fill *p3*, from parents chosen from *p1* and *p2* as described above. It does this in one “big

bang” and then disables itself by setting `self.active` to `0`, so that it runs only a single time; we have seen this strategy in various recipes before, such as those in sections 16.3 and 16.15. The workings of this callback should be fairly obvious. The important point to notice here is that you are allowed to define more than one `reproduction()` callback in a model; SLiM will run all of the callbacks that are enabled for a given tick, and each callback can do its own reproduction. For this reason, the generic `reproduction()` callback we defined before will run as usual in tick `1000`, and will reproduce `p1` and `p2` as usual; in tick `1000` the `reproduction()` callback above will run in *addition*, and will generate *additional* offspring that will go into `p3`. Biologically, this might represent surplus offspring that were unmodeled in other ticks (since the surplus individuals would just die as a result of density-dependence anyway), but that in tick `1000` happen to find their way to a new habitat due to some chance event.

After reproduction has finished, `early()` events run, and the `1000 early()` event here just empties out `p1` and `p2` by setting their `fitnessScaling` value to `0`. All individuals in those populations will subsequently die, since their fitness will be `0`. This approach keeps `p1` and `p2` in existence, but empty. (We could call `removeSubpopulation()` on them instead, to actually remove them, but in SLiM 3.5 there was a bug that caused that approach to throw an error during SLiM’s mutation tallying, so let’s avoid that.)

Finally, we want to split off a new subpopulation, `p4`, from `p3` in tick `2000`. Rather than doing this by moving founders from `p3` to `p4` with `takeMigrants()` – which we certainly could do – we want to generate new founders for `p4` from parents in `p3`. The implementation for that looks like this:

```
1999 late() {
    // create a zero-size subpop for the split
    sim.addSubpop("p4", 0).setValue("K", 100);
}
2000 reproduction() {
    // generate juveniles to seed p4
    founderCount = rdunif(1, 10, 20);
    all_inds = p3.individuals;

    for (i in seqLen(founderCount))
    {
        // select parent1/parent2 with equal probabilities
        parent1 = sample(all_inds, 1);
        parent2 = sample(all_inds, 1);

        // generate the offspring into p4
        p4.addCrossed(parent1, parent2);
    }

    // we're done, don't run again this tick
    self.active = 0;
}
```

The approach here should seem very familiar; it is quite parallel to the subpopulation joining code we saw before. We create `p4` in tick `1999` with a size of `0`, and give it its carrying capacity. Then in `2000` a custom `reproduction()` callback selects parents from `p3` (with no bias, in this case) and generates offspring into `p4`. Again, our generic `reproduction()` callback will run in addition to this, reproducing `p3` as usual; here we are generating offspring in *addition* to that. We allow `p3` to continue to exist in this case, since we’re modeling a split instead of a join.

That’s all there is to it; however, the strategies shown here can easily be generalized. You could select parents with whatever criteria you wish, to generate founding juveniles with whatever

parental bias you desire in terms of age, sex, genetics, or other traits. You could do an  $n$ -way split or an  $n$ -way join, instead of the 2-way split and 2-way join shown here, selecting parents from the source populations in whatever proportions you desire. And all of this can be generalized beyond the concept of splits and joins; you can write a `recombination()` callback to generate offspring in any way you wish, from any parents you choose, into any destination subpopulations. As always, the rule should be: start with the biology you want to model, and then ask “What’s the best way to model this biology?” Whether your biological system involves seasonal migration of adults or one-time dispersal of juveniles, whether it includes monogamous mating or broadcast spawning, whether mating is assortative or random, etc., should dictate the structure of the reproduction code you write. Just about any reproductive strategy can be implemented in a nonWF model.

## 16.22 Sperm storage with a `survival()` callback

Sometimes it is desirable to put individuals into “cold storage” – keeping them around for later purposes, without having them act like normal living organisms in the model. One example of this is a “seed bank”, in which seeds wait to germinate at a later time; another is “resting eggs” that lie dormant until they hatch, such as exist in species like *Daphnia*. Seeds and resting eggs can be implemented by generating the individual that they would produce (the germinated plant, the hatched *Daphnia*) but keeping it in a cold storage subpopulation for an indeterminate length of time. When the seed is deemed to germinate, or the resting egg is deemed to hatch, the individual can be moved from cold storage into a regular subpopulation, assigned an age of zero, and handled normally thenceforth. Individuals in such cold storage do not reproduce, and may not die for a long time; they are essentially frozen in stasis, awaiting further instructions. This strategy is quite straightforward to implement, so we will not examine it further.

Another use for a “cold storage” subpopulation is to simulate sperm storage. Females of some species will mate with a male and store sperm from that male for a long period of time, using the stored sperm to fertilize eggs over time without needing to re-mate. The stored sperm may last much longer than the male that produced it, in fact; the male may continue siring progeny for quite some time after it has actually died, thanks to the storage of its sperm. Simulating this is less straightforward than implementing a seed bank or resting eggs, since males need to be kept in the simulation after they die – but not kept indefinitely, since the number of dead males would grow without bound. We need strategies for moving males into cold storage when they die, for finding them in cold storage when their sperm is used to generate new offspring, and for clearing them out of cold storage once there is no longer any living female with stored sperm from them.

To implement this recipe, we will use a couple of techniques that we have seen very little in this manual thus far. Most importantly, we will use `survival()` callbacks, documented in section 25.10. These callbacks govern the decisions made by SLiM during the nonWF tick cycle’s viability/survival selection stage (see chapter 23). Normally, in this stage, individuals live or die probabilistically, based upon their fitness. A `survival()` callback lets the model intervene in those choices, forcing particular individuals to live (by returning T) or die (by returning F) regardless of fitness. In addition, a `survival()` callback can actually move individuals; by returning a `Subpopulation` object, the callback can request that the individual ought to live, but ought to be moved to a particular subpopulation. This facility could conceivably be used to implement a general-purpose migration scheme in a nonWF model, without the use of `takeMigrants()`; here, we will use it to divert males that are slated for death into a cold storage subpopulation instead, so that they can continue to be used to sire progeny as a result of their previously stored sperm.

With no further ado, let’s plunge into the model. This recipe is based upon a model developed by Anita Lerch, with her kind permission; `survival()` callbacks were then added to SLiM specifically to better support the implementation of models like hers, as shown here.



Let's start with the less interesting boilerplate like initialization and output:

```
initialize() {
  initializeSLiMModelType("nonWF");
  initializeSLiMOptions(keepPedigrees=T);
  initializeSex("A");
  defineConstant("K", 500);

  initializeMutationRate(1e-7);
  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 100);
  plinds = p1.individuals;
  plinds.age = rdunif(size(p1.individuals), min=0, max=10);
  plinds.tag = -1;

  sim.addSubpop("p1000", 0); // cold storage for dead males
}
late() {
  catn(sim.cycle + ": p1 (" + p1.individualCount + ") " +
    ", p1000 (" + p1000.individualCount + ")");
}
10000 late() {
  sim.outputFixedMutations();
}
```

We turn on pedigree tracking, because we will use pedigree IDs to find males when we need to generate new offspring from their stored sperm. This is a sexual model, so we set that up with `initializeSex("A")`. It has overlapping generations and a minimum reproductive age, so we set initial ages for the first generation.

One notable thing in the above code is that we set the `tag` property for all initial individuals to `-1`. The `tag` value of females will be used in this model to indicate the male from which a female has stored sperm; if it is `-1`, the female has not stored sperm yet. (The `tag` values of males is unimportant at this point, and can be set to `-1` here as well for simplicity.) The other notable thing is that we create a special subpopulation, `p1000`, with an initial size of `0`; this is the “cold storage” subpopulation where we will put males when they die, to preserve their genetics as a source of stored sperm for the females.

Next, let's look at how males end up in cold storage. This is really the heart of this recipe, and it is done by a `survival()` callback:

```
survival(p1) {
  // move dying males into cold storage in case they have mated
  if (!surviving)
    if (individual.sex == "M")
      return p1000;
  return NULL;
}
```

This callback governs what happens to `p1` individuals during the viability/survival tick cycle stage; we declare it specifically as `survival(p1)` because we want the rules here to apply only to

p1. The callback is called once for each individual in p1, and makes a decision regarding that focal individual. Here, it first checks the surviving flag (a pseudo-parameter supplied to the callback by SLiM; see section 25.10), which indicates whether the focal individual would survive (T) or die (F) according to SLiM's standard fitness-driven machinery. If it is slated for death, the callback then checks whether the focal individual is male. If it is, it returns p1000; this requests that SLiM *not* kill the focal individual as it normally would, but instead move it to the p1000 cold storage subpopulation. In all other cases, the callback returns NULL; this tells SLiM to proceed with its default action, as if there were no survival() callback. The end result, then, is that mortality in p1 proceeds according to SLiM's standard fitness-driven behavior, except that when males die they get spirited off to p1000 instead of actually dying.

So far, so good. Now we need to write the code to govern female reproduction in p1 (there is no reproduction() callback in the model that applies to p1000, so the cold storage males do not reproduce at all). We do this in “big bang” fashion, getting all reproductive females (minAge=7), reproducing them all in a loop, and then using self.active = 0 to deactivate the callback, as we have seen in some previous models:

```
reproduction(p1) {
  matureFemales = subpop.subsetIndividuals(sex="F", minAge=7);

  for (female in matureFemales)
  {
    if (female.tag < 0) {
      // the female has not yet chosen a mate, so choose one now
      mate = subpop.sampleIndividuals(1, sex="M", minAge=7);
    } else {
      // the female has already chosen a mate; look it up by id
      mate = sim.individualsWithPedigreeIDs(female.tag);
    }
    if (mate.size()) {
      female.tag = mate.pedigreeID;
      litterSize = rpois(1, 5);
      for (i in seqLen(litterSize))
        subpop.addCrossed(female, mate);
    } else {
      catn(sim.cycle + ": No mate found for tag " + female.tag);
    }
  }
  self.active = 0;
}
```

For each female in the matureFemales vector, we need to deal with mate choice, stored sperm, and cold storage lookup, as well as actual offspring generation. We start by checking whether the female's tag is < 0, indicating that she has not yet mated; in that case we choose a mate, which must be a male of reproductive age. Otherwise – if the female has already mated – her tag value will be the pedigree ID of the male she mated with (and stored sperm from); we look up that male using the individualsWithPedigreeIDs() method, which provides convenient and fast lookup. Usually, the code will now have found the female's male mate (this can fail if the female is unmated and there are no males of reproductive age, so we need to check). Given that mate, we do a couple of things. First, we store the male's pedigreeID in the female's tag; this is, in effect, the moment when sperm storage occurs. Then we draw a litter size and generate offspring using addCrossed() in a loop, as we have seen in many other recipes. That completes the female's mating for this tick; but since this is an overlapping-generations model and females may live for

several ticks, the focal female may have an opportunity to reproduce again. If so, she will utilize the stored sperm from her mate rather than searching for a mate again.

We need to do a little book-keeping, in an `early()` event (which should be added *after* the existing 1 `early()` event for correct order of operations):

```
early() {
  // fix all new female tags; faster to do this vectorized
  offspringFemales = p1.subsetIndividuals(sex="F", maxAge=0);
  offspringFemales.tag = -1;

  // p1 is governed by standard density-dependence
  p1.fitnessScaling = K / p1.individualCount;

  // cold storage individuals are kept until unreferenced
  p1000.individuals.tag = 0;
  maleRefs = p1.subsetIndividuals(sex="F").tag;
  maleRefs = maleRefs[maleRefs != -1];
  referencedDeadMales = sim.individualsWithPedigreeIDs(maleRefs, p1000);
  referencedDeadMales.tag = 1;
}
```

This does three things. In the first section, we set the `tag` values of all newly generated female offspring to `-1`, to indicate that they are unmated. We could have done this in the `reproduction()` callback as we generated offspring one by one, or we could have done it in a `modifyChild()` callback, but it is simpler and more efficient to do it here in vectorized fashion. Since we do it in an `early()` event (running immediately after reproduction, in nonWF models), the `tag` values will be correctly set up before any other code uses them.

In the second section we just implement standard density-dependence for `p1`, based upon the defined carrying capacity `K`.

The third section is the most interesting bit; this is where we manage the cold storage subpopulation, purging males once they are unreferenced so that the size of the cold storage doesn't grow without bound. To do this, we need to figure out which males are unreferenced; the strategy employed is simple. First, we assume that all males in `p1000` are unreferenced, and set their `tag` property to `0` to indicate that. Next, we look up all females in `p1` and get their `tag` values; this is effectively a list of the males that are referenced. Unmated females will have a `tag` of `-1`, which we're not interested in, so we filter out those `-1` values from `maleRefs`. Next, we use the handy `individualsWithPedigreeIDs()` method to look up all of the referenced males, specifically in `p1000` (we do not want referenced males that are still alive, and thus still in `p1`). Finally, we set the `tag` of those referenced `p1000` males to `1`, overriding the value of `0` set previously and indicating that they are referenced.

The final piece of the puzzle is that we need to use the male `tag` values that we just set, to govern survival of `p1000` males. We do that with another `survival()` callback:

```
survival(p1000) {
  return (individual.tag == 1);
}
```

This callback, specific to `p1000`, simply looks at the `tag` value of the male. If it is `1`, that indicates that the male is referenced, so we return `T` to indicate that the male should survive. If it is not `1`, that indicates that the male is unreferenced, so we return `F` to indicate that the male should now die. (Conceptually, of course, it already died at the moment it was moved to cold storage; the point is that now we no longer need its genetics to generate offspring from its stored sperm.) This `survival()` callback therefore completely overrides SLiM's default fitness-based mortality. It

doesn't matter what mutations the males possess; they could even possess completely lethal mutations that make their fitness 0. Regardless, they will be kept until they become unreferenced by the females; and then, once unreferenced, they will die regardless of how many beneficial mutations they may possess. We have complete control over the fate of the p1000 males.

When this model is run, the output is unexciting, but does indicate that the cold storage is working:

```
1: p1 (100), p1000 (0)
2: p1 (218), p1000 (0)
3: p1 (348), p1000 (0)
4: p1 (504), p1000 (7)
5: p1 (505), p1000 (73)
6: p1 (488), p1000 (97)
7: p1 (482), p1000 (80)
8: p1 (509), p1000 (54)
9: p1 (495), p1000 (110)
10: p1 (502), p1000 (116)
11: p1 (491), p1000 (128)
12: p1 (515), p1000 (100)
13: p1 (481), p1000 (149)
14: p1 (497), p1000 (66)
15: p1 (512), p1000 (72)
...
9995: p1 (495), p1000 (121)
9996: p1 (492), p1000 (97)
9997: p1 (503), p1000 (94)
9998: p1 (505), p1000 (72)
9999: p1 (508), p1000 (91)
10000: p1 (499), p1000 (110)
```

The p1 subpopulation grows from its initial size of 100 to a dynamic equilibrium around 500, its carrying capacity, as expected. The p1000 cold storage subpopulation starts at size 0, and grows when males die and get moved to it, but since only referenced males are kept, its size stays bounded over time. We do not see the "No mate found for tag" log, indicating that all females are finding male mates and that cold storage lookup of males is working correctly.

Sperm storage can have more complex dynamics than what is implemented in this recipe. One wrinkle is that male-male competition can occur as a result of female choice: females can mate with multiple males, and either use the stored sperm from the male they deem best, or actually use stored sperm from several different males depending upon conditions. Another possibility is that male-male competition can result from sperm competition: a female that has mated with multiple males may be host to a tiny internal battle between the sperm of different males, with the victors surviving to fertilize the female's eggs. These and other possibilities would be simple extensions to the recipe shown here, with females keeping references to multiple male mates (probably using `setValue()` instead of `tag`, since `tag` is a singleton property), and the dynamics of female choice or sperm competition playing out later, perhaps in the `reproduction()` callback.

This recipe illustrates the power and flexibility of the `survival()` callback mechanism, added in SLiM 3.7. These callbacks provide complete control over the survival decisions made by SLiM, including the option to move individuals between subpopulations in lieu of mortality. Indeed, the movement of individuals by `survival()` callbacks need not represent mortality at all; it could be used to implement ordinary migration, or "life history" transitions such as attainment of reproductive maturity. Those sorts of movements could also, more conventionally, be implemented using `takeMigrants()` as we have seen previously, but in some situations `survival()` may be simpler – particularly when movement is related to fitness and selection. Since

`survival()` callbacks are provided with information on the survival decision made by SLiM for every individual, including each individual's final calculated fitness value, they can also be used purely observationally – for example, to log out the fitness value of each individual along with the outcome of selection for that individual, or similar information (see section 16.12).

### 16.23 Tracking separate sexes in script, nonWF style

Section 12.5 showed how to track the sex of individuals yourself in script in a WF model, using the `tag` property to represent the sex of each individual. In this section we will look at the same idea implemented in a nonWF model.

This approach can be useful for implementing unusual mating systems that don't fit well into SLiM's conception of sex. In sexual models in SLiM, for example, SLiM does not allow selfing; it is assumed that females produce eggs and male individuals produce sperm, and that one egg and one sperm are required for fertilization. There are some sexual species, however, that can reproduce by methods such as *automixis* – the fusion of nuclei or gametes produced by the same individual – that violate this assumption. In this section, by way of illustration, we will implement a sexual species that usually reproduces by biparental mating between a female and a male, to produce an offspring of either sex, but in which a female can also sometimes reproduce by automixis, without a male mate, to produce a daughter.

There are various forms of automixis; for simplicity, we will here implement a form of automixis in which any two randomly chosen gametes from the focal female fuse, but we will discuss other types of automixis at the end. The focus here is not really on automixis; that is just used as a pedagogical example to justify the explicit tracking of sex in script, rather than using `initializeSex("A")` and letting SLiM do it for us.

Here's the whole model except the `reproduction()` callback:

```
initialize() {
  initializeSLiMModelType("nonWF");
  defineConstant("K", 500);           // carrying capacity
  initializeMutationType("m1", 0.5, "f", 0.0);
  m1.convertToSubstitution = T;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", K);

  // assign random sexes (0 = male, 1 = female)
  p1.individuals.tag = rbinom(p1.individualCount, 1, 0.5);
}
early() {
  p1.fitnessScaling = K / p1.individualCount;
}
1:2000 late() {
  ratio = sum(p1.individuals.tag == 0) / p1.individualCount;
  catn(sim.cycle + ": " + ratio);
}
```

The initialization is boilerplate; note that we do not call `initializeSex()` here, since we do not want SLiM to track sex itself. In the 1 `early()` event we draw a random sex for each individual from a binomial distribution, and assign those sex values into the `tag` property of the individuals;

completely arbitrarily, we designate 0 as male and 1 as female. In the 1:2000 `late()` event we output the sex ratio in each cycle (calculated as  $M:(M+F)$ , following the definition used by SLiM), since the sex ratio will provide a test that the model is working as intended.

The `reproduction()` callback is also quite simple:

```
reproduction() {
  // we focus on the reproduction of the females here
  if (individual.tag == 1)
  {
    if (runif(1) < 0.7)
    {
      // choose a male mate and produce a son or daughter
      mate = subpop.sampleIndividuals(1, tag=0);
      offspring = subpop.addCrossed(individual, mate);
      offspring.tag = rbinom(1, 1, 0.5);
    }
    else
    {
      // reproduce through automixis to produce a daughter
      offspring = subpop.addSelfed(individual);
      offspring.tag = 1;
    }
  }
}
```

We want to focus on the reproduction of females here; usually we could do that by declaring the callback as `reproduction(NULL, "F")`, but since we're tracking sex ourselves in `tag`, we need to test that the focal individual is female ourselves. If the focal individual is female, we then roll the dice; with a 70% chance, we choose a mate with `sampleIndividuals()`, narrowing the pool of eligible individuals to males using `tag=0`, and then call `addCrossed()` to produce one offspring from that biparental mating. That offspring is assigned a sex randomly, with equal probability. The remaining 30% of the time, we have the female reproduce by automixis, by calling `addSelfed()`. This is the step that SLiM would not allow in a normal sexual model; `addSelfed()` can only be called in non-sexual models. (This limitation was put in place to clarify how to implement selfing when modeling sex chromosomes.) Since this is, in SLiM's view, a non-sexual model, we are allowed to call it. The offspring resulting from this automixis is always female (following the biology of some species with automixis).

When the model is run, it can be seen that the calculated sex ratio starts around 0.5 but quickly falls to an average of about 0.35. This is expected, since males are produced only by biparental mating, which happens 70% of the time. Any given offspring will therefore be male with a probability of  $0.7 \times 0.5 = 0.35$ . Our tracking of sex appears to be working properly.

This model can easily be extended in various directions. One could track more than just two sexes or mating types; the `tag` property can be used however you wish, and you can control how it affects reproduction with your own custom script. You could model fish that change sex over the course of their lives (as some do), alternative sexual types like "sneaker males", sex determination according to environmental cues like temperature, species with ZW sex chromosomes instead of XY (if there are biological details that make that distinction important for your model), or anything else of the sort.

As mentioned at the outset, automixis is a complex topic, and it is not our real focus here. We have implemented automixis by simply calling `addSelfed()`, which generates two haploid gametes from the focal female using SLiM's standard recombination machinery and fuses them to form the diploid offspring. The biological reality is that depending upon whether a species uses "central



fusion automixis”, “terminal fusion automixis”, or some other type, the gametes that fuse to produce the offspring may have certain properties because they are derived from certain specific points in the process of meiosis (Wikipedia has a helpful diagram in its article on automixis). In this case, `addSelfed()` does not provide the desired behavior, and the `addRecombinant()` method is the more powerful alternative. With `addRecombinant()`, you can control the exact set of recombination breakpoints used to generate each of the two gametes used by SLiM in the offspring, making them exhibit the necessary properties. We will not show an example of that approach here, but other recipes in this manual do use `addRecombinant()` for various other purposes; section 16.17 might be particularly relevant as an example of the general approach. Generating your own breakpoints is reasonably straightforward, either by using the `drawBreakpoints()` method of `Chromosome` (if SLiM’s default breakpoint generation is acceptable, perhaps with some modification), or by generating them yourself using functions like `rbinom()` to determine the number of breakpoints and `rdunif()` to draw each breakpoint’s position. Section 1.5.6 discusses SLiM’s default recombination breakpoint generation algorithm in detail; you might wish to pattern your own breakpoint generation algorithm after it.

## 16.24 Modeling haplodiploidy with `addRecombinant()`

We have seen ways of handling a variety of mating systems in SLiM, including selfing (section 6.3.1), cloning (section 6.3.2), haploidy (sections 14.9, 16.13, and 16.14), and alternation of generations (section 16.16), as well as how to track separate sexes yourself in script in order to implement more unusual mating systems such as automixis (sections 12.5 and 16.23). One fairly common mating system we have not covered yet is haplodiploidy. In haplodiploidy (sometimes called arrhenotoky), males develop from unfertilized eggs and are haploid, while females develop from fertilized eggs and are diploid. It is particularly well-known as the sex-determination system in the Hymenoptera (bees, ants, and wasps), and has interesting effects on the relatedness of individuals that may promote the emergence of eusociality.

The haplodiploidy model presented here was developed in collaboration with Rodrigo Pracana, Richard Burns, Robert L. Hammond, and Yannick Wurm, and a related preprint has been posted on bioRxiv at <https://doi.org/10.1101/2021.10.25.465450>. The model presented in that publication is fairly different from the one shown here; it implements a Wright–Fisher model design (but written as a nonWF SLiM model), with non-overlapping generations, a fixed population size, and fitness that acts through mating success. In that model, all reproduction occurs in a single “big bang”, generating all of the individuals needed to fill out the next juvenile cohort. It therefore follows the recipe of section 16.15, which shows how to implement a Wright–Fisher model as a nonWF model. (In fact it goes even further than that recipe does, in making fitness influence mating success rather than reproduction!) That paper will likely be of interest to those modeling haplodiploidy in SLiM, as it explores some of the consequences of haplodiploidy for deleterious and beneficial mutations and compares simulations in SLiM to some prior analytical work. If the SLiM code from *either* that paper or this section is adapted for use in a publication, a citation to Pracana, Burns, Hammond, Haller, & Wurm (2021) would be much appreciated. Thanks to my collaborators for welcoming the publication of this recipe here.

In this section, on the other hand, we will look at a very minimal nonWF haplodiploidy model; to keep things simple it will follow typical nonWF model conventions such as overlapping generations, density-dependent population regulation, and fitness that acts through survival. Each female will reproduce independently, via a separate callout to the `reproduction()` callback. In all these respects, this recipe follows the very simple nonWF template of the recipe in section 16.1.

OK, with that background information out of the way, let’s look at the model! Note that this model requires SLiM 3.7 to run. Here’s the `initialize()` callback:

```

initialize() {
  defineConstant("K", 2000);
  defineConstant("P_OFFSPRING_MALE", 0.8);
  initializeSLiMModelType("nonWF");
  initializeMutationRate(1e-8);
  initializeMutationType("m1", 0.0, "f", 0.0);
  m1.convertToSubstitution = T;
  m1.haploidDominanceCoeff = 1.0;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 999999);
  initializeRecombinationRate(1e-6);
  initializeSex("A");
}

```

We define a carrying capacity  $K$  of 2000. We also define a parameter, `P_OFFSPRING_MALE`, that is the probability that a given offspring generated by a female will be a (haploid) male, as opposed to a (diploid) female. Here that is set to 0.8, quite arbitrarily; offspring will be mostly males.

We set up a neutral mutation type, `m1`; this model can also accommodate non-neutral mutations, but we won't do that here for simplicity. We set `convertToSubstitution` to `T` so these neutral mutations get substituted by SLiM when they fix (see section 1.5.2); note that SLiM will correctly detect fixation even though the model contains a mixture of haploids and diploids. We also set the `haploidDominanceCoeff` property to 1.0. This property is used as the “dominance” coefficient in haploids, where mutations can only be present in a single copy. Recall that in diploids, a homozygous mutation has a fitness effect of  $1+s$ , where  $s$  is the selection coefficient of the mutation, and a heterozygous mutation has a fitness effect of  $1+hs$ , where  $h$  is the dominance coefficient kept in the `dominanceCoeff` property of `MutationType`. You might want a mutation to have a different fitness effect when found in a haploid – not  $1+s$ , and not  $1+hs$ , but rather  $1+zs$ , where  $z$  is the value of the `haploidDominanceCoeff` property (note that I just made up the use of the symbol  $z$  here, that is not a standard convention). Here we set this to 1.0, but it doesn't matter anyway since this is a neutral model. Note that if you wanted even finer control over the fitness effects of mutations in haploids you could achieve that with a `mutationEffect()` callback.

The rest of the initialization is quite standard. We use `initializeSex("A")` to ask SLiM to track separate sexes for us; we don't do anything here that requires us to use the techniques of section 16.23 to track sex ourselves.

Here's the rest of the framework of the model, apart from the `reproduction()` callback:

```

1 early() {
  // make an initial population with the right genetics
  mCount = asInteger(K * P_OFFSPRING_MALE);
  fCount = K - mCount;
  sim.addSubpop("p1", mCount, sexRatio=1.0, haploid=T); // males
  sim.addSubpop("p2", fCount, sexRatio=0.0, haploid=F); // females
  p1.takeMigrants(p2.individuals);
  p2.removeSubpopulation();
}
early() {
  p1.fitnessScaling = K / p1.individualCount;
}
10000 late() {
  sim.simulationFinished();
}

```

The `early()` event implements density-dependent fitness, and the `10000 late()` event defines the end of the simulation. The interesting bit is the `1 early()` event that creates the initial

population. First, it uses `P_OFFSPRING_MALE` to calculate the initial number of males and females in the subpopulation. Then it creates all the males in `p1`, passing `sexRatio=1.0` to make them all male, and `haploid=T` to make them all haploid. Similarly, it creates all the females in `p2`, passing `sexRatio=0.0` to make them all female, and `haploid=F` to make them all diploid. It moves the females into `p1` with `takeMigrants()`, and then removes `p2` from the model. This sleight of hand lets us use `addSubpop()` to easily create the males and females we want. We could avoid the use of `p2` by creating `p1` with an initial size of 0, and then filling it up with new empty individuals with the appropriate genetics using `addEmpty()` in a `1 reproduction()` callback, but this design is much simpler, and the ephemeral existence of `p2` is fairly harmless. (If you use tree-sequence recording with this model, the initial creation of the females in `p2` might make for problems with recapitation and so forth; in that case, the `addEmpty()` technique might be better. You could also simply call `sim.addSubpop("p1", K, sexRatio=P_OFFSPRING_MALE)` to create the initial subpopulation; this will make the males in the first generation be diploid, but that ought to be harmless since their second genomes will never be used for anything.)

Finally, here is the `reproduction()` callback, which runs only for females since it is declared with "F" for its focal sex:

```
reproduction(NULL, "F") {
  // choose an initial copy strand based on a coin flip
  strand = rbinom(1, 1, 0.5);
  gen1 = strand ? individual.genome1 else individual.genome2;
  gen2 = strand ? individual.genome2 else individual.genome1;
  breaks = sim.chromosome.drawBreakpoints(individual);

  // decide whether we're generating a haploid male or a diploid female
  if (rbinom(1, 1, P_OFFSPRING_MALE))
  {
    // didn't find a mate; make a haploid male from an unfertilized egg:
    // - one genome comes from recombination of the female's genomes
    // - the other genome is a null genome (a placeholder)
    subpop.addRecombinant(gen1, gen2, breaks, NULL, NULL, NULL, "M");
  }
  else
  {
    // found a mate; make a diploid female from a fertilized egg:
    // - one genome comes from recombination of the female's genomes
    // - the other genome comes from the mate (a haploid male)
    mate = subpop.sampleIndividuals(1, sex="M");
    subpop.addRecombinant(gen1, gen2, breaks, mate.genome1, NULL, NULL, "F");
  }
}
```

This handles all the key details of haplodiploidy. We will generate the offspring using the method `addRecombinant()`; it is a very low-level method, so we will need to tell it exactly what to do (see section 24.15.12). For this reason, we need to choose which of the female's genomes will be the initial copy strand when generating the egg. We do that up front, using `rbinom()` as a coin flip, and assign `gen1` to be the initial copy strand and `gen2` to be the other strand. Next, since we will need crossover breakpoints to pass to `addRecombinant()` to generate the egg, we get them from `sim.chromosome`, which draws them following SLiM's standard procedure. Finally, we need to decide whether we're going to generate a (haploid) male or a (diploid) female offspring. We again use `rbinom()`, with `P_OFFSPRING_MALE` as the probability for male.

If the offspring is male, there is no mate; the egg is unfertilized. In this case, we call `addRecombinant()` with `gen1`, `gen2`, `breaks` for the parental genomes and breakpoints for the egg,

and `NULL`, `NULL`, `NULL` for the sperm to indicate that there is no sperm. This makes the second genome of the offspring a null genome, indicating that the offspring is haploid. Finally, we pass "M" to `addRecombinant()` to tell it the offspring is male.

If the offspring is female, we first have to choose a mate. We use `sampleIndividuals()` to do this, requiring only that the mate be male; of course, further restrictions could be placed on that selection. Then we call `addRecombinant()`, passing `gen1`, `gen2`, `breaks` as before for the egg. Here, however, we pass `mate.genome1`, `NULL`, `NULL` for the sperm, indicating that the sperm carries a copy of the male's haploid genome, without a second strand and without recombination. Finally, we pass "F" to tell `addRecombinant()` the offspring is male.

That's all there is to it; it's really quite straightforward. Implementing these sorts of mating system is mainly a matter of thinking through the biological logic: who mates with whom, who is what ploidy, and where each offspring chromosome comes from. Translate that logic into the appropriate calls to `addCrossed()`, `addCloned()`, `addSelfed()`, and `addRecombinant()` in your `reproduction()` callback, and things will often go quite smoothly. If you need to violate rules that SLiM normally enforces – having a non-hermaphroditic individual self, for example, or having more than two sexes – you may want to track the sex of individuals yourself, as shown in section 16.23.

Breaking with our usual procedure, we will not show results from this recipe here; demonstrating the correctness of this model is really beyond the scope of this manual. Instead, the reader is encouraged to consult Pracana, Burns, Hammond, Haller, & Wurm (2021), which analyzes results from a fairly similar model to show that they match the predictions for haplodiploidy made by analytical models.

## 17. Tree-sequence recording: tracking population history and true local ancestry

This manual has already discussed a variety of ways of tracking ancestry in SLiM models, such as pedigree recording (section 14.1; see also section 16.12 for a related model) and using mutations to mark the population of origin of chromosome positions (section 14.7). In this chapter we will look at a very powerful way of tracking ancestry called tree-sequence recording. Tree-sequence recording is a new feature added in SLiM 3, and is introduced in some detail in section 1.7; you should read that section now if you haven't already. This is an advanced feature, and so this chapter will assume a familiarity with general SLiM and Eidos topics and techniques.

The use of tree-sequence recording leans heavily upon Python, where the tree sequence saved in a `.trees` file can be read, analyzed, and modified. To run most of the recipes in this chapter, you will need to have Python installed (Python 3.6.12 was used to construct and test these recipes). Some familiarity with Python will come in useful, but the Python scripts here will not be extensive. You will also need to have three Python packages installed: `msprime`, `tskit`, and `pyslim`. The recipes in this chapter are based upon SLiM 4.0, `msprime` 1.2, `tskit` 0.4.1, and `pyslim` 1.0.

The `msprime` and `tskit` packages are used to analyze tree sequences, overlay mutations onto a tree sequence, and perform coalescent simulations, among other tasks (`tskit` is the foundation upon which `msprime` is built). The `msprime` project lives at <https://github.com/tskit-dev/msprime>; `tskit` lives at <https://github.com/tskit-dev/tskit>. The `pyslim` package essentially provides a bridge between SLiM and `msprime` and `tskit`; most operations on SLiM-produced tree sequences are done with `tskit`, while `pyslim` provides some additional methods to make common operations more straightforward (e.g., recapitation or preparing an `msprime` simulation for reading in to SLiM). The `pyslim` project lives at <https://github.com/tskit-dev/pyslim>. These packages have excellent documentation, including tutorials; see section 1.10 for links to online resources.

In general, tree-sequence recording is compatible with all other SLiM features. It can be used with both WF and nonWF models, with all types of callbacks, with any population structure, and so forth. In this chapter we will not attempt to show a tree-seq version of every recipe we've already seen; instead we will focus upon the usage of the tree-sequence recording feature itself, to show what it is capable of and how it should be used. The early recipes will use only `tskit` and `msprime`; later recipes will use `pyslim` for SLiM annotations and other advanced functionality.

### 17.1 A minimal tree-seq model

To begin, we will look at a minimal tree-sequence recording model based upon a simple neutral model. This recipe enables tree-seq recording and then runs much as usual:

```
initialize() {
    initializeTreeSeq();
    initializeMutationRate(0);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 1e8-1);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
5000 late() {
    sim.treeSeqOutput("./recipe_17.1.trees");
}
```

There are three notable differences here from the vanilla neutral model we have seen before. First, we now call `initializeTreeSeq()` to turn tree-seq recording on. This is all that is needed;

SLiM will now record all tree-sequence information throughout the run, and will automatically conduct simplification periodically (which can be customized with the `simplificationRatio` and `simplificationInterval` parameters to `initializeTreeSeq()`; see section 24.1).

Second, we now set the mutation rate to zero, because we don't want to model neutral mutations. Typically, when using tree-seq recording, neutral mutations are overlaid onto the ancestry tree after forward simulation has completed; we will show that technique in the next subsection, so in anticipation of that we have turned off neutral mutations here. Note that it is not *necessary* to turn off neutral mutations; there is no problem with including them in a tree-seq model, apart from the additional performance penalty of simulating them. It is just usually not desirable, because avoiding the overhead of simulating neutral mutations is one of the primary motivations for using tree-seq recording in the first place. Also, note that if the model had a mix of neutral and non-neutral mutations we would probably want to remove only the neutral mutations from the model, with a concomitant adjustment to the mutation rate; we will see an example of this in 17.3. (We would not want to remove the non-neutral mutations, since they affect the model's dynamics and thus cannot be overlaid after simulation.)

Third, we call `treeSeqOutput()` to write out a `.trees` file containing the full tree sequence that was recorded. This file is in a binary format defined by the `tskit` package (see section 26.4 for some details). However, it can be loaded back into SLiM with `readFromPopulationFile()`, just like a regular SLiM output file, and it can also be read in Python using `tskit`; we will see examples of both techniques in later recipes.

That's all there is to it; we will use the `.trees` file generated here in the next recipe, but for now we're done. It's worth noting, though, that an informal test of this model indicates that it runs in about 5 seconds, whereas the same model with tree-sequence recording turned off and a mutation rate of  $1e-7$  took 294 seconds to run. This performance improvement is not atypical; this is one major reason why tree-sequence recording is useful.

## 17.2 Overlaying neutral mutations

This recipe is actually a Python recipe, not a SLiM recipe. It depends upon having run the previous recipe, in section 17.1, to generate a `.trees` file representing the execution of a simple neutral model. Since mutation was turned off in that model, the `.trees` file contains no mutations; but it contains all of the ancestry information needed to overlay them. Here's the core of the solution (we'll add a little more to this recipe below):

```
import msprime, tskit

ts = tskit.load("./recipe_17.1.trees")
ts = ts.simplify()

mutated = msprime.sim_mutations(ts, rate=1e-7, random_seed=1, keep=True)
mutated.dump("./recipe_17.1_overlaid.trees")
```

First we import the `msprime` and `tskit` packages (which need to be installed, as mentioned previously). Then we load the saved `.trees` file into a tree sequence object with `tskit.load()`; it used to be necessary to use `pyslim.load()` instead, but now `tskit` can load SLiM tree sequences on its own without help from `pyslim`.

We then call `simplify()` to simplify the loaded tree sequence. SLiM 3.1 produces `.trees` files that contain the ancestry trees all the way back to the first-generation nodes in each new subpopulation created by `addSubpop()`, as long as those nodes are still ancestral to extant individuals. These first-generation nodes are useful for various purposes, such as tracing ancestry (see section 17.5) and recapitation (see section 17.10), so they are retained by SLiM for



convenience. However, they are not marked as being part of the sample, so they disappear by default when the tree sequence is simplified; this makes it easy to get rid of them when they are not wanted. Section 17.5 discusses this in more detail, and the `pyslim` manual has excellent discussion of this issue; see section 1.10 for relevant links. Here, our chosen goal is to overlay mutations only back to the point of coalescence, and so we call `simplify()` to strip away all ancestral information above the point of coalescence. (If we wanted to overlay fixed mutations as well, past coalescence back to the start of forward simulation, then we would not call `simplify()`.)

Next, we overlay mutations with `msprime.sim_mutations()` to generate a new tree sequence (with a given mutation rate and random number generator seed), and finally we write the mutated tree sequence out to a new `.trees` file. The `keep=True` parameter to `msprime.sim_mutations()` indicates that any existing mutations should be kept, not overwritten; in this example there are no mutations already present in the tree, but if there were, we would typically want to keep them.

This script runs in about 0.2 seconds. Why is it so much faster than modeling the neutral mutations in SLiM? The key is that when mutations are overlaid after the forward simulation has completed, they only need to be generated along those branches of the ancestry tree that led to extant individuals at the end of the run. All of the branches of the evolutionary tree that went extinct – the vast majority of branches, in most models – need not have mutations overlaid.

There is one detail that we’ve glossed over so far. As mentioned above, here we want to overlay mutations only back to the point of coalescence – only mutations that are still segregating in the population, in other words, rather than also including fixed mutations. To do that, we called `simplify()` above to strip away the ancestry trees above coalescence. However, there is no guarantee that our model, from section 17.1, will have coalesced in the first place; there might still be segments of the chromosome that trace back to different first-generation ancestors because they have not yet coalesced. In that case, the mutation overlay code above might not make much sense; it would implicitly make the assumption that the first generation in SLiM is genetically identical, with no segregating variation at all, since the subsequent mutation overlay would not provide any mutations prior to that first generation.

There are three typical solutions to this problem. (1) Run the SLiM model for longer. However, there is no number of generations that would *guarantee* that the model would fully coalesce, so you would have to just choose a large number, and then cross your fingers and hope. You could also explicitly run the SLiM model until it coalesces, using the techniques shown in section 17.6; however, there are issues with that approach too, as discussed in that section. (2) Start the SLiM simulation by loading a coalescent history, generated by `msprime`, from a `.trees` file to start the first generation; this is discussed in sections 17.8 and 17.9. This might not be appropriate if you want the history back to coalescence to involve non-neutral processes, however. (3) You could *recapitate* your tree sequence in Python after forward simulation has finished, by adding on a neutral coalescent history that starts at the first forward-simulated generation and goes backwards in time until coalescence (see section 17.10). This solution is faster and more elegant than the alternatives, but it assumes, like (2), that the history back to coalescence can be treated as neutral.

We won’t deal with this problem for now; instead, we’ll just check for coalescence and raise an error if it hasn’t occurred. Insert this code into the Python recipe, after we load the tree sequence with `tskit.load()` and before we overlay mutations with `msprime.sim_mutations()`:

```
for t in ts.trees():
    assert t.num_roots == 1, ("not coalesced! on segment {} to {}".
                             format(t.interval[0], t.interval[1]))
```

This code loops through the trees in the tree sequence and checks that they all have just a single root, which means that they coalesce at a single ancestral node. If you run this code now, you may find that the tree sequence generated by section 17.1’s recipe did not, in fact, coalesce. In

that case, if you want to see this recipe work before proceeding, try running that model again, but out to 15000 ticks instead; that will usually coalesce.

We decided to simplify because the tree sequence contains lines of ancestry going back to the first-generation nodes from whom the final generation are descended. You might wonder, at this point: why are those first-generation nodes present in SLiM's output in the first place? Doesn't SLiM simplify the tree sequence before saving? It does. However, the `simplify()` algorithm can run in two different modes. With the default argument `keep_input_roots=False`, it provides the behavior we've been working with in Python, above: it strips off history prior to coalescence. SLiM, however, runs `simplify()` in the other mode, in which ancestors above coalescence are retained in the tree sequence as long as they have extant descendants. By using this mode, SLiM preserves the first-generation ancestors so they can be used for recapitation and other purposes. This `simplify()` mode is available in Python by passing `keep_input_roots=True`. Since SLiM has already performed this mode of simplification on the tree sequence there is usually no reason to do it again; however, you might do so if, for example, you wanted to simplify down to a subsample of the extant individuals, retaining the necessary first-generation nodes for later recapitation. The `pyslim` documentation has an example of such simplification in its tutorial.

The new `.trees` file written out at the end could be read by a different Python script to perform further analysis or modification, again using `msprime` and `tskit`. We could also, instead, simply work with the new tree sequence object in further analysis code in this script; or we could write out the mutation information to a different file format, such as MS or VCF format, if the ancestry information encapsulated by the `.trees` file is no longer needed.

It should also be possible to read a `.trees` file with overlaid mutations back into SLiM with the `readFromPopulationFile()` method, to use its state as the starting point for further forward simulation. However, to do that the mutations provided by `msprime` would need to be annotated by `pyslim` with additional data about them that is required by SLiM, such as their selection coefficients and mutation types. The `pyslim` documentation has an example vignette showing this technique at [https://tskit.dev/pyslim/docs/stable/vignette\\_coalescent\\_diversity.html](https://tskit.dev/pyslim/docs/stable/vignette_coalescent_diversity.html).

### 17.3 Simulation conditional upon fixation of a sweep, preserving ancestry

In the recipe of section 17.1 we saw a tree-seq model for a trivial neutral model, but this method is not limited to neutral models. Here we will look at a model involving both neutral and deleterious mutations, into which a beneficial mutation is introduced. We want the beneficial mutation to sweep, and we want this model to run conditional upon a successful sweep. This will be quite similar to the recipe of section 9.2, then; however, here we have a background of deleterious as well as neutral mutations. When we convert the model to use tree-sequence recording, we will remove the neutral mutations from the model (since they can be overlaid later, as in section 17.2). The ancestry information for all of the individuals in the model will be preserved, even though the model will restart itself repeatedly until fixation is achieved.

First, here is the model without tree-sequence recording:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 0.5, "g", -0.01, 1.0); // deleterious
    initializeMutationType("m3", 1.0, "f", 0.05);       // introduced
    initializeGenomicElementType("g1", c(m1, m2), c(0.9, 0.1));
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
```

```

1 early() {
  defineConstant("simID", getSeed());
  sim.addSubpop("p1", 500);
}
1000 late() {
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m3, 10000);
  sim.outputFull(tempdir() + "slim_" + simID + ".txt");
}
1000:100000 late() {
  if (sim.countOfMutationsOfType(m3) == 0) {
    if (sum(sim.substitutions.mutationType == m3) == 1) {
      cat(simID + ": FIXED\n");
      sim.simulationFinished();
    } else {
      cat(simID + ": LOST - RESTARTING\n");

      sim.readFromPopulationFile(tempdir() + "slim_" + simID + ".txt");
      setSeed(rdunif(1, 0, asInteger(2^62) - 1));
    }
  }
}

```

Since this is very similar to the recipe of section 9.2, we won't discuss it here; see that section for discussion. Here, our goal is to convert it into a tree-seq model:

```

initialize() {
  initializeTreeSeq();
  initializeMutationRate(1e-8);
  initializeMutationType("m2", 0.5, "g", -0.01, 1.0); // deleterious
  initializeMutationType("m3", 1.0, "f", 0.05); // introduced
  initializeGenomicElementType("g1", m2, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeRecombinationRate(1e-8);
}
1 early() {
  defineConstant("simID", getSeed());
  sim.addSubpop("p1", 500);
}
1000 late() {
  target = sample(p1.genomes, 1);
  target.addNewDrawnMutation(m3, 10000);
  sim.treeSeqOutput(tempdir() + "slim_" + simID + ".trees");
}
1000:100000 late() {
  if (sim.countOfMutationsOfType(m3) == 0) {
    if (sum(sim.substitutions.mutationType == m3) == 1) {
      cat(simID + ": FIXED\n");
      sim.treeSeqOutput("slim_" + simID + "_FIXED.trees");
      sim.simulationFinished();
    } else {
      cat(simID + ": LOST - RESTARTING\n");

      sim.readFromPopulationFile(tempdir() + "slim_" + simID + ".trees");
      setSeed(rdunif(1, 0, asInteger(2^62) - 1));
    }
  }
}

```

We have added a call to `initializeTreeSeq()`, and changed the file save and load code to use a `.trees` file instead of a standard SLiM text output file. We also removed the neutral mutations from the model; note that this required changing the mutation rate. Since 90% of the mutations in the original model were neutral, the new model has a mutation rate one-tenth as high, so that the rate of deleterious mutations remains unchanged. In this simple model, with only one genomic element type, the adjustment to the mutation rate is very straightforward; in a model with a more complex genetic architecture, the necessary adjustment to the mutation rate after removal of neutral mutations might require switching to a mutation-rate map (see section 24.1).

Otherwise, the model is unchanged. This model will run in much the same manner as the first version, restarting itself whenever the `m3` mutation is lost until it achieves fixation. The deleterious mutations present when the beneficial mutation is introduced will be saved to the `.trees` file and restored each time the model restarts.

When this model achieves fixation, it writes out the final state of the model to a new `.trees` file (not in the temporary directory, this time). This file can be loaded into Python with `tskit`, as in the recipe of section 17.2, to overlay neutral mutations, or to perform any other analysis desired. Note that since this model contains non-neutral mutations as well as neutral mutations, the mutation rate used in neutral mutation overlay would not be the original rate of  $1e-7$ , but instead the rate specifically of *neutral* mutations along the chromosome, and that with a complex genetic architecture this might necessitate the specification of a mutation-rate map rather than the use of a single fixed rate, similarly to the issue with the mutation rate in SLiM discussed above.

## 17.4 Detecting the “dip in diversity”: analyzing tree heights in Python

It has been mentioned several times that one can perform “other analysis” in Python using the information saved in a `.trees` file. In this recipe and the next, we will look at two examples. Because `.trees` files contain information about the true local ancestry of every location along the genome of every extant individual, including the times when recombination and mutation events occurred in the past, there are many interesting analyses that can be conducted.

The recipe in this section will model “background selection”, which is the effect of selection against deleterious mutations upon nearby neutral sites. Background selection is, in a sense, similar to “genetic hitchhiking”, which is the effect of selection for beneficial mutations upon nearby neutral sites. Although they are different, both of these types of so-called “linked selection” have been found to reduce genetic diversity in non-coding regions near genes; because mutations within genes often have functional effects (whether positive or negative), they often exert linked selection upon nearby neutral regions that reduces diversity. The reduction in diversity falls off as distance to the nearest gene increases, producing a characteristic “dip in diversity” near genes (Charlesworth et al. 1993; Hudson 1994; Sattath et al. 2011; Elyashiv et al. 2016).

The SLiM model for this is complicated only because, for higher statistical power, we want to model many genes interspersed with many non-coding regions, so that a single run of the model generates enough data for us to see the effect we’re interested in:

```
initialize() {
    defineConstant("N", 10000); // pop size
    defineConstant("L", 1e8);    // total chromosome length
    defineConstant("L0", 200e3); // between genes
    defineConstant("L1", 1e3);   // gene length
    initializeTreeSeq();
    initializeMutationRate(1e-7);
    initializeRecombinationRate(1e-8, L-1);
    initializeMutationType("m2", 0.5, "g", -(5/N), 1.0);
    initializeGenomicElementType("g2", m2, 1.0);
}
```

```

        for (start in seq(from=L0, to=L-(L0+L1), by=(L0+L1)))
            initializeGenomicElement(g2, start, (start+L1)-1);
    }
    1 early() {
        sim.addSubpop("p1", N);
        community.rescheduleScriptBlock(s1, 10*N, 10*N);
    }
    s1 10 late() {
        sim.treeSeqOutput("./recipe_17.4.trees");
    }
}

```

The chromosome is defined as a set of genes of length  $L1$ , separated by non-coding regions of length  $L0$ . The model doesn't define the non-coding regions with genomic elements; we don't want mutations to occur in the non-coding regions, since we will not need any neutral mutations to conduct our analysis (and even if we did, it would be better to overlay them afterwards in Python). A subpopulation of size  $N$  is defined, and then the final output event is rescheduled to tick  $10N$  (a rough guess at a coalescence time for the model; the results will not be terribly sensitive to the accuracy of this guess since diversity near the genes will be suppressed continuously throughout the run of the model).

To run this model (which takes several hours!) and then conduct the analysis, we can run the following Python script (assuming that we're in the directory containing the SLiM model file):

```

import subprocess, tskit
import matplotlib.pyplot as plt
import numpy as np

# Run the SLiM model and load the resulting .trees
subprocess.check_output(["slim", "-m", "-s", "0", "./recipe_17.4.slim"])
ts = tskit.load("./recipe_17.4.trees")
ts = ts.simplify()

# Measure the tree height at each base position
height_for_pos = np.zeros(int(ts.sequence_length))
for tree in ts.trees():
    mean_height = np.mean([tree.time(root) for root in tree.roots])
    left, right = map(int, tree.interval)
    height_for_pos[left: right] = mean_height

# Convert heights along chromosome into heights at distances from gene
height_for_pos = height_for_pos - np.min(height_for_pos)
L, L0, L1 = int(1e8), int(200e3), int(1e3)
gene_starts = np.arange(L0, L - (L0 + L1) + 1, L0 + L1)
gene_ends = gene_starts + L1 - 1
max_d = L0 // 4
height_for_left = np.zeros(max_d)
height_for_right = np.zeros(max_d)
for d in range(max_d):
    height_for_left[d] = np.mean(height_for_pos[gene_starts - d - 1])
    height_for_right[d] = np.mean(height_for_pos[gene_ends + d + 1])
height_for_distance = np.hstack([height_for_left[::-1], height_for_right])
distances = np.hstack([np.arange(-max_d, 0), np.arange(1, max_d + 1)])

# Make a simple plot
plt.plot(distances, height_for_distance)
plt.show()

```

After importing packages, we run the SLiM model with `subprocess.check_output()`, which generates the `recipe_17.4.trees` file. We read that file in using `tskit.load()` and gather tree heights, which are a proxy for diversity, from it; but let's examine that process step by step.

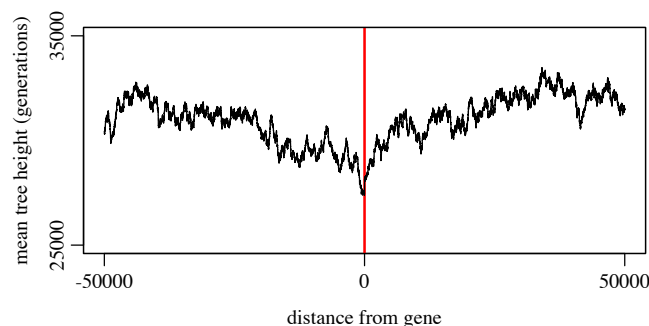
First of all, `tskit.load()` reads in the `.trees` file as a tree sequence, which is a collection of ancestry trees (section 1.7 introduces these concepts, but we will quickly review them here). The tree sequence thinks of the genome as being divided into successive intervals, each of which has a particular pattern of ancestry. Because the ancestry at adjacent positions along the genome tends to be highly correlated, this representation makes the tree sequence very compact and efficient. When we loop over the trees in `ts.trees`, we are actually looping over these chromosome intervals, getting the ancestry tree for each successive interval.

Second, the ancestry tree for a given interval is not quite a tree in the usual sense, because it can have multiple roots. This is because a given position will not necessarily share a common ancestor, in forward simulation; at the beginning of simulation every individual is an island unto itself, with no known relationship to any other individual, and ancestral relationships will be constructed by coalescence as the model runs forward. The tree for a given interval, in a simulation of  $N$  individuals, might therefore have anywhere from one root (if coalescence has produced a single common ancestor for the whole population) to  $N$  roots (if no coalescence has occurred yet). When we loop over roots in `tree.roots`, we are looping over common ancestors shared by subsets of the extant population.

The code gathers tree heights across all roots for the current interval, using `tree.time(root)`, and uses `np.mean()` to get the mean height. This is our metric of diversity for the interval. Since an interval can span more than one base position, we replicate that mean height across the tree's interval in `height_for_pos`, which we want to have a height value for each base position.

Note that the `simplify()` of the loaded tree sequence here is essential; without it, every tree would have a root in the first generation, in one or another original ancestor, and all the tree heights would be the same. The `simplify()` strips away the original ancestors, giving us trees with roots representing the most recent common ancestors for each tree.

With that done, we now need to convert that vector of tree heights along the chromosome into heights at a given distance from the nearest gene. This involves a sort of descrambling of the data based upon the genetic structure of the model, which (as described above) interleaved non-coding regions of length  $L_0$  with genes of length  $L_1$ . The details of this descrambling aren't worth getting into in detail; this is just data analysis, and is fairly routine Python code with no dependency upon `msprime` or `SLiM`. The end result of this analysis is a vector named `height_for_distance` that has mean heights for the corresponding distances in the vector `distances`. This is what we wanted, so we can plot it (this plot is prettified using R code not shown here, but the Python plot looks essentially the same):



This plot is noisy; that could presumably be smoothed out with more genes on a longer chromosome, a larger population size, and averaging across multiple runs of the model.



Nevertheless, the “dip in diversity” is very clear: the mean tree height drops to a clear minimum value precisely at zero on the plot.

Note that this analysis is not based upon the patterns of neutral mutation diversity around the simulated genes. Instead, the pattern of inheritance itself – the mean time to the most recent common ancestor at each base position – is used to generate the plot. This is far more powerful than using the pattern of neutral mutation diversity, because neutral mutations are sparse and stochastic. You certainly could overlay neutral mutations onto the tree sequence, as we did in section 17.2, and then feed that into the analysis methods used by empirical studies (and perhaps that would be a useful thing to do, to test the power or accuracy of those empirical methods). But the analysis here is, in a sense, the average across many such analyses – across an infinite number of such mutation overlays, in fact – and so it is far more powerful.

This recipe shows an analysis using just one metric provided by `msprime`, the height of a given root using `tree.time(root)`. The ancestry information provided by the tree sequence could be mined in countless other ways; we will see another in the next recipe.

## 17.5 Mapping admixture: analyzing ancestry in Python

It is often useful to be able to trace the true local ancestry at each location along the chromosome. In section 14.7, a recipe was presented that provided this ability by introducing a marker mutation at every base position along the chromosome in all of the individuals of one subpopulation, while leaving the other subpopulation unmarked. After admixture of the two subpopulations, the ancestry of each individual at each position could be determined by the presence or absence of the marker mutation. That method works, but it comes at a cost of considerable overhead in both runtime and memory usage. A chromosome of length  $1e6$  is close to the practical limit for that recipe; a chromosome of length  $1e8$  is estimated by extrapolation to take 7.2 days to run and – even worse – to require 8.1 TB of memory. Since the tree sequence is a sparse data structure that records information about the ancestry at each chromosome position in a highly compact form, one might suppose that it could provide information about true local ancestry more efficiently, and indeed it can.

In this section we will look at a recipe very similar to that of section 14.7; refer back to that section for further discussion of the design and motivation of that model. Here we will not use marker mutations; instead we will use tree-sequence recording:

```
initialize() {
    defineConstant("L", 1e8);
    initializeTreeSeq();
    initializeMutationRate(0);
    initializeMutationType("m1", 0.5, "f", 0.1);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-8);
}
1 late() {
    sim.addSubpop("p1", 500);
    sim.addSubpop("p2", 500);
    sim.treeSeqRememberIndividuals(sim.subpopulations.individuals);

    p1.genomes.addNewDrawnMutation(m1, asInteger(L * 0.2));
    p2.genomes.addNewDrawnMutation(m1, asInteger(L * 0.8));

    sim.addSubpop("p3", 1000);
    p3.setMigrationRates(c(p1, p2), c(0.5, 0.5));
}
```

```

2 late() {
    p3.setMigrationRates(c(p1, p2), c(0.0, 0.0));
    p1.setSubpopulationSize(0);
    p2.setSubpopulationSize(0);
}
2: late() {
    if (sim.mutationsOfType(m1).size() == 0)
    {
        sim.treeSeqOutput("./recipe_17.5.trees");
        sim.simulationFinished();
    }
}
10000 late() {
    stop("Did not reach fixation of beneficial alleles.");
}

```

Only two mutations ever exist in this model: the beneficial mutations introduced at 0.2L in p1 and at 0.8L in p2. After admixture to form p3, when both mutations have fixed, a `.trees` file is written out. The `.trees` file will then be read and analyzed in Python, as we will see below.

The only twist here is the call to `sim.treeSeqRememberIndividuals()`. Our goal is to trace the ancestry at each position in each individual to either p1 or p2; `treeSeqRememberIndividuals()` explicitly retains the first-generation individuals in p1 and p2 so that we can determine whether a particular genomic region originated in p1 or p2. This call is not strictly necessary, because the original genomes of each subpopulation created by `addSubpop()` are kept by SLiM automatically, as discussed in section 17.2. However, these would be removed if `simplify()` were called at any point during the post-processing in Python (unless the `keep_input_roots=True` flag were used), so it's good practice to explicitly remember them. Furthermore, explicitly remembering them retains information about the individuals, not just the genomes; we don't use that information in this recipe, but it can be useful to have. Finally, explicitly remembering the first generation makes the way that this recipe works more explicit and less magic. Note that you can remember any individuals you wish, during SLiM simulation; this is not limited to the first generation.

The run of the SLiM model, along with post-run analysis and plotting, is all done in a single Python script, as before:

```

import subprocess, tskit
import matplotlib.pyplot as plt
import numpy as np

# Run the SLiM model and load the resulting .trees file
subprocess.check_output(["slim", "-m", "-s", "0", "./recipe_17.5.slim"])
ts = tskit.load("./recipe_17.5.trees")

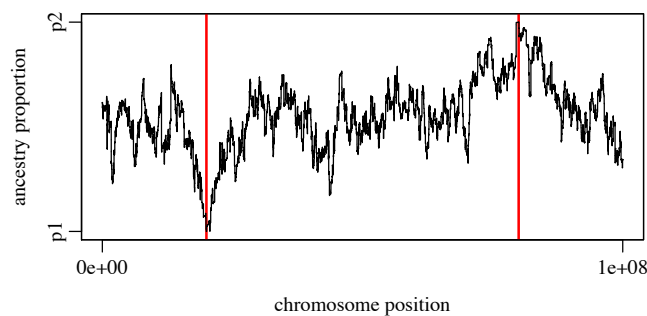
# Load the .trees file and assess true local ancestry
breaks = np.zeros(ts.num_trees + 1)
ancestry = np.zeros(ts.num_trees + 1)
for tree in ts.trees():
    subpop_sum, subpop_weights = 0, 0
    for root in tree.roots:
        leaves_count = tree.num_samples(root) - 1 # the root is a sample
        subpop_sum += tree.population(root) * leaves_count
        subpop_weights += leaves_count
    breaks[tree.index] = tree.interval[0]
    ancestry[tree.index] = subpop_sum / subpop_weights
breaks[-1] = ts.sequence_length
ancestry[-1] = ancestry[-2]

```

```
# Make a simple plot
plt.plot(breaks, ancestry)
plt.show()
```

After imports, the SLiM model (named `recipe_17.5.slim`) is run with `subprocess`, and the `.trees` file saved by the model is then read in using `tskit.load()` as usual. (We could call `simplify()` here if we wished, since we have explicitly remembered the ancestors we are interested in (see discussion above), but there is no reason to – SLiM has already simplified with `keep_input_roots=True` before saving). We then loop over the trees in the tree sequence, and over the roots for each tree (see section 17.4 for discussion of these concepts). For each root, we want to assess ancestry as tracing back to either `p1` or `p2`; this can be done simply by calling `tree.population(root)`, since every node in the tree sequence is marked with its subpopulation of origin. We average the ancestry of all roots for an interval to get the mean ancestry, but this is a weighted average; each root is weighted according to the number of descendants it has, effectively computing the average ancestry across all extant individuals, rather than across roots. The start positions of the trees on the chromosome are recorded in parallel with these calculated mean ancestry values, and a terminating entry in both vectors brings us to the end of the chromosome.

That's it for the analysis; this model is much simpler structurally than the previous recipe. All that is left is to plot the data, which is trivial with `matplotlib`. The final plot (produced by an R script for `polish`, rather than by the Python code here) looks like:



This is much the same as what the recipe of section 14.7 provided. Since the two beneficial mutations both fixed, the ancestry of the final population is pure, `p1` or `p2`, at the points where they were introduced. The ancestry then shades continuously (but stochastically) between those two points due to recombination (see section 14.7 for further discussion). The SLiM model here took only 0.415 seconds to run, however – somewhat of an improvement over 7.2 days. The post-run analysis took about 62 seconds; looping over all of the roots of all of the trees in the tree sequence is not entirely trivial, but is still far simpler than simulating  $1e8$  marker mutations in SLiM.

This recipe will generalize easily to any number of ancestral subpopulations, as long as the original founders of each subpopulation are remembered with `treeSeqRememberIndividuals()` so that ancestry can be traced back to them (of course taking the mean of the subpopulations of the tree roots wouldn't be the right analysis then). Since every node knows the subpopulation to which it belonged, the ancestral history at each position can be assessed by walking up the ancestry chain from the extant nodes to the roots, as long as one ensures that the appropriate ancestors are remembered forever; one might remember every individual that migrates to a new subpopulation, for example, using the `migrant` property of `Individual`, so that the migratory history through the ancestry tree can be traced. All of this goes far beyond what would be reasonable to implement with marker mutations using the methods of section 14.7.

## 17.6 Measuring the coalescence time of a model

A common problem in forward simulation is deciding how to conduct model burn-in. A burn-in period is a period of simulation executed in order to arrive at an appropriate starting state for the model one wishes to execute; often this starting state is for a neutral model at equilibrium, but not necessarily so. But how long should the burn-in period be, to provide such an equilibrium state? Running until coalescence ensures that no genetic diversity in the population could date back to the start of the simulation, so all polymorphic loci derive from after the start of the simulation. However, to attain equilibrium takes longer: in a truly neutral population of constant size, a total of two or three times the time required to reach coalescence should suffice. But that just kicks the can down the road; how long does it take to achieve coalescence? A heuristic of  $10N$  is often used: run for ten times the population size ( $N$ ), in generations. But this heuristic is less than satisfactory; the model may not have coalesced even at  $10N$  generations, or it may have coalesced long before (meaning wasted simulation time). And with any additional complexity, such as multiple subpopulations connected by migration, the  $10N$  rule is potentially even more problematic. What to do?

Tree-sequence recording provides a very easy solution, because the ancestry information that it records is well-suited to determining whether the model has achieved coalescence. The only complication is that coalescence can only be evaluated immediately after simplification has been performed; the coalescence test requires that the tree sequence be in a simplified state. SLiM largely hides this detail from the user; you can query the coalescence state at any time, but the answer you get will actually be the coalescence state at the time that the last simplification was performed, not the state at the present time. Sometimes, then, one will want to exert some control over the simplification process, so that one knows at what granularity coalescence is actually being assessed, as described below.

Here is a recipe demonstrating the coalescence-detection technique:

```
initialize() {
    initializeTreeSeq(checkCoalescence=T);
    initializeMutationRate(0);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 1e8-1);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
1: late() {
    if (sim.treeSeqCoalesced())
    {
        catn(sim.cycle + ": COALESCED");
        sim.simulationFinished();
    }
}
100000 late() {
    catn("NO COALESCENCE BY CYCLE 100000");
}
```

This model turns on coalescence checking with `checkCoalescence=T` (see section 24.1); coalescence checking needs to be turned on explicitly because there is a small performance penalty associated with it. Then, every tick, we check whether coalescence has occurred with `sim.treeSeqCoalesced()`. As mentioned above, this only tells us whether coalescence was observed at the last simplification; when auto-simplification finds that coalescence has occurred,

`treeSeqCoalesced()` will then return `T` when it is next called. Usually it is not necessary to detect coalescence in the exact tick in which it occurs; it is typically enough to know that it has occurred at some recent time, as this model does.

When this model is run, typical output looks like this:

9984: COALESCED

Coalescence was probably achieved some time before tick 9984; 9984 is just the tick in which auto-simplification *noticed* that coalescence. Note that this is higher than the  $10N$  value of 5000 (and auto-simplification occurs every couple of hundred ticks in this model); so at tick 5000 the model would not yet have coalesced. The time to coalescence is variable, depending as it does upon stochastic events, but repeated runs of this model will show that in fact it typically coalesces around 10000 ticks, or approximately  $20N$ .

To control the granularity of coalescence checking more precisely, one may pass a `simplificationInterval` value to `initializeTreeSeq()` to simplify (and check for coalescence) at a regular interval, such as every hundredth tick. For even finer control, one may turn off auto-simplification entirely, by passing `simplificationRatio=INF` to `initializeTreeSeq()`, and then call `treeSeqSimplify()` to simplify on demand, after which `treeSeqCoalesced()` will return an up-to-date assessment. See section 24.1 for further details on the use of these optional arguments to `initializeTreeSeq()`.

Note that certain actions in script, such as adding a new subpopulation, can break the coalescence of a model; all of the individuals in the population no longer share a common ancestor, even though they previously did. The value returned by `treeSeqCoalesced()` will not immediately reflect this; instead, as usual, that value will reflect the coalescence state after the last simplification that was performed. If this poses a problem, one can always explicitly simplify immediately after such model events, forcing the coalescence state to be re-checked.

Coalescence checking will work in all types of models (with the above caveats about timing and granularity), regardless of selection, population structure, etc. However, coalescence is only a useful indicator of the timescale needed for equilibration in fairly simple neutral models. If model dynamics during burn-in change over time, or are substantially non-neutral, coalescence may be a poor indicator of equilibration; indeed, such models may not even have an equilibrium state. What constitutes a proper burn-in for such models is a difficult question.

It is worth noting that the point of coalescence, in a forward simulation, is not itself an unbiased or equilibrium state. In particular, as a forward simulation runs the mean tree height will grow over time until coalescence is reached, at which point it will drop suddenly (because a more recent common ancestor has just emerged); then it will rise steadily again until the next coalescence, at which point it will again drop, and so forth. A plot of mean tree height over time therefore exhibits a “saw-tooth” pattern, and the moment of any given coalescence event is the bottom point of one saw-tooth – a special point in time, not a typical or average one. It is therefore advisable to continue neutral burn-in well beyond the point of coalescence; this is why, at the beginning of this subsection, we recommended running for two or three times the time required to reach coalescence. For an even better solution to this problem of neutral burn-in, see the “recapitation” technique described in section 17.10.

## 17.7 Analyzing selection coefficients in Python with `tskit`

So far we have used tree-sequences in SLiM-agnostic ways, just loading tree sequences, overlaying mutations, and looking at their heights and ancestry. We will now start exploring uses of the `tskit` and `pyslim` packages that are more SLiM-specific, related to the metadata stored in the `.trees` file generated by SLiM and annotation of tree sequences with such metadata.

Let's begin with a simple SLiM model of beneficial and neutral mutations on a uniform chromosome:

```
initialize() {
    initializeTreeSeq();
    initializeMutationRate(1e-10);
    initializeMutationType("m1", 0.5, "g", 0.1, 0.1);
    initializeMutationType("m2", 0.5, "g", -0.1, 0.1);
    initializeGenomicElementType("g1", c(m1, m2), c(1.0, 1.0));
    initializeGenomicElement(g1, 0, 1e8-1);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
20000 late() { sim.treeSeqOutput("./recipe_17.7.trees"); }
```

Beneficial mutations are drawn from a gamma DFE with a mean of 0.1, deleterious mutations from a gamma DFE with a mean of -0.1. After the model has run for a while, the expectation would be that the mutations generated as the model runs would indeed have those means, but that the mutations fixed would be biased towards the higher end of the distribution for both mutation types. This would be simple enough to evaluate in Eidos at the end of the model run, but let's do it in Python instead, using `tskit`, as a proof of concept. Here is a Python script that assumes that a `.trees` file has been saved to the path used by the model above (we've shown already how to run a SLiM model from inside Python using `subprocess.check_output()`, so there's no need to keep reiterating that point):

```
import tskit

ts = tskit.load("recipe_17.7.trees")

coeffs = []
for mut in ts.mutations():
    md = mut.metadata
    sel = [x["selection_coeff"] for x in md["mutation_list"]]
    if any([s != 0 for s in sel]):
        coeffs += sel

b = [x for x in coeffs if x > 0]
d = [x for x in coeffs if x < 0]

print("Beneficial: " + str(len(b)) + ", mean " + str(sum(b) / len(b)))
print("Deleterious: " + str(len(d)) + ", mean " + str(sum(d) / len(d)))
```

We start by loading the `.trees` file with `tskit.load()` as usual. We then loop through all of the mutations in the tree sequence, provided by `ts.mutations()`. We are interested in the selection coefficients of the mutations, which are not part of the base information provided by the `.trees` format; instead, they are stored in SLiM-specific metadata attached to each mutation. To access them, then, we ask for the metadata using the `metadata` attribute; this decodes the metadata from the binary information stored in the `.trees` file using a standard “schema”. We can then fetch selection coefficients (more on this in a moment) and append them to the `coeffs` list if they are non-neutral (just for illustration – all selection coefficients in the SLiM model will be non-neutral anyway). Once we have that list, it is a trivial matter to select the beneficial and deleterious mutations from it and print a summary of each. A test run produces this output:



Beneficial: 3580, mean 1.0382685732466397  
Deleterious: 103, mean -0.04902286211917154

A great many more beneficial mutations than deleterious mutations are present, even though they arise at equal rates in the SLiM model, which is unsurprising. Furthermore, the mean of the mutations present is considerably biased relative to the mean of the DFEs for these mutation types (which were 0.1 and -0.1).

There are a few points to note here, regarding the way in which the selection coefficients are gathered. First of all, the tree sequence retains fixed mutations, unlike SLiM. As the SLiM model ran, mutations that fixed were converted to substitutions as usual in SLiM; but in the tree sequence no such conversion occurs. The list of mutations provided by the tree sequence therefore includes both fixed and segregating mutations; no distinction is made.

Second, note that the value of the `metadata` attribute is a dictionary with named keys. We get the value for the key `"mutation_list"` from this; that is, as the name suggests, a list of dictionaries, each containing information about a SLiM mutation. We fetch a selection coefficient from each. The reason for this has to do with mutation stacking in SLiM (see section 1.5.3 for a review of this concept). A “mutation”, as far as the tree sequence is concerned, is a unique *mutational state* at a given position, which encompasses all of the mutations that have stacked at that position. When we get the metadata from `tskit`, it helpfully deconstructs this stacked state into the component mutations within it. If a particular mutation occurs in more than one configuration in the tree sequence – by itself at a position and also stacked with another mutation at that position, in different genomes, say – we will actually encounter that mutation more than once during this process, and we will therefore overcount it. Since stacking will be extremely uncommon in this model, we don’t worry about it much; it will not skew our results noticeably. If we wanted to be more rigorous, however, we could use the mutation IDs from SLiM (also available through `tskit`) to construct a unique list of mutations, with each mutation counted only once even if it is stacked with other mutations in various ways, and could then do the rest of the analysis using that unique list. Since that is just elementary Python wrangling, we will not go into that level of detail here.

Various other SLiM metadata information is also available in Python through `tskit`, such as the age and sex of individuals, the “pedigree IDs” of genomes and individuals, and the mutation type ids for mutations. Section 26.4 details the available metadata information.

## 17.8 Starting a hermaphroditic WF model with a coalescent history

In section 17.6, we saw how to assess whether a model has coalesced or not using `treeSeqCoalesced()`, allowing us to run a model for an appropriate burn-in period rather than relying upon the (problematic)  $10N$  heuristic. If the burn-in period for a model is neutral and can be run with `msprime`’s coalescent simulation, we can avoid running our burn-in with forward simulation at all. Instead, we can run the burn-in period using the coalescent, save the result as a `.trees` file (annotated with `pyslim` as needed), and load the result into SLiM where we continue the simulation from the endpoint of the coalescent. This is actually quite simple to do.

However, note that starting a simulation with the coalescent in this way is often not the best technique. In many cases, the “recapitation” technique presented in section 17.10 is superior, for reasons that will be explained there. This recipe is for primarily for illustration.

We begin with a Python script:

```
import msprime, pyslim

ts = msprime.sim_ancestry(samples=5000, population_size=5000,
    sequence_length=1e8, recombination_rate=1e-8)
```

```
slim_ts = pyslim.annotate(ts, model_type="WF", tick=1)
slim_ts.dump("recipe_17.8.trees")
```

The `msprime.sim_ancestry()` method (which has replaced the old `simulate()` method in `msprime`) is used to run a coalescent simulation with a population size of 5000 diploids, a chromosome of length 1e8, and a recombination rate of 1e-8. We do not overlay mutations onto the simulated ancestry, since we only want the coalescent history, not mutations within it (those can be overlaid later if needed). This returns a tree sequence object, but it is in `msprime`'s format; it does not have any of the metadata annotations expected by SLiM. The next step is to use `pyslim.annotate()` to add those annotations. We tell it to annotate the data for a WF SLiM model, and to align the times in the tree sequence so that the current tick has index 1; when we load the model into SLiM, it will start at tick 1 even though there are many generations of history stretching back in time. After annotating, the `.trees` file is saved.

Now we can run a SLiM model that loads it and continues the run with some non-neutral dynamics:

```
initialize() {
    initializeTreeSeq();
    initializeMutationRate(0);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 0.5, "f", 0.1);
    initializeGenomicElementType("g1", m2, 1.0);
    initializeGenomicElement(g1, 0, 1e8-1);
    initializeRecombinationRate(1e-8);
}
1 late() {
    sim.readFromPopulationFile("recipe_17.8.trees");
    target = sample(sim.subpopulations.genomes, 1);
    target.addNewDrawnMutation(m2, 10000);
}
1: late() {
    if (sim.mutationsOfType(m2).size() == 0) {
        print(sim.substitutions.size() ? "FIXED" else "LOST");
        sim.treeSeqOutput("recipe_17.8_II.trees");
        sim.simulationFinished();
    }
}
2000 early() { sim.simulationFinished(); }
```

At the end of this model (which produces fixation, ideally, but that depends upon the random number seed), a new `.trees` file is saved out. This contains the full ancestry information for the simulation – not just for the period simulated in SLiM, but also for the coalescent. Just to verify here that the full history is present, let's check the final tree sequence:

```
import tskit

ts = tskit.load("recipe_17.8_II.trees").simplify()

for tree in ts.trees():
    for root in tree.roots:
        print(tree.time(root))
```

This just prints the heights of the trees in the final tree sequence; in a test run, it prints a wide range of values, ranging from less than 10,000 to more than 60,000. The simulation is fully coalesced (that could be confirmed by checking that the number of roots per tree is always exactly 1), but it reached coalescence at different times at different positions along the chromosome, so the

tree heights are not all identical. However, all of the heights are considerably larger than the 324 generations that the SLiM model ran for; the rest is the height of the coalescent history, which has indeed been preserved. We could now overlay neutral mutations upon the final `.trees` file as we did in section 17.2, or perform ancestry analyses with it such as those we did in section 17.4 and 17.5, or whatever else we might wish to do.

## 17.9 Starting a sexual nonWF model with a coalescent history

In the previous recipe, we saw how to run an `msprime` coalescent simulation as a burn-in period and save it as a SLiM-compliant `.trees` file which could then be used to run non-neutral dynamics on top of the coalescent burn-in history. The scenario presented there was very simple, however: a hermaphroditic WF model. What if we want to use `msprime` to construct a coalescent burn-in for a more complex model? We will need to annotate the tree sequence appropriately for the type of model that we intend to load it into in SLiM. Here we will see how to do this for a sexual nonWF model. As mentioned in the previous recipe, however, starting a simulation with the coalescent is often not the best technique; the “recapitation” technique presented in section 17.10 is usually preferable, for reasons discussed there. This recipe is offered for illustration of the relevant techniques.

To get SLiM to accept the `.trees` file from the burn-in as legitimate input for a sexual nonWF model, we will need to assign sexes to each individual. We will assign ages too; if we didn’t do so, all of the individuals would be given a default age of 0, but here we would like the initial population to have some age structure. Finally, we will tell `pyslim` to mark the tree sequence as being from a nonWF simulation, so that it matches SLiM’s expectations.

Here’s the first part of the Python script, which does the `msprime` simulation and `pyslim` annotation that we’ve discussed so far:

```
import msprime, pyslim, random
import numpy as np

ts = msprime.sim_ancestry(samples=5000, population_size=5000,
                          sequence_length=1e8, recombination_rate=1e-8)

tables = ts.dump_tables()
pyslim.annotate_tables(tables, model_type="nonWF", tick=1)

individual_metadata = [ind.metadata for ind in tables.individuals]
for md in individual_metadata:
    md["sex"] = random.choice(
        [pyslim.INDIVIDUAL_TYPE_FEMALE, pyslim.INDIVIDUAL_TYPE_MALE])
    md["age"] = random.choice([0, 1, 2, 3, 4])

ims = tables.individuals.metadata_schema
tables.individuals.packset_metadata(
    [ims.validate_and_encode_row(md) for md in individual_metadata])
```

As in section 17.8, we start by running a coalescent simulation with `msprime`, generating a tree sequence. We want to modify the individual metadata here, so at this point we need to switch to `msprime`’s table representation with `ts.dump_tables()`, which unpacks the tree sequence into modifiable tables (tree sequences are immutable, for efficiency reasons, so modifications must be made to tables). We can then perform the default nonWF annotation on those tables using `pyslim.annotate_tables()`; this is parallel to our use of `pyslim.annotate()` in section 17.8. We then extract a list of individual metadata records from the tables, and loop through it setting

random sexes and ages. (The age structure used here is just a placeholder for a more realistic distribution, of course.)

We want our SLiM simulation to execute a selective sweep starting from this neutral history. In section 17.8, we did this by adding a new beneficial mutation to one of the genomes in SLiM just after loading the `.trees` file. Here, for the sake of introducing new techniques, we will add the new beneficial mutation in Python instead, by further modifying the tree-sequence tables:

```
ind_id = random.choice(range(tables.individuals.num_rows))
node_id = random.choice(np.where(tables.nodes.individual == ind_id)[0])
node = tables.nodes[node_id]
mut_metadata = {
    "mutation_list": [{
        "mutation_type": 2,
        "selection_coeff": 0.1,
        "subpopulation": node.population,
        "slim_time": int(tables.metadata['SLiM']['tick'] - node.time),
        "nucleotide": -1
    }]
}
site_num = tables.sites.add_row(position=5000, ancestral_state='')
tables.mutations.add_row(
    node=node_id,
    site=site_num,
    derived_state='1',
    time=node.time,
    metadata=mut_metadata)
```

This is fairly straightforward (note that the snippet above is really only six lines of code; a lot of line-wrapping is going on due to long lines). First we select a random extant individual from the individuals table with `random.choice()`, and then choose a random node from that individual (one of its two genomes, in SLiM terminology) by choosing a random node id and then looking up the corresponding node object from the nodes table. (All of this low-level machinery for tree sequence tables is discussed in detail in the `tskit` documentation.)

Next we put together a metadata dictionary for the new beneficial mutation we want to create. It has a mutation list containing one dictionary, for the one mutation we will make. That dictionary provides the mutation type (it will be of type `m2` in SLiM), selection coefficient, and so forth. We mark it with the time of the node it will be attached to, since that mutation conceptually arose when the chosen individual was born.

Now we're ready to create the mutation. We add an entry to the sites table, providing the position the mutation occurs at, and an entry to the mutations table that gives the node, site, time, and metadata for the new mutation. The value for `derived_state` is `'1'`, which gives the ID of this mutation as represented in SLiM (with the `id` property of `Mutation`). Since there are no other mutations in this tree sequence, 1 is fine, but if other mutations were present you would need to choose a new unique value to avoid a collision with an existing mutation; in that case, choosing the largest value already used, plus 1, would suffice.

Now that the tables have been modified, they need to be used to create a new SLiM-annotated tree sequence, which will be written out to a `.trees` file:

```
slim_ts = table.tree_sequence()
slim_ts.dump("recipe_17.9.trees")
```

This is the typical workflow when modifying metadata: convert to tables, fetch the metadata from the tables, modify it, load it back into the tables, and finally recreate a tree sequence from the

modified tables. This is necessary because the tree sequence is an immutable object, not designed to allow modification of this sort.

Having produced a `.trees` file from the steps above, we can load it into a matching SLiM model and run, as we did before. For clarity, we'll leave everything else the same about this model, so that the essential differences can be seen more easily:

```
initialize() {
    initializeSLiMModelType("nonWF");
    initializeTreeSeq();
    initializeSex("A");
    initializeMutationRate(0);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeMutationType("m2", 0.5, "f", 0.1);
    m2.convertToSubstitution=T;
    initializeGenomicElementType("g1", m2, 1.0);
    initializeGenomicElement(g1, 0, 1e8-1);
    initializeRecombinationRate(1e-8);
}
reproduction(NULL, "F") {
    subpop.addCrossed(individual, subpop.sampleIndividuals(1, sex="M"));
}
1 early() {
    sim.readFromPopulationFile("recipe_17.9.trees");
}
early() {
    p0.fitnessScaling = 5000 / p0.individualCount;
}
1: late() {
    if (sim.mutationsOfType(m2).size() == 0) {
        print(sim.substitutions.size() ? "FIXED" else "LOST");
        sim.treeSeqOutput("recipe_17.9_II.trees");
        sim.simulationFinished();
    }
}
2000 early() { sim.simulationFinished(); }
```

Since this is a nonWF model, we have added a minimal `reproduction()` callback and an `early()` event that provides population regulation through density-dependence, as well as a call to `initializeSLiMModelType()` (see chapter 16 for discussion). Since it is a sexual model, we also added a call to `initializeSex()`.

The `m2` mutation type defined here is used by the mutation we added in Python; we have to define `m2` here so that SLiM has something to hang its hat on, but the DFE for `m2` is unimportant, and need not match the selection coefficient of the `m2` mutation from the `.trees` file. This is because the DFE for a mutation type is used only for creating new mutations, and no further `m2` mutations will be created by SLiM. We want the model to terminate upon fixation or loss, as before, but in nonWF models mutations are not converted to substitutions automatically, so we add `m2.convertToSubstitution=T` to ensure that; of course we could just adapt the termination code to check the frequency of the introduced mutation instead. The rest of the model is unchanged from section 17.8, apart from filenames.

Note that the subpopulation loaded in from the `.trees` file is `p0`, not `p1`; `msprime` starts counting subpopulation IDs from zero. It is conventional in SLiM, in general, to number subpopulations from `p1`, but of course it doesn't really matter; it would be possible to reassign the subpopulation index before writing out the `.trees` file, or to use the remapping facility provided by `readFromPopulationFile()`'s `subpopMap` parameter, but we do not do so here.

The only important caveat here is that the coalescent was not run with separate sexes, nor with overlapping generations, so it does not precisely reflect the dynamics involved in the non-neutral portion of the simulation. The coalescent is an approximation anyway; in fact, this would have been a concern in the previous recipe too, if exact Wright–Fisher dynamics were needed. Whether these sorts of deviations from exact SLiM dynamics are a concern or not will depend upon the nature of the analysis to be conducted downstream. If it is an issue, a further burn-in period could be run in SLiM with the correct dynamics, prior to introducing the sweep mutation; that burn-in would perhaps not need to be very long to wipe away any important traces of the incorrect burn-in dynamics (but it would be a good idea to confirm that with appropriate tests, of course). If absolutely exact dynamics are needed then the burn-in will have to be simulated in SLiM; that can still be done with tree-sequence recording with neutral mutations turned off, though, as in section 17.1, so it will still be much faster than a regular SLiM burn-in would have been.

When this model is run, we see the sweep occur (if the mutation isn’t lost), and then a new `.trees` file is written out. We could conduct the same post-run analysis as before, printing out tree heights to verify that this burn-in procedure worked, but to avoid repetitiveness we will just end here. It is worth noting, in closing, that `pyslim` allows all sorts of annotation; it would be possible, with a similar strategy, to mark genomes as being X or Y chromosomes and then load them into a sex-chromosome simulation in SLiM, or to set the spatial positions of individuals before loading them into a spatial SLiM model, or whatever is needed. Of course, getting coalescent simulation results for such complex scenarios, even without any selected loci, may be a challenge.

## 17.10 Adding a neutral burn-in after simulation with recapitation

Very often, in forward genetic simulation, a “burn-in” period of neutral dynamics is desirable to allow the model to reach an equilibrium state of mutation–drift balance before non-neutral dynamics begin. Without a burn-in period, the pattern of neutral mutations observed at the end of the simulation may depend as much upon the initial state of the model as on the model’s dynamics, which can make it difficult to interpret results. The modeling of this burn-in period is a persistent problem, however, because it is generally so time-consuming. An often-quoted rule of thumb is that the burn-in period should run for  $10N$  generations – ten times the initial population size. For a model with 100,000 individuals, then, a million generations of burn-in is recommended, which – with such a large population size – will take an exceedingly long time. Worse, the  $10N$  rule is often an underestimate of the time needed to equilibrate, particularly when simulating a very long chromosome; and there is no number of generations at which coalescence is guaranteed.

To cope with this issue, a wide variety of techniques are attempted. One might try to rescale the model during the burn-in period (see section 5.5), although this can introduce significant artifacts; or one might initialize the model with the output from a coalescent simulation (see sections 17.8 and 17.9); or one might run the burn-in period without neutral mutations, using tree-sequence recording to preserve the ancestry information that will allow them to be overlaid later (see sections 17.1 and 17.2). With SLiM 3.1, there is an option that is often better than any of these, which we will examine in this section: recapitation.

Recapitation is the addition of a neutral coalescent history to a simulation *after the fact*. The non-neutral portion of the simulation is run first, in SLiM, with tree-sequence recording enabled so that the genealogical history of all extant individuals is preserved. The result is saved to a `.trees` file, as in previous recipes. That `.trees` file is then loaded in Python, and `msprime` and `pyslim` are used to construct a coalescent history stretching back in time from the original ancestors of the simulation. The details of this are fairly conceptually complex; the `pyslim` manual has excellent



discussion of both the concepts and the practical implications, in both its Introduction and its Tutorial. That is recommended reading for SLiM users doing recapitation (or any other work involving tree sequences, in fact); see section 1.10 for relevant links. In general, if your forward simulation does not need to actually depend upon any state from the burn-in period (converting an existing neutral mutation from the burn-in into a selected mutation, say; see below for discussion of such cases), and if the burn-in period is completely neutral and non-spatial, recapitation can typically be used for greater speed.

Enough discussion; let's see it in action. We will begin with a SLiM model of non-neutral dynamics, specifically a selective sweep:

```
initialize() {
    initializeTreeSeq(simplificationRatio=INF);
    initializeMutationRate(0);
    initializeMutationType("m2", 0.5, "f", 1.0);
    m2.convertToSubstitution = F;
    initializeGenomicElementType("g1", m2, 1);
    initializeGenomicElement(g1, 0, 1e6 - 1);
    initializeRecombinationRate(3e-10);
}
1 late() {
    sim.addSubpop("p1", 1e5);
}
100 late() {
    sample(p1.genomes, 1).addNewDrawnMutation(m2, 5e5);
}
100:10000 late() {
    mut = sim.mutationsOfType(m2);
    if (mut.size() != 1)
        stop(sim.cycle + ": LOST");
    else if (sum(sim.mutationFrequencies(NULL, mut)) == 1.0)
    {
        sim.treeSeqOutput("recipe_17.10_decap.trees");
        sim.simulationFinished();
    }
}
```

This is a pretty typical tree-sequence-based model, with a mutation rate of zero. It involves a fairly large population size (1e5), and a reasonably long chromosome (1e6), so running the neutral burn-in for this model in SLiM would take quite a long time. Incidentally, the size of this simulation also means that simplification can take quite a long time, and so as well as recapitating, we have also chosen to speed up the simulation by setting the `simplificationRatio` parameter of `initializeTreeSeq()` to `INF`. We have not discussed the `simplificationRatio` option much before: in general, it controls how often simplification occurs, and a value specifically of `INF` tells SLiM not to simplify the tree sequence at all until a `.trees` file is generated (see section 24.1). Although this speeds up the model's execution considerably, it can greatly increase memory usage, so it should be done with care; it is easy to overflow the memory available to the process. As this particular example has a short runtime, we're probably safe to trade off memory to achieve maximum simulation speed, but for longer-running simulations you will probably want to tune the simplification interval as appropriate before recapitating.

This model involves a selective sweep, introduced in tick 100. We do not attempt to run this simulation conditional on fixation (see chapter 9); for simplicity, we just detect fixation or loss in the 100:10000 `late()` event. On fixation, the model produces a `.trees` file as output and stops.

So now, if we run the model repeatedly until we get a run in which the sweep mutation fixes, we have a file named `recipe_17.10_decap.trees` that contains the genealogical history of that run. Now we run a Python script that performs the recapitulation:

```
import msprime, pyslim
import numpy as np
import matplotlib.pyplot as plt

# Load the .trees file
ts = tskit.load("recipe_17.10_decap.trees")    # no simplify!

# Calculate tree heights, giving uncoalesced sites the maximum time
def tree_heights(ts):
    heights = np.zeros(ts.num_trees + 1)
    for tree in ts.trees():
        if tree.num_roots > 1: # not fully coalesced
            heights[tree.index] = ts.metadata['SLiM']['tick']
        else:
            children = tree.children(tree.root)
            real_root = tree.root if len(children) > 1 else children[0]
            heights[tree.index] = tree.time(real_root)
    heights[-1] = heights[-2] # repeat the last entry for plotting
    return heights

# Plot tree heights before recapitulation
breakpoints = list(ts.breakpoints())
heights = tree_heights(ts)
plt.step(breakpoints, heights, where='post')
plt.show()

# Recapitate!
recap = pyslim.recapitate(ts, ancestral_Ne=1e5, recombination_rate=3e-10,
    random_seed=1)
recap.dump("recipe_17.10_recap.trees")

# Plot the tree heights after recapitulation
breakpoints = list(recap.breakpoints())
heights = tree_heights(recap)
plt.step(breakpoints, heights, where='post')
plt.show()
```

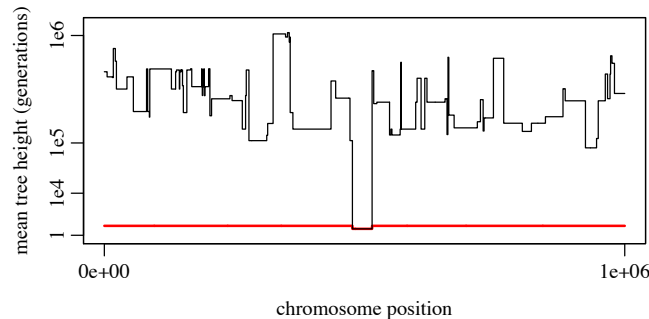
The first step is to load the `.trees` file. Note that we specifically do not call `simplify()` here, because we need the first generation individuals to recapitate from; this is, in fact, precisely why SLiM preserves those individuals for us by simplifying with the `keep_input_roots=True` flag.

Then we define a function that calculates tree heights along the chromosome. This is similar to what we did in section 17.4, but here we have to be a bit smarter because of those original ancestors preserved in the tree sequence. Every tree will have a root in one of those original ancestors, but we are interested in whether the tree has coalesced below that original ancestor (and if so, at what height), or if the tree still has multiple roots, indicating that it has not yet coalesced. We therefore check the number of roots for the tree, and if there is just a single root, we use the height of its child when the root has only a single child (indicating that coalescence occurred in the child, not in the original ancestor).

We use that function to plot tree heights along the chromosome before recapitulation. Then we recapitate, which is very easy – just a single method call on our tree sequence, returning a new tree sequence that has had a coalescent history traced backward from its original ancestors. The

recapitation process just needs to know the recombination rate to use, and the population size ( $N_e$ ) to use for the coalescent.

Finally, we plot again, after recapitation, to see what it has done. A combined plot of these results, made in R, looks like this:



Note that the y-axis is rescaled according to the cube root of the number of generations, to bring out detail at both the low and high ends of the scale. The red line here shows the tree heights along the chromosome prior to recapitation. The area surrounding the sweep has coalesced at the generation in which the sweep mutation was introduced, due to strong hitchhiking in the vicinity of the mutation. The area further out has not coalesced, and therefore has a tree height that dates back to the beginning of forward simulation, 100 generations before the sweep began. The black line shows tree heights after recapitation; here the uncoalesced regions farther from the sweep have been coalesced backward in time as far as a million generations (reflecting the fact that we would have needed to run the burn-in in SLiM for at least a million generations to have much likelihood of coalescence). Regions closer to the sweep tend to coalesce more recently, reflecting the effect of the sweep on the diversity present at different regions along the chromosome at the end of simulation.

In short, we can see that recapitation has provided us with a full neutral burn-in history for the simulation, after the fact. Notably, this is very fast; this example run executed in 0.41 seconds. If we wanted neutral mutations overlaid over the whole population history, including the recapitated burn-in, that would also be extremely fast; using the technique shown in section 17.2, that operation would take another 0.58 seconds. These times can be compared to an estimate, obtained by extrapolation, of how long the burn-in would take in SLiM: more than 114 hours.

Recapitation, then, is optimal for computing burn-in for several reasons. One reason is that it is extremely fast, as we have seen; the only branches that need to be coalesced are those leading to the final individuals present at the end of forward simulation, which is generally a small minority of the branches that were present at the beginning of forward simulation. Recapitation is therefore much, much faster than simulating the burn-in period in SLiM, and should even be faster than constructing an initial population state with the coalescent; it is based upon the coalescent, but needs to do much less work since far fewer branches typically need to be coalesced. Then too, recapitation is very convenient, since it can be done after forward simulation, even as an afterthought on existing model output. In this way, it allows a focus on the non-neutral dynamics of interest, with burn-in handled later. It also allows one to ignore the question of how long to run burn-in for – the whole “10N” question – since recapitation will coalesce back in time as far as necessary to achieve full coalescence.

Of course, recapitation can only be used in scenarios where a neutral coalescent process is appropriate for burn-in. In some cases the burn-in period needs to itself be non-neutral; in this case using tree-sequence recording to run without neutral mutations may be the best one can do

(see sections 17.1 and 17.2). But for many models, recapitation will enable modeling at a larger scale than previously possible, by greatly reducing the overhead of the burn-in period.

A caveat that comes with the above recipe is that it recapitates a WF model, which means that the simulation involves non-overlapping generations. Recapitation can be used with nonWF models too, but if the model involves overlapping generations, the timescale for recapitation needs to be matched carefully to the timescale of the forward simulation by rescaling the recapitation parameters as appropriate, since recapitation is performed assuming time units of “generations”. Note that if your nonWF model does in fact simulate non-overlapping generations, you can signal that to `tskit/msprime` by passing `timeUnit="generations"` to `initializeTreeSeq()`; this may allow you to avoid a warning about timescale issues when you recapitate. (Of course, doing that if the time unit is *not* generations would result in incorrect recapitation results.) The upshot is: think carefully about time units, or you risk getting an incorrect result from recapitation.

One point was glossed over in the discussion above: why was the sweep mutation introduced in tick 100, and not, say, in tick 2 immediately after the simulation has started? The easy answer is: to make the plot look nice. In the mean tree height plot above, there is a nice stair-step in the red line (showing mean tree height prior to recapitation), and that bump is there because of the delay between starting forward simulation and introducing the sweep mutation. But there is a deeper benefit as well: running a little bit of forward simulation after recapitation can smooth out differences introduced by the coalescent burn-in. The standard Kingman coalescent, as simulated here via the `recapitate()` command, produces evolutionary dynamics that are extremely similar to a neutral forward Wright–Fisher simulation such as the model shown above; but the two are not, in fact, exactly identical, and small differences introduced by the use of the coalescent could conceivably produce detectable bias in simulation results if non-neutral dynamics commenced immediately after the end of the coalescent. Doing a little extra neutral burn-in after the recapitated period shuffles things around so that any such bias is minimized.

Indeed, this technique can sometimes be used even to approximate a non-neutral burn-in period using the coalescent; one can forward-simulate the non-neutral burn-in dynamics in SLiM for some relatively short period of time (like 100 or 1000 generations), and then add a coalescent history to that using recapitation as a sort of convenient approximation to the truth. This is obviously not ideal; but if forward-simulating the full non-neutral burn-in period would take a prohibitively long time, it may be the only option available, and it may, for some models, prove to produce acceptable results. Of course any approximation like this should be carefully tested to ensure that any bias or artifacts introduced by it are within acceptable bounds.

If the forward simulation actually needs to do something with the mutations that exist from the burn-in period, recapitation cannot be used. For example, you might want to model a soft sweep in which a neutral mutation from “deep time”, back in the burn-in period, becomes non-neutral due to an environmental change. The non-neutral time period must be forward-simulated in SLiM, as usual. The burn-in needs to be simulated *first*, with the coalescent in `msprime`, and then needs to have mutations overlaid onto it before forward simulation begins, so that those overlaid mutations get imported into SLiM for its use. The `pyslim` documentation has a nice example of this technique at [https://tskit.dev/pyslim/docs/stable/vignette\\_coalescent\\_diversity.html](https://tskit.dev/pyslim/docs/stable/vignette_coalescent_diversity.html).

Finally, it is worth noting that the recapitation technique described here, to add a neutral simulation period *prior* to the time period run by forward simulation, has a complementary technique that we could call “continuation”: adding a neutral simulation period *subsequent* to the time period run by forward simulation. This could be useful, for example, to look at the genomic signature left behind by a selective sweep, not immediately after the sweep has completed, but many generations later, to understand how the initial sweep signature becomes obscured by subsequent drift and mutation. We will not present a continuation recipe here, but the `pyslim` documentation has a vignette at [https://tskit.dev/pyslim/docs/stable/vignette\\_continuing.html](https://tskit.dev/pyslim/docs/stable/vignette_continuing.html).

## 18. Modeling explicit nucleotides

So far, the models we have explored have generally involved an abstracted concept of mutations in which mutations have properties like their selection coefficient and their position in the genome, but do not explicitly represent a change in the nucleotide sequence. In section 1.8, however, it was mentioned that SLiM (in version 3.3 and later) supports models that track an explicit nucleotide sequence. That section laid out many of the foundational concepts for such models, and should be considered required reading as a prelude to this chapter. In this chapter, we will look at examples of nucleotide-based models. Most of the concepts we have explored in previous chapters continue to apply in nucleotide-based models; however, there are some major differences (such as the mutational model), and there are some quirks and surprises.

### 18.1 A simple neutral nucleotide-based model

We will begin with a simple neutral WF model, very similar to the models we have seen before, but converted to be nucleotide-based:

```
initialize() {
  defineConstant("L", 1e6);
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(L));
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-7));
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
}
2000 late() { sim.outputFixedMutations(); }
```

There are several important changes here. First of all, we tell SLiM up front that the model is nucleotide-based, by passing `nucleotideBased=T` to `initializeSLiMOptions()`. This has various downstream consequences, as we will see.

Next, we call `initializeAncestralNucleotides()` to supply an ancestral nucleotide sequence. This call is required in all nucleotide-based models; SLiM needs to know the ancestral sequence in order to correctly handle mutations, which are based upon the existing nucleotide sequence as we will see below, so omitting it produces an error. Here we pass a random nucleotide sequence generated by the `randomNucleotides()` function. This function simply generates a random sequence of a given length, with equal probabilities for the four bases by default (or a vector of probabilities may be supplied). For example, in the Eidos console:

```
> randomNucleotides(50)
"GAAGAATGAAATTTGCGAGGCTATGTCTAGCATATGCCACCGCCCTGACC"
```

Next we create a new mutation type, representing neutral mutations. We want this mutation type to be nucleotide-based, so we call `initializeMutationTypeNuc()` instead of the usual call to `initializeMutationType()`. The parameters to `initializeMutationTypeNuc()` are the same as for `initializeMutationType()`; it simply creates a nucleotide-based mutation type instead of a non-nucleotide-based mutation type. (It is legal to create non-nucleotide-based mutation types in a nucleotide-based model, as we will see in section 18.6; the two can be freely mixed in the same model.)

Then we set up a new genomic element type by calling `initializeGenomicElementType()`. Here, this call has an extra parameter at the end that we have never used before. The name of that parameter is `mutationMatrix`, and we pass the result of a function call, `mmJukesCantor(1e-7)`, for it. Let's look at what that function call returns, again in the Eidos console in SLiMgui:

```
> mmJukesCantor(1e-7)
      [,0] [,1] [,2] [,3]
[0,]    0 1e-07 1e-07 1e-07
[1,] 1e-07    0 1e-07 1e-07
[2,] 1e-07 1e-07    0 1e-07
[3,] 1e-07 1e-07 1e-07    0
```

This is a 4×4 matrix of type `float`, representing a specific instance, for  $\alpha = 1e-7$ , of the Jukes–Cantor (1969) mutational model with a mutation rate  $\alpha$  to each alternative nucleotide at a given site (see section 24.17.1):

$$\begin{pmatrix} 0 & \alpha & \alpha & \alpha \\ \alpha & 0 & \alpha & \alpha \\ \alpha & \alpha & 0 & \alpha \\ \alpha & \alpha & \alpha & 0 \end{pmatrix}$$

We haven't seen matrices in Eidos much in this manual, but matrices (and more generally arrays) are built into the Eidos language, and work in much the same way as they do in R (see the Eidos manual for details). This matrix specifies sequence-specific mutation rates that will be used within genomic elements of this genomic element type. Each row of the matrix represents one possible sequence state prior to mutation; there could be an A, C, G, or T present at the mutating site, represented by rows 0 through 3 respectively. Each column provides the corresponding mutation rates for the mutation of a given state to a new state: A, C, G, or T, as represented by columns 0 through 3 respectively. For a mutating site that presently has a C in it (row 1), for example, the probabilities of mutating to an A, G, or T are all given as  $1e-7$  (columns 0, 2, and 3). The probability of an “identity” substitution, of a nucleotide by itself, must conventionally be specified as zero, as it is here (the zero values along the matrix diagonal). Note that the *overall* mutation rate in this model, per base position per gamete, will be  $3e-7$ , not  $1e-7$ ;  $\alpha$  is not the overall mutation rate, but rather the rate at which a given nucleotide will mutate to a specific alternative nucleotide.

Any mutation matrix may be supplied to SLiM. The Jukes–Cantor model involves equal probabilities of mutation away from and toward every nucleotide state, but other mutational models can be specified instead with, e.g., different rates for transitions versus transversions. It is also possible to supply a 64×4 matrix instead of a 4×4 matrix, specifying the mutation rates for the central nucleotide of every possible trinucleotide sequence (of which there are 64); this allows mutational models in which, for example, CpG sites display a higher rate of mutation than other sites (see section 18.5). This generality means that the realized mutation rate may actually be sequence-dependent; a CpG-rich genome may mutate more rapidly than a CpG-poor genome, for example. SLiM will manage these mutational details accurately, producing the mutation rates requested by the matrix supplied (using a rejection sampling method under the hood).

It is also worth noting that the mutation matrix is not a global parameter, but a property of the genomic element type. It is possible, therefore, to vary the mutation matrix along the chromosome. One might wish the mutational properties of non-coding regions to differ from those of coding regions, for example, if the process of transcription is thought to be mutagenic within coding regions. That would be easily implemented using different genomic element types for coding versus non-coding regions, utilizing different mutation matrices.



Conspicuous by its absence here is any call to `initializeMutationRate()`. We already configured the mutation rate with `mutationMatrix`; in fact, calling `initializeMutationRate()` in nucleotide-based models is illegal. It is still possible to configure non-sequence-based variation in the mutation rate along the chromosome (i.e., mutational hot spots and cold spots), using a “hotspot map” as we will see in section 18.10. Since we don’t need a hotspot map in this model, though, the mutation rate is already fully configured.

The rest of the model contains no surprises. We set up the chromosome with a single `g1` genomic element, set the recombination rate, create a new subpopulation of size 500, run to tick 2000, and output fixed mutations. The output of the model looks like this:

```
#OUT: 2000 2000 F
Mutations:
0 11796 m1 372475 0 0.5 p1 53 782 C
1 1698 m1 393353 0 0.5 p1 8 782 G
2 1842 m1 393686 0 0.5 p1 9 782 G
3 23624 m1 413964 0 0.5 p1 106 782 C
4 41111 m1 416291 0 0.5 p1 183 782 T
...
```

Note that each fixed mutation that is output here includes a designation of its nucleotide, at the end of the output line; that output field is added when `outputFixedMutations()` is called from a nucleotide-based model. Many of the other built-in output methods in SLiM similarly provide additional nucleotide information in nucleotide-based models, as we will see in later sections. Chapter 26 provides complete reference documentation for all output formats.

That is all there is to our introductory nucleotide-based model. We will explore some subtle twists and complications in the rest of this chapter, but here we have seen the fundamentals of defining a nucleotide-based model: declaring it with `nucleotideBased=T`, providing an ancestral nucleotide sequence with `initializeAncestralNucleotides()`, designating nucleotide-based mutation types with `initializeMutationTypeNuc()`, and setting up sequence-based mutation rates with the `mutationMatrix` parameter to `initializeGenomicElementType()`.

## 18.2 Reading an ancestral nucleotide sequence from a FASTA file

In the previous section we looked at a trivial nucleotide-based model that generated an ancestral sequence using the `randomNucleotides()` function. Often the ancestral sequence will instead come from a FASTA file; this use case is common enough that it is directly supported by SLiM. Modifying the previous recipe, we can do:

```
initialize() {
    initializeSLiMOptions(nucleotideBased=T);
    defineConstant("L", initializeAncestralNucleotides("FASTA.txt"));
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0, mmKimura(1.8e-07, 6e-08));
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
2000 late() { sim.outputFixedMutations(); }
```

A path to the FASTA file is passed to `initializeAncestralNucleotides()`; that file happens, here, to contain the following data:

```

>HSBGP Human gene for bone gla protein (BGP)
GGCAGATTCCCCCTAGACCCGCCCGCACCATTGGTCAGGCATGCCCCCTCCTCATCGCTGGGCACAGCCAGAGGGT
ATAAACAGTGCTGGAGGCTGGCGGGCAGGCCAGCTGAGTCTGAGCAGCAGCCAGCGCAGCCACCGAGACACC
ATGAGAGCCCTCACACTCTCGCCCTATTGGCCCTGGCCGCACTTTGCATCGCTGGCCAGGCAGGTGAGTGCCCC
CACCTCCCCTCAGGCCGATTGCACTGGGGCTGAGAGGAGGAAGCACCATGGCCCACTCTTCTCACCCCTTTG
GCTGGCAGTCCCTTTGCACTAACCACCTTGTGTCAGGCTCAATCCATTGCCCCAGCTCTGCCCTTGCAAGG
GAGAGAGGGAAGAGCAAGCTGCCCCGAGACGAGGGGAAGGAGGATGAGGGCCCTGGGGATGAGCTGGGGTGAAC
CAGGCTCCCTTTCTTTGCAAGTGCGAAGCCAGCGGTGCAGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACC
TGATGGGTTCTGGACCCTCCCTCTCACCTGGTCCCTCAGTCTCATTCCCCACTCTGCCACCTCCTGTCTG
GCCATCAGGAAGGCCAGCCTGCTCCCCACCTGATCTCCCAAACCCAGAGCCACCTGATGCCTGCCCTCTGTCTC
CACAGCCTTTGTGTCAAGCAGGAGGGCAGCGAGGTAGTGAAGAGACCCAGGCGCTACCTGTATCAATGGCTGGG
GTGAGAGAAAAGGCAGAGCTGGGCCAAGGCCCTGCCTCTCCGGGATGGTCTGTGGGGGAGCTGCAGCAGGGAGTG
GCCTCTCTGGGTGTGGTGGGGGTACAGGCAGCCTGCCCTGGTGGGCACCCCTGGAGCCCCATGTGTAGGGAGAGG
AGGGATGGGCATTTTGACGGGGGGTGTGCCACACGTCGGGTGTCTCAGAGCCCCAGTCCCCTACCCGGATCC
CCTGGAGCCCAGGAGGGAGGTGTGTGAGCTCAATCCGAGCTGTGACGAGTTGGCTGACCACATCGGCTTTTCAGGA
GGCCTATCGGCGCTTCTACGGCCCGGTCTAGGGTGTCTGCTCTGGCTGGCCGCGCAACCCAGTTCTGCTCCT
CTCCAGGCACCTTCTTCTCTTCCCTTGGCCCTGGCCCTGACCTCCAGCCCTATGGATGTGGGGTCCCCATC
ATCCAGCTGCTCCCAATAAACTCCAGAAG
>HSLTH1 Human theta 1-globin gene
CCACTGCACTCACCGCACCCGGCCAATTTTGTGTTTTAGTAGAGACTAAATACCATATAGTGAACACCTAAGA
CGGGGGCCTTGGATCCAGGGCGATTGAGAGGGCCCGGTGCGAGCTGTGCGAGATTGAGCGCGCGCGGTCCCGG
GATCTCCGACGAGGCCCTGGACCCCGGGCGGCGAAGCTGCGGCGCGCGGCCCTGGAGGCCGCGGGACCCCTG
GCCGGTCCGCGCAGGCGCAGCGGGTGCAGGGCGCGGGGTTCAGCGCGGGGATGGCGCTGTCCGCGGAGGA
CCGGGCGCTGTGCGCGCCCTGTGGAAGAAGCTGGGCAGCAACGTCGGCGTCTACACGACAGAGGCCCTGGAAG
GTGCGGCAGGCTGGGCGCCCCGCCCCAGGGGCCCTCCCTCCCCAAGCCCCCGGACGCGCCTACCCACGTTT
CTCTCGCAGGACCTTCTGGCTTTCCCGCCACGAAGACCTACTTCTCCACCTGGACCTGAGCCCCGGCTCCTC
ACAAGTCAGAGCCCACGGCCAGAAGGTGGCGGACGCGCTGAGCCTCGCCGTGGAGCGCCTGACGACCTACCCCA
CGCGTGTCCGCGTGAACACCTGCACGCGTGCCAGCTGCGAGTGGACCCGGCCAGCTTCCAGGTGAGCGGCTG
CCGTGCTGGGCCCCTGTCCCGGGAGGGCCCCGGCGGGTGGGTGCGGGGGCGTGGGGGCGGTGCAGGCGAG
TGAGCCTTGAGCGCTCGCCGAGCTCTGGCCACTGCTGCTGTAACCTCGCCCGGCACTACCCCGGAGACT
TCAGCCCCGCGTGCAGGCGTCGCTGGACAAGTTCCTGAGCCACGTTATCTCGGCGCTGGTTTCCGAGTACCGCT
GAAGTGTGGTGGGTGGCGCGGGATCCCCAGGCGACCTTCCCGTGTGTTAGTAAAGCCTCTCCAGGAGCAGC
CTTCTTCCGTGCTCTCTGAGGTGAGGACGCGAGAGGAAGGCGC

```

Note that this file contains two sequences; in such cases, `initializeAncestralNucleotides()` always uses the first sequence defined in the file. The model needs to know the chromosome length (here defined as the constant `L`); `initializeAncestralNucleotides()` returns the length of the ancestral sequence it is given, so we use that to define `L` correctly without knowing the length of the FASTA sequence ahead of time.

The FASTA format allows specifications such as `R` for a purine (A or G), as well as amino-acid specifications such as `F` for Phenylalanine; those are not allowed in FASTA files read by SLiM. Only the four DNA nucleotide bases – A, C, G, and T – are allowed. Lines may be of any length. Description lines must begin with `>` or `;`, and may contain any text; they are ignored.

For fun, we have also switched the mutational model from Jukes–Cantor (1969) to Kimura (1980), which allows transition and transversion rates to differ. The `mmKimura()` function takes two parameters, `alpha` and `beta`, for the transition and transversion rates desired (see section 24.17.1). Here the values chosen will again produce an overall mutation rate of  $3e-7$ , as in section 18.1, since for the Kimura (1980) model the overall rate is  $\alpha + 2\beta$ . However, the transition/transversion rate ratio  $\kappa = \alpha/\beta$  is 3.0 here – a strong departure from the Jukes–Cantor model where  $\kappa = 1.0$ . (Note that  $\kappa$  is the transition/transversion *rate* ratio, not the transition/transversion ratio!)

### 18.3 Sequence output from nucleotide-based models

When doing a nucleotide-based simulation, one often wants to obtain nucleotides for the mutations in the simulation, as well as the nucleotide sequences of particular genomes. In section 18.1 we saw that `outputFixedMutations()` provides the nucleotides for fixed mutations in nucleotide-based models; `outputMutations()`, `outputFull()`, etc. do as well. Here we will explore several other ways of getting this type of information, with a very simple recipe:

```

initialize() {
  defineConstant("L", 10);
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(L));
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(2.5e-5));
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);
}
1 early() {
  c = sim.chromosome;
  catn("Ancestral: " + c.ancestralNucleotides());
  catn("Ancestral: " + paste(c.ancestralNucleotides(format="char")));
  catn("Ancestral: " + paste(c.ancestralNucleotides(format="integer")));
  catn("positions: " + paste(0:(L-1)));
  catn();

  sim.addSubpop("p1", 500);
}
5000 late() {
  catn("Fixed:      " + paste(sim.substitutions.nucleotide));
  catn("Fixed:      " + paste(sim.substitutions.nucleotideValue));
  catn("positions: " + paste(sim.substitutions.position));
  catn();

  c = sim.chromosome;
  catn("Ancestral: " + c.ancestralNucleotides());
  catn("Ancestral: " + paste(c.ancestralNucleotides(format="char")));
  catn("Ancestral: " + paste(c.ancestralNucleotides(format="integer")));
  catn("positions: " + paste(0:(L-1)));
  catn();

  g = p1.genomes[0];

  catn("SNPs:       " + paste(g.mutations.nucleotide));
  catn("SNPs:       " + paste(g.mutations.nucleotideValue));
  catn("positions: " + paste(g.mutations.position));
  catn();

  catn("Derived:    " + g.nucleotides());
  catn("Derived:    " + paste(g.nucleotides(format="char")));
  catn("Derived:    " + paste(g.nucleotides(format="integer")));
  catn("positions: " + paste(0:(L-1)));
}

```

The model itself is trivial – a chromosome ten bases long, with a high mutation rate so that we get fixed mutations and genetic diversity quickly. The interesting thing is the output generated. In tick 1 we print the ancestral sequence that the model begins with, by calling the `ancestralNucleotides()` method on the chromosome:

```

Ancestral: TACTGTAAGC
Ancestral: T A C T G T A A G C
Ancestral: 3 0 1 3 2 3 0 0 2 1
positions: 0 1 2 3 4 5 6 7 8 9

```

By default `ancestralNucleotides()` returns a singleton string for the whole sequence, as shown by the first call. Other formats for the return value are also supported, however, such as "char" (a vector of single-character strings for the nucleotides) and "integer" (a vector of integers,

where 0=A, 1=C, 2=G, and 3=T), as shown by the next two calls. This method has several other options not shown here; you can get just a subrange of the ancestral sequence by supplying a start and/or end position, for example, and you can request the return value to be in "codon" format (integers in [0,63] representing trinucleotide groups); see section 24.2.2 for details.

After producing that initial output, the model runs for 5000 ticks and then prints several further pieces of information. The first is a list of the mutations that fixed during those 5000 ticks; those fixed mutations have been converted to `Substitution` objects as usual in SLiM, and are available through the `substitutions` property of `sim`. There are three of them in this example run. The nucleotides of the substitutions are available through the `nucleotide` property (for strings) or the `nucleotideValue` property (for integers), producing the output shown:

```
Fixed:      T T C
Fixed:      3 3 1
positions:  9 8 6
```

An important point is that when nucleotide-based mutations fix and are converted to substitutions, the ancestral sequence is updated to reflect the new ancestral state. When we print the ancestral sequence again in tick 5000, in other words, it has a T at positions 8 and 9, and a C at position 6, reflecting the substitutions that have occurred (as output earlier):

```
Ancestral:  TACTGTCATT
Ancestral:  T A C T G T C A T T
Ancestral:  3 0 1 3 2 3 1 0 3 3
positions:  0 1 2 3 4 5 6 7 8 9
```

The ancestral sequence in SLiM is *not*, in general, the sequence that the simulation started with (although initially it is); it is the reference sequence *currently* used to represent the state of a genome that contains no nucleotide-based mutations. As such, it changes when substitutions occur.

Next we select a single genome from the model (for simplicity), and we print the SNPs that are present in that genome:

```
SNPs:      G C
SNPs:      2 1
positions:  4 8
```

`Mutation`, like `Substitution`, has `nucleotide` and `nucleotideValue` properties that provide the nucleotide for the mutation as either a string or an integer. (Accessing these properties on mutations or substitutions that are not nucleotide-based produces an error, if you are wondering.)

Finally, we obtain the full nucleotide sequence for the chosen genome using the `nucleotides()` method. This method is very similar to the `ancestralNucleotides()` method we saw before; it supports the same options and formats. The only difference is that after obtaining the ancestral sequence, `nucleotides()` overlays all of the SNPs present in the genome to produce the derived sequence:

```
Derived:    TACTGTCACT
Derived:    T A C T G T C A C T
Derived:    3 0 1 3 2 3 1 0 1 3
positions:  0 1 2 3 4 5 6 7 8 9
```

The sequences output here are the same as the ancestral sequence except that the two SNPs in the genome – a G at position 4 and a C at position 8 – are overlaid.

Those paying close attention will notice a surprising thing: the ancestral sequence at position 4 is G, and yet we have a mutation to G at that position segregating in the model, as shown by the SNPs in the genome's output. But SLiM does not allow a nucleotide to mutate to itself, and in any case the Jukes–Cantor mutation matrix used here has zeroes along its diagonal to express that such mutations cannot occur, as we saw in section 18.1. So how did we get a G segregating on an ancestral background of G?

The answer is: back-mutation. At some point during the 5000 ticks of runtime, a mutation occurred at position 4 that changed it to some non-G nucleotide, but that mutation did not fix (or we would see it in the list of fixed mutations that we printed). Some time later, another mutation occurred at position 4 *on the background of the segregating non-G mutation*, changing it back to a G – which was still the base present at that position in the ancestral sequence. When back-mutation like this happens, SLiM does not remove the non-G mutation and revert the location back to the ancestral sequence by leaving it empty; instead, after removing the non-G mutation, SLiM adds a new G mutation at the position in question, just as it would had the mutation been to any other nucleotide.

This behavior may seem surprising at first glance, but it is more consistent (since a new mutation is created in all cases), and it allows back-mutations to be differentiated from sites that have not mutated at all (which can often be useful). It would be possible to “unique down” such states, by checking for new mutations in each tick that are identical to the ancestral state “underneath” them and removing them. However, this is not recommended in most cases. The next section will discuss back-mutations and other unusual states in further detail. For further discussion of back-mutations in nucleotide-based models, see also section 18.15.

## 18.4 Back-mutations, independent mutational lineages, and VCF output

In the previous section, we saw various ways to output nucleotides and nucleotide sequences from a model. The output generated ended up revealing an important fact about modeling nucleotides in SLiM: that back-mutations are represented with a new mutation object, not by reversion of the genome sequence to an “empty” state at the back-mutated site. In this section we will explore VCF output, another way to output nucleotide-based information from a model, and we will use it to further examine the details of how nucleotide-based simulations in SLiM work under the hood. (See section 18.15 for further discussion of back-mutations in nucleotide-based models.)

The recipe we will use here is very straightforward:

```
initialize() {
    defineConstant("L", 10);
    initializeSLiMOptions(nucleotideBased=T);
    initializeAncestralNucleotides(randomNucleotides(L));
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(2.5e-5));
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-8);
}
1 late() {
    sim.addSubpop("p1", 500);
}
5000 late() {
    g = p1.sampleIndividuals(5).genomes;
    g.outputVCF(simplifyNucleotides=F);
    g.outputVCF(simplifyNucleotides=T);
}
```

Again we are using a very short chromosome and a high mutation rate for simplicity. After 5000 ticks of neutral evolution, we take a sample of individuals from p1 and output it in VCF format in two different ways.

The first `outputVCF()` call uses `simplifyNucleotides=F`, which is the default. The VCF output it produces looks like this (call lines only):

```
1 2 . G T,C,G 1000 PASS MID=2904,3371,3778;S=0,0,0;DOM=0.5,0.5,0.5;PO=1,1,1;
T0=3814,4445,4963;MT=1,1,1;AC=5,4,1;DP=1000;AA=G GT 1|2 2|2 1|1 3|2 1|1
1 3 . A G,G 1000 PASS MID=2759,2854;S=0,0;DOM=0.5,0.5;PO=1,1;T0=3612,3756;
MT=1,1;AC=5,5;DP=1000;AA=A GT 1|2 2|2 1|1 2|2 1|1
1 8 . T A,C 1000 PASS MID=2584,3761;S=0,0;DOM=0.5,0.5;PO=1,1;T0=3346,4944;
MT=1,1;AC=2,3;DP=1000;AA=T GT 2|0 0|0 1|2 0|0 2|1
1 9 . A T 1000 PASS MID=3244;S=0;DOM=0.5;PO=1;T0=4282;MT=1;AC=5;DP=1000;AA=A
GT 0|1 1|1 0|0 1|1 0|0
1 10 . T C 1000 PASS MID=1303;S=0;DOM=0.5;PO=1;T0=1731;MT=1;AC=5;DP=1000;AA=T
GT 0|1 1|1 0|0 1|1 0|0
```

This is fairly standard VCF output, which we're not going to discuss at length here; chapter 26 discusses output formats in detail, and of course a reference on the VCF format itself would be useful for interpreting many aspects of this output. We have skipped the header lines to focus on the call lines, which call the segregating alleles present in the five individuals sampled. Five positions have segregating variation; the positions are given in the second column of the output, so they are 2, 3, 8, 9, and 10.

Positions 9 and 10 are straightforward: position 9 represents the mutation of an ancestral A to a derived T (present in five of the ten sampled genomes), and position 10 represents the mutation of an ancestral T to a derived C (also present in five of the ten sampled genomes, so these two mutations perhaps comprise a haplotype). The ancestral nucleotide is given in the fourth column, and the derived allele(s) in the fifth. The rest of the lines give various state related to the mutations – mutation IDs, selection coefficients, and so forth – and the calls for the sampled genomes (after the GT marker). The details are not important for our purposes here.

The call line for position 8 is a bit more complex. The ancestral state is T, but the derived state is A,C rather than a single nucleotide. This means there are two different segregating mutations at position 8, one to A and one to C. Many of the fields in the remainder of the call line are doubled, because there are two mutation IDs, two selection coefficients, etc. – one for each of the mutations segregating at this position. The calls for the sampled genomes now use values of 0 (for the ancestral state), 1, and 2.

The state at position 3 is similar but a bit odd; the ancestral nucleotide is A, with derived state of G,G. This is somewhat unconventional in VCF output; what is the difference between G and G? The difference is that these are separate mutational lineages, which are tracked separately in SLiM as always (see section 1.5). It feels a bit more strange when there are nucleotides involved, because we usually think that G is G is G; but each mutational lineage in SLiM is distinct. If the DFE for these mutations is non-neutral, they could have different selection coefficients; or they could even belong to different mutation types. This is often useful and desirable, and is unlikely to cause any important departure from biological reality for typical mutation rates (remember, the mutation rate in this model is extremely high precisely because we want to cause such phenomena to occur). However, see section 18.14 for a technique to merge mutational lineages by “uniquing” mutations.

Finally, the call line for position 2 shows an ancestral state of G, with derived state of T,C,G. This means that there are three segregating mutations at this position, with nucleotides of T, C, and G. As in section 18.3, the G mutation segregating on an ancestral background of G represents a back-mutation; a mutation arose earlier in the simulation at this position, to some non-G state, and then a back-mutation occurred to the G state, represented by a new mutation rather than by reverting to the “empty” ancestral state.



That explains the output, then; but the fact remains that this style of VCF output is unconventional, and some VCF processing tools may object to it; derived states are not usually the same as ancestral states, and different derived states at the same position do not usually use the same nucleotide. To allow easier processing by such VCF tools, SLiM allows for more standard VCF output to be generated with the flag `simplifyNucleotides=T`. The second output call in this recipe uses that flag, and the call lines it produces – for the same sample as before – look like:

```
1 2 . G C,T 1000 PASS AC=4,5;DP=1000;AA=G GT 2|1 1|1 2|2 0|1 2|2
1 3 . A G 1000 PASS AC=10;DP=1000;AA=A GT 1|1 1|1 1|1 1|1 1|1
1 8 . T A,C 1000 PASS AC=2,3;DP=1000;AA=T GT 2|0 0|0 1|2 0|0 2|1
1 9 . A T 1000 PASS AC=5;DP=1000;AA=A GT 0|1 1|1 0|0 1|1 0|0
1 10 . T C 1000 PASS AC=5;DP=1000;AA=T GT 0|1 1|1 0|0 1|1 0|0
```

Here the back-mutation is represented as simply belonging to the ancestral state, and the independent G mutational lineages at position 3 are represented as a single derived G allele. This means that in some cases one allele in the VCF file represents more than one SLiM mutation, so all of the SLiM-specific information – mutation IDs, selection coefficients, etc. – is not available in this format. If that information is needed, the `simplifyNucleotides=F` format must be used.

Again, chapter 26 has further information about output formats in SLiM. The goal of this recipe was not so much to completely explain VCF output as to introduce it, and to use it to illustrate the ways in which nucleotide-based SLiM simulations might differ from one's expectations, specifically with respect to back-mutations and independent mutational lineages.

## 18.5 Modeling elevated CpG mutation rates and equilibrium nucleotide frequencies

Thus far, the nucleotide-based models we have looked at have used pre-made mutation matrices from the functions `mmJukesCantor()` and `mmKimura()`. It is possible, of course, to make your own mutation matrix to represent more sophisticated mutational models. Also, the mutation matrices we have used have been 4×4 matrices, expressing the rate of mutation from each of the four possible initial nucleotides to each of the four possible final nucleotides; but as section 18.1 mentioned, one can instead supply a 64×4 mutation matrix that expresses the rate of mutation for the central nucleotide of every possible trinucleotide (of which there are 64) to each of the four possible final nucleotides (see section 24.1). In this section, then, we will construct a 64×4 mutation matrix and look at the effect it has on the equilibrium nucleotide frequencies in a model.

Let's start out with a simpler model:

```
initialize() {
    defineConstant("L", 1e5);
    defineConstant("mu", 7.5e-6);

    initializeSLiMOptions(nucleotideBased=T);
    initializeAncestralNucleotides(randomNucleotides(L));
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);

    mm = matrix(c(0, mu, mu, mu, rep(0, 12)), ncol=4);
    initializeGenomicElementType("g1", m1, 1.0, mutationMatrix=mm);
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 10);
}
```

```

1:10000000 early() {
  if (sim.cycle % 10000 == 1) {
    cat(sim.cycle + ": ");
    print(nucleotideFrequencies(sim.chromosome.ancestralNucleotides()));
  }
}

```

This model constructs a 4×4 mutation matrix, and then runs for 10000000 ticks, printing the frequencies of the four nucleotides in the ancestral sequence every 10000 ticks. We can look at what the mutation matrix looks like, in the Eidos console:

```

> g1.mutationMatrix
      [,0] [,1] [,2] [,3]
[0,]    0    0    0    0
[1,] 7.5e-06 0    0    0
[2,] 7.5e-06 0    0    0
[3,] 7.5e-06 0    0    0

```

Remember that each row gives rates for each initial nucleotide (A, C, G, T), and that each column gives the rates for mutation of that initial nucleotide to each final nucleotide (A, C, G, T). This matrix, then, says that A does not mutate at all in this model (the whole row is zero), whereas C, G, and T do mutate – but only to A. As a result, the whole genome will slowly mutate to an all-A sequence. We can see this in the frequencies that are output by the `nucleotideFrequencies()` function, which tabulates the frequencies of nucleotides in a given sequence:

```

1: 0.25129 0.24927 0.25029 0.24915
10001: 0.30401 0.23187 0.23214 0.23198
20001: 0.35508 0.21558 0.21463 0.21471
30001: 0.40258 0.19961 0.19893 0.19888
40001: 0.44556 0.18466 0.18517 0.18461
50001: 0.48547 0.17126 0.17214 0.17113
...
300001: 0.91991 0.02725 0.026 0.02684
310001: 0.92594 0.02496 0.02414 0.02496
320001: 0.9311 0.02323 0.02247 0.0232
...
1360001: 0.99998 0 0 2e-05
1370001: 0.99998 0 0 2e-05
1380001: 0.99999 0 0 1e-05
1390001: 1 0 0 0
1400001: 1 0 0 0
...

```

Interestingly, the genome changes to A relatively rapidly at first – it gets from 25% to almost 50% in only 50,000 ticks – but that pace slows dramatically toward the end, making for an asymptotic approach to the endpoint, which is finally reached at 1,390,000 ticks or so. This is counterintuitive at first, since any given non-A site is mutating to A at the same rate at the end as it was at the beginning – shouldn't the rate of progress be the same throughout? The explanation is that the mutations to A need not only to arise, but to fix; and since there are many more of them segregating simultaneously at the beginning than at the end, the genome-wide rate of *fixation* is much higher earlier, even though the per-non-A-site rate of *mutation* is the same. An intuitive explanation might be that drift makes random genotypes win, and at the beginning any random genotype is likely to have some mutations to A in it, but at the end almost all of the genotypes have no segregating mutations in them.

So far so good; we have developed our intuitions a little regarding how the equilibrium nucleotide frequency is approached. Now let's modify the model by replacing our one-line mutation matrix setup, which looked like this:

```
mm = matrix(c(0, mu, mu, mu, rep(0, 12)), ncol=4);
```

with something more interesting:

```
mm = mm16To256(mmJukesCantor(mu / 3));
xcg = c("ACG", "CCG", "GCG", "TCG");
xcg_codons = nucleotidesToCodons(paste0(xcg));
mm[xcg_codons,3] = mm[xcg_codons,3] * 20; // rates to T
cgx = c("CGA", "CGC", "CGG", "CGT");
cgx_codons = nucleotidesToCodons(paste0(cgx));
mm[cgx_codons,0] = mm[cgx_codons,0] * 20; // rates to A
```

Let's unpack what's going on here. First of all, mmJukesCantor(mu/3) gives us this:

```
> mmJukesCantor(mu/3)
      [,0] [,1] [,2] [,3]
[0,]      0 2.5e-06 2.5e-06 2.5e-06
[1,] 2.5e-06      0 2.5e-06 2.5e-06
[2,] 2.5e-06 2.5e-06      0 2.5e-06
[3,] 2.5e-06 2.5e-06 2.5e-06      0
```

The mm16To256() function expands that 4×4 matrix into a 64×4 matrix:

```
> mm16To256(mmJukesCantor(mu/3))
      [,0] [,1] [,2] [,3]
[0,]      0 2.5e-06 2.5e-06 2.5e-06
[1,]      0 2.5e-06 2.5e-06 2.5e-06
[2,]      0 2.5e-06 2.5e-06 2.5e-06
[3,]      0 2.5e-06 2.5e-06 2.5e-06
[4,] 2.5e-06      0 2.5e-06 2.5e-06
[5,] 2.5e-06      0 2.5e-06 2.5e-06
[6,] 2.5e-06      0 2.5e-06 2.5e-06
[7,] 2.5e-06      0 2.5e-06 2.5e-06
...
[60,] 2.5e-06 2.5e-06 2.5e-06      0
[61,] 2.5e-06 2.5e-06 2.5e-06      0
[62,] 2.5e-06 2.5e-06 2.5e-06      0
[63,] 2.5e-06 2.5e-06 2.5e-06      0
```

Each row of this matrix provides the mutation rates for a given trinucleotide, counting upward in base 4: AAA, AAC, AAG, AAT, ACA, ACC, ACG, ACT, AGA, AGC, and on up to TTT at row 64 (see section 24.1). The result is functionally equivalent to the 4×4 matrix; that means that the last nucleotide of the trinucleotide doesn't matter (so each row pattern repeats four times, for ACGT in the last position), and the first nucleotide doesn't matter (so the initial block of 16 rows repeats four times, for ACGT in the first position). This gives us a foundation that we can modify to achieve our goal.

What is our goal? To model the high mutation rate of CpG sites. It turns out that a CG dinucleotide mutates to a TG dinucleotide at an elevated rate in many organisms. Since a CG dinucleotide on the focal strand is also a CG dinucleotide on its complementary strand (reading in the opposite direction), one can also say that the CG dinucleotide on the focal strand mutates to a CA dinucleotide at a very high rate (the complementary strand mutated to TG). So to accurately model this process from the perspective of the focal strand we will model both of those changes.

First we set up `xcg` to be a vector of the trinucleotides where a central C mutates to T at a high rate because it is followed by a G: ACG, CCG, GCG, and TCG. We paste those trinucleotides together into a single string, ACGCCGGCGTCG, and use `nucleotidesToCodons()`, a utility function, to convert the sequence into integer codons in [0,63]; these are the rows in the mutation matrix that govern the mutation rate of those four codons. Finally, we multiply the values in column 3 of those rows – the mutation rate to T – by 20.

After doing a similar thing for the trinucleotides where a central G mutates to A at a high rate because it is preceded by a C (because that means there is a CG on the complementary strand), we end up with the mutation matrix we want, which we can print in the running model:

```
> g1.mutationMatrix
      [,0] [,1] [,2] [,3]
[0,]      0 2.5e-06 2.5e-06 2.5e-06
[1,]      0 2.5e-06 2.5e-06 2.5e-06
[2,]      0 2.5e-06 2.5e-06 2.5e-06
[3,]      0 2.5e-06 2.5e-06 2.5e-06
[4,] 2.5e-06      0 2.5e-06 2.5e-06
[5,] 2.5e-06      0 2.5e-06 2.5e-06
[6,] 2.5e-06      0 2.5e-06 5e-05
[7,] 2.5e-06      0 2.5e-06 2.5e-06
[8,] 2.5e-06 2.5e-06      0 2.5e-06
[9,] 2.5e-06 2.5e-06      0 2.5e-06
[10,] 2.5e-06 2.5e-06      0 2.5e-06
[11,] 2.5e-06 2.5e-06      0 2.5e-06
[12,] 2.5e-06 2.5e-06 2.5e-06      0
[13,] 2.5e-06 2.5e-06 2.5e-06      0
[14,] 2.5e-06 2.5e-06 2.5e-06      0
[15,] 2.5e-06 2.5e-06 2.5e-06      0
[16,]      0 2.5e-06 2.5e-06 2.5e-06
[17,]      0 2.5e-06 2.5e-06 2.5e-06
[18,]      0 2.5e-06 2.5e-06 2.5e-06
[19,]      0 2.5e-06 2.5e-06 2.5e-06
[20,] 2.5e-06      0 2.5e-06 2.5e-06
[21,] 2.5e-06      0 2.5e-06 2.5e-06
[22,] 2.5e-06      0 2.5e-06 5e-05
[23,] 2.5e-06      0 2.5e-06 2.5e-06
[24,] 5e-05 2.5e-06      0 2.5e-06
[25,] 5e-05 2.5e-06      0 2.5e-06
[26,] 5e-05 2.5e-06      0 2.5e-06
[27,] 5e-05 2.5e-06      0 2.5e-06
[28,] 2.5e-06 2.5e-06 2.5e-06      0
[29,] 2.5e-06 2.5e-06 2.5e-06      0
[30,] 2.5e-06 2.5e-06 2.5e-06      0
[31,] 2.5e-06 2.5e-06 2.5e-06      0
[32,]      0 2.5e-06 2.5e-06 2.5e-06
[33,]      0 2.5e-06 2.5e-06 2.5e-06
[34,]      0 2.5e-06 2.5e-06 2.5e-06
[35,]      0 2.5e-06 2.5e-06 2.5e-06
[36,] 2.5e-06      0 2.5e-06 2.5e-06
[37,] 2.5e-06      0 2.5e-06 2.5e-06
[38,] 2.5e-06      0 2.5e-06 5e-05
[39,] 2.5e-06      0 2.5e-06 2.5e-06
[40,] 2.5e-06 2.5e-06      0 2.5e-06
[41,] 2.5e-06 2.5e-06      0 2.5e-06
[42,] 2.5e-06 2.5e-06      0 2.5e-06
[43,] 2.5e-06 2.5e-06      0 2.5e-06
[44,] 2.5e-06 2.5e-06 2.5e-06      0
[45,] 2.5e-06 2.5e-06 2.5e-06      0
[46,] 2.5e-06 2.5e-06 2.5e-06      0
[47,] 2.5e-06 2.5e-06 2.5e-06      0
[48,]      0 2.5e-06 2.5e-06 2.5e-06
[49,]      0 2.5e-06 2.5e-06 2.5e-06
[50,]      0 2.5e-06 2.5e-06 2.5e-06
[51,]      0 2.5e-06 2.5e-06 2.5e-06
[52,] 2.5e-06      0 2.5e-06 2.5e-06
[53,] 2.5e-06      0 2.5e-06 2.5e-06
[54,] 2.5e-06      0 2.5e-06 5e-05
[55,] 2.5e-06      0 2.5e-06 2.5e-06
[56,] 2.5e-06 2.5e-06      0 2.5e-06
[57,] 2.5e-06 2.5e-06      0 2.5e-06
[58,] 2.5e-06 2.5e-06      0 2.5e-06
[59,] 2.5e-06 2.5e-06      0 2.5e-06
[60,] 2.5e-06 2.5e-06 2.5e-06      0
[61,] 2.5e-06 2.5e-06 2.5e-06      0
[62,] 2.5e-06 2.5e-06 2.5e-06      0
[63,] 2.5e-06 2.5e-06 2.5e-06      0
```

You might need a magnifying glass to read that, but eight entries have a twenty-fold increase in their rates: [6,3], [22,3], [38,3], and [54,3] for the CG-to-TG mutations, and [24,0], [25,0], [26,0], and [27,0] for the CG-to-CA mutations (which are CG-to-TG mutations on the complementary strand). This model seems to equilibrate at least as quickly as the previous one:

```
1: 0.24866 0.25147 0.25051 0.24936
10001: 0.26807 0.23266 0.23179 0.26748
20001: 0.27538 0.22543 0.22402 0.27517
30001: 0.27899 0.22239 0.21949 0.27913
40001: 0.28241 0.21986 0.21724 0.28049
...
300001: 0.2935 0.20856 0.20556 0.29238
310001: 0.29249 0.20837 0.20564 0.2935
320001: 0.29305 0.2086 0.20462 0.29373
330001: 0.29351 0.2074 0.20494 0.29415
340001: 0.29409 0.20731 0.20522 0.29338
...
1220001: 0.29337 0.20553 0.20956 0.29154
1230001: 0.29271 0.20636 0.20966 0.29127
1240001: 0.29301 0.20627 0.20914 0.29158
1250001: 0.29237 0.20623 0.20929 0.29211
1260001: 0.29272 0.2058 0.20874 0.29274
...
```

It's a bit harder to tell since the equilibrium here is dynamic, and the frequencies will fluctuate stochastically forever; but certainly by generation 300,000 it is quite close to equilibrium, which seems to be somewhere near 30% A and T, 20% C and G. If we run this model out to ten million generations the frequencies appear unchanged, but we can do better: we can average the printed frequencies for the whole ten-million-generation run over its last nine million generations. This should be quite close to the expected equilibrium value from theory. For one run the result is ~29.3% A and T, ~20.7% C and G. Deriving the theoretical expectation is beyond the scope of this manual.

In short, then, we have seen that it is straightforward to build a 64×4 mutation matrix that gives custom mutation rates for particular trinucleotide sequences, and that the equilibrium nucleotide frequencies resulting from such a mutation matrix can easily be assessed with the `nucleotideFrequencies()` function (a `nucleotideCounts()` function is also provided, if absolute counts are needed). The recipes here used an artificially high overall mutation rate and a very small population size in order to obtain quicker results, but those choices should not influence the final outcome, only the runtime to reach it.

## 18.6 A nucleotide-based model with introduced non-nucleotide-based mutations

Many nucleotide-based models will want to use only nucleotide-based mutations; if a mutation occurs, it ought to have an associated nucleotide, biologically, one might reason. That makes some sense, but sometimes it will nevertheless be desirable to use non-nucleotide-based mutations as well, in a “mixed model”. Non-nucleotide-based mutations can represent modifications to the genome that are otherwise unrepresentable in SLiM, such as inversions, indels, and epigenetic modifications (see, for example, section 14.4). It can also be convenient to model a mutation as non-nucleotide-based if you simply don't care what nucleotide is associated with it; that is what we will model in this recipe.

The recipe here is quite simple, establishing a nucleotide-based mutation type (`m1`) for neutral mutations and a non-nucleotide-based mutation type (`m2`) for a sweep mutation that is introduced in tick 1:

```

initialize() {
  defineConstant("L", 1e5);
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(L));
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  initializeMutationType("m2", 0.5, "f", 0.1);
  m2.convertToSubstitution = F;
  m2.color = "red";

  initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-7));
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);
}
1 late() {
  sim.addSubpop("p1", 500);
  sample(p1.genomes, 10).addNewDrawnMutation(m2, 20000);
}
2000 late() {
  print(sim.mutationsOfTypes(m2));
}

```

If the m2 sweep mutation is not lost, it fixes, but is not converted to a substitution since `convertToSubstitution` has been set to F for m2. Sometimes it will be lost due to drift, but after a few runs one will generally see a run in which it fixes and is printed.

The interesting thing about this model is that it sets up a non-nucleotide-based mutation type in a nucleotide-based model. Note that the non-nucleotide-based mutation is added with `addNewDrawnMutation()`; this is the only way for non-nucleotide-based mutations to occur in a mixed model, in fact (or `addNewMutation()`, equivalently, of course). It is not legal for a non-nucleotide-based mutation type to be used as one of the mutation types contributing to a given genomic element type; we could not provide something like `c(m1,m2)` and `c(1.0,0.01)` to `initializeGenomicElementType()`, as we might otherwise do.

The reason for this restriction is that in nucleotide-based models the mutation rate at a given position is sequence-dependent; as we saw in the previous section, the mutation rate at a CpG site might be 20× higher than at a non-CpG site. What, then, would it mean to say that the relative fractions of mutations of types m1 and m2 are 1.0 to 0.01 if m1 is nucleotide-based (and thus its mutation rate is sequence-dependent) and m2 is not nucleotide-based (and thus its mutation rate is not sequence-dependent)? The requested ratio between the mutation types could not be guaranteed unless the mutation rate for m2 were also sequence-dependent; but that seems to make little sense for a non-nucleotide-based mutation type. And what mutation rate would m2, the non-nucleotide-based mutation type, even use given that in nucleotide-based models the mutation rates are specified by the mutation matrix, in which the rates are sequence-dependent?

Of course it would be possible to decide upon some policy regarding these questions; but for the time being, at least, the policy we decided upon was to simply make such models illegal. Non-nucleotide-based mutations can be used in nucleotide-based models, but only by introducing them in script – they are never auto-generated by SLiM.

If a nucleotide-based model wants non-nucleotide-based mutations to arise in a regular, probabilistic fashion, similar to how they would in a typical non-nucleotide-based SLiM model, that is quite easy to script: for each new offspring genome, just use `rbinom()` or `rpois()` to draw the number of mutations that occurred (based on the rate and the genome length), draw random positions for them (using `rdunif()` across the range of the genome's length), and then call `addNewDrawnMutation()` to add all of the drawn mutations to the genome. Unless a non-uniform mutation rate map is needed, this should be just a couple of lines of code.



## 18.7 Using standard SLiM fitness effects with nucleotides: modeling synonymous sites

Thus far, the nucleotide-based mutations we've modeled have been neutral, but the usual SLiM fitness effects, due to selection coefficients, dominance coefficients, and `mutationEffect()` callbacks, can be applied to nucleotide-based mutations too. In some cases one might want fitness effects to be sequence-dependent in some specific way (see sections 18.8 and 18.9); but often it may be enough, as in other SLiM models, for the effects of nucleotide-based mutations to be drawn from some distribution of fitness effects (DFE). That is what we will look at here.

A nucleotide-based SLiM model can use whatever genomic structure and DFEs are wanted; the recipes of previous chapters could generally be converted to be nucleotide-based easily. However, when one starts to think of a model in terms of explicit nucleotides there are some ideas about fitness effects that spring to mind, such as the phenomenon of “synonymous substitutions”. Synonymous substitutions are mutations that have no effect upon the amino acid coded for by a given trinucleotide, because of the degeneracy of the genetic code – the fact that many amino acids are coded for by more than one trinucleotide sequence. For example, CAT and CAC both code for histidine, so a mutation from T to C in that trinucleotide will have no effect upon the amino acid generated (although it might still have some effect on fitness due to factors such as translational efficiency). The pattern of the genetic code's degeneracy is such that mutations in the third position are much more likely to be synonymous than mutations in the first or second positions. One could model that explicitly and precisely with a sequence-based strategy (see section 18.9), but to a first approximation one might simply model it using different DFEs for the first and second position versus the third position of each trinucleotide. The setup for such a strategy might look like this:

```
initialize() {
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(3e5));

  mm = mmJukesCantor(2.5e-8);
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0); // neutral
  initializeMutationTypeNuc("m2", 0.1, "g", -0.03, 0.2); // deleterious
  initializeGenomicElementType("g1", c(m1,m2), c(3,3), mm); // pos 1/2
  initializeGenomicElementType("g2", c(m1,m2), c(5,1), mm); // pos 3
  initializeRecombinationRate(1e-8);

  types = rep(c(g1,g2), 1e5);
  starts = repEach(seqLen(1e5) * 3, 2) + rep(c(0,2), 1e5);
  ends = starts + rep(c(1,0), 1e5);
  initializeGenomicElement(types, starts, ends);
}
1 early() {
  sim.addSubpop("p1", 500);
}
```

This model defines a neutral mutation type (m1) and a deleterious mutation type (m2); we won't model beneficial mutations at all, for simplicity. Approximately two-thirds of mutations in the third position are synonymous in the standard genetic code. If we assume that synonymous mutations are always neutral, then about two-thirds of mutations that would be deleterious in the first or second position should be neutral when in the third position (ignoring the few cases where mutations in the first or second positions are synonymous). We set up the mix of m1 and m2 mutations accordingly for g1 (used for first and second positions) and g2 (used for third positions).

Then we need to set up the chromosome structure. In this model, we just assume that the whole chromosome is a single big exon, so we alternate two g1 positions followed by one g2

position all down the length of the chromosome. A more complex model could, of course, model non-coding regions and introns and so forth; the scheme should work fine as long as the exonic portions of the genome are correctly tiled with g1 and g2 (perhaps using other genomic element types for other kinds of genomic regions). Note that we make a single vectorized call to `initializeGenomicElement()` here; this is more efficient since we are creating a large number of genomic elements, especially since we provide the elements in ascending order by position.

All that remains is to implement an output event that allows us to see how well this scheme is working:

```
1e6 late() {
  sub = sim.substitutions;
  pos3 = (sub.position % 3 == 2);
  pos12 = !pos3;

  catn(size(sub) + " substitutions occurred.");
  catn(mean(sub.mutationType == m1)*100 + "% are neutral.");
  catn(mean(sub.mutationType == m2)*100 + "% are non-neutral.");
  catn();
  catn(size(sub[pos12]) + " substitutions are at position 1 or 2.");
  catn(mean(sub[pos12].mutationType == m1)*100 + "% are neutral.");
  catn(mean(sub[pos12].mutationType == m2)*100 + "% are non-neutral.");
  catn();
  catn(size(sub[pos3]) + " substitutions are at position 3.");
  catn(mean(sub[pos3].mutationType == m1)*100 + "% are neutral.");
  catn(mean(sub[pos3].mutationType == m2)*100 + "% are non-neutral.");
  catn();
}
```

This gets the substitutions from the model, creates logical vectors `pos12` and `pos3` to allow the substitutions at those positions to be selected easily with subsetting, and then prints out some very basic summary statistics about the substitutions that occurred:

```
17990 substitutions occurred.
76.3591% are neutral.
23.6409% are non-neutral.

11019 substitutions are at position 1 or 2.
67.1114% are neutral.
32.8886% are non-neutral.

6971 substitutions are at position 3.
90.9769% are neutral.
9.0231% are non-neutral.
```

First of all, we can see that neutral mutations are more likely to fix; the DFE for the deleterious mutations defines a fairly weak average effect (using a gamma distribution with a mean of  $-0.03$ ), but selection is certainly having an effect here. We can see this because about ~61% of new mutations should be neutral according to the genetic model (from a little quick math), whereas ~76% of the mutations that actually fixed are neutral. This makes sense; selection is working; good.

Second, we can see that the different DFEs for positions 1 and 2 versus position 3 are clearly making a difference; ~91% of substitutions at position 3 are neutral, versus ~67% at positions 1 and 2. This is because the proportion of mutations that are neutral is much higher at the third position to begin with, as we defined it to be.

## 18.8 Defining sequence-based fitness effects at the nucleotide level

In the previous section, we saw how to define fitness effects for nucleotide-based mutations using SLiM's standard fitness machinery of selection coefficients drawn from some distribution of fitness effects. For many models that is useful, but sometimes it is desirable to define a specific nucleotide-based fitness effect: if the nucleotide at this position is such-and-such, the fitness effect will be such-and-such. One would not generally want to try to define such specific nucleotide-based fitness effects for a large number of positions, but at a particular locus of interest, where perhaps the protein coded for by a gene is well-understood and the effects of mutations have been characterized experimentally, it could certainly be desirable.

In this recipe, we will show how to model such an effect. We will model a chromosome that is generally under neutral evolution, but that has one specific nucleotide that has selective effects we wish to model explicitly. The recipe:

```
initialize() {
  defineConstant("L", 1e4);
  defineConstant("EFF", c(1.0, 0.1, 1.5, 3.0));
  initializeSLiMOptions(nucleotideBased=T);

  seq = randomNucleotides(100) + 'A' + randomNucleotides(1e4 - 101);
  initializeAncestralNucleotides(seq);

  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(2.5e-7));
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);
}
1 early() {
  sim.addSubpop("p1", 500);
}
s1 fitnessEffect() {
  nuc1 = individual.genome1.nucleotides(100, 100, format="integer");
  nuc2 = individual.genome2.nucleotides(100, 100, format="integer");
  return EFF[nuc1] * EFF[nuc2];
}
10000 late() {
  subs = sim.substitutions[sim.substitutions.position == 100];

  for (sub in subs)
    catn("Sub to " + sub.nucleotide+ " in " + sub.fixationTick);
}
```

In the `initialize()` callback we define a constant, `EFF`, with the fitness effect we want to apply to an individual carrying a given nucleotide. We will model the effects of the nucleotides in a given individual's two genomes as being independent and multiplicative, but this is not hard to change. The four values in `EFF` correspond to the four nucleotides A, C, G, and T; the values chosen mean that at the selected position an A will be neutral (1.0), a C will be nearly lethal (0.1), a G will be beneficial (1.5), and a T will be strongly beneficial (3.0). We can therefore expect that the final state of the model will have a T at the selected position, and that we might pass through an intervening state of G en route to that final state; mutations to C are very unlikely to persist.

We then construct an ancestral sequence that contains random nucleotides everywhere but position 100, which begins as an A. Apart from those changes, the initialization is predictable. Note that there is only one mutation type, with a neutral DFE, and that the genome is uniform, composed of a single genomic element; we are not treating the selected position specially in any way in the genetic model apart from setting its initial nucleotide state to A.

After creating the subpopulation, we have a `fitnessEffect()` callback named `s1`. Remember that `fitnessEffect()` callbacks are called once per individual per cycle, giving an opportunity to provide customized per-individual fitness effects (see section 25.3). In this callback, we get the nucleotide that is present at position `100` in each genome of the focal individual. We get these nucleotides with format "integer", so they will be integers in `[0,3]`, representing A/C/G/T respectively – perfect for subsetting our `EFF` vector to get a fitness effect. And so that's what we do, returning the product of the two `EFF` values since we want the fitness effect in this model to be multiplicative. Notice that this `fitnessEffect()` callback looks only at the nucleotide sequences of the genomes, not directly at the mutations present; it works exactly the same whether mutations at position `100` are present or not.

The final output event just gets a vector of any substitutions that occurred at the selected position and prints out their nucleotide and fixation tick. One run of the model produces:

```
Sub to G in 668
Sub to T in 5522
```

The model did, in this run, mutate to G first and then end up at T. In other runs, T is reached first, or no substitution occurs at the selected position at all before the run ends. But we never see back-mutation to A, or fixation of C, since those events are selected against so strongly.

The performance of this model is noticeably slower than for a vanilla neutral model without the nucleotide-based fitness effect, because the `fitnessEffect()` callback runs in every cycle and checks the nucleotide at position `100` of both genomes of every individual. Because this is a WF model, where only relative fitness matters, we can eliminate this overhead quite easily with a little trick:

```
late() {
  if (sum(sim.mutations.position == 100) == 0)
    s1.active = 0;
}
```

This `late()` event checks whether any of the mutations currently segregating in the model are at position `100`. If not, the effect of the `fitnessEffect()` callback would be uniform, altering the fitness of every individual in exactly the same way, and in a WF model that makes no difference. In that case, it therefore deactivates the callback, for that cycle, by setting its `active` property to `0`. (Re-activating the callback is not necessary, since SLiM automatically re-activates every script block at the start of every tick.) With this addition, the model runs just about as fast as the equivalent neutral model; the `fitnessEffect()` callback is used in only a few cycles.

In a nonWF model the same optimization could be used except that when `s1` is deactivated, `p1.fitnessScaling` should also be set to the appropriate value to produce the correct fitness effect for the whole population, calculated from `EFF` given the nucleotide currently at the selected position in the ancestral sequence.

Obviously the mechanics of this model would be a bit clunky if you wanted to implement specific nucleotide-based effects at, say,  $n=100$  different locations. In that event you would probably want to generalize the code here to be driven by a vector of  $n$  positions and a matrix of  $n \times 4$  nucleotide effects. That wouldn't actually be that painful to implement, but it might become prohibitively slow since so much of the model's logic would be happening in script. If the positions that you want to give custom effects for are all at one locus, it would at least be faster to fetch the nucleotides from `genome1` and `genome2` just a single time, using a start and end position for the fetch that spans the whole region of interest, and then subset into the resulting vector of integers to get the nucleotide state at a specific position; that should be faster than fetching each nucleotide individually.

## 18.9 Defining sequence-based fitness effects at the amino acid level

In the previous recipe, we saw how to make fitness dependent upon the nucleotide sequence of an individual's genomes. That method could be extended to model realistic effects of nucleotide mutations on the amino acid sequence of a gene, but it would be rather cumbersome to implement in practice. However, SLiM provides some support for operating directly at the level of amino acids in a model, smoothing the way for models of this type. In this section we will explore a model of amino-acid-based fitness. In particular, we will model neutral dynamics *except* that any mutation that generates a stop codon within the coding sequence of a gene is lethal. Let's start with the `initialize()` callback and the subpopulation creation:

```
initialize() {
    defineConstant("L", 1e4);
    defineConstant("TAA", nucleotidesToCodons("TAA"));
    defineConstant("TAG", nucleotidesToCodons("TAG"));
    defineConstant("TGA", nucleotidesToCodons("TGA"));
    defineConstant("STOP", c(TAA, TAG, TGA));
    defineConstant("NONSTOP", (0:63)[match(0:63, STOP) < 0]);

    codons = sample(NONSTOP, 194, replace=T);
    seq1 = randomNucleotides(253);
    seq2 = paste0(codonsToNucleotides(codons, format="char")[0:417]);
    seq3 = randomNucleotides(200);
    seq4 = paste0(codonsToNucleotides(codons, format="char")[418:581]);
    seq5 = randomNucleotides(L-1035);
    seq = seq1 + seq2 + seq3 + seq4 + seq5;
    catn("Initial AA sequence: " + codonsToAminoAcids(codons));

    initializeSLiMOptions(nucleotideBased=T);
    initializeAncestralNucleotides(seq);
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(2.5e-6));
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-8);
}
1 early() {
    sim.addSubpop("p1", 500);
}
```

This model defines a chromosome of length 1e4, with a gene that is comprised of two exons, one spanning [253,670] in the nucleotide sequence, the other spanning [871,1034]. The first exon is 418 bases long, the second 164 bases long; neither is a multiple of three in length, but together, when transcribed and spliced, the mRNA for the gene will be 582 bases long, which is a multiple of three comprising 194 codons.

The first thing this model does is define a set of constants that will be useful for working with codons. Codons are represented in SLiM as integers in [0,63], where AAA is 0, AAC is 1, AAG is 2, AAT is 3, and thus upward, counting in base 4, to TTT which is 63. These integer values can be a little confusing to work with, so we define constants TAA, TAG, and TGA to represent the codon values of the three stop codons in the standard genetic code (which turn out to be represented by 48, 50, and 56). The integer values for the codons are provided by a utility function, `nucleotidesToCodons()`. Once we have those codon values, we define a vector named STOP that represents all three stop codons, and then derive a vector named NONSTOP that contains all the codon values that are not stops. If you are curious, you can see the amino acids represented by these vectors in the Eidos console using the `codonsToAminoAcids()` utility function, which converts integer codon values to the standard amino acid letters:

```
> codonsToAminoAcids(STOP)
"XXX"
> codonsToAminoAcids(NONSTOP)
"KNKNTTTTRSRSIIMI IQHQPPPPRRRRLLLLLEDEDAAGGGGVVVVYYSSSSCWCLFLF"
```

The stop codons are represented by X; the rest have their usual letters as well.

So far, so good. Having gotten those ducks in a row, we then construct an ancestral sequence. So as to not start the model in a lethal state, we have to create an ancestral sequence that, within the two exons, contains no stop codons. We do this by drawing 194 random codons from `NONSTOP` to represent the initial codon sequence, and translating it back into nucleotides with `codonsToNucleotides()`, another utility function. Once we have done that, we assemble the ancestral sequence as appropriate, with completely random nucleotides outside of the two exons. Finally, we print the amino acid sequence for the gene, using `codonsToAminoAcids()`, for future reference.

The rest of the initialization is unsurprising; we just set up a uniform chromosome using nucleotide-based mutations with a high-mutation-rate Jukes–Cantor model.

Next let's define a `fitnessEffect()` callback that makes stop codons lethal:

```
fitnessEffect() {
  for (g in individual.genomes)
  {
    seq = g.nucleotides(253, 670) + g.nucleotides(871, 1034);
    codons = nucleotidesToCodons(seq);
    if (sum(match(codons, STOP) >= 0))
      return 0.0;
  }
  return 1.0;
}
```

We loop over the two genomes of the focal individual (we deem a stop codon in either one of the two to be lethal, in this model, but it would be easy to model dominance such that both genomes must contain a stop codon to produce lethality). For each genome, we get the nucleotides from the two exons, paste them together into a single sequence string, and convert the sequence to codons with `nucleotidesToCodons()`. Finally, we use the `match()` function to find any matches between the codon sequence and any of the stop codons in `STOP`. If any are found, the callback returns `0.0`; otherwise it returns a neutral fitness effect of `1.0`.

Finally, we have an output event:

```
100000 late() {
  catn(sim.substitutions.size() + " fixed mutations.");

  as1 = sim.chromosome.ancestralNucleotides(253, 670, "integer");
  as2 = sim.chromosome.ancestralNucleotides(871, 1034, "integer");
  as = c(as1, as2);
  codons = nucleotidesToCodons(as);
  catn("Final AA sequence: " + codonsToAminoAcids(codons));
}
```

This prints the number of mutations and the final amino acid sequence (to be compared to the initial sequence printed in the `initialize()` callback). The relevant output from this model looks like this, for one sample run:



Initial AA sequence: VRGSKRAYTVAWATTYDREEADLTRSPGGLPRANYFEDHFLTSPQLYSSRRSS  
 RWNLSVVSrvVEYRFCLGANGLIQYIPFSLLPVTPPREDKGLTANLQGLRVKRVLAPQPHLKTGVLTDIRFTLY  
 LATNMITSLQSGLATINVQHRGLLGRLGLSPNQNVYLWDRQCIRLRIAAPTGAKHITIRHVDENHL

7211 fixed mutations.

Final AA sequence: ASSSTQSFTLENSRTPIRHGIDKPISQGCMTCATVYEVSRRRRQQRSTRHASRL  
 TPGGVVDHALGNPSCAGHSGVTRYILFILAPLMVSRDENGSSIVVGLRQAPVHALQQRHKSVVYPLIQIPYSLA  
 TNIRTQLQTV CSTLETRETHGIGRACLPPMSPRYDWAPRFSLLL SVGVWRRPQKNTNLHMTSHRA

It can be seen that many, if not most, of the codons in the gene sequence have mutated (thanks to the high mutation rate used); nevertheless, even though three of the possible 64 codon values are stops, the final sequence contains no stop codon. The model has avoided stop codons because they are lethal. Indeed, if you run this model under SLiMgui you will see the occasional individual colored black instead of yellow; those are individuals that received a mutation that gave them a stop codon within the gene, and so they have a fitness of zero. The operation of the lethality mechanism can thus be seen graphically, as well as in the final sequence.

The fitness scheme here is simple: three codons are lethal, all others are neutral. There is no reason that this model could not be extended to more complex fitness functions, however. In fact, if one had a command-line program that evaluated a given amino acid sequence for its efficacy in performing some enzymatic task – perhaps looking at the folded structure of the protein and so forth – one could, in principle, use the `system()` function of Eidos to delegate fitness evaluation to such an external program, evolving a gene sequence according to an algorithmic evaluation of the fitness of the resulting protein by an external tool. This would probably not be *fast*, but it should be possible! (Those feeling sufficiently adventurous could define a new built-in SLiM function, in C++, for their evaluator instead; working in SLiM's C++ code is not generally recommended, but this might be a case where it would be worthwhile.)

## 18.10 Varying the mutation rate along the chromosome in a nucleotide-based model

The nucleotide-based models we have seen so far have involved a mutation rate that can be sequence-dependent (as in section 18.5) but that is otherwise uniform along the chromosome, without “hot spots” or “cold spots”. In non-nucleotide-based models, varying the mutation rate along the chromosome is done with an explicit map of mutation rates set up with `initializeMutationRate()` (as in, e.g., section 14.10; see also sections 6.1.1 and 6.1.2 for the similar issue of setting up an explicit recombination rate map). In nucleotide-based models, it is instead done with a map of mutation rate *multipliers* set up with `initializeHotspotMap()`. These multipliers scale the local sequence-dependent mutation rate up or down within particular genomic regions. The recipe here sets up a random hotspot map, in a similar manner to the random recombination rate map of section 6.1.1:

```
initialize() {
  defineConstant("L", 1e5);
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(L));
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  m1.color = "black";
  initializeGenomicElementType("g1", m1, 1.0, mmKimura(1.8e-07, 6e-08));
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-8);

  ends = c(sort(sample(0:(L-2), 99)), L-1);
  multipliers = rlnorm(100, 0.0, 0.75);
  initializeHotspotMap(multipliers, ends);
}
```


```

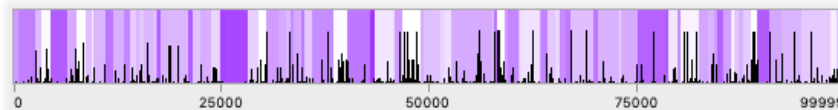
1 early() {
    sim.addSubpop("p1", 500);
}
2000 late() { sim.outputFixedMutations(); }

```

The initialization here is generally familiar. The `m1` mutation type is set to draw in black just for better visual contrast in the screenshot below. A Kimura (1980) mutational model is used, as in section 18.2, to illustrate that setting up a hotspot map is compatible with sequence-based variation in the mutation rate; the realized mutation rate at a given position will be the multiplicative product of the sequence-based rate at that position (as expressed by the `mutationMatrix` of the governing genomic element type) and the hotspot multiplier at that position (as expressed by the hotspot map).

The interesting bit is at the end of the `initialize()` callback, where we generate a hotspot map. A total of 100 different hotspot map regions are generated, using endpoints drawn from the set of possible positions along the chromosome (guaranteeing that `L-1` is the final endpoint, so that the map is complete). Multipliers are drawn from a lognormal distribution using `rlnorm()`, not with any biological basis for that choice, but simply because it is a convenient choice of distribution to provide values in  $(0, \infty)$  with a known mean and standard deviation.

A snapshot of the end result (with display of the mutation rate map enabled in SLiMgui with the  button to the right of the chromosome view):



Darker regions represent mutational cold spots, with multipliers as low as 0.2 in this run of the model, and lighter regions represent hot spots, with multipliers as high as 7.0. The effect of the hotspot map on the mutational dynamics can be seen clearly; the darkest regions show few mutations, while the lightest regions show many. At any given site, the relative rates of transitions versus transversions from the Kimura (1980) model is maintained, although that cannot be seen here.

A hotspot map could easily be read in from a file instead of being generated randomly; see section 6.1.2 for an example (reading in a recombination rate map, there, but the technique would be the same).

### 18.11 Modeling GC-biased gene conversion (gBGC)

SLiM provides both a simple “crossover breakpoint” recombination model and a more mechanistically detailed “double-stranded break (DSB)” model. The “DSB” model provides support for gene conversion, including the phenomenon of biased gene conversion. These foundational concepts are discussed in detail in section 1.5.6, and section 6.1.3 showed a recipe for a simple model including unbiased gene conversion.

In SLiM 3.1, a recipe for biased gene conversion was introduced (section 14.19, at that time) using the script-based simulation of nucleotides that was available at that time. That recipe is now obsolete, and has been retired; in SLiM 3.3, since we now have built-in support for both nucleotide-based models and biased gene conversion, a much better and simpler model can be constructed. In this section, then, we will look at a model for a nucleotide-based model that includes simulation of GC-biased gene conversion, and will see the effect that that has upon the equilibrium nucleotide frequency – the average GC content of the genome, specifically.

The recipe is quite straightforward:

```

initialize() {
  defineConstant("L", 1e5);
  defineConstant("alpha", 2.5e-6);
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(L));
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(alpha));
  initializeGenomicElement(g1, 0, L-1);
  initializeRecombinationRate(1e-5);
  initializeGeneConversion(0.7, 1500, 0.80, 0.10);
}
1 early() {
  sim.addSubpop("p1", 500);
}
1:1000001 early() {
  if (sim.cycle % 1000 == 1) {
    cat(sim.cycle + ": ");
    print(nucleotideFrequencies(sim.chromosome.ancestralNucleotides()));
  }
}

```

A chromosome of length 1e5 is set up, and elevated mutation and recombination rates are used so the model makes more rapid progress for demonstration purposes. A simple Jukes–Cantor mutational model is used, so no nucleotide bias is introduced by the mutational model itself; with no other source of bias, the nucleotide frequencies in the model should be equal (with some stochasticity). The nucleotide frequencies in the ancestral sequence (which incorporates all fixed mutations) are printed every thousand cycles until the model finishes (see section 18.5, which similarly prints nucleotide frequencies, for discussion).

The interesting bit is the call to `initializeGeneConversion()`. This function takes four parameters (although the last parameter is optional). The first, passed an argument of 0.7 here, is the fraction of recombination events (i.e., double-stranded breaks or “DSBs”) that are non-crossovers; so 70% of DSBs will be non-crossovers here (involving only a gene conversion tract) and 30% will be crossovers (involving a gene conversion tract and a switch in the copy strand). The second, passed 1500 here, is the mean gene conversion tract length, in bases. The third, passed 0.80 here, is the fraction of gene conversion tracts that are “simple”, not involving any heteroduplex mismatch repair; 80% of gene conversion tracts in this model will be “simple”. Finally, the fourth parameter, passed 0.10 here, is the GC bias coefficient used to determine whether, and to what extent, G/C alleles are preferred to A/T alleles during heteroduplex mismatch repair. A bias of 0.0 (the default) expresses no preference, so the mismatch will be repaired toward one or the other parental strand randomly, whereas 1.0 would represent an absolute preference for the G/C allele, and -1.0 would represent an absolute preference for the A/T allele. In this model, the bias of 0.10 means that the G/C allele will be preferred 55% of the time, a small but important effect as we will see below. Section 1.5.6 describes the mechanistic consequences of these parameters in detail, so we will not discuss them further here.

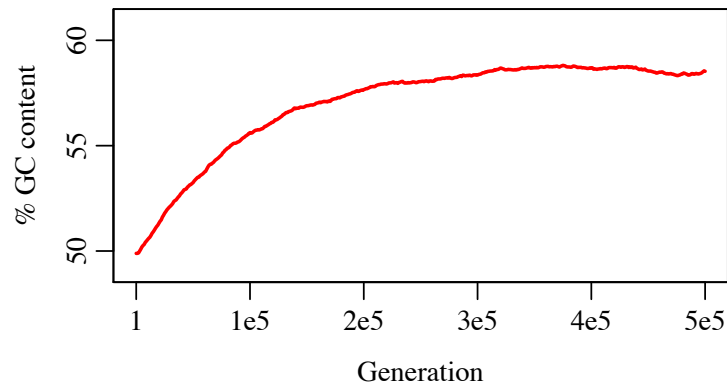
This model takes quite a while to run because it runs for so many ticks, but when it does complete, the output generated looks something like this:

```

1: 0.24885 0.25057 0.2483 0.25228
1001: 0.24885 0.25055 0.24835 0.25225
2001: 0.2488 0.25072 0.24834 0.25214
...
498001: 0.20598 0.29034 0.29469 0.20899
499001: 0.20598 0.29027 0.29511 0.20864
500001: 0.20587 0.29046 0.29485 0.20882

```

This output shows the GC content of the ancestral sequence rising steadily from ~50% initially to a more or less equilibrium level of ~59% by the end of the run. A quick plot of this in R shows the trajectory of the rise:



An equilibrium GC content is approached, rather than the GC content just rising until it reaches 100%, for at least two reasons. First, as GC content increases the tendency of random mutation to change G/C positions back to A/T gets stronger. If you imagine a sequence that was 100% G/C, two-thirds of mutations would change a G/C position to an A/T, one-third would change a G to a C or *vice versa*, and none would change an A/T to a G/C. Second, as GC content increases the number of heteroduplex mismatch sites involving a G/C mismatched with an A/T decreases, and so the frequency with which biased gene conversion promotes G/C alleles decreases. Again, if you imagine a 100% GC sequence, mismatch repair would never have an opportunity to prefer G/C over A/T; the effect of the gene conversion bias would completely disappear.

The equilibrium GC content, and the speed at which that equilibrium is reached, will depend upon the gene conversion parameters, the recombination rate, and other model parameters. In particular, as Duret & Galtier (2009) note, the strength of the evolutionary response in GC content is expected to be proportional to the effective population size. The population size of 500 used here leads to a smaller evolutionary response (but a relatively fast model runtime, for purposes of illustration). Because of this dependency upon the effective population size, model rescaling (see section 5.5) should be used with caution when biased gene conversion is enabled, as either the speed of equilibration or the equilibrium GC content level may be distorted.

The parameter values used here are not intended to be a realistic depiction of biased gene conversion in any particular species; however, they are loosely based upon empirical parameter estimates. It would be simple to extend this model to be more biologically realistic; indeed, isochore structure should emerge naturally if a variable recombination rate map along the chromosome is used, with high GC content near recombination hotspots (where gene conversion is more active) and lower GC content in the rest of the genome.

## 18.12 Reading VCF files to create nucleotide-based SNPs

One of the benefits of nucleotide-based models is the ability to simulate real populations down to the level of the actual segregating SNPs. Such data is often represented with a VCF file that contains the SNPs for the sequenced genomes, in addition to a FASTA file that contains the “reference” sequence from which those SNPs differ. Given such files, SLiM can easily set up a simulation of the population, as we will explore in this section’s recipe using human genomic data from the 1000 Genomes Project. This recipe will be rather more empirically based than usual, so we will first deal with obtaining and massaging the genomic data used.

A prerequisite for this work is installing two open-source packages: samtools and VCFtools. The procedure for installing these will be platform-specific. I installed samtools on macOS using MacPorts (port install samtools), and used version 1.10. VCFtools was not on MacPorts, as of this writing, so I downloaded the source and built it in Terminal following the instructions at <https://vcftools.github.io/examples.html>, obtaining version 0.1.17.

Next, I went to <http://www.internationalgenome.org/data> and downloaded two files. One was the sample information, a file named 20130606\_sample\_info.xlsx. The other was the Phase 3 VCF file, named ALL.chr22.phase3\_shapeit2\_mvncall\_integrated\_v5a.20130502.genotypes.vcf, for chromosome 22. I decided to focus on the samples from people in Finland, and I filtered out indels and mutations that were, in the Finnish samples, either fixed or absent, with the command:

```
vcftools --vcf
ALL.chr22.phase3_shapeit2_mvncall_integrated_v5a.20130502.genotypes.vcf --
keep FIN_samples.txt --remove-indels
--mac 1 --recode --out chr22_filtered
```

This reduced the VCF file size from 11.2 GB to 67.9 MB, making it manageable for a quick demo like this; 99 out of 2054 individuals were kept (all those specified as Finnish by the sample information), and 155518 out of 1103547 segregating sites were retained. (SLiM would be able to handle the full dataset; these reductions were made just to make the example quicker to run.)

Next, I downloaded the reference nucleotide sequence; this was specified in the ##reference header field of the VCF file as being located at [ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/phase2\\_reference\\_assembly\\_sequence/hs37d5.fa.gz](ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/phase2_reference_assembly_sequence/hs37d5.fa.gz). Once it was unzipped, I ran `grep ">" /Users/bhaller/Desktop/1000Genomes.phase3/reference/hs37d5.fa` to see what headers it contained, and saw that chromosome 22 was listed with the header line `>22 dna:chromosome chromosome:GRCh37:22:1:51304566:1`. I therefore extracted chromosome 22's reference sequence into its own file by first running `samtools faidx hs37d5.fa` (to create an index file) and then `samtools faidx hs37d5.fa "22" > hs37d5_chr22.fa` (to actually extract chromosome 22 to a new file). The original FASTA file was 3.19 GB; the extracted sequence for chromosome 22 was 52.2 MB.

The extracted reference nucleotide sequence contained a large number of sites, particularly at the beginning and end of the chromosome, designated as N in the FASTA data, meaning that the nucleotide at that position is unknown. SLiM requires that the ancestral sequence be composed only of A/C/G/T nucleotides; N is not a legal option. For this reason, I changed all of the N positions to A; those sites were not involved in SNPs anyway, so this was harmless for our purposes here.

After these steps, I had a FASTA file with the ancestral sequence (hs37d5\_chr22\_patched.fa), and a VCF file with the SNPs (chr22\_filtered.recode.vcf). Then it was time to load these files into SLiM. First of all, the initialize() callback should read in the reference sequence and make it the ancestral sequence used by the simulation, with initializeAncestralNucleotides():

```
initialize() {
    initializeSLiMOptions(nucleotideBased=T);
    length = initializeAncestralNucleotides("hs37d5_chr22_patched.fa");
    defineConstant("L", length);
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
    initializeMutationTypeNuc("m2", 0.5, "f", 0.0);
    m2.color = "red";
    m2.convertToSubstitution = F;
    initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(0.0));
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-8);
}
```

This reads the ancestral sequence from the FASTA file at the specified path, as we saw in section 18.2. We set up a chromosome of the right length (as determined by the FASTA sequence), using neutral mutations of type `m1`. We also define an `m2` mutation type, which we will use below, that displays in red and has `convertToSubstitution=F` (see section 24.11.1). We intend to do a simulation from standing genetic variation here, so the mutation rate is set to zero with `mmJukesCantor(0.0)`.

Next, in a `1 late()` event we set up the initial population state by reading SNPs from a VCF file:

```
1 late() {
  sim.addSubpop("p1", 99);
  p1.genomes.readFromVCF("chr22_filtered.recode.vcf", m1);
  p1.setSubpopulationSize(1000);
}
```

This calls the `readFromVCF()` method of `Genome` on `p1.genomes`, which creates the SNPs from the VCF file as mutations in the target genomes. The number of genomes in the target vector ( $99 \times 2 = 198$ ) must match the number of genomes specified by the VCF file, and the length of the simulated chromosome must encompass all of the SNPs. The created mutations will use the nucleotides specified by the VCF file. The mutation type passed in, `m1`, is used for the created mutations since there are no overriding MT fields in the `INFO` sections of the call lines; similarly, selection coefficients are drawn from the mutation type (here neutral), since there are no overriding `S` fields in the `INFO` sections providing selection coefficients. Finally, we scale up to 1000 individuals so that drift will have less of an effect on the simulation.

The result is a population that replicates the SNPs sampled in the empirical Finnish population. We now want to model a specific scenario: one of those SNPs becomes strongly beneficial, and may sweep, and we want to predict what other SNPs would be carried to fixation along with it by hitchhiking. In tick 5, we randomly choose a SNP of interest and change its selection coefficient to be beneficial:

```
5 late() {
  mut = sample(sim.mutations, 1);
  mut.setMutationType(m2);
  mut.setSelectionCoeff(0.5);
}
```

We also change it to use mutation type `m2` (making it red in SLiMgui, and more importantly preventing it from being substituted on fixation). Then at the end of the run we dump the SNPs that fixed:

```
1:2000 late() {
  mut = sim.mutationsOfType(m2);
  if (mut.size()) {
    f = sim.mutationFrequencies(p1, mut);
    catn(sim.cycle + ": " + sim.mutations.size() + ", f = " + f);

    if (f == 1.0) {
      catn("\nFIXED in cycle " + sim.cycle);
      catn(sim.substitutions.size() + " substitutions.");
      catn(paste(sim.substitutions.nucleotide));
      sim.simulationFinished();
    }
  } else {
    catn(sim.cycle + ": " + sim.mutations.size());
  }
}
```

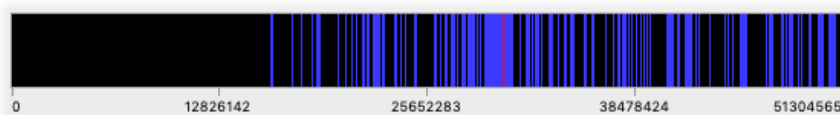


Of course in some runs of this model the beneficial SNP will be lost due to drift; but for a run in which the SNP fixes, the model's output looks like this:

```
1: 155787
2: 155787
3: 155772
4: 155655
5: 155446, f = 0.0375
6: 155221, f = 0.0455
7: 154303, f = 0.0515
...
54: 116697, f = 0.999
55: 116248, f = 0.9995
56: 115748, f = 1

FIXED in cycle 56
946 substitutions.
C T G G T C T G C G A T A C C C G A A A T ...
```

The SNP started in tick 5 at a frequency of 0.0375 – just a handful of occurrences in the original VCF dataset. There was probably some variation among the carrier genomes at other sites, so this was likely a “soft” selective sweep. Nevertheless, 946 mutations did get carried along to fixation during the sweep, out of the original 155518 segregating sites in the Finnish population. Here is a screenshot of the fixed mutations (in blue) with the beneficial mutation (the red line):



It looks like the frequency of fixation increases with proximity to the beneficial mutation, but it's hard to be sure without running some stats. This pattern, if real, is probably driven largely by the pattern of linkage disequilibrium that existed in the initial population with respect to the mutation chosen to be beneficial; but recombination would also have broken down the association between the beneficial allele and more distant mutations during the course of the sweep. (The left third of the chromosome is all N in the original FASTA file, and contains no SNPs.)

We haven't really leveraged the nucleotides themselves in this model; indeed, we could do much the same thing as this recipe without modeling nucleotides explicitly. But the nucleotides are there, and could be used for whatever purpose we wish – sequence-dependent mutations, codon-based fitness, etc., as we have seen in previous recipes. Or, if it were of interest, we could simply emit the final nucleotide sequences of all of the individuals in the simulation.

This recipe has a somewhat magical component to it, which is the call to `readFromVCF()`. Section 26.2.3 and 26.2.4 contain details on the VCF format extensions used by SLiM that allow specification of things like the mutation type and selection coefficient for each mutation; you can supply these fields in the VCF files you read in, to configure the mutational state created by SLiM. The details of what exactly `readFromVCF()` does are fairly complex; see section 24.4.2 for further discussion of its mechanics. However, the recipe shown here should provide a good starting point for simulations of SNPs in empirical populations.

One final note: many downstream tools for analysis of SNPs are apparently not prepared to handle sites that are triallelic (with not just one, but two, segregating variants). You may wish to filter such sites out of your input VCF file, or your output VCF file, or both. Zuxi Cui has a GitHub repo covering solutions for this and many other issues related to whole-genome SLiM simulations, at [https://github.com/zxc307/GWAS\\_simulation\\_handbook](https://github.com/zxc307/GWAS_simulation_handbook).

### 18.13 Tree-sequence recording and nucleotide-based models

We haven't used tree-sequence recording with a nucleotide-based model yet, but the two features are entirely compatible. When tree-sequence recording is enabled the nucleotides associated with each mutation are kept as metadata in the tree-sequence tables, and are written out to the `.trees` file along with the ancestral nucleotide sequence. That information is all accessible with `pyslim`, as we will see in this section's recipe.

Let's start by running a very simple nucleotide-based SLiM model:

```
initialize() {
    defineConstant("L", 1e5);
    initializeSLiMOptions(nucleotideBased=T);
    initializeTreeSeq();
    initializeAncestralNucleotides(randomNucleotides(L));
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-6));
    initializeGenomicElement(g1, 0, L-1);
    initializeRecombinationRate(1e-6);
}
1 early() {
    sim.addSubpop("p1", 1000);
}
1000 late() {
    sim.treeSeqOutput("recipe_nucleotides.trees");
}
```

This just churns out a bunch of mutation and recombination, and then dumps a `.trees` file at the end for analysis. We will look at two short Python scripts that analyze the nucleotides recorded in this tree sequence.

In nucleotide models, the mutation rate can depend on the nucleotide context: for instance, transitions may be more common than transversions. From genetic data only we cannot tell for sure whether an A/T polymorphism was due to an A-to-T or a T-to-A mutation. But, since tree sequences record mutations in their genealogical context, we can tabulate the realized mutation spectrum from a simulation – e.g., how many A-to-G mutations occurred during the simulation, how many A-to-T mutations, and so forth. Actually, we won't calculate the number of each type of mutation that occurred during the entire simulation, but rather the number of mutations that have occurred on the genealogical trees of the final population – and so, are segregating in the final population, or would be if other mutations had not reversed their effects.

Doing this is simple: we iterate over all mutations, identify the nucleotide state before and after the mutation, and add a count to the corresponding entry in the output table (below called `M`). Suppose a particular mutation changed an A to a T. Since mutations record the “derived state”, i.e., the allelic state of anyone who inherits that mutation, the mutation object itself knows about the derived state of T. Since mutations can “stack” in SLiM, a single mutation as seen by `pyslim` may correspond to a *collection* of SLiM mutations. The way this is recorded in the tree sequence is that a mutation's metadata contains a *list* of metadata dictionaries; we want the dictionary for the most recent mutation contained in the derived state. The time that each mutation occurred is associated with the `slim_time` key of the metadata. Therefore, the nucleotide resulting from mutation `mut` is `mut_list[k]["nucleotide"]`, where `k` is the value with the largest `mut_list[k]["slim_time"]`. To find the A, i.e., the nucleotide state *before* that mutation occurred, we use the mutation's “parent mutation” property: the value of `mut.parent` is the index of the mutation on whose background `mut` occurred. If the nucleotide state when the mutation `mut` occurred was not determined by a prior mutation (and so it was still in the ancestral state), then `mut.parent` is `-1`, telling us that we must look for the prior nucleotide state in the reference sequence. A final detail

is that the reference sequence is stored as a string of ACGT, while the nucleotide property of a mutation is a number 0123.

Here's the Python code to do that analysis:

```
import tskit, pyslim
import numpy as np

ts = tskit.load("recipe_nucleotides.trees")

M = [[0 for _ in pyslim.NUCLEOTIDES] for _ in pyslim.NUCLEOTIDES]
for mut in ts.mutations():
    mut_list = mut.metadata["mutation_list"]
    k = np.argmax([u["slim_time"] for u in mut_list])
    derived_nuc = mut_list[k]["nucleotide"]
    if mut.parent == -1:
        acgt = ts.reference_sequence.data[int(ts.site(mut.site).position)]
        parent_nuc = pyslim.NUCLEOTIDES.index(acgt)
    else:
        parent_mut = ts.mutation(mut.parent)
        assert(parent_mut.site == mut.site)
        parent_nuc = parent_mut.metadata["mutation_list"][0]["nucleotide"]
    M[parent_nuc][derived_nuc] += 1

print("{}\t{}\t{}".format('ancestral', 'derived', 'count'))
for j, a in enumerate(pyslim.NUCLEOTIDES):
    for k, b in enumerate(pyslim.NUCLEOTIDES):
        print("{}\t{}\t{}".format(a, b, M[j][k]))
```

So, line-by-line, the code above (1) sets up M, an empty 4x4 array to put the mutation counts in; (2) loops over mutations; (3) gets the dictionary of metadata for the SLiM mutations contained by each pyslim mutational “derived state”; (4) finds the most recent SLiM mutation; (5) finds the derived nucleotide; (6) checks if the mutation occurred on the background of another; (7) if not, finds the previous nucleotide in the reference sequence and (8) converts it to 0123; (9) otherwise, (10) finds the index of the previous mutation, (11) double-checks this mutation is actually at the same site; (12) finds the nucleotide resulting from the previous mutation; and finally (13) adds one to the corresponding entry of M. Then, it prints out the table, telling you for each possible pair of nucleotides X and Y, how many mutations converted an X to a Y.

The output from this analysis will look something like this:

ancestral	derived	count
A	A	0
A	C	667
A	G	670
A	T	710
C	A	616
C	C	0
C	G	649
C	T	670
G	A	665
G	C	645
G	G	0
G	T	616
T	A	639
T	C	633
T	G	647
T	T	0

This shows that A mutated to C a total of 667 times, for example, whereas C mutated to A only 616 times; since we know there is no mutational bias (because this model uses the Jukes–Cantor mutational model), this difference must be due to chance. The counts for A-to-A, C-to-C, etc., are zero since those types of mutations do not occur in SLiM’s nucleotide-based mutational model.

In nucleotide models, more complex context-dependence is possible: for instance, the probability that a G mutates to a T might depend on whether there is a C to the left of the G (see section 18.5). For our second Python analysis, we modify the code above to tabulate context-dependent mutations according to the flanking nucleotides on the left and the right. For instance, how many mutations turned a trinucleotide AAA to ATA? And, how many turned CAC to CTC?

Here is the Python code for the second analysis:

```
import tskit, pyslim
import numpy as np

ts = tskit.load("recipe_nucleotides.trees")
slim_gen = ts.metadata["SLiM"]["tick"]

M = np.zeros((4,4,4,4), dtype='int')
for mut in ts.mutations():
    pos = ts.site(mut.site).position
    # skip mutations at the end of the sequence
    if pos > 0 and pos < ts.sequence_length - 1:
        mut_list = mut.metadata["mutation_list"]
        k = np.argmax([u["slim_time"] for u in mut_list])
        derived_nuc = mut_list[k]["nucleotide"]
        pretime = mut.time + 1.0
        left_nuc = pyslim.nucleotide_at(ts, mut.node, pos - 1, time = pretime)
        right_nuc = pyslim.nucleotide_at(ts, mut.node, pos + 1, time = pretime)
        parent_nuc = pyslim.nucleotide_at(ts, mut.node, pos, time = pretime)
        M[left_nuc, parent_nuc, right_nuc, derived_nuc] += 1

print("{}\t{}\t{}".format('ancestral', 'derived', 'count'))
for j0, a0 in enumerate(pyslim.NUCLEOTIDES):
    for j1, a1 in enumerate(pyslim.NUCLEOTIDES):
        for j2, a2 in enumerate(pyslim.NUCLEOTIDES):
            for k, b in enumerate(pyslim.NUCLEOTIDES):
                print("{}{}{}\t{}\t{}\t{}".format(a0, a1, a2, a0, b, a2,
                    M[j0, j1, j2, k]))
```

This code follows the same structure as before, except that the resulting counts are stored in a four-dimensional (4x4x4x4) array, and we have to look up what the nucleotide state was before each mutation occurred, using the `pyslim` function `nucleotide_at()`. After a little while, the result is the array `M`, for which, for instance, `M[0, 0, 0, 3]` is the number of mutations that changed an AAA to ATA, and `M[0, 1, 2, 3]` is the number of mutations that changed from ACG to ATG.

Then we print `M`, which produces output like this:

ancestral	derived	count
AAA	AAA	0
AAA	ACA	44
AAA	AGA	43
AAA	ATA	45
AAC	AAC	0
AAC	ACC	36
AAC	AGC	42
AAC	ATC	44
...		

TTG	TAG	37
TTG	TCG	29
TTG	TGG	33
TTG	TTG	0
TTT	TAT	43
TTT	TCT	50
TTT	TGT	35
TTT	TTT	0

Here we can see that, for example, AAA changed to ATA a total of 45 times. As before, the counts for nucleotides changing to themselves, such as AAA-to-AAA, AAC-to-AAC, etc., are all zero.

The original rationale for doing this analysis was to look at sequence-dependent variation in mutation rates, so let's go beyond the recipe a bit by changing `mmJukesCantor(1e-6)` to `mmKimura(1.8e-06, 6e-07)`. With these particular parameters, the Kimura 1980 mutational model provides the same overall mutation rate as the Jukes–Cantor model did, but with an elevated transition-to-transversion rate ratio (see section 18.2). When we run the first analysis again, it produces this table:

ancestral	derived	count
A	A	0
A	C	414
A	G	1236
A	T	399
C	A	371
C	C	0
C	G	386
C	T	1176
G	A	1190
G	C	380
G	G	0
G	T	357
T	A	393
T	C	1125
T	G	400
T	T	0

This shows that transitions (between A and G, and between C and T) have occurred roughly three times as often as transversions (all other mutations), as requested with the chosen parameters to the Kimura 1980 model.

We won't look at the results of the second analysis here, since we know that there is no interesting variation in mutation rates that depends upon the trinucleotide sequence. If this analysis were run on the output from a model with an elevated CpG mutation rate, however, such as that of section 18.5, it would presumably reveal the higher mutation rate prevailing at those positions.

#### 18.14 Modeling identity by state (IBS): unifying mutations with a `mutation()` callback

We have seen in previous sections (see sections 1.5.2 and 18.4, for example) that SLiM tracks each “mutational lineage” with a separate `Mutation` object, so independent mutations will be tracked separately even if they represent exactly the same genetic state. This can be particularly surprising in nucleotide-based models, where it is easy to end up with, for example, multiple mutations all representing a G at a particular position in the genome. These multiple mutations might even end up having different selection coefficients, which is probably not what you want. Even if that is not an issue (perhaps your mutation type uses a fixed DFE, providing a constant

selection coefficient), SLiM will not consider the model to be fixed for G at that position unless just *one* of the G mutations fixes across the whole population. Dealing with these issues can be troublesome.

Beginning with SLiM 3.5, there is a straightforward solution, which we will explore in this section: writing a `mutation()` callback that uniques new mutations down to pre-existing mutations to provide “identity by state” (IBS) rather than SLiM’s default behavior of “identity by descent” (IBD). Before launching into the recipe, however, it is worth noting that there are good reasons for SLiM’s default IBD behavior, so you should override it with caution, and be sure that you know what you’re doing. In particular, if you are using tree-sequence recording, you should be aware that the “same” mutation may then occur on different branches of the tree, as if it had jumped from one lineage to another by horizontal gene transfer; make sure that you understand the consequences for your analysis code.

With that caveat, let’s start with a truncated version of this section’s recipe, which is designed to illustrate the problem that we plan to solve:

```
initialize() {
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(100));
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-4 / 3));
  initializeGenomicElement(g1, 0, 99);
  initializeRecombinationRate(1e-3);
}
1 early() {
  sim.addSubpop("p1", 500);
}
1000 late() {
  for (pos in 0:99)
  {
    muts = sim.subsetMutations(position=pos);
    nucs = muts.nucleotide;
    cat(pos + " : " + paste(nucs));
    if (size(nucs) != size(unique(nucs)))
      cat("      DUPLICATES!");
    catn();
  }
}
```

This is a very simple nucleotide-based model with a chromosome just 100 base positions long, and a rather high mutation rate so that we get some duplicate mutations (i.e., mutations that are distinct mutational lineages in SLiM, but would be considered “the same mutation” from an IBS perspective). The only interesting code here is the output `late()` event in tick 1000. It loops through the 100 base positions, fetches the mutations at that position, and prints out the nucleotides for them. It then checks for any duplicates – segregating mutations at the same position with the same nucleotide, which would be considered identical from an IBS perspective – and if it finds any, it prints a “DUPLICATES!” warning.

If we run this model, the output indicates that there are indeed duplicates:

```
0 : A A A C      DUPLICATES!
1 :
2 : A
3 : T T      DUPLICATES!
4 : T
...

```



So far this is just a recap of what we covered in section 18.4. The interesting and new part is that we can shift SLiM over to the IBS perspective by “uniquing down” mutations: using a pre-existing mutation, for a given position and nucleotide, rather than making a new mutation, whenever such a pre-existing mutation is present. To do this, we can just add a short `mutation()` callback to the model:

```
mutation() {
    m = sim.subsetMutations(position=mut.position, nucleotide=mut.nucleotide);
    if (m.length())
        return m;
    return T;
}
```

We have seen `mutation()` callbacks a couple of times before, in sections 10.6, 13.5, and 14.4, and they are fully documented in section 25.9. In brief, a `mutation()` callback is called whenever SLiM is planning to create a new mutation while constructing a gamete during offspring generation. The callback provides an opportunity for the script to approve or veto the proposed mutation, or to modify it. These callbacks are therefore rather like `modifyChild()` callbacks, but for new mutations instead of new offspring individuals. In SLiM 3.5 and later, a `mutation()` callback can also choose to replace the proposed mutation with a pre-existing mutation, allowing “uniquing down” of mutations in various ways, and that’s what we do here. We first call `subsetMutations()` to get any pre-existing mutation at the same position, and with the same nucleotide, as the mutation SLiM proposes to create (passed to the callback as the pseudo-parameter `mut`). If any such mutation exists, we simply return it, which tells SLiM to add *that* mutation to the gamete it is generating, rather than the mutation it had proposed. If, on the other hand, no pre-existing mutation matches the specifications, we return `T` to tell SLiM to go ahead with creating the new `Mutation` object.

Now when we run the model we no longer see any duplicates:

```
0 : T
1 : G T
2 : A C
3 : G
4 : A
...
```

Uniquing is working as planned, and we now have an IBS nucleotide-based model. Of course the same technique can be used to provide IBS mutations in a non-nucleotide-based model, or to “unique down” or replace mutations in other ways, based on other criteria. The only criterion that must be upheld is that the replacement mutation must be at the same base position as the originally proposed mutation.

There is one subtle point that requires a digression at this point. The `mutation()` callback was declared as `mutation()`, not as `mutation(m1)`; this means that it will be called for *all* new mutations, regardless of their mutation type. If you want an IBS model, this is probably what you want. If it were declared as `mutation(m1)` instead, it would be called only for new `m1` mutations, and that would mean that (in a model with multiple mutation types, with similar callbacks defined for each type) you could, for example, still have more than one `T` mutation segregating at position 924 – one of type `m1`, one of type `m2`, and so forth. Maybe that *could* be what you want, but it seems unlikely. By declaring the callback as `mutation()`, we have guaranteed that that will not happen; there will be only one `T` mutation at 924 (whatever mutation type it might turn out to be).

However, uniquing down mutations of *all* types, as we have done here, does have an odd consequence. Suppose we have two mutation types in a model: `m1` is neutral, `m2` is deleterious. If

the first mutation to T at position 924 happens to be an m1 mutation, a T at position 924 is now “reserved” for m1 as long as that mutation is still segregating; any new mutations to T at position 924 will be unique down to the first, even if SLiM drew them as m2 mutations. This means that the requested proportion of mutations in the genetic configuration of the model may not be upheld; m2 mutations will be turned into m1 mutations by the uniquing mechanism (or, at a different position, m1 mutations might be turned into m2). Biologically, this makes sense; it would be surprising (without epigenetics) to have one T at position 924 be neutral while a different T at position 924 was deleterious! Nevertheless, it does violate other assumptions that one might have about how the model will behave, such as the proportion of new mutations of particular types. And there is an aspect of this that is *not* biological, too: if the T mutation at 924 is lost from the model, or is fixed and substituted, then a future mutation to T at 924 would be allowed to be an m2 mutation by the uniquing mechanism. We have prevented discrepancies that are simultaneous in time, then, but we have not prevented discrepancies that are separated in time; to prevent those, you would need to write some additional script that would remember, for the whole lifetime of the model, that a T at 924 is supposed to be neutral, and would enforce that in some manner. All of this gets quite complicated if you think about enough! Happily, for the vast majority of models these issues can probably just be ignored; they are unlikely to occur, and unlikely to make a significant difference for results. But they are good to be aware of, just in case they do matter for you.

It might be worth playing around a little with both versions of the recipe in SLiMgui. You will notice that the two models appear to do very different things, because fixation of mutations is far more likely to occur in the IBS version of the model. In the IBD model, mutational lineages that represent the same nucleotide at the same position are competing against each other, and fixation does not occur until just one of those lineages fixes across the whole population. In the IBS model, those mutational lineages are effectively merged, and fixation occurs whenever a given nucleotide fixes at a given position, even if that outcome is actually the result of multiple independent lineages fixing jointly. This illustrates that the conceptual shift here can have real consequences, and needs to be taken into account in the analysis you do of model results; naively computing a metric like “mean fixation events per generation” for these two models would indicate that they were experiencing very different evolutionary dynamics, when in fact the dynamics are identical – only the lens through which those dynamics are viewed differs.

This recipe still leaves in place in aspect of nucleotide-based modeling in SLiM that is sometimes undesirable, having to do with back-mutations to the ancestral state. We will explain that, and address it, in the next section.

## 18.15 Modeling identity by state (IBS): uniquing back-mutations to the ancestral state

In the previous section we saw how to “unique down” nucleotide-based mutations so that only one mutation for a given nucleotide at a given position can be segregating at the same time, effectively switching SLiM from its default “identity by descent” (IBD) model to an “identity by state” (IBS) model. That worked nicely – but as was hinted at the end, it has a potential flaw having to do with back-mutations to the ancestral state.

Suppose a particular position in SLiM has an ancestral nucleotide of A; if no mutation is present at that position, then the assumed nucleotide, courtesy of the ancestral sequence, is A. Now suppose that in some generation that A mutates to a G in a particular genome; now we have an ancestral state of A and a segregating G mutation. Later, an individual carrying the G mutation generates an offspring which experiences a mutation of the G to an A. This is a “back-mutation”, a mutation from a derived state back to the ancestral state. By default in SLiM, this will result in a segregating A mutation, which makes sense from the IBD perspective; the mutation to A represents

a new mutational lineage, and so it gets a new mutation. It is a different mutational state than that of genomes that are empty (ancestral) at the position. It also makes sense from the perspective of tree-sequence recording, where mutations normally apply across the tree sequence from the point where they occur. Nevertheless, it may not be what you want! If you want an IBS model because you want to be able to compare two genomes and know whether or not they are “the same” (meaning “the same sequence of nucleotides”), having segregating mutation objects that represent back-mutations is still inconvenient. Depending upon exactly what one means by IBS, the previous section’s recipe did not get us all the way there. We will go the remaining distance here. Thanks to Qiming Long for inspiring this recipe by asking good questions.

Let’s start with a modified version of the 18.4 recipe:

```
initialize() {
  initializeSLiMOptions(nucleotideBased=T);
  initializeAncestralNucleotides(randomNucleotides(100));
  initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
  initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-4 / 3));
  initializeGenomicElement(g1, 0, 99);
  initializeRecombinationRate(1e-3);
}
1 early() {
  sim.addSubpop("p1", 500);
}
mutation() {
  m = sim.subsetMutations(position=mut.position, nucleotide=mut.nucleotide);
  if (m.length())
    return m;
  return T;
}
1000 late() {
  for (pos in 0:99)
  {
    muts = sim.subsetMutations(position=pos);
    nucs = muts.nucleotide;
    ancestral = sim.chromosome.ancestralNucleotides(pos, pos);
    cat(pos + " : " + paste(nucs));
    if (size(nucs) != size(unique(nucs)))
      cat("      DUPLICATES!");
    if (any(nucs == ancestral))
      cat("      BACK-MUTATION (" + ancestral + ")!");
    catn();
  }
}
```

This model is the same except for the final output event. That event adds an additional check, using the `ancestralNucleotides()` method, to look for any mutation at a given position that has the same nucleotide as the ancestral state at that position. If such a condition is found, that represents the sort of back-mutation that we want to get rid of, so it logs.

If we run the model as it stands now, we will see occasional output lines like this:

```
17 : A C G      BACK-MUTATION (C)!
```

or this:

```
23 : A          BACK-MUTATION (A)!
```

Now let's fix it! We will do this with a fairly simple `late()` event (which should be added above the final output event in the script, so that it runs first):

```
late() {
  // unique new mutations down to the ancestral state
  muts = sim.mutations;
  new_muts = muts[muts.originTick == community.tick];
  back_muts = NULL;
  for (mut in new_muts)
  {
    pos = mut.position;
    if (mut.nucleotide == sim.chromosome.ancestralNucleotides(pos, pos))
      back_muts = c(back_muts, mut);
  }
  if (size(back_muts))
    sim.subpopulations.genomes.removeMutations(back_muts);
}
```

First this gets all of the mutations in the simulation, and narrows down to those that are new in this tick; only those need to be checked, which will save a lot of time in large models. Then the script loops over those new mutations and checks each against the ancestral nucleotide at its position; if a mutation is a back-mutation to the ancestral state, it adds it to the accumulator vector `back_muts` with `c()`. Finally, it removes the mutations in `back_muts` from all genomes in the simulation.

That should do the trick! If we run the final recipe now, we see no “BACK-MUTATION” logs. Mischief managed.

Now for the caveats. As in section 18.14, it must be noted that this is often *not* what you want; do not reflexively add this code to every nucleotide-based model you write! One side effect of this is that, as in section 18.14, it may distort the ratios and rates of new mutations; some fraction of new mutations will be suppressed by this code, which, depending upon how you do your analysis, might distort things.

A more worrisome side effect is that it will literally prevent any new mutation at a given site that happens to go to the ancestral state. Suppose, for example, that you have a model in which every mutation has a deleterious effect that is drawn from some DFE; there are no neutral mutations. Now suppose that position 924 is ancestral for T. All back-mutations to a state of T at 924 will then be converted into a removal of the derived mutation, reverting to a neutral state at that position rather than allowing a new, deleterious T mutation to be created. Maybe that is what you want; it does make biological sense if you have defined the ancestral sequence as “neutral” and every derived state that departs from it as “deleterious”, as in a model of purifying selection acting upon an optimal nucleotide sequence. But it might be surprising – and mess up your analysis – that when a mutation to G at 924 has almost fixed, and the ancestral state of T is at a frequency of just 0.0000013, a mutation from G to T is nevertheless beneficial (removing a deleterious mutation to revert to a neutral state), whereas a moment later, when G has fixed and is thus the new ancestral state, a mutation from G to T is now deleterious (changing the neutral state of G to a deleterious mutation of T). That is rather arbitrary!

The point here, with these caveats in both section 18.14 and here, is to say: there is no one right answer for all models. SLiM has a default behavior of IBD, which may or may not be what you want. Changing that default behavior with these recipes may or may not be what you want, too. Tread carefully, and beware unintended consequences.

Finally, note that this IBS recipe and the previous one are orthogonal. You could unique down mutations without uniquing down to the ancestral state; or you could unique down to the ancestral

state without uniquing down mutations otherwise; or you could do both, as we have done here. Once again, there is no one right answer, and you will want to think carefully about what you really want the behavior to be, for the model you are writing, which might be dictated by the analysis you want to perform, or might be dictated by the scientific question you are trying to answer.

## 19. Multispecies modeling

So far – for the first 500-plus pages of this manual! – we have concerned ourselves with SLiM models that involve only a single species (perhaps, at most, looking at the process of speciation, or at reproductive isolation between closely related groups). However, beginning in SLiM 4, SLiM also supports models of more than one species, even species with radically different genetics and behavior – foxes and mice, say. Such models are termed *multispecies* models. Section 1.9 provided a quick conceptual overview of multispecies models in SLiM, and it is recommended that you read that section before embarking upon this chapter. Furthermore, all of the preceding content about single-species models will be assumed knowledge here; if unfamiliar topics arise, you may wish to jump back to earlier chapters for review.

### 19.1 A simple multispecies model

We begin with an extremely simple multispecies model – so simple, in fact, that it actually contains only one species:

```
species sim initialize()
{
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
ticks all 1 early()
{
    sim.addSubpop("p1", 500);
}
ticks all 2000 late()
{
    sim.outputFixedMutations();
}
```

This is technically a “multispecies” model, according to SLiM, because it uses an *explicit species declaration*. That is this line:

```
species sim initialize()
```

whereas a single-species model would have:

```
initialize()
```

The explicit species declaration simply states that the species being initialized by this particular `initialize()` callback is named `sim`. In itself, this is unexciting; but it opens the door to having *another* `initialize()` callback that initializes some *other* species with some *other* name. Through that door lies true multispecies modeling, as we shall see.

The only other thing to note here is that the `early()` and `late()` events are now declared with a *ticks specifier*, the `ticks all` specification in their declarations:

```
ticks all 1 early()
ticks all 2000 late()
```

This tells SLiM that these events should run in every tick (within the range of ticks otherwise specified – tick 1 for the `early()` event, tick 2000 for the `late()` event). It would also be possible to use a different species specifier:



```

ticks sim 1 early()
ticks sim 2000 late()

```

This would tell SLiM that these events should run only in ticks when species `sim` is active – when that species is executing its cycle. By default, every species is active in every tick, so in this simple model using `ticks sim` would make no practical difference; we will see the importance of the `ticks` specifier for events later on, when we get into the topic of scheduling species.

When explicit species declarations are used – when writing a multispecies model – it becomes mandatory to provide a `ticks` specifier for every event. This makes the scheduling policy for each event explicit.

That concludes our first multispecies model. Of course this recipe is kind of silly; there is no reason to explicitly declare a species if you only intend to model one species. For this reason, multispecies models in SLiM will virtually always declare two or more species, and that is what we will look at next.

## 19.2 A two-species model

Moving on from the trivial example of the previous section, let’s look at a simple two-species model. This uses explicit species declarations to set up two different species, one representing foxes, the other representing mice:

```

species fox initialize() {
    initializeSpecies(tickModulo=3, tickPhase=5, avatar="🦊");
}
species mouse initialize() {
    initializeSpecies(tickModulo=1, tickPhase=1, avatar="🐭");
}
ticks all 1 early() {
    fox.addSubpop("p1", 50);
    mouse.addSubpop("p2", 500);
}
ticks all 2000 late() {
    fox.outputFixedMutations();
    mouse.outputFixedMutations();
}

```

There are a couple of new things to notice here. This model declares two species, one with `species fox` and one with `species mouse`, each with its own `initialize()` callback. Those callbacks don’t set up any genetics at all; beginning in SLiM 4 it is legal to skip the genetic initialization calls, in which case you get a zero-length chromosome with mutation and recombination rates of zero. These species are called “no-genetics species”, and were briefly introduced in section 4.1. No-genetics species tend to be more useful in a multispecies context, since multispecies models are more likely to be interested in modeling ecology, and might thus neglect the genetics (at least for one species; we could initialize the genetics of one species while omitting the other, if we wanted just one of the species to evolve).

When a single-species model (such as we saw in previous chapters) implicitly defines the species of the model, a global symbol named `sim` is automatically created by SLiM to represent that species. When explicit species declarations are employed (as here), global symbols for each species are automatically created to represent each declared species. In this model, we thus have global symbols named `fox` and `mouse` that represent the two declared species, and we use those symbols just as we would use `sim`, to create new subpopulations for each species:

```

ticks all 1 early() {
  fox.addSubpop("p1", 50);
  mouse.addSubpop("p2", 500);
}

```

As in the previous section, a ticks specifier of `ticks all` is given. This event creates a new subpopulation of 50 foxes, named p1, and a new subpopulation of 500 mice, named p2, by calling the `addSubpop()` method on the global `fox` and `mouse` objects respectively. Both `fox` and `mouse` are objects of class `Species`, just as `sim` is in single-species models, so we can do all the same things with them that we could do before with `sim`.

The other thing to note is the two new initialization calls:

```

initializeSpecies(tickModulo=3, tickPhase=5, avatar="🦊");
initializeSpecies(tickModulo=1, tickPhase=1, avatar="🐭");

```

These calls, to a function named `initializeSpecies()` that we have not seen previously, configure the two species to have different *schedules*. The fox species runs every 3 ticks (`tickModulo=3`), starting in tick 5 (`tickPhase=5`), whereas the mouse species runs every tick (`tickModulo=1`) starting in tick 1 (`tickPhase=1`). This might represent foxes having an annual reproductive schedule, while mice reproduce more frequently. (Again, these biological details are fictional, however, not empirical.) We can visualize what the species schedules would look like for this example:

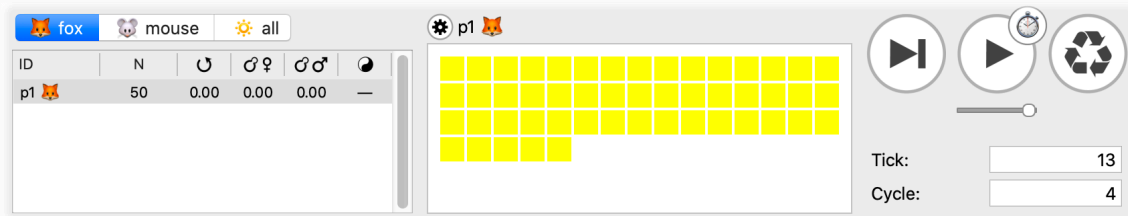
tick	species fox modulo 3 phase 5	species mouse modulo 1 phase 1
1		generation 1
2		generation 2
3		generation 3
4		generation 4
5	generation 1	generation 5
6		generation 6
7		generation 7
8	generation 2	generation 8
9		generation 9

In many ticks, only one species is active, but in the ticks outlined in red, both species are active and their cycles are interleaved – the Community runs the tick cycle simultaneously for all species, which is why it is called a “tick cycle”, not a “species cycle”. Exactly what that entails will be discussed later.

The other thing the `initializeSpecies()` calls do is set an *avatar* for each species. This is a string value – typically a single emoji character, as here with 🦊 and 🐭 – that is used to represent the species in output, particularly in SLiMgui. It is convenient that many emojis for animals and plants already exist, but of course if your study species has no assigned emoji, you ought to petition the appropriate body for relief; see <https://unicode.org/emoji/proposals.html>. It would be a better world if there were emojis for bdelloid rotifers and stick insects! In the meantime, you can use whatever emojis (or other single characters) you wish; 🧟 for an invasive species, perhaps, or

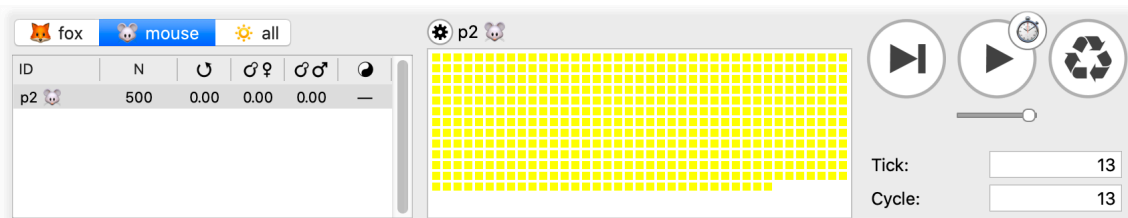
🐼 for a cold-adapted species, or whatever. Incidentally, if emojis don't show up properly on your machine, see section 2.7 for tips.

Now might be a good time to note that SLiMgui is aware of multispecies models. The above model looks like this in SLiMgui, when run forward a little:

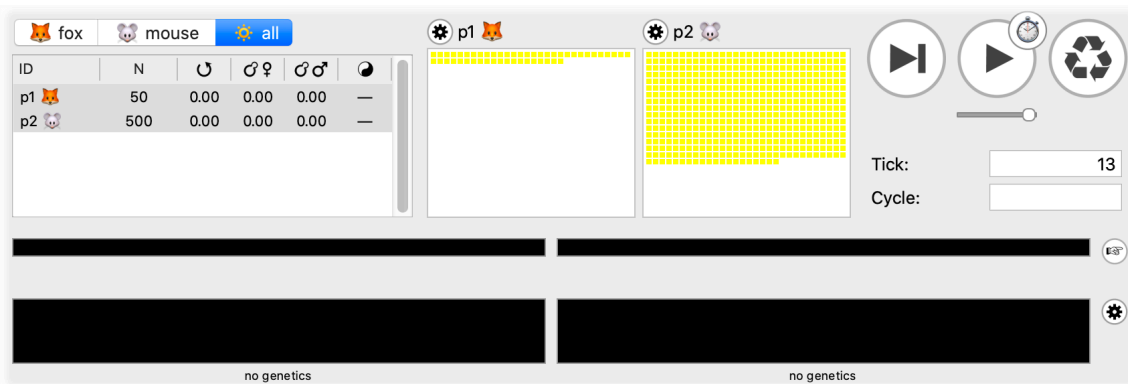


Note the new species tab bar above the population view, showing the two species that have been declared. At the moment, the fox species is selected for display; the population view shows fox subpopulations (i.e., p1), and the individuals view shows 50 foxes. Since we are about to execute tick 13, the cycle counter is 4 indicating that the next fox cycle to execute will be cycle 4 (which may or may not happen in tick 13, depending on the schedule for fox; if fox is inactive in tick 13, it will happen at some later time).

A click on the mouse tab in the species bar changes the display:



Now we see p2, with 500 mice, and the mouse cycle counter is at 13. The chromosome view's display also changes depending upon the selected species (not shown above). Finally, we can click on the all tab to see both species:



This screenshot is bigger, to show that a separate chromosome view is displayed for each species when on the all tab; they are in the same order as the tabs, so fox is on the left. Note that the tick counter still shows 13, but the cycle counter is blank; the tick is a community property, so it is always the same, but on the all tab there is no one selected species for which to show the cycle. We will explore other aspects of multispecies modeling in SLiMgui in later recipes.

So now we have defined a two-species model. However, our two species do not interact in any way – notably, the foxes don't eat the mice. We will address that in the next section.

### 19.3 A deterministic host–parasitoid model

In the previous section we defined a two-species model of foxes and mice, but the two species did not interact at all – not terribly interesting. Now we will begin to build interactions between species.

To begin with, we will do this in a top-down fashion, dictating a new population size to each species in each cycle based upon the state of the previous cycle. This follows the approach of traditional analytical models such as the Lotka–Volterra model, which does not explicitly model individual-level interactions (a particular mouse being eaten by a particular fox), but simply gives the change in population sizes as a function of the current state. This also fits well with the fact that our recipe, thus far, has used the WF model type in which population size is a top-down parameter of the model. We will see a more individual-based approach, using the nonWF model type, in the next section.

It would be nice to continue with a Lotka–Volterra model of foxes and mice, but unfortunately the Lotka–Volterra model is not stable in discrete time. There are modified versions of it in the literature that *are* stable in discrete time, but rather than get into that, we will shift gears here to model a host–parasitoid system instead. The mathematical model here was developed by Sebastian Schreiber; thanks to him for his kind assistance in developing this recipe (and the next as well).

The two equations that drive this model are:

$$x_1[t+1] = x_1[t] * \exp(r - x_1[t]/K - a*x_2[t])$$

$$x_2[t+1] = x_1[t] * (1 - \exp(-a*x_2[t]))$$

where  $x_1$  is the host density and  $x_2$  is the parasitoid density,  $t$  is the time (in discrete time, not continuous time), and  $a$ ,  $r$ , and  $K$  are model parameters as described in Faure and Schreiber (2014).

The biological motivations for these equations are that (in sequential order):

- with probability  $1 - \exp(-a*x_2[t])$  each host is killed by a parasitoid;
- with probability  $\exp(-x_1[t]/K)$  each host then escapes death via competition;
- each surviving host produces a Poisson number of offspring with mean  $\exp(r)$ ;
- all hosts killed by a parasitoid turn into a parasitoid in the next year.

This assumes that parasitism acts before density dependence. Because  $x_1$  and  $x_2$  represent density in this model, not population size, we will also use a model parameter  $S$  to represent the size of the habitat; so if  $N_1$  is the population size for the host and  $N_2$  is the population size for the parasitoid, then  $N_1 = Sx_1$  and  $N_2 = Sx_2$ . If you want to read more about this model, see Faure and Schreiber (2014), particularly the type of model discussed on their page 585. Sebastian also has a link to talk slides about this model, [here](#).

Translating this into a SLiM script is quite straightforward:

```
species all initialize() {
  defineConstant("K", 100);
  defineConstant("R", log(20));
  defineConstant("A", 0.015);
  defineConstant("S", 10^2); // larger is more stable, but slower
  defineConstant("N0_host", asInteger((135.6217 + 0.01) * S));
  defineConstant("N0_parasitoid", asInteger((109.3010 + 0.01) * S));
}
```

```

species host initialize() {
    initializeSpecies(avatar="🐛", color="cornflowerblue");
}
species parasitoid initialize() {
    initializeSpecies(avatar="🦟", color="red");
}
ticks all 1 early() {
    host.addSubpop("p1", N0_host);
    parasitoid.addSubpop("p2", N0_parasitoid);
}
ticks all late() {
    x1 = p1.individualCount / S;      // host density
    x2 = p2.individualCount / S;      // parasitoid density

    x1' = x1 * exp(R - x1/K - A*x2); // x1[t+1]
    x2' = x1 * (1 - exp(-A*x2));     // x2[t+1]

    p1.setSubpopulationSize(asInteger(round(S * x1')));
    p2.setSubpopulationSize(asInteger(round(S * x2')));
}
ticks all 250 late() {
}

```

There are a few things to notice here. First of all, we have a type of `initialize()` callback here that we haven't seen before, declared with a species specifier of `species all`. This is the appropriate place for initialization that is not species-specific, but community-wide; we are therefore initializing our model parameters in this `species all initialize()` callback. The parameter values come from Sebastian; the initial  $N_0$  for the two species is close to a stable equilibrium point for the dynamics, slightly perturbed here (adding  $0.01$ ) to kick off the cyclical behavior.

Second, notice that we're now using a caterpillar emoji for the host. The parasitoid is represented by a mosquito emoji, which looks somewhat similar to a parasitoid wasp; we'll have to live with that since the astoundingly diverse world of parasitoid wasps has been neglected by the Unicode Consortium. Also, we are now providing a `color` argument to `initializeSpecies()`; this optional parameter tells SLiMgui what color to use to represent the species in its user interface. For example, once you have the recipe open in SLiMgui, step once (so that the `initialize()` callbacks are run, setting up the color properties for the species); you will see that SLiMgui's script editor now uses the species colors to attribute code in the script pane:

```

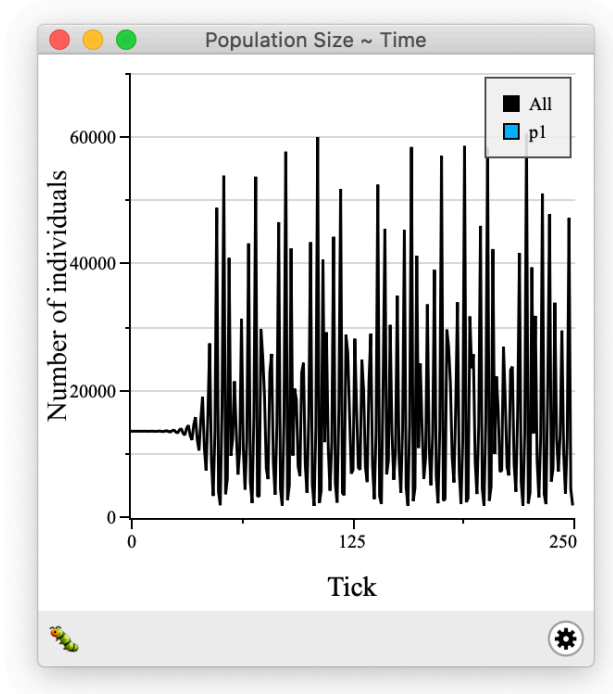
9   defineConstant("N0_parasitoid", asInteger((109.3010 + 0.01) * S));
10 }
11 species host initialize() {
12     initializeSpecies(avatar="🐛", color="cornflowerblue");
13 }
14 species parasitoid initialize() {
15     initializeSpecies(avatar="🦟", color="red");
16 }
17 ticks all 1 early() {
18     host.addSubpop("p1", N0_host);

```

Script blocks that are species-specific (with either a species specifier or a ticks specifier, which is not `species all` or `ticks all`) show the species they are associated with using the color of that species. Of course the hope is that this will make it easier to navigate in your code. SLiMgui will use species colors in other ways too, as we will see below.

Third, since Sebastian’s model operates in terms of density while SLiM models actual populations of individuals, we have to translate back and forth between the two by scaling by  $S$ . Also, the new density values for the next cycle (represented in the equations as  $x_1[t+1]$  and  $x_2[t+1]$ ) are represented in the Eidos code as  $x_1'$  and  $x_2'$ . The  $'$  character is a prime, which is a Unicode mathematical symbol commonly used to represent a new or modified version of a variable. Just as Eidos allows variables names to contain emojis, accented letters, and so forth, it also allows such Unicode mathematical symbols, which is useful in cases like this.

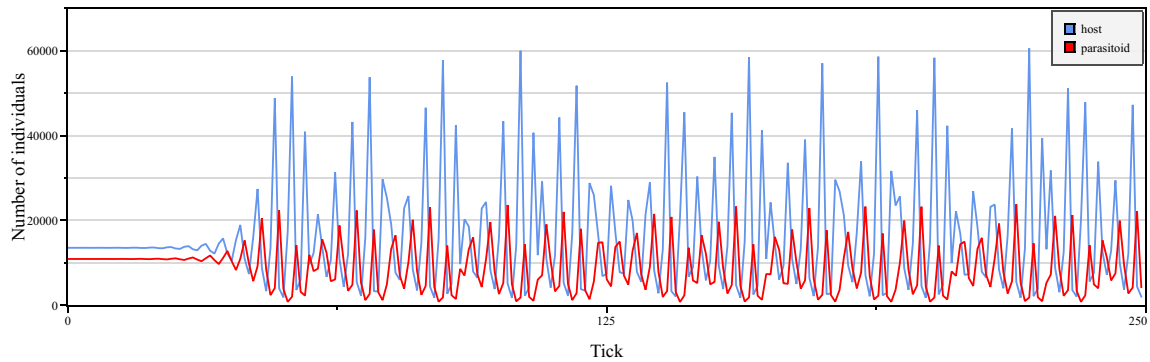
To see the cyclical population size dynamics for both species, we can open a graph. Let’s open SLiMgui’s “Graph Population Size ~ Time” graph; here’s what we get:



Not what we wanted! Most of the graphs in SLiMgui are species-specific: they focus on just one species (the currently selected species in SLiMgui’s user interface). Since this graph window was opened when the host species tab was selected in SLiMgui, it shows the population size as a function of time for the host species. You can tell that this is the case by looking at the avatar shown in the lower left of the graph window; it is the caterpillar avatar that we set for host. This graph window will remain targeted to the host species even if we switch tabs; indeed, if we switch to the parasitoid tab and then choose “Graph Population Size ~ Time” again, we will now have two plot windows, one showing population size dynamics for host, the other for parasitoid. These plot windows will remember the species they are targeted to even across a recycle; they are permanently associated with their target species.

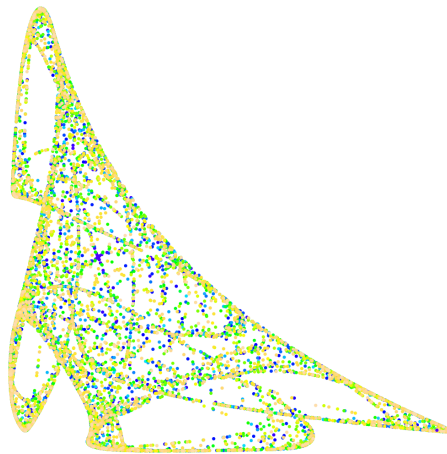
So, we could open host and parasitoid population size plot windows in this way, and try to look at the relationship between the two by lining up those two plot windows, one exactly above the other. There is a much better alternative, however. Instead of doing that, let’s open the “Multispecies Population Size ~ Time” graph, a new plot type added in SLiM 4. This plot is non-species-specific and shows the size of all subpopulations across all species. That makes it much easier to see the complex dynamics playing out in this model, since the plotted lines are superimposed:





Notice that the color we set for each species is being used in the plot; so the blue line is the host species, while the red line is the parasitoid. They are both cycling quite vigorously; it looks like the parasitoid's cycles often mirror those of the host, but it is not always that simple. Sometimes the host has a cycle that reaches a particularly large population size, and yet the parasitoid's next peak is quite low, perhaps because it is recovering slowly from a particularly low abundance. The system also takes a while to start cycling strongly, because it started so close to an equilibrium point; in its later cycling, however, it never seems to return to the vicinity of that equilibrium point (if it did, there would presumably be another period of stasis).

Sebastian has plotted these dynamics in what I hope I am correct in calling “phase space”, with  $x_1$  plotted against  $x_2$ :



This plot shows the rather interesting-looking “attractor” followed by this model. Colors represent time; the most recent points are yellow, so the paths that are mostly yellow are the most-visited paths in phase space. This plot was made over 10,000 iterations with  $S = 10^5$ , so it probably shows a more exact picture of the attractor than our model with  $S = 10^2$  would; the smaller value of  $S$  will tend to cause noisier dynamics since the effects of rounding off to the nearest integer in each cycle will be larger.

So, this is all quite nifty, but we are really just using SLiM as a glorified calculator; the individuals in this model don't matter at all. There is no individual-level behavior, and no genetics; we are just iterating the equations in a deterministic model, which we could do just as well (and much faster) in R or Python. In the next section we will move a step closer to a true coevolutionary model by introducing individual-level behavior.

## 19.4 An individual-based host–parasitoid model

The previous section introduced a host–parasite model from Sebastian Schreiber. It didn't display any individual-level behavior, however, nor did it have any genetics; it simply used SLiM as a backdrop against which the equations of the deterministic model were iterated. These dynamics can be converted to a more individual-based form, however, and that is what we will explore in this section. Again, the model shown here is discussed in more detail in Faure and Schreiber (2014); and again, thanks to Sebastian for his help in putting together this recipe.

As discussed in section 19.3, this model is based upon a few simple behaviors:

- with probability  $1 - \exp(-a \cdot x_2[t])$  each host is killed by a parasitoid;
- with probability  $\exp(-x_1[t]/K)$  each host then escapes death via competition;
- each surviving host produces a Poisson number of offspring with mean  $\exp(r)$ ;
- all hosts killed by a parasitoid turn into a parasitoid in the next year.

Recall that  $x_1$  is the host density and  $x_2$  is the parasitoid density,  $t$  is the time (in discrete time, not continuous time),  $a$ ,  $r$ , and  $K$  are model parameters, and  $S$  is the habitat size (which converts densities into population sizes). In section 19.3 those behaviors were encapsulated by non-individual-based iterative equations. In this section, we will convert them to an individual-based form.

There is one feature of this design that is unusual from the SLiM perspective: it begins with mortality, and ends with reproduction. In SLiM, reproduction is the first thing to happen in the tick cycle, and mortality happens towards the end, in both WF and nonWF models. We will discuss that wrinkle more below, but keep it in the back of your mind.

Let's start with the model's initialization:

```
species all initialize() {
  defineConstant("K", 100);
  defineConstant("R", log(20));
  defineConstant("A", 0.015);
  defineConstant("S", 10^2); // larger is more stable, but slower
  defineConstant("N0_host", asInteger((135.6217 + 0.01) * S));
  defineConstant("N0_parasitoid", asInteger((109.3010 + 0.01) * S));

  initializeSLiMModelType("nonWF");
}
species host initialize() {
  initializeSpecies(avatar="🐛", color="cornflowerblue");
  // tag values in host indicate survival (0) or death (1)
}
species parasitoid initialize() {
  initializeSpecies(avatar="🐜", color="red");
  // tag values in parasitoid count successful hunts
}
```

This is essentially the same as for section 19.3; the parameters are all the same, as are the species. I've added comments in the species-specific `initialize()` callbacks that note how tag values will be used in each species; especially with the complexity of a multispecies model it is nice to be very explicit about such things with comments. The only other change is the addition of the call to `initializeSLiMModelType("nonWF")` in the `species all initialize()` callback; since this will be a more individual-based model in which the population size cycling will be emergent, rather than top-down, we need it to be a nonWF model.

Next let's add the population setup and the simulation end event:

```
ticks all 1 early() {
  host.addSubpop("p1", N0_host);
  parasitoid.addSubpop("p2", N0_parasitoid);
}
ticks all 250 late() {
}
```

These are identical to the same events in the previous recipe; nothing worth commenting upon. Next – we are building carefully here, one step at a time, since it will get complicated at the end! – we still want non-overlapping generations even though this is a nonWF model, so we'll add `survival()` callbacks that simply kill off the parental generation of each species (see section 16.15 for further discussion):

```
// non-overlapping generations: parents die, offspring live
species host survival() {
  return (individual.age == 0);
}
species parasitoid survival() {
  return (individual.age == 0);
}
```

If an individual's age is 0 (a new juvenile), it survives; if it is older, it dies.

Next, let's look at the `reproduction()` callbacks for both species. It is worthwhile to compare and contrast them, so let's look at them together:

```
species host reproduction() {
  // only hosts tagged 0 (survived) get to reproduce
  if (individual.tag != 0)
    return;

  // reproduce each host with a mean of exp(r) offspring
  litterSize = rpois(1, exp(R));

  if (litterSize > 0)
  {
    mate = subpop.sampleIndividuals(1, exclude=individual);
    for (i in seqLen(litterSize))
      subpop.addCrossed(individual, mate);
  }
}
species parasitoid reproduction() {
  // reproduce each parasitoid as many times as it parasitized
  litterSize = individual.tag;

  if (litterSize > 0)
  {
    mate = subpop.sampleIndividuals(1, exclude=individual);
    for (i in seqLen(litterSize))
      subpop.addCrossed(individual, mate);
  }
}
```

In the initialization code, it was mentioned that tag values play different roles in these species:

```
// tag values in host indicate survival (0) or death (1)
// tag values in parasitoid count successful hunts
```

These reproduction() callbacks are where those tag values get put to work. For host, the reproduction() callback simply checks the tag value at the top; if the focal individual has a tag value of 0 it gets to reproduce, but otherwise the callback returns immediately and that host does not reproduce – because, conceptually, it is dead. The rest of the callback follows a familiar pattern: draw the number of offspring for the focal individual from a Poisson distribution, choose a mate, and loop to create the number of offspring drawn.

For parasitoid, the reproduction() callback works a bit differently. The number of offspring for the parasitoids is not drawn from a Poisson distribution; instead, it is the number of successful hunts that parasitoid had, which is stored in its tag. Each successful hunt results in one offspring, so we loop over individual.tag to produce offspring.

Of course biological details such as the mating behavior of the host could be modified. For example, the code above makes both species choose a single mate and produce all offspring with that mate; simply moving the mate choice inside the for loop would make mate choice instead be done independently for each offspring. Similarly, we could convert the model to a sexual model and make only the females of both species reproduce, choosing a male mate. Such things have been covered in previous chapters, though; here our focus is on the multispecies implementation.

There is only one piece left, and that is the event that implements hunting and competition. It is a first() event, so it executes at the beginning of each cycle, prior to reproduction, and it sets up the tag values that the reproduction() callbacks use to govern their behavior:

```
ticks all first()
{
    hosts = host.subpopulations.individuals;
    parasitoids = parasitoid.subpopulations.individuals;

    // assess densities
    x1 = hosts.size() / S;           // host density
    x2 = parasitoids.size() / S;     // parasitoid density

    // hunt: each parasitoid counts its successes, and
    // each host tracks whether it was killed
    parasitoids.tag = 0;
    P_parasitized = 1 - exp(-A * x2);
    killed = rbinom(hosts.size(), 1, P_parasitized);
    hosts.tag = killed; // 1 means killed
    hunters = sample(parasitoids, sum(killed), replace=T);
    for (hunter in hunters)
        hunter.tag = hunter.tag + 1;
    survivors = hosts[killed == 0];

    // competition: kill a fraction of survivors; note
    // that this is based on pre-parasitism density
    P_survives = exp(-x1 / K);
    survived = rbinom(survivors.size(), 1, P_survives);
    dead = survivors[survived == 0]; // 1 means survived
    dead.tag = 1;                   // mark as dead
}
```

It starts by getting the hosts and parasitoids and calculating their densities, using  $S$ . Then, in vectorized fashion, it uses rbinom() to determine which hosts get parasitized, and marks their fates in their tag values. For each host that got parasitized, it uses sample() to draw the parasitoid that did it, and it loops over those parasitoids and increments their tag values for each kill. Finally, for the hosts that survived, it implements competition as simple density-dependent selection, using rbinom() with the calculated probability of survival, and again uses tag to mark hosts that died.

Note that, following the deterministic equations, the strength of competition depends upon the pre-parasitism density of the host. I think this might make biological sense: the hosts that have had an egg laid in them by a parasitoid will eventually die, but before that happens they will live for a while, eating leaves and exerting competition upon their fellow hosts. They have been marked for death, but they are not yet dead.

The design of this model is a bit odd; we “kill” hosts in the `first()` event by setting their `tag` values, and those “deaths” affect the model because the “killed” hosts do not reproduce (because the `reproduction()` callback checks each focal individual’s `tag` value). The only *actual* death that occurs in the model, however, is due to the `survival()` callbacks, which run after reproduction has finished. We structured the model this way because of the order of events specified at the beginning – remember when I said you should keep that in the back of your mind? The design of the model mirrors that order of events, and so it begins with death and ends with reproduction. Since SLiM’s tick cycle begins with reproduction and ends with death, we needed to model the deaths of hosts due to predation and competition using `tag`. You might be thinking: we could simply shift our perspective, since the model actually cycles, death-birth/death-birth/death-birth – we could instead cycle birth-death/birth-death/birth-death. Yes! The mathematics would be the same, and it would work fine. The only problem would be that the population size plot in SLiMgui would no longer show what we want it to; to match the plot for the deterministic model, we want it to show population size after birth, not after death, and so we need to model death-birth/death-birth/death-birth (since SLiMgui plots the state of the model at the end of each cycle). Here is the behavior event for that second version of the model:

```
ticks all early()
{
    hosts = host.subpopulations.individuals;
    parasitoids = parasitoid.subpopulations.individuals;

    // first, kill off the parental generation
    host_age = hosts.age;
    hosts[host_age > 0].fitnessScaling = 0.0;

    parasitoid_age = parasitoids.age;
    parasitoids[parasitoid_age > 0].fitnessScaling = 0.0;

    // narrow down to the juveniles and assess densities
    hosts = hosts[host_age == 0];
    parasitoids = parasitoids[parasitoid_age == 0];

    x1 = hosts.size() / S;           // host density
    x2 = parasitoids.size() / S;     // parasitoid density

    // next, hunt; each parasitoid counts its successes
    parasitoids.tag = 0;
    P_parasitized = 1 - exp(-A * x2);
    luck = rbinom(hosts.size(), 1, P_parasitized);
    dead = hosts[luck == 1];
    dead.fitnessScaling = 0.0;
    hunters = sample(parasitoids, dead.size(), replace=T);
    for (hunter in hunters)
        hunter.tag = hunter.tag + 1;
    hosts = hosts[luck == 0];

    // finally, competition kills a fraction of survivors
    // this is based on pre-parasitism density
    P_survives = exp(-x1 / K);
```

```

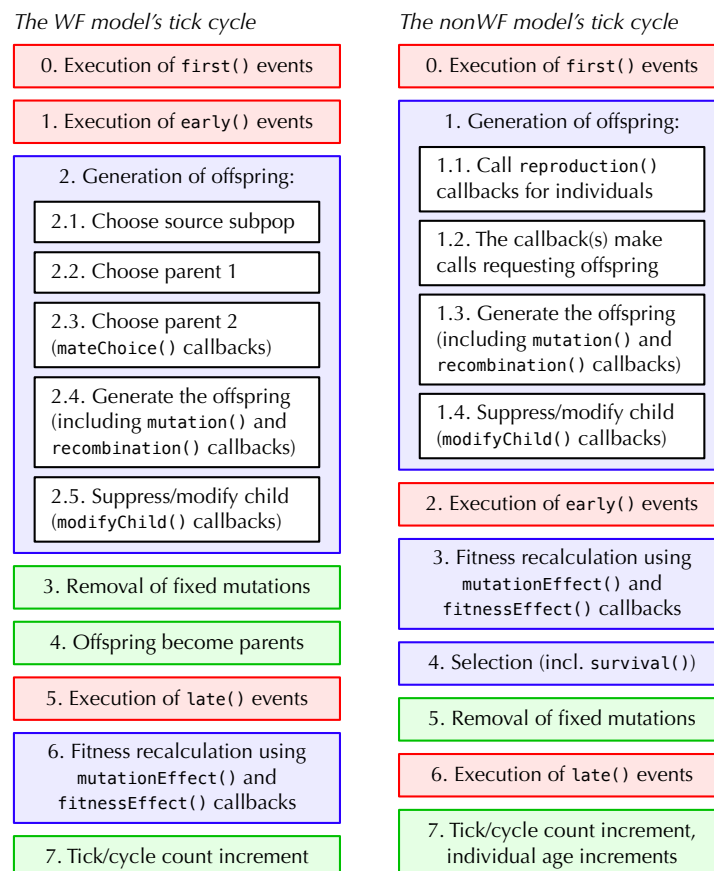
    luck = rbinom(hosts.size(), 1, P_survives);
    dead = hosts[luck == 0];
    dead.fitnessScaling = 0.0;
}

```

Note that it is now an `early()` event, which runs *after* reproduction in nonWF models. We are now doing birth-death/birth-death/birth-death. Since we want the parents to die before any other behavior occurs, we now do that at the top of the event, rather than in `survival()` callbacks (again, section 16.15 discusses these varying approaches to implementing non-overlapping generations). We now use `fitnessScaling` to manage all deaths, and at each stage we narrow down to focus upon the survivors of the previous step. SLiM will then automatically kill all individuals with `fitnessScaling == 0` during the viability/survival cycle stage. This version of the model still uses `tag` for the parasitoids, to count their kills, but does not use `tag` for the hosts since their mortality is managed directly with `fitnessScaling`. The only other change needed for this second version of the model, then is to remove the check of the `tag` value at the top of the host reproduction() callback; if the hosts make it that far, they get to reproduce. (The full recipe is available online and in SLiMgui, as usual.)

Note that the order in which these two reproduction() callbacks are declared in the script does not matter. If you look back to the timing diagram in section 19.2, that discussion mentions that when the tick schedule for two species coincide in a given tick, “both species are active and their cycles are interleaved”. It’s time to talk about what that really means.

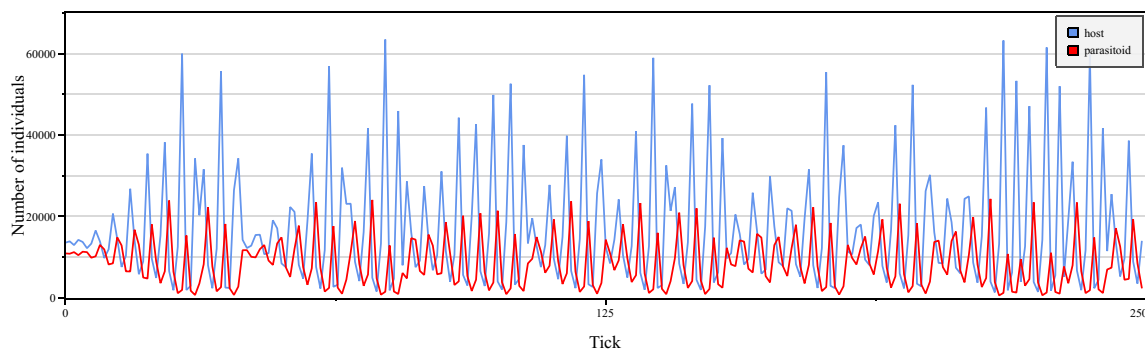
The tick cycle for WF models was introduced in section 1.3, and you may have also seen the tick cycle diagrams for WF and nonWF models in chapters 22 and 23. Until SLiM 4 those diagrams were black and white, though, so you may not have seen the new color versions:





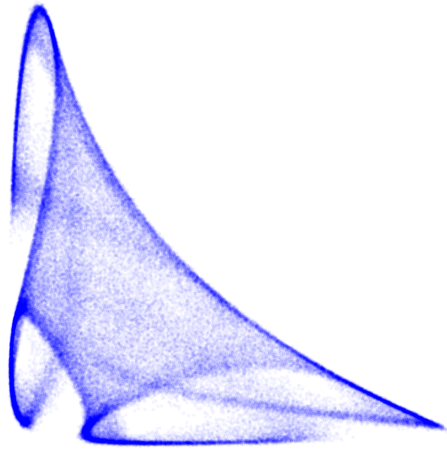
In single-species models, the colors shown here are unimportant; the tick cycle works just as it did in SLiM 3. In multispecies models, the colors tell you how each stage is managed when more than one species is active in the same tick. Stages that are colored **red** run events: `first()` events, `early()` events, and `late()` events. These stages run their events in *definition order*: the order in which the events are defined, from top to bottom in the model script, just as in SLiM 3. Stages that are colored **blue** involve callbacks having to do with reproduction, fitness calculation, and survival. Each of these stages will execute, for multiple species, in *species declaration order*: the order in which the species are declared, with `species name initialize()` declarations, from top to bottom in the model script. In this model species `host` is declared first and species `parasitoid` is declared second; when executing the reproduction stage of the WF tick cycle, SLiM therefore reproduces all of the hosts first, and then reproduces all of the parasitoids second. The order in which the `reproduction()` callbacks are defined does not matter at all (but it is good practice to define them in the same order as the species declarations, to reduce confusion). Finally, stages that are colored **green** will be run in whatever order SLiM chooses; these stages do not involve the execution of any Eidos script, and the order in which they process their work does not matter since the effect on each species is independent.

OK, that was a lot of discussion. Does the model work? If we run it (the first version of it, with the death-birth/death-birth/death-birth life cycle, so that SLiMgui's multispecies population size graph shows what we want it to show), here's what we get:

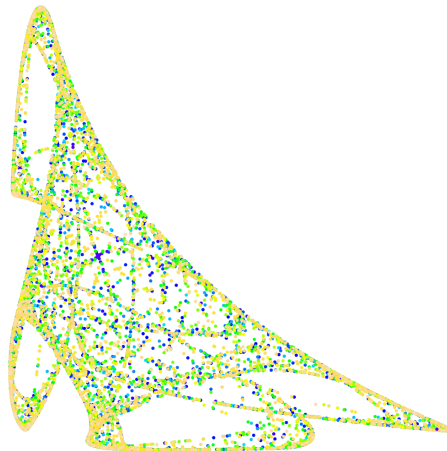


If you compare this to the plot of the deterministic model's dynamics in section 19.3, you can see that they are extremely similar; we do seem to have captured the model correctly. One difference, though, is that the model diverges from steady-state dynamics into cycling much more quickly; the deterministic model was approximately steady for perhaps 25 ticks, whereas this individual-based version of the model starts cycling almost immediately. This is because the individual-based version includes stochasticity, both in the parasitism and the competition; that moves the host population size away from the steady-state equilibrium more rapidly.

We have an even better way to check that this model matches the deterministic model, though: we can replicate the phase-space diagram that we saw in section 19.3. To do so, let's increase  $S$  to  $10^3$  (to make the dynamics more accurate and decrease the effect of stochasticity), and change the end tick to 100000 (so we get lots of data points). Recycle and run to completion, which will take a while (several hours) – with  $S = 10^3$  the population sizes are often as high as half a million, so this is a pretty big model. Then open the multispecies population size graph. Click the ⚙️ “action button” at lower right in the graph window, choose **Export Data...** from the menu that pops up, and save the data for the graph to a file. We won't go through the remaining steps in detail here, but in short, you can bring the exported data into R or Python, and plot  $x_1$  versus  $x_2$  (host density versus parasitoid density) to get a phase space diagram as in section 19.3:



It looks pretty much the same as the deterministic model's attractor (except for the color scheme used by Sebastian, which indicates the time at which each point was visited):



This indicates that the model is following the same attractor – following the same dynamics, in other words. It would likely look even better at  $S = 10^4$ , but that would take quite a bit of time and memory!

There is lots more we could do to make this model more realistic. We could convert the host and parasitoid species to be sexual, and have the females reproduce after finding a male mate. We could add genetics to the host that govern some kind of resistance to the parasitoid (altered cuticular hydrocarbons?), and add genetics to the parasitoid that can evolve to counter the host's evolution (altered olfactory preferences?); this could then enable coevolutionary dynamics where the host evolves to evade the parasitoid, and the parasitoid “chases” the host's evolution. We could also introduce a spatial element to the model, so that the density of hosts and parasitoids can vary across the landscape. With such changes, we could see whether the dynamics tend to stabilize or destabilize the cycling behavior of the model, whether they make extinction of one of the species more or less likely, and so forth. All of these are interesting angles! Since we have never scripted spatial interactions between individuals in different species, however, that seems like the most interesting problem to explore next.

## 19.5 A continuous-space host–parasitoid model

The previous two sections explored a host–parasite model from Sebastian Schreiber, as described in Faure and Schreiber (2014). The first version of the model was deterministic, while the second version was individual-based; in other respects, as we saw with various plots, the two models behaved similarly. We will now depart from Sebastian’s prior work to build a derivative of the previous recipe that adds continuous space, to see what that does to the dynamics of the model. The techniques used will be similar to those employed for continuous-space models in previous chapters; you may wish to review those concepts before continuing (see, for example, chapter 15 and section 16.10).

The model is quite similar to the first recipe of section 19.4, just with spatiality added. Here are the `initialize()` callbacks for the recipe:

```
species all initialize() {
  defineConstant("K", 100);
  defineConstant("R", log(20));
  defineConstant("A", 0.015);
  defineConstant("SIDE", 10); // one side length; square root of S
  defineConstant("S", SIDE*SIDE);
  defineConstant("N0_host", asInteger((135.6217 + 0.01) * S));
  defineConstant("N0_parasitoid", asInteger((109.3010 + 0.01) * S));

  defineConstant("S_P", 0.5); // parasitoid hunting/mating kernel width
  defineConstant("S_H", 0.2); // host competition/mating kernel width
  defineConstant("D_H", 0.2); // host dispersal kernel width
  defineConstant("CROSS_SCRIPT", "subpop.addCrossed(individual, mate);");

  initializeSLiMModelType("nonWF");

  // parasitoids looking for things (long search distance)
  initializeInteractionType(1, "xy", maxDistance=S_P);
  i1.setInteractionFunction("l", 1.0);

  // hosts looking for things (short search distance)
  initializeInteractionType(2, "xy", maxDistance=S_H);
  i2.setInteractionFunction("l", 1.0);
}

species host initialize() {
  initializeSpecies(avatar="🐛", color="cornflowerblue");
  initializeSLiMOptions(dimensionality="xy");
  // tag values in host indicate survival (0) or death (1)
}

species parasitoid initialize() {
  initializeSpecies(avatar="🐜", color="red");
  initializeSLiMOptions(dimensionality="xy");
  // tag values in parasitoid count successful hunts
}
```

We’ve added one `InteractionType` using a long maximum interaction distance for parasitoid searches (for mates and prey), and another with a shorter distance for host searches (for mates). Note that the interaction types are defined in the `species all initialize()` callback; this is required, since interaction types are not species-specific (although they might be used in a species-specific way, as `i2` will be used only for hosts to search for mates). One design decision to note

here is that both interactions use a linear (type "l") interaction function. This is substantially faster to calculate than, for example, a Gaussian kernel, although perhaps less biologically accurate.

Several new parameters were added to support those spatial behaviors, too, such as the interaction and dispersal kernel widths. The parasitoids don't get a dispersal kernel width because they will be born at the location of the host they hatched from. Of course we also declare the dimensionality of both species to be "xy" now; we will use reflecting boundaries (for greater speed), so we don't declare any periodicity. Hopefully edge effects will not be too strong, since the landscape size is fairly large compared to the interaction distances; if they prove to be a problem, it would be trivial to switch to periodic boundaries (see section 15.12).

We now define  $S$ , the total habitat size, as a derived parameter; it is now the square of  $SIDE$ , which is the length of one side of the (square) landscape we will set up. In fact the model no longer uses  $S$  at all, though, so it is defined just for comparison with the previous models. (Remember that we used  $S$  to convert from population sizes to densities; but now all densities will be *local* densities, computed using `InteractionType`).

The only other change is the addition of a defined `lambda`, `CROSS_SCRIPT`, that we will use for bulk reproduction, as shown below.

Here's the population setup and the end event:

```
ticks all 1 early() {
  host.addSubpop("p1", N0_host);
  p1.setSpatialBounds(c(0, 0, SIDE, SIDE));
  p1.individuals.setSpatialPosition(p1.pointUniform(N0_host));

  parasitoid.addSubpop("p2", N0_parasitoid);
  p2.setSpatialBounds(c(0, 0, SIDE, SIDE));
  p2.individuals.setSpatialPosition(p2.pointUniform(N0_parasitoid));
}
ticks all 250 late() {
}
```

We now set up the bounds of the  $SIDE \times SIDE$  landscape with a call to `setSpatialBounds()` for each subpopulation. Then the initial individuals of each subpopulation get distributed randomly across the landscape using `pointUniform()`.

Age-based mortality is exactly the same as it was in the previous recipe, using `survival()` callbacks to kill off non-juveniles:

```
// non-overlapping generations: parents die, offspring live
species host survival() {
  return (individual.age == 0);
}
species parasitoid survival() {
  return (individual.age == 0);
}
```

Let's look at the parasitoid's `reproduction()` callback next, since it is simpler:

```
species parasitoid reproduction() {
  // reproduce each parasitoid as many times as it parasitized
  litterSize = individual.tag;

  if (litterSize > 0)
  {
    mate = i1.drawByStrength(individual);
```

```

    if (mate.size())
    {
        offspring = sapply(seqLen(litterSize), CROSS_SCRIPT);

        // vectorized set of offspring positions to prey positions
        offspring.setSpatialPosition(individual.getValue("PREY_POS"));
    }
}

```

We now find a mate using `drawByStrength()`. All offspring are generated with a single call to `sapply()` using `CROSS_SCRIPT`; this gathers the returned offspring into a vector for us, making it easy to set their positions from state saved under the "PREY\_POS" key of the parasitoid (as we will see) using a single vectorized call. Now here is the host's `reproduction()` callback:

```

species host reproduction() {
    // only hosts tagged 0 (survived) get to reproduce
    if (individual.tag != 0)
        return;

    // reproduce each host with a mean of exp(r) offspring
    mate = i2.drawByStrength(individual);

    if (mate.size())
    {
        litterSize = rpois(1, exp(R));

        if (litterSize > 0)
        {
            offspring = sapply(seqLen(litterSize), CROSS_SCRIPT);

            // vectorized set of offspring spatial positions, based
            // on the first parent position plus dispersal D_H
            positions = rep(individual.spatialPosition, litterSize);
            positions = positions + rnorm(litterSize * 2, 0, D_H);
            positions = p1.pointReflected(positions);
            offspring.setSpatialPosition(positions);
        }
    }
}

```

As before, the `tag` of the focal individual indicates whether it survived to reproduce; see the previous section for discussion of this design. We use `drawByStrength()` to find a mate, and do a bulk reproduction with `sapply()`, just as we did for the parasitoids. Here, however, we set offspring positions based upon the position of the first parent (`individual.position`) plus a random dispersal drawn with `rnorm()` from a Gaussian dispersal kernel of width `D_H`, with reflecting boundaries. All of that can be done with vectorized calls for greater efficiency; this is somewhat important since the mean litter size of the hosts is so large.

Finally – saving the best for last! – here is the behavior event that handles hunting and competition:

```

ticks all 2: first()
{
    host_pop = host.subpopulations;
    hosts = host_pop.individuals;
    parasitoid_pop = parasitoid.subpopulations;
    parasitoids = parasitoid_pop.individuals;
}

```

```

// assess densities, per individual
i1.evaluate(c(host_pop, parasitoid_pop));
i2.evaluate(host_pop);
parasitoid_density_byhost = i1.localPopulationDensity(hosts, parasitoid_pop);

// hunt: each parasitoid counts its successes,
// and remembers the positions of its prey;
// each host tracks whether it was killed
parasitoids.tag = 0;
parasitoids.setValue("PREY_POS", NULL);
P_parasitized_byhost = 1 - exp(-A * parasitoid_density_byhost);
killed = (runif(hosts.size()) < P_parasitized_byhost);
hosts.tag = asInteger(killed);           // T/1 means killed
preys = hosts[killed];
for (prey in preys)
{
  hunter = i1.drawByStrength(prey, 1, parasitoid_pop);
  preyPos = prey.spatialPosition;
  preyPos = c(hunter.getValue("PREY_POS"), preyPos);
  hunter.tag = hunter.tag + 1;
  hunter.setValue("PREY_POS", preyPos);
}
unhunted = hosts[!killed];

// competition: kill a fraction of unhunted; note
// that this is based on pre-parasitism density
host_density_by_unhunted = i2.localPopulationDensity(unhunted, host_pop);
P_survives_by_unhunted = exp(-host_density_by_unhunted / K);
survived = (runif(unhunted.size()) < P_survives_by_unhunted);
dead = unhunted[!survived];           // T means survived
dead.tag = 1;                         // mark as dead
}

```

The logic of this code is parallel to that of the previous recipe; it has just been “spatialized”. We start off by evaluating each interaction for the subpopulations that will be used in queries (both in this event and in the reproduction() callbacks that execute after this event). To assess densities, we now use the localPopulationDensity() method to assess the local density of parasitoids in the vicinity of each host. This gives us a vector of density values, one per host; the variable name parasitoid\_density\_byhost is a reminder of that. We use that to compute a vector of parasitism probabilities, one for each host; and then to draw, with runif(), the actual fate (parasitized or not) of each host. We set the tag values of the hosts accordingly; those values will be checked in the host reproduction() callback, as we already saw above.

Next we loop over the hosts that were killed. For each one, we choose the parasitoid that got the kill using drawByStrength() and we credit the kill to that individual by incrementing its tag value, as before. Now we do an additional bit of bookkeeping as well: we get the spatial position of the prey, and save it under the "PREY\_POS" key on the parasitoid. As we saw above, the parasitoid reproduction() callback will use that information to generate its offspring at the position of the killed host, a nice bit of biological accuracy that might actually matter to the dynamics since it causes the offspring of the parasitoids to, in effect, disperse towards the nearest groups of hosts.

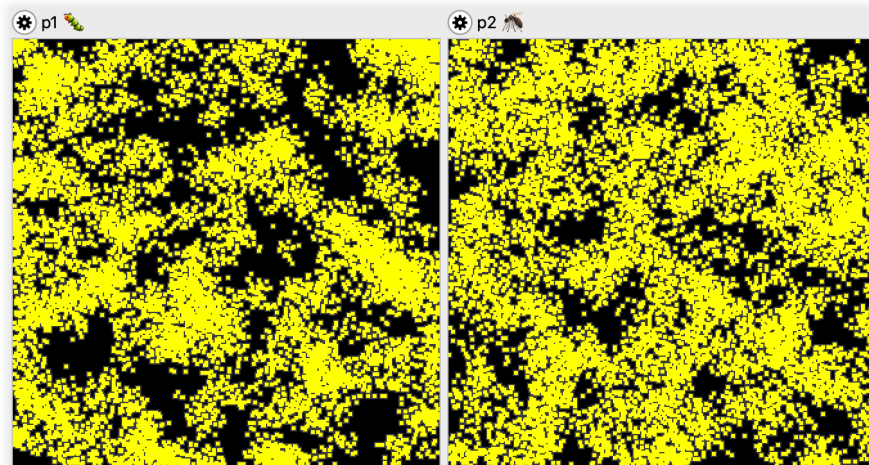
Finally, we implement spatial competition. Instead of basing this upon overall density, it is now based upon the local density of hosts around each individual, using localPopulationDensity().




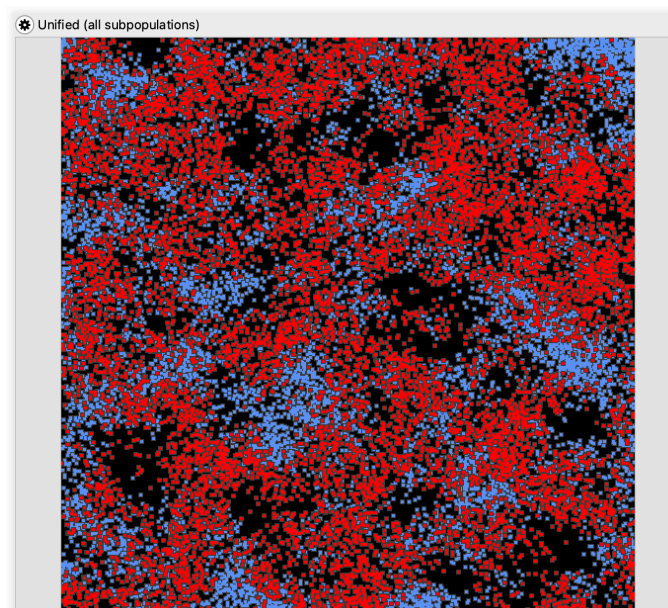
The competition equation is otherwise unchanged, but the probabilities and outcomes are calculated as vectors by host, just as the parasitism dynamics were earlier.

It is worth emphasizing the vectorized design of this code, because it is quite important for performance. It is possible to vectorize most uses of `InteractionType`, as well as most code for getting and setting of positions, enforcing of boundary conditions, and so forth. It is worth the effort to do so; as well as improving runtimes, it also makes your code simpler, and thus less likely to contain bugs. (It would be nice to be able to vectorize the `for` loop over the preys as well; the `setValue()` call is hard to convert to a vectorized design, though, unfortunately.)

So, how does this model look in practice? First of all, here's a snapshot of the individuals view, on the `all` tab, showing the host (left) and parasitoid (right) distributions in a typical tick:

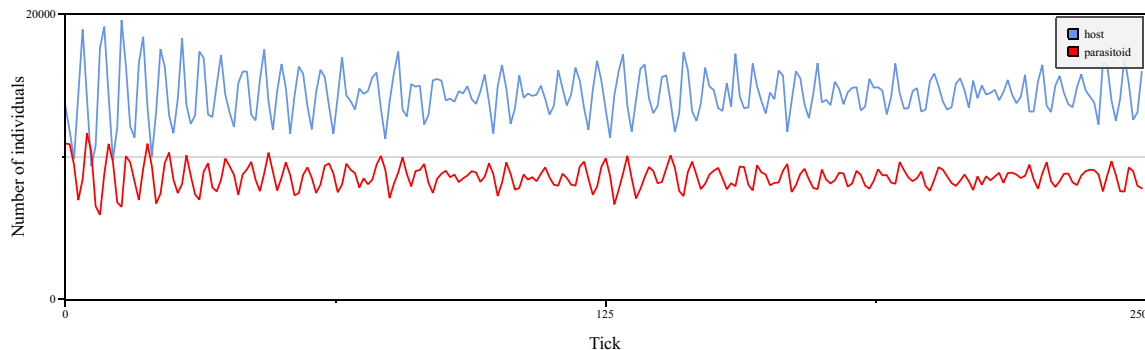


Their distributions are fairly clumpy. They look inversely correlated to me, but it is not really clear – it's hard to accurately judge such things by eye, since the human brain is so finely tuned for finding patterns (even where none exist). Happily, when on the `all` species tab SLiMgui can assist us. If we click on the  action button for either of the subpopulations displayed in the individuals view, we can choose “Display spatial (unified)” from the resulting menu. That shows us this view of the individuals instead (after resizing vertically to make the view bigger):



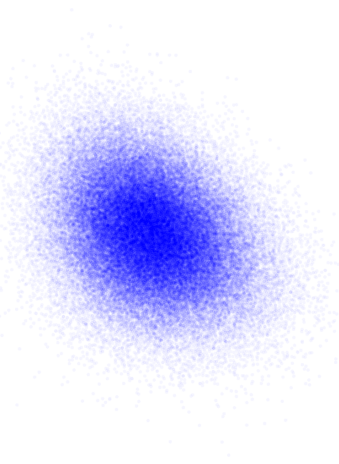
The individuals here are colored using the colors previously set in `initializeSpecies()` – the same colors that SLiMgui has been using for the multispecies population size plots – so that the two species can be easily distinguished. In this view it is at least somewhat easier to see that the distributions of the two species are indeed inversely correlated. If you step forward, you see that the distributions change rapidly; presumably the parasitoids are fairly effective at hunting in dense clusters of hosts.

A plot of population sizes over time shows that the population dynamics have been markedly stabilized by spatiality:



Of course this may depend upon the particular parameter values used; quite a bit of work would need to be done before drawing any general conclusions about this model. Still, this result is in line with expectations (since spatiality allows each local area to “decouple” from other areas and do its own thing, stabilizing the overall population size since it is summed across multiple decoupled areas).

Finally, let’s look at the phase space diagram for this version of the model (using `SIDE == 10`, and thus `S == 10^2`, and a total time of 100000 ticks) to see what has happened to the attractor:



It is now very blurry and ill-defined. This is probably again due to the “decoupling” of local areas, each of which can progress somewhat independently. It may be that local areas on the landscape still follow the same attractor, more or less; the more decoupled that local areas become (due to shorter dispersal and interaction distances, for example), the more that might be true. Ironically, the more well-defined that local areas become due to decoupling, the more that averaging the dynamics across all of those local areas will tend to produce shapeless mush, as above. On the other hand, the more tightly coupled that local areas are (due to longer dispersal

and interaction distances, for example), the more the whole landscape will presumably follow the mathematical dynamics in unison, producing a sharper phase space plot. At the limit of panmixia (or perhaps I should call it “paninteraxia”?), one would expect to recapture the behavior of the non-spatial model seen in the previous section.

## 19.6 A coevolutionary host–parasitoid trait-matching model

We have been exploring the host–parasitoid model of Faure and Schreiber (2014) in the past several sections. We will continue that exploration here, returning to the simple individual-based model of section 19.4 (in other words, *not* bringing forward the continuous-space dynamics of section 19.5) and then adding genetics. The end result will be a model of coevolution between the host and the parasitoid, with the host trying to evolve a phenotype that escapes from the predation of the parasitoid, and the parasitoid evolving in response to continue to hunt the host despite its changing phenotype.

Let’s start, as usual, with the initialization code:

```
species all initialize() {
  defineConstant("K", 100);
  defineConstant("R", log(20));
  defineConstant("A", 0.015);
  defineConstant("S", 10^2); // larger is more stable, but slower
  defineConstant("N0_host", asInteger((135.6217 + 0.01) * S));
  defineConstant("N0_parasitoid", asInteger((109.3010 + 0.01) * S));

  defineConstant("S_M", 1.0); // parasitoid/host matching width
  defineConstant("S_S", 2.0); // stabilizing fitness function width

  initializeSLiMModelType("nonWF");
}
species host initialize() {
  initializeSpecies(avatar="🐛", color="cornflowerblue");
  // tag values in host indicate survival (0) or death (1)
  // tagF values in host are QTL-based additive phenotypes

  // one short QTL controlling parasitism avoidance
  initializeMutationType("m1", 0.5, "n", 0.0, 0.1);
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 9999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
species parasitoid initialize() {
  initializeSpecies(avatar="🐜", color="red");
  // tag values in parasitoid count successful hunts
  // tagF values in parasitoid are QTL-based additive phenotypes

  // one short QTL controlling host trait matching
  initializeMutationType("m2", 0.5, "n", 0.0, 0.1);
  initializeGenomicElementType("g2", m2, 1.0);
  initializeGenomicElement(g2, 0, 9999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}
```

This is similar to what we have seen in previous iterations of this model, setting each species avatar and color with `initializeSpecies()`. However, each species now has genetics: a mutation type, a genomic element type, a chromosome defined by a single short genomic element (representing one gene, a single QTL 10,000 bases long that controls a single quantitative trait in each species), and mutation and recombination rates. We also define a couple of new model parameters, `S_M` and `S_S`, that control the phenotypic effects in the model.

Since the genetics we are modeling here are quantitative, we define `mutationEffect()` callbacks that set the fitness effect of `m1` and `m2` to neutral:

```
species host mutationEffect(m1) { return 1.0; }
species parasitoid mutationEffect(m2) { return 1.0; }
```

This technique may be familiar by now, but if not, see chapter 13 for discussion of quantitative trait modeling in SLiM.

Next let's look at the population setup:

```
ticks all 1 early() {
  host.addSubpop("p1", N0_host);
  parasitoid.addSubpop("p2", N0_parasitoid);

  log = community.createLogFile("host-parasite log.txt", logInterval=1);
  log.addTick();
  log.addPopulationSize(host);
  log.addMeanSDColumns("host", "p1.individuals.tagF;");
  log.addPopulationSize(parasitoid);
  log.addMeanSDColumns("parasitoid", "p2.individuals.tagF;");
}
```

This is the same except that we create a `LogFile` object that we use to log out information in each tick: the size of the host population and the mean and standard deviation of its quantitative trait value, and the size of the parasitoid population and the mean and standard deviation of its quantitative trait value. We've seen `LogFile` several times before, so that should be review.

The termination event is exactly the same as before:

```
ticks all 10000 late() {
}
```

Remember that section 19.4 actually had two versions of its recipe; the first used `survival()` callbacks to implement non-overlapping generations, while the second used `fitnessScaling` in an `early()` event. We will follow the second version, using an `early()` event; however, as discussed in section 19.4, this means that the population sizes plotted in SLiMgui (and those logged out by `LogFile`'s automatic logging facility, which happens at the end of each tick) will not be directly comparable to those in the deterministic model. We will do it nevertheless; it makes for a simpler and more efficient model design. So just keep in mind that we can't make that direct comparison to previous results.

In addition to enforcing non-overlapping generations, the version of this `early()` event in this recipe now exerts stabilizing selection upon the quantitative traits of both species, keeping them in bounds. We want the traits to be under stabilizing selection so they don't just run off to infinity; but in any case, most biological traits are subject to some kind of constraint and cannot just evolve directionally forever.

The script for the early() event:

```
ticks all early() {
  // calculate phenotypes and implement stabilizing selection
  scale = dnorm(0.0, 0.0, S_S);

  hosts = host.subpopulations.individuals;
  phenotypes = hosts.sumOfMutationsOfType(m1);
  hosts.fitnessScaling = dnorm(phenotypes, 0.0, S_S) / scale;
  hosts.tagF = phenotypes;

  parasitoids = parasitoid.subpopulations.individuals;
  phenotypes = parasitoids.sumOfMutationsOfType(m2);
  parasitoids.fitnessScaling = dnorm(phenotypes, 0.0, S_S) / scale;
  parasitoids.tagF = phenotypes;

  // non-overlapping generations: parents die, offspring live
  hosts[hosts.age > 0].fitnessScaling = 0.0;
  parasitoids[parasitoids.age > 0].fitnessScaling = 0.0;
}
```

We calculate phenotypic trait values with `sumOfMutationsOfType()` as usual (see chapter 13), and put those values into the `tagF` property of the individuals for later use. Stabilizing selection is implemented with `dnorm()`, as in, for example, section 13.6; we scale the results of `dnorm()` so that an individual whose phenotype is at the optimum (a value of 0.0, for both species) has a fitness that is neutral, 1.0. Again, all of this is more or less review, so let's move on.

The interesting part is how the logic in the `first()` event changes. Here's the new script for that event:

```
ticks all 2: first()
{
  hosts = host.subpopulations.individuals;
  parasitoids = parasitoid.subpopulations.individuals;

  // assess densities
  x1 = hosts.size() / S;           // host density
  x2 = parasitoids.size() / S;     // parasitoid density

  // assess matches between hosts and the mean parasitoid
  host_values = hosts.tagF;
  parasitoid_values = parasitoids.tagF;
  mean_parasitoid = mean(parasitoid_values);
  scale = dnorm(0.0, 0.0, S_M);
  host_match = dnorm(host_values, mean_parasitoid, S_M) / scale;

  // hunt: each parasitoid counts its successes, and
  // each host tracks whether it was killed
  parasitoids.tag = 0;
  P_parasitized_byhost = 1 - exp(-A * x2 * host_match);
  killed = rbinom(hosts.size(), 1, P_parasitized_byhost);
  hosts.tag = killed; // 1 means killed
  hunters = sapply(hosts[killed == 1], "sample(parasitoids, 1, " +
    "weights=dnorm(applyValue.tagF - parasitoid_values, 0.0, S_M));");
  for (hunter in hunters)
    hunter.tag = hunter.tag + 1;
  survivors = hosts[killed == 0];
}
```



```

    // competition: kill a fraction of survivors; note
    // that this is based on pre-parasitism density
    P_survives = exp(-x1 / K);
    survived = rbinom(survivors.size(), 1, P_survives);
    dead = survivors[survived == 0]; // 1 means survived
    dead.tag = 1; // mark as dead
}

```

This version starts off the same, calculating the density of each species, since those values feed into the equations below. Next we have some new code that uses `dnorm()` to assess the matches between hosts and parasitoids – or more specifically, between the quantitative trait value of each host and the mean trait value of the parasitoids. A host with a trait value close equal to the parasitoid mean has a match value of 1.0; the more different the host trait value is from the mean, the smaller the match value. (This sort of model is often called a *trait-matching model*, in fact.)

Those match values, in `host_match`, feed in to the calculation of the probability that a given host will be parasitized. In section 19.4, we calculated a value `P_parasitized` that was the probability for every host – there were no individual differences. Now we instead calculate a vector of probabilities, `P_parasitized_byhost`, that has the probability for each host depending upon its match value. A vectorized call to `rbinom()` determines which hosts get parasitized, using that vector of host-specific probabilities. Remember that `x2` is the density of parasitoids; in effect, hosts with a low match score experience a lower effective density of parasitoids, since `x2` is now multiplied by the match value for each host.

Next we determine which parasitoid killed each host. Again, in section 19.4 there were no individual differences; the code there thus drew all of the parasitoids with a single call to `sample()`, with equal probabilities for every parasitoid. Now, we need to make a separate call to `sample()` for each host, with a specially calculated vector of weights that represent the trait-based match between the focal host and every parasitoid. We use the `sapply()` function to do that relatively efficiently, although this is still by far the predominant bottleneck in this model – it takes almost 98% of the runtime of this model. (Note that a continuous-space version of this model would probably actually run much *faster* than this version, because only spatially local interactions would need to be considered – the number of possible parasitoid matches for a given host would only be a small fraction of the whole population!)

At the end, we give successful hunters their reproduction boost by incrementing their `tag` value, and handle density-dependent competition between hosts; this code is essentially unchanged.

Finally, we need to reproduce both species. The code for reproduction is unchanged from section 19.4; each host generates a litter of mean size  $\exp(R)$ , while each parasitoid generates an offspring for each host it killed. We'll show that reproduction code again here for completeness:

```

species host reproduction() {
    // only hosts tagged 0 (survived) get to reproduce
    if (individual.tag != 0)
        return;

    // reproduce each host with a mean of exp(r) offspring
    litterSize = rpois(1, exp(R));

    if (litterSize > 0)
    {
        mate = subpop.sampleIndividuals(1, exclude=individual);
        for (i in seqLen(litterSize))
            subpop.addCrossed(individual, mate);
    }
}

```



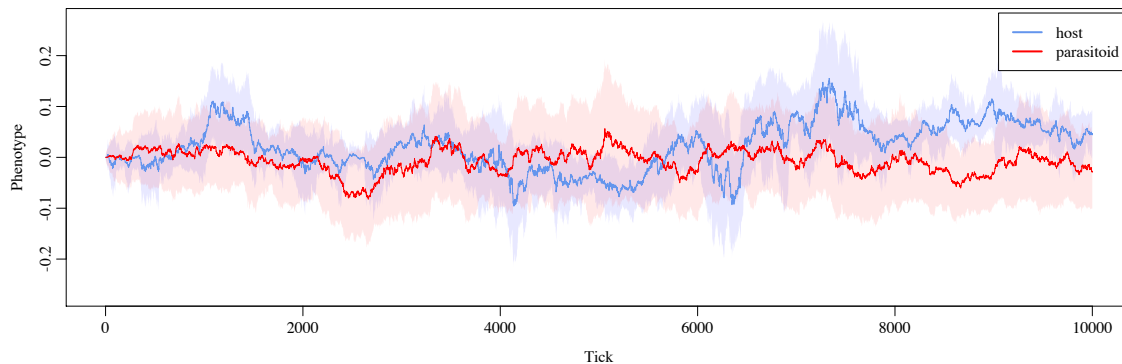
```

species parasitoid reproduction() {
  // reproduce each parasitoid as many times as it parasitized
  litterSize = individual.tag;

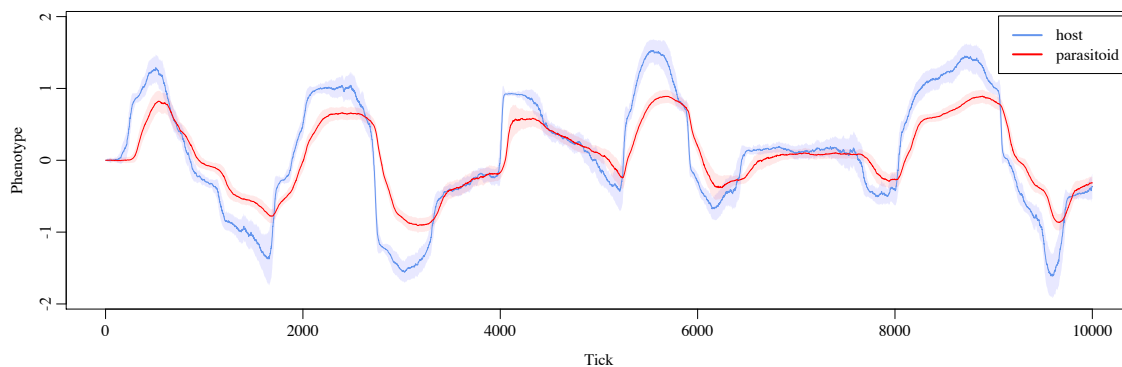
  if (litterSize > 0)
  {
    mate = subpop.sampleIndividuals(1, exclude=individual);
    for (i in seqLen(litterSize))
      subpop.addCrossed(individual, mate);
  }
}

```

So, how does this model behave? Well, first let's look at a plot of the model's dynamics with the trait-based matching in the hunt code turned off (following recipe 19.4). Stabilizing selection is still on, so both phenotypic traits drift constrained by that stabilizing selection, producing a sort of "random walk". Note that we are now plotting mean phenotype on the y axis against time (ticks) on the x, with shading to show the standard deviation around the mean:

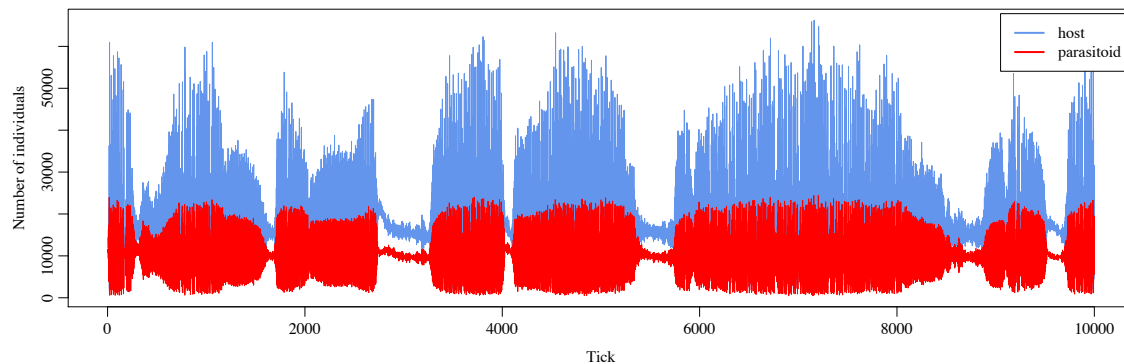


This plot was made in R using the data logged by `LogFile`. Now let's compare the dynamics shown by the full model, with parasitoid hunting efficacy depending upon the match to the host:



The dynamics are completely different. We now see the host's mean trait value evolve away from that of the parasitoid, and the parasitoid's mean trait value chases after it. However, as the host's value gets further from 0.0 the pressure back towards 0.0 exerted by stabilizing selection gets stronger and stronger. At a certain point, it becomes worthwhile for the host to actually jump across the parasitoid and run back towards 0.0 and beyond into negative values. The two species therefore oscillate back and forth between positive and negative trait values in an endless "Red Queen" chase. Phenotypic variance stays low, because both species are under strong directional selection much of the time.

Finally, it is interesting to look at the population size plot for the full model (remembering that this plot is not directly comparable to those of previous sections, because population sizes in this model are logged after the parental generation has died):



Whenever the host population gets pushed hard against the wall of stabilizing selection, its population size crashes; individual fitness is greatly diminished by the distance from the phenotypic optimum, and meanwhile, hosts are still being killed by the parasitoids as well. This bottleneck persists, with a diminished parasitoid population size as well (since there are so few hosts to lay eggs in), until the hosts find the solution of jumping over the parasitoids; presumably each such jump involves a new mutant of large effect size that rapidly sweeps to fixation, but that would be interesting to confirm by looking at the history of the mutations. There are also a couple of times in this run of the model when the hosts jump over the parasitoids without first being pushed all the way to the wall; those jumps do not involve a population bottleneck.

So, with about two pages of code we have a two-species model of coevolutionary dynamics involving quantitative traits that influence the ecological interactions between the species to produce “Red Queen” phenotypic chasing. Of course this could be taken in any number of directions to explore a wide variety of interesting research questions. One obvious step would be to combine it with the recipe of section 19.5 to add in spatiality; since that should be quite straightforward, it will be left as an exercise for the reader.

As mentioned above, this type of genetic interaction between species is called a *trait-matching model*, because the strength of the interaction depends upon the degree to which the phenotypic trait values of individuals match. Of course interspecies interactions can depend upon genetics in other ways too, as we will see in the next section.

## 19.7 A coevolutionary host–parasite matching-allele model

In the previous section we included genetics in a multispecies model for the first time, looking at a *trait-matching model* in which phenotypic traits governed by quantitative genetics determined the strength of interaction between individuals. Here we will look at a simpler type of genetic interaction called a *matching-allele model*. In such models the genetics involve simple Mendelian alleles, rather than quantitative traits, and the strength of interaction between individuals of different species depends upon the match (or lack thereof) between the alleles they carry.

At this point, we will also move on from the host–parasitoid models of previous sections to explore some other possibilities. Here, we will shift from parasitoids (which kill their host to gain an offspring) to parasites (which just derive some benefit from their host, to the host’s detriment). We will model ungulates, such as deer, infected by parasitic worms. In each tick, the parasites will try to infect one particular host. If that host carries an allele that matches their own, the infection

will be more successful, giving the parasite a fitness boost while decreasing the fitness of the host. This model was originally written by Bob Week; thanks Bob!

Let's start as usual with initialization:

```
species all initialize() {
    defineConstant("L", 1);
}
species host initialize() {
    initializeSpecies(avatar="🦋");

    // one nucleotide controlling infection
    initializeSLiMOptions(nucleotideBased=T);
    initializeAncestralNucleotides(randomNucleotides(L));
    initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-3));
    initializeGenomicElement(g1, 0, L - 1);
    initializeRecombinationRate(1e-8);
}
species parasite initialize() {
    initializeSpecies(avatar="🐛");

    // one nucleotide controlling resistance
    initializeSLiMOptions(nucleotideBased=T);
    initializeAncestralNucleotides(randomNucleotides(L));
    initializeMutationTypeNuc("m2", 0.5, "f", 0.0);
    initializeGenomicElementType("g2", m2, 1.0, mmJukesCantor(1e-3));
    initializeGenomicElement(g2, 0, L - 1);
    initializeRecombinationRate(1e-8);
}
```

The parasitic worms are represented by the caterpillar emoji since the version of macOS I'm on does not have the recently-added worm emoji; such is life. We start by defining a constant  $L$  that is the length of the genetic locus, in nucleotides, that governs the allele matching. Here we use a length of just 1; I think the mechanics of the model ought to work if  $L$  is increased, but the logging code we will see below will not work, so that has not been tested. In any case, if a long matching locus is used then a match between the two species will become less likely, so to drive strong dynamics we will stick with  $L$  of 1 here.

Then we initialize the two species. They are both nucleotide-based, with a single locus of length  $L$  that begins with random nucleotides. A high mutation rate is used here to decrease waiting time, but the model will work similarly with a lower rate.

The other thing to note here is that this is a WF model, unlike the previous couple of sections. Because of that choice, the population size of each species will be fixed throughout, modeling “soft selection”. This means that the extent to which hosts are infected by parasites will affect their relative mating success, but will not influence their overall population size. This could be biologically realistic, if population regulation is due to extrinsic factors such as the amount of available food or territory. It also means that how successful the parasites are at infecting their hosts will not affect *their* population size either, but only their relative reproductive success; this seems a bit harder to justify biologically. In any case, it would be easy to convert this recipe to a nonWF model, and there are lots of other things about it that are biologically unrealistic too! Our focus here is on how to implement the matching-allele model.

Next let's look at the population setup:

```

function (float$)nucleotideFreq(o<Species>$ species, s$ nuc) {
  nucs = species.subpopulations.individuals.genomes.nucleotides();
  return mean(nucs == nuc);
}

ticks all 1 early() {
  host.addSubpop("p0", 1000);
  parasite.addSubpop("p1", 1000);

  log = community.createLogFile("host-parasite log.txt", logInterval=1);
  log.addTick();
  log.addCustomColumn("hA", "nucleotideFreq(host, 'A');");
  log.addCustomColumn("hC", "nucleotideFreq(host, 'C');");
  log.addCustomColumn("hG", "nucleotideFreq(host, 'G');");
  log.addCustomColumn("hT", "nucleotideFreq(host, 'T');");
  log.addCustomColumn("pA", "nucleotideFreq(parasite, 'A');");
  log.addCustomColumn("pC", "nucleotideFreq(parasite, 'C');");
  log.addCustomColumn("pG", "nucleotideFreq(parasite, 'G');");
  log.addCustomColumn("pT", "nucleotideFreq(parasite, 'T');");
}

```

Each species has a fixed size of 1000 individuals; unexciting. We also set up a LogFile object that will log out nucleotide frequencies for the host and parasite in each tick, using a function called nucleotideFreq() that we define. (This is the part of the model that won't work properly with  $L > 1$ ; we would have to think of a better frequency-logging strategy.)

Here's the end event:

```

ticks all 2000 late() {
}

```

As usual we've saved the best for last. Here's implementation of the matching-allele model:

```

ticks all late() {
  // each parasite will pick a random host and try to infect it
  parasites = p1.individuals;
  chosen_hosts = sample(p0.individuals, size(parasites), replace=T);

  for (index in seqAlong(parasites))
  {
    // infection depends upon a match (diploid matching-allele model)
    p = parasites[index];
    h = chosen_hosts[index];
    all_nucleotides = c(p.genomes.nucleotides(), h.genomes.nucleotides());

    if (size(unique(all_nucleotides, preserveOrder=F)) == 1)
    {
      // parasite and host are both homozygous for the same nucleotide(s)
      // with a match, the parasite's fitness increases, the host's decreases
      p.fitnessScaling = 1.5 * p.fitnessScaling;
      h.fitnessScaling = 0.5 * h.fitnessScaling;
    }
  }
}

```

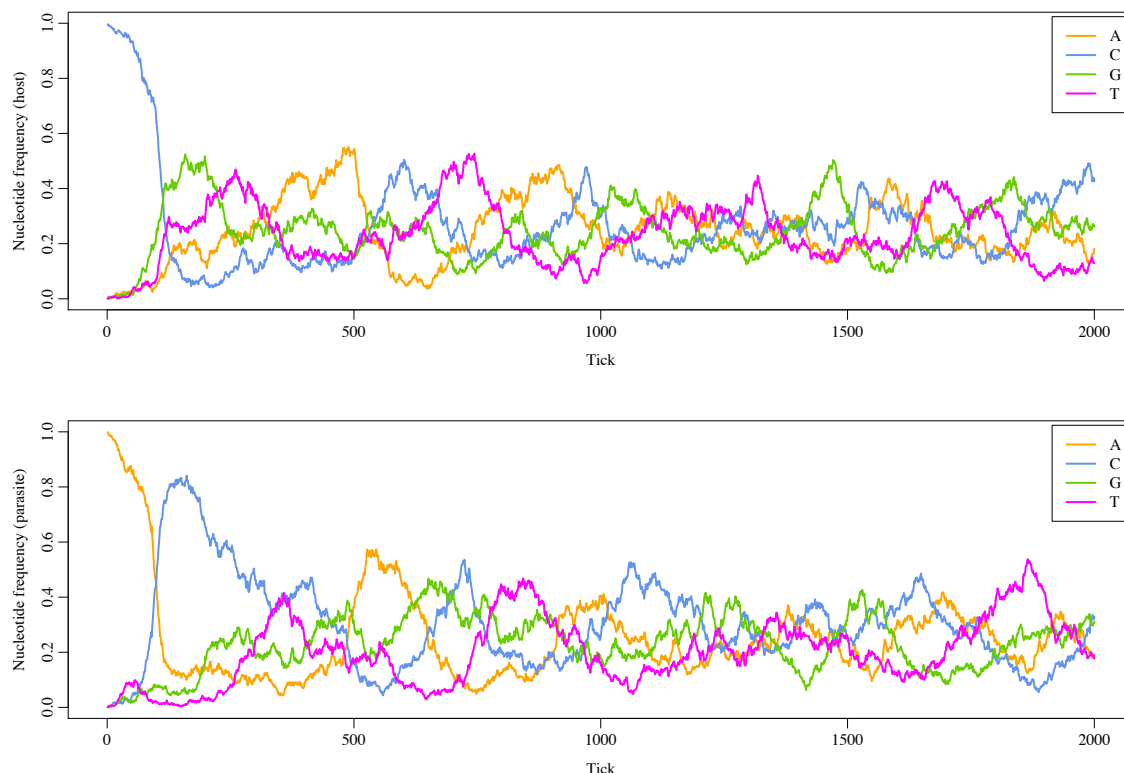
This late() event executes just before fitness calculations, since this is a WF model. It starts by choosing a host for each parasite, using random sampling. This random sampling mimics biology in which a parasite simply attacks the first host it encounters in the environment. One could make

the choice of host be governed by the allele-matching mechanism instead, to mimic biology in which the parasite actively seeks out a host that is vulnerable to it, but we have not done that here; the previous sections illustrate some strategies for that type of implementation.

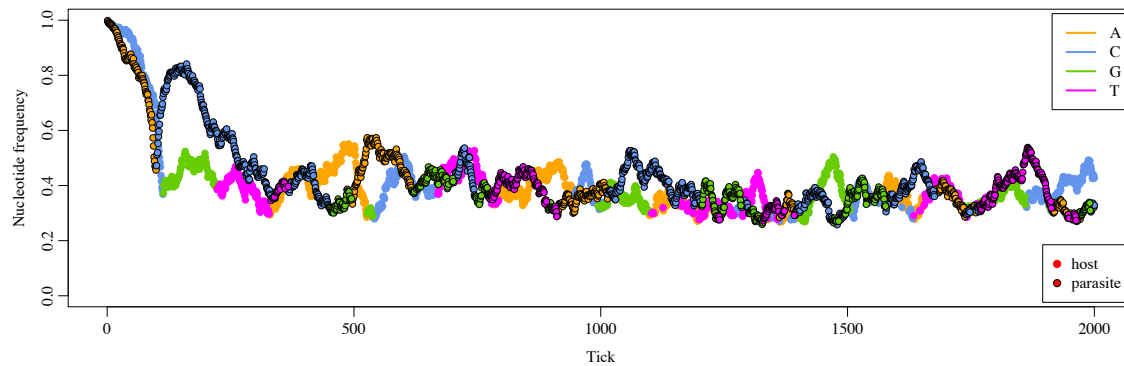
Then we loop over the parasites and their chosen hosts. For each host–parasite pair, we retrieve their nucleotide sequences; this results in a vector containing four `string` elements, two for the (diploid) parasite and two for the (diploid) host. In this particular model we want to require an exact diploid match between host and parasite, so we use `unique()` to check that all four sequences are identical; if you want different matching logic you would just implement that instead. This matching criterion basically means that the matching mechanism is recessive, and so both alleles of the host and parasite must be the same for that recessive allele to be expressed; you could instead model complete dominance, or some kind of incomplete-dominance scheme, by replacing the call to `unique()`.

Finally, if the host and parasite have a complete diploid match then we model increased fitness for the parasite, by scaling up its `fitnessScaling` value, and decreased fitness for the host. We do this by multiplying the existing `fitnessScaling` value; this produces multiplicative effects for hosts that are involved in multiple successful infections. In this implementation we do *not* modify the fitness of a parasite that does not match its host; it remains `1.0`, whereas parasites with a successful match get `1.5`. This means that all parasites have a chance to reproduce; the ones with a match are just more *likely* to reproduce. One could instead set the `fitnessScaling` of non-matched parasites to `0.0`, reflecting biology in which they fail to reproduce. As usual, start with the biology of your system and write model logic that implements that biology (without overcomplicating your model with biological details that are unimportant to your research question, of course).

That’s the whole model; since it doesn’t involve spatiality or quantitative traits it is quite simple really – about a page of code, most of which is the setup of the genetic structure and the logging. What does it do? Here are plots of allele frequencies over time:

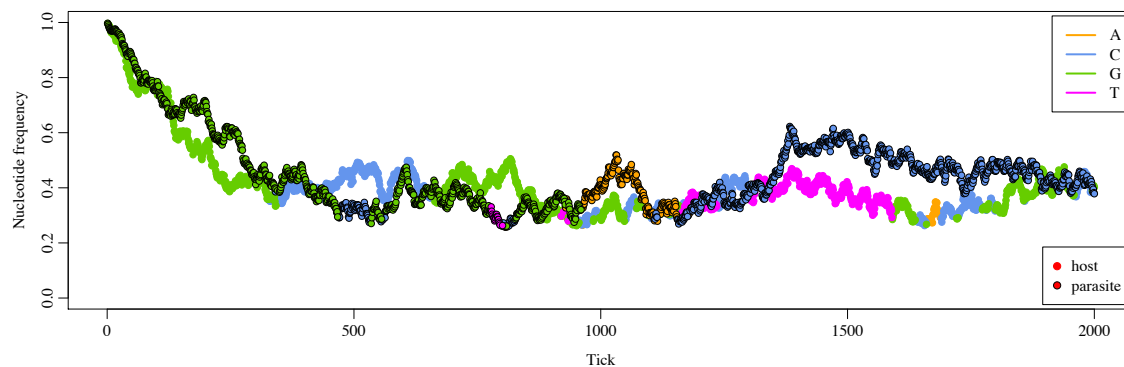


If you look closely, you can see a general pattern: whatever nucleotide is highest frequency in the host soon becomes the highest-frequency nucleotide in the parasite, and then the host moves to a different nucleotide. These are “Red Queen” chasing dynamics as seen in the previous section also. These plots are not terribly easy to read, though. Here is a different style of plot in which only the nucleotide of highest frequency is shown, allowing us to plot host and parasite together without rendering the plot unreadable (I believe Tony Long calls this a “Pollock plot”):



Perhaps it is easier here to see that the highest-frequency allele in the host (circles without outlines) is soon matched by the highest-frequency allele in the parasite (circles with outlines).

Finally, for those who think that this pattern might just be a coincidence – seeing a pattern visually where none really exists – here is the same recipe run neutrally, simply by commenting out the `late()` event that changes the fitness of matching hosts and parasites:



Here there is clearly no “chasing” of the host species by the parasite species; each species is just doing a random walk in allele frequencies. The qualitative difference between the neutral model and the selective model is very clear, although one could also develop a statistical test of temporal correlation to show it more quantitatively.

As an aside: this model allows four different allelic values at the locus, following the biology of DNA, but often one might want the model to be biallelic to match analytical models that assume that (or perhaps one’s biological system is actually biallelic for some biological reason). It’s easy to modify this recipe to be biallelic – particularly since it compares nucleotide strings to assess matching, rather than mutation objects (so you don’t need to worry about separate mutational lineages or back-mutations). To do that, start out by ensuring that the ancestral nucleotide at initialization is one of the two possible nucleotides (A or T, say). Then ensure that new mutations are only to A or T as well; you could write a `mutation()` callback for each species that rejects all new mutations that are not mutations to either A or T, or you could use a custom mutation-rate matrix that only models A-to-T and T-to-A mutations, rather than using the Jukes–Cantor model as



this recipe does. Either way, if you constrain the model to only use A and T, it ought to work nicely as a biallelic model; making it triallelic would be similarly easy.

Many other ways of modeling the genetics of coevolutionary species interactions are possible, of course, but with this recipe and the previous one we have covered a lot of the basic choices – quantitative versus Mendelian, trait-based versus allele-based, nucleotide-based versus non-nucleotide-based, nonWF versus WF. Hopefully this methodological survey helps you to craft the genetic interactions you want for your biological system.

## 19.8 Within-host reproduction in a host–pathogen model

We have looked at host–parasitoid and host–parasite models in previous sections, but in these models the parasitoid or parasite was free-living, finding a new host in every generation. There is an alternative life history in which the dependent species actually colonizes a host and reproduces within it for multiple generations. This is particularly typical of pathogens, so we will call this section's recipe, which explores such dynamics, a host–pathogen model. Such a model could involve genetics, with local adaptation to a particular host occurring over multiple generations of the pathogen inside that host; this could also bring in complex tradeoffs between transmissibility and virulence, for example, as well as adaptation of the host to resist the pathogen. However, we will not model such genetics for simplicity; here we are primarily concerned with the question of how to construct a model of multiple generations of within-host reproduction of a pathogen, so we will keep other aspects of the model simple.

To begin with, here's the initialization code, and the termination event:

```
species all initialize() {
  initializeSLiMModelType("nonWF");

  defineConstant("K_MONKEY", 100);           // monkey carrying capacity
  defineConstant("F_MONKEY", 3.0);           // monkey fecundity
  defineConstant("G_MONKEY", 20);            // relative generation timescale
  defineConstant("P_TRANSMISSION", 0.0001);  // probability of transmission
  defineConstant("SUPPRESSION_μ", 30000);    // optimal size for suppression
  defineConstant("SUPPRESSION_σ", 10000);    // width for suppression
  defineConstant("SUPPRESSION_STRENGTH", 0.4); // strength of suppression
  defineConstant("DEATH", 100000);          // level for certain death
}
species monkey initialize() {
  initializeSpecies(tickModulo=G_MONKEY, avatar="🐵", color="tan3");
  initializeSLiMOptions(keepPedigrees=T);
  initializeSex("A");
  // pedigree IDs are used to identify host–pathogen matches
  // colors: blue == uninfected, yellow -> red == infected, black == dead
}
species pathogen initialize() {
  initializeSpecies(avatar="🦠", color="chartreuse3");
  // pathogen.tag is a counter of the next subpop ID to use
  // subpopulation.tag is the pedigree ID of the host for the subpop
}

ticks all 2000 late() { }
```

This is a nonWF model. That is convenient because we want to control the transmission of pathogens from host to host with some precision; we will use the `takeMigrants()` method to do that, which is only available in nonWF models. We also want to be able to kill hosts sometimes,

when they succumb to infection; we will use the `killIndividuals()` method to do that, which is again only available in nonWF models. This model also involves overlapping generations, which is... well, you know.

We set up various constants that govern the model's dynamics; the comments give some indication of their function, and we will discuss them further as they arise in the code. We have two species, monkey and pathogen. This model uses different timescales for the two species (last seen in section 19.2); for every monkey cycle we will model 20 pathogen cycles, as governed by the `G_MONKEY` parameter. The monkeys are also modeled as being sexual, and will reproduce through biparental mating, whereas the pathogens are not, and will reproduce clonally.

The only other thing to note is that, as explained in the comments above, this model keeps track of which pathogens are living in which hosts using the `pedigreeID` property of the monkeys. We pass `keepPedigrees=T` for the monkeys so that each monkey has a unique `pedigreeID` value. For each monkey host that is infected by the pathogen, there will exist a separate pathogen subpopulation containing all of the pathogens that are living in that host; if fifteen of the monkeys are infected, there will be fifteen pathogen subpopulations. For a given host with a particular `pedigreeID`, the corresponding pathogen subpopulation's `tag` value will be equal to the host's `pedigreeID`. This scheme allows us to find the host for a given pathogen (it is the host with a `pedigreeID` equal to the host's subpopulation's `tag` value), and to find the pathogens infecting any given monkey (they are the ones in the pathogen subpopulation with a `tag` value equal to the monkey's `pedigreeID`). It's a bit confusing at first, but we will soon see how it is used in practice, which should make things clearer.

Next, let's look at the initial population setup, in a `ticks all 1 early()` event:

```
ticks all 1 early() {
  monkey.addSubpop("p1", K_MONKEY);

  // choose initial hosts carrying the infection
  initial_hosts = p1.sampleIndividuals(5, replace=F);
  pathogen.tag = 3;

  for (initial_host in initial_hosts)
  {
    // make a pathogen subpop for the host
    pathogen_subpop = pathogen.addSubpop(pathogen.tag, 1);
    pathogen_subpop.tag = initial_host.pedigreeID;
    pathogen.tag = pathogen.tag + 1;
  }

  // log some basic output
  logfile = community.createLogFile("host_pathogen_log.txt", logInterval=1);
  logfile.addTick();
  logfile.addSubpopulationSize(p1);
  logfile.addPopulationSize(pathogen);
  logfile.addCustomColumn("host_count", "pathogen.subpopulations.size()");
  logfile.addMeanSDColumns("monkeyAge", "p1.individuals.age");
}
```

We make a single monkey subpopulation, `p1`, for all the monkeys, which starts at the monkey carrying capacity of `K_MONKEY`. Then we start an initial infection by choosing five monkeys at random (a size chosen because it makes early stochastic extinction of the pathogen relatively unlikely, but if you were interested in such establishment dynamics you could start with a single infected individual). Those monkeys each receive a single pathogen. This is done by creating a new pathogen subpopulation of size 1 for each target monkey, with a `tag` value for the

subpopulation equal to the target monkey's pedigreeID. This establishes the correspondence between the hosts and the pathogens infecting them. Each new pathogen subpopulation gets a new id – p3, p4, p5, etc. – based upon the value of the tag value on the pathogen species; that tag value gets incremented after each new subpopulation is created, providing unique identifier values.

Finally, we set up a LogFile object to log out some basic information so we can see what's going on; we'll see plots from this dataset below. Now let's look at some population management code, including extinction, reproduction, and population regulation:

```
ticks all 2: first() {
  if (p1.individualCount == 0)
    stop(monkey.avatar + " extinct");
  if (pathogen.subpopulations.size() == 0)
    stop(pathogen.avatar + " extinct");
}

species monkey reproduction(p1, "F") {
  // monkeys reproduce sexually, non-monogamously
  litterSize = rpois(1, F_MONKEY);

  for (i in seqLen(litterSize))
  {
    mate = subpop.sampleIndividuals(1, sex="M");
    subpop.addCrossed(individual, mate);
  }
}

species pathogen reproduction() {
  // the pathogen reproduces clonally
  subpop.addCloned(individual);
}

ticks monkey early() {
  // monkey population regulation
  p1.fitnessScaling = K_MONKEY / p1.individualCount;
}
```

The `first()` event just detects extinction of either species and, if it has occurred, stops with a diagnostic message. The message includes the avatar of the extinct species (🐵 or 🦠) just for fun.

The reproduction code is totally unsurprising: monkey females find a male mate and reproduce sexually (non-monogamously) with a mean fecundity of `F_MONKEY`, while the pathogens reproduce clonally. Keep in mind that the monkey species is active only every twentieth tick, due to its `tickModulo` value, so these two `reproduction()` callbacks are running on different timescales! As usual, the biology here is not intended to be realistic; the actual generation time and fecundity of monkeys, the effects of their social structure on mate choice, etc., are all neglected.

Finally, we have an `early()` event that implements population regulation for the monkeys. This works in the usual way that we have used in this manual, with `fitnessScaling` influenced by the population size relative to the carrying capacity of `K_MONKEY`. Note that this event is declared with `ticks monkey`, and thus will only execute every twentieth tick; if it were `ticks all` it would execute every tick, but in the ticks when the monkey species was inactive it wouldn't actually have any effect, since survival/mortality would not occur for monkey in those ticks.

All of the above, except the setup of the initial infections, was pretty routine. Now it's time to get into the meat of the recipe. There's a fair bit of code involved, but it really isn't as complicated as it looks. Basically, in every tick we run three `early()` events, one after the other, to manage

horizontal transmission of pathogens between hosts (the first event), disease outcomes of death or suppression (the second event), and coloring of monkeys by their infection level for visualization in SLiMgui (the third event). We'll discuss them with fairly broad brush strokes, since a line-by-line dissection would just be too long.

Here's the first one:

```
ticks all early() {
  // horizontal transmission
  if (p1.individualCount > 1)
  {
    pathogenSubpops = pathogen.subpopulations;
    allPathogens = pathogenSubpops.individuals;
    isTransmitted = (rbinom(allPathogens.size(), 1, P_TRANSMISSION) == 1);
    moving = allPathogens[isTransmitted];

    for (ind in moving)
    {
      // figure out which host we are in
      hostID = ind.subpopulation.tag;
      currentHost = monkey.individualsWithPedigreeIDs(hostID);

      // choose a different host and get its ID
      newHost = p1.sampleIndividuals(1, exclude=currentHost);
      newHostID = newHost.pedigreeID;

      // find/create a subpop for the new host and move to it
      newSubpop = pathogenSubpops[pathogenSubpops.tag == newHostID];

      if (newSubpop.size() == 0)
      {
        newSubpop = pathogen.addSubpop(pathogen.tag, 0);
        pathogen.tag = pathogen.tag + 1;
        newSubpop.tag = newHostID;

        // need to incorporate the new subpop in case it receives another
        pathogenSubpops = c(pathogenSubpops, newSubpop);
      }
      newSubpop.takeMigrants(ind);
    }
  }
}
```

For horizontal transmission to occur, we need at least two monkeys. Given that, we use `rbinom()` to decide whether each individual pathogen will be transmitted to a new host or not, with a constant probability of `P_TRANSMISSION`. For each pathogen that is being transmitted, we first figure out which host it is in, `currentHost`, by looking up the monkey with the right `pedigreeID`. Then we choose its new host, `newHost`, with `sampleIndividuals()`. That host might already contain pathogens; if so, a pathogen subpopulation will already exist for it, and we just call `takeMigrants()` to move the target pathogen over. If not – if the lookup of `newSubpop` returns an empty vector – we need to create a new pathogen subpopulation for `newHost`. That is done in the same way as it was for the initial infection, and then `takeMigrants()` moves the pathogen into the new subpopulation. Whenever we create a new pathogen subpopulation we add it to `pathogenSubpops`, so that that variable is up to date and the subpopulation lookup code continues to work correctly as the loop continues.

Next comes the event that handles disease outcomes:

```
ticks all early() {
    // disease outcomes: either suppression or mortality
    suppress_scale = SUPPRESSION_STRENGTH / dnorm(0.0, 0.0, SUPPRESSION_σ);

    for (pathogenSubpop in pathogen.subpopulations)
    {
        popsize = asFloat(pathogenSubpop.individualCount);
        P_supp = (dnorm(popsize, SUPPRESSION_μ, SUPPRESSION_σ) * suppress_scale);
        P_death = popsize / DEATH;

        if (runif(1) < P_supp)
        {
            // the infection is suppressed; host lives, pathogen dies
            pathogenSubpop.removeSubpopulation();
        }
        else if (runif(1) < P_death)
        {
            // the host has died; kill it and its pathogens
            host = monkey.individualsWithPedigreeIDs(pathogenSubpop.tag);
            monkey.killIndividuals(host);
            pathogenSubpop.removeSubpopulation();
        }
    }
}
```

This loops through the pathogen subpopulations one by one, and determines their fate – and the fate of the hosts that contain them. For each one, we calculate the probability of two possible outcomes: suppression (meaning that the host’s immune system kicks in and gets rid of the infection) and death (meaning the host dies, and its pathogen subpopulation with it).

The probability of suppression follows a Gaussian function, as modeled with `dnorm()`: the probability of suppression is highest when the infection is at an intermediate optimum level of `SUPPRESSION_μ`, with a width for the Gaussian function of `SUPPRESSION_σ` that governs how quickly the likelihood of suppression falls off with a non-optimum infection level. The overall strength of suppression is governed by `SUPPRESSION_STRENGTH`; this is the probability that a host with an infection at exactly the optimum level will suppress it. Overall, what we are trying to (loosely) model here is the idea that a very slight infection is unlikely to be suppressed because the host’s immune system has not yet had time to detect the infection and learn to combat it, while a very large infection has probably just gotten so out of control that even a full-scale immune response by the host is unlikely to clear it; at some intermediate level, suppression is most likely. Of course this is all just hand-waving, without any basis in empirical data, but it shows an approach that might be useful.

The probability of death is more straightforward. It is directly proportional to the size of the infection, and reaches a probability of 1.0 when the infection level is equal to the parameter `DEATH`. The bigger the infection, the more likely you are to die.

Of course all of this could be modeled with much more biological detail: the health level of the individual monkey, the strength of its immune response (including a faster and more vigorous response in monkeys that have been previously infected), recovery via a slow battle between immune system and pathogen rather than instantaneous suppression, and so forth. That is all just scripting detail, not worth getting into here; and of course, remember that you will never succeed in modeling every biological detail, so you should focus on the processes that are important to your question of interest and strive to keep other aspects of your model as simple as possible.

Anyhow, the rest of the event implements the decision for the given pathogen subpopulation. If its host suppressed it, we use `removeSubpopulation()` to remove the pathogen subpopulation from the model entirely. If, on the other hand, the host was killed by the pathogen, we look up the host and kill it by calling `killIndividuals()` for it, while also removing the pathogens that it was host to. We saw `killIndividuals()` before in sections 16.5 and 16.19; we will discuss it more below.

The third `early()` event is just for SLiMgui visualization purposes:

```
ticks all early() {
    // color monkeys by their infection level
    pathogenSubpops = pathogen.subpopulations;

    for (host in p1.individuals)
    {
        hostID = host.pedigreeID;
        pathogenSubpop = pathogenSubpops[pathogenSubpops.tag == hostID];

        if (pathogenSubpop.size() == 1)
        {
            pathogenCount = pathogenSubpop.individualCount;
            hue = max(0.0, 1.0 - pathogenCount / DEATH) * 0.15;
            host.color = rgb2color(hsv2rgb(c(hue, 1, 1)));
        }
        else
            host.color = "cornflowerblue";
    }
}
```

It loops through all the monkeys, looks up their corresponding pathogen subpopulation to determine the size of their infection, and colors them either blue (if uninfected) or a shade from yellow to red corresponding to increasing levels of infection (up to the threshold of `DEATH`).

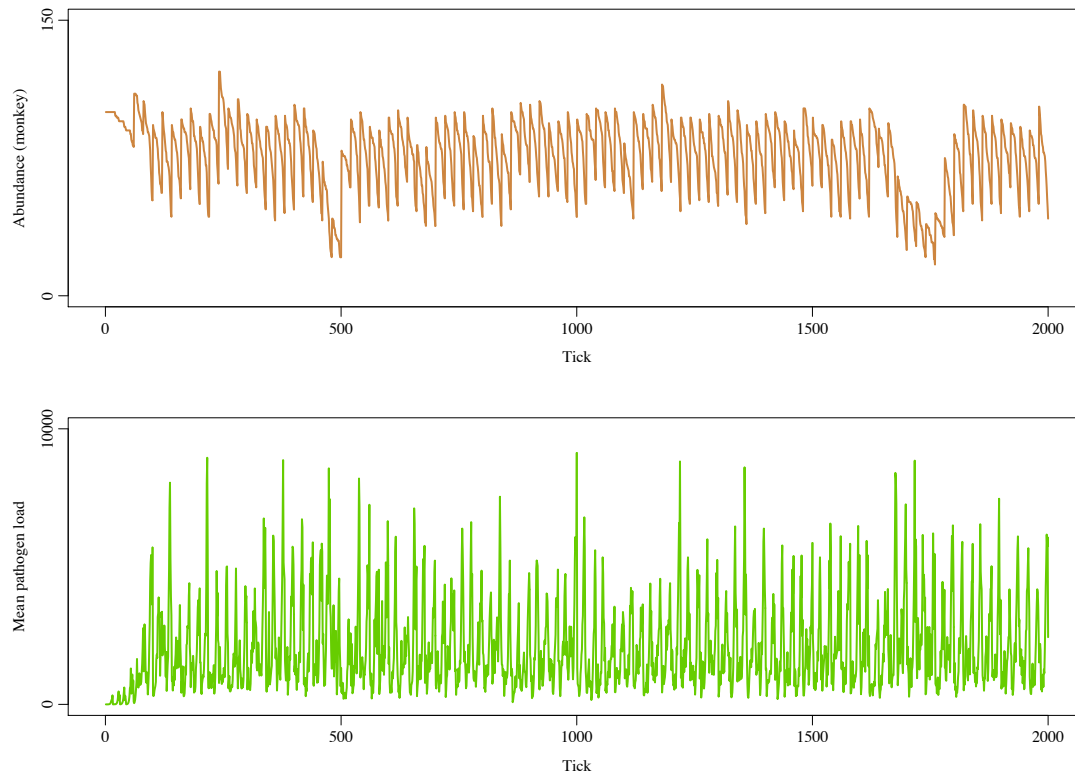
Besides dying from infection, monkeys also die of natural causes, as a result of the density-dependent `fitnessScaling` implemented earlier. When that happens, the pathogen subpopulation for the monkey that died needs to be removed. We do that with a `survival()` callback:

```
species monkey survival(p1) {
    // when a monkey dies, any pathogens in it die
    if (!surviving)
    {
        hostID = individual.pedigreeID;
        pathogenSubpops = pathogen.subpopulations;
        pathogenSubpop = pathogenSubpops[pathogenSubpops.tag == hostID];
        pathogenSubpop.removeSubpopulation();
    }
    return NULL;
}
```

This checks whether SLiM plans to kill the focal individual monkey, using the surviving pseudo-parameter; if so, it looks up the corresponding pathogen subpopulation and removes it. It ends by returning `NULL`, which tells SLiM to proceed with its planned action. We thus never change SLiM's survival decision for the monkeys; we just react to it, to update our bookkeeping.

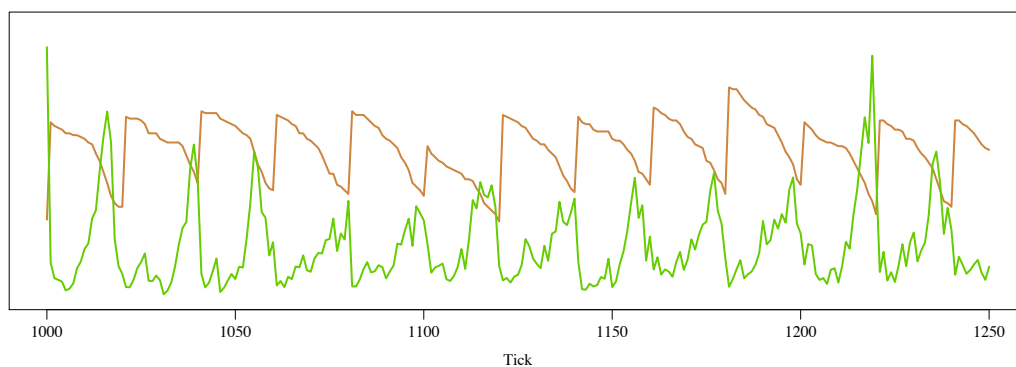
That's the whole model. What does it do? Sometimes it just goes extinct fairly quickly. When it doesn't, though, the dynamics it shows can be kind of interesting. Once the infection gets established the model's dynamics are generally pretty stable, although bad luck – stochasticity in the demographics, transmission, etc. – can bump the monkeys over the edge to extinction. Here are two plots generated from the data logged out by `LogFile`:






On the top is the abundance of monkeys through time. The cycling here is not a product of the interspecies interaction, exactly; it is a consequence of the monkey's life history, in which (in this model) they reproduce in a synchronized fashion every 20 ticks and then, if they exceed their carrying capacity, get cut back by density-dependent fitness. The top of each sawtooth is the moment after reproduction, when the population is at its peak; then deaths due to infection whittle away the population until the next reproduction event. There are two times, near ticks 500 and 1700, when the population crashes for no obvious reason.

On the bottom is the mean pathogen load per monkey (including uninfected monkeys). Interestingly, this cycles with the same periodicity even though it is pathogen load *per monkey*; the infection load is cycling in synchrony with the monkey's life history dynamics. To examine this further, we can zoom in on a representative period of time and overplot the data (the y-axis range for each line is the same as in the plots above):



What we can see here is that the pathogen load spikes, and the mortality rate of the monkeys is increasing, a little before each monkey reproduction event. The pathogen load usually drops a little bit before the monkey reproduction event; this seems to be due to a tranche of the most heavily-infected monkeys dying. The monkeys then reproduce, which generates a new batch of juveniles that are uninfected, further decreasing the mean pathogen load.

It seems like there are interesting questions to investigate here. One obvious question is: why do this model's dynamics tend to be stable, with neither the monkeys nor the pathogens going extinct unless a particularly unlucky set of chance events occurs, even though the model is not stabilized by spatiality as we saw in section 19.5? I think this *might* be due to the Gaussian function that governs the suppression of infection in hosts. The fact that suppression is unlikely when at low levels gives a helping hand to the pathogen when its abundance is low, preventing the monkeys from eradicating it, while the per-pathogen probability of transmission remains constant; this encourages the pathogen to survive and spread to new hosts. When the pathogen's abundance is high, on the other hand, the Gaussian function's shape tends to force the pathogen even higher, until all of the hosts with high levels of infection are rapidly killed off and the pathogen's abundance crashes again, leaving behind monkeys (and soon, juveniles) that have little or no pathogen load. But that is just a tentative hypothesis; lots of work would be needed to confirm this mechanistic explanation!

More importantly, the take-home point here is that the model appears to be working; you can observe in SLiMgui that infected monkeys change color according to their infection load, and you can examine the dynamics of the pathogen subpopulations and the temporary monkey graveyard. There is one feature of SLiMgui we haven't seen before that is also helpful for verifying that the model is working as intended, given the complexity of the timing of the two species and the various events in the tick cycle. If you open the debugging output viewer with the button , you will see a tab named **Scheduling** that you can select. Here's an example of what it shows:

```
...
viability/survival: species pathogen
# tick 1980: species monkey INACTIVE, species pathogen active (cycle 1980)
event: ticks all 2: first() [line 27]
offspring generation: species pathogen
event: ticks all early() [line 78]
event: ticks all early() [line 113]
event: ticks all early() [line 138]
fitness recalculation: species pathogen
viability/survival: species pathogen
# tick 1981: species monkey active (cycle 100), species pathogen active (cycle 1981)
event: ticks all 2: first() [line 27]
offspring generation: species monkey
offspring generation: species pathogen
event: ticks monkey early() [line 74]
event: ticks all early() [line 78]
event: ticks all early() [line 113]
event: ticks all early() [line 138]
fitness recalculation: species monkey
fitness recalculation: species pathogen
viability/survival: species monkey
viability/survival: species pathogen
# tick 1982: species monkey INACTIVE, species pathogen active (cycle 1982)
event: ticks all 2: first() [line 27]
...
```

For every tick, it provides a summary of which species are active, what their cycle counters are, and then the sequence of events and other occurrences during the tick cycle. This allows us to verify that the overall design of the model is working as intended.

Finally, as promised, let's discuss the use of `killIndividuals()` above, because it's more tricky than it appears. The trickiness is caused by the different timescales of the two species and the way that infection kills monkeys. The monkey species is active only every twentieth tick, and that means that it executes its survival/mortality phase only every twentieth tick. On the other hand, the pathogen species multiplies in every tick – exponentially, no less. When the infection in a given monkey gets out of control, we want that monkey to die immediately. (If it didn't die until the next monkey cycle, it would be a sort of zombie, continuing to transmit pathogens to the other monkeys; we don't want that here, although for some diseases it might actually be kind of realistic.) We could set its `fitnessScaling` to `0.0`, but that wouldn't actually kill it until the next monkey cycle. What to do?

One possible solution would be to change the design of the model to make the monkeys active in every tick, and just decrease their fecundity by a factor of 20 to compensate, or set up their reproduction code to only run every twentieth cycle (while allowing mortality to occur in every cycle). That would be perfectly reasonable for some models; however, sometimes you really do want the modeled species to run on different timescales, so let's consider this solution to be unacceptable for our purposes here.

Initially, I solved this problem by creating a special subpopulation I called the “temporary graveyard”. I used `takeMigrants()` to move dead monkeys from `p1` into it, and there they would rest – perhaps not entirely in peace, but without spreading disease or otherwise participating in model dynamics – until monkey was next active, at which time they would be killed by a `survival()` callback. However, ecology-induced mortality is very common, so this felt like a general problem for which a simpler and more intuitive solution was needed; and so the `killIndividuals()` method was born.

You might wonder why I've explained all of this. Partly, it seems important to explain why using `fitnessScaling` would not work well in this recipe, due to the timescale difference between the species. Also, though, even though the “temporary graveyard” got replaced by `killIndividuals()` in the end, I feel like I should preserve the idea for posterity. Its strategy was reminiscent of the “cold storage” subpopulation used in section 16.22, which stored dead individuals in order to be able to continue to use their sperm, obtained before their death by females, to posthumously fertilize eggs. As section 16.22 discusses, the same strategy can be useful for modeling “resting eggs” in *Daphnia*, or segregating individuals into “reproductive” and “non-reproductive” groups that behave and interact differently, or many other purposes. The “temporary graveyard” was a nice demonstration of the flexibility of using subpopulations to partition individuals for model-specific purposes. That's a worthwhile lesson to keep in mind, even if this recipe has evolved towards a simpler design in the end.

## 20. Runtime control

In this chapter we'll look at topics related to what might be called "runtime control": manipulating the random number generator, saving simulation snapshots and data to files, executing "lambdas" as a way of making code dynamic and/or reusable, and debugging your models in SLiMgui. We will no longer be looking at complete recipes for models as we did in the previous chapters; instead, we will use shorter snippets of code to briefly explore a few topics likely to come up when you are developing models of your own.

### 20.1 The random number generator

We have encountered the (pseudo-)random number generator in various recipes, but it is worth focusing on it briefly since random numbers are quite important for SLiM simulations.

SLiM and Eidos primarily use a random number generator that is part of the GNU Scientific Library (GSL), the GSL's `taus2` generator. It is possible to change this by modifying Eidos's internal code, but it is not recommended since Eidos and SLiM rely heavily on various properties of this generator, such as the number of random bits it outputs per draw, and the independence of the bits within each draw. It is a good generator with a long period, and should be suitable for most purposes. Starting in SLiM 3.0, a 64-bit Mersenne Twister (MT64) generator is also used, for purposes where 64-bit random numbers are needed (since the `taus2` generator is only a 32-bit generator, but is significantly faster than the Mersenne Twister generator); the seeds of the two generators are synchronized, and the fact that there are two independent generators used under the hood should be essentially invisible to users of SLiM and Eidos; it can, for practical purposes, be considered a single generator.

The random number generator is seeded at the very beginning of each simulation run (each press of the Recycle button, in SLiMgui) using a combination of the current Unix process ID and the clock time. This is not a particularly robust way to seed the generator; if you are doing production runs of a model, you will generally want to ensure that each run uses a different seed value. Perhaps the simplest way to ensure this is to pass a seed value to SLiM on the command line via the `-seed` or `-s` option; for example:

```
slim -seed 12345 ./script.txt
```

If you write a Unix script (or an R script, or whatever) to launch all of your SLiM model runs, that script can use this command-line option to set up each model run with a distinct seed value.

Within the code for a model, it is also possible to manipulate the random number generator's seed value; the recipe in section 9.2 did this in order to re-run the model from the same starting point with different seeds, for example. The `getSeed()` function returns the last seed value set, and the `setSeed()` function sets a new seed value. Note that calling `setSeed(getSeed())` generally changes the state of the random number generator; the seed returned by `getSeed()` is the last seed value set to initiate a random sequence, but since random numbers may have been generated since that seed was set, the seed does not necessarily reflect the current state of the generator.

The recipe of section 9.2, and some other recipes in the cookbook, change the random number seed with a particular formula:

```
setSeed(rdunif(1, 0, asInteger(2^62) - 1));
```

This is a useful way to change to a new seed (allowing the subsequent run to be reproduced directly by starting at the saved simulation point with the same new seed value) while still preserving a dependency upon the original seed value. These recipes used to do `setSeed(getSeed() + 1)` instead, but that could be problematic if a set of replicate runs are done

using sequential initial seed values; the replicate run starting with seed 1 will graduate to successive seeds of 2, 3, etc., and will thus end up using exactly the same pseudorandom sequence as the replicate runs that started with (or graduated to) those higher seed values. Since the identical pseudorandom sequences would be used in conjunction with a different simulation state, it *might* not end up mattering; but there is no point in taking the risk of accidental correlation between runs, so it is best to follow this new strategy, which should avoid this issue. (Thanks to Matthew Hartfield for pointing this out.)

The `rdunif()` call generates a seed within the range of the MT64 generator (probably rather conservatively, with  $2^{62}$ ), aiming to avoid any repetition of sequences that might occur if a given seed happens to lead to drawing the very same seed again and thus repeating the same sequence over and over (thanks to Graham Gower for pointing this issue out). The `taus2` generator has only an unsigned 32-bit range for seeds; values outside this range still work, however (they are taken modulo  $2^{32}$ ). There is still a risk, then, that the `taus2` generator will exhibit a repetition of sequences, but for many models the involvement of the MT64 generator in some aspects of SLiM's dynamics will nevertheless kick the model out of its previous path. Seeding random number generators is a tricky business; if you do it, be very careful and check that your results are not distorted by such effects.

When running a SLiM model, the seed value set when the simulation is initialized is automatically printed to the output. For example, you might see:

```
// Initial random seed:
1455193095666
```

If a particular model run catches your interest and you wish to reproduce it, you can copy that initial seed value and add a statement to the beginning of your model's `initialize()` callback like:

```
setSeed(1455193095666);
```

If you recycle the model after adding such a line, you will see a different initial seed value printed; but after you step over the `initialize()` phase, you should see the model repeat its previous sequence, because the different initial seed value was replaced by the `setSeed()` call.

## 20.2 Defining constants on the command-line

It is common to want to run a model many times, perhaps varying some of the basic parameters that define the model, or perhaps just varying the random number seed in a predictable way in order to produce independent replicates of the simulation. To make this simpler, SLiM provides a command-line option, `-define` (or just `-d`) that allows you to specify definitions for Eidos constants that your script can then use. For example, consider this simple script:

```
initialize() {
    setSeed(seed);
    initializeMutationRate(mu);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(r);
}
1 early() {
    sim.addSubpop("p1", N);
}
2000 late() { sim.outputFixedMutations(); }
```

This is just a simple neutral drift model, as we have used for the basis of many recipes in this cookbook. However, it references four undefined variables: the random number seed `seed`, the mutation rate `mu`, the recombination rate `r`, and the population size `N`. Run “as is”, the script would therefore fail with an “undefined identifier” error.

But when invoking this model at the command line, these values can be defined as Eidos constants using the `-d` command-line option. Assuming that `slim` and the script `test_defines.txt` are at findable paths, this would be an example invocation of the model:

```
slim -d seed=7 -d mu=1e-7 -d r=1e-8 -d N=500 test_defines.txt
```

The values would then be defined and available to the script. (Note that setting the random number seed at the command line can also be achieved with the `-seed` command-line option; see section 20.1). With this mechanism, it is easy to set up a script that runs a large number of `slim` jobs on a computing cluster, for example, to produce replicated runs across a whole parameter space. A call to `catn()` at the beginning of your script could output the values for all of the constants defined externally, so that the output from each run of SLiM is marked with the parameter values that generated it.

Incidentally, one might wish to write the above model such that the final tick depends upon the defined value of `N`; it is common, for example, to run a neutral burn-in period for  $10N$  generations (although this widespread practice is not without problems). It would be nice to be able to write:

```
10*N late() { sim.outputFixedMutations(); }
```

Unfortunately, this does not work; the tick range for script blocks must use simple integer constants. (This is difficult to fix because the values of constants may not be known at the time the script is parsed, such as when they are defined with `defineConstant()`.) Instead, the standard “trick” is to define the script block with a symbol, like this:

```
s1 2000 late() { sim.outputFixedMutations(); }
```

The symbol `s1` now refers to this script block. Then the script block can easily be rescheduled to the desired tick (or tick range) using the `rescheduleScriptBlock()` method, as in this rewritten tick 1 `early()` event:

```
1 early() {
    sim.addSubpop("p1", N);
    sim.rescheduleScriptBlock(s1, start=10*N, end=10*N);
}
```

The `s1` event will now run in tick  $10N$  as desired. Note that the tick declared for `s1` – 2000, here – is no longer important, since `s1` will be rescheduled anyway. The only caveat is that the declared tick should be *after* the tick in which the rescheduling occurs, to ensure that `s1` does not get executed before the rescheduling takes place. See section 25.1 for further discussion on the declaration of event script blocks.

One might wish to supply definitions of constants in this manner when running at the command line, but have the model still run properly under SLiMgui, with default values for constants, rather than producing an “undefined identifier” error. Here is a version of the `initialize()` callback for the above model that achieves this:

```
initialize() {
    if (exists("slimgui"))
    {
        defineConstant("seed", 1);
        defineConstant("mu", 1e-7);
    }
}
```



```

        defineConstant("r", 1e-8);
        defineConstant("N", 1000);
    }

    setSeed(seed);
    initializeMutationRate(mu);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(r);
}

```

The `slimgui` object, which provides an Eidos interface to SLiMgui (see section 24.13), is defined only when running under SLiMgui, so the constants will be defined only in that case.

One could similarly use `exists()` to test for the existence of each constant, and define their values only if they are undefined; this would allow the model to run at the command line with no `-d` definitions supplied, using default values, but would allow those default values to be overridden with a `-d` command-line flag when desired. For example, this defines `seed` only if it has not already been supplied:

```

if (!exists("seed"))
    defineConstant("seed", 1);

```

Defined constants can be of type `logical`, `integer`, `float`, or `string`; defining `string` constants probably requires playing quoting games with your Un\*x shell, such as:

```
slim -d "foo='bar'" test.txt
```

The fact that Eidos strings such as `'bar'` can be enclosed in either single or double quotes comes in useful here. If the `-l[ong]` command-line option is supplied (turning on more verbose output from SLiM), the argument for each `-d[efine]` command-line option will be printed as it was received by SLiM after processing by the shell, making it somewhat simpler to diagnose quoting issues.

In fact, the values for defined constants can be any Eidos expression, and can even reference previously defined constants. For example, one may accomplish scaling of model parameters by the population size with an invocation such as:

```
slim -d N=1000 -d THETA=5 -d "mu=THETA/(4*N)" model.txt
```

Here the mutation rate `mu` is calculated from the population size `N` and the scaled model parameter `THETA` (note that time would also need to be rescaled, and that various other rescaling issues might need to be addressed; see section 5.5). Note that with such expressions, as with `string` literals, quoting is probably necessary to avoid issues with your Un\*x command shell, as shown above. Expressions of any kind are allowed, including calls to Eidos functions, and the result of the expression need not be a singleton. The random number generator will be initialized (with the supplied `-seed` value, if any; see section 20.1) before the command-line definition expressions are evaluated, so functions that rely upon random numbers will use values based upon the model's initial seed.

It is a good idea to use unique names for defined constants that do not collide with any symbols defined by Eidos or SLiM. Some such names will be flagged as errors; others, such as collisions with the names of pseudo-parameters that SLiM provides to callbacks, may just cause confusion. You might wish to employ a naming convention for your constants to avoid all possibility of collisions, such as using names that begin with `d_`.

See section 14.6 for an example of defining constants on the command line with ABC-MCMC parameter estimation.

### 20.3 Other command-line options

Previous sections have touched upon a few of the command-line options for SLiM. In particular, section 20.1 showed how to pass a seed for the random number generator using the `-s` or `-seed` option, and section 20.2 showed how to define Eidos constants on the command line with the `-d` or `-define` option. There are a few other command-line options supported by SLiM, which we will briefly summarize here.

First of all, there are options that are not related to running simulations. The `-v` or `-version` option prints out the version number of the slim executable you are using:

```
$ slim -v
SLiM version 3.2, built Nov  6 2018 09:41:03
```

The `-u` or `-usage` option causes SLiM to print out a summary of how it can be invoked on the command line (i.e., more or less the same information we're summarizing here), in case you forget an option and want a quick reminder.

The `-testEidos` or `-te` option makes SLiM run a self-test of the Eidos language interpreter. Similarly, the `-testSLiM` or `-ts` option runs a self-test of the SLiM core. Typically you would use these after building SLiM to verify that the built executable is functioning properly (see section 2.6):

```
$ slim -te
SUCCESS count: 5439
$ slim -ts
SUCCESS count: 68424
```

The success counts are not important, and will depend upon the version of SLiM being run; the important thing is that no failed tests are reported.

Then there are command-line options that influence a simulation run in some way. Setting the random number seed with the `-s` / `-seed` option, and defining Eidos constants at the command line with the `-d` / `-define` option, have already been discussed; several other options also exist.

The `-l` or `-long` option enables “long” output, which provides additional information about the run. At the moment this is primarily useful for getting information about mutation run usage (see section 21.4); other long output may be added in future. The `-l` option can actually take an option level argument, `-l <level>`; the default level is 1, and using `-l` with no argument sets the level to 2, requesting greater verbosity. Levels greater than 2 produce no additional output. You can also set the level to 0, with `-l 0`, which turns off all of SLiM's normal output (echoing configuration settings, etc.), so that only output explicitly generated by your script will be emitted.

The `-t` or `-time` option enables output of the total runtime of the simulation (see section 21.2). When this option is enabled, a final line will be printed showing the CPU usage of the process (in seconds), like:

```
// ***** CPU time used: 0.11412
```

The `-m` or `-mem` option similarly enables output about the memory usage of the simulation (see section 21.3). When this option is enabled, final lines will be printed showing the initial and peak memory usage of the simulation (in bytes, K, and MB), like:

```
// ***** Initial memory usage: 1069056 bytes (1044K, 1.01953MB)
// ***** Peak memory usage: 4325376 bytes (4224K, 4.125MB)
```

The `-M` or `-Memhist` option provides more extensive output regarding the memory usage of the process as it runs, in the form of a final dump of usage statistics encapsulated within R code that can be copied and pasted into an R interpreter to produce a plot of memory usage over time (see section 21.3). This is not likely to be useful to end users; it is mostly a tool for debugging SLiM itself.

The `-x` option disables some of SLiM's runtime checks for consistency and safety. It is not recommended for general use, since it may mean that error conditions are not caught and reported. However, it may occasionally be useful if one of SLiM's runtime checks is actually faulty and needs to be disabled (see section 21.3).

When running a simulation, the path to the SLiM script file is typically supplied at the end of the command line. After SLiM 3.2, this may be omitted if a script file is instead piped in to SLiM's standard input ("`stdin`", in Unix parlance). In other words, this invocation of SLiM:

```
$ slim ~/Desktop/foo.slim
```

may instead be written as:

```
$ cat ~/Desktop/foo.slim | slim
```

This allows the input script to be assembled dynamically (in Python, say) and passed to SLiM via `stdin`, without having to create a temporary script file on disk to pass to SLiM.

## 20.4 File input and output

Output generated by a simulation with `print()`, `cat()`, and similar output functions goes into SLiM's output stream, which (assuming you are running at the command line) can be redirected to a file to save the results of the model run. Often, however, you will want a model to explicitly write output to a file. Eidos provides a few simple functions for file input and output, as well as operators and functions for string manipulation, and SLiM adds to those capabilities with some model-specific output functions. It should be possible for you to use these facilities to achieve whatever sort of customized output you wish; it is trivial, for example, to output information about the current state of a model as a comma-separated value (CSV) file that can be read by R and other such software in order to perform further analysis.

The simplest way to output simulation state to a file is to use the `Species` method `outputFull()`, such as by calling:

```
sim.outputFull("~/model_output.txt");
```

This produces a population dump in a standard format that is compatible with the `Species` method `readFromPopulationFile()`; these two methods can thus be used to save and restore the population state at any point in time, as discussed further in section 9.2.

But suppose this standard output format is not suitable; instead, you want to save a CSV file with a list of all of the mutations currently active in the simulation, listing their positions and selection coefficients. A recipe to achieve that might look like:

```
lines = NULL;
for (mut in sim.mutations)
{
    mutLine = paste0(mut.position, ", ", mut.selectionCoeff, "\n");
    lines = c(lines, mutLine);
}
file = paste0(lines);
file = "position, selcoeff\n" + file;
```

```

if (!writeFile("~/out.txt", file))
  stop("Error writing file.");

```

This recipe assembles a vector of output lines, naturally called `lines`, starting with `NULL` and adding new lines with `c(lines, mutLine)`. A `for` loop is used to loop over all the mutations in the simulation, and for each mutation an output line is assembled using `paste0()`. Each line ends with a newline pasted on to the end with `"\n"`.

After the `for` loop, the next line uses `paste0()` to join all of the lines produced by the loop into a single string, and the following line prepends a header line with column names for the output. At this point, `file` contains the string to be written to a file in the filesystem. Writing it out is achieved simply by calling `writeFile()`, passing the filesystem path for the first parameter and the string to write as the second. The `writeFile()` function returns a logical value: `T` if the file was written successfully, or `F` if a filesystem error occurred. It is a good idea to check the return value, as shown here, so that you are aware if your model's output is not working as expected.

The recipe above will work fine, but the use of the `for` loop is quite inefficient and is not true Eidos style; Eidos is a vectorized language, and it is generally better to use vectorized solutions when possible. Similarly, assembling a long vector with a series of `c()` calls to add one element at a time is highly inefficient in Eidos, requiring memory allocation and bulk copying of data with each successive addition. A better recipe to achieve the same result, then, would be:

```

lines = sapply(sim.mutations, "paste0(applyValue.position, ', ',
                                applyValue.selectionCoeff, '\\n');"");
file = paste0(lines);
file = "position, selcoeff\n" + file;
if (!writeFile("~/out.txt", file))
  stop("Error writing file.");

```

The first statement (wrapped onto two lines in the code shown above, but entered as a single line of code without the newline in the middle of the string literal) uses the `sapply()` function to assemble a vector containing string output lines for each mutation in the simulation. Conceptually, this is similar to the `for` loop across `sim.mutations` in the previous example, with `applyValue` as the index variable for the loop, and executing the Eidos code within `sapply()`'s second parameter as the `for` loop's body. However, `sapply()` also collects the result of each iteration of the loop and assembles all of those results in a single vector, here assigned into the variable `lines`; it thus does the work that `lines = NULL` and `c(lines, mutLine)` did above, but *much* more efficiently. The `sapply()` function is immensely useful for bulk processing of data, and should be a part of every Eidos programmer's toolbox; see the Eidos manual for documentation.

Note that because we want the code executed by `sapply()` to paste a newline at the end of each line using the escape sequence `\n`, we have to escape the backslash; `\\n` in the original code becomes `\n` in the string literal representing the code run by `sapply()`, which becomes a newline in the string literal passed to `paste0()`. Similarly, the double-quoted parameters to `paste0()` in the first version of the recipe are now single-quoted, allowing the quotes to nest without escaping issues. Confusing, yes; section 20.5 below will show a different approach that may be clearer and easier.

## 20.5 Lambda execution

A few recipes have touched on the ability of Eidos to execute code that was dynamically assembled as a string. For example, section 9.5.3 added new script blocks to the running simulation to schedule future mutation events, and section 20.4 passed a string representing an executable code block to the `sapply()` function. Such dynamic execution can be very powerful.

The first step in this technique is simply to assemble a string containing the Eidos code you wish to execute. This is generally done with the `+` operator, which performs string concatenation when given at least one string operand, and the `paste()` function, which pastes together a vector of strings joined by a fixed separator string. Sometimes, as in section 20.4, the string can even just be a literal; this technique does not necessarily involve dynamically generated code. Usually the main difficulties in this step involve correct escaping of special characters, and the related issue of quoting and nested quotes. The fact that Eidos allows strings to be quoted with either single or double quotes can be helpful; see, for example, the way that section 20.4 avoids having to escape quote characters by using single quotes inside the double-quoted code string. Another useful technique for handling quoting and escaping difficulties is the so-called “here document” style of string literal in Eidos (a terrible term, which I did not invent), documented in the Eidos manual. For example, section 20.4 used a small snippet of code:

```
lines = sapply(sim.mutations, "paste0(applyValue.position, ', ',
                                applyValue.selectionCoeff, '\\n');"");
```

This could be rewritten with the “here document” style as:

```
lines = sapply(sim.mutations, <---
paste0(applyValue.position, ", ", applyValue.selectionCoeff, "\\n");
>>---);
```

Note that double quotes can now be used without any escaping issues, and that the newline escape sequence desired in the string literal can be given directly as `\n`, without the necessity of quoting it as `\\n` to prevent an actual newline character from being inserted in the string. It is worth getting used to this string literal style if you work with Eidos code as string literals.

In any case, once the code string in this example is assembled, it is passed to `sapply()`, and `sapply()` treats it as Eidos code – the string gets parsed and interpreted just like the code in your model’s script. The `sapply()` function is one of several functions in Eidos that executes a string as code in this manner; another is the `executeLambda()` function, which simply executes the string it is passed as code. This can be used, to some extent, in a similar way to how functions are used in many other programming languages, but with a more dynamic twist. For example, suppose we wanted to write code that performed an operation on pairs of operands – but we don’t know what operation we want to perform ahead of time; it might be `+`, `-`, `*`, `/`, or `^`. All we have is a string for the desired operation. Without `executeLambda()`, we would have to do:

```
[... code that sets up operand1, operand2, and operator somehow ...]

if (operator == "+")
  result = operand1 + operand2;
else if (operator == "-")
  result = operand1 - operand2;
else if (operator == "*")
  result = operand1 * operand2;
else if (operator == "/")
  result = operand1 / operand2;
else if (operator == "^")
  result = operand1 ^ operand2;
```

With `executeLambda()`, this becomes trivial:

```
[... code that sets up operand1, operand2, and operator somehow ...]

result = executeLambda(paste("operand1", operator, "operand2;"));
```

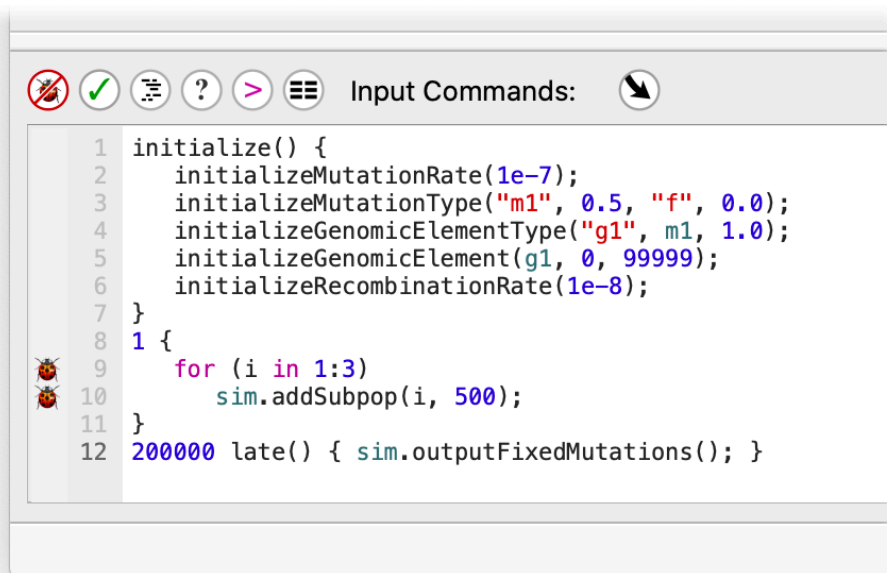
This is a rather contrived example, admittedly, but such situations do arise from time to time; it is worth being aware of this facility.

SLiM leverages this facility in Eidos to allow you to add new script blocks to a simulation dynamically, with code that can be assembled dynamically as a string and passed in to the SLiM engine. Both Eidos events and callbacks can be added, using the Community methods `registerFirstEvent()`, `registerEarlyEvent()`, and `registerLateEvent()`, and the Species methods `registerMateChoiceCallback()`, `registerModifyChildCallback()`, `registerRecombinationCallback()`, `registerMutationCallback()`, `registerMutationEffectCallback()`, `registerSurvivalCallback()`, and `registerReproductionCallback()`. It is also possible to create a mutation type that draws selection coefficients by calling a lambda that generates them; see the type "s" DFE in section 24.11, which is used in the recipe of section 13.3.

## 20.6 Debugging

One of the weaker points of the Eidos language is its facilities for debugging. There is no runtime debugger for Eidos – no ability to pause executing code at an arbitrary point in order to examine variables. However, there are still several ways to debug your code.

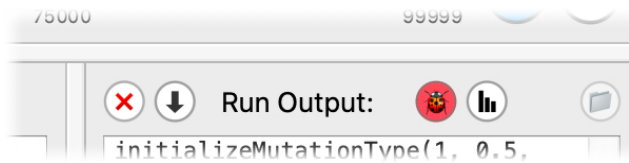
Perhaps the most important was added in SLiM 3.6: *debug points*. These are points in your code that you have flagged in SLiMgui. Here's a simple model with two debug points set:



The debug points, shown with little ladybug icons, were set simply by clicking in the left-hand margin of the script area, where the icons are shown. They request that SLiMgui generate additional debugging output regarding these lines of code. (Note that debug points are a feature of SLiMgui; there is no analogous feature available when running at the command line.)

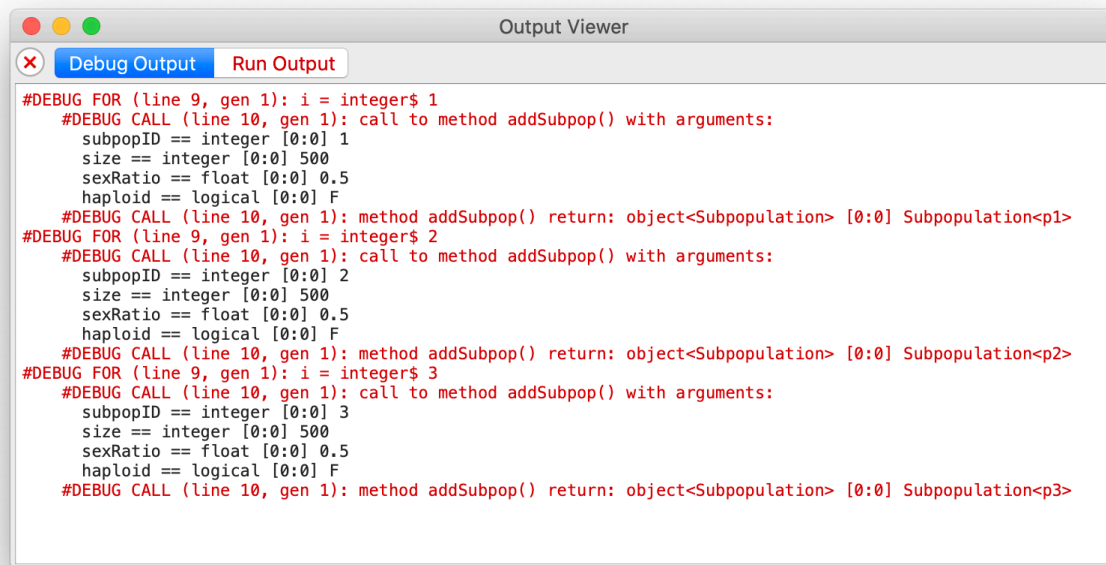
If we now run the model by clicking the Play button, debugging output will be generated. It doesn't go into the main output area on the right of SLiMgui's window, because we don't want to clutter up the model output with the debugging logs (which can be quite extensive). Instead, they go into a dedicated debugging output window in SLiMgui. That window is accessed through a button, marked with the same ladybug icon 🐞, just above the output area:





This button doesn't usually have a red background as shown here; but when new debugging output is generated, the button pulses red a couple of times to call your attention to the presence of new output. The screenshot above catches it in mid-pulse, after the debug points were triggered by execution of the script.

If we click on this button, the debugging output viewer opens:




First of all, notice that this window displays a tab view, with tab buttons across the top, to present all the different output streams from your simulation. Above you can see the output from debug points, under the Debug Output tab. The Run Output tab shows the same simulation output stream that is shown in the output area of the main SLiMgui window; if you prefer to have more space for scripting, you can use the splitter in the main window to hide that output area, and view the simulation's output stream in this output viewer instead. Each LogFile object in use in the simulation (see section 24.9) will be shown under its own separate tab here, presenting the logged data in tabular form, as seen in section 4.2.5. Finally, each file that the simulation writes to with the writeFile() function will also appear as its own tab in the output viewer. The output viewer window thus acts as the central clearinghouse for all simulation output in SLiMgui, and LogFile and writeFile() can be useful for debugging in SLiMgui since the output they generate is easily viewed here.


Looking at the screenshot above, you can see that the debug point on the for loop emits a #DEBUG FOR line for each iteration of the loop, stating the new value for the loop iterator variable, i. Similarly, the debug point on the call to sim.addSubpop() emits a #DEBUG CALL line each time that statement is executed, logging out the values of the parameters passed to addSubpop(), including the default value of 0.5 passed for the optional sexRatio parameter since we didn't supply a different value. Another #DEBUG CALL line gets output when each call to addSubpop() returns, giving the value of the object returned – in this case, the new Subpopulation object that


was created. The indentation of the output indicates the nesting of the code during execution; the call to `addSubpop()` occurs inside the `for` loop, so its output is indented inside of the `for` loop's output.

You can add debug points to almost every line of code in your script (except empty lines), and they will usually produce useful output for you when that line executes. A debug point on an assignment, like `x = 1:10;`, will log out a `#DEBUG ASSIGN` line giving the value that was assigned to `x`, for example. A debug point on the declaration of an Eidos user-defined function, or on the declaration of a SLiM event or callback, will produce a log each time that function, event, or callback is called. Debug points on most other types of statements will similarly log out something informative. Experiment with it to see what you get; it won't break anything, and indeed, debug points have no effect at all on the running SLiM model.

Beyond debug points, there are a couple of other debugging strategies you might try. The first is simply the ability to print to the output console at any point in your code. If you are wondering whether a calculation you have written produces the correct value, simply add a call to `print()` or `cat()` to see the value in the output. This is primitive, admittedly, but still powerful, and in fact is often quicker and simpler to produce an answer than a debugger would be. Note that beginning in SLiM 3.6 you can send output from `cat()`, `catn()`, and `print()` to the SLiMgui output viewer's Debug Output tab (shown in the screenshot above), instead of to the main Eidos output stream (i.e., to the output viewer's Run Output tab), by passing `error=T` to the function. When you are not running under SLiMgui, such output will go to the Unix `stderr` stream instead of the Unix `stdout` stream; you can redirect `stderr` output to a different file from the `stdout` output, if you wish, using the usual shell facilities for doing such things (see your manual for `bash`, `csh`, `zsh`, or whatever shell you are using in your terminal window).

The variable browser, opened with the Show Eidos Variable Browser button  in SLiMgui's main window, is another useful debugging facility. This is described in section 3.4, and is quite useful since it can browse into SLiM's state as well as your own variables. Unfortunately it can only show you the state of things in between ticks; there is no way to pause while inside the execution of a `mutationEffect()` callback, say, and browse the variables at that moment in time. Nevertheless, it is a powerful tool.

The console window, opened with the Show Eidos Console button  in SLiMgui's main window, provides a third debugging facility. This is described in section 3.3. In the console window, you can work with Eidos interactively, defining your own variables and executing your own code. All of SLiM's top-level variables are available to you in the console, so you can test out code that manipulates genomes and mutations and so forth. If you set up some dummy variables with the same names that SLiM uses in callbacks (such as `mut`, `homozygous`, `effect`, etc., for a `mutationEffect()` callback), you can test out your callback code interactively in the console. It's not quite the same as working in a debugger, but it's not bad. Note that all of the variables you define in the console are visible in the variable browser, too. You can even use the console window to change the state of the running SLiM simulation; code executed in the console is executed in the same Eidos context as the simulation to which the console is attached.

Often the help documentation, available through the Script Help button , is an overlooked tool for debugging. If your code isn't doing what you think it should do, often the best approach is to examine your underlying assumptions: is your mental model correct regarding all of the objects, methods, properties, functions, etc., used by your code? There is a reason why RTFM is often the first debugging advice given by experienced programmers. The help window is documented further in section 3.2.

A key step in debugging is often to find a reproducible case. A bug that crops up rarely and unpredictably can be terribly hard to track down; a bug that you can make happen over and over,

while examining it from different angles, is usually much more tractable. In SLiM, since so much of what happens is usually guided by the random number generator, using `setSeed()` to make a simulation follow the same path in each run is often an important step in debugging. See section 20.1 for more information on working with the random number generator.

Debugging is never easy; it is detective work, often reminiscent of the thought process advocated by Sherlock Holmes: eliminate hypotheses one by one until only one possibility remains, and that must be the answer. Except that actually it usually turns out to be some other possibility that one never even dreamed of – a problem that never seemed to happen to Sherlock Holmes. He must have been very good at debugging. Good luck!

## 21. Implementation and performance

Following the lead of the previous chapter, this chapter will also present shorter snippets of Eidos code, rather than complete models, to illustrate selected topics in model development. In this chapter, we will focus on topics related to technical considerations in model implementation and performance: speed, memory usage, and evaluation of performance.

### 21.1 Writing fast SLiM simulations

Evolutionary simulations are often limited not by ideas (there are always lots of interesting questions to explore), and not by the difficulty of writing the models to test those ideas (especially when using tools such as SLiM and Eidos that facilitate that process), but rather by time and processing power. Individual-based modeling is computationally intensive, and since it is usually not practical to spend years of computer time on a single problem, it often becomes necessary to limit the scope of one's investigation. Naturally this is undesirable, and so squeezing every last bit of speed out of one's simulation is beneficial; it may allow you to ask a broader question, or explore a larger parameter space, or use a larger population size.

When using a high-level modeling framework like SLiM, the most important thing in gaining high performance is to understand the design of the framework; often there are different ways to solve the same problem that provide vastly different performance. For example, SLiM allows you to define a `mutationEffect()` callback with an optional subpopulation constraint. Without using that optional constraint, you might write:

```
mutationEffect(m2) {  
    if (subpop == p2)  
        return 0.5;  
    else  
        return effect;  
}
```

Using the optional constraint feature, you would instead write:

```
mutationEffect(m2, p2) {  
    return 0.5;  
}
```

These are identical in their behavior, but the second version will perform orders of magnitude (literally) better than the first version. There are a bunch of reasons for this. The first version requires that the `mutationEffect()` callback be called for every mutation in every subpopulation, rather than just for the mutations in subpopulation `p2`, and the setup and teardown costs for running a callback are non-trivial, so that in itself has a large impact. Looking up the values of variables such as `effect` and `subpop` is also time-consuming. Doing the `(subpop == p2)` test in interpreted Eidos code is vastly slower than doing the same test internally in SLiM, since SLiM's core code is compiled and optimized C++. Even beyond all of those considerations, however, there is also the fact that the second callback above is specifically optimized by SLiM: a callback that does nothing except return a constant value gets short-circuited by SLiM's internal machinery completely. In that case, there is no setup and teardown for an Eidos interpreter to run the callback, because there is no interpreted execution of the callback's code at all; SLiM is smart enough to know to simply use the value `0.5` for the relative fitness in the cases where that callback would execute. The same optimization cannot be done (well, *is not done*) for the first version.

To some extent, these sorts of implementation details of the SLiM engine are private and should not be relied upon; they might change from version to version of SLiM. But an overall take-home

point is clear: whenever possible, write as little Eidos code as possible, and let the internals of SLiM and Eidos do as much work for you as possible.

There are lots of other examples of this principle. For example, you should use the vectorized facilities of Eidos whenever possible. To add a bunch of numbers, use one call to `sum()`, not a `for` loop performing sequential addition with the `+` operator. Similarly, to perform some repetitive operation on every element of a vector, consider using `sapply()` instead of a `for` loop. If the repetitive operation involves a conditional – if you’re considering writing a `for` loop with an `if` statement inside it – consider using `ifelse()` instead (or at least, again, using `sapply()`). In general, it is safe to say that whenever you start writing a `for` loop in Eidos you should stop and ask yourself, “Can this be vectorized instead?”

Another example is the problem of looking up mutations of a given type in SLiM. Often you want to get a list of all mutations in the simulation that are of type `m2` (for example). To do this, the natural Eidos idiom, as in R, would be to subset with a `logical` vector, like:

```
muts = sim.mutations[sim.mutations.mutationType == m2];
```

To execute this statement for a simulation with  $N$  active mutations, Eidos must (1) fetch the `mutations` property from `sim`, assembling a vector of size  $N$ , (2) fetch the `mutationType` property of that vector, constructing a new vector with size  $N$ , (3) do an `==` comparison with `m2`, constructing a `logical` vector of size  $N$ , (4) fetch the `mutations` property from `sim` a second time, again assembling a vector of size  $N$ , and (6) do a size- $N$  subset operation with the `[]` operator, using the two vectors constructed in the previous steps, to produce the final result. This is a huge amount of work, often to construct a result vector that might have just one element in it (since often the mutation type of interest involves a single introduced mutation). Precisely because this is such a common task, SLiM provides a shortcut:

```
muts = sim.mutationsOfType(m2);
```

Conceptually, this does the same thing as the previous version, but it does it in C++, inside SLiM’s internal code, and that allows it to be orders of magnitude faster. When SLiM offers you facilities like this, use them! On `Species`, another such method is `countOfMutationsOfType()`, which is even faster than taking the `size()` of the result of `mutationsOfType()` if you just need to know how many there are without getting the objects themselves. Similar methods exist on some other SLiM classes, such as `Genome`.

Eidos itself also has quite a few such time-saving functions, from standard fare like `min()` and `max()` to more esoteric functions like `cumProduct()`, `unique()`, and `sapply()`. Indeed, most of the functions defined by Eidos are technically unnecessary; the tasks they perform could be written in pure Eidos code without the use of any functions. Eidos nevertheless provides a large library of predefined functions, for convenience, clarity, reuse, and speed; learn and use this toolkit. If you find yourself writing out some operation in lengthy and inefficient Eidos code, it might be time to take a step back and ask whether some or all of that operation could be recast in terms of built-in Eidos functions. Some Eidos functions, such as `which()` and `sapply()`, take some effort to learn to use effectively, but the payoff can be large.

Another performance consideration is that defining and looking up variables in Eidos code is relatively slow. This is not an issue most of the time, but in code that gets executed a lot (a `mutationEffect()` callback on a common mutation type, for example) it can make a large difference. Using variables to represent temporary, intermediate computations can make code much easier to understand, but unfortunately it can also run much more slowly. For example, consider this code to compute the Euclidean distance between two points,  $(x_1, y_1)$  and  $(x_2, y_2)$ , and execute some code only if that distance is less than a constant threshold distance:

```

dx = x1 - x2;
dx_sq = dx * dx;
dy = y1 - y2;
dy_sq = dy * dy;
dist = sqrt(dx_sq + dy_sq);
if (dist < 4.0)
    ...

```

Don't do that unless you don't care at all how slowly it runs (which often is true – optimize only when you need to optimize). Instead, if you care about speed, do this:

```

if (sqrt((x1 - x2)^2 + (y1 - y2)^2) < 4.0)
    ...

```

Sometimes, of course, defining a temporary variable can improve performance, if the temporary value is used more than once. If the value of that Euclidean distance computation will be used more than once, then by all means assign it into a variable; looking up variable values may be slow, but it's not nearly as slow as performing a complex, multistep calculation. But at least avoid defining the temporary variables `dx`, `dx_sq`, `dy`, and `dy_sq` if those values are used only once.

Of course it also pays to actually think about the necessity of the calculations you're performing. In the above example, there is actually no point to calculating the square root; instead, you can just compare the square of the distance to the square of the threshold:

```

if ((x1 - x2)^2 + (y1 - y2)^2 < 16.0)
    ...

```

Learning to see such opportunities for optimization is largely a matter of patiently and methodically examining each line of your code to think about how steps might be folded together or eliminated entirely. A few hours of such effort might save you weeks or months of runtime.

These tips provide some general approaches and rules of thumb for improving simulation performance. The following sections will discuss more concrete tools that can be brought to bear.

## 21.2 Performance evaluation

Sometimes you want to know exactly how long a piece of code or a whole simulation takes to run; this can be useful when trying to optimize the performance of a model, for example, by comparing alternative formulations of the model quantitatively. SLiM and Eidos provide several tools for this purpose. (See also section 21.5, on profiling simulations in SLiMgui.)

First of all, to measure the total execution time of an entire simulation run on the command line, you can pass the command-line option `-time` or `-t` to SLiM and it will print a total time measurement to the output at the end of the run. When using the `-time` option, keep in mind that a given model may exhibit a wide variance in execution time depending upon the random number sequence used by a run; comparing runs using a fixed random number seed is therefore wise. See section 20.1 for details on using `-seed` or `setSeed()` to set up a reproducible model run.

If you want to measure the performance of Eidos code, the `clock()` function can be useful. This function returns the amount of CPU time used by the running process; the difference between the result of `clock()` at one point versus another point is thus the amount of CPU time that was used between the two points. Measuring a chunk of code is therefore trivial:

```

start = clock();
for (i in 1:100000)
    q = i;
cat("Elapsed: " + (clock() - start));

```



Measuring the performance of code that spans multiple code blocks (such as the full execution time of a SLiM model) is just a little more complicated, because the start variable would not be persistent for long enough. Using the `defineConstant()` function of Eidos to store the start time fixes that problem. So in the `initialize()` callback of your model, you could put this:

```
defineConstant("start", clock());
```

And then in a `late()` event in the final tick, you could write:

```
cat("Elapsed: " + (clock() - start));
```

The elapsed time printed is in seconds, but not of user (i.e. wall-clock) time; rather, it is in CPU time (the amount of time your computer's processor actually dedicated to running the model, which might be much less than the wall-clock time, particularly if your computer is busy doing other things as well).

For measuring the performance of just a small block of code, the `executeLambda()` function has a timing option that can also be useful. Just pass `T` as the second, optional parameter to `executeLambda()` and it will print a time measurement for the lambda. For example, suppose we wanted to measure the time it takes to execute this code:

```
mean(runif(10000000) * 10);
```

We could simply execute it inside an `executeLambda()` call, like so:

```
executeLambda("mean(runif(10000000) * 10);", T);
```

On my machine, this produces this output:

```
// ***** executeLambda() elapsed time: 1.07904
5.00037
```

The return value of the call is `5.00037`, and running the lambda took `1.07904` seconds. Supposing that to be unacceptable, we could try a simple optimization, changing the range of the `runif()` call, which draws random deviates from a uniform distribution, rather than reshaping the drawn values with multiplication afterwards. Since we're drawing ten million values, it is reasonable to wonder whether this might make a difference, so let's try it:

```
executeLambda("mean(runif(10000000, 0, 10));", T);
```

This produces this output:

```
// ***** executeLambda() elapsed time: 0.644127
4.99945
```

So incorporating the scaling into the `runif()` call sped up the code by about a third; not bad. The final result in the two cases is different, of course, because the random number generator is not in the same state; you could use `setSeed()` to do tests with identical random number sequences if you wished, but it should be irrelevant in this case.

Section 20.5 has further advice about how to use the `executeLambda()` function, including a discussion of the "here document" string literal style, which makes it much less painful to convert Eidos code into the form of a string literal that can be passed to `executeLambda()`.

Section 21.5 provides an overview of profiling simulations in SLiMgui, which provides a particularly useful way of evaluating simulation performance.

## 21.3 Memory usage considerations

Sometimes memory usage is a major concern when running individual-based simulations. SLiM has been engineered to keep its memory usage relatively low, but simulations involving large population sizes and large numbers of mutations can burn through megabytes and even gigabytes of memory. Reducing memory usage can be difficult, unless you are willing to change the parameters of your simulation, but it is at least possible to assess SLiM's memory usage.

Several tools are provided to assess SLiM's *total* memory usage, including SLiM's code, operating system overhead, and so forth. The first tool is the `-mem` or `-m` command-line option, which causes SLiM to keep track of the high-water mark of its memory usage and print out that high-water mark when the model finishes. The second tool is the `-Memhist` or `-M` command-line option, which causes SLiM to track and print the memory usage of the simulation over time, rather than just the high-water mark. A third tool is the Eidos function `usage()`, which returns the current or peak memory usage of the running process, in megabytes (MB).

To get more detail about SLiM's internal memory usage, the `outputUsage()` method of `Community` can be called at any time to get a breakdown of the current memory usage by the simulation (see section 24.3.2), and in SLiMgui the same information is available in profile reports (see section 21.6). These facilities do not assess SLiM's *total* memory usage; instead, they provide a detailed picture of the memory that SLiM itself allocates and has direct control over. For large simulations this should be the large majority of total memory usage, however, so this should not prove limiting in practice. These features are covered in more detail in section 21.6.

It is usually the case that the large majority of the memory used by SLiM is for the references kept by `Genome` objects to the `Mutation` objects that the genomes contain (by `MutationRun` objects, internally, beginning in SLiM 2.4, but those objects are not visible to the user of SLiM; see section 21.4). Genomes refer to mutations using 32-bit indexes (4 bytes), for memory efficiency (pointers, by comparison, are typically 64-bits, or 8 bytes, on modern systems). A single genome containing 1000 mutations thus takes about 4K of memory, a diploid individual with that mutational density would take ~8K, and a population of 1000 such individuals would take ~8MB (although if shared haplotypes were common that overhead might be reduced by `MutationRun`'s ability to share mutation references between genomes). It is easy to see that with very large population sizes or very large numbers of active mutations the memory overhead becomes quite large (particularly if genetic diversity is high so that haplotype sharing is minimal). This overhead is more or less unavoidable, since a genome simply must keep a list of the mutations it contains; there would be possible ways to compress the memory footprint of that list, but such schemes would inevitably make SLiM much slower. If your simulation is taking too much memory, you typically really have only a few options: (1) make your population size and/or chromosome size smaller, (2) change your model to have a lower mean number of active mutations per genome (modeling fewer background neutral mutations, for example – or none at all, as tree-sequence recording can allow; see section 1.7), (3) change your model to have more haplotype sharing (with a lower recombination rate, for example), or (4) buy more memory.

Note that, interestingly, it is the references to the mutations that burn the memory, in most typical simulations, not the `Mutation` objects themselves. This is because in a typical simulation one mutation is often contained by a large number of individual genomes. A single `Mutation` object presently takes up 80 bytes on macOS (implementation details like this are of course subject to change). If a single reference to that `Mutation` object takes 4 bytes, as we saw above, then once 20 genomes contain that mutation, the memory footprint of the references is already as large as the footprint of the `Mutation` object itself. A mutation near fixation in a population of only 1000 diploid individuals would use the same 80 bytes for the `Mutation` object, but  $4 \times 1000 \times 2 = 8K$  bytes for the references to the object (again, potentially reduced by haplotype sharing through `MutationRun`). That footprint is so large that all other memory usage is typically irrelevant.

Beginning in SLiM 2.1, runtime checks of SLiM's memory usage are performed periodically, if and only if SLiM is running under a process memory limit (as indicated by the results of the `Un*x` function `getrlimit()`). Such memory limits are often enforced for jobs running on computing clusters, and exceeding the limits causes immediate termination of the process by the operating system. To make such memory overflow terminations easier to debug, SLiM will print a diagnostic message if its memory usage approaches within 10 MB of the limit, stating what SLiM was doing when the overflow occurred; if the operating system kills the process soon after that point, this diagnostic message may prove useful for determining the problem. This runtime checking should not add significant performance overhead to SLiM, but it can be disabled with the `-x` command-line flag if so desired. Note that some systems will report that there is no memory limit, even when a limit is actually in force, so this feature may or may not work on a given system.

## 21.4 Mutation runs and runtime optimization

Beginning in SLiM version 2.4, a change was made to SLiM's core engine. With this change, each genome in the simulation can be broken into multiple "mutation runs", each of which contains the mutations that occur within a given subsection of the genome. A simulation might use four mutation runs per genome, for example, in which case every genome is divided into four roughly equal-sized chunks – the mutation runs – that are stored separately. This is an internal implementation detail that is not visible to SLiM models in Eidos, and normally it is of no concern. However, in some circumstances an awareness of the existence of mutation runs can allow a model to be optimized to run faster than it otherwise would.

The purpose of mutation runs is to allow some operations performed by SLiM to be performed faster than they otherwise could be. For example, suppose a gamete needs to be produced by recombination of the two parental genomes, and a single recombination breakpoint has been drawn. Without mutation runs, generating the gamete would require copying all of the mutation pointers from the first parental genome up to the breakpoint, and then copying all of the mutation pointers from the second parental genome from the breakpoint onward; if a typical genome contains 1000 mutations, generating a gamete will require copying 1000 pointers. Now suppose the genomes are divided into 10 mutation runs each. The breakpoint occurs inside one of those 10 runs, and mutation pointers will need to be copied from the parental mutation runs for those; that will be about 100 pointer copies. But for the other nine runs – this is the crucial bit – only the pointer to the mutation run itself needs to be copied to the gamete, because the gamete can share the mutation runs with other genomes. That makes 109 pointer copies total – a big improvement on 1000. Similar gains can be realized in other parts of the SLiM core engine as well. Using mutation runs adds in some bookkeeping overhead, but it usually at least breaks even in terms of performance, and often it results in a substantial speedup. It also improves memory usage, since long runs of pointers to mutation objects can be shared among many genomes.

The tricky question is: how many mutation runs should be used? If too few are used, then most of the benefits evaporate. If too many are used, SLiM spends most of its time just handling the bookkeeping involved; if every genome contains 1000 mutation runs, there are now 1000 times more objects involved in the simulation than there were, with immense performance implications. Choosing the right number of mutation runs to use can thus be very important; the performance of the simulation can change dramatically depending upon this value. Unfortunately, there is no general way to make this choice; factors such as the chromosome length, mutation rate, and population size are important, but so are things like population dynamics, the type of selection being experienced, and the effects of custom Eidos events and callbacks in the SLiM script.

In order to manage this problem, SLiM 2.4 and later actually conducts little experiments continuously: it times every tick, while occasionally varying the number of mutation runs being

used. As it collects datasets, it compares those datasets against each other (using *t*-tests, in fact!) to determine the optimal number of mutation runs. These experiments take very little time, and run continuously in the background. This means that the end user of SLiM usually doesn't have to think about all of this; the optimal number of mutation runs is used most of the time, and simulations run a little bit (or a lot) faster with no effort on the part of the user. It also means that if simulation dynamics change partway through – perhaps a neutral burn-in ends and a regime of strong selection begins – SLiM will notice the changed dynamics and automatically adjust the number of mutation runs to be optimal in each part of the simulation.

So far, so good. However, all of these experiments do have a small effect on performance. This can be particularly true for models for which the optimal number of mutation runs is not clear or changes frequently, as well as models with a very short wall clock time per tick. Sometimes, telling SLiM to use a fixed number of mutation runs can be beneficial – but you need to know how many.

As an example, let's look at a very simple neutral model:

```
initialize() {
    initializeMutationRate(1e-5);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 1e5-1);
    initializeRecombinationRate(1e-5);
}
1 early() { sim.addSubpop("p1", 500); }
10000 late() { sim.outputFixedMutations(); }
```

In SLiM 2.3, this model takes approximately 205 seconds to run (on my machine). In SLiM 2.4, if a single mutation run is used, this model runs in approximately 77 seconds, thanks to other performance optimizations added to version 2.4. If 32 mutations are used in SLiM 2.4, on the other hand, it runs in approximately 24 seconds – more than three times faster! As it happens, 32 is the optimum for this model, for most of its runtime (a smaller number of runs is better early on, when neutral diversity is still building). Actually, the optimum might be different in a different hardware/software environment (things like how much processor cache memory is available can be very important), so to be precise, 32 is the optimum on my machine at the present moment.

Running the model in SLiM 2.4 with no mutation run count specified, and therefore allowing SLiM to perform its “experiments” continuously in the background, results in a runtime of 30 seconds, with 78.5% of the ticks in the simulation using 32 mutation runs (in one trial run; this will vary from run to run, even with the same random number seed, since it depends upon timing information). SLiM therefore does a reasonably good job of finding the optimum, but telling it explicitly to use 32 mutation runs rather than conducting experiments results in even better performance – about a 20% speedup.

So in practice, how could we arrive at this result? The first step is to run the model at the command line with a few extra command-line options:

```
$ ./slim -m -t -l -s 0 ~/neutral.slim
```

This command runs, at the \$ prompt in my Un\*x terminal, the `slim` executable in the current directory (`.`), executing the model named `neutral.slim` in my home directory (`~`). The “`-s 0`” option tells SLiM to use a random number seed of 0 (see section 20.1); I chose this so that all of my test runs use exactly the same modeled sequence of events, which you may or may not want to do as well. The “`-m`” flag turns on memory monitoring (see section 21.3), and the “`-t`” flag turns on timing of the overall runtime (see section 21.2); that is how I collected the timing information given above. Finally, the “`-l`” flag (short for “`-long`”) tells SLiM to output long (i.e., verbose)

output. Exactly what gets added to verbose output is version-specific (so try not to make assumptions about it, and probably don't use it in your production model runs), but one of the things added is information about the mutation run experiments conducted by SLiM. When the model finishes executing, among the verbose output is a snippet like this:

```
// Mutation run modal count: 32 (78.5% of cycles)
//
// It might (or might not) speed up your model to add a call to:
//
//     initializeSLiMOptions(mutationRuns=32);
//
// to your initialize() callback. The optimal value will change
// if your model changes. See the SLiM manual for more details.
```

This tells us what we want to know: 32 is the optimal mutation run count for this model, at least according to SLiM's internal "experiments". Again, that is on this hardware in the present software environment; if you plan to do your production runs on a computing cluster, for example, you should ascertain the optimal mutation run count on the cluster – ideally when it is busy running other tasks on its other cores – since the optimum there may be different than on your local machine. (Note that similar, and even more detailed, information on mutation run usage can be obtained in SLiMgui using its profiling feature; see section 21.5. However, this might or might not indicate the optimum number of mutation runs for command-line runs, since the runtime environment for SLiM is somewhat different when it is running inside SLiMgui. The above method using the `-l` command-line flag is therefore recommended.)

Knowing that 32 is the optimal mutation run count, we can now modify our model as the output above suggests by adding an `initializeSLiMOptions()` configuration call at the beginning of the `initialize()` callback that tells SLiM how many mutation runs to use (see section 24.1):

```
initializeSLiMOptions(mutationRuns=32);
```

This will yield the desired 20% speedup, because SLiM will no longer conduct mutation run experiments, and will instead simply use 32 mutation runs throughout the model's execution.

Note that in some cases this could actually make a model perform worse! Earlier, for example, we mentioned the possibility of a model involving a neutral burn-in period followed by a period of strong selection. The optimal number of mutation runs might be very different in those two phases of the model – 64 in the first half and 1 in the second half, say – but if SLiM is conducting its own mutation run experiments, it should be able to adjust to that fact. If you specify a fixed number of runs, however, then either the first half or the second half of the model might perform quite poorly. It is not possible to change the number of mutation runs dynamically in Eidos, at present, so in such cases the best option is to allow SLiM to run its experiments and do its runtime optimization. Such cases can be detected simply by looking at the total runtime of the model with and without a specified number of mutation runs; if specifying the number of runs makes the model run more slowly, then you probably shouldn't do it.

In summary, the ability to specify the mutation run count is an advanced feature, the correct use of which requires some experimentation and careful timing using the appropriate hardware and software. If done properly, however, it can produce performance gains of as much as 20%, or probably even more for some models. For very large models with long runtimes, it is therefore an important tool.



## 21.5 Profiling simulations in SLiMgui

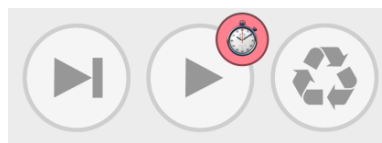
The previous sections have provided some pointers and tips regarding measuring and optimizing the memory usage and performance of a SLiM model. SLiMgui provides a particularly useful performance tool, called *profiling*, beginning in SLiM version 2.4.

For the purposes of this section, we will reuse the recipe from section 12.4, which shows how to use a `modifyChild()` callback to disable incidental selfing in hermaphroditic models (defined as occurring when the same individual happens to be chosen for both the first and second parent in a biparental mating event). Note that this recipe has been deprecated, since there is now a configuration flag for SLiM that disables incidental selfing more easily and efficiently (see section 12.4). Nevertheless, this recipe still works, and is a good one for our purposes here:

```
initialize() {
    initializeMutationRate(1e-7);
    initializeMutationType("m1", 0.5, "f", 0.0);
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 99999);
    initializeRecombinationRate(1e-8);
}
1 early() { sim.addSubpop("p1", 500); }
modifyChild()
{
    // prevent hermaphroditic selfing
    if (parent1 == parent2)
        return F;
    return T;
}
10000 late() { sim.outputFixedMutations(); }
```

This is a very simple neutral simulation; the only twist is the `modifyChild()` callback that checks whether the two parents of the focal offspring are the same, and if so, rejects the offspring by returning F. The recipe has been changed here to run for 10000 ticks, to provide more accurate measurements that are more focused on the steady state of the model rather than its initialization.

To profile this model in SLiMgui, simply click the profiling button that is overlaid with the play button in the SLiMgui main window (shown here in red, as it appears during profiling):



This will play the simulation forward from the current tick, just as the Play button would; the only difference is that SLiMgui will tabulate performance information about the simulation while it is running. You can start profiling at any point in a simulation run; for example, you can play up to the tick in which a critical section of your model begins, and then profile from there onward. Similarly, you can stop profiling at any time by clicking the profiling button again; you do not need to wait for the model run to complete. When the current profiling operation completes, a new window opens that shows a profile report. This report contains several sections; we will discuss them one by one.

The first section is the report header:



## Profile Report

Model: Untitled

Run start: 12/13/21, 12:19:36 AM

Run end: 12/13/21, 12:19:38 AM

Elapsed wall clock time: 2.27 s

Elapsed wall clock time inside SLiM core (corrected): 1.19 s

Elapsed CPU time inside SLiM core (uncorrected): 1.87 s

Elapsed generations: 10001 (including initialize)

Profile block external overhead: 21.86 ticks (2.186e-08 s)

Profile block internal lag: 18.15 ticks (1.815e-08 s)

The header contains general information: the title of the model, when the profiling run began and ended, and some overall timing and memory usage information. (i) The first “wall clock time” listed is the actual time spent running the model in SLiMgui; in this case, 2.27 seconds. (ii) The second wall clock time is time spent inside SLiM’s core code; this excludes time spent in SLiMgui, doing things like updating the user interface. It is also stated to be “corrected”; what this means is that an attempt has been made to subtract out time spent inside the profiling code itself, reading the system clock and tabulating profiling results. This number is thus SLiMgui’s best guess as to how long the model would take if it were run at the command line with the `sLim` command instead. Since this is the runtime that is generally of interest, it is used as a baseline in the profile report; later in the report, when timing percentages are reported, they are always percentages out of this corrected wall clock time. (iii) Third comes the elapsed CPU time inside the SLiM core; CPU time is time spent keeping the processor of the computer busy. This time can be quite different from the other times reported, as it is here. On the one hand, it excludes time spent in SLiMgui, so it is generally lower than the total elapsed wall clock time. On the other hand, it is uncorrected – it does not exclude time spent inside the profiling code itself – so it is typically longer than the corrected wall clock time. And then, too, CPU time is often different from wall clock times anyway, particularly if your machine is busy with other tasks that are also occupying the processor. (Incidentally, to obtain optimal profiling results it is a good idea to quit all other applications and run the profile when your machine is otherwise idle.) (iv) Next is shown the number of ticks over which the profile ran, including the `initialize()` phase. (v) Finally, two lines provide information about the measured overhead and lag of the profiling code; these numbers are used to produce the corrected wall clock time, and are not generally of interest to end users of SLiMgui. (Note that in SLiM 3.2 another two lines were added at the end, providing information about SLiM’s overall memory usage; these will be discussed in section 21.6.)

The next section is the “Cycle stage breakdown”:

### Generation stage breakdown

0.00 s ( 0.00%) : initialize() callback execution  
0.00 s ( 0.03%) : stage 0 – first() event execution  
0.00 s ( 0.04%) : stage 1 – early() event execution  
0.87 s (72.77%) : stage 2 – offspring generation  
0.09 s ( 7.14%) : stage 3 – bookkeeping (fixed mutation removal, etc.)  
0.01 s ( 1.24%) : stage 4 – generation swap  
0.00 s ( 0.05%) : stage 5 – late() event execution  
0.03 s ( 2.91%) : stage 6 – fitness calculation  
0.00 s ( 0.00%) : stage 7 – tree sequence auto-simplification

This is a tabulation of where SLiM is spending its time, broken down according to tick cycle stage. The fact that 72.77% of SLiM's time is being spent in offspring generation is a red flag; that tick cycle stage does often take quite a bit of time, but not typically *this* much. So this is something to note – depending upon the model it is not necessarily an indication of a problem, since different models do just show different profiles depending upon what parts of the SLiM engine they exercise more or less, but it is certainly an indication of where you ought to focus your optimization efforts.

Next is the “Callback type breakdown” section:

#### Callback type breakdown

```
0.00 s ( 0.00%) : initialize() callbacks
0.00 s ( 0.00%) : first() events
0.00 s ( 0.01%) : early() events
0.00 s ( 0.00%) : mateChoice() callbacks
0.00 s ( 0.00%) : recombination() callbacks
0.00 s ( 0.00%) : mutation() callbacks
0.46 s (38.30%) : modifyChild() callbacks
0.00 s ( 0.01%) : late() events
0.00 s ( 0.00%) : fitness() callbacks
0.00 s ( 0.00%) : fitness() callbacks (global)
0.00 s ( 0.00%) : interaction() callbacks
```

This section shows a tabulation of where SLiM is spending its time, broken down according to callback types. This does not add up to 100% because SLiM is spending about 60% of its time outside of callbacks, in the SLiM core engine. But it is spending 38.30% of its time inside `modifyChild()` callbacks – again, an indication of where to focus optimization efforts. You might also wonder why 38.30% is different from 72.77%. This is because the additional time, above 38.30%, is being spent by SLiM in offspring generation, but not inside `modifyChild()` callbacks. This time would be spent doing mutation generation and recombination, for example.

The next section is titled “Script block profiles (as a fraction of corrected wall clock time)”:

#### Script block profiles (as a fraction of corrected wall clock time)

```
0.00 s (0.01%):
1 { sim.addSubpop("p1", 500); }

0.19 s (15.60%):
modifyChild()
{
    // prevent hermaphroditic selfing
    if (parent1 == parent2)
        return F;
    return T;
}

0.00 s (0.01%):
10000 late() { sim.outputFixedMutations(); }
(blocks using < 0.01 s and < 0.01% of total wall clock time are not shown)
```

This shows the model's code, with color highlighting indicating where time is being spent. Colors range from white (essentially no time spent here) through yellow and up to orange and finally red (most of the simulation's time spent here). This section colors the code according to the

time spent as a fraction of the total corrected wall clock time. We can see that the `modifyChild()` callback is taking 15.60% of SLiM's time, and two parts of it are responsible: the test for the two parents being identical, and the returning of T to indicate that the child should be generated. The callback does occasionally return F, but that is so rare that that line is more or less white. Again you might wonder: why is 15.60% different from 38.30%? The reason is that 15.60% is spent specifically on interpreting the lines of code inside the callback – inside the Eidos interpreter – whereas 38.30% is spent overall on calling the callback. Calling out to callbacks involves a certain amount of overhead in SLiM; the Eidos interpreter environment for the callback must be set up, variable values for it must be initialized, and so forth. The difference between 15.60% and 38.30% reflects all of this overhead. The overhead is very high in this case because the callback is so simple; the more complex a callback's code is, the less this constant overhead will matter. But in this case, the profile report is telling us that it is not so much the callback's code that is hurting us – it is the fact that we have a callback at all.

As the italic comment at the bottom of the section notes, only script blocks that take more than a certain amount of time are shown, since script blocks that take a tiny fraction of the total time are generally not of interest from an optimization perspective. The `10000 late()` event is listed here, but takes only 0.01% of total time, which is not interesting for optimization.

Following that section, we next have “Script block profiles (as a fraction of within-block wall clock time)”:

#### Script block profiles (as a fraction of within-block wall clock time)

```
0.00 s (0.01%):
1 { sim.addSubpop("p1", 500); }

0.19 s (15.60%):
modifyChild()
{
    // prevent hermaphroditic selfing
    if (parent1 == parent2)
        return F;
    return T;
}

0.00 s (0.01%):
10000 late() { sim.outputFixedMutations(); }
(blocks using < 0.01 s and < 0.01% of total wall clock time are not shown)
```

This section shows a similar view of the model's code, but here the colors are scaled to the time spent within each block. Even a block that takes up almost none of the total time, therefore, will show “hotspots” that may be close to red. Here the two hotspot lines we noticed before in the `modifyChild()` callback are colored more strongly, but the first line is an orange that is almost red, indicating that it takes more time than the second line in yellow. This can also be seen in the shades of yellow used in the previous section of the report, but it is much more subtle; the point of this section, with colors according to within-block time, is precisely this: to increase the visibility of the hotspot lines. However, this section is usually of less interest than the previous section, since the proportion of the total corrected wall clock time is what ultimately matters. Underlining that fact, note that the `10000 late()` event has a line that is colored brick red, since that line takes 100% of the time inside that event – but it still takes only 0.01% of total time, as we noted before, so it is irrelevant. Take care when interpreting this section.

Finally, the last section is “MutationRun usage”:

## MutationRun usage

89.50% of generations : 1 mutation runs per genome  
3.50% of generations : 2 mutation runs per genome  
3.00% of generations : 4 mutation runs per genome  
3.00% of generations : 8 mutation runs per genome  
1.00% of generations : 16 mutation runs per genome

100.00% of generations : regime 1 (no fitness callbacks)  
0.00% of generations : regime 2 (constant neutral fitness callbacks only)  
0.00% of generations : regime 3 (unpredictable fitness callbacks present)

140337780 mutations referenced, summed across all generations  
0 mutations considered potentially nonneutral  
100.00% of mutations excluded from fitness calculations  
195 maximum simultaneous mutations

14849000 mutation runs referenced, summed across all generations  
910641 unique mutation runs maintained among those  
0.00% of mutation run nonneutral caches rebuilt per generation  
93.87% of mutation runs shared among genomes

This section contains fairly technical information about some of SLiM's internal bookkeeping mechanisms: mutation runs and their caching of fitness-related information, optimization of fitness calculations for neutral or constant-fitness mutations, and so forth. Section 21.4 of this manual describes the basic idea behind mutation runs, and so may shed some light on the meaning of some aspects of this section. Mostly, however, this section of the profile is intended for internal use by the developers of SLiM (the royal we, in other words); if you send us a profile report from your simulation, this section will help us diagnose the situation. We will not discuss this section further here.

Overall, this report focuses our attention very clearly on the source of the performance problem with this model: the `modifyChild()` callback. Happily, as section 12.4 discusses, we can eliminate that callback now with a call to `initializeSLiMOptions(preventIncidentalSelfing=T)`, which sets a configuration option that causes SLiM to block incidental selfing internally instead. If we add that call to the model, remove the `modifyChild()` callback, and profile the model again, the header of the report now looks like this:

## Profile Report

Model: Untitled

Run start: 12/13/21, 12:47:47 AM

Run end: 12/13/21, 12:47:48 AM

Elapsed wall clock time: 1.09 s

Elapsed wall clock time inside SLiM core (corrected): 0.64 s

Elapsed CPU time inside SLiM core (uncorrected): 0.66 s

Elapsed generations: 10001 (including initialize)

Profile block external overhead: 17.47 ticks (1.747e-08 s)

Profile block internal lag: 14.59 ticks (1.459e-08 s)

The total runtime is much shorter now – 0.64 seconds of corrected wall clock time inside the SLiM core, instead of 1.19 seconds! If we look at the tick cycle stage breakdown, it now shows a somewhat healthier mix of time spent in various cycle stages:

#### Generation stage breakdown

```
0.00 s ( 0.01%) : initialize() callback execution
0.00 s ( 0.07%) : stage 0 – first() event execution
0.00 s ( 0.07%) : stage 1 – early() event execution
0.33 s (52.32%) : stage 2 – offspring generation
0.09 s (13.42%) : stage 3 – bookkeeping (fixed mutation removal, etc.)
0.01 s ( 2.30%) : stage 4 – generation swap
0.00 s ( 0.09%) : stage 5 – late() event execution
0.03 s ( 4.67%) : stage 6 – fitness calculation
0.00 s ( 0.00%) : stage 7 – tree sequence auto-simplification
```

Offspring generation is now 52.32% instead of 72.77%, so SLiM is busier doing other things now; and even more encouraging, the total time spent in offspring generation is now 0.33 seconds instead of 0.87 seconds, so we have shaved off quite a bit of the actual time spent.

The callback type breakdown shows we are now spending very little time inside callbacks:

#### Callback type breakdown

```
0.00 s (0.01%) : initialize() callbacks
0.00 s (0.00%) : first() events
0.00 s (0.01%) : early() events
0.00 s (0.00%) : mateChoice() callbacks
0.00 s (0.00%) : recombination() callbacks
0.00 s (0.00%) : mutation() callbacks
0.00 s (0.00%) : modifyChild() callbacks
0.00 s (0.03%) : late() events
0.00 s (0.00%) : fitness() callbacks
0.00 s (0.00%) : fitness() callbacks (global)
0.00 s (0.00%) : interaction() callbacks
```

None is over 0.03% of the total time, so they are all basically irrelevant. What this says is that SLiM is no longer spending a significant amount of time inside the model's Eidos code; virtually all of the time is being spent inside the SLiM core. If you still wanted to make this model faster, you would therefore have to decrease SLiM's core workload, by doing things like reducing the population size, reducing the recombination rate or mutation rate, etc. If such revisions to the model are not possible, then the model might be running about as fast as it possibly can (assuming there are no design flaws in the model that are causing it to waste time).

The first script block profile section confirms this; there are no hotspots in the code at all:

#### Script block profiles (as a fraction of corrected wall clock time)

```
0.00 s (0.01%):
1 { sim.addSubpop("p1", 500); }

0.00 s (0.03%):
10000 late() { sim.outputFixedMutations(); }
(blocks using < 0.01 s and < 0.01% of total wall clock time are not shown)
```



The second script block profile section shows some hotspots:

**Script block profiles (as a fraction of within-block wall clock time)**

0.00 s (0.01%):

```
1 { sim.addSubpop("p1", 500); }
```

0.00 s (0.03%):

```
10000 late() { sim.outputFixedMutations(); }
```

*(blocks using < 0.01 s and < 0.01% of total wall clock time are not shown)*

Remember, however, that in this section the coloring indicates the time spent as a fraction of the within-block time. These lines might be taking 100% of the within-block time; indeed, they could hardly fail to do so, since each block is only one line long! But the fact remains that their fraction of the total runtime is inconsequential, so they should be ignored.

Particularly perspicacious readers may have noticed that if the original model took 1.19 seconds, 38.30% of which was dealing with `modifyChild()` callbacks, one might expect the modified model to take 0.73 seconds, but in fact it took only 0.64 seconds, so we got an even larger speedup than expected. Why is this?

One likely reason is probably that when callbacks are present, SLiM is often forced to take a slower code path, which adds runtime even beyond the overhead of handling the callbacks themselves. When `modifyChild()` callbacks are in force, for example, SLiM has to generate children in a random order, to eliminate possible order-dependency bugs that could otherwise crop up. When no such callbacks are active, SLiM can generate offspring in a fixed order – females before males, migrants before residents, etc. – which is substantially faster since it doesn't require randomization.

Another common reason is that the profiling itself is imprecise, due to various confounding factors, from measurement difficulties inside SLiMgui, to the simple fact that the exact same model, with the exact same random number seed, will be somewhat inconsistent in its runtime due to things likely how busy the operating system is, disk latency, memory cache performance, and a million other reasons. Measuring the runtime of software with precision is just hard. To compensate for this, you ought to do two things. First of all, you might take multiple profiles from your model if you are concerned about the precision of the profile results. Second, you ought to configure your model to run for a relatively long time. The example here runs for only a fraction of a second, which is silly; probably nobody would care about profiling and optimizing such a fast model, and the stochasticity in the runtime of the model will make the profile quite imprecise. If you can profile your model for several minutes, or even an hour or more, the profile will be more accurate since the imprecision will tend to average out. (Of course, if you modify your model to make it run longer, you have to be careful that that modification does not shift the profile of the model away from its usual behavior!)

All of this might prompt the question: “OK, I’ve got all this profile information, so what?” Well, in some cases it might allow you to see a way to modify your model to avoid the hotspot in question; if your profile showed that you were spending an inordinate amount of time doing addition in a `for` loop, for example, you might question why you are doing so much addition, and whether you could use `sum()` instead – or whether perhaps you don’t need to do all that addition at all. Eidos also provides quite a few functions that can perform various tasks very quickly; using functions like `unique()`, `setSymmetricDifference()`, `which()`, and so forth can lead to much faster code than attempting to code such logic in Eidos by hand.

In other cases, it will allow you to start a conversation with us about the performance problem you’re seeing; if you say “I’ve got this model and it’s spending 80% of its time inside the



`mateChoice()` callback running the second `for` loop, I've tried `x` and `y` and `z` but they didn't help much; what can I do about that?", that's a much better starting point than "My model is slow, help". If you are having to perform a multistep algorithm in Eidos for a task that is common, we might be able to add a new utility method to SLiM to perform this task for you with a single call. This can often result in a big speedup for the code in question, and it benefits all of SLiM's users for such utility methods to exist. The `sumOfMutationsOfType()` method, for example, was added in response to a performance-related question from a SLiM user, and is now available to speed up a wide variety of QTL-based models that need to calculate the additive effects of all mutations of a given type (see chapter 13).

## 21.6 Profiling memory usage in SLiMgui, or with `outputUsage()`

The previous section introduced SLiMgui's profiling feature, with an extended example showing how it can be useful for improving the runtime of a simulation. Beginning in SLiM 3.2, the profiling feature has been extended to include information about memory usage as well; we will discuss that extension in this section. Note that sections 21.3 and 21.4 cover some important ideas about SLiM's memory usage; you may wish to read those sections first.

First of all, when running SLiM 3.2 or later the header section of a profile report will be a bit longer due to a subsection added at the end. For a somewhat large spatial simulation with tree-sequence recording enabled, the report might show something like this:

```
Profile block internal lag: 22.81 ticks (2.281e-08 s)
Average generation SLiM memory use: 0.56 GB
Final generation SLiM memory use: 1.17 GB
```

These lines report statistics about SLiM's overall memory usage. The first line gives the *average* usage, across samples taken once per tick during the profiling period. The second line gives the sampled usage in the final tick profiled. Note that these memory usage samples are always taken at the end of each tick; if memory usage spikes within a tick but decreases again before the tick's end that will not be reflected in any of the profiling statistics discussed in this section (but *would* be captured by the overall usage statistics discussed in section 21.3). Also, importantly, these samples – which provide all of the information to be discussed in this section – reflect only memory allocated and directly controlled by SLiM. Memory can be used by other factors too: SLiM's own executable code takes up memory, the operating system has memory overhead of various types, the C++ standard library makes allocations that SLiM has no way to measure, etc. However, in large simulations where memory usage is a concern, the memory allocated and directly controlled by SLiM is typically the large majority of the total memory usage, so this caveat should not prove limiting in practice.

So here we discover that in an average tick SLiM was using 0.56 GB of memory, but that at the end of the simulation the usage was 1.17 GB in the final tick. That suggests that the memory usage was increasing over the course of the simulation, which is typical since genetic diversity often builds up over time. Since the peak memory usage is typically the main concern, the final tick's statistics may usually be of the most interest. One might also wish to profile only the last 100 ticks of a model, for example, in order to get averaged statistics that are not biased downward by the low memory usage toward the beginning of a run.

So far, so good; but these are only summary statistics. How does the memory usage by SLiM break down? What is all that memory actually being used for? A new section at the end of the profile report gives much more information:

### SLiM memory usage (average / final generation)

0.73 KB / 0.73 KB : Chromosome object  
56 bytes / 56 bytes : mutation rate maps  
32 bytes / 32 bytes : recombination rate maps

0.73 MB / 0.73 MB : Genome objects (7992.01 / 8000)  
315.43 KB / 1.95 MB : external MutationRun\* buffers  
17.98 KB / 18.00 KB : unused pool space  
0 bytes / 0 bytes : unused pool buffers

40 bytes / 40 bytes : GenomicElement objects (1.00 / 1)  
160 bytes / 160 bytes : GenomicElementType objects (1.00 / 1)

0.88 MB / 0.89 MB : Individual objects (3996.00 / 4000)  
21.73 KB / 21.75 KB : unused pool space

352 bytes / 352 bytes : InteractionType objects (2.00 / 2)  
187.31 KB / 187.50 KB : k-d trees  
31.22 KB / 31.25 KB : position caches  
11.96 MB / 12.01 MB : sparse arrays

30.90 MB / 37.99 MB : Mutation objects (404963.41 / 497883)  
2.30 MB / 4.00 MB : refcount buffer  
15.04 MB / 42.01 MB : unused pool space

1.19 MB / 5.87 MB : MutationRun objects (17333.44 / 85518)  
134.08 MB / 316.42 MB : external MutationIndex buffers  
0 bytes / 0 bytes : nonneutral mutation caches  
1.31 MB / 8.54 MB : unused pool space  
153.97 MB / 472.74 MB : unused pool buffers

0.62 KB / 0.62 KB : MutationType objects (2.00 / 2)

2.34 KB / 2.34 KB : SLiMSim object  
220.07 MB / 290.31 MB : tree-sequence tables

0.73 KB / 0.73 KB : Subpopulation objects (1.00 / 1)  
15.61 KB / 15.62 KB : fitness caches  
31.22 KB / 31.25 KB : parent tables  
223 bytes / 224 bytes : spatial maps  
510.86 KB / 511.89 KB : spatial map display (SLiMgui only)

0 bytes / 0 bytes : Substitution objects (0.00 / 0)

Eidos:  
136.00 KB / 136.00 KB : EidosASTNode pool  
32.00 KB / 32.00 KB : EidosSymbolTable pool  
391.44 KB / 392.00 KB : EidosValue pool

Each line in this section gives average and final-tick usage statistics, separated by a slash, as suggested by the header. Usage is in bytes, KB, MB, GB, or potentially TB; since the units are mixed they must be noted carefully when comparing numbers. To make it easier to pick out the areas of highest impact, the usage statistics are color-coded in a similar way to the coloring of timing statistics earlier in the profile report; white is the lowest proportional usage (inconsequential in the big picture), and then shades of yellow, orange, and finally red reflect increasing proportions of total memory usage.

The report is broken down by the object responsible for the usage, sorted alphabetically; Chromosome comes first, Substitution last, with a small section on memory usage by Eidos at the end. The first line of each of these subsections gives the memory usage for the objects themselves; for the Chromosome section it gives the memory used by the one Chromosome object present in the simulation, for example. Subsequent lines give additional memory usage, by those objects, for particular purposes. For Chromosome, for example, the memory used by mutation rate maps and

recombination rate maps is listed, as well as the usage by the ancestral nucleotide sequence in nucleotide-based models (not shown above because the screenshot is old). This memory is not part of the `Chromosome` object itself; it is allocated *by* the `Chromosome` object. Each line thus stands on its own; the usage reported by lines under a given object type is not included in the first line. In other words, the hierarchy here is a conceptual hierarchy, not a breakdown of totals into sub-totals and sub-sub-totals. For most object types the number of objects allocated is also reported in parentheses; for example, there were 404963.41 mutation objects allocated in an average tick, and 497883 allocated in the final tick.

This breakdown shows that the majority of memory is taken up by `MutationIndex` buffers allocated by `MutationRun`; some are being used by currently allocated `MutationRun` objects, while others are attached to currently unused `MutationRun` objects in a pool of reusable mutation runs kept by SLiM for speed. These `MutationIndex` buffers are the way that SLiM keeps track of which mutations are present in each genome; they are like pointers to `Mutation` objects, but more compact since they are 32-bit indexes instead of 64-bit pointers. The `Mutation` objects themselves take up far less space, and are colored just a light yellow here; this is unsurprising and typical, for reasons discussed further in section 21.3.

Quite a substantial amount of memory is also taken up by the tree-sequence recording tables kept by SLiM, since tree-sequence recording is enabled in this model (see section 1.7). Tree-sequence recording can take up quite a bit of memory, but that memory usage can often be controlled (at the price of longer runtimes) by controlling the frequency of simplification of the tree-sequence tables; see the `simplificationRatio` and `simplificationInterval` parameters to `initializeTreeSeq()` in section 24.1. The model profiled here includes neutral mutations even though tree-sequence recording is enabled, which is usually not desirable or necessary, so it is paying a hefty price in memory usage (for illustration purposes).

In nucleotide-based models the ancestral sequence (shown under `Chromosome`) can take quite a lot of memory, if the chromosome is long. The model shown here is not nucleotide-based, however, and so no memory is used for that.

Finally, a faint yellow tinge tells us that the sparse arrays kept by `InteractionType` are taking up a small but noticeable amount of memory. In SLiM 3 (when these screenshots were taken), these sparse arrays kept track of the distances and interaction strengths between individuals, and could use a lot of memory – even accounting for the large majority of SLiM’s memory usage, in models with many individuals interacting. In part to mitigate this memory overhead, SLiM 4 has switched to a different data structure, sparse vectors, for which the memory overhead is typically much lower. If this model were run under SLiM 4, the memory overhead of `InteractionType` would likely no longer be noticeable.

These memory usage statistics can, in some cases, provide a clear picture of how SLiM’s memory usage could be reduced. This report, for example, could serve as a reminder that we probably want to turn off neutral mutations in the model, since tree-sequence recording would allow us to overlay them after simulation has completed (see section 17.2). If most of the memory usage of a simulation is in the `MutationIndex` buffers kept by `MutationRun`, that would similarly suggest that the simulation might benefit from the use of tree-sequence recording. When that is not possible – when the mutations being simulated are non-neutral, for example – it may be necessary to reduce the scale of the model, in terms of population size, chromosome length, mutation rate, recombination rate, etc. Sections 5.5 and 21.3 have some further discussion of this.

## PART II: THE SLIM REFERENCE

## 22. SLiM architecture (WF models)

By default, SLiM uses a Wright–Fisher model of evolution, known in SLiM as a WF model (see section 1.6). This chapter will discuss the details of the tick cycle in WF models; chapter 23 provides the same type of discussion for nonWF models, a more advanced type of SLiM model.

Section 1.3 presented a summary of the tick cycle followed by SLiM within each tick in WF models. The figure shown in that section is reproduced at right, but with colors to indicate the way in which each tick cycle step is handled by SLiM in multispecies models (see below). In this chapter, we will examine each of these tick cycle stages in more detail, in order to provide a more complete specification of the internal mechanics of SLiM.

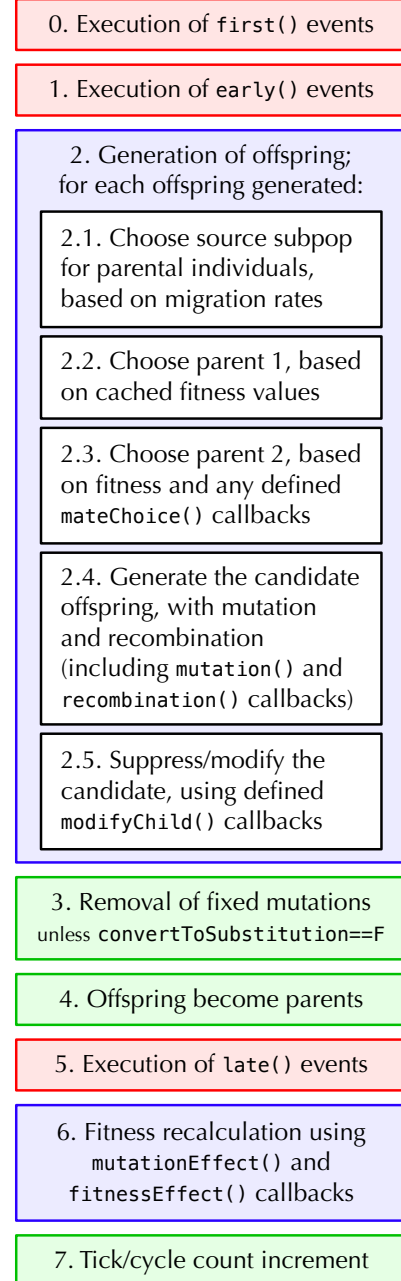
In single-species models the colors in the tick cycle diagram at right may be ignored, and this paragraph may be skipped. In multispecies models (see section 1.9 and chapter 19), however, more than one species might be active in the same tick. In this case, the tick cycle is executed for all active species in an interleaved fashion. Steps in red, which involve execution of Eidos events, execute those events in *definition order*, as the events are defined from top to bottom in the SLiM script, just as in single-species models; this is true even if some events have a `ticks` specifier (except that such events might not be active in the current tick). Steps in blue, which involve execution of Eidos callbacks, execute one species at a time, in *species declaration order*, as the species are declared from top to bottom in the SLiM script. Steps in green, which involve neither events nor callbacks, are executed in whatever order SLiM wishes; there are no dependencies between species in these steps, so the execution order is arbitrary and unspecified. Given those execution order rules for multispecies models, the remainder of this chapter will largely focus on the single-species case for simplicity.

### 22.0 Step 0: Execution of `first()` Eidos events

The very first thing that happens in each tick is that the `active` property of all script blocks is reset to `-1` at the beginning of the tick, activating all script blocks for the remainder of the tick unless they are explicitly deactivated again. This is followed by the execution of `first()` Eidos events defined by the user, if any. Details on how to specify an Eidos event are given in section 25.1, with some further details in section 25.11 regarding their scheduling and the `active` property.

The salient point here is primarily that these events occur first in each tick, prior to the generation of offspring, and prior even to `early()` events. This provides a good opportunity to set the `active` property of other script blocks, call `skipTick()` in multispecies models, and do other tasks related to planning the work for the tick. If you wish to execute an Eidos event after offspring generation has completed, you should use a `late()` event; see section 22.5. Since the details of this step depend entirely upon the event script you write, there is little more to say about it.

The WF model's tick cycle





## 22.1 Step 1: Execution of `early()` Eidos events

After `first()` events have executed, `early()` events are run next. Again, details regarding these events and their scheduling are in section 25.1 and 25.11. In general, `early()` events are more commonly used as a catch-all place to do things before generation of offspring occurs; `first()` events tend to be used rarely, for tasks that really need to happen *first* – before *any* other work is done for the tick. This distinction is fairly arbitrary, however; you may choose to divide tasks between these two event stages in whatever way you wish.

## 22.2 Step 2: Generation of offspring

This is the most complex step in SLiM’s architecture, and it is broken down into five sub-steps that are executed for each offspring generated, as shown in the tick cycle diagram. Processes involved in the generation of offspring include migration, mate choice, mutation, recombination, and the actual production of offspring individuals. Four different types of callbacks can be involved in modifying the offspring generation process in WF models.

### 22.2.1 The order of offspring generation

For each offspring generated, there are generally several decisions to be made: (1) is the offspring local, or if not, from which other subpopulation are its parents drawn, (2) is it male or female (in sexual simulations), and (3) is it produced by cloning, selfing, or biparental mating? In WF models in SLiM, the sex ratio specified for a subpopulation is deterministic; SLiM will produce that exact sex ratio in each tick (so as to avoid the possibility of extinction due to the chance production of a single-sex child generation). The other decisions are made stochastically; migration rates, selfing rates, and cloning rates are all probabilities, not deterministic ratios, and you can think of SLiM as rolling the dice to make these decisions for each offspring individual. This means that the sex ratio of a subpopulation does not fluctuate over time, but the fraction of offspring that are migrants, or clones, or selfed, will vary stochastically around the specified rates.

The order in which offspring are generated, with respect to these decisions, depends upon the details of your simulation. In the base case, SLiM produces offspring in deterministic tranches; for example, migrants before locals, and within that, males before females, and within *that*, cloned before selfed before biparental. The specifics of this ordering are not guaranteed; the main point is that you should not rely on the order of offspring being random. In particular, you should randomly select genomes when doing things like inserting new mutations, to overcome the possibility of order-dependency and bias (as shown in the recipe in section 9.1). If `mateChoice()`, `modifyChild()`, or `recombination()` callbacks are defined, SLiM switches its behavior to generate offspring in a randomized order, rather than in these deterministic tranches. This presents mating and offspring decisions to those callbacks in a random order, so that bias is not inadvertently introduced by callbacks.

SLiM is, for WF models, fundamentally a model of juvenile migration, not migration at the adult stage. This has several consequences for these decisions that are made for each offspring. First of all, if the offspring is a migrant produced biparentally, it should be noted that both parents will be drawn from the same source subpopulation; matings between parents in different subpopulations never occur in WF models in SLiM, since adults do not migrate. Second, it should be noted that the parental source subpopulation is “in charge” of most of the decisions made regarding the offspring, since the offspring is produced within that source subpopulation by the mating of parents in that subpopulation. In particular, the source subpopulation determines the cloning rate and selfing rate, as well as the `mateChoice()`, `modifyChild()`, `recombination()`, and `mutation()` callbacks used. The only exception is sex ratio; the sex ratio of the destination subpopulation governs the ratio of males to females produced in that subpopulation, regardless of the sex ratios



specified by the various source subpopulations contributing migrants. If these policies seem restrictive, you may want to use a nonWF model instead (see section 1.6).

### 22.2.2 *Mate choice*

Once the decisions outlined in the previous section have been made for a given offspring (parental subpopulation, sex, clonal/selfed/biparental), the next step in offspring generation is choosing the parent(s) for the offspring. The precise way in which this is done depends upon the type of offspring being produced:

(1) If the offspring is to be clonal, a single parent is drawn randomly, according to probabilities proportional to fitness, from the source subpopulation. Any `mateChoice()` callbacks defined are not called; there is presently no way to influence the choice of clonal parents except by modifying fitness values – with a `mutationEffect()` or `fitnessEffect()` callback, and/or by changing individual `fitnessScaling` values. Offspring generation proceeds along a different path in the clonal case, introducing mutations as usual (see below) but without recombination.

(2) If the offspring is to be selfed, a single parent is drawn according to fitness, as with cloning. That parent is then considered to be a forced choice for the second parent; `mateChoice()` callbacks are not used. Offspring generation proceeds thereafter as with biparental mating.

(3) If the offspring is to be the result of biparental mating, a first parent is drawn according to fitness; in sexual simulations the first parent will always be female, since WF models assume female choice. If no `mateChoice()` callbacks are defined, a second parent is then drawn according to fitness – in sexual simulations, always a male. If `mateChoice()` callbacks are defined, on the other hand, those callbacks will be called to determine mating weights for all eligible parents given the chosen first parent, as detailed in section 25.4, and a second parent will then be drawn with probabilities proportional to those mating weights. If the `mateChoice()` callbacks completely reject the first parent, offspring generation will go back to almost the beginning of the process; the source subpopulation will remain unchanged, as will the sex of the offspring to be produced, but the cloned/selfed/biparental decision, and the choice of the first parent, will be made over from scratch.

### 22.2.3 *Mutation and recombination*

Once a parent or parents have been successfully selected, as detailed in the previous subsection, a candidate offspring is generated from the chosen parent(s). As the first step in this process, the mutations to be introduced into each of the two offspring genomes are generated. The number of mutations is determined by the current mutation rate, using a draw from the appropriate Poisson distribution. The particular mutations are then generated one by one, each with a position drawn at random from within all of the defined genomic elements in the chromosome. Given that position, the identity of the genomic element and thus the controlling genomic element type is determined. Using the list of mutation types and associated probabilities for the genomic element type, a particular mutation type is chosen probabilistically. Finally, a selection coefficient is drawn from the chosen mutation type (see section 24.11), and a new `Mutation` object is constructed with that selection coefficient. If `mutation()` callbacks are defined, they will be called immediately after each new mutation is generated (assuming they apply to the source subpopulation generating the offspring), providing an opportunity for the new mutation to be modified or suppressed (see section 25.9). In this manner, all of the new mutations to be introduced into each of the two offspring genomes are generated.

In SLiM 3.5 and later, this procedure is modified slightly. The procedure described above, used in SLiM versions prior to 3.5, makes each mutational event completely independent of all others, and therefore it is possible for two new mutations to occur at the same base position during the generation of the same offspring genome. This turned out to cause some minor issues for things

like `mutation()` callbacks, and so in SLiM 3.5 mutational events are no longer completely independent. Specifically, all of the positions for new mutations in a given gamete are now drawn up-front by SLiM, and are then sorted and uniqued. After that, mutations are then generated one by one, from the uniqued positions, as described above. It is therefore no longer possible to get two new mutations at the same position during the generation of a single offspring genome. (This only applies to *new* mutations; it is still possible, by design, to get “stacked” mutations in SLiM when a new mutation stacks on top of an existing mutation, as discussed in section 1.5.3). This new mutational procedure means that sometimes – very rarely, for typical mutation rates – a mutational event will be rejected by the uniquing step, because its drawn position matches another drawn position in the same gamete-generation process. If SLiM didn’t compensate for this, the effective mutation rate would be less than the requested mutation rate; it would be the probability that a draw from  $\text{Poisson}(\mu)$  is greater than zero, which is  $1 - \exp(-\mu)$ . To prevent this discrepancy, SLiM 3.5 now adjusts the mutation rate upward, internally, so that the realized rate after position uniquing equals the requested rate. The adjusted rate, used internally and not visible to the running script, is  $-\log(1 - \mu)$ , computed as  $-\log_{10} p(-\mu)$ ; for typical mutation rates the difference is negligible (differing from  $\mu$  by on the order of  $\mu^2/2$ ), but for a very high mutation rate like, e.g.,  $\mu = 0.1$ , the rate used internally would be  $-\log_{10} p(-0.1) = 0.1053605$ , so the correction is significant at that point. These rate adjustments are performed internally for each segment of the supplied mutation rate map or hotspot map. Note that this implies that mutation rates of  $\mu \geq 1$  are no longer legal in SLiM, because they would entail an infinite adjusted rate.

In nucleotide-based models, the mutation process is somewhat different. The overall mutation rate is determined by the highest sequence-based rate expressed by any genomic element type’s mutation matrix; if CpG sites, for example, have a particularly high mutation rate, SLiM begins with the assumption that every site is a CpG site, and generates a number of mutations and then a vector of positions for new mutations based upon that assumption. (If a hotspot map is defined, a mutation rate map is used that is the multiplicative product of the highest sequence-based rate and the hotspot map.) Given the mutation positions, the genomic element and thus the controlling genomic element type are determined as usual. At this point, SLiM compares the highest sequence-based rate to the actual sequence-based rate that exists at the mutation’s location in the relevant parental genome. If the rates are not the same – if the parental genome’s sequence turns out not to be a CpG site, in our hypothetical example – then the proposed mutation will be rejected with the appropriate probability to produce the correct sequence-based rate of mutation (a strategy called *rejection sampling*). If the rates are the same, then the mutation is never rejected. Proposed mutations that are not rejected will be generated as usual (drawing the mutation type probabilistically from the genomic element type’s list, then drawing a selection coefficient for the mutation, as described above). In this way, SLiM produces the requested sequence-based mutation rate without having to recalculate the overall mutation rate and the mutation rate map every time the nucleotide sequence changes.

If the offspring is to be produced clonally, what follows is then relatively simple: conceptually, the parental genomes are replicated exactly in the offspring, and then the mutations are interleaved into the offspring genomes at their particular positions.

If the offspring is to be produced by selfing or biparentally (which are identical at this stage of the process), recombination is also involved, making the process somewhat more complex. The first offspring genome is produced via recombination between the two genomes of the first parent, and the second offspring genome is produced via recombination between the two genomes of the second parent (mimicking the process of meiosis to produce haploid gametes that merge to form a fertilized egg). For each offspring genome, the number of recombination breakpoints is drawn from a Poisson distribution based upon the overall recombination rate (computed internally by SLiM). The position of each breakpoint is then drawn, based upon the recombination ranges and

rates set on the chromosome. If the “crossover breakpoints” model is being used, these breakpoints are then used directly as crossover points; if the “double-stranded breaks (DSB)” model is being used, these breakpoints are double-stranded breaks that will become the basis of gene conversion tracts, crossovers, and heteroduplex mismatch repair (see section 1.5.6 for details).

If `recombination()` callbacks are defined, they are called at this point to allow them to modify the crossover breakpoints. Then, given the two genomes from the parent, the list of crossover breakpoints, and the set of mutations to be introduced, SLiM weaves together the final genome of the offspring, alternating between the two parental genomes as dictated by the recombination breakpoints, and introducing mutations at the chosen positions. If complex gene conversion tracts exist, resolution of any heteroduplex mismatches is the final step (see section 1.5.6).

By default, SLiM allows multiple mutations to exist at the same site in a single individual – “stacked” mutations, as they are called in SLiM (see section 1.5.3). This behavior is often desirable, but sometimes it is useful to prevent stacked mutations of a given mutation type. The stacking policy of a given mutation type can be changed using `MutationType`’s `mutationStackPolicy` and `mutationStackGroup` properties, as documented in section 24.11.1. The chosen stacking policy will be applied as mutations are overlaid during the process of weaving together the offspring genomes.

#### 22.2.4 Child modification

Once the two genomes of the candidate offspring have been generated, as described in the previous section, child modification by `modifyChild()` callbacks occurs (described in section 25.5), if any callbacks are active. These callbacks are called regardless of whether the offspring was the product of cloning, selfing, or biparental mating; all types of offspring may be modified.

These callbacks may modify the candidate offspring in any way desired. For our purposes here, the only complication arises with the possibility that a `modifyChild()` callback will suppress the candidate offspring altogether, rather than just modifying it. This can be thought of as representing juvenile mortality, if you wish; it could also represent postmating reproductive isolation, such as infertility or developmental inviability. In some cases, suppression of a candidate offspring can also be thought of as representing a type of mate choice.

When a candidate offspring is suppressed, generation of the offspring goes back almost to the beginning of the process, as with rejection of the first parent by a `mateChoice()` callback. In fact, in this case the process goes back even further; a new source subpopulation for the offspring is chosen, in addition to re-making the cloned/selfed/biparental decision and re-choosing the parents. The only aspect of the offspring generation process that is not re-decided in this case is the sex of the planned offspring individual; that remains fixed since the sex ratio is deterministic, not stochastic. Note that this means that if a `modifyChild()` callback suppresses a candidate offspring, the next time it is called the new candidate will always be the same sex as the previously suppressed candidate. Because of this, it is not possible for a `modifyChild()` callback to influence the sex ratio of a subpopulation in WF models; attempting to do so will produce an infinite loop.

#### 22.2.5 Child generation

Once a candidate offspring has been generated and modified, and was not suppressed by a `modifyChild()` callback, it is added to the target subpopulation. Note that the newly added child will not be visible as a member of the subpopulation until the point in the tick cycle when the child generation becomes the parental generation (see section 22.4). This prevents order-dependencies in which the first children generated might otherwise influence the remainder of the child generation process.

### 22.3 Step 3: Removal of fixed mutations

After all offspring have been generated for all subpopulations, SLiM performs bookkeeping regarding the mutations in the simulation. In particular, it scans through every genome in every individual in every subpopulation, and tallies up how many times each of the mutations in the simulation is actually present in the child generation.

The results from this scan are used to clean house. If a mutation is no longer referenced by any genome in the simulation, that mutation has been lost (whether due to selection or drift), and SLiM forgets about it. If, on the other hand, a mutation is now contained by every genome in the simulation, that mutation has fixed. In this case, SLiM normally creates a new `Substitution` object as a placeholder, to record the details of the fixed mutation, and then removes the mutation from the simulation. This is essentially an optimization for efficiency; if fixed mutations were not removed, a long-running simulation would accumulate ever more mutations needing to be tracked. In general the substitution is harmless; a fixed mutation cannot generally decrease in frequency, and generally no longer influences fitness in a WF model (since it is possessed by all individuals, and thus has an identical effect on the relative fitness of all individuals).

However, there are specific circumstances in which the removal of fixed mutations is not desirable. In particular, if the fixed mutation would continue exerting a varying effect on fitness among individuals (because of epistasis, for example), the substitution of the mutation would result in incorrect fitness values. Also, if the script for a simulation intends to remove the mutation from some genomes later in the simulation run (perhaps simulating a back-mutation) then it would be desirable to prevent the substitution of the mutation. To accommodate these possibilities, the `convertToSubstitution` property of a mutation type can be set to `F` to suppress substitution of mutations of that type; see section 24.11.1.

When nucleotide-based mutations fix and are converted to `Substitution` objects in a nucleotide-based model, the ancestral nucleotide sequence kept by the `Chromosome` object is updated with the new, fixed nucleotide.

### 22.4 Step 4: Offspring become parents

As mentioned in section 22.2.5, newly generated offspring are kept by SLiM as members of a child generation that is not visible to the model mechanics (or to your Eidos scripts), to avoid order-dependencies and other confusion. After mutations that are fixed in the child generation have been found and substituted, as described in the previous section, the child generation becomes the new parental generation, and the old parental generation is discarded. Generations are always non-overlapping in WF models.

### 22.5 Step 5: Execution of `late()` Eidos events

After the child generation is promoted to the parental generation, the next step is to execute `late()` Eidos events defined by the user, if any. Details on how to specify an Eidos event are given in section 25.1, with some further details in section 25.11 regarding their scheduling and the `active` property. Eidos `late()` events are most often used when output is being generated (since you typically want to output the state of the simulation at the end of a tick, not at the beginning), and when adding or removing mutations (since you want those changes to be reflected in the fitness values calculated for individuals, prior to the next offspring generation step). Changing of selection or dominance coefficients should also typically be done in a `late()` event, for the same reason. See sections 4.2.1, 9.1, and 9.6.1 for discussion and examples.

## 22.6 Step 6: Fitness value recalculation

After any active `late()` events have executed, the next step is to compute fitness values for the new parental generation, including the effects of any `mutationEffect()` and `fitnessEffect()` callbacks. Fitness values computed at the end of one tick are actually used during mating in the following tick; this is something to keep in mind if you are designing `mutationEffect()` or `fitnessEffect()` callbacks that you want to be active only across a specific tick range. The reason SLiM does this has to do mostly with running in SLiMgui; it is desirable that SLiMgui should show newly-calculated fitness values for the new parental generation when single-stepping through ticks. If fitness values were not calculated until the beginning of the next tick, they would not yet be available for display.

If `mutationEffect()` callbacks are not active for a given subpopulation, calculating the fitness of an individual is relatively straightforward. SLiM uses a model of multiplicative fitness between sites. An initial relative fitness  $w$  of `1.0` is assumed for the individual (unless `fitnessScaling` values have been set; in fact, the initial fitness  $w$  is the product of the individual's `fitnessScaling` property and its subpopulation's `fitnessScaling` property, but these properties are `1.0` by default). Then, each mutation possessed by the individual is evaluated as to whether it is present in just one of the individual's genomes (i.e., is heterozygous) or is present in both genomes (i.e., is homozygous). If a mutation is homozygous, the individual's relative fitness is updated as:

$$w = w * (1.0 + \text{selectionCoefficient}),$$

whereas if the mutation is heterozygous, the relative fitness is updated as:

$$w = w * (1.0 + \text{dominanceCoeff} * \text{selectionCoeff}).$$

where the dominance coefficient of the mutation is defined by the mutation's mutation type. If the new relative fitness is zero or less, the mutation just evaluated was lethal, and so the final relative fitness of the individual is `0.0`. Otherwise, SLiM proceeds with evaluating the next mutation, until all mutations have been evaluated to produce a final relative fitness value.

There is a wrinkle here to be explained, however: a mutation can be heterozygous in one of two different ways. If the individual being evaluated has two non-null genomes – two autosomes, say, or two X chromosomes – then the dominance coefficient used for this fitness calculation is the usual one, defined in `MutationType` as the `dominanceCoeff` property and supplied as a parameter to `initializeMutationType()`. This is the situation that will be relevant for 99% of models. If the individual being evaluated has a “null genome” for one of its genomes, on the other hand – the other 1% of cases – then the dominance coefficient used is defined in `MutationType` as the `haploidDominanceCoeff` property; this property is `1.0` by default, but may be changed by setting that property. This will arise in several situations: for example, if you are simulating the X chromosome (see `initializeSex()`, section 24.1) then XY males will have a null genome in place of the Y, so `haploidDominanceCoeff` will be used to evaluate the fitness effects of X-linked mutations in XY males (whereas in XX females, `dominanceCoeff` will be used). Similarly, if you are simulating the Y chromosome then XY males will have a null genome in place of the X, and so `haploidDominanceCoeff` will be used to evaluate the fitness effects of Y-linked mutations in XY males (whereas XX females will have two null genomes, and thus no mutations at all). This also arises in nonWF simulations of haploids (see section 16.13 and 16.14); typically (but not necessarily), in such models the second genome of every haploid individual is a null genome, and so `haploidDominanceCoeff` is used when evaluating the fitness effects of mutations in such haploids. (Haploids in WF models do not use null genomes, since that feature is not available to them, so `dominanceCoeff` would be used.) This may all seem overly complicated, but the point of it is to allow the fitness effect of mutations to be different depending upon whether they are truly



heterozygous (present in only one of two copies of a given gene) or hemizygous or haploid (present in the single copy of the gene that the individual possesses).

If `mutationEffect()` callbacks are defined (as described in section 25.2), this procedure is modified slightly. In this case, the multiplicative effect on relative fitness that would be produced by a given mutation is calculated, exactly as above, but instead of simply multiplying `w` by that fitness effect, SLiM calls out to applicable `mutationEffect()` callbacks to allow them to modify the relative fitness value. After callbacks, `w` is multiplied by the final fitness effect for the mutation. The `mutationEffect()` callback calculates the relative fitness of the mutation in the individual, whether the mutation is heterozygous or homozygous; if you want the calculated fitness value to be different in those two cases, then the `mutationEffect()` callback needs to explicitly take the heterozygosity of the mutation into account (which is part of the information provided to the callback by SLiM, so doing so is not difficult; see section 25.2).

In SLiM version 2.3 and later, it is also possible to define `fitnessEffect()` callbacks, which are applied exactly once to every individual without reference to a focal mutation or a particular mutation type (see section 25.3). The fitness values returned by `fitnessEffect()` callbacks are multiplied into the fitness value previously computed for the individual, as:

$$w = w * \text{fitnessEffect}.$$

The fitness effects of `fitnessEffect()` callbacks thus combine multiplicatively with all of the fitness effects of mutations, and multiple `fitnessEffect()` callbacks may be defined. Unlike other types of callbacks, the order in which `fitnessEffect()` callbacks are called is formally undefined, both relative to other `fitnessEffect()` callbacks and relative to `mutationEffect()` callbacks. Also note that `fitnessEffect()` callbacks might not be called at all for a given individual if that individual's fitness has already been determined, by previous callbacks or fitness effects, to be equal to (or less than) zero. Models should therefore be extremely cautious in making any assumptions regarding the timing or order in which `fitnessEffect()` callbacks will be executed, or whether they will be executed at all; `fitnessEffect()` callbacks that have external side effects, such as changing the active property of script blocks or defining Eidos constants, are not recommended.

In SLiM 3.0 and later, the `fitnessScaling` property may be set on the subpopulation or the individual (or both) to multiplicatively influence individual fitness values, as mentioned above. This is often a more efficient and simpler alternative to defining a `fitnessEffect()` callback.

By default (unless the `randomizeCallbacks` parameter to `initializeSLiMOptions()` is set to F), fitness calculations are done in a random order for all of the individuals within each subpopulation. Nevertheless, it is generally a good idea for `mutationEffect()` and `fitnessEffect()` callbacks to be written in such a way as to make each fitness computation independent of all others, and independent of the order in which they are done, unless you actually want such non-independent fitness effects. If `randomizeCallbacks` is F, individuals within each subpopulation will have their fitness evaluated in sequential order, which is more prone to order-dependency issues (but is slightly faster); see section 24.1.

See section 8.4 for further discussion of fitness in SLiM and SLiMgui.

## 22.7 Step 7: Tick/cycle count increment

The final step in each tick is that the cycle counter is incremented for each species that was active during the tick, and the tick counter kept by the `Community` is incremented. SLiM then checks whether the simulation is over; if there are no events or callbacks scheduled to execute in the new tick or any subsequent tick (not counting events and callbacks with no specified end tick), the simulation is deemed to be over, and execution halts.



## 23. SLiM architecture (nonWF models)

By default, SLiM uses a Wright–Fisher model of evolution, known in SLiM as a WF model – but an alternative non-Wright–Fisher, or nonWF, model type may be chosen instead (see section 1.6 and chapter 16). This chapter will discuss the details of the tick cycle in such models. Chapter 22 provides parallel discussion for WF models; to avoid a lot of duplicated verbiage, this chapter will assume familiarity with the WF tick cycle as described in chapter 22, and will make reference to that chapter rather than spelling out every detail a second time.

The figure shown at right is a summary of the tick cycle followed by SLiM within each tick in nonWF models. As in chapter 22, this tick cycle diagram uses colors to indicate the way in which each tick cycle step is handled by SLiM in multispecies models (see section 1.9 and chapter 19); these colors can be ignored when working with single-species models. For multispecies models, when more than one species is active in a given tick, red represents steps involving events, which are executed in *definition order*; blue represents steps involving callbacks, which are executed one species at a time in *species declaration order*; and green represents steps in which the ordering makes no difference and is thus arbitrary and undefined (see chapter 22 for further discussion). Given those execution order rules for multispecies models, the remainder of this chapter will largely focus on the single-species case. In this chapter, we will examine each of these tick cycle stages in more detail, in order to provide a more complete specification of the internal mechanics of SLiM.

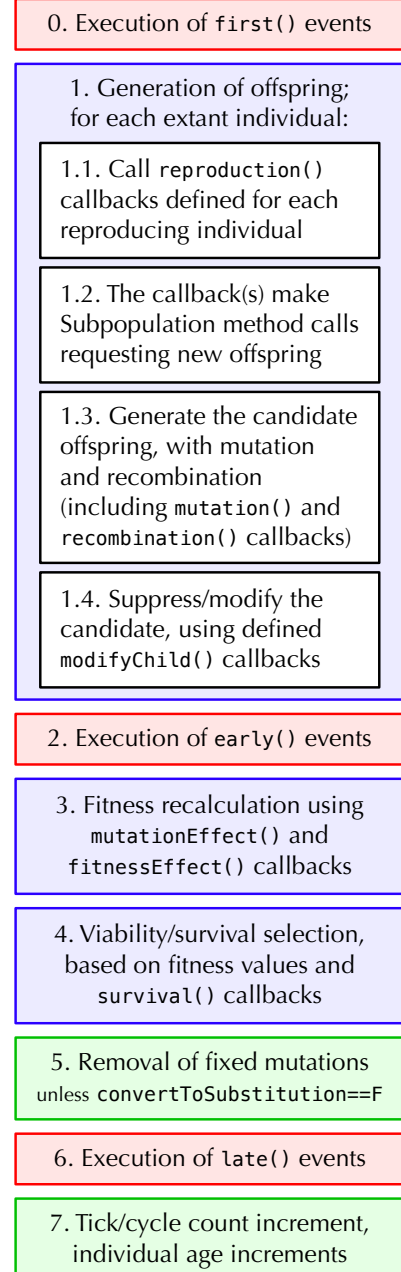
### 23.0 Step 0: Execution of `first()` Eidos events

As in WF models, the first thing to happen in the tick cycle is resetting of the active property of all script blocks (see section 22.1), followed by execution of `first()` events. This event stage (added in SLiM 3.7) provides a point at which script can be executed prior to reproduction in nonWF models. The use of `first()` events is relatively rare; since fitness values have already been calculated at the end of the previous tick, in preparation for reproduction, `first()` events in nonWF models should not do anything that modifies fitness, and so they are somewhat constrained. However, `first()` events can be useful as an opportunity to prepare for reproduction. They also provide a good opportunity to make top-level decisions about what will happen in the tick that is about to execute, such as setting the active property of other script blocks, or calling `skipTick()` in multispecies models.

### 23.1 Step 1: Generation of offspring

After `first()` events, offspring generation occurs. This is the most complex step in SLiM's architecture, and in nonWF models it is broken down into four sub-steps that are executed for

*The nonWF model's tick cycle*



each offspring generated, as shown in the tick cycle diagram. Processes involved in the generation of offspring include mate choice, mutation, recombination, and the actual production of offspring individuals. (In WF models, migration is also effected during offspring generation, but this is not necessarily the case in nonWF models, although it is possible.)

### 23.1.1 *The order of offspring generation*

For each offspring generated, there are generally several decisions to be made: (1) what are its parents, (2) is it male or female (in sexual simulations), and (3) is it produced by cloning, selfing, or biparental mating? In WF models, these decisions are made by SLiM's core engine, as described in section 22.2.1, based upon parameters such as the sex ratio, cloning rate, and selfing rate, as well as upon individual fitness values and the results from `mateChoice()` callbacks. In nonWF models, in contrast, all of these decisions are made by the model's script: `reproduction()` callbacks are called once for each individual in the model, to request that each individual should then generate its own offspring to be added to the population.

The order in which individuals are asked to generate their offspring is somewhat complex. Subpopulations will reproduce sequentially; all of the individuals in a given subpopulation will reproduce, by receiving a `reproduction()` callback, before the next subpopulation begins to reproduce. The order in which subpopulations reproduce – which subpopulation goes first, second, etc. – is undefined and should not be relied upon. Within a given subpopulation, by default (unless the `randomizeCallbacks` parameter to `initializeSLiMOptions()` is set to F) individuals will reproduce in a randomized order to try to prevent order-dependencies from biasing offspring generation. If `randomizeCallbacks` is F, individuals within each subpopulation will reproduce in sequential order, which is more prone to order-dependency issues (but is slightly faster); see section 24.1. Even when the reproduction order is randomized, however, it is important to design your `reproduction()` callbacks to be independent; in general, the reproductive behavior of each individual should probably be independent of the reproductive behavior of every other individual, so that whether A is asked to reproduce before B, or B before A, does not bias the outcome of the model. There are certainly cases where you might violate this principle; in a model of monogamous mating, for example, the first female asked to reproduce might be able to claim her choice of males, whereas the last female asked to reproduce might have little or no choice since most males would already be claimed. Sometimes such asymmetries are acceptable or even desirable; sometimes, however, they would constitute a bug. So the message of this subsection is primarily: think carefully about the order of offspring generation, and about the independence of each reproductive event (or the lack thereof), and make sure your model does what you really want it to.

### 23.1.2 *Individual-based reproduction with `reproduction()` callbacks*

As mentioned above, `reproduction()` callbacks are called for each individual in the model. The task of each callback is to generate the offspring from one focal individual. In sexual models, perhaps females would generate offspring and males would not (or perhaps the opposite, if males are the choosy sex in the biological system being modeled); the focal sex of a `reproduction()` callback can therefore be specified (see section 25.8). In monogamous-mating models, a single mate might be chosen and then a litter of offspring would be generated through crosses with that mate; in a non-monogamous model each offspring generated might be with a different mate. Some offspring might be generated via cloning or selfing, rather than biparental mating. The sex of offspring in sexual models might be equally likely to be male or female, or there might be some sex-ratio bias. Most importantly, in nonWF models all of these decisions can depend upon the genetics and other state of each individual, rather than being dictated by overall parameters as in WF models.

Regardless of how a given model makes these decisions, it always expresses them to SLiM in one way: by making method calls on `Subpopulation` objects, requesting the addition of new offspring. There are five methods that can be called: `addCrossed()` to add an offspring resulting from a biparental cross, `addSelfed()` to add an offspring resulting from selfing (i.e., sexual self-fertilization in a hermaphroditic individual), `addCloned()` to add an offspring resulting from clonal reproduction, `addEmpty()` to add an offspring with no parents and empty genomes (presumably to be filled in some special way by script, subsequently), or `addRecombinant()` to add an offspring using explicitly specified parental genomes and recombination breakpoints. Each such method call made sets off the chain of events described in the following subsections: mutation and recombination, child modification, and child generation.

### 23.1.3 Mutation and recombination

Mutation and recombination occur in nonWF models exactly as they do in WF models; see section 22.2.3 for details.

### 23.1.4 Child modification

Child modification, via `modifyChild()` callbacks, occurs in nonWF models largely as it does in WF models; see section 22.2.4 for details. The important difference is that whereas WF models have a target subpopulation size to reach, and thus must keep generating offspring until that target is reached, the same is not true of nonWF models. In nonWF models, therefore, if a `modifyChild()` callback returns `F` to indicate that a given offspring should not be generated (due to postmating reproductive isolation or genetic incompatibility, for example), that is the end of it. That offspring will simply not be generated; there will be one fewer offspring individual, in the end, than there would have been if the `modifyChild()` callback had returned `T`. In this case, the `Subpopulation` method call that initiated offspring generation, such as `addCrossed()`, will return `NULL` to its caller. This is generally desirable; see section 25.8 for further discussion of how this behavior interacts with `reproduction()` callbacks.

### 23.1.5 Child generation

Once a candidate offspring has been generated and modified, and was not suppressed by a `modifyChild()` callback, it is queued for addition to the target subpopulation. Note that the newly added child will not be visible as a member of the subpopulation until the end of offspring generation for the entire population. This prevents newly generated offspring from being chosen as mates, or otherwise influencing the remainder of the child generation process.

## 23.2 Step 2: Execution of `early()` Eidos events

Offspring generation is followed by the execution of `early()` Eidos events defined by the user, if any. The mechanics of this are the same as in WF models, but the semantics of `early()` versus `late()` events are fairly different in nonWF models, in some ways.

In WF models, `early()` events occur just before offspring generation and `late()` events happen just after. In nonWF models this positioning is different, because offspring generation has been moved earlier in the tick cycle – now, `first()` events occur just before offspring generation and `early()` events happen just after. In WF models, an `early()` event thus provides an opportunity to execute script immediately before reproduction (but after fitness calculations); in nonWF models, `first()` events offer that same opportunity.

Similarly, in WF models `late()` events are usually the best place to add new mutations, because the next tick cycle stage is fitness evaluation, which immediately incorporates the fitness effects of the new mutations; but in nonWF models `early()` events are usually the best place to add new mutations, for the same reason.

In WF models, `late()` events are generally the best place to put output events so that they reflect the final state at the end of a tick; in nonWF models, however, output can make sense in either an `early()` or a `late()` event, depending upon whether you want to see the state of the population before or after viability selection. However, reading in a previously written output file, or otherwise setting up new population state, is generally best done in an `early()` event in nonWF models so that fitness values are recalculated immediately after the change (just as with adding new mutations). So if you call `outputFull()` with the intention of reading the output back in with `readFromPopulationFile()` later, you will probably want to do the output in an `early()` event so that you can, correspondingly, do the read in an `early()` event as well without skipping or doubling any tick cycle stages.

For those who are curious: the reason for this reordering of the tick cycle in nonWF models is the addition of the viability/survival tick cycle stage, which does not exist in WF models. It is desirable to have an opportunity for scripted events between offspring generation and selection, and then between selection and the next offspring generation stage. It is also desirable for selection to happen *after* offspring generation in the cycle, so that the population state at the end of each tick (as displayed in SLiMgui, for example) is after selection has occurred. These constraints, taken together, dictate the order of the tick cycle in nonWF models.

### 23.3 Step 3: Fitness value recalculation

After the execution of `early()` events, the next stage in nonWF models is to compute fitness values for all individuals, including the effects of any `mutationEffect()` and `fitnessEffect()` callbacks. The mechanics of this are exactly the same in nonWF models as in WF models; see the extensive discussion in section 22.6.

However, because of the reordering of the tick cycle, the semantics of fitness evaluation in nonWF models are a bit different. In WF models, as section 22.6 explains, the fitness values calculated in tick *X* are actually used to influence mating in tick *X*+1. In nonWF models, however, fitness values calculated in tick *X* are then used immediately, in tick *X*, to influence survival (see the next subsection). This oddity of WF models is therefore not present in nonWF models, making them a bit conceptually simpler.

The actual meaning of individual fitness values is also fundamentally different between WF and nonWF models; see the next section for discussion. See also section 8.4 for further discussion of fitness in SLiM and SLiMgui.

### 23.4 Step 4: Viability/survival

After fitness values have been recalculated for all individuals, viability selection then occurs immediately. In WF models, viability selection does not exist (or you could consider new offspring to have a survival rate of 100%, and the parental generation to have a survival rate of 0%, if you like). In nonWF models, in contrast, viability selection is the primary way in which differential fitness is expressed; individual fitness values influence survival, not mating success.

Viability selection in nonWF models is mechanistically simple. For a given individual, a fitness of **0.0** or less results in certain death; that individual is immediately removed from its subpopulation. A fitness of **1.0** or greater results in certain survival; that individual remains in its subpopulation, and will live into the next tick cycle. A fitness greater than **0.0** and less than **1.0** is interpreted as a survival probability; SLiM will do a random draw to determine whether the individual survives or not.

This change in the way individual fitness is used has large consequences. First of all, it means that in nonWF models fitness is *absolute* fitness, whereas in WF models fitness is *relative* fitness.



Second, it means that in nonWF models selection is generally *hard* selection, reducing the size of the population proportionate to mean population fitness, whereas in WF models it is generally *soft* selection, changing the relative success of particular individuals but not changing the size of the population. Third, it means that in nonWF models the population is not automatically regulated – both extinction and unbounded exponential growth are very real possibilities – whereas in WF models SLiM automatically regulates the population size. These three observations are really different ways of looking at the same basic fact.

Because of this shift, nonWF models need to treat individual fitness differently than WF models. For one thing, there is often a need to introduce some sort of density-dependent fitness, using `fitnessScaling` or a `fitnessEffect()` callback, that prevents exponential growth by decreasing individual fitness as the population size gets larger. (Density-dependent fecundity or emigration could similarly regulate population size.) Second, there may be a need to rethink how beneficial mutations work, since increasing the fitness of an individual above 1.0 has no effect in nonWF models (since guaranteed survival is as good as it gets); it may be desirable to have a baseline fitness, for individuals possessing empty genomes, of less than 1.0 so that beneficial mutations can increase the probability of survival above that baseline. This could also be achieved with `fitnessScaling` or a `fitnessEffect()` callback; this might be done in conjunction with density-dependence (or might simply result from density-dependence, if it is strong enough to reduce mean fitness sufficiently by itself). All of this is entirely up to the model's script.

Finally, it is worth noting that it is certainly possible to have genetics and other individual state influence mating success and/or fecundity in nonWF models, as in WF models. In nonWF models that is done by influencing the dynamics of the offspring generation stage in script, however, as expressed in `reproduction()` callbacks; it is not an automatic consequence of the fitness values calculated by SLiM.

Beginning in SLiM 3.7, the fitness-based decisions made by SLiM in the viability/survival tick cycle stage can be modified in script, with the use of a `survival()` callback. This is an advanced feature that should be used with caution, since if SLiM's default decisions are overridden fitness may no longer influence survival, density-dependence may no longer regulate population size, and so forth. For models that do require this facility, `survival()` callbacks are documented in section 25.10. The basic operation of this tick cycle stage is the same when `survival()` callbacks are active; however, it should be noted that `survival()` callbacks will be called for all individuals without any modification of the population state, and then once that process is complete, only then will SLiM enact the decisions that were made by killing and/or moving individuals as requested.

As with reproduction (see section 23.1.1), `survival()` callbacks are called one subpopulation at a time. Within a given subpopulation, by default (unless the `randomizeCallbacks` parameter to `initializeSLiMOptions()` is set to F), `survival()` callbacks will be called in a randomized order to try to prevent order-dependencies from biasing the process. If `randomizeCallbacks` is F, `survival()` callbacks within each subpopulation will be called in sequential order, which is more prone to order-dependency issues (but is slightly faster); see section 24.1. Even when the order is randomized, however, it is usually a good idea to design your `survival()` callbacks to be independent; in general, the survival outcome of each individual should probably be independent of the survival outcome of every other individual, so that the processing order does not bias the outcome of the model. There are certainly cases where you might violate this principle; in a model of predation, for example, the first prey individual evaluated might be likely to be eaten by a hungry predator, whereas the last prey individual evaluated might be unlikely to be eaten by the now-sated predators. Sometimes such asymmetries are acceptable or even desirable; sometimes, however, they would constitute a bug. As always, think carefully about the biology of the system you are modeling and what you really want your model to do.

### 23.5 Step 5: Removal of fixed mutations

After viability selection, SLiM tallies mutation frequencies and removes fixed and lost mutations. The mechanics of this are essentially the same as in WF models; see section 22.3.

However, there is one important difference here between WF and nonWF models. In WF models, fixed mutations can generally be removed because they no longer influence evolutionary dynamics, as a consequence of fitness values reflecting *relative* fitness. If every individual in the population is fixed for a given mutation, then that mutation has no effect on relative fitness, regardless of what its selection coefficient might be; the only exceptions are cases where the fitness effect of the mutation varies from individual to individual, such as when epistatic interactions with other segregating mutations are present. In WF models, the `convertToSubstitution` property of mutation types therefore defaults to T, allowing SLiM to remove fixed mutations by default; when that is not desirable, models need to set `convertToSubstitution` to F to prevent that automatic removal.

In nonWF models, in contrast, the `convertToSubstitution` property defaults to F; since fitness values in nonWF models reflect *absolute* fitness, in general it is not safe for SLiM to remove fixed mutations. For this reason, nonWF models need to set `convertToSubstitution` to T to allow automatic removal of fixed mutations. Generally, in nonWF models automatic removal of fixed mutations only makes sense if several conditions are all met: (1) the mutation is neutral, (2) the mutation has no direct non-neutral effects due to any `mutationEffect()` callback, and (3) the mutation has no indirect non-neutral effects on the model through epistasis, mate choice, fecundity, or any other such influences on the model. If these conditions are met – as is commonly the case for simple neutral background mutations – it is very important to set `convertToSubstitution` to T; the effect on SLiM's performance can be very large!

### 23.6 Step 6: Execution of `late()` Eidos events

Once fixed mutations have been removed, the next step is to execute any defined and active `late()` events. This works identically to WF models (see section 22.5). The only important difference is the way in which the semantics and common uses of `first()`, `early()`, and `late()` events differ between WF and nonWF models; see the discussion in section 23.2.

### 23.7 Step 7: Tick/cycle count increment

As in WF models, the last stage of each tick cycle is the incrementing of the cycle counter for all active species, the incrementing of the tick counter kept by the `Community`, and the check for the simulation being finished (see section 22.7). In nonWF models, the age of all surviving individuals in active species is also incremented by one during this stage.



## 24. SLiM classes and built-in functions

This chapter presents reference documentation for all of the Eidos classes and functions that are built into SLiM. It assumes an understanding of the syntax of type-specifiers and function/method signatures, including the use of optional arguments and default values. It also assumes an understanding of how classes are used in Eidos, including how properties are accessed and how methods are called. Part I of this manual attempts to introduce those topics to some degree, but see the Eidos manual for more extensive discussion and documentation of these fundamental language features.

The same color-coding scheme used in the rest of the manual also applies here:

sections that apply only to WF models

sections that apply only to nonWF models

sections that apply only to nucleotide-based models

### 24.1 Simulation initialization: `initialize()` callbacks

Before a SLiM simulation can be run, the various classes underlying the simulation need to be set up with an initial configuration. Simulation configuration in SLiM is done in *initialize()* *callbacks* that run prior to the beginning of simulation execution. Eidos callbacks are discussed more broadly in chapter 25, but for our present purposes, the idea is very simple. In your input file, you can write something like this:

```
initialize()
{
    ...
}
```

The `initialize()` declaration specifies that the script block is to be executed as an `initialize()` callback before the simulation starts. The script between the braces `{}` would set up various aspects of the simulation by calling *initialization functions*. These are SLiM functions that may be called only in an `initialize()` callback, and their names begin with `initialize` to mark them clearly as such. You may also use other Eidos functionality in these callbacks; for example, you might automate generating a complex genetic structure containing many genes by using a `for` loop.

In general, it is required for a species to set up its genetic structure in an `initialize()` callback with calls to `initializeMutationRate()`, `initializeRecombinationRate()`, `initializeMutationType()`, `initializeGenomicElementType()`, and `initializeGenomicElement()`; species must call all of these, setting up at least one mutation type, at least one genomic element type, and at least one genomic element. The exception to this general rule is for species that have no genetics at all – species that are modeled purely on an ecological/behavioral level. Such species may be defined by calling *none* of those initialization functions; in this case, SLiM will default to a zero-length chromosome with mutation and recombination rates of zero. A middle ground between these two configuration paths is not allowed; either a species has no genetics, or it fully defines its genetics.

One thing worth mentioning is that in the context of an `initialize()` callback, the `sim` global representing the species being simulated is not defined. This is because the state of the simulation is not yet constructed fully, and accessing partially constructed state would not be safe. (Similarly, in multispecies models, the `community` object and the objects representing individual species are not yet defined.)

The above `initialize()` callback syntax *implicitly* declares a single species, with the default name of `sim`, and therefore sets up a single-species model. It is also possible to *explicitly* declare a species, which is done with this extended syntax (using a species name of `fox` as an example):

```
species fox initialize() { ... }
```

This sets up a multispecies model (although it might, in fact, declare only a single species, `fox`; the term “multispecies”, in SLiM parlance, really means “explicitly declared species”, but multispecies models almost always *do* contain multiple species, so the distinction is unimportant). See section 1.9 and chapter 19 for further discussion of multispecies models; in most respects they work identically to single-species models, so we will tend to focus on the single-species case in the reference documentation, with a species name of `sim`, for simplicity and clarity.

In single-species models all initialization can be done in a single `initialize()` callback (or you can have more than one, if you wish). In multispecies models, each species must be initialized with its own callback(s), as shown above. In addition, multispecies models also support an optional community-level initialization callback that is declared as follows:

```
species all initialize() { ... }
```

These callbacks, technically called *non-species-specific initialize() callbacks*, provide a place for community-level initialization to occur. They are run before any species-specific `initialize()` callbacks are run, so you might wish to set up all of your model parameters in one, providing a single location for all parameters. In multispecies models, the initialization functions `initializeSLiMModelType()` and `initializeInteractionType()` may only be called from a non-species-specific `initialize()` callback, since those aspects of model configuration span the entire community. In single-species models, these functions may be called from an ordinary `initialize()` callback for simplicity and backward compatibility.

Without further ado, then, here are the initialization functions provided by SLiM:

```
(integer$)initializeAncestralNucleotides(is sequence)
```

This function, which may be called only in nucleotide-based models (see section 1.8), supplies an ancestral nucleotide sequence for the model. The `sequence` parameter may be an `integer` vector providing nucleotide values (A=0, C=1, G=2, T=3), or a `string` vector providing single-character nucleotides ("A", "C", "G", "T"), or a singleton `string` providing the sequence as one string ("ACGT..."), or a singleton `string` providing the filesystem path of a FASTA file which will be read in to provide the sequence (if the file contains than one sequence, the first sequence will be used). Only A/C/G/T nucleotide values may be provided; other symbols, such as those for amino acids, gaps, or nucleotides of uncertain identity, are not allowed. The two semantic meanings of `sequence` that involve a singleton `string` value are distinguished heuristically; a singleton `string` that contains only the letters ACGT will be assumed to be a nucleotide sequence rather than a filename. The length of the ancestral sequence is returned.

A utility function, `randomNucleotides()`, is provided by SLiM to assist in generating simple random nucleotide sequences; see section 24.17.1.

```
(void)initializeGeneConversion(numeric$ nonCrossoverFraction, numeric$ meanLength,
    numeric$ simpleConversionFraction, [numeric$ bias = 0])
```

Calling this function switches the recombination model from a “simple crossover” model to a “double-stranded break (DSB)” model, and configures the details of the gene conversion tracts that will therefore be modeled (see section 1.5.6 for discussion of these models). The fraction of DSBs that will be modeled as non-crossover events is given by `nonCrossoverFraction`. The mean length of gene conversion tracts (whether associated with crossover or non-crossover events) is given by `meanLength`; the actual extent of a gene conversion tract will be the sum of two independent draws from a geometric distribution with mean `meanLength/2`. The fraction of gene conversion tracts that

are modeled as “simple” is given by `simpleConversionFraction`; the remainder will be modeled as “complex”, involving repair of heteroduplex mismatches. Finally, the GC bias during heteroduplex mismatch repair is given by `bias`, with the default of `0.0` indicating no bias, `1.0` indicating an absolute preference for G/C mutations over A/T mutations, and `-1.0` indicating an absolute preference for A/T mutations over G/C mutations. A non-zero bias may only be set in nucleotide-based models (see section 1.8). This function, and the way that gene conversion is modeled, fundamentally changed in SLiM 3.3; see section 1.5.6 for discussion.

`(object<GenomicElement>) initializeGenomicElement(io<GenomicElementType> genomicElementType, integer start, integer end)`

Add a genomic element to the chromosome at initialization time. The `start` and `end` parameters give the first and last base positions to be spanned by the new genomic element. The new element will be based upon the genomic element type identified by `genomicElementType`, which can be either an `integer`, representing the ID of the desired element type, or an `object` of type `GenomicElementType` specified directly.

Beginning in SLiM 3.3, this function is vectorized: the `genomicElementType`, `start`, and `end` parameters do not have to be singletons. In particular, `start` and `end` may be of any length, but must be equal in length; each `start/end` element pair will generate one new genomic element spanning the given base positions. In this case, `genomicElementType` may still be a singleton, providing the genomic element type to be used for all of the new genomic elements, or it may be equal in length to `start` and `end`, providing an independent genomic element type for each new element. When adding a large number of genomic elements, it will be much faster to add them in order of ascending position with a vectorized call.

The return value provides the genomic element(s) created by the call, in the order in which they were specified in the parameters to `initializeGenomicElement()`.

`(object<GenomicElementType>$) initializeGenomicElementType(is$ id, io<MutationType> mutationTypes, numeric proportions, [Nf mutationMatrix = NULL])`

Add a genomic element type at initialization time. The `id` must not already be used for any genomic element type in the simulation. The `mutationTypes` vector identifies the mutation types used by the genomic element, and the `proportions` vector should be of equal length, specifying the relative proportion of mutations that will be drawn from the corresponding mutation type (proportions do not need to add up to one; they are interpreted relatively). The `id` parameter may be either an `integer` giving the ID of the new genomic element type, or a `string` giving the name of the new genomic element type (such as “g5” to specify an ID of 5). The `mutationTypes` parameter may be either an `integer` vector representing the IDs of the desired mutation types, or an `object` vector of `MutationType` elements specified directly. The global symbol for the new genomic element type is immediately available; the return value also provides the new object.

The `mutationMatrix` parameter is `NULL` by default, and in non-nucleotide-based models it must be `NULL`. In nucleotide-based models, on the other hand, it must be non-`NULL`, and therefore must be supplied. In that case, `mutationMatrix` should take one of two standard forms. For sequence-based mutation rates that depend upon only the single nucleotide at a mutation site, `mutationMatrix` should be a 4×4 `float` matrix, specifying mutation rates for an existing nucleotide state (rows from 0–3 representing A/C/G/T) to each of the four possible derived nucleotide states (columns, with the same meaning):

$$\begin{bmatrix} * & P_{A \rightarrow C} & P_{A \rightarrow G} & P_{A \rightarrow T} \\ P_{C \rightarrow A} & * & P_{C \rightarrow G} & P_{C \rightarrow T} \\ P_{G \rightarrow A} & P_{G \rightarrow C} & * & P_{G \rightarrow T} \\ P_{T \rightarrow A} & P_{T \rightarrow C} & P_{T \rightarrow G} & * \end{bmatrix}$$

The mutation rates in this matrix are absolute rates, per nucleotide per gamete; they will be used by SLiM directly unless they are multiplied by a factor from the hotspot map (see `initializeHotspotMap()`). Rates in `mutationMatrix` that involve the mutation of a nucleotide to

itself (A to A, C to C, etc.) are not used by SLiM and must be **0.0** by convention (shown above with asterisks).

It is important to note that the order of the rows and columns used in SLiM, A/C/G/T, is not a universal convention; other sources will present substitution-rate/transition-rate matrices using different conventions, and so care must be taken when importing such matrices into SLiM.

For sequence-based mutation rates that depend upon the trinucleotide sequence centered upon a mutation site (the adjacent bases to the left and right, in other words, as well as the mutating nucleotide itself), `mutationMatrix` should be a 64×4 float matrix, specifying mutation rates for the central nucleotide of an existing trinucleotide sequence (rows from 0–63, representing trinucleotides as described in the documentation for the `ancestralNucleotides()` method of `Chromosome`) to each of the four possible derived nucleotide states (columns from 0–3 for A/C/G/T as before):

*	$P_{AAA \rightarrow ACA}$	$P_{AAA \rightarrow AGA}$	$P_{AAA \rightarrow ATA}$
*	$P_{AAC \rightarrow ACC}$	$P_{AAC \rightarrow AGC}$	$P_{AAC \rightarrow ATC}$
*	$P_{AAG \rightarrow ACG}$	$P_{AAG \rightarrow AGG}$	$P_{AAG \rightarrow ATG}$
*	$P_{AAT \rightarrow ACT}$	$P_{AAT \rightarrow AGT}$	$P_{AAT \rightarrow ATT}$
$P_{ACA \rightarrow AAA}$	*	$P_{ACA \rightarrow AGA}$	$P_{ACA \rightarrow ATA}$
$P_{ACC \rightarrow AAC}$	*	$P_{ACC \rightarrow AGC}$	$P_{ACC \rightarrow ATC}$
$P_{ACG \rightarrow AAG}$	*	$P_{ACG \rightarrow AGG}$	$P_{ACG \rightarrow ATG}$
...	...	...	...
$P_{TTC \rightarrow TAC}$	$P_{TTC \rightarrow TCC}$	$P_{TTC \rightarrow TGC}$	*
$P_{TTG \rightarrow TAG}$	$P_{TTG \rightarrow TCG}$	$P_{TTG \rightarrow TGG}$	*
$P_{TTT \rightarrow TAT}$	$P_{TTT \rightarrow TCT}$	$P_{TTT \rightarrow TGT}$	*

Note that in every case it is the central nucleotide of the trinucleotide sequence that is mutating, but rates can be specified independently based upon the nucleotides in the first and third positions as well, with this type of mutation matrix.

Several helper functions are defined to construct common types of mutation matrices, such as `mmJukesCantor()` to create a mutation matrix for a Jukes–Cantor model; see section 24.17.1. See chapter 18 for practical examples of mutation matrices, and section 22.2.3 for further discussion of the mutational paradigm in nucleotide-based models.

```
(void)initializeHotspotMap(numeric multipliers, [Ni ends = NULL],
    [string$ sex = "*"])
```

In nucleotide-based models, set the mutation rate *multiplier* along the chromosome. Nucleotide-based models define sequence-based mutation rates that are set up with the `mutationMatrix` parameter to `initializeGenomicElementType()`. If no hotspot map is specified by calling `initializeHotspotMap()`, a hotspot map with a multiplier of **1.0** across the whole chromosome is assumed (and so the sequence-based rates are the absolute mutation rates used by SLiM). A hotspot map modifies the sequence-based rates by scaling them up in some regions, with multipliers greater than **1.0** (representing mutational hot spots), and/or scaling them down in some regions, with multipliers less than **1.0** (representing mutational cold spots).

There are two ways to call this function. If the optional `ends` parameter is `NULL` (the default), then `multipliers` must be a singleton value that specifies a single multiplier to be used along the entire chromosome (typically **1.0**, but not required to be). If, on the other hand, `ends` is supplied, then `multipliers` and `ends` must be the same length, and the values in `ends` must be specified in ascending order. In that case, `multipliers` and `ends` taken together specify the multipliers to be used along successive contiguous stretches of the chromosome, from beginning to end; the last position specified in `ends` should extend to the end of the chromosome (i.e. at least to the end of the last genomic element, if not further).

For example, if the following call is made:

```
initializeHotspotMap(c(1.0, 1.2), c(5000, 9999));
```

then the result is that the mutation rate multiplier for bases 0...5000 (inclusive) will be 1.0 (and so the specified sequence-based mutation rates will be used verbatim), and the multiplier for bases 5001...9999 (inclusive) will be 1.2 (and so the sequence-based mutation rates will be multiplied by 1.2 within the region).

Note that mutations are generated by SLiM only within genomic elements, regardless of the hotspot map. In effect, the hotspot map given is intersected with the coverage area of the genomic elements defined; areas outside of any genomic element are given a multiplier of zero. There is no harm in supplying a hotspot map that specifies multipliers for areas outside of the genomic elements defined; the excess information is simply not used.

If the optional `sex` parameter is "\*" (the default), then the supplied hotspot map will be used for both sexes (which is the only option for hermaphroditic simulations). In sexual simulations `sex` may be "M" or "F" instead, in which case the supplied hotspot map is used only for that sex (i.e., when generating a gamete from a parent of that sex). In this case, two calls must be made to `initializeHotspotMap()`, one for each sex, even if a multiplier of 1.0 is desired for the other sex; no default hotspot map is supplied.

```
(object<InteractionType>$)initializeInteractionType(is$ id, string$ spatiality,
  [logical$ reciprocal = F], [numeric$ maxDistance = INF],
  [string$ sexSegregation = "**"])
```

Add an interaction type at initialization time. The `id` must not already be used for any interaction type in the simulation. The `id` parameter may be either an integer giving the ID of the new interaction type, or a string giving the name of the new interaction type (such as "i5" to specify an ID of 5).

The `spatiality` may be "", for non-spatial interactions (i.e., interactions that do not depend upon the distance between individuals); "x", "y", or "z" for one-dimensional interactions; "xy", "xz", or "yz" for two-dimensional interactions; or "xyz" for three-dimensional interactions. The dimensions referenced by `spatiality` must be defined as spatial dimensions with `initializeSLiMOptions()`; if the simulation has dimensionality "xy", for example, then interactions in the simulation may have spatiality "", "x", "y", or "xy", but may not reference spatial dimension z and thus may not have spatiality "xz", "yz", or "xyz". If no spatial dimensions have been configured, only non-spatial interactions may be defined.

The `reciprocal` flag may be T, in which case the interaction is guaranteed by the user to be *reciprocal*: whatever the interaction strength is for exiter B upon receiver A, it will be equal (in magnitude and sign) for exiter A upon receiver B. In principle, this allows the `InteractionType` to reduce the amount of computation necessary by up to a factor of two (although it may or may not be used). If `reciprocal` is F, the interaction is not guaranteed to be reciprocal and each interaction will be computed independently. The built-in interaction formulas are all reciprocal, but if you implement an `interaction()` callback (see section 25.7), you must consider whether the callback you have implemented preserves reciprocity or not. For this reason, the default is `reciprocal=F`, so that bugs are not inadvertently introduced by an invalid assumption of reciprocity. See below for a note regarding reciprocity in sexual simulations when using the `sexSegregation` flag.

The `maxDistance` parameter supplies the maximum distance over which interactions of this type will be evaluated; at greater distances, the interaction strength is considered to be zero (for efficiency). The default value of `maxDistance`, `INF` (positive infinity), indicates that there is no maximum interaction distance; note that this can make some interaction queries much less efficient, and is therefore not recommended. In SLiM 3.1 and later, a warning will be issued if a spatial interaction type is defined with no maximum distance to encourage a maximum distance to be defined.

The `sexSegregation` parameter governs the applicability of the interaction to each sex, in sexual simulations. It does not affect distance calculations in any way; it only modifies the way in which interaction strengths are calculated. The default, "\*\*", implies that the interaction is felt by both sexes (the first character of the string value) and is exerted by both sexes (the second character of the string value). Either or both characters may be M or F instead; for example, "MM" would indicate a male-male interaction, such as male-male competition, whereas "FM" would indicate an interaction influencing only female receivers that is influenced only by male exiterers, such as male mating



displays that influence female attraction. This parameter may be set only to "\*" unless sex has been enabled with `initializeSex()`. Note that a value of `sexSegregation` other than "\*" may imply some degree of non-reciprocity, but it is not necessary to specify `reciprocal` to be F for this reason; SLiM will take the sex-segregation of the interaction into account for you. The value of `reciprocal` may therefore be interpreted as meaning: in those cases, if any, in which A interacts with B and B interacts with A, is the interaction strength guaranteed to be the same in both directions?

By default, the interaction strength is 1.0 for all interactions within `maxDistance`. Often it is desirable to change the interaction function using `setInteractionFunction()`; modifying interaction strengths can also be achieved with `interaction()` callbacks if necessary (see section 25.7). In any case, interactions beyond `maxDistance` always have a strength of 0.0, and the interaction strength of an individual with itself is always 0.0, regardless of the interaction function or callbacks.

The global symbol for the new interaction type is immediately available; the return value also provides the new object. Note that in multispecies models, `initializeInteractionType()` must be called from a non-species-specific `interaction()` callback (declared as `species all initialize()`), since interactions are managed at the community level.

`(void)initializeMutationRate(numeric rates, [Ni ends = NULL], [string$ sex = "*"])`

Set the mutation rate per base position per gamete. To be precise, this mutation rate is the expected mean number of mutations that will occur per base position per gamete; note that this is different from how the recombination rate is defined (see `initializeRecombinationRate()`). The number of mutations that actually occurs at a given base position when generating an offspring genome is, in effect, drawn from a Poisson distribution with that expected mean (but under the hood SLiM uses a mathematically equivalent but much more efficient strategy). It is possible for this Poisson draw to indicate that two or more new mutations have arisen at the same base position, particularly when the mutation rate is very high; in this case, the new mutations will be added to the site one at a time, and as always the mutation stacking policy (see section 1.5.3) will be followed.

There are two ways to call this function. If the optional `ends` parameter is `NULL` (the default), then `rates` must be a singleton value that specifies a single mutation rate to be used along the entire chromosome. If, on the other hand, `ends` is supplied, then `rates` and `ends` must be the same length, and the values in `ends` must be specified in ascending order. In that case, `rates` and `ends` taken together specify the mutation rates to be used along successive contiguous stretches of the chromosome, from beginning to end; the last position specified in `ends` should extend to the end of the chromosome (i.e. at least to the end of the last genomic element, if not further).

For example, if the following call is made:

```
initializeMutationRate(c(1e-7, 2.5e-8), c(5000, 9999));
```

then the result is that the mutation rate for bases 0...5000 (inclusive) will be 1e-7, and the rate for bases 5001...9999 (inclusive) will be 2.5e-8.

Note that mutations are generated by SLiM only within genomic elements, regardless of the mutation rate map. In effect, the mutation rate map given is intersected with the coverage area of the genomic elements defined; areas outside of any genomic element are given a mutation rate of zero. There is no harm in supplying a mutation rate map that specifies rates for areas outside of the genomic elements defined; that rate information is simply not used. The `overallMutationRate` family of properties on `Chromosome` provide the overall mutation rate after genomic element coverage has been taken into account, so it will reflect the rate at which new mutations will actually be generated in the simulation as configured.

If the optional `sex` parameter is "\*" (the default), then the supplied mutation rate map will be used for both sexes (which is the only option for hermaphroditic simulations). In sexual simulations `sex` may be "M" or "F" instead, in which case the supplied mutation rate map is used only for that sex (i.e., when generating a gamete from a parent of that sex). In this case, two calls must be made to `initializeMutationRate()`, one for each sex, even if a rate of zero is desired for the other sex; no default mutation rate map is supplied.



In nucleotide-based models, `initializeMutationRate()` may not be called. Instead, the desired sequence-based mutation rate(s) should be expressed in the `mutationMatrix` parameter to `initializeGenomicElementType()`. If variation in the mutation rate along the chromosome is desired, `initializeHotspotMap()` should be used.

```
(object<MutationType>$)initializeMutationType(is$ id, numeric$ dominanceCoeff,  
  string$ distributionType, ...)
```

Add a mutation type at initialization time. The `id` must not already be used for any mutation type in the simulation. The `id` parameter may be either an integer giving the ID of the new mutation type, or a string giving the name of the new mutation type (such as "m5" to specify an ID of 5). The `dominanceCoeff` parameter supplies the dominance coefficient for the mutation type; `0.0` produces no dominance, `1.0` complete dominance, and values greater than `1.0`, overdominance. The `distributionType` may be "f", in which case the ellipsis ... should supply a numeric\$ fixed selection coefficient; "e", in which case the ellipsis should supply a numeric\$ mean selection coefficient for an exponential distribution; "g", in which case the ellipsis should supply a numeric\$ mean selection coefficient and a numeric\$ alpha shape parameter for a gamma distribution; "n", in which case the ellipsis should supply a numeric\$ mean selection coefficient and a numeric\$ sigma (standard deviation) parameter for a normal distribution; "w", in which case the ellipsis should supply a numeric\$  $\lambda$  scale parameter and a numeric\$ k shape parameter for a Weibull distribution; or "s", in which case the ellipsis should supply a string\$ Eidos script parameter. See section 24.11 for discussion of the various DFEs and their uses. The global symbol for the new mutation type is immediately available; the return value also provides the new object.

Note that by default in WF models, all mutations of a given mutation type will be converted into `Substitution` objects when they reach fixation, for efficiency reasons. If you need to disable this conversion, to keep mutations of a given type active in the simulation even after they have fixed, you can do so by setting the `convertToSubstitution` property of `MutationType` to `F`.

In contrast, by default in nonWF models mutations will not be converted into `Substitution` objects when they reach fixation; `convertToSubstitution` is `F` by default in nonWF models. To enable conversion in nonWF models for neutral mutation types with no indirect fitness effects, you should therefore set `convertToSubstitution` to `T`.

See sections 22.3, 23.5, and 24.11.1 for further discussion regarding the `convertToSubstitution` property.

```
(object<MutationType>$)initializeMutationTypeNuc(is$ id, numeric$ dominanceCoeff,  
  string$ distributionType, ...)
```

Add a nucleotide-based mutation type at initialization time. This function is identical to `initializeMutationType()` except that the new mutation type will be nucleotide-based – in other words, mutations belonging to the new mutation type will have an associated nucleotide. This function may be called only in nucleotide-based models (as enabled by the `nucleotideBased` parameter to `initializeSLiMOptions()`).

Nucleotide-based mutations always use a `mutationStackGroup` of `-1` and a `mutationStackPolicy` of `"l"`. This ensures that a new nucleotide mutation always replaces any previously existing nucleotide mutation at a given position, regardless of the mutation types of the nucleotide mutations. These values are set automatically by `initializeMutationTypeNuc()`, and may not be changed.

See the documentation for `initializeMutationType()` for all other discussion.

```
(void)initializeRecombinationRate(numeric rates, [Ni ends = NULL],  
  [string$ sex = "*"])
```

Set the recombination rate per base position per gamete. To be precise, this recombination rate is the probability that a breakpoint will occur between one base and the next base; note that this is different from how the mutation rate is defined (see `initializeMutationRate()`). All rates must be in the interval `[0.0, 0.5]`. A rate of `0.5` implies complete independence between the adjacent bases, which might be used to implement independent assortment of loci located on different chromosomes (see

the example below). Whether a breakpoint occurs between two bases is then, in effect, determined by a binomial draw with a single trial and the given rate as probability (but under the hood SLiM uses a mathematically equivalent but much more efficient strategy). The recombinational process in SLiM will never generate more than one crossover between one base and the next (in one generation/genome), and a supplied rate of **0.5** will therefore result in an actual probability of **0.5** for a crossover at the relevant position. (Note that this was not true in SLiM 2.x and earlier, however; their implementation of recombination resulted in a crossover probability of about 39.3% for a rate of **0.5**, due to the use of an inaccurate approximation method. Recombination rates lower than about **0.01** would have been essentially exact, since the approximation error became large only as the rate approached **0.5**.)

There are two ways to call this function. If the optional **ends** parameter is **NULL** (the default), then **rates** must be a singleton value that specifies a single recombination rate to be used along the entire chromosome. If, on the other hand, **ends** is supplied, then **rates** and **ends** must be the same length, and the values in **ends** must be specified in ascending order. In that case, **rates** and **ends** taken together specify the recombination rates to be used along successive contiguous stretches of the chromosome, from beginning to end; the last position specified in **ends** should extend to the end of the chromosome (i.e. at least to the end of the last genomic element, if not further). Note that a recombination rate of 1 centimorgan/Mbp corresponds to a recombination rate of **1e-8** in the units used by SLiM.

For example, if the following call is made:

```
initializeRecombinationRate(c(0, 0.5, 0), c(5000, 5001, 9999));
```

then the result is that the recombination rates between bases **0 / 1**, **1 / 2**, ..., **4999 / 5000** will be **0**, the rate between bases **5000 / 5001** will be **0.5**, and the rate between bases **5001 / 5002** onward (up to **9998 / 9999**) will again be **0**. Setting the recombination rate between one specific pair of bases to **0.5** forces recombination to occur with a probability of **0.5** between those bases, which effectively breaks the simulated locus into separate chromosomes at that point; this example effectively has one simulated chromosome from base position **0** to **5000**, and another from **5001** to **9999**.

If the optional **sex** parameter is **"\*"** (the default), then the supplied recombination rate map will be used for both sexes (which is the only option for hermaphroditic simulations). In sexual simulations **sex** may be **"M"** or **"F"** instead, in which case the supplied recombination map is used only for that sex. In this case, two calls must be made to **initializeRecombinationRate()**, one for each sex, even if a rate of zero is desired for the other sex; no default recombination map is supplied.

**(void)initializeSex(string\$ chromosomeType)**

Enable and configure sex in the simulation. The argument **chromosomeType** gives the type of chromosome to be simulated; this should be **"A"**, **"X"**, or **"Y"**. Calling this function has the side effect of enabling sex in the simulation; individuals will be male and female (rather than hermaphroditic) regardless of the **chromosomeType** chosen for simulation. There is no way to disable sex once it has been enabled; if you don't want to have sex, don't call this function.

**The **xDominanceCoeff** parameter has been deprecated and removed.** In SLiM 3.7 and later, use the **haploidDominanceCoeff** property of **MutationType** instead. If the **chromosomeType** is **"X"**, the optional **xDominanceCoeff** parameter can supply the dominance coefficient used when a mutation is present in an XY male, and is thus "heterozygous" (but in a different sense than the heterozygosity of an XX female with one copy of the mutation).

**(void)initializeSLiMModelType(string\$ modelType)**

Configure the type of SLiM model used for the simulation. At present, one of two model types may be selected. If **modelType** is **"WF"**, SLiM will use a Wright-Fisher (WF) model; this is the model type that has always been supported by SLiM, and is the model type used if **initializeSLiMModelType()** is not called. If **modelType** is **"nonWF"**, SLiM will use a non-Wright-Fisher (nonWF) model instead; this is a new model type supported by SLiM 3.0 and above (see section 1.6).

If `initializeSLiMModelType()` is called at all then it must be called before any other initialization function, so that SLiM knows from the outset which features are enabled and which are not.

```
(void)initializeSLiMOptions([logical$ keepPedigrees = F],  
  [string$ dimensionality = ""], [string$ periodicity = ""],  
  [integer$ mutationRuns = 0], [logical$ preventIncidentalSelfing = F],  
  [logical$ nucleotideBased = F], [logical$ randomizeCallbacks = F])
```

Configure options for the simulation. If `initializeSLiMOptions()` is called at all then it must be called before any other initialization function (except `initializeSLiMModelType()`), so that SLiM knows from the outset which optional features are enabled and which are not.

If `keepPedigrees` is T, SLiM will keep pedigree information for every individual in the simulation, tracking the identity of its parents and grandparents. This allows individuals to assess their degree of pedigree-based relatedness to other individuals (see `Individual's relatedness()` method, section 24.7.2), as well as allowing a model to find “trios” (two parents and an offspring they generated) using the pedigree properties of `Individual` (section 24.7.1). As a side effect of `keepPedigrees` being T, the `pedigreeID`, `pedigreeParentIDs`, and `pedigreeGrandparentIDs` properties of `Individual` will have defined values (see section 24.7.1), as will the `genomePedigreeID` property of `Genome` (see section 24.4.1). Note that pedigree-based relatedness doesn't necessarily correspond to genetic relatedness, due to effects such as assortment and recombination. For an overview of other ways of tracking genetic ancestry, including true local ancestry at each position on the chromosome, see sections 1.7 and 14.7. Beginning in SLiM 3.5, `keepPedigrees=T` also enables tracking of individual reproductive output, available through the `reproductiveOutput` property of `Individual` (see section 24.7.1) and the `lifetimeReproductiveOutput` property of `Subpopulation` (see section 24.15.1).

If `dimensionality` is not "", SLiM will enable its optional “continuous space” facility. Three values for `dimensionality` are presently supported: “x”, “xy”, and “xyz”, specifying that continuous space should be enabled for one, two, or three dimensions, respectively, using (x), (x, y), and (x, y, z) coordinates respectively. This has a number of side effects. First of all, it means that the specified properties of `Individual` (x, y, and/or z) will be interpreted by SLiM as spatial positions; in particular, SLiMgui will use those properties to display subpopulations spatially. Second, it allows spatial interactions to be defined, evaluated, and queried using `initializeInteractionType()` and `interaction()` callbacks. And third, it enables the use of any other properties and methods related to continuous space, such as setting the spatial boundaries of subpopulations, which would otherwise raise an error.

If `periodicity` is not "", SLiM will designate the specified spatial dimensions as being periodic – wrapping around at the edges of the spatial boundaries of that dimension. This option may only be used if the `dimensionality` parameter to `initializeSLiMOptions()` has been used to enable spatiality in the model, and only spatial dimensions that were specified in the `dimensionality` of the model may be declared to be periodic (but if desired, it is permissible to make just a subset of those dimensions periodic; it is not an all-or-none proposition). For example, if the specified `dimensionality` is “xy”, the model's periodicity may be “x”, “y”, or “xy” (or “”, the default, to specify that there are no periodic dimensions). A one-dimensional periodic model would model a space like the perimeter of a circle. A two-dimensional model periodic in one of those dimensions would model a space like a cylinder without its end caps; if periodic in both dimensions, the modeled space is a torus. The shapes of three-dimensional periodic models are harder to visualize, but are essentially higher-dimensional analogues of these concepts. Periodic boundary conditions are commonly used to model spatial scenarios without “edge effects”, since there are no edges in the periodic spatial dimensions. The `pointPeriodic()` method of `Subpopulation` is typically used in conjunction with this option, to actually implement the periodic boundary condition for the specified dimensions.

If `mutationRuns` is not 0, SLiM will use the value given as the number of mutation runs inside `Genome` objects; if it is 0 (the default), SLiM will calculate a number of mutation runs that it estimates will work well. Internally, SLiM divides genomes into a sequence of consecutive mutation runs, allowing more efficient internal computations. The optimal mutation run length is short enough that each mutation

run is relatively unlikely to be modified by mutation/recombination events when inherited, but long enough that each mutation run is likely to contain a relatively large number of mutations; these priorities are in tension, so an intermediate balance between them is generally desirable. The optimal number of mutation runs will depend upon the machine and even the compiler used to build SLiM, so SLiM's default value may not be optimal; for maximal performance it can thus be beneficial to experiment with different values and find the optimal value for the simulation – a process which SLiM can assist with (see section 21.4). Specifying the number of mutation runs is an advanced technique, but in certain cases it can improve performance significantly.

If `preventIncidentalSelfing` is T, incidental selfing in hermaphroditic models will be prevented by SLiM. By default (i.e., if `preventIncidentalSelfing` is F), SLiM chooses the first and second parents in a biparental mating event independently. It is therefore possible for the same individual to be chosen as both the first and second parent, resulting in selfing events even when the selfing rate is zero. In many models this is unimportant, since it happens fairly infrequently and does not have large consequences. This behavior is SLiM's default because it is the simplest option, and produces results that most closely align with simple analytical population genetics models. However, in some models this selfing can be undesirable and problematic. In particular, models that involve very high variance in fitness or very small effective population sizes may see elevated rates of selfing that substantially influence model results. If `preventIncidentalSelfing` is set to T, all such incidental selfing will be prevented (by choosing a new second parent if the first parent was chosen again). Non-incidental selfing, as requested by the selfing rate, will still be permitted. Note that if incidental selfing is prevented, SLiM will hang if it is unable to find a different second parent; there must always be at least two individuals in the population with non-zero fitness, and `mateChoice()` and `modifyChild()` callbacks must not absolutely prevent those two individuals from producing viable offspring. Enforcement of the prohibition on incidental selfing will occur after `mateChoice()` callbacks have been called (and thus the default mating weights provided to `mateChoice()` callbacks will *not* exclude the first parent!), but will occur before `modifyChild()` callbacks are called (so those callbacks may assume that the first and second parents are distinct).

If `nucleotideBased` is T, the model will be nucleotide-based. In this case, auto-generated mutations (i.e., mutation types used by genomic element types) must be nucleotide-based, and an ancestral nucleotide sequence must be supplied with `initializeAncestralNucleotides()`. Non-nucleotide-based mutations may still be used, but may not be referenced by genomic element types. A mutation rate (or rate map) may not be supplied with `initializeMutationRate()`; instead, a hotspot map may (optionally) be supplied with `initializeHotspotMap()`. This choice has many consequences across SLiM; see section 1.8 for further discussion.

If `randomizeCallbacks` is T (the default), the order in which individuals are processed in callbacks will be randomized to make it easier to avoid order-dependency bugs. This flag exists because the order of individuals in each subpopulation is non-random; most notably, females always come before males in the individuals vector, but non-random ordering may also occur with respect to things like migrant versus non-migrant status, origin by selfing versus cloning versus biparental mating, and other factors. When this option is F, individuals in a subpopulation are processed in the order of the individuals vector in each tick cycle stage, which may lead to order-dependency issues if there is an enabled callback whose behavior is not fully independent between calls. Setting this option to T will cause individuals within each subpopulation to be processed in a randomized order in each tick cycle stage; specifically, this randomizes the order of calls to `mutationEffect()` callbacks in both WF and nonWF models, and calls to `reproduction()` and `survival()` callbacks in nonWF models. Each subpopulation is still processed separately, in sequential order, so order-dependency issues between subpopulations are still possible if callbacks have effects that are not fully independent. This feature was added in SLiM 4, breaking backward compatibility; to recover the behavior of previous versions of SLiM, pass F for this option (but then be very careful about order-dependency issues in your script). The default of T is the safe option, but a small speed penalty is incurred by the randomization of the processing order – for most models the difference will be less than 1%, but in the worst case it may approach 10%. Models that do not have any order-dependency issue may therefore run somewhat faster if this is set to F. Note that anywhere that your script uses the `individuals` property of



Subpopulation, the order of individuals returned will be non-random (regardless of the setting of this option); you should use `sample()` to shuffle the order of the individuals vector if necessary to avoid order-dependency issues in your script.

This function will likely be extended with further options in the future, added on to the end of the argument list. Using named arguments with this call is recommended for readability. Note that turning on optional features may increase the runtime and memory footprint of SLiM.

```
(void)initializeSpecies([integer$ tickModulo = 1], [integer$ tickPhase = 1],  
  [string$ avatar = ""], [string$ color = ""])
```

Configure options for the species being initialized. This initialization function may only be called in multispecies models (i.e., models with explicit species declarations); in single-species models, the default values are assumed and cannot be changed.

The `tickModulo` and `tickPhase` parameters determine the activation schedule for the species. The active property of the species will be set to T (thus activating the species) every `tickModulo` ticks, beginning in tick `tickPhase`. (However, when the species is activated in a given tick, the `skipTick()` method may still be called in a `first()` event to deactivate it.) See the active property of `Species` (section 24.14.1) for more details.

The `avatar` parameter, if not "", sets a string value used to represent the species graphically, particularly in SLiMgui but perhaps in other contexts also. The `avatar` should generally be a single character – usually an emoji corresponding to the species, such as "🦊" for foxes or "🐭" for mice. If `avatar` is the empty string, "", SLiMgui will choose a default avatar.

The `color` parameter, if not "", sets a string color value used to represent the species in SLiMgui. Colors may be specified by name, or with hexadecimal RGB values of the form "#RRGGBB" (see the Eidos manual for details). If `color` is the empty string, "", SLiMgui will choose a default color.

```
(void)initializeTreeSeq([logical$ recordMutations = T],  
  [Nif$ simplificationRatio = NULL], [Ni$ simplificationInterval = NULL],  
  [logical$ checkCoalescence = F], [logical$ runCrosschecks = F],  
  [logical$ retainCoalescentOnly = T], [Ns$ timeUnit = NULL])
```

Configure options for tree sequence recording. Calling this function turns on tree sequence recording, as a side effect, for later reconstruction of the simulation's evolutionary dynamics; if you do not want tree sequence recording to be enabled, do not call this function. Note that tree-sequence recording internally uses SLiM's "pedigree tracking" feature to uniquely identify individuals and genomes; however, if you want to use pedigree tracking in your script you must still enable it yourself with `initializeSLiMOptions(keepPedigrees=T)`.

The `recordMutations` flag controls whether information about individual mutations is recorded or not. Such recording takes time and memory, and so can be turned off if only the tree sequence itself is needed, but it is turned on by default since mutation recording is generally useful.

The `simplificationRatio` and `simplificationInterval` parameters control how often automatic simplification of the recorded tree sequence occurs. This is a speed–memory tradeoff: more frequent simplification (lower `simplificationRatio` or smaller `simplificationInterval`) means the stored tree sequences will use less memory, but at a cost of somewhat longer run times. Conversely, a larger `simplificationRatio` or `simplificationInterval` means that SLiM will wait longer between simplifications. There are three ways these parameters can be used. With the first option, with a non-NULL `simplificationRatio` and a NULL value for `simplificationInterval`, SLiM will try to find an optimal tick interval for simplification such that the ratio of the memory used by the tree sequence tables, (before:after) simplification, is close to the requested ratio. The default of 10 (used if both `simplificationRatio` and `simplificationInterval` are NULL) thus requests that SLiM try to find a tick interval such that the maximum size of the stored tree sequences is ten times the size after simplification. INF may be supplied to indicate that automatic simplification should never occur; 0 may be supplied to indicate that automatic simplification should be performed at the end of every tick. Alternatively – the second option – `simplificationRatio` may be NULL and `simplificationInterval` may be set to the interval, in ticks, between simplifications. This may

provide more reliable performance, but the interval must be chosen carefully to avoid exceeding the available memory. The `simplificationInterval` value may be a very large number to specify that simplification should never occur (not `INF`, though, since it is an integer value), or 1 to simplify every tick. Finally – the third option – both parameters may be non-NULL, in which case `simplificationRatio` is used as described above, while `simplificationInterval` provides the *initial* interval first used by SLiM (and then subsequently increased or decreased to try to match the requested simplification ratio). The default initial interval, used when `simplificationInterval` is NULL, is usually 20; this is chosen to be relatively frequent, and thus unlikely to lead to a memory overflow, but it can result in rather slow spool-up for models where the equilibrium simplification interval, as determined by the simplification ratio, is much longer. It can therefore be helpful to set a larger initial interval so that the early part of the model run is not excessively bogged down in simplification.

The `checkCoalescence` parameter controls whether a check for full coalescence is conducted after each simplification. If a model will call `treeSeqCoalesced()` to check for coalescence during its execution, `checkCoalescence` should be set to T. Since the coalescence checks entail a performance penalty, the default of F is preferable otherwise. See the documentation for `treeSeqCoalesced()` for further discussion.

The `runCrosschecks` parameter controls whether cross-checks between SLiM's internal data structures and the tree-sequence recording data structures will be conducted. These two sets of data structures record much the same thing (mutations in genomes), but using completely different representations, so such cross-checks can be useful to confirm that the two data structures do indeed represent the same conceptual state. This slows down the model considerably, however, and would normally be turned on only for debugging purposes, so it is turned off by default.

The `retainCoalescentOnly` parameter controls how, exactly, simplification of the tree-sequence data is performed in SLiM (both for auto-simplification and for calls to `treeSeqSimplify()`). More specifically, this parameter controls the behavior of simplification for individuals and genomes that have been “retained” by calling `treeSeqRememberIndividuals()` with the parameter `permanent=F`. The default of `retainCoalescentOnly=T` helps to keep the number of retained individuals relatively small, which is helpful if your simulation regularly flags many individuals for retaining. In this case, changing `retainCoalescentOnly` to F may dramatically increase memory usage and runtime, in a similar way to permanently remembering all the individuals. See the documentation of `treeSeqRememberIndividuals()` for further discussion (section 24.14.2).

The `timeUnit` parameter controls the time unit stated in the tree sequence when it is saved (which can be accessed through `tskit` APIs); it has no effect on the running simulation whatsoever. The default value, NULL, means that a time unit of “ticks” will be used for all model types. (In SLiM 3.7 / 3.7.1, NULL implied a time unit of “generations” for WF models, but “ticks” for nonWF models; given the new multispecies timescale parameters in SLiM 4, a default of “ticks” makes sense in all cases since now even in WF models one tick might not equal one biological generation.) It may be helpful to set `timeUnit` to “generations” explicitly when modeling non-overlapping generations in which one tick equals one generation, to tell `tskit` that the time unit does in fact represent biological generations; doing so may avoid warnings from `tskit` or `msprime` regarding the time unit, in cases such as recapitation where the simulation timescale is important.

Once all `initialize()` callbacks have executed, in the order in which they are specified in the SLiM input file, the simulation will begin. The tick number at which it starts is determined by the Eidos events you have defined (see section 25.1); the first tick in which an Eidos event is scheduled to execute is the tick at which the simulation starts. Similarly, the simulation will terminate after the last tick for which a script block (either an event or a callback) is registered to execute, unless the `stop()` function or the `simulationFinished()` method of `Community` or `Species` are called to end the simulation earlier.



## 24.2 Class Chromosome

*Superclass: Dictionary*

This class represents the layout and properties of the chromosome being simulated. The chromosome currently being simulated is available through the `sim.chromosome` global. Section 1.5.4 presents an overview of the conceptual role of this class.

### 24.2.1 Chromosome properties

`colorSubstitution <=> (string$)`

The color used to display substitutions in SLiMgui when both mutations and substitutions are being displayed in the chromosome view. Outside of SLiMgui, this property still exists, but is not used by SLiM. Colors may be specified by name, or with hexadecimal RGB values of the form `"#RRGGBB"` (see the Eidos manual). If `colorSubstitution` is the empty string, `" "`, SLiMgui will defer to the color scheme of each `MutationType`, just as it does when only substitutions are being displayed. The default, `"3333FF"`, causes all substitutions to be shown as dark blue when displayed in conjunction with mutations, to prevent the view from becoming too noisy. Note that when substitutions are displayed without mutations also being displayed, this value is ignored by SLiMgui and the substitutions use the color scheme of each `MutationType`.

`geneConversionEnabled => (logical$)`

When gene conversion has been enabled by calling `initializeGeneConversion()`, switching to the DSB recombination model, this property is `T`; otherwise, when using the crossover breakpoints model, it is `F`.

`geneConversionGCBias => (float$)`

The gene conversion bias coefficient, which expresses a bias in the resolution of heteroduplex mismatches in complex gene conversion tracts. When gene conversion has not been enabled by calling `initializeGeneConversion()`, this property will be unavailable.

`geneConversionNonCrossoverFraction => (float$)`

The fraction of double-stranded breaks that result in non-crossover events. When gene conversion has not been enabled by calling `initializeGeneConversion()`, this property will be unavailable.

`geneConversionMeanLength => (float$)`

The mean length of a gene conversion tract (in base positions). When gene conversion has not been enabled by calling `initializeGeneConversion()`, this property will be unavailable.

`geneConversionSimpleConversionFraction => (float$)`

The fraction of gene conversion tracts that are “simple” (i.e., not involving resolution of heteroduplex mismatches); the remainder will be “complex”. When gene conversion has not been enabled by calling `initializeGeneConversion()`, this property will be unavailable.

`genomicElements => (object<GenomicElement>)`

All of the `GenomicElement` objects that comprise the chromosome.

`hotspotEndPositions => (integer)`

The end positions for hotspot map regions along the chromosome. Each hotspot map region is assumed to start at the position following the end of the previous hotspot map region; in other words, the regions are assumed to be contiguous. When using sex-specific hotspot maps, this property will be unavailable; see `hotspotEndPositionsF` and `hotspotEndPositionsM`.

`hotspotEndPositionsF => (integer)`

The end positions for hotspot map regions for females, when using sex-specific hotspot maps; unavailable otherwise. See `hotspotEndPositions` for further explanation.

**hotspotEndPositionsM => (integer)**

The end positions for hotspot map regions for males, when using sex-specific hotspot maps; unavailable otherwise. See `hotspotEndPositions` for further explanation.

**hotspotMultipliers => (float)**

The hotspot multiplier for each of the hotspot map regions specified by `hotspotEndPositions`. When using sex-specific hotspot maps, this property will be unavailable; see `hotspotMultipliersF` and `hotspotMultipliersM`.

**hotspotMultipliersF => (float)**

The hotspot multiplier for each of the hotspot map regions specified by `hotspotEndPositionsF`, when using sex-specific hotspot maps; unavailable otherwise.

**hotspotMultipliersM => (float)**

The hotspot multiplier for each of the hotspot map regions specified by `hotspotEndPositionsM`, when using sex-specific hotspot maps; unavailable otherwise.

**lastPosition => (integer\$)**

The last valid position in the chromosome; its length, essentially. Note that the chromosome length is determined by the *maximum* of the end of the last genomic element, the end of the last recombination region, and the end of the last mutation map region (or hotspot map region).

**mutationEndPositions => (integer)**

The end positions for mutation rate regions along the chromosome. Each mutation rate region is assumed to start at the position following the end of the previous mutation rate region; in other words, the regions are assumed to be contiguous. When using sex-specific mutation rate maps, this property will be unavailable; see `mutationEndPositionsF` and `mutationEndPositionsM`.

This property is unavailable in nucleotide-based models.

**mutationEndPositionsF => (integer)**

The end positions for mutation rate regions for females, when using sex-specific mutation rate maps; unavailable otherwise. See `mutationEndPositions` for further explanation.

This property is unavailable in nucleotide-based models.

**mutationEndPositionsM => (integer)**

The end positions for mutation rate regions for males, when using sex-specific mutation rate maps; unavailable otherwise. See `mutationEndPositions` for further explanation.

This property is unavailable in nucleotide-based models.

**mutationRates => (float)**

The mutation rate for each of the mutation rate regions specified by `mutationEndPositions`. When using sex-specific mutation rate maps, this property will be unavailable; see `mutationRatesF` and `mutationRatesM`.

This property is unavailable in nucleotide-based models.

**mutationRatesF => (float)**

The mutation rate for each of the mutation rate regions specified by `mutationEndPositionsF`, when using sex-specific mutation rate maps; unavailable otherwise.

This property is unavailable in nucleotide-based models.

**mutationRatesM => (float)**

The mutation rate for each of the mutation rate regions specified by `mutationEndPositionsM`, when using sex-specific mutation rate maps; unavailable otherwise.

This property is unavailable in nucleotide-based models.

**overallMutationRate** => (float\$)

The overall mutation rate across the whole chromosome determining the overall number of mutation events that will occur anywhere in the chromosome, as calculated from the individual mutation ranges and rates as well as the coverage of the chromosome by genomic elements (since mutations are only generated within genomic elements, regardless of the mutation rate map). When using sex-specific mutation rate maps, this property will be unavailable; see **overallMutationRateF** and **overallMutationRateM**.

This property is unavailable in nucleotide-based models.

**overallMutationRateF** => (float\$)

The overall mutation rate for females, when using sex-specific mutation rate maps; unavailable otherwise. See **overallMutationRate** for further explanation.

This property is unavailable in nucleotide-based models.

**overallMutationRateM** => (float\$)

The overall mutation rate for males, when using sex-specific mutation rate maps; unavailable otherwise. See **overallMutationRate** for further explanation.

This property is unavailable in nucleotide-based models.

**overallRecombinationRate** => (float\$)

The overall recombination rate across the whole chromosome determining the overall number of recombination events that will occur anywhere in the chromosome, as calculated from the individual recombination ranges and rates. When using sex-specific recombination maps, this property will be unavailable; see **overallRecombinationRateF** and **overallRecombinationRateM**.

**overallRecombinationRateF** => (float\$)

The overall recombination rate for females, when using sex-specific recombination maps; unavailable otherwise. See **overallRecombinationRate** for further explanation.

**overallRecombinationRateM** => (float\$)

The overall recombination rate for males, when using sex-specific recombination maps; unavailable otherwise. See **overallRecombinationRate** for further explanation.

**recombinationEndPositions** => (integer)

The end positions for recombination regions along the chromosome. Each recombination region is assumed to start at the position following the end of the previous recombination region; in other words, the regions are assumed to be contiguous. When using sex-specific recombination maps, this property will be unavailable; see **recombinationEndPositionsF** and **recombinationEndPositionsM**.

**recombinationEndPositionsF** => (integer)

The end positions for recombination regions for females, when using sex-specific recombination maps; unavailable otherwise. See **recombinationEndPositions** for further explanation.

**recombinationEndPositionsM** => (integer)

The end positions for recombination regions for males, when using sex-specific recombination maps; unavailable otherwise. See **recombinationEndPositions** for further explanation.

**recombinationRates** => (float)

The recombination rate for each of the recombination regions specified by **recombinationEndPositions**. When using sex-specific recombination maps, this property will be unavailable; see **recombinationRatesF** and **recombinationRatesM**.

`recombinationRatesF => (float)`

The recombination rate for each of the recombination regions specified by `recombinationEndPositionsF`, when using sex-specific recombination maps; unavailable otherwise.

`recombinationRatesM => (float)`

The recombination rate for each of the recombination regions specified by `recombinationEndPositionsM`, when using sex-specific recombination maps; unavailable otherwise.

`species => (object<Species> $)`

The species to which the target object belongs.

`tag <=> (integer $)`

A user-defined integer value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use.

#### 24.2.2 Chromosome methods

– `(is)ancestralNucleotides([Ni$ start = NULL], [Ni$ end = NULL], [s$ format = "string"])`

Returns the ancestral nucleotide sequence originally supplied to `initializeAncestralNucleotides()`, including any sequence changes due to nucleotide mutations that have fixed and substituted. This nucleotide sequence is the reference sequence for positions in a genome that do not contain a nucleotide-based mutation. The range of the returned sequence may be constrained by a start position given in `start` and/or an end position given in `end`; nucleotides will be returned from `start` to `end`, inclusive. The default value of `NULL` for `start` and `end` represent the first and last base positions of the chromosome, respectively.

The format of the returned sequence is controlled by the `format` parameter. A format of `"string"` will return the sequence as a singleton string (e.g., `"TATA"`). A format of `"char"` will return a string vector with one element per nucleotide (e.g., `"T", "A", "T", "A"`). A format of `"integer"` will return an integer vector with values A=0, C=1, G=2, T=3 (e.g., `3, 0, 3, 0`). If the sequence returned is likely to be long, the `"string"` format will be the most memory-efficient, and may also be the fastest (but may be harder to work with).

For purposes related to interpreting the nucleotide sequence as a coding sequence, a format of `"codon"` is also supported. This format will return an integer vector with values from 0 to 63, based upon successive nucleotide triplets in the sequence (which, for this format, must have a length that is a multiple of three). The codon value for a given nucleotide triplet XYZ is  $16X + 4Y + Z$ , where X, Y, and Z have the usual values A=0, C=1, G=2, T=3. For example, the triplet AAA has a codon value of 0, AAC is 1, AAG is 2, AAT is 3, ACA is 4, and on upward to TTT which is 63. If the nucleotide sequence AACACATTT is requested in codon format, the codon vector `1 4 63` will therefore be returned. These codon values can be useful in themselves; they can also be passed to `codonsToAminoAcids()` to translate them into the corresponding amino acid sequence if desired (see section 24.17.1).

– `(integer)drawBreakpoints([No<Individual>$ parent = NULL], [Ni$ n = NULL])`

Draw recombination breakpoints, using the chromosome's recombination rate map, the current gene conversion parameters, and (in some cases – see below) any active and applicable `recombination()` callbacks. The number of breakpoints to generate, `n`, may be supplied; if it is `NULL` (the default), the number of breakpoints will be drawn based upon the overall recombination rate and the chromosome length (following the standard procedure in SLiM). Note that if the double-stranded breaks model has been chosen, the number of breakpoints generated will probably not be equal to the number

requested, because most breakpoints will entail gene conversion tracts, which entail additional crossover breakpoints.

It is generally recommended that the parent individual be supplied to this method, but `parent` is `NULL` by default. The individual supplied in `parent` is used for two purposes. First, in sexual models that define separate recombination rate maps for males versus females, the sex of `parent` will be used to determine which map is used; in this case, a non-`NULL` value *must* be supplied for `parent`, since the choice of recombination rate map must be determined. Second, in models that define `recombination()` callbacks, `parent` is used to determine the various pseudo-parameters that are passed to `recombination()` callbacks (`individual`, `genome1`, `genome2`, `subpop`), and the subpopulation to which `parent` belongs is used to select which `recombination()` callbacks are applicable; given the necessity of this information, `recombination()` callbacks will not be called as a side effect of this method if `parent` is `NULL`. Apart from these two uses, `parent` is not used, and the caller does not guarantee that the generated breakpoints will actually be used to recombine the genomes of `parent` in particular. If a `recombination()` callback is called, `genome1` for that callback will always be `parent.genome1`; in other words, `drawBreakpoints()` will always treat `parent.genome1` as the initial copy strand. If the caller wishes to randomly choose an initial copy strand (which is usually desirable), they should do that themselves.

– (integer\$)setAncestralNucleotides(is sequence)

This method, which may be called only in nucleotide-based models (see section 1.8), replaces the ancestral nucleotide sequence for the model. The `sequence` parameter is interpreted exactly as it is in the `initializeAncestralSequence()` function; see that documentation for details (section 24.1). The length of the ancestral sequence is returned.

It is unusual to replace the ancestral sequence in a running simulation, since the nucleotide states of segregating and fixed mutations will depend upon the original ancestral sequence. It can be useful when loading a new population state with `readFromMS()` or `readFromVCF()`, such as when resetting the simulation state to an earlier state in a conditional simulation; however, that is more commonly done using `readFromPopulationFile()` with a SLiM or .trees file.

– (void)setGeneConversion(numeric\$ nonCrossoverFraction, numeric\$ meanLength, numeric\$ simpleConversionFraction, [numeric\$ bias = 0])

This method switches the recombination model to the “double-stranded break (DSB)” model (if it is not already set to that), and configures the details of the gene conversion tracts that will therefore be modeled (see section 1.5.6 for discussion of the “DSB” recombination model). The meanings and effects of the parameters exactly mirror the `initializeGeneConversion()` function; see section 24.1 for details.

– (void)setHotspotMap(numeric multipliers, [Ni ends = NULL], [string\$ sex = "\*"])

In nucleotide-based models, set the mutation rate *multiplier* along the chromosome. There are two ways to call this method. If the optional `ends` parameter is `NULL` (the default), then `multipliers` must be a singleton value that specifies a single multiplier to be used along the entire chromosome. If, on the other hand, `ends` is supplied, then `multipliers` and `ends` must be the same length, and the values in `ends` must be specified in ascending order. In that case, `multipliers` and `ends` taken together specify the multipliers to be used along successive contiguous stretches of the chromosome, from beginning to end; the last position specified in `ends` should extend to the end of the chromosome (as previously determined, during simulation initialization). See the `initializeHotspotMap()` function for further discussion of precisely how these multipliers and positions are interpreted.

If the optional `sex` parameter is `"*"` (the default), then the supplied hotspot map will be used for both sexes (which is the only option for hermaphroditic simulations). In sexual simulations `sex` may be `"M"` or `"F"` instead, in which case the supplied hotspot map is used only for that sex. Note that whether sex-specific hotspot maps will be used is set by the way that the simulation is initially configured with `initializeHotspot()`, and cannot be changed with this method; so if the

simulation was set up to use sex-specific hotspot maps then sex must be "M" or "F" here, whereas if it was set up not to, then sex must be "\*" or unsupplied here. If a simulation needs sex-specific hotspot maps only some of the time, the male and female maps can simply be set to be identical the rest of the time.

The hotspot map is normally constant in simulations, so be sure you know what you are doing.

- (void)setMutationRate(numeric rates, [Ni ends = NULL], [string\$ sex = "\*"])

Set the mutation rate per base position per gamete. There are two ways to call this method. If the optional ends parameter is NULL (the default), then rates must be a singleton value that specifies a single mutation rate to be used along the entire chromosome. If, on the other hand, ends is supplied, then rates and ends must be the same length, and the values in ends must be specified in ascending order. In that case, rates and ends taken together specify the mutation rates to be used along successive contiguous stretches of the chromosome, from beginning to end; the last position specified in ends should extend to the end of the chromosome (as previously determined, during simulation initialization). See the initializeMutationRate() function for further discussion of precisely how these rates and positions are interpreted.

If the optional sex parameter is "\*" (the default), then the supplied mutation rate map will be used for both sexes (which is the only option for hermaphroditic simulations). In sexual simulations sex may be "M" or "F" instead, in which case the supplied mutation rate map is used only for that sex. Note that whether sex-specific mutation rate maps will be used is set by the way that the simulation is initially configured with initializeMutationRate(), and cannot be changed with this method; so if the simulation was set up to use sex-specific mutation rate maps then sex must be "M" or "F" here, whereas if it was set up not to, then sex must be "\*" or unsupplied here. If a simulation needs sex-specific mutation rate maps only some of the time, the male and female maps can simply be set to be identical the rest of the time.

The mutation rate intervals are normally a constant in simulations, so be sure you know what you are doing.

In nucleotide-based models, setMutationRate() may not be called. If variation in the mutation rate along the chromosome is desired, setHotspotMap() should be used.

- (void)setRecombinationRate(numeric rates, [Ni ends = NULL], [string\$ sex = "\*"])

Set the recombination rate per base position per gamete. All rates must be in the interval [0.0, 0.5]. There are two ways to call this method. If the optional ends parameter is NULL (the default), then rates must be a singleton value that specifies a single recombination rate to be used along the entire chromosome. If, on the other hand, ends is supplied, then rates and ends must be the same length, and the values in ends must be specified in ascending order. In that case, rates and ends taken together specify the recombination rates to be used along successive contiguous stretches of the chromosome, from beginning to end; the last position specified in ends should extend to the end of the chromosome (as previously determined, during simulation initialization). See the initializeRecombinationRate() function for further discussion of precisely how these rates and positions are interpreted.

If the optional sex parameter is "\*" (the default), then the supplied recombination rate map will be used for both sexes (which is the only option for hermaphroditic simulations). In sexual simulations sex may be "M" or "F" instead, in which case the supplied recombination map is used only for that sex. Note that whether sex-specific recombination maps will be used is set by the way that the simulation is initially configured with initializeRecombinationRate(), and cannot be changed with this method; so if the simulation was set up to use sex-specific recombination maps then sex must be "M" or "F" here, whereas if it was set up not to, then sex must be "\*" or unsupplied here. If a simulation needs sex-specific recombination maps only some of the time, the male and female maps can simply be set to be identical the rest of the time.

The recombination intervals are normally a constant in simulations, so be sure you know what you are doing.



## 24.3 Class Community

*Superclass: Dictionary*

This class represents the top level of a SLiM simulation: a community containing one or more species (represented by class `Species`). The community, represented by a global `Community` object with the variable name `community`, is responsible for execution of the tick cycle for active species in each tick, management of the script blocks associated with a simulation, and creation of log files (by creating and owning instances of the `LogFile` class). The community exists even in single-species models.

### 24.3.1 Community properties

`allGenomicElementTypes => (object<GenomicElementType>)`

All of the `GenomicElementType` objects defined in the simulation.

`allInteractionTypes => (object<InteractionType>)`

All of the `InteractionType` objects defined in the simulation.

`allMutationTypes => (object<MutationType>)`

All of the `MutationType` objects defined in the simulation.

`allScriptBlocks => (object<SLiMEidosBlock>)`

All registered `SLiMEidosBlock` objects in the simulation.

`allSpecies => (object<Species>)`

All of the `Species` objects defined in the simulation (in species declaration order).

`allSubpopulations => (object<Subpopulation>)`

All of the `Subpopulation` objects defined in the simulation.

`cycleStage => (string$)`

The current cycle stage, as a `string`. The values of this property essentially mirror the cycle stages of WF and nonWF models (see chapters 22 and 23). Common values include `"first"` (during execution of `first()` events), `"early"` (during execution of `early()` events), `"reproduction"` (during offspring generation), `"fitness"` (during fitness evaluation), `"survival"` (while applying selection and mortality in nonWF models), and `"late"` (during execution of `late()` events).

Other possible values include `"begin"` (during internal setup before each cycle), `"tally"` (while tallying mutation reference counts and removing fixed mutations), `"swap"` (while swapping the offspring generation into the parental generation in WF models), `"end"` (during internal bookkeeping after each cycle), and `"console"` (during the in-between-ticks state in which commands in SLiMgui's Eidos console are executed). It would probably be a good idea not to use this latter set of values; they are probably not user-visible during ordinary model execution anyway.

During execution of `initialize()` callbacks, no `Community` object yet exists and so this property cannot be accessed. To detect this state, use `exists("community")`; if that is F, `community` does not exist, and therefore your code is executing during `initialize()` callbacks (or outside of SLiM entirely, in some other Eidos-based context).

`logFiles => (object<LogFile>)`

The `LogFile` objects being used in the simulation.

`modelType => (string$)`

The type of model being simulated, as specified in `initializeSLiMModelType()`. This will be `"WF"` for WF models (Wright–Fisher models, the default), or `"nonWF"` for nonWF models (non-Wright–Fisher

models; see section 1.6 for discussion). This must be the same for all species in the community; it is therefore a property on `Community`, not `Species`.

`tag <=> (integer$)`

A user-defined `integer` value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the `Dictionary` class; see the Eidos manual), for another way of attaching state to the simulation.

`tick <=> (integer$)`

The current tick number.

`verbosity <=> (integer$)`

The verbosity level, for SLiM's logging of information about the simulation. This is 1 by default, but can be changed at the command line with the `-l[ong]` option. It is provided here so that scripts can consult it to govern the level of verbosity of their own output, or set the verbosity level for particular sections of their code. A verbosity level of 0 suppresses most of SLiM's optional output; 2 adds some extra output beyond SLiM's standard output. See sections 20.3 and 21.4 for more information.

### 24.3.2 *Community methods*

- `(object<LogFile>$)createLogFile(string$ filePath, [Ns initialContents = NULL], [logical$ append = F], [logical$ compress = F], [string$ sep = ","], [Ni$ logInterval = NULL], [Ni$ flushInterval = NULL])`

Creates and returns a new `LogFile` object that logs data from the simulation (see the documentation for the `LogFile` class for details). Logged data will be written to the file at `filePath`, overwriting any existing file at that path by default, or appending to it instead if `append` is `T` (successive rows of the log table will always be appended to the previously written content, of course). Before the header line for the log is written out, any `string` elements in `initialContents` will be written first, separated by newlines, allowing for a user-defined file header. If `compress` is `T`, the contents will be compressed with `zlib` as they are written, and the standard `.gz` extension for gzip-compressed files will be appended to the filename in `filePath` if it is not already present.

The `sep` parameter specifies the separator between data values within a row. The default of `","` will generate a “comma-separated value” (CSV) file, while passing `sep="\t"` will use a tab separator instead to generate a “tab-separated value” (TSV) file. Other values for `sep` may also be used, but are less standard.

`LogFile` supports periodic automatic logging of a new row of data, enabled by supplying a non-NULL value for `logInterval`. In this case, a new row will be logged (as if `logRow()` were called on the `LogFile`) at the end of every `logInterval` ticks (just before the tick counter increments, in both WF and nonWF models), starting at the end of the tick in which the `LogFile` was created. A `logInterval` of 1 will cause automatic logging at the end of every tick, whereas a `logInterval` of NULL disables automatic logging. Automatic logging can always be disabled or reconfigured later with the `LogFile` method `setLogInterval()`, or logging can be triggered manually by calling `logRow()`.

When compression is enabled, `LogFile` flushes new data lazily by default, for performance reasons, buffering data for multiple rows before writing to disk. Passing a non-NULL value for `flushInterval` requests a flush every `flushInterval` rows (with a value of 1 providing unbuffered operation). Note that flushing very frequently will likely result in both lower performance and a larger final file size (in one simple test, 48943 bytes instead of 4280 bytes, or more than a 10x increase in size). Alternatively, passing a very large value for `flushInterval` will effectively disable automatic flushing, except at the end of the simulation (but be aware that this may use a large amount of memory for large log files). In any case, the log file will be created immediately, with its requested initial contents; the initial write is not buffered. When compression is not enabled, the `flushInterval` setting is ignored.

The `LogFile` documentation discusses how to configure and use `LogFile` to write out the data you are interested in from your simulation; see section 24.10.

- `(void)deregisterScriptBlock(io<SLiMEidosBlock> scriptBlocks)`  
All `SLiMEidosBlock` objects specified by `scriptBlocks` (either with `SLiMEidosBlock` objects or with integer identifiers) will be scheduled for deregistration. The deregistered blocks remain valid, and may even still be executed in the current stage of the current tick (see section 25.11); the blocks are not actually deregistered and deallocated until sometime after the currently executing script block has completed. To immediately prevent a script block from executing, even when it is scheduled to execute in the current stage of the current tick, use the `active` property of the script block (see sections 24.12.1 and 25.11).
- `(object<GenomicElementType>)genomicElementTypesWithIDs(integer ids)`  
Find and return the `GenomicElementType` objects with `id` values matching the values in `ids`. If no matching `GenomicElementType` object can be found with a given `id`, an error results.
- `(object<InteractionType>)interactionTypesWithIDs(integer ids)`  
Find and return the `InteractionType` objects with `id` values matching the values in `ids`. If no matching `InteractionType` object can be found with a given `id`, an error results.
- `(object<MutationType>)mutationTypesWithIDs(integer ids)`  
Find and return the `MutationType` objects with `id` values matching the values in `ids`. If no matching `MutationType` object can be found with a given `id`, an error results.
- `(void)outputUsage(void)`  
Output the current memory usage of the simulation to Eidos's output stream. The specifics of what is printed, and in what format, should not be relied upon as they may change from version to version of SLiM. This method is primarily useful for understanding where the memory usage of a simulation predominantly resides, for debugging or optimization. Note that it does not capture *all* memory usage by the process; rather, it summarizes the memory usage by SLiM and Eidos in directly allocated objects and buffers. To get the *total* memory usage of the running process (either current or peak), use the Eidos function `usage()`.
- `(object<SLiMEidosBlock>)$registerEarlyEvent(Nis$ id, string$ source, [Ni$ start = NULL], [Ni$ end = NULL], [No<Species>$ ticksSpec = NULL])`  
Register a block of Eidos source code, represented as the string singleton `source`, as an Eidos `early()` event in the current simulation, with optional `start` and `end` ticks (and, for multispecies models, optional ticks specifier `ticksSpec`) limiting its applicability. The script block will be given identifier `id` (specified as an integer, or as a string symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered event is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.
- `(object<SLiMEidosBlock>)$registerFirstEvent(Nis$ id, string$ source, [Ni$ start = NULL], [Ni$ end = NULL], [No<Species>$ ticksSpec = NULL])`  
Register a block of Eidos source code, represented as the string singleton `source`, as an Eidos `first()` event in the current simulation, with optional `start` and `end` ticks (and, for multispecies models, optional ticks specifier `ticksSpec`) limiting its applicability. The script block will be given identifier `id` (specified as an integer, or as a string symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered event is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerInteractionCallback(Nis\$ id, string\$ source, io<InteractionType>\$ intType, [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the string singleton `source`, as an Eidos `interaction()` callback in the current simulation (global to the community), with a required interaction type `intType` (which may be an integer identifier), optional exorter subpopulation `subpop` (which may also be an integer identifier, or `NULL`, the default, to indicate all subpopulations), and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an integer, or as a string symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it will be eligible to execute the next time an `InteractionType` is evaluated. The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerLateEvent(Nis\$ id, string\$ source, [Ni\$ start = NULL], [Ni\$ end = NULL], [No<Species>\$ ticksSpec = NULL])

Register a block of Eidos source code, represented as the string singleton `source`, as an Eidos `late()` event in the current simulation, with optional `start` and `end` ticks (and, for multispecies models, optional ticks specifier `ticksSpec`) limiting its applicability. The script block will be given identifier `id` (specified as an integer, or as a string symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered event is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it may be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>)rescheduleScriptBlock(io<SLiMEidosBlock>\$ block, [Ni\$ start = NULL], [Ni\$ end = NULL], [Ni ticks = NULL])

Reschedule the target script block given by `block` to execute in a specified set of ticks. The `block` parameter may be either an integer representing the ID of the desired script block, or a `SLiMScriptBlock` specified directly.

The first way to specify the tick set is with `start` and `end` parameter values; `block` will then execute from `start` to `end`, inclusive. In this case, `block` is returned.

The second way to specify the tick set is using the `ticks` parameter; this is more flexible but more complicated. Since script blocks execute across a contiguous span of ticks defined by their `start` and `end` properties, this may result in the duplication of `block`; one script block will be used for each contiguous span of ticks in `ticks`. The `block` object itself will be rescheduled to cover the first such span, whereas duplicates of `block` will be created to cover subsequent contiguous spans. A vector containing all of the script blocks scheduled by this method, including `block`, will be returned; this vector is guaranteed to be sorted by the (ascending) scheduled execution order of the blocks. Any duplicates of `block` created will be given values for the `active`, `source`, `tag`, and `type` properties equal to the current values for `block`, but will be given an `id` of -1 since script block identifiers must be unique; if it is necessary to find the duplicated blocks again later, their `tag` property should be used. The vector supplied for `ticks` does not need to be in sorted order, but it must not contain any duplicates.

Because this method can create a large number of duplicate script blocks, it can sometimes be better to handle script block scheduling in other ways. If an `early()` event needs to execute every tenth tick over the whole duration of a long model run, for example, it would not be advisable to use a call like `community.rescheduleScriptBlock(s1, ticks=seq(10, 100000, 10))` for that purpose, since that would result in thousands of duplicate script blocks. Instead, it would be preferable to add a test such as `if (community.tick % 10 != 0) return;` at the beginning of the event. It is legal to reschedule a script block while the block is executing; a call like `community.rescheduleScriptBlock(self, community.tick + 10, community.tick + 10);`

made inside a given block would therefore also cause the block to execute every tenth tick, although this sort of self-rescheduling code is probably harder to read, maintain, and debug.

Whichever way of specifying the tick set is used, the discussion in section 25.11 applies: `block` may continue to be executed during the current tick cycle stage even after it has been rescheduled, unless it is made inactive using its `active` property, and similarly, the block may not execute during the current tick cycle stage if it was not already scheduled to do so. Rescheduling script blocks during the tick and tick cycle stage in which they are executing, or in which they are intended to execute, should be avoided. Also, as mentioned in section 22.7, script blocks which are open-ended (i.e., with no specified end tick), are not used in determining whether the end of the simulation has been reached (because then the simulation would run forever); if you reschedule a block to be open-ended, and to start after the end of the last closed-ended block, the rescheduled block will therefore not run at all (just as such a block would not run at all in other circumstances, too).

Note that new script blocks can also be created and scheduled using the `register...()` methods of `Community` and `Species`; by using the same source as a template script block, the template can be duplicated and scheduled for different ticks. In fact, `rescheduleScriptBlock()` does essentially that internally. In multispecies models, note that all new script blocks created as a side effect of `rescheduleScriptBlock()` will have the same `species` and `ticks` specifier as `block`; use the `register...()` methods to create a new block with a different `species` or `ticks` specifier.

- `(object<SLiMEidosBlock>)scriptBlocksWithIDs(integer ids)`  
Find and return the `SLiMEidosBlock` objects with `id` values matching the values in `ids`. If no matching `SLiMEidosBlock` object can be found with a given `id`, an error results.
- `(void)simulationFinished(void)`  
Declare the current simulation finished. Normally SLiM ends a simulation when, at the end of a tick, there are no script events or callbacks registered for any future tick (excluding scripts with no declared end tick). If you wish to end a simulation before this condition is met, a call to `simulationFinished()` will cause the current simulation to end at the end of the current tick. For example, a simulation might self-terminate if a test for a dynamic equilibrium condition is satisfied. Note that the current tick will finish executing; if you want the simulation to stop immediately, you can use the Eidos method `stop()`, which raises an error condition.
- `(object<Species>)speciesWithIDs(integer ids)`  
Find and return the `Species` objects with `id` values matching the values in `ids`. If no matching `Species` object can be found with a given `id`, an error results.
- `(object<Subpopulation>)subpopulationsWithIDs(integer ids)`  
Find and return the `Subpopulation` objects with `id` values matching the values in `ids`. If no matching `Subpopulation` object can be found with a given `id`, an error results.

## 24.4 Class Genome

*Superclass: Object*

This class represents one full genome of an individual (one of the two genomes contained by a diploid individual, that is, in the way that SLiM uses the term), composed of the mutations carried by that individual. Section 1.5.1 presents an overview of the conceptual role of this class.

### 24.4.1 Genome properties

`genomePedigreeID => (integer$)`

If pedigree tracking is turned on with `initializeSLiMOptions(keepPedigrees=T)`, `genomePedigreeID` is a unique non-negative identifier for each genome in a simulation, never re-used throughout the duration of the simulation run. Furthermore, the `genomePedigreeID` of a given genome will be equal to either  $(2 * \text{pedigreeID})$  or  $(2 * \text{pedigreeID} + 1)$  of the individual that the



genome belongs to (the former for the first genome of the individual, the latter for the second genome of the individual); this invariant relationship is guaranteed. If pedigree tracking is not enabled, this property is unavailable.

`genomeType => (string$)`

The type of chromosome represented by this genome; one of "A", "X", or "Y".

`individual => (object<Individual>$)`

The `Individual` object to which this genome belongs.

`isNullGenome => (logical$)`

T if the genome is a "null" genome, F if it is an ordinary genome object. When a sex chromosome (X or Y) is simulated, the other sex chromosome also exists in the simulation, but it is a "null" genome that does not carry any mutations. Instead, it is a placeholder, present to allow SLiM's code to operate in much the same way as it does when an autosome is simulated. Null genomes should not be accessed or manipulated.

`mutations => (object<Mutation>)`

All of the `Mutation` objects present in this genome.

`tag <=> (integer$)`

A user-defined `integer` value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use. Note that the `Genome` objects used by SLiM are new with every new individual, so the `tag` value of each new offspring generated in each tick will be initially undefined.

#### 24.4.2 Genome methods

+ `(void)addMutations(object<Mutation> mutations)`

Add the existing mutations in `mutations` to the genome, if they are not already present (if they are already present, they will be ignored), and if the addition is not prevented by the mutation stacking policy (see the `mutationStackPolicy` property of `MutationType`, section 24.11.1).

Calling this will normally affect the fitness values calculated toward the end of the current tick; if you want current fitness values to be affected, you can call the `Species` method `recalculateFitness()` – but see the documentation of that method for caveats.

Note that in nonWF models that use tree-sequence recording, mutations cannot be added to an individual after the tick in which the individual is created (i.e., when the `age` of the individual is greater than 0), to prevent the possibility of inconsistencies in the recorded tree sequence.

+ `(object<Mutation>)addNewDrawnMutation(io<MutationType> mutationType, integer position, [Ni originTick = NULL], [Nio<Subpopulation> originSubpop = NULL], [Nis nucleotide = NULL])`

Add new mutations to the target genome(s) with the specified `mutationType` (specified by the `MutationType` object or by `integer` identifier), `position`, `originTick` (which may be `NULL`, the default, to specify the current tick; otherwise, beginning in SLiM 3.5, it must be equal to the current tick anyway, as other uses of this property have been deprecated), and `originSubpop` (specified by the `Subpopulation` object or by `integer` identifier, or by `NULL`, the default, to specify the subpopulation to which the first target genome belongs). If `originSubpop` is supplied as an `integer`, it is intentionally not checked for validity; you may use arbitrary values of `originSubpop` to "tag" the mutations that you create (see section 24.10.1). The selection coefficients of the mutations are drawn from their mutation types; `addNewMutation()` may be used instead if you wish to specify selection coefficients.

In non-nucleotide-based models, `mutationType` will always be a non-nucleotide-based mutation type, and so `nucleotide` must be `NULL` (the default). In a nucleotide-based model,



`mutationType` might still be non-nucleotide-based (in which case `nucleotide` must still be `NULL`), or `mutationType` might be nucleotide-based, in which case a non-`NULL` value must be supplied for `nucleotide`, specifying the nucleotide(s) to be associated with the new mutation(s). Nucleotides may be specified with string values ("A", "C", "G", or "T"), or with integer values (A=0, C=1, G=2, T=3). If a nucleotide mutation already exists at the mutating position, it is replaced automatically in accordance with the stacking policy for nucleotide-based mutation types. No check is performed that a new mutation's nucleotide differs from the ancestral sequence, or that its selection coefficient is consistent with other mutations that may already exist at the given position with the same nucleotide; model consistency is the responsibility of the model.

Beginning in SLiM 2.5 this method is vectorized, so all of these parameters may be singletons (in which case that single value is used for all mutations created by the call) or non-singleton vectors (in which case one element is used for each corresponding mutation created). Non-singleton parameters must match in length, since their elements need to be matched up one-to-one.

The new mutations created by this method are returned, even if their actual addition is prevented by the mutation stacking policy (see the `mutationStackPolicy` property of `MutationType`, section 24.11.1). However, the order of the mutations in the returned vector is not guaranteed to be the same as the order in which the values are specified in parameter vectors, unless the `position` parameter is specified in ascending order. In other words, pre-sorting the parameters to this method into ascending order by position, using `order()` and subsetting, will guarantee that the order of the returned vector of mutations corresponds to the order of elements in the parameters to this method; otherwise, no such guarantee exists.

Beginning in SLiM 2.1, this is a class method, not an instance method. This means that it does not get multiplexed out to all of the elements of the receiver (which would add a different new mutation to each element); instead, it is performed as a single operation, adding the same new mutation objects to all of the elements of the receiver. Before SLiM 2.1, to add the same mutations to multiple genomes, it was necessary to call `addNewDrawnMutation()` on one of the genomes, and then add the returned `Mutation` object to all of the other genomes using `addMutations()`. That is not necessary in SLiM 2.1 and later, because of this change (although doing it the old way does no harm and produces identical behavior). Pre-2.1 code that actually relied upon the old multiplexing behavior will no longer work correctly (but this is expected to be an extremely rare pattern of usage).

Before SLiM 4, this method also took an `originGeneration` parameter. This was deprecated (the origin generation was then required to be equal to the current generation, for internal consistency), and was removed in SLiM 4.

Calling this will normally affect the fitness values calculated at the end of the current tick (but not sooner); if you want current fitness values to be affected, you can call the `Species` method `recalculateFitness()` – but see the documentation of that method for caveats.

Note that in nonWF models that use tree-sequence recording, mutations cannot be added to an individual after the tick in which the individual is created (i.e., when the `age` of the individual is greater than 0), to prevent the possibility of inconsistencies in the recorded tree sequence.

```
+ (object<Mutation>)addNewMutation(io<MutationType> mutationType,
    numeric selectionCoeff, integer position, [Ni originTick = NULL],
    [Nio<Subpopulation> originSubpop = NULL], [Nis nucleotide = NULL])
```

Add new mutations to the target genome(s) with the specified `mutationType` (specified by the `MutationType` object or by integer identifier), `selectionCoeff`, `position`, `originTick` (which may be `NULL`, the default, to specify the current tick; otherwise, beginning in SLiM 3.5, it must be equal to the current tick anyway, as other uses of this property have been deprecated), and `originSubpop` (specified by the `Subpopulation` object or by integer identifier, or by `NULL`, the default, to specify the subpopulation to which the first target genome belongs). If `originSubpop` is supplied as an integer, it is intentionally not checked

for validity; you may use arbitrary values of `originSubpop` to “tag” the mutations that you create (see section 24.10.1). The `addNewDrawnMutation()` method may be used instead if you wish selection coefficients to be drawn from the mutation types of the mutations.

In non-nucleotide-based models, `mutationType` will always be a non-nucleotide-based mutation type, and so `nucleotide` must be `NULL` (the default). In a nucleotide-based model, `mutationType` might still be non-nucleotide-based (in which case `nucleotide` must still be `NULL`), or `mutationType` might be nucleotide-based, in which case a non-`NULL` value must be supplied for `nucleotide`, specifying the nucleotide(s) to be associated with the new mutation(s). Nucleotides may be specified with string values ("A", "C", "G", or "T"), or with integer values (A=0, C=1, G=2, T=3). If a nucleotide mutation already exists at the mutating position, it is replaced automatically in accordance with the stacking policy for nucleotide-based mutation types. No check is performed that a new mutation's nucleotide differs from the ancestral sequence, or that its selection coefficient is consistent with other mutations that may already exist at the given position with the same nucleotide; model consistency is the responsibility of the model.

The new mutations created by this method are returned, even if their actual addition is prevented by the mutation stacking policy (see the `mutationStackPolicy` property of `MutationType`, section 24.11.1). However, the order of the mutations in the returned vector is not guaranteed to be the same as the order in which the values are specified in parameter vectors, unless the `position` parameter is specified in ascending order. In other words, pre-sorting the parameters to this method into ascending order by position, using `order()` and subsetting, will guarantee that the order of the returned vector of mutations corresponds to the order of elements in the parameters to this method; otherwise, no such guarantee exists.

Beginning in SLiM 2.1, this is a class method, not an instance method. This means that it does not get multiplexed out to all of the elements of the receiver (which would add a different new mutation to each element); instead, it is performed as a single operation, adding the same new mutation object to all of the elements of the receiver. Before SLiM 2.1, to add the same mutation to multiple genomes, it was necessary to call `addNewMutation()` on one of the genomes, and then add the returned `Mutation` object to all of the other genomes using `addMutations()`. That is not necessary in SLiM 2.1 and later, because of this change (although doing it the old way does no harm and produces identical behavior). Pre-2.1 code that actually relied upon the old multiplexing behavior will no longer work correctly (but this is expected to be an extremely rare pattern of usage).

Before SLiM 4, this method also took a `originGeneration` parameter. This was deprecated (the origin generation was then required to be equal to the current generation, for internal consistency), and was removed in SLiM 4.

Calling this will normally affect the fitness values calculated at the end of the current tick (but not sooner); if you want current fitness values to be affected, you can call the `Species` method `recalculateFitness()` – but see the documentation of that method for caveats.

Note that in nonWF models that use tree-sequence recording, mutations cannot be added to an individual after the tick in which the individual is created (i.e., when the `age` of the individual is greater than 0), to prevent the possibility of inconsistencies in the recorded tree sequence.

- `(Nlo<Mutation>$)containsMarkerMutation(io<MutationType>$ mutType, integer$ position, [logical$ returnMutation = F])`

Returns T if the genome contains a mutation of type `mutType` at `position`, F otherwise (if `returnMutation` has its default value of F; see below). This method is, as its name suggests, intended for checking for “marker mutations”: mutations of a special mutation type that are not literally mutations in the usual sense, but instead are added in to particular genomes to mark them as possessing some property. Marker mutations are not typically added by SLiM’s mutation-generating machinery; instead they are added explicitly with `addNewMutation()` or `addNewDrawnMutation()` at a known, constant position in the genome. This method provides a check for whether a marker

mutation of a given type exists in a particular genome; because the position to check is known in advance, that check can be done much faster than the equivalent check with `containsMutations()` or `countOfMutationsOfType()`, using a binary search of the genome. See section 14.4 for one example of a model that uses marker mutations – in that case, to mark chromosomes that possess an inversion.

If `returnMutation` is `T` (an option added in SLiM 3), this method returns the actual mutation found, rather than just `T` or `F`. More specifically, the *first* mutation found of `mutType` at `position` will be returned; if more than one such mutation exists in the target genome, which one is returned is not defined. If `returnMutation` is `T` and no mutation of `mutType` is found at `position`, `NULL` will be returned.

– `(logical)containsMutations(object<Mutation> mutations)`

Returns a `logical` vector indicating whether each of the mutations in `mutations` is present in the genome; each element in the returned vector indicates whether the corresponding mutation is present (`T`) or absent (`F`). This method is provided for speed; it is much faster than the corresponding Eidos code.

– `(integer$)countOfMutationsOfType(io<MutationType>$ mutType)`

Returns the number of mutations that are of the type specified by `mutType`, out of all of the mutations in the genome. If you need a vector of the matching `Mutation` objects, rather than just a count, use `-mutationsOfType()`. This method is provided for speed; it is much faster than the corresponding Eidos code.

+ `(integer)mutationCountsInGenomes([No<Mutation> mutations = NULL])`

Return an `integer` vector with the frequency counts of all of the `Mutation` objects passed in `mutations`, within the target `Genome` vector. If the optional `mutations` argument is `NULL` (the default), frequency counts will be returned for all of the active `Mutation` objects in the simulation – the same `Mutation` objects, and in the same order, as would be returned by the `mutations` property of `sim`, in other words.

See the `+mutationFrequenciesInGenomes()` method to obtain `float` frequencies instead of `integer` counts. See also the `Species` methods `mutationCounts()` and `mutationFrequencies()`, which may be more efficient for getting counts/frequencies for whole subpopulations or for the whole simulation.

+ `(float)mutationFrequenciesInGenomes([No<Mutation> mutations = NULL])`

Return a `float` vector with the frequencies of all of the `Mutation` objects passed in `mutations`, within the target `Genome` vector. If the optional `mutations` argument is `NULL` (the default), frequencies will be returned for all of the active `Mutation` objects in the simulation – the same `Mutation` objects, and in the same order, as would be returned by the `mutations` property of `sim`, in other words.

See the `+mutationCountsInGenomes()` method to obtain `integer` counts instead of `float` frequencies. See also the `Species` methods `mutationCounts()` and `mutationFrequencies()`, which may be more efficient for getting counts/frequencies for whole subpopulations or for the whole simulation.

– `(object<Mutation>)mutationsOfType(io<MutationType>$ mutType)`

Returns an `object` vector of all the mutations that are of the type specified by `mutType`, out of all of the mutations in the genome. If you just need a count of the matching `Mutation` objects, rather than a vector of the matches, use `-countOfMutationsOfType()`; if you need just the positions of matching `Mutation` objects, use `-positionsOfMutationsOfType()`; and if you are aiming for a sum of the selection coefficients of matching `Mutation` objects, use `-sumOfMutationsOfType()`. This method is provided for speed; it is much faster than the corresponding Eidos code.

- (is)nucleotides([Ni\$ start = NULL], [Ni\$ end = NULL],  
[string\$ format = "string"])

Returns the nucleotide sequence for the genome. This is the current ancestral sequence, as would be returned by the `Chromosome` method `ancestralNucleotides()`, with the nucleotides for any nucleotide-based mutations in the genome overlaid. The range of the returned sequence may be constrained by a start position given in `start` and/or an end position given in `end`; nucleotides will be returned from `start` to `end`, inclusive. The default value of `NULL` for `start` and `end` represent the first and last base positions of the chromosome, respectively.

The format of the returned sequence is controlled by the `format` parameter. A format of `"string"` will return the sequence as a singleton string (e.g., `"TATA"`). A format of `"char"` will return a string vector with one element per nucleotide (e.g., `"T", "A", "T", "A"`). A format of `"integer"` will return an integer vector with values A=0, C=1, G=2, T=3 (e.g., `3, 0, 3, 0`). A format of `"codon"` will return an integer vector with values from 0 to 63, based upon successive nucleotide triplets in the sequence (which, for this format, must have a length that is a multiple of three); see the `ancestralNucleotides()` documentation for details. If the sequence returned is likely to be long, the `"string"` format will be the most memory-efficient, and may also be the fastest (but may be harder to work with).

Several helper functions are provided for working with sequences, such as `nucleotideCounts()` to get the counts of A/C/G/T nucleotides in a sequence, `nucleotideFrequencies()` to get the same information as frequencies, and `codonsToAminoAcids()` to convert a codon sequence (such as provided by the codon format described above) to an amino acid sequence; see section 24.17.1.

+ (void)output([Ns\$ filePath = NULL], [logical\$ append = F])

Output the target genomes in SLiM's native format (see section 26.3.1 for output format details). This low-level output method may be used to output any sample of `Genome` objects (the Eidos function `sample()` may be useful for constructing custom samples, as may the SLiM class `Individual`). For output of a sample from a single `Subpopulation`, the `outputSample()` of `Subpopulation` may be more straightforward to use. If the optional parameter `filePath` is `NULL` (the default), output is directed to SLiM's standard output. Otherwise, the output is sent to the file specified by `filePath`, overwriting that file if `append` is `F`, or appending to the end of it if `append` is `T`.

See `outputMS()` and `outputVCF()` for other output formats. Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

+ (void)outputMS([Ns\$ filePath = NULL], [logical\$ append = F],  
[logical\$ filterMonomorphic = F])

Output the target genomes in MS format (see section 26.3.2 for output format details). This low-level output method may be used to output any sample of `Genome` objects (the Eidos function `sample()` may be useful for constructing custom samples, as may the SLiM class `Individual`). For output of a sample from a single `Subpopulation`, the `outputMSSample()` of `Subpopulation` may be more straightforward to use. If the optional parameter `filePath` is `NULL` (the default), output is directed to SLiM's standard output. Otherwise, the output is sent to the file specified by `filePath`, overwriting that file if `append` is `F`, or appending to the end of it if `append` is `T`. Positions in the output will span the interval [0,1].

If `filterMonomorphic` is `F` (the default), all mutations that are present in the sample will be included in the output. This means that some mutations may be included that are actually monomorphic within the sample (i.e., that exist in every sampled genome, and are thus apparently fixed). These may be filtered out with `filterMonomorphic = T` if desired; note that this option means that some mutations that do exist in the sampled genomes might not be included in the output, simply because they exist in every sampled genome.

See `output()` and `outputVCF()` for other output formats. Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

```
+ (void)outputVCF([Ns$ filePath = NULL], [logical$ outputMultiallelics = T],
  [logical$ append = F], [logical$ simplifyNucleotides = F],
  [logical$ outputNonnucleotides = T])
```

Output the target genomes in VCF format (see sections 26.2.3, 26.2.4, and 26.3.3 for output format details). The target genomes are treated as pairs comprising individuals for purposes of structuring the VCF output, so an even number of genomes is required. This low-level output method may be used to output any sample of `Genome` objects (the `Eidos` function `sample()` may be useful for constructing custom samples, as may the `SLiM` class `Individual`). For output of a sample from a single `Subpopulation`, the `outputVCFsample()` of `Subpopulation` may be more straightforward to use. If the optional parameter `filePath` is `NULL` (the default), output is directed to `SLiM`'s standard output. Otherwise, the output is sent to the file specified by `filePath`, overwriting that file if `append` is `F`, or appending to the end of it if `append` is `T`.

The parameters `outputMultiallelics`, `simplifyNucleotides`, and `outputNonnucleotides` affect the format of the output produced; see sections 26.2.3 and 26.2.4 for further discussion.

See `outputMS()` and `output()` for other output formats. Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

```
- (integer)positionsOfMutationsOfType(io<MutationType>$ mutType)
```

Returns the positions of mutations that are of the type specified by `mutType`, out of all of the mutations in the genome. If you need a vector of the matching `Mutation` objects, rather than just positions, use `-mutationsOfType()`. This method is provided for speed; it is much faster than the corresponding `Eidos` code.

```
+ (object<Mutation>)readFromMS(string$ filePath, io<MutationType>$ mutationType)
```

Read new mutations from the MS format file at `filePath` and add them to the target genomes. The number of target genomes must match the number of genomes represented in the MS file. To read into all of the genomes in a given subpopulation `pN`, simply call `pN.genomes.readFromMS()`, assuming the subpopulation's size matches that of the MS file. A vector containing all of the mutations created by `readFromMS()` is returned.

Each mutation is created at the position specified in the file, using the mutation type given by `mutationType`. Positions are expected to be in `[0,1]`, and are scaled to the length of the chromosome by multiplying by the last valid base position of the chromosome (i.e., one less than the chromosome length). Selection coefficients are drawn from the mutation type. The population of origin for each mutation is set to `-1`, and the tick of origin is set to the current tick. In a nucleotide-based model, if `mutationType` is nucleotide-based, a random nucleotide different from the ancestral nucleotide at the position will be chosen with equal probability.

The target genomes correspond, in order, to the call lines in the MS file. In sex-based models that simulate the X or Y chromosome, null genomes in the target vector will be skipped, and will not be used to correspond to any call line; however, care should be taken in this case that the lines in the MS file correspond to the target genomes in the manner desired.

```
+ (object<Mutation>)readFromVCF(string$ filePath,
  [Nio<MutationType>$ mutationType = NULL])
```

Read new mutations from the VCF format file at `filePath` and add them to the target genomes. The number of target genomes must match the number of genomes represented in the VCF file (i.e., two times the number of samples, if each sample is diploid). To read into all of the genomes in a given subpopulation `pN`, simply call `pN.genomes.readFromVCF()`, assuming the subpopulation's size matches that of the VCF file taking ploidy into account. A vector containing all of the mutations created by `readFromVCF()` is returned.

`SLiM`'s VCF parsing is quite primitive. The header is parsed only inasmuch as `SLiM` looks to see whether `SLiM`-specific VCF fields (see sections 26.2.3 and 26.2.4) are defined or not; the rest of the header information is ignored. Call lines are assumed to follow the format:

```
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT i0...iN
```



The **CHROM**, **ID**, **QUAL**, **FILTER**, and **FORMAT** fields are ignored, and information in the genotype fields beyond the **GT** genotype subfield are also ignored. SLiM's own VCF annotations (see section 26.2.3) are honored; in particular, mutations will be created using the given values of **MID**, **S**, **P0**, **T0**, and **MT** if those subfields are present, and **DOM**, if it is present, must match the dominance coefficient of the mutation type. The parameter **mutationType** (a **MutationType** object or id) will be used for any mutations that have no supplied mutation type id in the **MT** subfield; if **mutationType** would be used but is **NULL** an error will result. Mutation IDs supplied in **MID** will be used if no mutation IDs have been used in the simulation so far; if any have been used, it is difficult for SLiM to guarantee that there are no conflicts, so a warning will be emitted and the **MID** values will be ignored. If selection coefficients are not supplied with the **S** subfield, they will be drawn from the mutation type used for the mutation. If a population of origin is not supplied with the **P0** subfield, -1 will be used. If a tick of origin is not supplied with the **T0** subfield (or a generation of origin **G0** field, which was the SLiM convention before SLiM 4), the current tick will be used.

**REF** and **ALT** must always be comprised of simple nucleotides (A/C/G/T) rather than values representing indels or other complex states. Beyond this, the handling of the **REF** and **ALT** fields depends upon several factors. First of all, these fields are ignored in non-nucleotide-based models, although they are still checked for conformance. In nucleotide-based models, when a header definition for SLiM's **NONNUC** tag is present (as when nucleotide-based output is generated by SLiM): Second, if a **NONNUC** field is present in the **INFO** field the call line is taken to represent a non-nucleotide-based mutation, and **REF** and **ALT** are again ignored. In this case the mutation type used must be non-nucleotide-based. Third, if **NONNUC** is *not* present the call line is taken to represent a nucleotide-based mutation. In this case, the mutation type used must be nucleotide-based. Also, in this case, the specified reference nucleotide must match the existing ancestral nucleotide at the given position. In nucleotide-based models, when a header definition for SLiM's **NONNUC** tag is not present (as when loading a non-SLiM-generated VCF file): The mutation type will govern the way nucleotides are handled. If the mutation type used for a mutation is nucleotide-based, the nucleotide provided in the VCF file for that allele will be used. If the mutation type is non-nucleotide-based, the nucleotide provided will be ignored.

If multiple alleles using the same nucleotide at the same position are specified in the VCF file, a separate mutation will be created for each, mirroring SLiM's behavior with independent mutational lineages when writing VCF (see section 26.2.4). The **MULTIALLELIC** flag is ignored by **readFromVCF()**; call lines for mutations at the same base position in the same genome will result in stacked mutations whether or not **MULTIALLELIC** is present.

The target genomes correspond, in order, to the haploid or diploid calls provided for **i0...iN** (the sample IDs) in the VCF file. In sex-based models that simulate the X or Y chromosome, null genomes in the target vector will be skipped, and will not be used to correspond to any of **i0...iN**; however, care should be taken in this case that the genomes in the VCF file correspond to the target genomes in the manner desired.

+ (void)removeMutations([No<Mutation> mutations = NULL], [logical\$ substitute = F])  
Remove the mutations in **mutations** from the target genome(s), if they are present (if they are not present, they will be ignored). If **NULL** is passed for **mutations** (which is the default), then all mutations will be removed from the target genomes; in this case, **substitute** must be **F** (a specific vector of mutations to be substituted is required). Note that the **Mutation** objects removed remain valid, and will still be in the simulation's mutation registry (i.e., will be returned by the **Species** property **mutations**), until the next tick.

Removing mutations will normally affect the fitness values calculated at the end of the current tick; if you want current fitness values to be affected, you can call the **Species** method **recalculateFitness()** – but see the documentation of that method for caveats.

The optional parameter **substitute** was added in SLiM 2.2, with a default of **F** for backward compatibility. If **substitute** is **T**, **Substitution** objects will be created for all of the removed mutations so that they are recorded in the simulation as having fixed, just as if they had reached



fixation and been removed by SLiM's own internal machinery. This will occur regardless of whether the mutations have in fact fixed, regardless of the `convertToSubstitution` property of the relevant mutation types, and regardless of whether all copies of the mutations have even been removed from the simulation (making it possible to create `Substitution` objects for mutations that are still segregating). It is up to the caller to perform whatever checks are necessary to preserve the integrity of the simulation's records. Typically `substitute` will only be set to `T` in the context of calls like `sim.subpopulations.genomes.removeMutations(muts, T)`, such that the substituted mutations are guaranteed to be entirely removed from circulation. As mentioned above, `substitute` may not be `T` if `mutations` is `NULL`.

- `(float$)sumOfMutationsOfType(io<MutationType>$ mutType)`

Returns the sum of the selection coefficients of all mutations that are of the type specified by `mutType`, out of all of the mutations in the genome. This is often useful in models that use a particular mutation type to represent QTLs with additive effects; in that context, `sumOfMutationsOfType()` will provide the sum of the additive effects of the QTLs for the given mutation type. This method is provided for speed; it is much faster than the corresponding Eidos code. Note that this method also exists on `Individual`, for cases in which the sum across both genomes of an individual is desired.

## 24.5 Class `GenomicElement`

*Superclass: `Object`*

This class represents a genomic element of a particular genomic element type, with a start and end; the chromosome is composed of a series of such genomic elements. Section 1.5.4 presents an overview of the conceptual role of this class.

### 24.5.1 *GenomicElement* properties

`endPosition => (integer$)`

The last position in the chromosome contained by this genomic element.

`genomicElementType => (object<GenomicElementType>$)`

The `GenomicElementType` object that defines the behavior of this genomic element.

`startPosition => (integer$)`

The first position in the chromosome contained by this genomic element.

`tag <=> (integer$)`

A user-defined `integer` value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use.

### 24.5.2 *GenomicElement* methods

- `(void)setGenomicElementType(io<GenomicElementType>$ genomicElementType)`

Set the genomic element type used for a genomic element (see section 1.5.4). The `genomicElementType` parameter should supply the new genomic element type for the element, either as a `GenomicElementType` object or as an `integer` identifier. The genomic element type for a genomic element is normally a constant in simulations, so be sure you know what you are doing.

## 24.6 Class `GenomicElementType`

*Superclass: `Dictionary`*

This class represents a type of genomic element, with particular mutation types. The genomic element types currently defined in the simulation are defined as global constants with the same

names used in the SLiM input file – `g1`, `g2`, and so forth. Section 1.5.4 presents an overview of the conceptual role of this class.

#### 24.6.1 *GenomicElementType* properties

`color <=> (string$)`

The color used to display genomic elements of this type in SLiMgui. Outside of SLiMgui, this property still exists, but is not used by SLiM. Colors may be specified by name, or with hexadecimal RGB values of the form `"#RRGGBB"` (see the Eidos manual). If `color` is the empty string, `""`, SLiMgui's default color scheme is used; this is the default for new *GenomicElementType* objects.

`id => (integer$)`

The identifier for this genomic element type; for genomic element type `g3`, for example, this is 3.

`mutationFractions => (float)`

For each *MutationType* represented in this genomic element type, this property has the corresponding fraction of all mutations that will be drawn from that *MutationType*.

`mutationMatrix => (float)`

The nucleotide mutation matrix used for this genomic element type, set up by `initializeGenomicElementType()` and `setMutationMatrix()`. This property is only defined in nucleotide-based models; it is unavailable otherwise.

`mutationTypes => (object<MutationType>)`

The *MutationType* instances used by this genomic element type.

`species => (object<Species>$)`

The species to which the target object belongs.

`tag <=> (integer$)`

A user-defined integer value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the *Dictionary* class; see the Eidos manual), for another way of attaching state to genomic element types.

#### 24.6.2 *GenomicElementType* methods

– `(void)setMutationFractions(io<MutationType> mutationTypes, numeric proportions)`

Set the mutation type fractions contributing to a genomic element type. The `mutationTypes` vector should supply the mutation types used by the genomic element (either as *MutationType* objects or as integer identifiers), and the `proportions` vector should be of equal length, specifying the relative proportion of mutations that will be drawn from each corresponding type (see section 1.5.4). This is normally a constant in simulations, so be sure you know what you are doing.

– `(void)setMutationMatrix(float mutationMatrix)`

Sets a new nucleotide mutation matrix for the genomic element type. This replaces the mutation matrix originally set by `initializeGenomicElementType()`. This method may only be called in nucleotide-based models.

### 24.7 Class Individual

*Superclass: Dictionary*

This class represents a single simulated individual. Individuals in SLiM are diploid, and thus contain two *Genome* objects. Most functionality in SLiM is contained in the *Genome* class; the

`Individual` class is mostly a convenient way to treat the pairs of genomes associated with an individual as a single object, and to associate a `tag` value with individuals. Section 1.5.1 presents an overview of the conceptual role of this class.

### 24.7.1 Individual properties

#### `age <=> (integer$)`

The age of the individual, measured in cycles. A newly generated offspring individual will have an age of 0 in the same tick in which it was created. The age of every individual is incremented by one at the same point that its species cycle counter is incremented, at the end of the tick cycle, *if and only if* its species was active in that tick. The age of individuals may be changed; usually this only makes sense when setting up the initial state of a model, however.

#### `color <=> (string$)`

The color used to display the individual in SLiMgui. Outside of SLiMgui, this property still exists, but is not used by SLiM. Colors may be specified by name, or with hexadecimal RGB values of the form `"#RRGGBB"` (see the Eidos manual). If `color` is the empty string, `""`, SLiMgui's default (fitness-based) color scheme is used; this is the default for new `Individual` objects. Note that named colors will be converted to RGB internally, so the value of this property will always be a hexadecimal RGB color string (or `""`).

#### `fitnessScaling <=> (float$)`

A `float` scaling factor applied to the individual's fitness (i.e., the fitness value computed for the individual will be multiplied by this value). This provides a simple, fast way to modify the fitness of an individual; conceptually it is similar to returning a fitness effect for the individual from a `fitnessEffect()` callback, but without the complexity and performance overhead of implementing such a callback. To scale the fitness of all individuals in a subpopulation by the same factor, see the `fitnessScaling` property of `Subpopulation`.

The value of `fitnessScaling` is reset to 1.0 every tick, so that any scaling factor set lasts for only a single tick. This reset occurs immediately after fitness values are calculated, in both WF and nonWF models.

#### `genomes => (object<Genome>)`

The pair of `Genome` objects associated with this individual. If only one of the two genomes is desired, the `genome1` or `genome2` property may be used.

#### `genomesNonNull => (object<Genome>)`

The pair of `Genome` objects associated with this individual, as with the `genomes` property, if both are not null genomes. If one or both are null genomes, the null genomes are excluded from the returned vector. This is a convenience shorthand, sometimes useful in models that involve null genomes.

#### `genome1 => (object<Genome>)`

The first `Genome` object associated with this individual. This property is particularly useful when you want the first genome from each of a vector of individuals, as often arises in haploid models.

#### `genome2 => (object<Genome>)`

The second `Genome` object associated with this individual. This property is particularly useful when you want the second genome from each of a vector of individuals, as often arises in haploid models.

#### `index => (integer$)`

The index of the individual in the `individuals` vector of its `Subpopulation`.

#### `meanParentAge => (float$)`

The average age of the parents of this individual, measured in cycles. Parentless individuals will have a `meanParentAge` of 0.0. The mean parent age is determined when a new offspring is generated,

from the `age` property of the parent or parents involved in generating the offspring. For `addRecombinant()` that is somewhat complex; see that method for details.

`migrant => (logical$)`

Set to T if the individual is a recent migrant, F otherwise. The definition of “recent” depends upon the model type (WF or nonWF).

In WF models, this flag is set at the point when a new child is generated if it is a migrant (i.e., if its source subpopulation is not the same as its subpopulation), and remains valid, with the same value, for the rest of the individual’s lifetime.

In nonWF models, this flag is F for all new individuals, is set to F in all individuals at the end of the reproduction tick cycle stage, and is set to T on all individuals moved to a new subpopulation by `takeMigrants()` or a `survival()` callback; the T value set by `takeMigrants()` or `survival()` will remain until it is reset at the end of the next reproduction tick cycle stage.

`pedigreeID => (integer$)`

If pedigree tracking is turned on with `initializeSLiMOptions(keepPedigrees=T)`, `pedigreeID` is a unique non-negative identifier for each individual in a simulation, never re-used throughout the duration of the simulation run. If pedigree tracking is not enabled, this property is unavailable.

`pedigreeParentIDs => (integer)`

If pedigree tracking is turned on with `initializeSLiMOptions(keepPedigrees=T)`, `pedigreeParentIDs` contains the values of `pedigreeID` that were possessed by the parents of an individual; it is thus a vector of two values. If pedigree tracking is not enabled, this property is unavailable. Parental values may be -1 if insufficient ticks have elapsed for that information to be available (because the simulation just started, or because a subpopulation is new).

`pedigreeGrandparentIDs => (integer)`

If pedigree tracking is turned on with `initializeSLiMOptions(keepPedigrees=T)`, `pedigreeGrandparentIDs` contains the values of `pedigreeID` that were possessed by the grandparents of an individual; it is thus a vector of four values. If pedigree tracking is not enabled, this property is unavailable. Grandparental values may be -1 if insufficient ticks have elapsed for that information to be available (because the simulation just started, or because a subpopulation is new).

`reproductiveOutput => (integer$)`

If pedigree tracking is turned on with `initializeSLiMOptions(keepPedigrees=T)`, `reproductiveOutput` contains the number of offspring for which this individual has been a parent. If pedigree tracking is not enabled, this property is unavailable. If an individual is a parent by cloning or selfing, or as *both* parents for a biparental mating, this value is incremented by two. Involvement of an individual as a parent for an `addRecombinant()` call does not change this property’s value, since the reproductive contribution in that case is unclear; one must conduct separate bookkeeping for that case if necessary.

See also the Subpopulation property `lifetimeReproductiveOutput`.

`sex => (string$)`

The sex of the individual. This will be "H" if sex is not enabled in the simulation (i.e., for hermaphrodites), otherwise "F" or "M" as appropriate.

`spatialPosition => (float)`

The spatial position of the individual. The length of the `spatialPosition` property (the number of coordinates in the spatial position of an individual) depends upon the spatial dimensionality declared with `initializeSLiMOptions()`. If the spatial dimensionality is zero (as it is by default), it is an error to access this property. The elements of this property are identical to the values of the `x`, `y`, and `z` properties (if those properties are encompassed by the spatial dimensionality of the simulation). In other words, if the declared dimensionality is "xy", the `individual.spatialPosition` property is

equivalent to `c(individual.x, individual.y)`; `individual.z` is not used since it is not encompassed by the simulation's dimensionality. This property cannot be set, but the `setSpatialPosition()` method may be used to achieve the same thing.

`subpopulation => (object<Subpopulation>$)`

The `Subpopulation` object to which the individual belongs.

`tag <=> (integer$)`

A user-defined `integer` value (as opposed to `tagF`, which is of type `float`). The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the `Dictionary` class; see the Eidos manual), for another way of attaching state to individuals. Note that the `Individual` objects used by SLiM are new for every new offspring, so the `tag` value of each new offspring generated in each tick will be initially undefined.

`tagF <=> (float$)`

A user-defined `float` value (as opposed to `tag`, which is of type `integer`). The value of `tagF` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tagF` is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the `Dictionary` class; see the Eidos manual), for another way of attaching state to individuals.

Note that at present, although many classes in SLiM have an `integer`-type `tag` property, only `Individual` has a `float`-type `tagF` property, because attaching model state to individuals seems to be particularly common and useful. If a `tagF` property would be helpful on another class, it would be easy to add.

See the description of the `tag` property above for additional comments.

`uniqueMutations => (object<Mutation>)`

All of the `Mutation` objects present in this individual. Mutations present in both genomes will occur only once in this property, and the mutations will be given in sorted order by `position`, so this property is similar to `sortBy(unique(individual.genomes.mutations), "position")`. It is not identical to that call, only because if multiple mutations exist at the exact same position, they may be sorted differently by this method than they would be by `sortBy()`. This method is provided primarily for speed; it executes much faster than the Eidos equivalent above. Indeed, it is faster than just `individual.genomes.mutations`, and gives uniquing and sorting on top of that, so it is advantageous unless duplicate entries for homozygous mutations are actually needed.

`x <=> (float$)`

A user-defined `float` value. The value of `x` is initially undefined (i.e., has an effectively random value that could be different every time you run your model); if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code, typically in a `modifyChild()` callback. The value of `x` is not used by SLiM unless the optional "continuous space" facility is enabled with the `dimensionality` parameter to `initializeSLiMOptions()`, in which case `x` will be understood to represent the `x` coordinate of the individual in space. If continuous space is not enabled, you may use `x` as an additional tag value of type `float`.

`y <=> (float$)`

A user-defined `float` value. The value of `y` is initially undefined (i.e., has an effectively random value that could be different every time you run your model); if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code, typically in a `modifyChild()` callback. The value of `y` is not used by SLiM unless the optional

“continuous space” facility is enabled with the `dimensionality` parameter to `initializeSLiMOptions()`, in which case `y` will be understood to represent the *y* coordinate of the individual in space (if the dimensionality is “xy” or “xyz”). If continuous space is not enabled, or the dimensionality is not “xy” or “xyz”, you may use `y` as an additional tag value of type `float`.

`z <-> (float$)`

A user-defined `float` value. The value of `z` is initially undefined (i.e., has an effectively random value that could be different every time you run your model); if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code, typically in a `modifyChild()` callback. The value of `z` is not used by SLiM unless the optional “continuous space” facility is enabled with the `dimensionality` parameter to `initializeSLiMOptions()`, in which case `z` will be understood to represent the *z* coordinate of the individual in space (if the dimensionality is “xyz”). If continuous space is not enabled, or the dimensionality is not “xyz”, you may use `z` as an additional tag value of type `float`.

### 24.7.2 Individual methods

– `(logical)containsMutations(object<Mutation> mutations)`

Returns a `logical` vector indicating whether each of the mutations in `mutations` is present in the individual (in either of its genomes); each element in the returned vector indicates whether the corresponding mutation is present (T) or absent (F). This method is provided for speed; it is much faster than the corresponding Eidos code.

– `(integer$)countOfMutationsOfType(io<MutationType>$ mutType)`

Returns the number of mutations that are of the type specified by `mutType`, out of all of the mutations in the individual (in both of its genomes; a mutation that is present in both genomes counts twice). If you need a vector of the matching `Mutation` objects, rather than just a count, you should probably use `uniqueMutationsOfType()`. This method is provided for speed; it is much faster than the corresponding Eidos code.

– `(float)relatedness(object<Individual> individuals)`

Returns a vector containing the degrees of relatedness between the receiver and each of the individuals in `individuals`. The relatedness between A and B is always `1.0` if A and B are actually the same individual; this facility works even if SLiM’s optional pedigree tracking is not enabled (in which case all other relatedness values will be `0.0`). Otherwise, if pedigree tracking is turned on with `initializeSLiMOptions(keepPedigrees=T)`, this method will use the pedigree information described in section 24.7.1 to construct a relatedness estimate.

More specifically, this method uses all available pedigree information from the grandparental and parental pedigree records of A and B to compute an estimate of the degree of consanguinity between A and B. Siblings have a relatedness of `0.5`, as do parents to their children and vice versa; cousins have a relatedness of `0.125`; and so forth. If, according to the pedigree information available, A and B have no blood relationship, the value returned is `0.0`. Note that the value returned by `relatedness()` is what is called the “coefficient of relationship” between the two individuals (Wright, 1922; <https://doi.org/10.1086/279872>), and ranges from `0.0` to `1.0`.

There is another commonly used metric of relatedness, called the “kinship coefficient”, that reflects the probability of identity by descent between two individuals A and B. In general, it is approximately equal to one-half of the coefficient of relationship; if an approximate estimate of the kinship coefficient is acceptable, especially in models in which individuals are expected to be outbred, you can simply divide `relatedness()` by two. However, it should be noted that Wright’s coefficient of relationship is *not* a measure of the probability of identity by descent, and so it is not exactly double the kinship coefficient; they actually measure different things. More precisely, the relationship between them is  $r = 2\phi / \sqrt{(1+f_A)(1+f_B)}$ , where  $r$  is Wright’s coefficient of relatedness,  $\phi$  is the kinship coefficient, and  $f_A$  and  $f_B$  are the inbreeding coefficients of A and B respectively.



Note that this relatedness is simply pedigree-based relatedness, and does not necessarily correspond to genetic relatedness, because of the effects of factors like assortment and recombination. If a metric of actual genetic relatedness is desired, tree-sequence recording can be used after simulation is complete, to compute the exact genetic relatedness between individuals based upon the complete ancestry tree (a topic which is beyond the scope of this manual). Actual genetic relatedness cannot presently be calculated during a simulation run; the information is implicitly contained in the recorded tree-sequence tables, but calculating it is too computationally expensive to be reasonable.

This method makes a couple of assumptions. One assumption is that the grandparents (or the parents, if grandparental information is not available) are themselves unrelated and that they are not inbred; this assumption is necessary because we have no information about their parentage, since SLiM's pedigree tracking information only goes back two generations. Be aware that in a model where inbreeding or selfing occurs at all (including "incidental selfing", where a hermaphroditic individual happens to choose itself as a mate), some level of "background relatedness" will be present and this assumption will be violated. In such circumstances, `relatedness()` will therefore tend to underestimate the degree of relatedness between individuals, and the greater the degree of inbreeding, the greater the underestimation will be. If inbreeding is allowed in a model – and particularly if it is common – the results of `relatedness()` should therefore not be taken as an estimate of *absolute* relatedness, but can still be useful as an estimate of *relative* relatedness (indicating that, say, A appears from the information available to be more closely related to B than it is to C).

A second assumption is that, in nonWF models, `addRecombinant()` is not being used, since the pedigree relationships involved are then too complex to trace. Indeed, `addRecombinant()` does not record pedigree information at all, for this reason, and so the relatedness of individuals produced by `addRecombinant()` will be 0.0.

#### + (void)setSpatialPosition(float position)

Sets the spatial position of the individual (as accessed through the `spatialPosition` property). The length of `position` (the number of coordinates in the spatial position of an individual) depends upon the spatial dimensionality declared with `initializeSLiMOptions()`. If the spatial dimensionality is zero (as it is by default), it is an error to call this method. The elements of `position` are set into the values of the `x`, `y`, and `z` properties (if those properties are encompassed by the spatial dimensionality of the simulation). In other words, if the declared dimensionality is "xy", calling `individual.setSpatialPosition(c(1.0, 0.5))` property is equivalent to `individual.x = 1.0; individual.y = 0.5; individual.z` is not set (even if a third value is supplied in `position`) since it is not encompassed by the simulation's dimensionality in this example.

Note that this is an Eidos class method, somewhat unusually, which allows it to work in a special way when called on a vector of individuals. When the target vector of individuals is non-singleton, this method can do one of two things. If `position` contains just a single point (i.e., is equal in length to the spatial dimensionality of the model), the spatial position of all of the target individuals will be set to the given point. Alternatively, if `position` contains one point per target individual (i.e., is equal in length to the number of individuals multiplied by the spatial dimensionality of the model), the spatial position of each target individual will be set to the corresponding point from `position` (where the point data is concatenated, not interleaved, just as it would be returned by accessing the `spatialPosition` property on the vector of target individuals). Calling this method with a `position` vector of any other length is an error.

#### – (float\$)sumOfMutationsOfType(io<MutationType>\$ mutType)

Returns the sum of the selection coefficients of all mutations that are of the type specified by `mutType`, out of all of the mutations in the genomes of the individual. This is often useful in models that use a particular mutation type to represent QTLs with additive effects; in that context, `sumOfMutationsOfType()` will provide the sum of the additive effects of the QTLs for the given mutation type. This method is provided for speed; it is much faster than the corresponding Eidos code. Note that this method also exists on `Genome`, for cases in which the sum for just one genome is desired.

- (object<Mutation>)uniqueMutationsOfType(io<MutationType>\$ mutType)

Returns an object vector of all the mutations that are of the type specified by `mutType`, out of all of the mutations in the individual. Mutations present in both genomes will occur only once in the result of this method, and the mutations will be given in sorted order by `position`, so this method is similar to `sortBy(unique(individual.genomes.mutationsOfType(mutType)), "position")`. It is not identical to that call, only because if multiple mutations exist at the exact same position, they may be sorted differently by this method than they would be by `sortBy()`. If you just need a count of the matching `Mutation` objects, rather than a vector of the matches, use `-countOfMutationsOfType()`. This method is provided for speed; it is much faster than the corresponding Eidos code. Indeed, it is faster than just `individual.genomes.mutationsOfType(mutType)`, and gives uniquing and sorting on top of that, so it is advantageous unless duplicate entries for homozygous mutations are actually needed.

## 24.8 Class InteractionType

*Superclass: Dictionary*

This class represents a type of interaction between individuals. This is an advanced feature, the use of which is optional. Once an interaction type is set up with `initializeInteractionType()` (see section 24.1), it can be evaluated and then queried to give information such as the nearest interacting neighbors of an individual, or the total strength of interactions felt by an individual, relatively efficiently.

There are two types of individual, in the paradigm provided by `InteractionType`: the *receiver* of an interaction, and the *exerter* of that interaction. The same individual might be a receiver for one interaction and the exerter for another interaction, and both of those interactions might be governed by the same `InteractionType`, but nevertheless, for any given interaction the distinction remains important. The distinction is important because `InteractionType` enforces this directionality – *from* exerters, *to* receivers – throughout its design. Interactions therefore fundamentally define a one-to-many relationship, from one receiver to the (potentially) many exerters that act upon that receiver. In using `InteractionType`, you will generally start with a receiver, and ask an `InteractionType` object to handle a query about that receiver, such as “what are the ten nearest exerters to this receiver?”. `InteractionType` is optimized to find and return the set of exerters influencing a given receiver; it is *not* optimized for the reverse, finding and returning the set of receivers influenced by a given exerter. (If that seems desirable, you might wish to flip your perspective and regard the interaction as actually working in the opposite direction!)

Interactions are usually spatial, depending upon the spatial dimensionality established with `initializeSLiMOptions()` (section 24.1), but they do not have to be spatial. For spatial interactions, the strength of the interaction from an exerter to a receiver often depends (partly, at least) upon the distance between the two; nearby exerters often wield a stronger influence upon a receiver than more distant exerters do. Non-spatial interactions, on the other hand, are of course unrelated to distance, and the strength of interaction between two individuals depends entirely upon other factors, expressed by an `interaction()` callback in the model script. Spatial interactions can use `interaction()` callbacks too, to modify interaction strengths calculated by SLiM, if factors other than distance need to influence the strength of interactions.

Note that if there are  $N$  receivers to be assessed, each of which potentially interacts with  $M$  possible exerters, then – depending upon the queries executed – `InteractionType` may take computational time proportional to  $N \times \log(N) \times M$ . If every individual interacts with every other,  $M$  is equal to  $N$  and there will be  $N^2$  interactions to be evaluated, and the overall computational time may be as bad as  $N^2 \log(N)$ , although in practice it is perhaps closer to  $N^2$  – still bad. Modeling interactions with large population sizes can therefore be very expensive, although `InteractionType` goes to considerable lengths to minimize the overhead. To reduce this

computational burden for your models, it is essential to reduce  $N$ ,  $M$  or both. Spatial interactions can have – and almost always should have – a maximum distance, which allows them to be evaluated much more efficiently (since all interactions beyond the maximum distance can be assumed to have a strength of zero); setting this maximum distance to be as small as possible, without introducing unwanted artifacts, is the single most important factor for using `InteractionType` efficiently. For sexual models, interactions that are specific to the sexes in a particular way (males competing with other males, males competing to mate with females, etc.) can be declared to be sex-specific, which can also substantially reduce the overhead of querying them.

We will focus, in the remaining discussion, on spatial interactions since they are more common. The first step in `InteractionType`'s evaluation of a spatial interaction is to determine the distance from the individual receiving the interaction (the receiver) to the individual exerting the interaction (the exiter). This is computed as the Euclidean distance between the spatial positions of the individuals, based upon the spatiality of the interaction (i.e., the spatial dimensions used by the interaction, which may be less than the dimensionality of the simulation as a whole). If the receiver and exiter occupy different subpopulations, it is assumed that they nevertheless occupy the same coordinate system; this can be particularly useful for evaluating interactions between individuals of different species. Second, this distance is compared to the maximum distance for the interaction type; if it is beyond that limit, the interaction strength is always zero (and it is also always zero for the interaction of an individual with itself). Third (when the distance is less than the maximum), the distance is converted into an interaction strength by an *interaction function* (IF), which is a characteristic of the `InteractionType`. Finally, this interaction strength may be modified by the `interaction()` callbacks currently active in the simulation, if any (see section 25.7).

`InteractionType` is actually a wrapper for three different spatial query engines that share some of their data but work very differently. The first engine is a brute-force engine that simply computes distances and interaction strengths in response to queries. This engine is usually used in response to queries for simple information, such as the `distance()`, `distanceToPoint()`, and `strength()` methods.

The second engine is based upon a data structure called a “ $k$ -d tree” that is designed to optimize searches for spatially proximate points. This engine is usually used directly in response to queries involving “neighbors”, such as `nearestNeighbors()` and `nearestNeighborsOfPoint()`. In SLiM, the term “neighbor” means an individual that is within the maximum interaction distance of a receiver (excluding the receiver itself) or a focal point; the neighbors of the receiver or point are therefore those that fall within the fixed radius defined by the maximum interaction distance. Calls with “neighbor” in their name explicitly use the  $k$ -d tree engine, and may therefore be called only for spatial interactions; in non-spatial interactions there is no concept of a “neighbor”. In terms of computational complexity, finding the nearest neighbor of a given receiver by brute force would be an  $O(N)$  computation, whereas with the  $k$ -d tree it is typically an  $O(\log N)$  computation – a very important difference, especially for large  $N$ . In general, to get the best performance from a spatial model, you should use neighbor-based calls to make minimal queries when possible. If all you really care about is the distance to the nearest neighbor of a receiver, for example, then use `nearestNeighbors()` to find the neighbor and then call `distance()` to get the distance to that neighbor, rather than getting the distances to all individuals with `distance()` and then using `min()` to select the smallest.

The third engine, introduced in SLiM 3.1 and radically modified in SLiM 4, is based upon a data structure called a “sparse array” that is designed to track sparse non-zero values within a dataset that contains mostly zeros. It applies to spatial interactions because most pairs of individuals probably interact with a strength of zero (because typically  $N \gg M$ , because few individuals fall within the maximum interaction radius from a given individual). In SLiM 4, the full sparse array of

interactions is no longer calculated (as it was in SLiM 3); instead, single rows of the sparse array are calculated on demand, providing most of the benefits of the data structure with only a tiny fraction of the memory overhead. In `InteractionType` parlance, such a single row of the sparse array is called a *sparse vector*. Sparse vectors are used to temporarily cache calculated distances and strengths for interactions within a given subpopulation. They are built using the *k*-d tree to find the interacting neighbors of each individual, and once built they can respond extremely quickly to queries from methods such as `totalOfNeighborStrengths()`; the interacting neighbors of a given individual are already known, allowing response in  $O(M)$  time. These sparse vectors are built on demand, when queries that would benefit from them are made. For them to be effective, it is particularly important that a maximum interaction distance be used that is as small as possible, so beginning with SLiM 3.1 a warning is issued when no maximum distance is defined for spatial interactions.

As mentioned above, once an interaction distance has been found it is translated into an interaction strength by an interaction function. There are currently five options for interaction functions (IFs) in SLiM, represented by single-character codes:

"f" – a **fixed** interaction strength. This IF type has a single parameter, the interaction strength to be used for all interactions of this type. By default, interaction types use a type "f" IF with a value of `1.0`, so interactions are binary: on within the maximum distance, off outside.

"l" – a **linear** interaction strength. This IF type has a single parameter, the maximum interaction strength to be used at distance `0.0`. The interaction strength falls off linearly, reaching exactly zero at the maximum distance. In other words, for distance  $d$ , maximum interaction distance  $d_{\max}$ , and maximum interaction strength  $f_{\max}$ , the formula for this IF is  $f(d) = f_{\max}(1 - d / d_{\max})$ .

"e" – A **negative exponential** interaction strength. This IF type is specified by two parameters, a maximum interaction strength and a shape parameter. The interaction strength falls off non-linearly from the maximum, and cuts off discontinuously at the maximum distance; typically a maximum distance is chosen such that the interaction strength at that distance is very small anyway. The IF for this type is  $f(d) = f_{\max} \exp(-\lambda d)$ , where  $\lambda$  is the specified shape parameter. Note that this parameterization is *not* the same as for the Eidos function `rexp()`.

"n" – A **normal** interaction strength (i.e., Gaussian, but "g" is avoided to prevent confusion with the gamma-function option provided for, e.g., `MutationType`). The interaction strength falls off non-linearly from the maximum, and cuts off discontinuously at the maximum distance; typically a maximum distance is chosen such that the interaction strength at that distance is very small anyway. This IF type is specified by two parameters, a maximum interaction strength and a standard deviation. The Gaussian IF for this type is  $f(d) = f_{\max} \exp(-d^2 / 2\sigma^2)$ , where  $\sigma$  is the standard deviation parameter. Note that this parameterization is *not* the same as for the Eidos function `rnorm()`. A Gaussian function is often used to model spatial interactions, but is relatively computation-intensive.

"c" – A **Cauchy-distributed** interaction strength. The interaction strength falls off non-linearly from the maximum, and cuts off discontinuously at the maximum distance; typically a maximum distance is chosen such that the interaction strength at that distance is very small anyway. This IF type is specified by two parameters, a maximum interaction strength and a scale parameter. The IF for this type is  $f(d) = f_{\max} / (1 + (d/\lambda)^2)$ , where  $\lambda$  is the scale parameter. Note that this parameterization is *not* the same as for the Eidos function `rcauchy()`. A Cauchy distribution can be used to model interactions with relatively fat tails.

An `InteractionType` may be created using the `initializeInteractionType()` function (see section 24.1). It must then be evaluated, with the `evaluate()` method, for the subpopulations containing exerts and receivers before it will respond to queries regarding individuals in those subpopulations; querying with exerts or receivers whose subpopulations have not been

evaluated will result in an error. Calling `evaluate()` causes the positions of all receivers and exerters to be cached, thus defining a snapshot in time that the `InteractionType` will then use to respond to queries (allowing it to build its  $k$ -d tree based upon the snapshot positions). In WF models, this evaluated state will last until the current parental generation expires, at the end of the next offspring-generation stage. Before the `InteractionType` may be used with the new parental generation (the offspring of the old parental generation), the interaction must be evaluated again. In nonWF models, interactions are invalidated twice during the tick cycle: once after new offspring are produced (just before `early()` events), and once just before individuals die (just after fitness calculations); they are also invalidated any time `takeMigrants()` is called to move individuals between subpopulations. Note that `interaction()` callbacks are called when queries are served, not when `evaluate()` is called.

`InteractionType` will automatically account for any periodic spatial boundaries established with the `periodicity` parameter of `initializeSLIMOptions()`; interactions will wrap around the periodic boundaries without any additional configuration of the interaction. Interactions involving periodic spatial boundaries entail some additional overhead in both memory usage and processor time; in particular, setting up the  $k$ -d tree after the interaction is evaluated may take many times longer than in the non-periodic case (but this is rarely a bottleneck anyway). Once the  $k$ -d tree has been set up, responses to spatial queries involving it should then be nearly as fast as in the non-periodic case. Spatial queries that do not involve the  $k$ -d tree will generally be marginally slower than in the non-periodic case, but the difference should not be large.

`InteractionType` provides a fairly large and confusingly similar set of methods, designed to answer every common type of spatial query efficiently. To help find the right method for the job, here is a summary of the methods that involve distances or interaction strengths, either between receivers and exerters, or between a focal point and exerters:

Spatial query type	Individual-Individual interactions	Individual-Point interactions
Distance measurement to <b>all</b> individuals	<code>distance()</code>	<code>distanceFromPoint()</code>
Distance measurement to <b>interacting</b> individuals	<code>interactionDistance()</code>	—
Counting <b>all</b> neighbors	<code>neighborCount()</code>	<code>neighborCountOfPoint()</code>
Counting <b>interacting</b> neighbors	<code>interactingNeighborCount()</code>	—
Getting <b>all</b> neighbors	<code>nearestNeighbors()</code>	<code>nearestNeighborsOfPoint()</code>
Getting <b>interacting</b> neighbors	<code>nearestInteractingNeighbors()</code>	—
Interaction strength measurement	<code>strength()</code>	—
Total interaction strength of interacting neighbors	<code>totalOfNeighborStrengths()</code>	—
Population density of interacting neighbors	<code>localPopulationDensity()</code>	—
Drawing proportional to interaction strength	<code>drawByStrength()</code>	—

Consider whether you want to query for neighbors (all individual near the receiver), for interacting neighbors (nearby individuals that exert an interaction strength upon the receiver,



regardless of what that strength is), or for something about the actual interaction strengths. In general, the simpler queries will be faster; finding neighbors is faster than finding interacting neighbors, which is in turn faster than actually evaluating strengths. Furthermore, counting individuals is faster than actually returning the individuals in question. The last three methods in the table have to evaluate the interaction strength between a receiver and every exorter that interacts with it, so they can be quite slow; if you can, for example, simply count the number of neighbors with `neighborCount()`, rather than using `totalOfNeighborStrengths()` to sum up their interaction strengths, the former will likely be significantly faster than the latter. As has been mentioned above, for best performance the maximum interaction distance should be set to as small a value as possible.

#### 24.8.1 *InteractionType* properties

`id => (integer$)`

The identifier for this interaction type; for interaction type `i3`, for example, this is 3.

`maxDistance <=> (float$)`

The maximum distance over which this interaction will be evaluated. For inter-individual distances greater than `maxDistance`, the interaction strength will be zero.

`reciprocal => (logical$)`

The reciprocity of the interaction, as specified in `initializeInteractionType()`. This will be `T` for reciprocal interactions (those for which the interaction strength of B upon A is equal to the interaction strength of A upon B), and `F` otherwise.

`sexSegregation => (string$)`

The sex-segregation of the interaction, as specified in `initializeInteractionType()`. For non-sexual simulations, this will be `"**"`. For sexual simulations, this `string` value indicates the sex of individuals feeling the interaction, and the sex of individuals exerting the interaction; see `initializeInteractionType()` for details.

`spatiality => (string$)`

The spatial dimensions used by the interaction, as specified in `initializeInteractionType()`. This will be `""` (the empty string) for non-spatial interactions, or `"x"`, `"y"`, `"z"`, `"xy"`, `"xz"`, `"yz"`, or `"xyz"`, for interactions using those spatial dimensions respectively. The specified dimensions are used to calculate the distances between individuals for this interaction. The value of this property is always the same as the value given to `initializeInteractionType()`.

`tag <=> (integer$)`

A user-defined `integer` value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the `Dictionary` class; see the Eidos manual), for another way of attaching state to interaction types.

#### 24.8.2 *InteractionType* methods

– `(float)clippedIntegral(No<Individual> receivers)`

Returns a vector containing the integral of the interaction function as experienced by each of the individuals in `receivers`. For each given individual, the interaction function is clipped to the edges of the spatial bounds of the subpopulation that individual inhabits; the individual's spatial position must be within bounds or an error is raised. A periodic boundary will, correctly, not clip the interaction function. The interaction function is also clipped to the interaction's maximum distance; that distance must be less than half of the extent of the spatial bounds in each dimension (so that, for a given dimension, the interaction function is clipped by the spatial bounds on only one side), otherwise



an error is raised. Note that for sex-specific interaction types, the sex of the receiver does not matter; an individual might not actually receive any interactions because of its sex, but it is still considered to have the same interaction function integral. If `receivers` is `NULL`, the maximal integral is returned, as would be experienced by an individual farther than the maximum distance from any edge. The `evaluate()` method must have been previously called for the receiver subpopulation, and positions saved at evaluation time will be used. If the `InteractionType` is non-spatial, this method may not be called.

The computed value of the integral is not exact; it is calculated by an approximate numerical method designed to be fast, but the error should be fairly small (typically less than 1% from the true value). A large amount of computation will occur the first time this method is called (perhaps taking more than a second, depending upon hardware), but subsequent calls should be very fast. This method does not invoke `interaction()` callbacks; the calculated integrals are only for the interaction function itself, and so will not be accurate if `interaction()` callbacks modify the relationship between distance and interaction strength. For this reason, the overhead of the first call will *not* reoccur when individuals move or when the interaction is re-evaluated; for typical models, the initial overhead will be incurred only once. The initial overhead will reoccur, however, if the interaction function itself, or the maximum interaction distance, are changed; frequent change of those parameters may render the performance of this method unacceptable.

The integral values returned by `clippedIntegral()` can be useful for computing interaction metrics that are scaled by the amount of “interaction field” (to coin a term) that is present for a given individual, producing metrics of interaction *density*. Notably, the `localPopulationDensity()` method automatically incorporates the mechanics of `clippedIntegral()` into the calculations it performs; see that method’s documentation for further discussion of this concept. This approach can also be useful with the `interactingNeighborCount()` method, provided that the interaction function is of type “f” (since the neighbor count does not depend upon interaction strength).

- `(float)distance(object<Individual>$ receiver, [No<Individual> exerters = NULL])`  
Returns a vector containing distances between `receiver` and the individuals in `exerters`. If `exerters` is `NULL` (the default), then a vector of the distances from `receiver` to all individuals in its subpopulation (including itself) is returned; this case may be handled differently internally, for greater speed, so supplying `NULL` is preferable to supplying the vector of all individuals in the subpopulation explicitly. Otherwise, all individuals in `exerters` must belong to a single subpopulation (but not necessarily the same subpopulation as `receiver`). The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used. If the `InteractionType` is non-spatial, this method may not be called.

Importantly, distances are calculated according to the spatiality of the `InteractionType` (as declared in `initializeInteractionType()`), not the dimensionality of the model as a whole (as declared in `initializeSLIMOptions()`). The distances returned are therefore the distances that would be used to calculate interaction strengths. However, `distance()` will return finite distances for all pairs of individuals, even if the individuals are non-interacting; the `distance()` between an individual and itself will thus be 0. See `interactionDistance()` for an alternative distance definition.

- `(float)distanceFromPoint(float point, object<Individual> exerters)`  
Returns a vector containing distances between the point given by the spatial coordinates in `point`, which may be thought of as the “receiver”, and individuals in `exerters`. The `point` vector is interpreted as providing coordinates precisely as specified by the spatiality of the interaction type; if the interaction type’s spatiality is “xz”, for example, then `point[0]` is assumed to be an x value, and `point[1]` is assumed to be a z value. Be careful; this means that in general it is not safe to pass an individual’s `spatialPosition` property for `point`, for example (although it is safe if the spatiality of the interaction matches the dimensionality of the simulation). A coordinate for a periodic spatial dimension must be within the spatial bounds for that dimension, since coordinates outside of periodic bounds are meaningless (`pointPeriodic()` may be used to ensure this); coordinates for non-periodic spatial dimensions are not restricted. All individuals in `exerters` must belong to a single

subpopulation; the `evaluate()` method must have been previously called for that subpopulation, and positions saved at evaluation time will be used. If the `InteractionType` is non-spatial, this method may not be called. The vector `point` must be exactly as long as the spatiality of the `InteractionType`.

Importantly, distances are calculated according to the spatiality of the `InteractionType` (as declared in `initializeInteractionType()`) not the dimensionality of the model as a whole (as declared in `initializeSLiMOptions()`). The distances are therefore interaction distances: the distances that are used to calculate interaction strengths.

This method replaces the `distanceToPoint()` method that existed prior to SLiM 4.

- `(object<Individual>)drawByStrength(object<Individual>$ receiver, [integer$ count = 1], [No<Subpopulation>$ exorterSubpop = NULL])`

Returns up to `count` individuals drawn from `exorterSubpop`, or if that is `NULL` (the default), then from the subpopulation of `receiver`. The probability of drawing particular individuals is proportional to the strength of interaction they exert upon `receiver`. All exorters must belong to a single subpopulation (but not necessarily the same subpopulation as `receiver`). The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used.

This method may be used with either spatial or non-spatial interactions, but will be more efficient with spatial interactions that set a short maximum interaction distance. Draws are done with replacement, so the same individual may be drawn more than once; sometimes using `unique()` on the result of this call is therefore desirable. If more than one draw will be needed, it is much more efficient to use a single call to `drawByStrength()`, rather than drawing individuals one at a time. Note that if no individuals exert a non-zero interaction strength upon `receiver`, the vector returned will be zero-length; it is important to consider this possibility.

- `(void)evaluate(io<Subpopulation> subpops)`

Snapshots model state in preparation for the use of the interaction, for the receiver and exorter subpopulations specified by `subpops`. The subpopulations may be supplied either as `integer` IDs, or as `Subpopulation` objects. This method will discard all previously cached data for the subpopulation(s), and will cache the current spatial positions of all individuals they contain (so that the spatial positions of those individuals may then change without disturbing the state of the interaction at the moment of evaluation). Particular interaction distances and strengths are not computed by `evaluate()`, and `interaction()` callbacks will not be called in response to this method; that work is deferred until required to satisfy a query (at which point the tick and cycle counters may have advanced, so be careful with the tick ranges used in defining `interaction()` callbacks).

You must explicitly call `evaluate()` at an appropriate time in the tick cycle before the interaction is used, but after any relevant changes have been made to the population. SLiM will invalidate any existing interactions after any portion of the tick cycle in which new individuals have been born or existing individuals have died. In a WF model, this occurs just before `late()` events execute (see the WF tick cycle diagram in chapter 22), so `late()` events are often the appropriate place to put `evaluate()` calls, but `first()` or `early()` events can work too if the interaction is not needed until that point in the tick cycle anyway. In nonWF models, on the other hand, new offspring are produced just before `early()` events and then individuals die just before `late()` events (see the nonWF tick cycle diagram in chapter 23), so interactions will be invalidated twice during each tick cycle. This means that in a nonWF model, an interaction that influences reproduction should usually be evaluated in a `first()` event, while an interaction that influences fitness or mortality should usually be evaluated in an `early()` event (and an interaction that affects both may need to be evaluated at both times).

If an interaction is never evaluated for a given subpopulation, it is guaranteed that there will be essentially no memory or computational overhead associated with the interaction for that

subpopulation. Furthermore, attempting to query an interaction for a receiver or exorter in a subpopulation that has not been evaluated is guaranteed to raise an error.

- (integer)interactingNeighborCount(object<Individual> receivers,  
[No<Subpopulation>\$ exorterSubpop = NULL])

Returns the number of interacting individuals for each individual in `receivers`, within the maximum interaction distance according to the distance metric of the `InteractionType`, from among the exorters in `exorterSubpop` (or, if that is `NULL`, then from among all individuals in the receiver's subpopulation). More specifically, this method counts the number of individuals which can exert an interaction upon each receiver. All of the receivers must belong to a single subpopulation, and all of the exorters must belong to a single subpopulation, but those two subpopulations do not need to be the same. The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used.

This method is similar to `nearestInteractingNeighbors()` (when passed a large count so as to guarantee that all interacting individuals are returned), but this method returns only a count of the interacting individuals, not a vector containing the individuals. This method may also be called in a vectorized fashion, with a non-singleton vector of individuals, unlike `nearestInteractingNeighbors()`.

Note that this method uses interaction eligibility as a criterion; it will not count neighbors that cannot exert an interaction upon a receiver (due to sex-segregation, e.g.). (It also does not count a receiver as a neighbor of itself.) If a count of all neighbors is desired, rather than just interacting neighbors, use `neighborCount()`. If the `InteractionType` is non-spatial, this method may not be called.

- (float)interactionDistance(object<Individual>\$ receiver,  
[No<Individual> exorters = NULL])

Returns a vector containing interaction-dependent distances between `receiver` and individuals in `exorters` that exert an interaction strength upon `receiver`. If `exorters` is `NULL` (the default), then a vector of the interaction-dependent distances from `receiver` to all individuals in its subpopulation (including `receiver` itself) is returned; this case may be handled much more efficiently than if a vector of all individuals in the subpopulation is explicitly provided. Otherwise, all individuals in `exorters` must belong to a single subpopulation (but not necessarily the same subpopulation as `receiver`). The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used. If the `InteractionType` is non-spatial, this method may not be called.

Importantly, distances are calculated according to the spatiality of the `InteractionType` (as declared in `initializeInteractionType()`), not the dimensionality of the model as a whole (as declared in `initializeSLIMOptions()`). The distances returned are therefore the distances that would be used to calculate interaction strengths. In addition, `interactionDistance()` will return `INF` as the distance between `receiver` and any individual which does not exert an interaction upon `receiver`; the `interactionDistance()` between an individual and itself will thus be `INF`, and likewise for pairs excluded from interacting by the sex segregation or max distance of the interaction type. See `distance()` for an alternative distance definition.

- (float)localPopulationDensity(object<Individual> receivers,  
[No<Subpopulation>\$ exorterSubpop = NULL])

Returns a vector of the local population density present at the location of each individual in `receivers`, which does not need to be a singleton; indeed, it can be a vector of all of the individuals in a given subpopulation. However, all receivers must be in the same subpopulation. The local population density is computed from exorters in `exorterSubpop`, or if that is `NULL` (the default), then from the receiver's subpopulation. The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used.

Population density is estimated using interaction strengths, effectively doing a kernel density estimate using the interaction function as the kernel. What is returned is computed as the total interaction strength present at a given point, divided by the integral of the interaction function around that point

after clipping by the spatial bounds of the exorter subpopulation (what one might think of as the amount of “interaction field” around the point). This provides an estimate of local population density, in units of individuals per unit area, as a weighted average over the area covered by the interaction function, where the weight of each exorter in the average is the value of the interaction function at that exorter’s position. This can also be thought of as a measure of the amount of interaction happening per unit of interaction field in the space surrounding the point.

To calculate the clipped integral of the interaction function, this method uses the same numerical estimator used by the `clippedIntegral()` method of `InteractionType`, and all of the caveats described for that method apply here also; notably, all individuals must be within spatial bounds, the maximum interaction distance must be less than half the spatial extent of the subpopulation, and `interaction()` callbacks are not used (and so, for this method, are not allowed to be active). See the documentation for `clippedIntegral()` for further discussion of the details of these calculations.

To calculate the total interaction strength around the position of a receiver, this method uses the same machinery as the `totalOfNeighborStrengths()` method of `InteractionType`, except that – in contrast to other `InteractionType` methods – the interaction strength exerted by the receiver itself is included in the total (if the exorter subpopulation is the receiver’s own subpopulation). This is because population density at the location of an individual includes the individual itself. If this is not desirable, the `totalOfNeighborStrengths()` method should probably be used.

To see the point of this method, consider a receiver located near the edge of the spatial bounds of its subpopulation. Some portion of the interaction function that surrounds that receiver falls outside the spatial bounds of its subpopulation, and will therefore never contain an interacting exorter. If, for example, interaction strengths are used as a measure of competition, this receiver will therefore have an advantage, because it will never feel any competition from the portion of its range that falls outside spatial bounds. However, that portion of its range is presumably also not available to the receiver itself, for foraging or hunting, in which case this advantage is not biologically realistic, but is instead just an undesirable “edge effect” artifact. Dividing by the integral of the interaction function, clipped to the spatial bounds, provides a way to compensate for this edge effect. A nice side effect of using local population densities instead of total interaction strengths is that the maximum interaction strength passed to `setInteractionFunction()` no longer matters; it cancels out when the total interaction strength is divided by the receiver’s clipped integral. However, the *shape* of the interaction function does still matter; it determines the relative weights used for exorters at different distances from the position of the receiver.

- (object<Individual>)nearestInteractingNeighbors(object<Individual>\$ receiver, [integer\$ count = 1], [No<Subpopulation>\$ exorterSubpop = NULL])

Returns up to `count` interacting individuals that are spatially closest to `receiver`, according to the distance metric of the `InteractionType`, from among the exorters in `exorterSubpop` (or, if that is `NULL`, then from among all individuals in the receiver’s subpopulation). More specifically, this method returns only individuals which can exert an interaction upon the receiver. To obtain all of the interacting individuals within the maximum interaction distance of `individual`, simply pass a value for `count` that is greater than or equal to the size of the exorter subpopulation. Note that if fewer than `count` interacting individuals are within the maximum interaction distance, the vector returned may be shorter than `count`, or even zero-length; it is important to check for this possibility even when requesting a single neighbor. If only the number of interacting individuals is needed, use `interactingNeighborCount()` instead. The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used. If the `InteractionType` is non-spatial, this method may not be called.

Note that this method uses interaction eligibility as a criterion; it will not return neighbors that cannot exert an interaction upon the receiver (due to sex-segregation, e.g.). (It will also never return the receiver as a neighbor of itself.) To find all neighbors of the receiver, whether they can interact with it or not, use `nearestNeighbors()`.

- (object<Individual>)nearestNeighbors(object<Individual>\$ receiver, [integer\$ count = 1], [No<Subpopulation>\$ exorterSubpop = NULL])

Returns up to count individuals that are spatially closest to individual, according to the distance metric of the InteractionType, from among the exorters in exorterSubpop (or, if that is NULL, then from among all individuals in the receiver's subpopulation). To obtain all of the individuals within the maximum interaction distance of individual, simply pass a value for count that is greater than or equal to the size of individual's subpopulation. Note that if fewer than count individuals are within the maximum interaction distance, the vector returned may be shorter than count, or even zero-length; it is important to check for this possibility even when requesting a single neighbor. The evaluate() method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used. If the InteractionType is non-spatial, this method may not be called.

Note that this method does not use interaction eligibility as a criterion; it will return neighbors that could not interact with the receiver due to sex-segregation. (It will never return the receiver as a neighbor of itself, however.) To find only neighbors that are eligible to exert an interaction upon the receiver, use nearestInteractingNeighbors().

- (object<Individual>)nearestNeighborsOfPoint(float point, io<Subpopulation>\$ exorterSubpop, [integer\$ count = 1])

Returns up to count individuals in exorterSubpop that are spatially closest to point, according to the distance metric of the InteractionType. The subpopulation may be supplied either as an integer ID, or as a Subpopulation object. To obtain all of the individuals within the maximum interaction distance of point, simply pass a value for count that is greater than or equal to the size of exorterSubpop. Note that if fewer than count individuals are within the maximum interaction distance, the vector returned may be shorter than count, or even zero-length; it is important to check for this possibility even when requesting a single neighbor. The evaluate() method must have been previously called for exorterSubpop, and positions saved at evaluation time will be used. If the InteractionType is non-spatial, this method may not be called.

- (integer)neighborCount(object<Individual> receivers, [No<Subpopulation>\$ exorterSubpop = NULL])

Returns the number of neighbors for each individual in receivers, within the maximum interaction distance according to the distance metric of the InteractionType, from among the individuals in exorterSubpop (or, if that is NULL, then from among all individuals in the receiver's subpopulation). All of the receivers must belong to a single subpopulation, and all of the exorters must belong to a single subpopulation, but those two subpopulations do not need to be the same. The evaluate() method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used.

This method is similar to nearestNeighbors() (when passed a large count so as to guarantee that all neighbors are returned), but this method returns only a count of the individuals, not a vector containing the individuals. This method may also be called in a vectorized fashion, with a non-singleton vector of individuals, unlike nearestNeighbors().

Note that this method does not use interaction eligibility as a criterion; it will count neighbors that cannot exert an interaction upon a receiver (due to sex-segregation, e.g.). (It still does not count a receiver as a neighbor of itself, however.) If a count of only interacting neighbors is desired, use interactingNeighborCount(). If the InteractionType is non-spatial, this method may not be called.

- (integer\$)neighborCountOfPoint(float point, io<Subpopulation>\$ exorterSubpop)

Returns the number of individuals in exorterSubpop that are within the maximum interaction distance according to the distance metric of the InteractionType. The subpopulation may be supplied either as an integer ID, or as a Subpopulation object. The evaluate() method must have been previously called for exorterSubpop, and positions saved at evaluation time will be used. If the InteractionType is non-spatial, this method may not be called.



This method is similar to `nearestNeighborsOfPoint()` (when passed a large count so as to guarantee that all neighbors are returned), but this method returns only a count of the individuals, not a vector containing the individuals.

– `(void)setInteractionFunction(string$ functionType, ...)`

Set the function used to translate spatial distances into interaction strengths for an interaction type. The `functionType` may be "f", in which case the ellipsis ... should supply a `numeric$` fixed interaction strength; "l", in which case the ellipsis should supply a `numeric$` maximum strength for a linear function; "e", in which case the ellipsis should supply a `numeric$` maximum strength and a `numeric$` lambda (shape) parameter for a negative exponential function; "n", in which case the ellipsis should supply a `numeric$` maximum strength and a `numeric$` sigma (standard deviation) parameter for a Gaussian function; or "c", in which case the ellipsis should supply a `numeric$` maximum strength and a `numeric$` scale parameter for a Cauchy distribution function. See section 24.8 above for discussions of these interaction functions. Non-spatial interactions must use function type "f", since no distance values are available in that case.

The interaction function for an interaction type is normally a constant in simulations; in any case, it cannot be changed when an interaction has already been evaluated, so either it should be set prior to evaluation, or `unevaluate()` should be called first.

– `(float)strength(object<Individual>$ receiver, [No<Individual> exerters = NULL])`

Returns a vector containing the interaction strengths exerted upon `receiver` by the individuals in `exerters`. If `exerters` is `NULL` (the default), then a vector of the interaction strengths exerted by all individuals in the subpopulation of `receiver` (including `receiver` itself) is returned; this case may be handled much more efficiently than if a vector of all individuals in the subpopulation is explicitly provided. Otherwise, all individuals in `exerters` must belong to a single subpopulation (but not necessarily the same subpopulation as `receiver`). The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used.

If the strengths of interactions exerted by a single individual upon multiple individuals are needed instead (the inverse of what this method provides), multiple calls to this method will be necessary, one per pairwise interaction queried; the interaction engine is not optimized for the inverse case, and so it will likely be quite slow to compute. If the interaction is reciprocal and sex-symmetric, the opposite query should provide identical results in a single efficient call (because then the interactions exerted are equal to the interactions received); otherwise, the best approach might be to define a second interaction type representing the inverse interaction that you wish to be able to query efficiently.

– `(float)totalOfNeighborStrengths(object<Individual> receivers, [No<Subpopulation>$ exorterSubpop = NULL])`

Returns a vector of the total interaction strength felt by each individual in `receivers` by the exerters in `exorterSubpop` (or, if that is `NULL`, then by all individuals in the receiver's subpopulation). The `receivers` parameter does not need to be a singleton; indeed, it can be a vector of all of the individuals in a given subpopulation. All of the receivers must belong to a single subpopulation, and all of the exerters must belong to a single subpopulation, but those two subpopulations do not need to be the same. The `evaluate()` method must have been previously called for the receiver and exorter subpopulations, and positions saved at evaluation time will be used. If the `InteractionType` is non-spatial, this method may not be called.

For one individual, this is essentially the same as calling `nearestInteractingNeighbors()` with a large count so as to obtain the complete vector of all interacting neighbors, calling `strength()` for each of those interactions to get each interaction strength, and adding those interaction strengths together with `sum()`. This method is much faster than that implementation, however, since all of that work is done as a single operation. Also, `totalOfNeighborStrengths()` can total up interactions for more than one receiver in a single call.



Similarly, for one individual this is essentially the same as calling `strength()` to get the interaction strengths between a receiver and all individuals in the exorter subpopulation, and then calling `sum()`. Again, this method should be much faster, since this algorithm looks only at neighbors, whereas calling `strength()` directly assesses interaction strengths with all other individuals. This will make a particularly large difference when the subpopulation size is large and the maximum distance of the `InteractionType` is small.

See `localPopulationDensity()` for a related method that calculates the total interaction strength divided by the amount of “interaction field” present for an individual (i.e., the integral of the interaction function clipped to the spatial bounds of the subpopulation) to provide an estimate of the “interaction density” felt by an individual.

– `(void)unevaluate(void)`

Discards all evaluation of this interaction, for all subpopulations. The state of the `InteractionType` is reset to a state prior to evaluation. This can be useful if the model state has changed in such a way that the evaluation already conducted is no longer valid. For example, if the maximum distance or the interaction function of the `InteractionType` need to be changed with immediate effect, or if the data used by an `interaction()` callback has changed in such a way that previously calculated interaction strengths are no longer correct, `unevaluate()` allows the interaction to begin again from scratch.

In WF models, all interactions are automatically reset to an unevaluated state at the moment when the new offspring generation becomes the parental generation (at step 4 in the tick cycle; see section 22.4).

In nonWF models, all interactions are automatically reset to an unevaluated state twice per tick: immediately after `reproduction()` callbacks have completed (after step 1 in the tick cycle; see section 23.1), and immediately before viability/survival selection (before step 4 in the tick cycle; see section 23.4).

Given this automatic invalidation, most simulations have no reason to call `unevaluate()`.

## 24.9 Class `LogFile`

*Superclass: `Dictionary`*

`LogFile` is a class that can, optionally, be used to log out a table of information about the running simulation to a text file. The information logged out is completely configurable with *generators*, including the ability to use custom Eidos code as a generator. The resulting file can be formatted with comma separators (a CSV file), tab separators (a TSV file), or with any custom separator string between data values. The file can be plain text or can be compressed in `.gz` format (decompressible at the command line with the `gunzip` utility, among other tools). When compression is enabled, writes out to disk are buffered in memory for better performance and smaller file size. A new `LogFile` object attached to the simulation can be created with the `Community` method `createLogFile()`, discussed in section 24.3.2. Any number of `LogFile` objects can be active simultaneously, writing to different files.

Logging can be done automatically, at the end of ticks at a given periodic interval (e.g., every 10th tick). This automatic logging is optional, and a log row can always be generated explicitly by calling `logRow()`. Flushing compressed data to disk can be done automatically after a given number of rows have been generated, or can be requested explicitly with a call to `flush()`.

Generators always generate a single value in each logged row, resulting in a single column of data in the log file. Some built-in generators are provided by `LogFile` for the most common cases; these can be added to a given `LogFile` by calling the `add...()` methods documented below. A generator with custom Eidos code can be added with `addCustomColumn()`, and a generator that expects to be supplied with the value to write, rather than generating the value itself, can be added with `addSuppliedColumn()`. `LogFile` expects to be fully configured, with calls to `add...()` methods to add generators, before the first row of data is written out, to ensure consistency in the

file's data. When the first row of data is written (or buffered), the `LogFile`'s configuration will then be frozen, and calls to `add...()` will no longer be allowed. Columns will be written, in each row, in the order in which `add...()` calls are made on the `LogFile`, and they will be named in the file's header line as documented in those methods. It is an error to have two columns with the same name.

`LogFile` is a subclass of the Eidos class `Dictionary`, but unlike other SLiM classes that are `Dictionary` subclasses, this does not allow you to attach arbitrary key-value pairs to the object. Instead, `LogFile` uses its `Dictionary`-ness to make the data from the last logged row available through `getValue()`, using the name of the generator (i.e., the name of the data column) as the key. One quirk of `LogFile` is that because its keys are defined by the columns that it generates, and columns can sometimes contain an NA value, `LogFile`'s dictionary can, in effect, contain NULL values (representing NA); this is not normally allowed by `Dictionary`. This should cause no trouble; just be aware that `getValue()` on a `LogFile` might return NULL for a key, representing NA, and that the result of `serialize()` might contain NA values.

Finally: as a general rule, if a `Subpopulation` is referenced by one of the `add...X()` methods below it may be supplied as an object if it already exists (p1), or by id (1) even if it does not yet exist. The resulting column will generally have a name of the form `pX_colname`, where X is the id of the specified subpopulation. If a subpopulation-specific data logger refers to a `Subpopulation` that does not exist at the time a given row is logged, NA will be written.

#### 24.9.1 *LogFile properties*

`allKeys => (string)`

This `Dictionary` property has an override in `LogFile` to return, in order, the names of the log file columns, which are the keys for `LogFile`'s dictionary.

`filePath => (string$)`

The path of the log file being written to. This may be changed with `setFilePath()`.

`logInterval => (integer$)`

The interval for automatic logging; a new row of data will be logged every `logInterval` ticks. This may be changed with `setLogInterval()`. If automatic logging has been disabled, this property will be 0.

`precision <=> (integer$)`

The precision of float output. To be exact, `precision` specifies the preferred number of significant digits that will be output for float values. The default is 6; values in [1,22] are legal, but 17 is probably the largest value that makes sense given the limits of double-precision floating point.

`tag <=> (integer$)`

A user-defined integer value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use.

#### 24.9.2 *LogFile methods*

– (void)`addCustomColumn(string$ columnName, string$ source, [* context = NULL])`

Adds a new data column with its name provided by `columnName`. The value for the column, when a given row is generated, will be produced by the code supplied in `source`, which is expected to return either NULL (which will write out NA), or a singleton value of any non-object type.

The `context` parameter will be set up as a pseudo-parameter, named `context`, when `source` is called, allowing the same source code to be used to generate values for multiple data columns; you might, for example, provide the particular `Subpopulation` object here that you wish `source` to use

for its calculations. This is optional; if the default value of `NULL` is used, then `context` will be `NULL` when `source` is called.

See `addMeanSDColumns()` for a useful variant.

- `(void)addCycle([No<Species>$ species = NULL])`  
Adds a new data column that provides the cycle counter for `species` (the same as the value of the `cycle` property of that species). In single-species models, `species` may be `NULL` to indicate that single species. The column will simply be named `cycle` in single-species models; an underscore and the name of the species will be appended in multispecies models.
- `(void)addCycleStage(void)`  
Adds a new data column that provides the cycle stage, named `cycle_stage`. The stage is provided as a `string`, and will typically be "first", "early", "late", or "end" (the latter used for the point in time at which end-of-tick automatic logging occurs). Other possible values are discussed in the documentation for the `cycleStage` property of `Community`, which this column reflects.
- `(void)addKeysAndValuesFrom(object$ source)`  
This `Dictionary` method has an override in `LogFile` to make it illegal to call, since `LogFile` manages its `Dictionary` entries.
- `(void)addMeanSDColumns(string$ columnName, string$ source, [* context = NULL])`  
Adds two new data columns with names of `columnName_mean` and `columnName_sd`. When a given row is generated, the code supplied in `source` is expected to return either a zero-length vector of any type including `NULL` (which will write out `NA` to both columns), or a non-zero-length vector of `integer` or `float` values. In the latter case, the result vector will be summarized in the two columns by its mean and standard deviation respectively. If the result vector has exactly one value, the standard deviation will be written as `NA`. The `context` parameter is set up as a pseudo-parameter when `source` is called, as described in `addCustomColumn()`.
- `(void)addPopulationSexRatio([No<Species>$ species = NULL])`  
Adds a new data column that provides the population sex ratio  $M:(M+F)$  for `species`. In single-species models, `species` may be `NULL` to indicate that single species. The column will simply be named `sex_ratio` in single-species models; an underscore and the name of the species will be appended in multispecies models. If the species is hermaphroditic, `NA` will be written.
- `(void)addPopulationSize([No<Species>$ species = NULL])`  
Adds a new data column that provides the total population size for `species`. In single-species models, `species` may be `NULL` to indicate that single species. The column will simply be named `num_individuals` in single-species models; an underscore and the name of the species will be appended in multispecies models.
- `(void)addSubpopulationSexRatio(io<Subpopulation>$ subpop)`  
Adds a new data column that provides the sex ratio  $M:(M+F)$  of the subpopulation `subpop`, named `pX_sex_ratio`. If the subpopulation exists but has a size of zero, `NA` will be written.
- `(void)addSubpopulationSize(io<Subpopulation>$ subpop)`  
Adds a new data column that provides the size of the subpopulation `subpop`, named `pX_num_individuals`. If the subpopulation exists but has a size of zero, `0` will be written.
- `(void)addSuppliedColumn(string$ columnName)`  
Adds a new data column with its name provided by `columnName`. The value for the column is initially undefined, and will be written as `NA`. A different value may (optionally) be provided by calling `setSuppliedValue()` with a value for `columnName`. That value will be used for the column the next time a row is generated (whether automatically or by a call to `logRow()`), and the column's value will subsequently be undefined again. In other words, for any given logged row the default of `NA` may be

kept, or a different value may be supplied. This allows the value for the column to be set at any point during the tick cycle, which can be convenient if the column's value depends upon transient state that is no longer available at the time the row is logged.

- (void)addTick(void)  
Adds a new data column, named `tick`, that provides the tick number for the simulation.
- (void)clearKeysAndValues(void)  
This Dictionary method has an override in `LogFile` to make it illegal to call, since `LogFile` manages its Dictionary entries.
- (void)flush(void)  
Flushes all buffered data to the output file, synchronously. This will make the contents of the file on disk be up-to-date with the running simulation. Flushing frequently may entail a small performance penalty. More importantly, if `.gz` compression has been requested with `compress=T` the size of the resulting file will be larger – potentially much larger – if `flush()` is called frequently. Note that automatic periodic flushing can be requested with the `flushInterval` parameter to `createLogFile()`.
- (void)logRow(void)  
This logs a new row of data, by evaluating all of the generators added to the `LogFile` with `add...()` calls. Note that the new row may be buffered, and thus may not be written out to disk immediately; see `flush()`. This method may be used instead of, or in conjunction with, automatic logging.  
You can get the `LogFile` instance, in order to call `logRow()` on it, from `community.logFiles`, or you can remember it in a global constant with `defineConstant()`.
- (void)setLogInterval([Ni\$ logInterval = NULL])  
Sets the automatic logging interval. A `logInterval` of `NULL` stops automatic logging immediately. Other values request that a new row should be logged (as if `logRow()` were called) at the end of every `logInterval` ticks (just before the tick count increment, in both WF and nonWF models), starting at the end of the tick in which `setLogInterval()` was called.
- (void)setFilePath(string\$ filePath, [Ns initialContents = NULL], [logical\$ append = F], [Nl\$ compress = NULL], [Ns\$ sep = NULL])  
Redirects the `LogFile` to write new rows to a new `filePath`. Any rows that have been buffered but not flushed will be written to the previous file first, as if `flush()` had been called. With this call, new `initialContents` may be supplied, which will either replace any existing file or will be appended to it, depending upon the value of `append`. New values may be supplied for `compress` and `sep`; the meaning of these parameters is identical to their meaning in `createLogFile()`, except that a value of `NULL` for these means “do not change this setting from its previous value”. In effect, then, this method lets you start a completely new log file at a new path, without having to create and configure a new `LogFile` object. The new file will be created (or appended) synchronously, with the specified initial contents.
- (void)setSuppliedValue(string\$ columnName, +\$ value)  
Registers a value, passed in `value`, to be used for the supplied column named `columnName` when a row is next logged. This column must have been added with `addSuppliedColumn()`. A value of `NULL` may be passed to log NA, but logging NA is the default behavior for supplied columns in any case. Otherwise, the value must be a singleton, and its type should match the values previously supplied for the column (otherwise the log file may be difficult to parse, since the values within the column will not be of one consistent type). See `addSuppliedColumn()` for further details.
- (void)setValue(string\$ key, \* value)  
This Dictionary method has an override in `LogFile` to make it illegal to call, since `LogFile` manages its Dictionary entries.

- `(logical$)willAutolog(void)`  
Returns T if the log file is configured to log a new row automatically at the end of the current tick; otherwise, returns F. This is useful for calculating a value that will be logged only in ticks when the value is needed.

## 24.10 Class Mutation

*Superclass: Dictionary*

This class represents a single point mutation. Mutations can be shared by the genomes of many individuals; if they reach fixation, they are converted to `Substitution` objects.

Although `Mutation` has a `tag` property, like most SLiM classes, the `subpopID` can also store custom values if you don't need to track the origin subpopulation of mutations (see below).

Section 1.5.2 presents an overview of the conceptual role of this class.

### 24.10.1 Mutation properties

`id => (integer$)`

The identifier for this mutation. Each mutation created during a run receives an immutable identifier that will be unique across the duration of the run. These identifiers are not re-used during a run, except that if a population file is loaded from disk, the loaded mutations will receive their original identifier values as saved in the population file.

`isFixed => (logical$)`

T if the mutation has fixed (in the SLiM sense of having been converted to a `Substitution` object), F otherwise. Since fixed/substituted mutations are removed from the simulation, you will only see this flag be T if you have held onto a mutation beyond its usual lifetime (see section 27.2).

`isSegregating => (logical$)`

T if the mutation is segregating (in the SLiM sense of not having been either lost or converted to a `Substitution` object), F otherwise. Since both lost and fixed/substituted mutations are removed from the simulation, you will only see this flag be F if you have held onto a mutation beyond its usual lifetime (see section 27.2). Note that if `isSegregating` is F, `isFixed` will let you determine whether the mutation is no longer segregating because it was lost, or because it fixed.

`mutationType => (object<MutationType>$)`

The `MutationType` from which this mutation was drawn.

`nucleotide <=> (string$)`

A string representing the nucleotide associated with this mutation; this will be "A", "C", "G", or "T". If the mutation is not nucleotide-based, this property is unavailable.

`nucleotideValue <=> (integer$)`

An integer representing the nucleotide associated with this mutation; this will be 0 (A), 1 (C), 2 (G), or 3 (T). If the mutation is not nucleotide-based, this property is unavailable.

`originTick => (integer$)`

The tick in which this mutation arose.

`position => (integer$)`

The position in the chromosome of this mutation.

`selectionCoeff => (float$)`

The selection coefficient of the mutation, drawn from the distribution of fitness effects of its `MutationType`. If a mutation has a `selectionCoeff` of *s*, the multiplicative fitness effect of the



mutation in a homozygote is  $1+s$ ; in a heterozygote it is  $1+hs$ , where  $h$  is the dominance coefficient kept by the mutation type (see section 24.11.1).

Note that this property has a quirk: it is stored internally in SLiM using a single-precision float, not the double-precision float type normally used by Eidos. This means that if you set a mutation `mut`'s selection coefficient to some number  $x$ , `mut.selectionCoeff==x` may be `F` due to floating-point rounding error. Comparisons of floating-point numbers for exact equality is often a bad idea, but this is one case where it may fail unexpectedly. Instead, it is recommended to use the `id` or `tag` properties to identify particular mutations.

**subpopID**  $\leftrightarrow$  (integer\$)

The identifier of the subpopulation in which this mutation arose. This property can be used to track the ancestry of mutations through their subpopulation of origin. For an overview of other ways of tracking genetic ancestry, including true local ancestry at each position on the chromosome, see sections 1.7 and 14.7.

If you don't care which subpopulation a mutation originated in, the **subpopID** may be used as an arbitrary integer "tag" value for any purpose you wish; SLiM does not do anything with the value of **subpopID** except propagate it to **Substitution** objects and report it in output. (It must still be  $\geq 0$ , however, since SLiM object identifiers are limited to nonnegative integers).

**tag**  $\leftrightarrow$  (integer\$)

A user-defined integer value. The value of **tag** is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of **tag** is not used by SLiM; it is free for you to use.

#### 24.10.2 Mutation methods

– (void)setMutationType(io<MutationType>\$ mutType)

Set the mutation type of the mutation to **mutType** (which may be specified as either an integer identifier or a **MutationType** object). This implicitly changes the dominance coefficient of the mutation to that of the new mutation type, since the dominance coefficient is a property of the mutation type. On the other hand, the selection coefficient of the mutation is not changed, since it is a property of the mutation object itself; it can be changed explicitly using the **setSelectionCoeff()** method if so desired.

The mutation type of a mutation is normally a constant in simulations, so be sure you know what you are doing. Changing this will normally affect the fitness values calculated toward the end of the current tick; if you want current fitness values to be affected, you can call the **Species** method **recalculateFitness()** – but see the documentation of that method for caveats.

In nucleotide-based models, a restriction applies: nucleotide-based mutations may not be changed to a non-nucleotide-based mutation type, and non-nucleotide-based mutations may not be changed to a nucleotide-based mutation type.

– (void)setSelectionCoeff(float\$ selectionCoeff)

Set the selection coefficient of the mutation to **selectionCoeff**. The selection coefficient will be changed for all individuals that possess the mutation, since they all share a single **Mutation** object (note that the dominance coefficient will remain unchanged, as it is determined by the mutation type).

This is normally a constant in simulations, so be sure you know what you are doing; often setting up a **mutationEffect()** callback (see section 25.2) is preferable, in order to modify the selection coefficient in a more limited and controlled fashion (see section 10.5 for further discussion of this point). Changing this will normally affect the fitness values calculated toward the end of the current tick; if you want current fitness values to be affected, you can call the **Species** method **recalculateFitness()** – but see the documentation of that method for caveats.



## 24.11 Class `MutationType`

*Superclass: Dictionary*

This class represents a type of mutation with a particular distribution of fitness effects, such as neutral mutations or weakly beneficial mutations. Sections 1.5.3 and 1.5.4 present an overview of the conceptual role of this class. The mutation types currently defined in the simulation are defined as global constants with the same names used in the SLiM input file – `m1`, `m2`, and so forth.

There are currently six options for the distribution of fitness effects in SLiM, represented by single-character codes:

"f" – A **fixed** fitness effect. This DFE type has a single parameter, the selection coefficient  $s$  to be used by all mutations of the mutation type.

"g" – A **gamma**-distributed fitness effect. This DFE type is specified by two parameters, a mean value and a shape parameter. The gamma distribution from which mutations are drawn is given by the probability density function  $P(s | \alpha, \beta) = [\Gamma(\alpha)\beta^\alpha]^{-1}s^{\alpha-1}\exp(-s/\beta)$ , where  $\alpha$  is the shape parameter, and the specified mean for the distribution is equal to  $\alpha\beta$ . Note that this parameterization is the same as for the Eidos function `rgamma()`. A gamma distribution is often used to model deleterious mutations at functional sites.

"e" – An **exponentially**-distributed fitness effect. This DFE type is specified by a single parameter, the mean of the distribution. The exponential distribution from which mutations are drawn is given by the probability density function  $P(s | \beta) = \beta^{-1}\exp(-s/\beta)$ , where  $\beta$  is the specified mean for the distribution. This parameterization is the same as for the Eidos function `rexp()`. An exponential distribution is often used to model beneficial mutations.

"n" – A **normally**-distributed fitness effect. This DFE type is specified by two parameters, a mean and a standard deviation. The normal distribution from which mutations are drawn is given by the probability density function  $P(s | \mu, \sigma) = (2\pi\sigma^2)^{-1/2}\exp(-(s-\mu)^2/2\sigma^2)$ , where  $\mu$  is the mean and  $\sigma$  is the standard deviation. This parameterization is the same as for the Eidos function `rnorm()`. A normal distribution is often used to model mutations that can be either beneficial or deleterious, since both tails of the distribution are unbounded.

"w" – A **Weibull**-distributed fitness effect. This DFE type is specified by a scale parameter and a shape parameter. The Weibull distribution from which mutations are drawn is given by the probability density function  $P(s | \lambda, k) = (k/\lambda^k)s^{k-1}\exp(-(s/\lambda)^k)$ , where  $\lambda$  is the scale parameter and  $k$  is the shape parameter. This parameterization is the same as for the Eidos function `rweibull()`. A Weibull distribution is often used to model mutations following extreme-value theory.

"s" – A **script**-based fitness effect. This DFE type is specified by a script parameter of type string, specifying an Eidos script to be executed to produce each new selection coefficient. For example, the script `"return rbinom(1);"` could be used to generate selection coefficients drawn from a binomial distribution, using the Eidos function `rbinom()`, even though that mutational distribution is not supported by SLiM directly. The script must return a singleton float or integer.

Note that these distributions can in principle produce selection coefficients smaller than  $-1.0$ . In that case, the mutations will be evaluated as "lethal" by SLiM, and the relative fitness of the individual will be set to  $0.0$ .

### 24.11.1 `MutationType` properties

`color <=> (string$)`

The color used to display mutations of this type in SLiMgui. Outside of SLiMgui, this property still exists, but is not used by SLiM. Colors may be specified by name, or with hexadecimal RGB values of the form `"#RRGGBB"` (see the Eidos manual). If `color` is the empty string, `""`, SLiMgui's default (selection-coefficient-based) color scheme is used; this is the default for new `MutationType` objects.

`colorSubstitution <=> (string$)`

The color used to display substitutions of this type in SLiMgui (see the discussion for the `colorSubstitution` property of the `Chromosome` class for details). Outside of SLiMgui, this property still exists, but is not used by SLiM. Colors may be specified by name, or with hexadecimal RGB values of the form `"#RRGGBB"` (see the Eidos manual). If `colorSubstitution` is the empty string, `""`, SLiMgui's default (selection-coefficient-based) color scheme is used; this is the default for new `MutationType` objects.

`convertToSubstitution <=> (logical$)`

This property governs whether mutations of this mutation type will be converted to `Substitution` objects when they reach fixation.

In WF models this property is `T` by default, since conversion to `Substitution` objects provides large speed benefits; it should be set to `F` only if necessary, and only on the mutation types for which it is necessary. This might be needed, for example, if you are using a `mutationEffect()` callback to implement an epistatic relationship between mutations; a mutation epistatically influencing the fitness of other mutations through a `mutationEffect()` callback would need to continue having that influence even after reaching fixation, but if the simulation were to replace the fixed mutation with a `Substitution` object the mutation would no longer be considered in fitness calculations (unless the callback explicitly consulted the list of `Substitution` objects kept by the simulation). Other script-defined behaviors in `mutationEffect()`, `interaction()`, `mateChoice()`, `modifyChild()`, and `recombination()` callbacks might also necessitate the disabling of substitution for a given mutation type; this is an important consideration to keep in mind. See section 22.3 for further discussion of `convertToSubstitution` in WF models.

In contrast, for nonWF models this property is `F` by default, because even mutations with no epistasis or other indirect fitness effects will continue to influence the survival probabilities of individuals. For nonWF models, only neutral mutation types with no epistasis or other side effects can safely be converted to substitutions upon fixation. When such a pure-neutral mutation type is defined in a nonWF model, this property should be set to `T` to tell SLiM that substitution is allowed; this may have very large positive effects on performance, so it is important to remember when modeling background neutral mutations. See section 23.5 for further discussion of `convertToSubstitution` in nonWF models.

SLiM consults this flag at the end of each tick when deciding whether to substitute each fixed mutation. If this flag is `T`, all eligible fixed mutations will be converted at the end of the current tick, even if they were previously left unconverted because of the previous value of the flag. Setting this flag to `F` will prevent future substitutions, but will not cause any existing `Substitution` objects to be converted back into `Mutation` objects.

`distributionParams => (float)`

The parameters that configure the chosen distribution of fitness effects. This will be of type `string` for DFE type `"s"`, and type `float` for all other DFE types.

`distributionType => (string$)`

The type of distribution of fitness effects; one of `"f"`, `"g"`, `"e"`, `"n"`, `"w"`, or `"s"` (see section 24.11, above).

`dominanceCoeff <=> (float$)`

The dominance coefficient used for mutations of this type when heterozygous. Changing this will normally affect the fitness values calculated toward the end of the current tick; if you want current fitness values to be affected, you can call the `Species` method `recalculateFitness()` – but see the documentation of that method for caveats.

Note that the dominance coefficient is not bounded. A dominance coefficient greater than `1.0` may be used to achieve an overdominance effect. By making the selection coefficient very small and the

dominance coefficient very large, an overdominance scenario in which both homozygotes have the same fitness may be approximated, to a nearly arbitrary degree of precision.

Note that this property has a quirk: it is stored internally in SLiM using a single-precision float, not the double-precision float type normally used by Eidos. This means that if you set a mutation type `muttype`'s dominance coefficient to some number `x`, `muttype.dominanceCoeff==x` may be `F` due to floating-point rounding error. Comparisons of floating-point numbers for exact equality is often a bad idea, but this is one case where it may fail unexpectedly. Instead, it is recommended to use the `id` or `tag` properties to identify particular mutation types.

`haploidDominanceCoeff <=> (float$)`

The dominance coefficient used for mutations of this type when they occur opposite a null genome (as in sex-chromosome models and models involving haploids). This defaults to `1.0`, and is used only in models where null genomes are present; the `dominanceCoeff` property is the dominance coefficient used in most circumstances. Changing this will normally affect the fitness values calculated toward the end of the current tick; if you want current fitness values to be affected, you can call the `Species` method `recalculateFitness()` – but see the documentation of that method for caveats.

As with the `dominanceCoeff` property, this is stored internally using a single-precision float; see the documentation for `dominanceCoeff` for discussion.

`id => (integer$)`

The identifier for this mutation type; for mutation type `m3`, for example, this is `3`.

`mutationStackGroup <=> (integer$)`

The group into which this mutation type belongs for purposes of mutation stacking policy. This is equal to the mutation type's `id` by default. See `mutationStackPolicy`, below, for discussion.

In nucleotide-based models, the stacking group for nucleotide-based mutation types is always `-1`, and cannot be changed. Non-nucleotide-based mutation types may also be set to share the `-1` stacking group, if they should participate in the same stacking policy as nucleotide-based mutations, but that would be quite unusual.

`mutationStackPolicy <=> (string$)`

This property and the `mutationStackGroup` property together govern whether mutations of this mutation type's stacking group can “stack” – can occupy the same position in a single individual. A set of mutation types with the same value for `mutationStackGroup` is called a “stacking group”, and all mutation types in a given stacking group must have the same `mutationStackPolicy` value, which defines the stacking behavior of all mutations of the mutation types in the stacking group. In other words, one stacking group might allow its mutations to stack, while another stacking group might not, but the policy within each stacking group must be unambiguous.

This property is `"s"` by default, indicating that mutations in this stacking group should be allowed to stack without restriction. If the policy is set to `"f"`, the *first* mutation of stacking group at a given site is retained; further mutations of this stacking group at the same site are discarded with no effect. This can be useful for modeling one-way changes; once a gene is disabled by a premature stop codon, for example, you might wish to assume, for simplicity, that further mutations cannot alter that fact. If the policy is set to `"l"`, the *last* mutation of this stacking group at a given site is retained; earlier mutation of this stacking group at the same site are discarded. This can be useful for modeling an “infinite-alleles” scenario in which every new mutation at a site generates a completely new allele, rather than retaining the previous mutations at the site.

The mutation stacking policy applies only within the given mutation type's stacking group; mutations of different stacking groups are always allowed to stack in SLiM. The policy applies to all mutations added to the model after the policy is set, whether those mutations are introduced by calls such as `addMutation()`, `addNewMutation()`, or `addNewDrawnMutation()`, or are added by SLiM's own mutation-generation machinery. However, no attempt is made to enforce the policy for mutations already existing at the time the policy is set; typically, therefore, the policy is set in an `initialize()`

callback so that it applies throughout the simulation. The policy is also not enforced upon the mutations loaded from a file with `readFromPopulationFile()`; such mutations were governed by whatever stacking policy was in effect when the population file was generated.

In nucleotide-based models, the stacking policy for nucleotide-based mutation types is always "l", and cannot be changed. This ensures that new nucleotide mutations always replace the previous nucleotide at a site, and that more than one nucleotide mutation is never present at the same position in a single genome.

**nucleotideBased => (logical\$)**

If the mutation type was created with `initializeMutationType()`, it is not nucleotide-based, and this property is F. If it was created with `initializeMutationTypeNuc()`, it is nucleotide-based, and this property is T. See those methods for further discussion.

**species => (object<Species>\$)**

The species to which the target object belongs.

**tag <=> (integer\$)**

A user-defined integer value. The value of **tag** is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of **tag** is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the `Dictionary` class; see the Eidos manual), for another way of attaching state to mutation types.

#### 24.11.2 *MutationType* methods

– **(float)drawSelectionCoefficient([integer\$ n = 1])**

Draws and returns a vector of n selection coefficients using the currently defined distribution of fitness effects (DFE) for the target mutation type. See section 24.11 above for discussion of the supported distributions and their uses. If the DFE is type "s", this method will result in synchronous execution of the DFE's script.

– **(void)setDistribution(string\$ distributionType, ...)**

Set the distribution of fitness effects for a mutation type. The **distributionType** may be "f", in which case the ellipsis ... should supply a numeric\$ fixed selection coefficient; "e", in which case the ellipsis should supply a numeric\$ mean selection coefficient for the exponential distribution; "g", in which case the ellipsis should supply a numeric\$ mean selection coefficient and a numeric\$ alpha shape parameter for a gamma distribution; "n", in which case the ellipsis should supply a numeric\$ mean selection coefficient and a numeric\$ sigma (standard deviation) parameter for a normal distribution; "w", in which case the ellipsis should supply a numeric\$  $\lambda$  scale parameter and a numeric\$ k shape parameter for a Weibull distribution; or "s", in which case the ellipsis should supply a string\$ Eidos script parameter. See section 24.11 above for discussions of these distributions and their uses. The DFE for a mutation type is normally a constant in simulations, so be sure you know what you are doing.

### 24.12 Class **SLiMEidosBlock**

*Superclass: Dictionary*

This class represents a block of Eidos code registered in a SLiM simulation. All Eidos events and Eidos callbacks defined in the SLiM input file of the current simulation are instantiated as **SLiMEidosBlock** objects and are available through the `allScriptBlocks` property of `Community` and the `scriptBlocks` property of `Species`; see sections 24.3.1 and 24.14.1. In addition, new script blocks can be created programmatically and registered with the simulation, and registered script blocks can be deregistered; see the `-register...()` and `-deregisterScriptBlock()` methods of

Community and Species in sections 24.3.2 and 24.14.2. The currently executing script block is available through the `self` global; see section 25.11.

### 24.12.1 *SLiMEidosBlock* properties

`active <=> (integer$)`

If this evaluates to logical F (i.e., is equal to 0), the script block is inactive and will not be called.

The value of `active` for all registered script blocks is reset to -1 at the beginning of each tick, prior to script events being called, thus activating all blocks (except callbacks associated with a species that is not active in that tick, which are deactivated as part of the deactivation of the species). Any `integer` value other than -1 may be used instead of -1 to represent that a block is active; for example, `active` may be used as a counter to make a block execute a fixed number of times in each tick. This value is not cached by SLiM; if it is changed, the new value takes effect immediately. For example, a callback might be activated and inactivated repeatedly during a single tick.

`end => (integer$)`

The last tick in which the script block is active.

`id => (integer$)`

The identifier for this script block; for script `s3`, for example, this is 3. A script block for which no `id` was given will have an `id` of -1.

`source => (string$)`

The source code string of the script block.

`speciesSpec => (object<Species>)`

The `species` specifier for the script block. The species specifier for a callback block indicates the callback's associated species; the callback is called to modify the default behavior for that species. If the script block has no `species` specifier, this property's value is a zero-length `object` vector of class `Species`. This property is read-only; normally it is set by preceding the definition of a callback with a `species` specifier, of the form `species <species-name>`.

`start => (integer$)`

The first tick in which the script block is active.

`tag <=> (integer$)`

A user-defined `integer` value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use.

`ticksSpec => (object<Species>)`

The `ticks` specifier for the script block. The `ticks` specifier for an event block indicates the event's associated species; the event executes only in ticks when that species is active. If the script block has no `ticks` specifier, this property's value is a zero-length `object` vector of class `Species`. This property is read-only; normally it is set by preceding the definition of an event with a `ticks` specifier, of the form `ticks <species-name>`.

`type => (string$)`

The type of the script block; this will be "first", "early", or "late" for the three types of Eidos events, or "initialize", "fitnessEffect", "interaction", "mateChoice", "modifyChild", "mutation", "mutationEffect", "recombination", "reproduction", or "survival" for the respective types of Eidos callbacks (see section 24.1 and chapter 25).



### 24.12.2 SLiMEidosBlock methods

SLiMEidosBlock provides no methods to modify a script block. You may, however, reschedule a block to run in a different set of ticks using the `rescheduleScriptBlock()` method of `Community`, or register a new script block using the source and type of an existing block using the `register...()` methods of `Community` and `Species` (see sections 24.3.2 and 24.14.2).

## 24.13 Class SLiMgui

*Superclass: Dictionary*

This class represents the SLiMgui application. When running under SLiMgui, a global object singleton constant of class SLiMgui will be defined, named `slimgui`. This object can be used to query and control the SLiMgui application. When running at the command line, the `slimgui` object will not exist; to determine whether the simulation is running under SLiMgui, one may therefore test `exists("slimgui")`. If a model needs to run both at the command line and under SLiMgui, all uses of the `slimgui` object should be protected by `if (exists("slimgui"))` to avoid errors.

### 24.13.1 SLiMgui properties

`pid => (integer$)`

The Unix process identifier (commonly called the “pid”) of the running SLiMgui application. This can be useful for scripts that wish to use system calls to influence the SLiMgui application.

### 24.13.2 SLiMgui methods

– `(void)openDocument(string$ filePath)`

Open the document at `filePath` in SLiMgui, if possible. Supported document types include SLiM model files (typically with a `.slim` path extension), text files (typically with a `.txt` path extension, and opened as untitled model files), and PNG, JPG/JPEG, BMP, and GIF image file formats (typically `.png` / `.jpg` / `.jpeg` / `.bmp` / `.gif`, respectively). (Note that in SLiMguiLegacy, PDF files (`.pdf`) are supported but these other image file formats are not.) This method can be particularly useful for opening images created by the simulation itself, often by sublaunching a plotting process in R or another environment; see section 14.8 for an example.

– `(void)pauseExecution(string$ filePath)`

Pauses a model that is playing in SLiMgui. This is essentially equivalent to clicking the “Play” button to stop the execution of the model. Execution can be resumed by the user, by clicking the “Play” button again; unlike calling `stop()` or `simulationFinished()`, the simulation is not terminated. This method can be useful for debugging or exploratory purposes, to pause the model at a point of interest. Execution is paused at the end of the currently executing tick, not mid-tick.

If the model is being profiled, or is executing forward to a tick number entered in the tick field, `pauseExecution()` will do nothing; by design, `pauseExecution()` only pauses execution when SLiMgui is doing a simple “Play” of the model.



## 24.14 Class Species

*Superclass: Dictionary*

This class represents a species in a SLiM simulation. In a single-species model, the single Species instance is defined as a global constant named `sim`; in multispecies models, each Species instance is defined as a global constant with the same name as the species.

### 24.14.1 Species properties

`avatar => (string$)`

The avatar string used to represent this species in SLiMgui. Outside of SLiMgui, this property still exists, but is not used by SLiM. Avatars are typically one-character strings, often using an emoji that symbolizes the species. This property is read-only; its value should be set with the `avatar` parameter of `initializeSpecies()`.

`chromosome => (object<Chromosome> $)`

The Chromosome object used by the species.

`chromosomeType => (string$)`

The type of chromosome being simulated by this species; this will be one of "A", "X", or "Y".

`color => (string$)`

The color used to display information about this species in SLiMgui. Outside of SLiMgui, this property still exists, but is not used by SLiM. Colors may be specified by name, or with hexadecimal RGB values of the form "#RRGGBB" (see the Eidos manual). This property is read-only; its value should be set with the `color` parameter of `initializeSpecies()`.

`cycle <=> (integer$)`

The current cycle count for this species. This counter begins at 1, and increments at the end of every tick in which the species is active. In models with non-overlapping generations, particularly WF models, this can be thought of as a generation counter.

`description <=> (string$)`

A human-readable string description for the species. By default, this is the empty string, "", however, it may be set to whatever you wish.

`dimensionality => (string$)`

The spatial dimensionality of the simulation for this species, as specified in `initializeSLiMOptions()`. This will be "" (the empty string) for non-spatial simulations (the default), or "x", "xy", or "xyz", for simulations using those spatial dimensions respectively.

`genomicElementTypes => (object<GenomicElementType>)`

The GenomicElementType objects being used in the species.

`id => (integer$)`

The identifier for this species. Species identifiers are determined by their declaration order in the script; the first declared species is given an `id` of 0, the second is given an `id` of 1, and so forth.

`mutationTypes => (object<MutationType>)`

The MutationType objects being used in the species.

`mutations => (object<Mutation>)`

The Mutation objects that are currently active in the species.

`name => (string$)`

A human-readable `string` name for the subpopulation. This is always the declared name of the species, as given in the explicit species declaration in script, and cannot be changed. The `name` of a species may appear as a label in SLiMgui, and it can be useful in generating output, debugging, and other purposes. See also the `description` property, which can be changed by the user and used for any purpose.

`nucleotideBased => (logical$)`

If `T`, the model for this species is nucleotide-based; if `F`, it is not. See the discussion of the `nucleotideBased` parameter to `initializeSLiMOptions()` for discussion.

`periodicity => (string$)`

The spatial periodicity of the simulation for this species, as specified in `initializeSLiMOptions()`. This will be `""` (the empty string) for non-spatial simulations and simulations with no periodic spatial dimensions (the default). Otherwise, it will be a string representing the subset of spatial dimensions that have been declared to be periodic, as specified to `initializeSLiMOptions()`.

`scriptBlocks => (object<SLiMEidosBlock>)`

All registered `SLiMEidosBlock` objects in the simulation that have been declared with this species as their `species` specifier (*not* `ticks` specifier). These will always be callback blocks; callbacks are species-specific, while other types of blocks are not.

`sexEnabled => (logical$)`

If `T`, sex is enabled for this species; if `F`, individuals are hermaphroditic.

`subpopulations => (object<Subpopulation>)`

The `Subpopulation` instances currently defined in the species.

`substitutions => (object<Substitution>)`

A vector of `Substitution` objects, representing all mutations that have been fixed in this species.

`tag <=> (integer$)`

A user-defined `integer` value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the `Dictionary` class; see the Eidos manual), for another way of attaching state to the simulation.

#### 24.14.2 Species methods

– `(object<Subpopulation>$)addSubpop(is$ subpopID, integer$ size, [float$ sexRatio = 0.5], [logical$ haploid = F])`

Add a new subpopulation with id `subpopID` and `size` individuals. The `subpopID` parameter may be either an `integer` giving the ID of the new subpopulation, or a `string` giving the name of the new subpopulation (such as `"p5"` to specify an ID of 5). Only if sex is enabled for the species, the initial sex ratio may optionally be specified as `sexRatio` (as the male fraction, M:M+F); if it is not specified, a default of `0.5` is used. The new subpopulation will be defined as a global variable immediately by this method (see section 24.15), and will also be returned by this method. Subpopulations added by this method will initially consist of individuals with empty genomes. In order to model subpopulations that split from an already existing subpopulation, use `addSubpopSplit()`.

Only in nonWF models, the `haploid` parameter may be `T`; in this case, the second genome of each new individual will be a null genome, rather than an empty genome. For even greater control in nonWF models, you can call `addSubpop()` with an initial size of `0` and then stock the population with new individuals created however you wish in the next tick's `reproduction()` callback.

- (object<Subpopulation>\$)addSubpopSplit(is\$ subpopID, integer\$ size, io<Subpopulation>\$ sourceSubpop, [float\$ sexRatio = 0.5])

Split off a new subpopulation with id `subpopID` and `size` individuals derived from subpopulation `sourceSubpop`. The `subpopID` parameter may be either an `integer` giving the ID of the new subpopulation, or a `string` giving the name of the new subpopulation (such as "p5" to specify an ID of 5). The `sourceSubpop` parameter may specify the source subpopulation either as a `Subpopulation` object or by `integer` identifier. Only if sex is enabled for the species, the initial sex ratio may optionally be specified as `sexRatio` (as the male fraction, M:M+F); if it is not specified, a default of 0.5 is used. The new subpopulation will be defined as a global variable immediately by this method (see section 24.15), and will also be returned by this method.

Subpopulations added by this method will consist of individuals that are clonal copies of individuals from the source subpopulation, randomly chosen with probabilities proportional to fitness. The fitness of all of these initial individuals is considered to be 1.0, to avoid a doubled round of selection in the initial tick, given that fitness values were already used to choose the individuals to clone. Once this initial set of individuals has mated to produce offspring, the model is effectively of parental individuals in the source subpopulation mating randomly according to fitness, as usual in SLiM, with juveniles migrating to the newly added subpopulation. Effectively, then, then new subpopulation is created empty, and is filled by migrating juveniles from the source subpopulation, in accordance with SLiM's usual model of juvenile migration.

- (integer\$)countOfMutationsOfType(io<MutationType>\$ mutType)

Returns the number of mutations that are of the type specified by `mutType`, out of all of the mutations that are currently active in the species. If you need a vector of the matching `Mutation` objects, rather than just a count, use `-mutationsOfType()`. This method is often used to determine whether an introduced mutation is still active (as opposed to being either lost or fixed). This method is provided for speed; it is much faster than the corresponding Eidos code.

- (object<Individual>)individualsWithPedigreeIDs(integer pedigreeIDs, [Nio<Subpopulation> subpops = NULL])

Looks up individuals by pedigree ID, optionally within specific subpopulations. Pedigree tracking must be turned on with `initializeSLiMOptions(keepPedigrees=T)` to use this method, otherwise an error will result. This method is vectorized; more than one pedigree id may be passed in `pedigreeID`, in which case the returned vector will contain all of the individuals for which a match was found (in the same order in which they were supplied). If a given id is not found, the returned vector will contain no entry for that id (so the length of the returned vector may not match the length of `pedigreeIDs`). If none of the given ids were found, the returned vector will be `object<Individual>(0)`, an empty `object` vector of class `Individual`. If you have more than one pedigree ID to look up, calling this method just once, in vectorized fashion, may be much faster than calling it once for each ID, due to internal optimizations.

To find individuals within all subpopulations, pass the default of `NULL` for `subpops`. If you are interested only in matches within a specific subpopulation, pass that subpopulation for `subpops`; that will make the search faster. Similarly, if you know that a particular subpopulation is the most likely to contain matches, you should supply that subpopulation first in the `subpops` vector so that it will be searched first; the supplied subpopulations are searched in order. Subpopulations may be supplied either as `integer` IDs, or as `Subpopulation` objects.

- (void)killIndividuals(object<Individual> individuals)

Immediately kills the individuals in `individuals`. This removes them from their subpopulation and gives them an index value of -1. The `Individual` objects are not freed immediately, since references to them could still exist in local Eidos variables; instead, the individuals are kept in a temporary "graveyard" until they can be freed safely. It therefore continues to be safe to use them and their genomes, except that accessing their `subpopulation` property will raise an error since they no longer have a subpopulation.

Note that the indices and order of individuals and genomes in all source subpopulations will change unpredictably as a side effect of this method. All evaluated interactions are invalidated as a side effect of calling this method.

Note that this method is only for use in nonWF models, in which mortality is managed manually by the model script. In WF models, mortality is managed automatically by the SLiM core when the new offspring generation becomes the parental generation and the previous parental generation dies; mortality does not otherwise occur in WF models. In nonWF models, mortality normally occurs during the survival stage of the tick cycle (see section 23.4), based upon the fitness values calculated by SLiM, and `survival()` callbacks can influence the outcome of that survival stage. Calls to `killIndividuals()`, on the other hand, can be made at any time during `first()`, `early()`, or `late()` events, and the result cannot be modified by `survival()` callbacks; the given individuals are simply immediately killed. This method therefore provides an alternative, and relatively rarely used, mortality mechanism that is disconnected from fitness.

- (integer) `mutationCounts(Nio<Subpopulation> subpops, [No<Mutation> mutations = NULL])`

Return an integer vector with the frequency counts of all of the `Mutation` objects passed in `mutations`, within the `Subpopulation` objects in `subpops`. The `subpops` argument is required, but you may pass `NULL` to get population-wide frequency counts. Subpopulations may be supplied either as integer IDs, or as `Subpopulation` objects. If the optional `mutations` argument is `NULL` (the default), frequency counts will be returned for all of the active `Mutation` objects in the species – the same `Mutation` objects, and in the same order, as would be returned by the `mutations` property of `sim`, in other words.

See the `-mutationFrequencies()` method to obtain float frequencies instead of integer counts. See also the `Genome` methods `mutationCountsInGenomes()` and `mutationFrequenciesInGenomes()`.

- (float) `mutationFrequencies(Nio<Subpopulation> subpops, [No<Mutation> mutations = NULL])`

Return a float vector with the frequencies of all of the `Mutation` objects passed in `mutations`, within the `Subpopulation` objects in `subpops`. The `subpops` argument is required, but you may pass `NULL` to get population-wide frequencies. Subpopulations may be supplied either as integer IDs, or as `Subpopulation` objects. If the optional `mutations` argument is `NULL` (the default), frequencies will be returned for all of the active `Mutation` objects in the species – the same `Mutation` objects, and in the same order, as would be returned by the `mutations` property of `sim`, in other words.

See the `-mutationCounts()` method to obtain integer counts instead of float frequencies. See also the `Genome` methods `mutationCountsInGenomes()` and `mutationFrequenciesInGenomes()`.

- (object<Mutation>) `mutationsOfType(io<MutationType>$ mutType)`

Returns an object vector of all the mutations that are of the type specified by `mutType`, out of all of the mutations that are currently active in the species. If you just need a count of the matching `Mutation` objects, rather than a vector of the matches, use `-countOfMutationsOfType()`. This method is often used to look up an introduced mutation at a later point in the simulation, since there is no way to keep persistent references to objects in SLiM. This method is provided for speed; it is much faster than the corresponding Eidos code.

- (void) `outputFixedMutations([Ns$ filePath = NULL], [logical$ append = F])`

Output all fixed mutations – all `Substitution` objects, in other words (see section 1.5.2) – in a SLiM native format (see section 26.1.2 for output format details). If the optional parameter `filePath` is `NULL` (the default), output will be sent to Eidos’s output stream (see section 4.2.1). Otherwise, output will be sent to the filesystem path specified by `filePath`, overwriting that file if `append` is `F`, or appending to the end of it if `append` is `T`. Mutations which have fixed but have not been turned into `Substitution` objects – typically because `convertToSubstitution` has been set to `F` for their

mutation type (see section 24.11.1) – are not output; they are still considered to be segregating mutations by SLiM.

In SLiM 3.3 and later, the output format includes the nucleotides associated with any nucleotide-based mutations; see section 26.1.2.

Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

- `(void)outputFull([Ns$ filePath = NULL], [logical$ binary = F], [logical$ append = F], [logical$ spatialPositions = T], [logical$ ages = T], [logical$ ancestralNucleotides = T], [logical$ pedigreeIDs = F])`

Output the state of the entire population (see section 26.1.1 for output format details). If the optional parameter `filePath` is `NULL` (the default), output will be sent to Eidos's output stream (see section 4.2.1). Otherwise, output will be sent to the filesystem path specified by `filePath`, overwriting that file if `append` is `F`, or appending to the end of it if `append` is `T`. When writing to a file, a `logical` flag, `binary`, may be supplied as well. If `binary` is `T`, the population state will be written as a binary file instead of a text file (binary data cannot be written to the standard output stream). The binary file is usually smaller, and in any case will be read much faster than the corresponding text file would be read. Binary files are not guaranteed to be portable between platforms; in other words, a binary file written on one machine may not be readable on a different machine (but in practice it usually will be, unless the platforms being used are fairly unusual). If `binary` is `F` (the default), a text file will be written.

Beginning with SLiM 2.3, the `spatialPositions` parameter may be used to control the output of the spatial positions of individuals in species for which continuous space has been enabled using the `dimensionality` option of `initializeSLiMOptions()`. If `spatialPositions` is `F`, the output will not contain spatial positions, and will be identical to the output generated by SLiM 2.1 and later. If `spatialPositions` is `T`, spatial position information will be output if it is available (see section 26.1.1 for format details). If the species does not have continuous space enabled, the `spatialPositions` parameter will be ignored. Positional information may be output for all output destinations – the Eidos output stream, a text file, or a binary file.

Beginning with SLiM 3.0, the `ages` parameter may be used to control the output of the ages of individuals in nonWF simulations. If `ages` is `F`, the output will not contain ages, preserving backward compatibility with the output format of SLiM 2.1 and later. If `ages` is `T`, ages will be output for nonWF models (see section 26.1.1 for format details). In WF simulations, the `ages` parameter will be ignored.

Beginning with SLiM 3.3, the `ancestralNucleotides` parameter may be used to control the output of the ancestral nucleotide sequence in nucleotide-based models (see section 26.1.1 for format details). If `ancestralNucleotides` is `F`, the output will not contain ancestral nucleotide information, and so the ancestral sequence will not be restored correctly if the saved file is loaded with `readPopulationFile()`. This option is provided because the ancestral sequence may be quite large, for models with a long chromosome (e.g., 1 GB if the chromosome is  $10^9$  bases long, when saved in text format, or 0.25 GB when saved in binary format). If the model is not nucleotide-based (as enabled with the `nucleotideBased` parameter to `initializeSLiMOptions()`), the `ancestralNucleotides` parameter will be ignored. Note that in nucleotide-based models the output format will *always* include the nucleotides associated with any nucleotide-based mutations; the `ancestralNucleotides` flag governs only the ancestral sequence.

Beginning with SLiM 3.5, the `pedigreeIDs` parameter may be used to request that pedigree IDs be written out (and read in by `readFromPopulationFile()`, subsequently). This option is turned off (`F`) by default, to preserve backward compatibility; if it is turned on (`T`), different file version values will be used, and backward compatibility with previous versions of SLiM will be lost (see section 26.1.1). This option may only be used if SLiM's optional pedigree tracking has been enabled with `initializeSLiMOptions(keepPedigrees=T)`.

Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.



- (void)outputMutations(object<Mutation> mutations, [Ns\$ filePath = NULL], [logical\$ append = F])

Output all of the given mutations (see section 26.1.3 for output format details). This can be used to output all mutations of a given mutation type, for example. If the optional parameter `filePath` is `NULL` (the default), output will be sent to Eidos's output stream (see section 4.2.1). Otherwise, output will be sent to the filesystem path specified by `filePath`, overwriting that file if `append` is `F`, or appending to the end of it if `append` is `T`.

In SLiM 3.3 and later, the output format includes the nucleotides associated with any nucleotide-based mutations; see section 26.1.3.

Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

- (integer\$)readFromPopulationFile(string\$ filePath, [No\$ subpopMap = NULL])

Read from a population initialization file, whether in text or binary format as previously specified to `outputFull()`, and return the tick counter value represented by the file's contents (i.e., the tick at which the file was generated). Although this is most commonly used to set up initial populations (often in an Eidos event set to run in tick 1, immediately after simulation initialization), it may be called in any `early()` or `late()` Eidos event; the current state of all populations in the target species will be wiped and replaced by the state in the file at `filePath`. All Eidos variables that are of type `object` and have element type `Subpopulation`, `Genome`, `Mutation`, `Individual`, or `Substitution` will be removed as a side effect of this method if they contain any element that belongs to the target species, because those objects will no longer exist in the SLiM simulation; if you want to preserve any of that state, you should output it or save it to a file prior to this call. New symbols will be defined to refer to the new `Subpopulation` objects loaded from the file.

If the file being read was written by a version of SLiM prior to 2.3, then for backward compatibility fitness values will be calculated immediately for any new subpopulations created by this call, which will trigger the calling of any activated and applicable `mutationEffect()` and `fitnessEffect()` callbacks. When reading files written by SLiM 2.3 or later, fitness values are not calculated as a side effect of this call (because the simulation will often need to evaluate interactions or modify other state prior to doing so).

In SLiM 2.3 and later when using the WF model, calling `readFromPopulationFile()` from any context other than a `late()` event causes a warning; calling from a `late()` event is almost always correct in WF models, so that fitness values can be automatically recalculated by SLiM at the usual time in the tick cycle without the need to force their recalculation (see chapter 22, and comments on `recalculateFitness()` below).

In SLiM 3.0 when using the nonWF model, calling `readFromPopulationFile()` from any context other than an `early()` event causes a warning; calling from an `early()` event is almost always correct in nonWF models, so that fitness values can be automatically recalculated by SLiM at the usual time in the tick cycle without the need to force their recalculation (see chapter 23, and comments on `recalculateFitness()` below).

As of SLiM 2.1, this method changes the tick and cycle counters to the tick and cycle read from the file. If you do not want these counters to be changed, you can change them back after reading, by setting `community.tick` and `sim.cycle` to whatever values you wish. Note that restoring a saved past state and running forward again will not yield the same simulation results, because the random number generator's state will not be the same; to ensure reproducibility from a given time point, `setSeed()` can be used to establish a new seed value. Any changes made to structure of the species (mutation types, genomic element types, etc.) will not be wiped and re-established by `readFromPopulationFile()`; this method loads only the population's state, not the species configuration, so care should be taken to ensure that the species structure meshes coherently with the loaded data. Indeed, state such as the selfing and cloning rates of subpopulations, values set into `tag` properties, and values set onto objects with `setValue()` will also be lost, since it is not saved out by `outputFull()`. Only information saved by `outputFull()` will be restored; all other state associated



with the species – subpopulations, individuals, genomes, mutations, and substitutions – will be lost, and should be re-established by the model if it is still needed.

As of SLiM 2.3, this method will read and restore the spatial positions of individuals if that information is present in the output file and the species has enabled continuous space (see `outputFull()` for details). If spatial positions are present in the output file but the species has not enabled continuous space (or the number of spatial dimensions does not match), an error will result. If the species has enabled continuous space but spatial positions are not present in the output file, the spatial positions of the individuals read will be undefined, but an error is not raised.

As of SLiM 3.0, this method will read and restore the ages of individuals if that information is present in the output file and the simulation is based upon the nonWF model. If ages are present but the simulation uses a WF model, an error will result; the WF model does not use age information. If ages are not present but the simulation uses a nonWF model, an error will also result; the nonWF model requires age information.

As of SLiM 3.3, this method will restore the nucleotides of nucleotide-based mutations, and will restore the ancestral nucleotide sequence, if that information is present in the output file. Loading an output file that contains nucleotide information in a non-nucleotide-based model, and *vice versa*, will produce an error.

As of SLiM 3.5, this method will read and restore the pedigree IDs of individuals and genomes if that information is present in the output file (as requested with `outputFull(pedigreeIDs=T)`) and if SLiM's optional pedigree tracking has been enabled with `initializeSLiMOptions(keepPedigrees=T)`.

This method can also be used to read tree-sequence (`.trees`) files saved by `treeSeqOutput()` or generated by the Python `pyslim` package. Beginning with SLiM 4, the `subpopMap` parameter may be supplied to re-order the populations of the input tree sequence when it is loaded in to SLiM. This parameter must have a value that is a `Dictionary`; the keys of this dictionary should be SLiM population identifiers as `string` values (e.g., "p2"), and the values should be indexes of populations in the input tree sequence; a key/value pair of "p2", 4 would mean that the fifth population in the input (the one at zero-based index 4) should become p2 on loading into SLiM. If `subpopMap` is non-NULL, *all* populations in the tree sequence must be explicitly mapped, even if their index will not change and even if they will not be used by SLiM; the only exception is for unused slots in the population table, which can be explicitly remapped but do not have to be. For instance, suppose we have a tree sequence in which population 0 is unused, population 1 is not a SLiM population (for example, an ancestral population produced by `msprime`), and population 2 is a SLiM population, and we want to load this in with population 2 as p0 in SLiM. To do this, we could supply a value of `Dictionary("p0", 2, "p1", 1, "p2", 0)` for `subpopMap`, or we could leave out slot 0 since it is unused, with `Dictionary("p0", 2, "p1", 1)`. Although this facility cannot be used to remove populations in the tree sequence, note that it may *add* populations that will be visible when `treeSeqOutput()` is called (although these will not be SLiM populations); if, in this example, we had used `Dictionary("p0", 0, "p1", 1, "p5", 2)` and then we wrote the result out with `treeSeqOutput()`, the resulting tree sequence would have six populations, although three of them would be empty and would not be used by SLiM. The use of `subpopMap` makes it easier to load simulation data that was generated in Python, since that typically uses an id of 0. The `subpopMap` parameter may not be used with file formats other than tree-sequence files, at the present time; setting up the correct subpopulation ids is typically easier when working with those other formats. Note the `tskit` command-line interface can be used, like `python3 -m tskit populations file.trees`, to find out the number of subpopulations in a tree-sequence file and their IDs.

When loading a tree sequence, a crosscheck of the loaded data will be performed to ensure that the tree sequence was well-formed and was loaded correctly. When running a Release build of SLiM, however, this crosscheck will only occur the first time that `readFromPopulationFile()` is called to load a tree sequence; subsequent calls will not perform this crosscheck, for greater speed when running models that load saved population state many times (such as models that are conditional on

fixation). If you suspect that a tree sequence file might be corrupted or read incorrectly, running a Debug build of SLiM enables crosschecks after every load.

– (void)recalculateFitness([Ni\$ tick = NULL])

Force an immediate recalculation of fitness values for all individuals in all subpopulations. Normally fitness values are calculated at a fixed point in each tick, and those values are cached and used until the next recalculation. If simulation parameters are changed in script in a way that affects fitness calculations, and if you wish those changes to take effect immediately rather than taking effect at the next automatic recalculation, you may call `recalculateFitness()` to force an immediate recalculation and recache.

The optional parameter `tick` provides the tick for which `mutationEffect()` and `fitnessEffect()` callbacks should be selected; if it is `NULL` (the default), the current tick value for the simulation, `community.tick`, is used. If you call `recalculateFitness()` in an `early()` event in a WF model, you may wish this to be `community.tick - 1` in order to utilize the `mutationEffect()` and `fitnessEffect()` callbacks for the previous tick, as if the changes that you have made to fitness-influencing parameters were already in effect at the end of the previous tick when the new generation was first created and evaluated (usually it is simpler to just make such changes in a `late()` event instead, however, in which case calling `recalculateFitness()` is probably not necessary at all since fitness values will be recalculated immediately afterwards). Regardless of the value supplied for `tick` here, `community.tick` inside callbacks will report the true tick number, so if your callbacks consult that parameter in order to create tick-specific fitness effects you will need to handle the discrepancy somehow. (Similar considerations apply for nonWF models that call `recalculateFitness()` in a `late()` event, which is also not advisable in general.)

After this call, the fitness values used for all purposes in SLiM will be the newly calculated values. Calling this method will trigger the calling of any enabled and applicable `mutationEffect()` and `fitnessEffect()` callbacks, so this is quite a heavyweight operation; you should think carefully about what side effects might result (which is why fitness recalculation does not just occur automatically after changes that might affect fitness values).

– (object<SLiMEidosBlock>\$)registerFitnessEffectCallback(Nis\$ id, string\$ source, [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the `string` singleton `source`, as an Eidos `fitnessEffect()` callback in the current simulation (specific to the target species), with an optional subpopulation `subpop` (which may be an `integer` identifier, or `NULL`, the default, to indicate all subpopulations), and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an `integer`, or as a `string` symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

– (object<SLiMEidosBlock>\$)registerMateChoiceCallback(Nis\$ id, string\$ source, [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the `string` singleton `source`, as an Eidos `mateChoice()` callback in the current simulation (specific to the target species), with optional subpopulation `subpop` (which may be an `integer` identifier, or `NULL`, the default, to indicate all subpopulations) and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an `integer`, or as a `string` symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerModifyChildCallback(Nis\$ id, string\$ source, [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the `string` singleton `source`, as an Eidos `modifyChild()` callback in the current simulation (specific to the target species), with optional subpopulation `subpop` (which may be an `integer` identifier, or `NULL`, the default, to indicate all subpopulations) and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an `integer`, or as a `string` symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerMutationCallback(Nis\$ id, string\$ source, [Nio<MutationType>\$ mutType = NULL], [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the `string` singleton `source`, as an Eidos `mutation()` callback in the current simulation (specific to the target species), with an optional mutation type `mutType` (which may be an `integer` mutation type identifier, or `NULL`, the default, to indicate all mutation types – see section 25.9), optional subpopulation `subpop` (which may also be an `integer` identifier, or `NULL`, the default, to indicate all subpopulations), and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an `integer`, or as a `string` symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerMutationEffectCallback(Nis\$ id, string\$ source, io<MutationType>\$ mutType, [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the `string` singleton `source`, as an Eidos `mutationEffect()` callback in the current simulation (specific to the target species), with a required mutation type `mutType` (which may be an `integer` mutation type identifier), optional subpopulation `subpop` (which may also be an `integer` identifier, or `NULL`, the default, to indicate all subpopulations), and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an `integer`, or as a `string` symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerRecombinationCallback(Nis\$ id, string\$ source, [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the `string` singleton `source`, as an Eidos `recombination()` callback in the current simulation (specific to the target species), with optional subpopulation `subpop` (which may be an `integer` identifier, or `NULL`, the default, to indicate all subpopulations) and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an `integer`, or as a `string` symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerReproductionCallback(Nis\$ id, string\$ source, [Nio<Subpopulation>\$ subpop = NULL], [Ns\$ sex = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the string singleton `source`, as an Eidos `reproduction()` callback in the current simulation (specific to the target species), with optional subpopulation `subpop` (which may be an integer identifier, or `NULL`, the default, to indicate all subpopulations), optional sex-specificity `sex` (which may be "M" or "F" in sexual species to make the callback specific to males or females respectively, or `NULL` for no sex-specificity), and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an integer, or as a string symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (object<SLiMEidosBlock>\$)registerSurvivalCallback(Nis\$ id, string\$ source, [Nio<Subpopulation>\$ subpop = NULL], [Ni\$ start = NULL], [Ni\$ end = NULL])

Register a block of Eidos source code, represented as the string singleton `source`, as an Eidos `survival()` callback in the current simulation (specific to the target species), with optional subpopulation `subpop` (which may be an integer identifier, or `NULL`, the default, to indicate all subpopulations) and optional `start` and `end` ticks all limiting its applicability. The script block will be given identifier `id` (specified as an integer, or as a string symbolic name such as "s5"); this may be `NULL` if there is no need to be able to refer to the block later. The registered callback is added to the end of the list of registered `SLiMEidosBlock` objects, and is active immediately; it *may* be eligible to execute in the current tick (see section 25.11 for details). The new `SLiMEidosBlock` will be defined as a global variable immediately by this method (see section 24.12), and will also be returned by this method.

- (void)simulationFinished(void)

Declare the current simulation finished. This method is equivalent to the `Community` method `simulationFinished()`, except that this method is only legal to call in single-species models (to provide backward compatibility). It is recommended that new code should call the `Community` method; this method may be deprecated in the future.

- (void)skipTick(void)

Deactivate the target species for the current tick. This sets the `active` property of the species to `F`; it also set the `active` property of all callbacks that belong to the species (with the species as their `species` specifier) to `F`, and sets the `active` property of all events that are synchronized with the species (with the species as their `ticks` specifier) to `F`. The cycle counter for the species will not be incremented at the end of the tick. This method may only be called in `first()` events, to ensure that species are either active or inactive throughout a given tick.

- (object<Mutation>)subsetMutations([No<Mutation>\$ exclude = NULL], [Nio<MutationType>\$ mutType = NULL], [Ni\$ position = NULL], [Nis\$ nucleotide = NULL], [Ni\$ tag = NULL], [Ni\$ id = NULL])

Returns a vector of mutations subset from the list of all active mutations in the species (as would be provided by the `mutations` property). The parameters specify constraints upon the subset of mutations that will be returned. Parameter `exclude`, if non-`NULL`, may specify a specific mutation that should not be included (typically the focal mutation in some operation). Parameter `mutType`, if non-`NULL`, may specify a mutation type for the mutations to be returned (as either a `MutationType` object or an integer identifier). Parameter `position`, if non-`NULL`, may specify a base position for the mutations to be returned. Parameter `nucleotide`, if non-`NULL`, may specify a nucleotide for the mutations to be returned (either as a string, "A" / "C" / "G" / "T", or as an integer, 0 / 1 / 2 / 3 respectively). Parameter `tag`, if non-`NULL`, may specify a tag value for the mutations to be returned.



Parameter `id`, if non-NULL, may specify a required value for the `id` property of the mutations to be returned.

This method is shorthand for getting the `mutations` property of the subpopulation, and then using operator `[]` to select only mutations with the desired properties; besides being much simpler than the equivalent Eidos code, it is also much faster. Note that if you only need to select on mutation type, the `mutationsOfType()` method will be even faster.

– `(logical$)treeSeqCoalesced(void)`

Returns the coalescence state for the recorded tree sequence at the last simplification. The returned value is a logical singleton flag, `T` to indicate that full coalescence was observed at the last tree-sequence simplification (meaning that there is a single ancestral individual that roots all ancestry trees at all sites along the chromosome – although not necessarily the *same* ancestor at all sites), or `F` if full coalescence was not observed. For simple models, reaching coalescence may indicate that the model has reached an equilibrium state, but this may not be true in models that modify the dynamics of the model during execution by changing migration rates, introducing new mutations programmatically, dictating non-random mating, etc., so be careful not to attach more meaning to coalescence than it is due; some models may require burn-in beyond coalescence to reach equilibrium, or may not have an equilibrium state at all. Also note that some actions by a model, such as adding a new subpopulation, may cause the coalescence state to revert from `T` back to `F` (at the next simplification), so a return value of `T` may not necessarily mean that the model is coalesced at the present moment – only that it was coalesced at the last simplification.

This method may only be called if tree sequence recording has been turned on with `initializeTreeSeq()`; in addition, `checkCoalescence=T` must have been supplied to `initializeTreeSeq()`, so that the necessary work is done during each tree-sequence simplification. Since this method does not perform coalescence checking itself, but instead simply returns the coalescence state observed at the last simplification, it may be desirable to call `treeSeqSimplify()` immediately before `treeSeqCoalesced()` to obtain up-to-date information. However, the speed penalty of doing this in every tick would be large, and most models do not need this level of precision; usually it is sufficient to know that the model has coalesced, without knowing whether that happened in the current tick or in a recent preceding tick.

– `(void)treeSeqOutput(string$ path, [logical$ simplify = T], [logical$ includeModel = T], [No$ metadata = NULL])`

Outputs the current tree sequence recording tables to the path specified by `path`. This method may only be called if tree sequence recording has been turned on with `initializeTreeSeq()`. If `simplify` is `T` (the default), simplification will be done immediately prior to output; this is almost always desirable, unless a model wishes to avoid simplification entirely. (Note that if simplification is not done, then all genomes since the last simplification will be marked as samples in the resulting tree sequence.) A binary tree sequence file will be written to the specified path; a filename extension of `.trees` is suggested for this type of file.

Normally, the full SLiM script used to generate the tree sequence is written out to the provenance entry of the tree sequence file, to the `model` subkey of the `parameters` top-level key. Supplying `F` for `includeModel` suppresses output of the full script; see section 26.4.6 for further discussion.

A `Dictionary` object containing user-generated metadata may be supplied with the `metadata` parameter. If present, this dictionary will be serialized as JSON and attached to the saved tree sequence under a key named `user_metadata`, within the `SLiM` key (see section 26.4.5). If `tskit` is used to read the tree sequence in Python, this metadata will automatically be deserialized and made available at `ts.metadata["SLiM"]["user_metadata"]`. This metadata dictionary is not used by SLiM, or by `pyslim`, `tskit`, or `msprime`; you may use it for any purpose you wish. Note that `metadata` may actually be any subclass of `Dictionary`, such as a `DataFrame`. It can even be a `Species` object such as `sim`, or a `LogFile` instance; however, only the keys and values contained by the object's `Dictionary` superclass state will be serialized into the metadata (properties of the subclass will be ignored).

- (void)treeSeqRememberIndividuals(object<Individual> individuals, [logical\$ permanent = T])

Mark the individuals specified by `individuals` to be kept across tree sequence table simplification. This method may only be called if tree sequence recording has been turned on with `initializeTreeSeq()`. All currently living individuals are always kept across simplification; this method does not need to be called, and indeed should not be called, for that purpose. Instead, `treeSeqRememberIndividuals()` allows any individual, including dead individuals, to be kept in the final tree sequence. Typically this would be used, for example, to keep particular individuals that you wanted to be able to trace ancestry back to in later analysis. However, this is not the typical usage pattern for tree sequence recording; most models will not need to call this method.

There are two ways to keep individuals across simplification. If `permanent` is `T` (the default), then the specified individuals will be permanently remembered: their genomes will be added to the current sample, and they will always be present in the tree sequence. Permanently remembering a large number of individuals will, of course, markedly increase memory usage and runtime.

Supplying `F` for `permanent` will instead mark the individuals only for (temporary) retention: their genomes will not be added to the sample, and they will appear in the final tree sequence only if one of their genomes is retained across simplification. In other words, the rule of thumb for retained individuals is simple: if a genome is kept by simplification, the genome's corresponding individual is kept also, *if* it is retained. Note that permanent remembering takes priority; calling this function with `permanent=F` on an individual that has previously been permanently remembered will not remove it from the sample.

The behavior of simplification for individuals retained with `permanent=F` depends upon the value of the `retainCoalescentOnly` flag passed to `initializeTreeSeq()`; here we will discuss the behavior of that flag in detail. First of all, genomes are *always* removed by simplification unless they are (a) part of the final generation (i.e., in a living individual when simplification occurs), (b) ancestral to the final generation, (c) a genome of a permanently remembered individual, or (d) ancestral to a permanently remembered individual. If `retainCoalescentOnly` is `T` (the default), they are *also* always removed if they are not a branch point (i.e., a coalescent node or most recent common ancestor) in the tree sequence. In some cases it may be useful to retain a genome and its associated individual when it is simply an intermediate node in the ancestry (i.e., in the middle of a branch). This can be enabled by setting `retainCoalescentOnly` to `F` in your call to `initializeTreeSeq()`. In this case, ancestral genomes that are intermediate ("unary nodes", in `tskit` parlance) and are within an individual that has been retained using the `permanent=F` flag here are kept, along with the retained individual itself. Since setting `retainCoalescentOnly` to `F` will prevent the unary nodes for retained individuals from being pruned, simplification may often be unable to prune very much at all from the tree sequence, and memory usage and runtime may increase rapidly. If you are retaining many individuals, this setting should therefore be used only with caution; it is not necessary if you are purely interested in the most recent common ancestors. See the `pyslim` documentation for further discussion of retaining and remembering individuals and the effects of the `retainCoalescentOnly` flag.

The metadata (age, location, etc) that are stored in the resulting tree sequence are those values present at either (a) the final generation, if the individual is alive when the tree sequence is output, or (b) the last time that the individual was remembered, if not. Calling `treeSeqRememberIndividuals()` on an individual that is already remembered will cause the archived information about the remembered individual to be updated to reflect the individual's current state; care should be taken to remember individuals at a point in time when their state is valid. A case where this is particularly important is for the spatial location of individuals in continuous-space models. `SLiM` automatically retains the portions of the genomes that comprise the first generation of any new subpopulation created with `addSubpop()` that are inherited by extant individuals, for easy recapitation and other analysis (see sections 17.2 and 17.10). However, the individuals of the first generation are not remembered automatically, only their needed genomic information.



– (void)treeSeqSimplify(void)

Triggers an immediate simplification of the tree sequence recording tables. This method may only be called if tree sequence recording has been turned on with `initializeTreeSeq()`. A call to this method will free up memory being used by entries that are no longer in the ancestral path of any individual within the current sample (currently living individuals, in other words, plus those explicitly added to the sample with `treeSeqRememberIndividuals()`), but it can also take a significant amount of time. Typically calling this method is not necessary; the automatic simplification performed occasionally by SLiM should be sufficient for most models.

## 24.15 Class Subpopulation

*Superclass: Dictionary*

This class represents one subpopulation in the simulated population. Section 1.5.5 presents an overview of the conceptual role of this class. The subpopulations currently defined in the simulation are defined as global constants with the same names used in the SLiM input file – `p1`, `p2`, and so forth.

### 24.15.1 Subpopulation properties

**cloningRate => (float)**

The fraction of children in the next generation that will be produced by cloning (as opposed to biparental mating). In non-sexual (i.e. hermaphroditic) simulations, this property is a singleton `float` representing the overall subpopulation cloning rate. In sexual simulations, this property is a `float` vector with two values: the cloning rate for females (at index 0) and for males (at index 1).

**description <=> (string\$)**

A human-readable `string` description for the subpopulation. By default, this is the empty string, `""`; however, it may be set to whatever you wish. When tree-sequence recording is enabled, `description` is persisted in the subpopulation's metadata in tree-sequence output.

**firstMaleIndex => (integer\$)**

The index of the first male individual in the subpopulation. The `genomes` vector is sorted into females first and males second; `firstMaleIndex` gives the position of the boundary between those sections. Note, however, that there are two genomes per diploid individual, and the `firstMaleIndex` is *not* premultiplied by 2; you must multiply it by 2 before using it to decide whether a given index into `genomes` is a genome for a male or a female. The `firstMaleIndex` property is also the number of females in the subpopulation, given this design. For non-sexual (i.e. hermaphroditic) simulations, this property has an undefined value and should not be used.

**fitnessScaling <=> (float\$)**

A `float` scaling factor applied to the fitness of all individuals in this subpopulation (i.e., the fitness value computed for each individual will be multiplied by this value). This is primarily of use in nonWF models, where fitness is absolute, rather than in WF models, where fitness is relative (and thus a constant factor multiplied into the fitness of every individual will make no difference); however, it may be used in either type of model. This provides a simple, fast way to modify the fitness of all individuals in a subpopulation; conceptually it is similar to returning the same fitness effect for all individuals in the subpopulation from a `fitnessEffect()` callback, but without the complexity and performance overhead of implementing such a callback. To scale the fitness of individuals by different (individual-specific) factors, see the `fitnessScaling` property of `Individual`.

The value of `fitnessScaling` is reset to `1.0` every tick, so that any scaling factor set lasts for only a single tick. This reset occurs immediately after fitness values are calculated, in both WF and nonWF models.

**genomes => (object<Genome>)**

All of the genomes contained by the subpopulation; there are two genomes per diploid individual.

**genomesNonNull => (object<Genome>)**

All of the genomes contained by the subpopulation, as with the **genomes** property, if all of them are not null genomes; any null genomes present are excluded from the returned vector. This is a convenience shorthand, sometimes useful in models that involve null genomes.

**id => (integer\$)**

The identifier for this subpopulation; for subpopulation **p3**, for example, this is 3.

**immigrantSubpopFractions => (float)**

The expected value of the fraction of children in the next generation that are immigrants arriving from particular subpopulations.

**immigrantSubpopIDs => (integer)**

The identifiers of the particular subpopulations from which immigrants will arrive in the next generation.

**individualCount => (integer\$)**

The number of individuals in the subpopulation; one-half of the number of genomes.

**individuals => (object<Individual>)**

All of the individuals contained by the subpopulation. Each individual is diploid and thus contains two **Genome** objects. See the **sampleIndividuals()** and **subsetIndividuals()** for fast ways to get a subset of the individuals in a subpopulation.

**lifetimeReproductiveOutput => (integer)**

If pedigree tracking is turned on with **initializeSLiMOptions(keepPedigrees=T)**, **lifetimeReproductiveOutput** contains the value of the **Individual** property **reproductiveOutput** for all individuals in the subpopulation that died in the last viability/survival tick cycle stage (or, for WF models, immediately after reproduction). This allows access to the lifetime reproductive output of individuals in the subpopulation at the end of their lives. If pedigree tracking is not on, this property is unavailable.

**lifetimeReproductiveOutputF => (integer)**

If pedigree tracking is turned on with **initializeSLiMOptions(keepPedigrees=T)**, **lifetimeReproductiveOutputF** contains the value of the **Individual** property **reproductiveOutput** for all female individuals in the subpopulation that died in the last viability/survival tick cycle stage (or, for WF models, immediately after reproduction). This property is undefined if separate sexes have not been enabled, or if pedigree tracking is not on.

**lifetimeReproductiveOutputM => (integer)**

If pedigree tracking is turned on with **initializeSLiMOptions(keepPedigrees=T)**, **lifetimeReproductiveOutputM** contains the value of the **Individual** property **reproductiveOutput** for all male individuals in the subpopulation that died in the last viability/survival tick cycle stage (or, for WF models, immediately after reproduction). This property is undefined if separate sexes have not been enabled, or if pedigree tracking is not on.

**name <=> (string\$)**

A human-readable **string** name for the subpopulation. By default, this is the subpopulation's symbol as a **string**; for subpopulation **p3**, for example, **name** defaults to "**p3**". However, it may be set to whatever you wish except that subpopulation names must be unique across time (two different subpopulations may not both have the name "**foo**", even if they never exist at the same time). A subpopulation's name may appear as a label in **SLiMgui**, and it can be useful in generating output,

debugging, and other purposes. When tree-sequence recording is enabled, `name` is persisted in the subpopulation's metadata in tree-sequence output, and can then be used in Python to identify the subpopulation; if you plan to take advantage of that feature, `name` should follow the syntax of Python identifiers: starting with a letter or underscore [`a-zA-Z_`], followed by letters, digits, or underscores [`a-zA-Z0-9_`], without spaces, hyphens, or other characters.

**selfingRate => (float\$)**

The expected value of the fraction of children in the next generation that will be produced by selfing (as opposed to biparental mating). Selfing is only possible in non-sexual (i.e. hermaphroditic) simulations; for sexual simulations this property always has a value of `0.0`.

**sexRatio => (float\$)**

For sexual simulations, the sex ratio for the subpopulation. This is defined, in SLiM, as the fraction of the subpopulation that is male; in other words, it is actually the  $M:(M+F)$  ratio. For non-sexual (i.e. hermaphroditic) simulations, this property has an undefined value and should not be used.

**spatialBounds => (float)**

The spatial boundaries of the subpopulation. The length of the `spatialBounds` property depends upon the spatial dimensionality declared with `initializeSLiMOptions()`. If the spatial dimensionality is zero (as it is by default), the value of this property is `float(0)` (a zero-length `float` vector). Otherwise, minimums are supplied for each coordinate used by the dimensionality of the simulation, followed by maximums for each. In other words, if the declared dimensionality is "xy", the `spatialBounds` property will contain values (`x0`, `y0`, `x1`, `y1`); bounds for the z coordinate will not be included in that case, since that coordinate is not used in the simulation's dimensionality. This property cannot be set, but the `setSpatialBounds()` method may be used to achieve the same thing.

**species => (object<Species>\$)**

The species to which the target object belongs.

**tag <=> (integer\$)**

A user-defined `integer` value. The value of `tag` is initially undefined, and it is an error to try to read it; if you wish it to have a defined value, you must arrange that yourself by explicitly setting its value prior to using it elsewhere in your code. The value of `tag` is not used by SLiM; it is free for you to use. See also the `getValue()` and `setValue()` methods (provided by the `Dictionary` class; see the Eidos manual), for another way of attaching state to subpopulations.

#### 24.15.2 Subpopulation methods

– (No<Individual>\$)addCloned(object<Individual>\$ parent)

Generates a new offspring individual from the given parent by clonal reproduction, queues it for addition to the target subpopulation, and returns it. The new offspring will not be visible as a member of the target subpopulation until the end of the offspring generation tick cycle stage. The subpopulation of `parent` will be used to locate applicable `mutation()` and `modifyChild()` callbacks governing the generation of the offspring individual.

Note that this method is only for use in nonWF models. See `addCrossed()` for further general notes on the addition of new offspring individuals.

– (No<Individual>\$)addCrossed(object<Individual>\$ parent1, object<Individual>\$ parent2, [Nfs\$ sex = NULL])

Generates a new offspring individual from the given parents by biparental sexual reproduction, queues it for addition to the target subpopulation, and returns it. The new offspring will not be visible as a member of the target subpopulation until the end of the offspring generation tick cycle stage. Attempting to use a newly generated offspring individual as a mate, or to reference it as a member of

the target subpopulation in any other way, will result in an error. In most models the returned individual is not used, but it is provided for maximal generality and flexibility.

The new offspring individual is generated from `parent1` and `parent2` by crossing them. In sexual models `parent1` must be female and `parent2` must be male; in hermaphroditic models, `parent1` and `parent2` are unrestricted. If `parent1` and `parent2` are the same individual in a hermaphroditic model, that parent self-fertilizes, or “selfs”, to generate the offspring sexually (note this is not the same as clonal reproduction). Such selfing is considered “incidental” by `addCrossed()`, however; if the `preventIncidentalSelfing` flag of `initializeSLiMOptions()` is T, supplying the same individual for `parent1` and `parent2` is an error (you must check for and prevent incidental selfing if you set that flag in a nonWF model). If non-incidental selfing is desired, `addSelfed()` should be used instead.

The `sex` parameter specifies the sex of the offspring. A value of `NULL` means “make the default choice”; in non-sexual models it is the only legal value for `sex`, and does nothing, whereas in sexual models it causes male or female to be chosen with equal probability. A value of “M” or “F” for `sex` specifies that the offspring should be male or female, respectively. Finally, a `float` value from `0.0` to `1.0` for `sex` provides the probability that the offspring will be male; a value of `0.0` will produce a female, a value of `1.0` will produce a male, and for intermediate values SLiM will draw the sex of the offspring randomly according to the specified probability. Unless you wish the bias the sex ratio of offspring, the default value of `NULL` should generally be used.

Note that any defined, active, and applicable `recombination()`, `mutation()`, and `modifyChild()` callbacks will be called as a side effect of calling this method, before this method even returns. For `recombination()` and `mutation()` callbacks, the subpopulation of the parent that is generating a given gamete is used; for `modifyChild()` callbacks the situation is more complex. In most biparental mating events, `parent1` and `parent2` will belong to the same subpopulation, and `modifyChild()` callbacks for that subpopulation will be used, just as in WF models. In certain models (such as models of pollen flow and broadcast spawning), however, biparental mating may occur between parents that are not from the same subpopulation; that is legal in nonWF models, and in that case, `modifyChild()` callbacks for the subpopulation of `parent1` are used (since that is the maternal parent).

If the `modifyChild()` callback process results in rejection of the proposed child (see section 25.5), a new offspring individual will not be generated, and this method will return `NULL`. To force the generation of an offspring individual from a given pair of parents, you could loop until `addCrossed()` succeeds, but note that if your `modifyChild()` callback rejects all proposed children from those particular parents, your model will then hang, so care must be taken with this approach. Usually, nonWF models do not force generation of offspring in this manner; rejection of a proposed offspring by a `modifyChild()` callback typically represents a phenomenon such as post-mating reproductive isolation or lethal genetic incompatibilities that would reduce the expected litter size, so the default behavior is typically desirable.

Note that this method is only for use in nonWF models, in which offspring generation is managed manually by the model script; in such models, `addCrossed()` must be called only from `reproduction()` callbacks, and may not be called at any other time. In WF models, offspring generation is managed automatically by the SLiM core.

```
– (No<Individual>$)addEmpty([Nfs$ sex = NULL], [Nl$ genome1Null = NULL],  
    [Nl$ genome2Null = NULL])
```

Generates a new offspring individual with empty genomes (i.e., containing no mutations), queues it for addition to the target subpopulation, and returns it. The new offspring will not be visible as a member of the target subpopulation until the end of the offspring generation tick cycle stage. No `recombination()` or `mutation()` callbacks will be called. The target subpopulation will be used to locate applicable `modifyChild()` callbacks governing the generation of the offspring individual (unlike the other `addX()` methods, because there is no parental individual to reference). The offspring is considered to have no parents for the purposes of pedigree tracking. The `sex` parameter is treated as in `addCrossed()`.

By default – when `genome1Null` and `genome2Null` are both `NULL` – null genomes will be generated instead of empty genomes only in sex-chromosome simulations, where the sex chromosome that is not being simulated is represented by a null genome; otherwise, empty genomes rather than null genomes will be created. This default behavior can be changed by passing `T` or `F` for `genome1Null` or `genome2Null`, which will force the corresponding offspring genome to be null (`T`) or non-null (`F`). The behavior in sex-chromosome simulations cannot be changed, since the presence of null genomes there is dictated by sex, but `T` or `F` may be passed as long as it matches what SLiM would do anyway. In all other simulations there is little point in passing `F` (since that would be the default behavior anyway), but passing `T` can be used to make one or both genomes be null genomes, which can be useful for, e.g., modeling haploids (for which, by convention, the *second* genome is usually a null genome in SLiM).

Note that this method is only for use in nonWF models. See `addCrossed()` for further general notes on the addition of new offspring individuals.

```
– (No<Individual>$)addRecombinant(No<Genome>$ strand1, No<Genome>$ strand2,
  Ni breaks1, No<Genome>$ strand3, No<Genome>$ strand4, Ni breaks2,
  [Nfs$ sex = NULL])
```

Generates a new offspring individual from the given parental genomes with the specified crossover breakpoints, queues it for addition to the target subpopulation, and returns it. The new offspring will not be visible as a member of the target subpopulation until the end of the offspring generation tick cycle stage. The target subpopulation will be used to locate applicable `mutation()` and `modifyChild()` callbacks governing the generation of the offspring individual (unlike the other `addX()` methods, because there are potentially up to four parental individuals to reference); `recombination()` callbacks will not be called by this method. This method is an advanced feature; most models will use `addCrossed()`, `addSelfed()`, or `addCloned()` instead.

This method supports several possible configurations for `strand1`, `strand2`, and `breaks1` (and the same applies for `strand3`, `strand4`, and `breaks2`). If `strand1` and `strand2` are both `NULL`, the corresponding genome in the generated offspring will be empty, as from `addEmpty()`, with no parental genomes and no added mutations; in this case, `breaks1` must be `NULL` or zero-length. If `strand1` is non-`NULL` but `strand2` is `NULL`, the corresponding genome in the generated offspring will be a clonal copy of `strand1` with mutations added, as from `addCloned()`; in this case, `breaks1` must similarly be `NULL` or zero-length. If `strand1` and `strand2` are both non-`NULL`, the corresponding genome in the generated offspring will result from recombination between `strand1` and `strand2` with mutations added, as from `addCrossed()`, with `strand1` being the initial copy strand; copying will switch between strands at each breakpoint in `breaks1`, which must be non-`NULL` but need not be sorted or unique (SLiM will sort and unique the supplied breakpoints internally). (It is not currently legal for `strand1` to be `NULL` and `strand2` non-`NULL`; that variant may be assigned some meaning in future.) Again, this discussion applies equally to `strand3`, `strand4`, and `breaks2`, *mutatis mutandis*. Note that when new mutations are generated by `addRecombinant()`, their `subpopID` property will be the `id` of the offspring's subpopulation, since the parental subpopulation is ambiguous in the general case; this behavior differs from the other `add...()` methods.

The `sex` parameter is interpreted exactly as in `addCrossed()`; see that method for discussion. If the offspring sex is specified in any way (i.e., if `sex` is non-`NULL`), the strands provided must be compatible with the sex chosen. If the offspring sex is not specified (i.e., if `sex` is `NULL`), the sex will be inferred from the strands provided where possible (when modeling an X or Y chromosome), or will be chosen randomly otherwise (when modeling autosomes); it will *not* be inferred from the sex of the individuals possessing the parental strands, even when the reproductive mode is essentially clonal from a single parent, since such inference would be ambiguous in the general case. Similarly, the offspring is considered to have no parents for the purposes of pedigree tracking, since there may be more than two “parents” in the general case. When modeling the X or Y, `strand1` and `strand2` must be X genomes (or `NULL`), and `strand3` and `strand4` must both be X genomes or both be Y genomes (or `NULL`).



These semantics allow several uses for `addRecombinant()`. When all strands are non-NULL, it is similar to `addCrossed()` except that the recombination breakpoints are specified explicitly, allowing very precise offspring generation without having to override SLiM's breakpoint generation with a `recombination()` callback. When only `strand1` and `strand3` are supplied, it is very similar to `addCloned()`, creating a clonal offspring, except that the two parental genomes need not belong to the same individual (whatever that might mean biologically). Supplying only `strand1` is useful for modeling clonally reproducing haploids; the second genome of every offspring will be kept empty and will not receive new mutations. For a model of clonally reproducing haploids that undergo horizontal gene transfer (HGT), supplying only `strand1` and `strand2` will allow HGT from `strand2` to replace segments of an otherwise clonal copy of `strand1`, while the second genome of the generated offspring will again be kept empty; this could be useful for modeling bacterial conjugation, for example. Other variations are also possible.

The value of the `meanParentAge` property of the generated offspring is calculated from the mean parent age of each of its two genomes (whether they turn out to be null genomes or not); that may be an average of two values (if both offspring genomes have at least one parent), a single value (if one offspring genome has no parent), or no values (if both offspring genomes have no parent, in which case `0.0` results). The mean parent age of a given offspring genome is the mean of the ages of the parents of the two strands used to generate that offspring genome; if one strand is NULL then the mean parent age for that offspring genome is the age of the parent of the non-NULL strand, while if both strands are NULL then that offspring genome is parentless and is not used in the final calculation. In other words, if one offspring genome has two parents with ages A and B, and the other offspring genome has one parent with age C, the `meanParentAge` of the offspring will be  $(A+B+C+C) / 4$ , not  $(A+B+C) / 3$ .

Note that gene conversion tracts are not explicitly supported by this method; the `breaks` vectors provide crossover breakpoints, which may be used to implement crossovers or simple gene conversion tracts. There is no way to specify complex gene conversion tracts with heteroduplex mismatch repair.

Note that this method is only for use in nonWF models. See `addCrossed()` for further general notes on the addition of new offspring individuals.

#### – (No<Individual>\$)addSelfed(object<Individual>\$ parent)

Generates a new offspring individual from the given parent by selfing, queues it for addition to the target subpopulation, and returns it. The new offspring will not be visible as a member of the target subpopulation until the end of the offspring generation tick cycle stage. The subpopulation of `parent` will be used to locate applicable `mutation()`, `recombination()`, and `modifyChild()` callbacks governing the generation of the offspring individual.

Since selfing requires that `parent` act as a source of both a male and a female gamete, this method may be called only in hermaphroditic models; calling it in sexual models will result in an error. This method represents a non-incidental selfing event, so the `preventIncidentalSelfing` flag of `initializeSLiMOptions()` has no effect on this method (in contrast to the behavior of `addCrossed()`, where selfing is assumed to be incidental).

Note that this method is only for use in nonWF models. See `addCrossed()` for further general notes on the addition of new offspring individuals.

#### – (float)cachedFitness(Ni indices)

The fitness values calculated for the individuals at the indices given are returned. If NULL is passed, fitness values for all individuals in the subpopulation are returned. The fitness values returned are cached values; `mutationEffect()` and `fitnessEffect()` callbacks are therefore not called as a side effect of this method. It is always an error to call `cachedFitness()` from inside a `mutationEffect()` or `fitnessEffect()` callback, since fitness values are in the middle of being set up. In WF models, it is also an error to call `cachedFitness()` from a `late()` event, because fitness values for the new offspring generation have not yet been calculated and are undefined. In nonWF models, the population may be a mixture of new and old individuals, so instead, `NAN` will be returned as the fitness of any new individuals whose fitness has not yet been calculated. When new



subpopulations are first created with `addSubpop()` or `addSubpopSplit()`, the fitness of all of the newly created individuals is considered to be `1.0` until fitness values are recalculated.

- `(void)configureDisplay([Nf center = NULL], [Nf$ scale = NULL], [Ns$ color = NULL])`

This method customizes the display of the subpopulation in SLiMgui's Population Visualization graph. When this method is called by a model running outside SLiMgui, it will do nothing except type-checking and bounds-checking its arguments. When called by a model running in SLiMgui, the position, size, and color of the subpopulation's displayed circle can be controlled as specified below.

The `center` parameter sets the coordinates of the center of the subpopulation's displayed circle; it must be a `float` vector of length two, such that `center[0]` provides the x-coordinate and `center[1]` provides the y-coordinate. The square central area of the Population Visualization occupies scaled coordinates in `[0,1]` for both `x` and `y`, so the values in `center` must be within those bounds. If a value of `NULL` is provided, SLiMgui's default center will be used (which currently arranges subpopulations in a circle).

The `scale` parameter sets a scaling factor to be applied to the radius of the subpopulation's displayed circle. The default radius used by SLiMgui is a function of the subpopulation's number of individuals; this default radius is then multiplied by `scale`. If a value of `NULL` is provided, the default radius will be used; this is equivalent to supplying a `scale` of `1.0`. Typically the same `scale` value should be used by all subpopulations, to scale all of their circles up or down uniformly, but that is not required.

The `color` parameter sets the color to be used for the displayed subpopulation's circle. Colors may be specified by name, or with hexadecimal RGB values of the form `"#RRGGBB"` (see the Eidos manual). If `color` is `NULL` or the empty string, `""`, SLiMgui's default (fitness-based) color will be used.

- `(void)defineSpatialMap(string$ name, string$ spatiality, numeric values, [logical$ interpolate = F], [Nif valueRange = NULL], [Ns colors = NULL])`

Defines a spatial map for the subpopulation. The map will henceforth be identified by `name`. The map uses the spatial dimensions referenced by `spatiality`, which must be a subset of the dimensions defined for the simulation in `initializeSLiMOptions()`. Spatiality `"x"` is permitted for dimensionality `"x"`; spatiality `"x"`, `"y"`, or `"xy"` for dimensionality `"xy"`; and spatiality `"x"`, `"y"`, `"z"`, `"xy"`, `"yz"`, `"xz"`, or `"xyz"` for dimensionality `"xyz"`. The spatial map is defined by a grid of values supplied in parameter `values`. The remaining optional parameters are described below.

Note that the semantics of this method changed in SLiM 3.5; in particular, the `gridSize` parameter was removed, and the interpretation of the `values` parameter changed as described below. Existing code written prior to SLiM 3.5 will produce an error, due to the removed `gridSize` parameter, and must be revised carefully to obtain the same result, even if `NULL` had been passed for `gridSize` previously.

Beginning in SLiM 3.5, the `values` parameter must be a vector/matrix/array with the number of dimensions appropriate for the declared spatiality of the map; for example, a map with spatiality `"x"` would require a (one-dimensional) vector, spatiality `"xy"` would require a (two-dimensional) matrix, and a map with spatiality of `"xyz"` would require a three-dimensional array. (See the Eidos manual for discussion of vectors, matrices, and arrays.) The data in `values` is interpreted in such a way that a two-dimensional matrix of values, with (0, 0) at upper left and values by column, is transformed into the format expected by SLiM, with (0, 0) at lower left and values by row; in other words, the two-dimensional matrix as it prints in the Eidos console will match the appearance of the two-dimensional spatial map as seen in SLiMgui. *This is a change in behavior from versions prior to SLiM 3.5*; it ensures that images loaded from disk with the Eidos class `Image` can be used directly as spatial maps, achieving the expected orientation, with no need for transposition or flipping. If the spatial map is a three-dimensional array, it is read as successive z-axis "planes", each of which is a two-dimensional matrix that is treated as described above.

Moving on to the other parameters of `defineSpatialMap()`: if `interpolate` is `F`, values across the spatial map are not interpolated; the value at a given point is equal to the nearest value defined by the grid of values specified. If `interpolate` is `T`, values across the spatial map will be interpolated (using

linear, bilinear, or trilinear interpolation as appropriate) to produce spatially continuous variation in values. In either case, the corners of the value grid are exactly aligned with the corners of the spatial boundaries of the subpopulation as specified by `setSpatialBoundary()`, and the value grid is then stretched across the spatial extent of the subpopulation in such a manner as to produce equal spacing between the values along each dimension. The setting of `interpolation` only affects how values between these grid points are calculated: by nearest-neighbor, or by linear interpolation. Interpolation of spatial maps with periodic boundaries is not handled specially; to ensure that the edges of a periodic spatial map join smoothly, simply ensure that the grid values at the edges of the map are identical, since they will be coincident after periodic wrapping.

The `valueRange` and `colors` parameters travel together; either both are unspecified, or both are specified. They control how map values will be transformed into colors, by SLiMgui and by the `spatialMapColor()` method. The `valueRange` parameter establishes the color-mapped range of spatial map values, as a vector of length two specifying a minimum and maximum; this does not need to match the actual range of values in the map. The `colors` parameter then establishes the corresponding colors for values within the interval defined by `valueRange`: values less than or equal to `valueRange[0]` will map to `colors[0]`, values greater than or equal to `valueRange[1]` will map to the last `colors` value, and intermediate values will shade continuously through the specified vector of colors, with interpolation between adjacent colors to produce a continuous spectrum. This is much simpler than it sounds in this description; see the recipes in chapter 15 for an illustration of its use.

Note that at present, SLiMgui will only display spatial maps of spatiality "x", "y", or "xy"; the color-mapping parameters will simply be ignored by SLiMgui for other spatiality values (even if the spatiality is a superset of these values; SLiMgui will not attempt to display an "xyz" spatial map, for example, since it has no way to choose which 2D slice through the xyz space it ought to display). The `spatialMapColor()` method will return translated color strings for any spatial map, however, even if SLiMgui is unable to display the spatial map. If there are multiple spatial maps that SLiMgui is capable of displaying, it choose one for display by default, but other maps may be selected from the context menu on the individuals view (with a right-click or control-click).

- (void)outputMSSample(integer\$ sampleSize, [logical\$ replace = T],  
[string\$ requestedSex = "\*"], [Ns\$ filePath = NULL], [logical\$ append = F],  
[logical\$ filterMonomorphic = F])

Output a random sample from the subpopulation in MS format (see section 26.2.2 for output format details). Positions in the output will span the interval [0,1]. A sample of genomes (not entire individuals, note) of size `sampleSize` from the subpopulation will be output. The sample may be done either with or without replacement, as specified by `replace`; the default is to sample with replacement. A particular sex of individuals may be requested for the sample, for simulations in which sex is enabled, by passing "M" or "F" for `requestedSex`; passing "\*", the default, indicates that genomes from individuals should be selected randomly, without respect to sex. If the sampling options provided by this method are not adequate, see the `outputMS()` method of `Genome` for a more flexible low-level option.

If the optional parameter `filePath` is `NULL` (the default), output will be sent to Eidos's output stream (see section 4.2.1). Otherwise, output will be sent to the filesystem path specified by `filePath`, overwriting that file if `append` is F, or appending to the end of it if `append` is T.

If `filterMonomorphic` is F (the default), all mutations that are present in the sample will be included in the output. This means that some mutations may be included that are actually monomorphic within the sample (i.e., that exist in every sampled genome, and are thus apparently fixed). These may be filtered out with `filterMonomorphic = T` if desired; note that this option means that some mutations that do exist in the sampled genomes might not be included in the output, simply because they exist in every sampled genome.

See `outputSample()` and `outputVCFsSample()` for other output formats. Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

- (void)outputSample(integer\$ sampleSize, [logical\$ replace = T], [string\$ requestedSex = "\*"], [Ns\$ filePath = NULL], [logical\$ append = F])

Output a random sample from the subpopulation in SLiM's native format (see section 26.2.1 for output format details). A sample of genomes (not entire individuals, note) of size `sampleSize` from the subpopulation will be output. The sample may be done either with or without replacement, as specified by `replace`; the default is to sample with replacement. A particular sex of individuals may be requested for the sample, for simulations in which sex is enabled, by passing "M" or "F" for `requestedSex`; passing "\*", the default, indicates that genomes from individuals should be selected randomly, without respect to sex. If the sampling options provided by this method are not adequate, see the `output()` method of `Genome` for a more flexible low-level option.

If the optional parameter `filePath` is `NULL` (the default), output will be sent to Eidos's output stream (see section 4.2.1). Otherwise, output will be sent to the filesystem path specified by `filePath`, overwriting that file if `append` is F, or appending to the end of it if `append` is T.

See `outputMSSample()` and `outputVCFsample()` for other output formats. Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

- (void)outputVCFsample(integer\$ sampleSize, [logical\$ replace = T], [string\$ requestedSex = "\*"], [logical\$ outputMultiallelics = T], [Ns\$ filePath = NULL], [logical\$ append = F], [logical\$ simplifyNucleotides = F], [logical\$ outputNonnucleotides = T])

Output a random sample from the subpopulation in VCF format (see sections 26.2.3 and 26.2.4 for output format details). A sample of individuals (not genomes, note – unlike the `outputSample()` and `outputMSSample()` methods) of size `sampleSize` from the subpopulation will be output. The sample may be done either with or without replacement, as specified by `replace`; the default is to sample with replacement. A particular sex of individuals may be requested for the sample, for simulations in which sex is enabled, by passing "M" or "F" for `requestedSex`; passing "\*", the default, indicates that genomes from individuals should be selected randomly, without respect to sex. If the sampling options provided by this method are not adequate, see the `outputVCF()` method of `Genome` for a more flexible low-level option.

If the optional parameter `filePath` is `NULL` (the default), output will be sent to Eidos's output stream (see section 4.2.1). Otherwise, output will be sent to the filesystem path specified by `filePath`, overwriting that file if `append` is F, or appending to the end of it if `append` is T.

The parameters `outputMultiallelics`, `simplifyNucleotides`, and `outputNonnucleotides` affect the format of the output produced; see sections 26.2.3 and 26.2.4 for further discussion.

See `outputMSSample()` and `outputSample()` for other output formats. Output is generally done in a `late()` event, so that the output reflects the state of the simulation at the end of a tick.

- (logical)pointInBounds(float point)

Returns T if `point` is inside the spatial boundaries of the subpopulation, F otherwise. For example, for a simulation with "xy" dimensionality, if `point` contains exactly two values constituting an (x,y) point, the result will be T if and only if `((point[0]>=x0) & (point[0]<=x1) & (point[1]>=y0) & (point[1]<=y1))` given spatial bounds (`x0`, `y0`, `x1`, `y1`). This method is useful for implementing absorbing or reprising boundary conditions. This may only be called in simulations for which continuous space has been enabled with `initializeSLiMOptions()`.

The length of `point` must be an exact multiple of the dimensionality of the simulation; in other words, `point` may contain values comprising more than one point. In this case, a logical vector will be returned in which each element is T if the corresponding point in `point` is inside the spatial boundaries of the subpopulation, F otherwise.

- (float)pointPeriodic(float point)

Returns a revised version of `point` that has been brought inside the periodic spatial boundaries of the subpopulation (as specified by the `periodicity` parameter of `initializeSLiMOptions()`) by wrapping around periodic spatial boundaries. In brief, if a coordinate of `point` lies beyond a periodic

spatial boundary, that coordinate is wrapped around the boundary, so that it lies inside the spatial extent by the same magnitude that it previously lay outside, but on the opposite side of the space; in effect, the two edges of the periodic spatial boundary are seamlessly joined. This is done iteratively until all coordinates lie inside the subpopulation's periodic boundaries. Note that non-periodic spatial boundaries are not enforced by this method; they should be enforced using `pointReflected()`, `pointStopped()`, or some other means of enforcing boundary constraints (which can be used after `pointPeriodic()` to bring the remaining coordinates into bounds; coordinates already brought into bounds by `pointPeriodic()` will be unaffected by those calls). This method is useful for implementing periodic boundary conditions. This may only be called in simulations for which continuous space and at least one periodic spatial dimension have been enabled with `initializeSLiMOptions()`.

The length of `point` must be an exact multiple of the dimensionality of the simulation; in other words, `point` may contain values comprising more than one point. In this case, each point will be processed as described above and a new vector containing all of the processed points will be returned.

– `(float)pointReflected(float point)`

Returns a revised version of `point` that has been brought inside the spatial boundaries of the subpopulation by reflection. In brief, if a coordinate of `point` lies beyond a spatial boundary, that coordinate is reflected across the boundary, so that it lies inside the boundary by the same magnitude that it previously lay outside the boundary. This is done iteratively until all coordinates lie inside the subpopulation's boundaries. This method is useful for implementing reflecting boundary conditions. This may only be called in simulations for which continuous space has been enabled with `initializeSLiMOptions()`.

The length of `point` must be an exact multiple of the dimensionality of the simulation; in other words, `point` may contain values comprising more than one point. In this case, each point will be processed as described above and a new vector containing all of the processed points will be returned.

– `(float)pointStopped(float point)`

Returns a revised version of `point` that has been brought inside the spatial boundaries of the subpopulation by clamping. In brief, if a coordinate of `point` lies beyond a spatial boundary, that coordinate is set to exactly the position of the boundary, so that it lies on the edge of the spatial boundary. This method is useful for implementing stopping boundary conditions. This may only be called in simulations for which continuous space has been enabled with `initializeSLiMOptions()`.

The length of `point` must be an exact multiple of the dimensionality of the simulation; in other words, `point` may contain values comprising more than one point. In this case, each point will be processed as described above and a new vector containing all of the processed points will be returned.

– `(float)pointUniform([integer$ n = 1])`

Returns a new point (or points, for  $n > 1$ ) generated from uniform draws for each coordinate, within the spatial boundaries of the subpopulation. The returned vector will contain  $n$  points, each comprised of a number of coordinates equal to the dimensionality of the simulation, so it will be of total length  $n \times \text{dimensionality}$ . This may only be called in simulations for which continuous space has been enabled with `initializeSLiMOptions()`.

– `(void)removeSubpopulation(void)`

Removes this subpopulation from the model. The subpopulation is immediately removed from the list of active subpopulations, and the symbol representing the subpopulation is undefined. The subpopulation object itself remains unchanged until children are next generated (at which point it is deallocated), but it is no longer part of the simulation and should not be used.

Note that this method is only for use in nonWF models, in which there is a distinction between a subpopulation being empty and a subpopulation being removed from the simulation; an empty subpopulation may be re-colonized by migrants, whereas as a removed subpopulation no longer exists at all. WF models do not make this distinction; when a subpopulation is empty it is

automatically removed. WF models should therefore call `setSubpopulationSize(0)` instead of this method; `setSubpopulationSize()` is the standard way for WF models to change the subpopulation size, including to a size of 0.

- `(object<Individual>)sampleIndividuals(integer$ size, [logical$ replace = F], [No<Individual>$ exclude = NULL], [Ns$ sex = NULL], [Ni$ tag = NULL], [Ni$ minAge = NULL], [Ni$ maxAge = NULL], [Nl$ migrant = NULL])`

Returns a vector of individuals, of size less than or equal to parameter `size`, sampled from the individuals in the target subpopulation. Sampling is done without replacement if `replace` is F (the default), or with replacement if `replace` is T. The remaining parameters specify constraints upon the pool of individuals that will be considered candidates for the sampling. Parameter `exclude`, if non-NULL, may specify a specific individual that should not be considered a candidate (typically the focal individual in some operation). Parameter `sex`, if non-NULL, may specify a sex ("M" or "F") for the individuals to be drawn, in sexual models. Parameter `tag`, if non-NULL, may specify a tag value for the individuals to be drawn. Parameters `minAge` and `maxAge`, if non-NULL, may specify a minimum or maximum age for the individuals to be drawn, in nonWF models. Parameter `migrant`, if non-NULL, may specify a required value for the `migrant` property of the individuals to be drawn (so T will require that individuals be migrants, F will require that they not be). If the candidate pool is smaller than the requested sample size, all eligible candidates will be returned (in randomized order); the result will be a zero-length vector if no eligible candidates exist (unlike `sample()`).

This method is similar to getting the `individuals` property of the subpopulation, using operator `[]` to select only individuals with the desired properties, and then using `sample()` to sample from that candidate pool. However, besides being much simpler than the equivalent Eidos code, it is also much faster, and it does not fail if less than the full sample size is available. See `subsetIndividuals()` for a similar method that returns a full subset, rather than a sample.

- `(void)setCloningRate(numeric rate)`

Set the cloning rate of this subpopulation. The rate is changed to `rate`, which should be between 0.0 and 1.0, inclusive. Clonal reproduction can be enabled in both non-sexual (i.e. hermaphroditic) and sexual simulations. In non-sexual simulations, `rate` must be a singleton value representing the overall clonal reproduction rate for the subpopulation. In sexual simulations, `rate` may be either a singleton (specifying the clonal reproduction rate for both sexes) or a vector containing two numeric values (the female and male cloning rates specified separately, at indices 0 and 1 respectively). During mating and offspring generation, the probability that any given offspring individual will be generated by cloning – by asexual reproduction without gametes or meiosis – will be equal to the cloning rate (for its sex, in sexual simulations) set in the parental (not the offspring!) subpopulation.

- `(void)setMigrationRates(io<Subpopulation> sourceSubpops, numeric rates)`

Set the migration rates to this subpopulation from the subpopulations in `sourceSubpops` to the corresponding rates specified in `rates`; in other words, `rates` gives the expected fractions of the children in this subpopulation that will subsequently be generated from parents in the subpopulations `sourceSubpops` (see section 22.2.1). This method will only set the migration fractions from the subpopulations given; migration rates from other subpopulations will be left unchanged (explicitly set a zero rate to turn off migration from a given subpopulation). The type of `sourceSubpops` may be either `integer`, specifying subpopulations by identifier, or `object`, specifying subpopulations directly.

- `(void)setSelfingRate(numeric$ rate)`

Set the selfing rate of this subpopulation. The rate is changed to `rate`, which should be between 0.0 and 1.0, inclusive. Selfing can only be enabled in non-sexual (i.e. hermaphroditic) simulations. During mating and offspring generation, the probability that any given offspring individual will be generated by selfing – by self-fertilization via gametes produced by meiosis by a single parent – will be equal to the selfing rate set in the parental (not the offspring!) subpopulation (see section 22.2.1).



– (void)setSexRatio(float\$ sexRatio)

Set the sex ratio of this subpopulation to **sexRatio**. As defined in SLiM, this is actually the fraction of the subpopulation that is male; in other words, the M:(M+F) ratio. This will take effect when children are next generated; it does not change the current subpopulation state. Unlike the selfing rate, the cloning rate, and migration rates, the sex ratio is deterministic: SLiM will generate offspring that exactly satisfy the requested sex ratio (within integer roundoff limits). See section 22.2.1 for further details.

– (void)setSpatialBounds(numeric bounds)

Set the spatial boundaries of the subpopulation to **bounds**. This method may be called only for simulations in which continuous space has been enabled with `initializeSLiMOptions()`. The length of **bounds** must be double the spatial dimensionality, so that it supplies both minimum and maximum values for each coordinate. More specifically, for a dimensionality of "x", **bounds** should supply (x0, x1) values; for dimensionality "xy" it should supply (x0, y0, x1, y1) values; and for dimensionality "xyz" it should supply (x0, y0, z0, x1, y1, z1) (in that order). These boundaries will be used by SLiMgui to calibrate the display of the subpopulation, and will be used by methods such as `pointInBounds()`, `pointReflected()`, `pointStopped()`, and `pointUniform()`. The default spatial boundaries for all subpopulations span the interval [0,1] in each dimension. Spatial dimensions that are periodic (as established with the `periodicity` parameter to `initializeSLiMOptions()`) must have a minimum coordinate value of 0.0 (a restriction that allows the handling of periodicity to be somewhat more efficient). The current spatial bounds for the subpopulation may be obtained through the `spatialBounds` property.

– (void)setSubpopulationSize(integer\$ size)

Set the size of this subpopulation to **size** individuals. This will take effect when children are next generated; it does not change the current subpopulation state. Setting a subpopulation to a size of 0 does have some immediate effects that serve to disconnect it from the simulation: the subpopulation is removed from the list of active subpopulations, the subpopulation is removed as a source of migration for all other subpopulations, and the symbol representing the subpopulation is undefined. In this case, the subpopulation itself remains unchanged until children are next generated (at which point it is deallocated), but it is no longer part of the simulation and should not be used.

– (string)spatialMapColor(string\$ name, numeric value)

Looks up the spatial map indicated by **name**, and uses its color-translation machinery (as defined by the `valueRange` and `colors` parameters to `defineSpatialMap()`) to translate each element of **value** into a corresponding color string. If the spatial map does not have color-translation capabilities, an error will result. See the documentation for `defineSpatialMap()` for information regarding the details of color translation. See the Eidos manual for further information on color strings.

– (object<Image>)\$spatialMapImage(string\$ name, [Ni\$ width = NULL],  
[Ni\$ height = NULL], [logical\$ centers = F], [logical\$ color = T])

Looks up the spatial map indicated by **name**, and returns an `Image` object sampled from it. The image will be **width** pixels wide and **height** pixels tall; the intrinsic size of the spatial map itself will be used if one of these parameters is `NULL`. The image will be oriented in the same way as it is displayed in SLiMgui (which conceptually entails a transformation from matrix coordinates, which store values by column, to standard image coordinates, which store values by row; see the Eidos manual's documentation of `Image` for details). This method may only be called for 2D spatial maps at present.

The sampling of the spatial map can be done in one of two ways, as controlled by the `centers` parameter. If `centers` is `T`, a  $(width+1) \times (height+1)$  grid of lines that delineates  $width \times height$  rectangular pixels will be overlaid on top of the spatial map, and values will be sampled from the spatial map at the *center* of each of these pixels. If `centers` is `F` (the default), a  $width \times height$  grid of lines will be overlaid on top of the spatial map, and values will be sampled from the spatial map at the *vertices* of the grid. If interpolation is not enabled for the spatial map, these two options will both recover the original matrix of values used to define the spatial map (assuming, here and below, that



width and height are NULL). If interpolation is enabled for the spatial map, however, `centers == F` will recover the original values, but will not capture the “typical” value of each pixel in the image; `centers == T`, on the other hand, will not recover the original values, but will capture the “typical” value of each pixel in the image (i.e., the value at the center of each pixel, as produced by interpolation). The figures in section 15.11 may be helpful for visualizing the difference between these options; the overlaid grids span the full extent of the spatial map, just as shown in that section.

If `color` is T (the default), the `valueRange` and `colors` parameters supplied to `defineSpatialMap()` will be used to translate map values to RGB color values as described in the documentation of that method, providing the same appearance as in SLiMgui; of course those parameters must have been supplied, otherwise an error will result. If `color` is F, on the other hand, a grayscale image will be produced that directly reflects the map values without color translation. In this case, this method needs to translate map values, which can have any float value, into grayscale pixel values that are integers in [0, 255]. To do so, the map values are multiplied by 255.0, clamped to [0.0, 255.0], and then rounded to the nearest integer. This translation scheme essentially assumes that map values are in [0, 1]; for spatial maps that were defined using the floatK channel of a grayscale PNG image, this should recover the original image’s pixel values. (If a different translation scheme is desired, `color=T` with the desired `valueRange` and `colors` should be used.)

– (float)spatialMapValue(string\$ name, float point)

Looks up the spatial map indicated by `name`, and uses its mapping machinery (as defined by the `gridSize`, `values`, and `interpolate` parameters to `defineSpatialMap()`) to translate the coordinates of `point` into a corresponding map value. The length of `point` must be equal to the spatiality of the spatial map; in other words, for a spatial map with spatiality “xz”, `point` must be of length 2, specifying the x and z coordinates of the point to be evaluated. Interpolation will automatically be used if it was enabled for the spatial map. Point coordinates are clamped into the range defined by the spatial boundaries, even if the spatial boundaries are periodic; use `pointPeriodic()` to wrap the point coordinates first if desired. See the documentation for `defineSpatialMap()` for information regarding the details of value mapping.

Beginning in SLiM 3.3, `point` may contain more than one point to be looked up. In this case, the length of `point` must be an exact multiple of the spatiality of the spatial map; for a spatial map with spatiality “xz”, for example, the length of `point` must be an exact multiple of 2, and successive pairs of elements from `point` (elements 0 and 1, then elements 2 and 3, etc.) will be taken as the x and z coordinates of the points to be evaluated. This allows `spatialMapValue()` to be used in a vectorized fashion.

– (object<Individual>)subsetIndividuals([No<Individual>\$ exclude = NULL],  
[Ns\$ sex = NULL], [Ni\$ tag = NULL], [Ni\$ minAge = NULL], [Ni\$ maxAge = NULL],  
[Nl\$ migrant = NULL])

Returns a vector of individuals subset from the individuals in the target subpopulation. The parameters specify constraints upon the subset of individuals that will be returned. Parameter `exclude`, if non-NULL, may specify a specific individual that should not be included (typically the focal individual in some operation). Parameter `sex`, if non-NULL, may specify a sex (“M” or “F”) for the individuals to be returned, in sexual models. Parameter `tag`, if non-NULL, may specify a tag value for the individuals to be returned. Parameters `minAge` and `maxAge`, if non-NULL, may specify a minimum or maximum age for the individuals to be returned, in nonWF models. Parameter `migrant`, if non-NULL, may specify a required value for the `migrant` property of the individuals to be returned (so T will require that individuals be migrants, F will require that they not be).

This method is shorthand for getting the `individuals` property of the subpopulation, and then using operator `[]` to select only individuals with the desired properties; besides being much simpler than the equivalent Eidos code, it is also much faster. See `sampleIndividuals()` for a similar method that returns a sample taken from a chosen subset of individuals.

– (void)takeMigrants(object<Individual> migrants)

Immediately moves the individuals in `migrants` to the target subpopulation (removing them from their previous subpopulation). Individuals in `migrants` that are already in the target subpopulation are unaffected. Note that the indices and order of individuals and genomes in both the target and source subpopulations will change unpredictably as a side effect of this method.

Note that this method is only for use in nonWF models, in which migration is managed manually by the model script. In WF models, migration is managed automatically by the SLiM core based upon the migration rates set for each subpopulation with `setMigrationRates()`.

## 24.16 Class Substitution

*Superclass: Dictionary*

This class represents a mutation that has been fixed; `Mutation` objects are converted to `Substitution` objects upon fixation. Its properties are thus very similar to those of `Mutation`. Section 1.5.2 presents an overview of the conceptual role of this class.

Since `Substitution` objects represent fixation events that occurred in the past, they are relatively immutable. However, since it may be useful to attach (possibly dynamic) state to substitutions, their `tag` and `subpopID` properties are mutable, and they also provide the same `getValue()` / `setValue()` functionality as `Mutation` (inherited from the Eidos class `Dictionary`). Values set on a `Mutation` object will carry over to the corresponding `Substitution` object automatically upon fixation.

### 24.16.1 Substitution properties

`id => (integer$)`

The identifier for this mutation. Each mutation created during a run receives an immutable identifier that will be unique across the duration of the run, and that identifier is carried over to the `Substitution` object when the mutation fixes.

`fixationTick => (integer$)`

The tick in which this mutation fixed.

`mutationType => (object<MutationType>$)`

The `MutationType` from which this mutation was drawn.

`nucleotide <=> (string$)`

A string representing the nucleotide associated with this mutation; this will be "A", "C", "G", or "T". If the mutation is not nucleotide-based, this property is unavailable.

`nucleotideValue <=> (integer$)`

An integer representing the nucleotide associated with this mutation; this will be 0 (A), 1 (C), 2 (G), or 3 (T). If the mutation is not nucleotide-based, this property is unavailable.

`originTick => (integer$)`

The tick in which this mutation arose.

`position => (integer$)`

The position in the chromosome of this mutation.

`selectionCoeff => (float$)`

The selection coefficient of the mutation, drawn from the distribution of fitness effects of its `MutationType`.

subpopID <=> (integer\$)

The identifier of the subpopulation in which this mutation arose. This value is carried over from the **Mutation** object directly; if a “tag” value was used in the **Mutation** object (see section 24.10.1), that value will carry over to the corresponding **Substitution** object. The **subpopID** in **Substitution** is a read-write property to allow it to be used as a “tag” in the same way, if the origin subpopulation identifier is not needed.

tag <=> (integer\$)

A user-defined integer value. The value of **tag** is carried over automatically from the original **Mutation** object. Apart from that, the value of **tag** is not used by SLiM; it is free for you to use.

#### 24.16.2 Substitution methods

Substitution has no methods apart from those inherited from its superclasses.

### 24.17 SLiM built-in functions

SLiM provides a small number of built-in functions, documented here. Note that these are distinct from the functions built into the Eidos language itself, which are documented in the Eidos manual.

#### 24.17.1 Nucleotide utilities

(is)codonsToAminoAcids(integer codons, [li\$ long = F], [logical\$ paste = T])

Returns the amino acid sequence corresponding to the codon sequence in **codons**. Codons should be represented with values in [0, 63] where AAA is 0, AAC is 1, AAG is 2, and TTT is 63; see **ancestralNucleotides()** for discussion of this encoding. If **long** is F (the default), the standard single-letter codes for amino acids will be used (where Serine is "S", etc.); if **long** is T, the standard three-letter codes will be used instead (where Serine is "Ser", etc.). Beginning in SLiM 3.5, if **long** is 0, integer codes will be used as follows (and **paste** will be ignored):

stop (TAA, TAG, TGA)	0
Alanine	1
Arginine	2
Asparagine	3
Aspartic acid (Aspartate)	4
Cysteine	5
Glutamine	6
Glutamic acid (Glutamate)	7
Glycine	8
Histidine	9
Isoleucine	10
Leucine	11
Lysine	12
Methionine	13
Phenylalanine	14
Proline	15
Serine	16
Threonine	17
Tryptophan	18
Tyrosine	19
Valine	20

There does not seem to be a widely used standard for integer coding of amino acids, so SLiM just numbers them alphabetically, making stop codons 0. If you want a different coding, you can make your own 64-element vector and use it to convert codons to whatever integer codes you need. Other

integer values of `long` are reserved for future use (to support other codings), and will currently produce an error.

When `long` is T or F and `paste` is T (the default), the amino acid sequence returned will be a singleton string, such as "LYATI" (when `long` is F) or "Leu-Tyr-Ala-Thr-Ile" (when `long` is T). When `long` is T or F and `paste` is F, the amino acid sequence will instead be returned as a string vector, with one element per amino acid, such as "L" "Y" "A" "T" "I" (when `long` is F) or "Leu" "Tyr" "Ala" "Thr" "Ile" (when `long` is T). Using the `paste=T` option is considerably faster than using `paste()` in script.

This function interprets the supplied codon sequence as the *sense* strand (i.e., the strand that is *not* transcribed, and which mirrors the mRNA's sequence). This uses the standard DNA codon table directly. For example, if the nucleotide sequence is CAA TTC, that will correspond to a codon vector of 16 61, and will result in the amino acid sequence Gln-Phe ("QF").

**(is)codonsToNucleotides(integer codons, [string\$ format = "string"])**

Returns the nucleotide sequence corresponding to the codon sequence supplied in `codons`. Codons should be represented with values in [0, 63] where AAA is 0, AAC is 1, AAG is 2, and TTT is 63; see `ancestralNucleotides()` for discussion of this encoding.

The `format` parameter controls the format of the returned sequence. It may be "string" to obtain the sequence as a singleton string (e.g., "TATACG"), "char" to obtain it as a string vector of single characters (e.g., "T", "A", "T", "A", "C", "G"), or "integer" to obtain it as an integer vector (e.g., 3, 0, 3, 0, 1, 2), using SLiM's standard code of A=0, C=1, G=2, T=3.

**(float)mm16To256(float mutationMatrix16)**

Returns a 64×4 mutation matrix that is functionally identical to the supplied 4×4 mutation matrix in `mutationMatrix16`. The mutation rate for each of the 64 trinucleotides will depend only upon the central nucleotide of the trinucleotide, and will be taken from the corresponding entry for the same nucleotide in `mutationMatrix16`. This function can be used to easily construct a simple trinucleotide-based mutation matrix which can then be modified so that specific trinucleotides sustain a mutation rate that does not depend only upon their central nucleotide.

See the documentation for `initializeGenomicElementType()` in section 24.1 for further discussion of how these 64×4 mutation matrices are interpreted and used.

**(float)mmJukesCantor(float\$ alpha)**

Returns a mutation matrix representing a Jukes-Cantor (1969) model with mutation rate `alpha` to each possible alternative nucleotide at a site:

$$\begin{pmatrix} 0 & \alpha & \alpha & \alpha \\ \alpha & 0 & \alpha & \alpha \\ \alpha & \alpha & 0 & \alpha \\ \alpha & \alpha & \alpha & 0 \end{pmatrix}$$

This 2×2 matrix is suitable for use with `initializeGenomicElementType()`. Note that the actual mutation rate produced by this matrix is 3\*alpha.

**(float)mmKimura(float\$ alpha, float\$ beta)**

Returns a mutation matrix representing a Kimura (1980) model with transition rate `alpha` and transversion rate `beta`:

$$\begin{pmatrix} 0 & \beta & \alpha & \beta \\ \beta & 0 & \beta & \alpha \\ \alpha & \beta & 0 & \beta \\ \beta & \alpha & \beta & 0 \end{pmatrix}$$

This 2×2 matrix is suitable for use with `initializeGenomicElementType()`. Note that the actual mutation rate produced by this model is alpha+2\*beta.

#### `(integer)nucleotideCounts(is sequence)`

A convenience function that returns an **integer** vector of length four, providing the number of occurrences of A / C / G / T nucleotides, respectively, in the supplied nucleotide sequence. The parameter `sequence` may be a singleton **string** (e.g., "TATA"), a **string** vector of single characters (e.g., "T", "A", "T", "A"), or an **integer** vector (e.g., 3, 0, 3, 0), using SLiM's standard code of A=0, C=1, G=2, T=3.

#### `(float)nucleotideFrequencies(is sequence)`

A convenience function that returns a **float** vector of length four, providing the frequencies of occurrences of A / C / G / T nucleotides, respectively, in the supplied nucleotide sequence. The parameter `sequence` may be a singleton **string** (e.g., "TATA"), a **string** vector of single characters (e.g., "T", "A", "T", "A"), or an **integer** vector (e.g., 3, 0, 3, 0), using SLiM's standard code of A=0, C=1, G=2, T=3.

#### `(integer)nucleotidesToCodons(is sequence)`

Returns the codon sequence corresponding to the nucleotide sequence in `sequence`. The codon sequence is an **integer** vector with values from 0 to 63, based upon successive nucleotide triplets in the nucleotide sequence. The codon value for a given nucleotide triplet XYZ is  $16X + 4Y + Z$ , where X, Y, and Z have the usual values A=0, C=1, G=2, T=3. For example, the triplet AAA has a codon value of 0, AAC is 1, AAG is 2, AAT is 3, ACA is 4, and on upward to TTT which is 63. If the nucleotide sequence AACACATTT is passed in, the codon vector 1 4 63 will therefore be returned. These codon values can be useful in themselves; they can also be passed to `codonsToAminoAcids()` to translate them into the corresponding amino acid sequence if desired.

The nucleotide sequence in `sequence` may be supplied in any of three formats: a **string** vector with single-letter nucleotides (e.g., "T", "A", "T", "A"), a singleton **string** of nucleotide letters (e.g., "TATA"), or an **integer** vector of nucleotide values (e.g., 3, 0, 3, 0) using SLiM's standard code of A=0, C=1, G=2, T=3. If the choice of format is not driven by other considerations, such as ease of manipulation, then the singleton **string** format will certainly be the most memory-efficient for long sequences, and will probably also be the fastest. The nucleotide sequence provided must be a multiple of three in length, so that it translates to an integral number of codons.

#### `(is)randomNucleotides(integer$ length, [Nif basis = NULL], [string$ format = "string"])`

Generates a new random nucleotide sequence with `length` bases. The four nucleotides ACGT are equally probable if `basis` is **NULL** (the default); otherwise, `basis` may be a 4-element **integer** or **float** vector providing relative fractions for A, C, G, and T respectively (these need not sum to 1.0, as they will be normalized). More complex generative models such as Markov processes are not supported intrinsically in SLiM at this time, but arbitrary generated sequences may always be loaded from files on disk.

The `format` parameter controls the format of the returned sequence. It may be "string" to obtain the generated sequence as a singleton **string** (e.g., "TATA"), "char" to obtain it as a **string** vector of single characters (e.g., "T", "A", "T", "A"), or "integer" to obtain it as an **integer** vector (e.g., 3, 0, 3, 0), using SLiM's standard code of A=0, C=1, G=2, T=3. For passing directly to `initializeAncestralNucleotides()`, format "string" (a singleton string) will certainly be the most memory-efficient, and probably also the fastest. Memory efficiency can be a significant consideration; the nucleotide sequence for a chromosome of length  $10^9$  will occupy approximately 1 GB of memory when stored as a singleton string (with one byte per nucleotide), and much more if stored in the other formats. However, the other formats can be easier to work with in Eidos, and so may be preferable for relatively short chromosomes if you are manipulating the generated sequence.

### 24.17.2 Population genetics utilities

SLiM now provides a handful of utility functions for calculating standard population genetic metrics. It should be noted that the correct method for calculating these metrics is often a contentious topic. For one thing, many metrics are defined in more than one way in the literature,

and different definitions of a given metric may yield different numerical results. In addition, applying those definitions to SLiM in particular can be ambiguous, particularly when stacked mutations are present (a sort of nod to the infinite-sites model within SLiM's discrete-sites model), or when multiple mutational lineages representing the same genetic mutation exist (representing identity by descent rather than identity by state); see section 1.5 for discussion of these topics. The functions here are provided for convenience, but represent only one particular set of choices regarding such questions; if your work requires a very specific definition of a metric that involves different choices on these questions, you may then need to implement these metrics in your own way, either in your own Eidos code, or perhaps with post-run analysis of VCF output, etc.

To facilitate an understanding of exactly what these functions do and the assumptions and choices upon which they are based, they are all implemented in Eidos, not in C++. Their Eidos source code can be viewed using the `functionSource()` function in the Eidos console in SLiMgui; for example, by executing `functionSource("calcFST")`.

```
(float$)calcFST(object<Genome> genomes1, object<Genome> genomes2,
  [No<Mutation> muts = NULL], [Ni$ start = NULL], [Ni$ end = NULL])
```

Calculates the  $F_{ST}$  between two `Genome` vectors – typically, but not necessarily, the genomes that constitute two different subpopulations (which we will assume for the purposes of this discussion). In general, higher  $F_{ST}$  indicates greater genetic divergence between subpopulations.

The calculation is done using only the mutations in `muts`; if `muts` is `NULL`, all mutations are used. The `muts` parameter can therefore be used to calculate the  $F_{ST}$  only for a particular mutation type (by passing only mutations of that type).

The calculation can be narrowed to apply to only a window – a subrange of the full chromosome – by passing the interval bounds [`start`, `end`] for the desired window. In this case, the vector of mutations used for the calculation will be subset to include only mutations within the specified window. The default behavior, with `start` and `end` of `NULL`, provides the genome-wide  $F_{ST}$ , which is often used to assess the overall level of genetic divergence between sister species or allopatric subpopulations.

The code for `calcFST()` is, roughly, an Eidos implementation of Wright's definition of  $F_{ST}$  (but see below for further discussion and clarification):

$$F_{ST} = 1 - \frac{H_S}{H_T}$$

where  $H_S$  is the average heterozygosity in the two subpopulations, and  $H_T$  is the total heterozygosity when both subpopulations are combined. In this implementation, the two genome vectors are weighted equally, not weighted by their size. In SLiM 3, the implementation followed Wright's definition closely, and returned the *average of ratios*: `mean(1.0 - H_s/H_t)`, in the Eidos code. In SLiM 4, it returns the *ratio of averages* instead: `1.0 - mean(H_s)/mean(H_t)`. In other words, the  $F_{ST}$  value reported by SLiM 4 is an average across the specified mutations in the two sets of genomes, where  $H_s$  and  $H_t$  are first averaged across all specified mutations prior to taking the ratio of the two. This ratio of averages is less biased than the average of ratios, and is generally considered to be best practice (see, e.g., Bhatia et al., 2013). This means that the behavior of `calcFST()` differs between SLiM 3 and SLiM 4.

The implementation of `calcFST()`, viewable with `functionSource()`, treats every mutation in `muts` as independent in the heterozygosity calculations; in other words, if mutations are stacked, the heterozygosity calculated is *by mutation*, not *by site*. Similarly, if multiple `Mutation` objects exist in different genomes at the same site (whether representing different genetic states, or multiple mutational lineages for the same genetic state), each `Mutation` object is treated separately for purposes of the heterozygosity calculation, just as if they were at different sites. One could regard these choices as embodying an infinite-sites interpretation of the segregating mutations. In most biologically realistic models, such genetic states will be quite rare, and so the impact of these choices will be negligible; however, in some models these distinctions may be important.



```
(float$)calcHeterozygosity(object<Genome> genomes, [No<Mutation> muts = NULL],
  [Ni$ start = NULL], [Ni$ end = NULL])
```

Calculates the heterozygosity for a vector of genomes, based upon the frequencies of mutations in the genomes. The result is the *expected* heterozygosity, for the individuals to which the genomes belong, assuming that they are under Hardy-Weinberg equilibrium; this can be compared to the *observed* heterozygosity of an individual, as calculated by `calcPairHeterozygosity()`. Often `genomes` will be all of the genomes in a subpopulation, or in the entire population, but any genome vector may be used. By default, with `muts=NULL`, the calculation is based upon all mutations in the simulation; the calculation can instead be based upon a subset of mutations, such as mutations of a specific mutation type, by passing the desired vector of mutations for `muts`.

The calculation can be narrowed to apply to only a window – a subrange of the full chromosome – by passing the interval bounds `[start, end]` for the desired window. In this case, the vector of mutations used for the calculation will be subset to include only mutations within the specified window. The default behavior, with `start` and `end` of `NULL`, provides the genome-wide heterozygosity.

The implementation of `calcHeterozygosity()`, viewable with `functionSource()`, treats every mutation as independent in the heterozygosity calculations. One could regard this choice as embodying an infinite-sites interpretation of the segregating mutations. In most biologically realistic models, such genetic states will be quite rare, and so the impact of this choice will be negligible; however, in some models this distinction may be important. See `calcPairHeterozygosity()` for further discussion.

```
(float$)calcInbreedingLoad(object<Genome> genomes,
  [No<MutationType>$ mutType = NULL])
```

Calculates inbreeding load (the haploid number of lethal equivalents, or *B*) for a vector of genomes passed in `genomes`. The calculation can be limited to a focal mutation type passed in `mutType`; if `mutType` is `NULL` (the default), all of the mutations for the focal species will be considered. In any case, only deleterious mutations (those with a negative selection coefficient) will be included in the final calculation.

The inbreeding load is a measure of the quantity of recessive deleterious variation that is heterozygous in a population and can contribute to fitness declines under inbreeding. This function implements the following equation from Morton et al. (1956), which assumes no epistasis and random mating:

$$B = \sum(qs) - \sum(q^2s) - 2\sum(q(1-q)sh)$$

where *q* is the frequency of a given deleterious allele, *s* is the absolute value of the selection coefficient, and *h* is its dominance coefficient. Note that the implementation sets a maximum  $|s|$  of **1.0** (i.e., a lethal allele);  $|s|$  can sometimes be greater than **1.0** when *s* is drawn from a distribution, but in practice an allele with  $s < -1.0$  has the same lethal effect as when  $s = -1.0$ . Also note that this implementation will not work when the model changes the dominance coefficients of mutations using `mutationEffect()` callbacks, since it relies on the `dominanceCoeff` property of `MutationType`. Finally, note that, to estimate the diploid number of lethal equivalents (*2B*), the result from this function can simply be multiplied by two.

This function was contributed by Chris Kyriazis; thanks, Chris!

```
(float$)calcPairHeterozygosity(object<Genome>$ genome1, object<Genome>$ genome2,
  [Ni$ start = NULL], [Ni$ end = NULL], [logical$ infiniteSites = T])
```

Calculates the heterozygosity for a pair of genomes; these will typically be the two genomes of a diploid individual (`individual.genome1` and `individual.genome2`), but any two genomes may be supplied.

The calculation can be narrowed to apply to only a window – a subrange of the full chromosome – by passing the interval bounds `[start, end]` for the desired window. In this case, the vector of mutations used for the calculation will be subset to include only mutations within the specified window. The default behavior, with `start` and `end` of `NULL`, provides the genome-wide heterozygosity.

The implementation `calcPairHeterozygosity()`, viewable with `functionSource()`, treats every mutation as independent in the heterozygosity calculations by default (i.e., with `infiniteSites=T`). If mutations are stacked, the heterozygosity calculated therefore depends upon the number of *unshared mutations*, not the number of *differing sites*. Similarly, if multiple `Mutation` objects exist in different genomes at the same site (whether representing different genetic states, or multiple mutational lineages for the same genetic state), each `Mutation` object is treated separately for purposes of the heterozygosity calculation, just as if they were at different sites. One could regard these choices as embodying an infinite-sites interpretation of the segregating mutations. In most biologically realistic models, such genetic states will be quite rare, and so the impact of this choice will be negligible; however, in some models this distinction may be important. The behavior of `calcPairHeterozygosity()` can be switched to calculate based upon the number of differing sites, rather than the number of unshared mutations, by passing `infiniteSites=F`.

```
(float$)calcVA(object<Individual> individuals, io<MutationType>$ mutType)
```

Calculates  $V_A$ , the additive genetic variance, among a vector `individuals`, in a particular mutation type `mutType` that represents quantitative trait loci (QTLs) influencing a quantitative phenotypic trait. The `mutType` parameter may be either an integer representing the ID of the desired mutation type, or a `MutationType` object specified directly.

This function assumes that mutations of type `mutType` encode their effect size upon the quantitative trait in their `selectionCoeff` property, as is fairly standard in SLiM (see, e.g., section 13.2). The implementation of `calcVA()`, which is viewable with `functionSource()`, is quite simple; if effect sizes are stored elsewhere (such as with `setValue()`, as in section 13.5), a new user-defined function following the pattern of `calcVA()` can easily be written.

```
(float$)calcWattersonsTheta(object<Genome> genomes, [No<Mutation> muts = NULL],
  [Ni$ start = NULL], [Ni$ end = NULL])
```

Calculates Watterson's theta (a metric of genetic diversity comparable to heterozygosity) for a vector of genomes, based upon the mutations in the genomes. Often `genomes` will be all of the genomes in a subpopulation, or in the entire population, but any genome vector may be used. By default, with `muts=NULL`, the calculation is based upon all mutations in the simulation; the calculation can instead be based upon a subset of mutations, such as mutations of a specific mutation type, by passing the desired vector of mutations for `muts`.

The calculation can be narrowed to apply to only a window – a subrange of the full chromosome – by passing the interval bounds `[start, end]` for the desired window. In this case, the vector of mutations used for the calculation will be subset to include only mutations within the specified window. The default behavior, with `start` and `end` of `NULL`, provides the genome-wide Watterson's theta.

The implementation of `calcWattersonsTheta()`, viewable with `functionSource()`, treats every mutation as independent in the heterozygosity calculations. One could regard this choice as embodying an infinite-sites interpretation of the segregating mutations, as with `calcHeterozygosity()`. In most biologically realistic models, such genetic states will be quite rare, and so the impact of this assumption will be negligible; however, in some models this distinction may be important. See `calcPairHeterozygosity()` for further discussion.

### 24.17.3 Other utilities

This section documents SLiM built-in functions that do not fit into the categories covered by the previous sections.

```
(float)summarizeIndividuals(object<Individual> individuals, integer dim,
  numeric spatialBounds, string$ operation, [Nlif$ empty = 0.0],
  [logical$ perUnitArea = F], [Ns$ spatiality = NULL])
```

Returns a vector, matrix, or array that summarizes spatial patterns of information related to the individuals in `individuals`. In essence, those individuals are assigned into *bins* according to their spatial position, and then a summary value for each bin is calculated based upon the individuals each bin contains. The individuals might be binned in one dimension (resulting in a vector of summary

values), in two dimensions (resulting in a matrix), or in three dimensions (resulting in an array). Typically the spatiality of the result (the dimensions into which the individuals are binned) will match the dimensionality of the model, as indicated by the default value of `NULL` for the optional `spatiality` parameter; for example, a two-dimensional ("xy") model would by default produce a two-dimensional matrix as a summary. However, a spatiality that is more restrictive than the model dimensionality may be passed; for example, in a two-dimensional ("xy") model a `spatiality` of "y" could be passed to summarize individuals into a vector, rather than a matrix, assigning them to bins based only upon their y position (i.e., the value of their y property). Whatever spatiality is chosen, the parameter `dim` provides the dimensions of the desired result, in the same form that the `dim()` function does: first the number of rows, then the number of columns, and then the number of planes, as needed (see the Eidos manual for discussion of matrices, arrays, and `dim()`). The length of `dims` must match the requested spatiality; for spatiality "xy", for example, `dims` might be `c(50,100)` to request that the returned matrix have 50 rows and 100 columns. The result vector/matrix/array is in the correct orientation to be directly usable as a spatial map, by passing it to the `defineSpatialMap()` method of `Subpopulation`. For further discussion of dimensionality and spatiality, see section 24.1 on `initializeInteractionType()`, and section 24.8 on `InteractionType`.

The `spatialBounds` parameter defines the spatial boundaries within which the individuals are binned. Typically this is the spatial bounds of a particular subpopulation, within which the individuals reside; for individuals in `p1`, for example, you would likely pass `p1.spatialBounds` for this. However, this is not required; individuals may come from any or all subpopulations in the model, and `spatialBounds` may be any bounds of non-zero area (if an individual falls outside of the given spatial bounds, it is excluded, as if it were not in `individuals` at all). If you have multiple subpopulations that conceptually reside within the same overall coordinate space, for example, that can be accommodated here. The bounds are supplied in the dimensionality of the model, in the same form as for `Subpopulation`; for an "xy" model, for example, they are supplied as a four-element vector of the form `c(x0, y0, x1, y1)` even if the summary is being produced with spatiality "y". To produce the result, a grid with dimensions defined by `dims` is conceptually stretched out across the given spatial bounds, such that the *centers* of the edge and corner grid squares are aligned with the limits of the spatial bounds. This matches the way that `defineSpatialMap()` defines its maps; see section 15.11 for illustration.

The particular summary produced depends upon the parameters `operation` and `empty`. Consider a single grid square represented by a single element in the result. That grid square contains zero or more of the individuals in `individuals`. If it contains zero individuals *and* `empty` is not `NULL`, the `empty` value is used for the result, regardless of `operation`, providing specific, separate control over the treatment of empty grid squares. If `empty` is `NULL`, this separate control over the treatment of empty grid squares is declined; empty grid squares will be handled through the standard mechanism described next. In all other cases for the given grid square – when it contains more than zero individuals, or when `empty` is `NULL` – `operation` is executed as an Eidos *lambda*, a small snippet of code, supplied as a singleton string, that is executed in a manner similar to a function call. Within the execution of the `operation lambda`, a constant named `individuals` is defined to be the focal individuals being evaluated – all of the individuals within that grid square. This `lambda` should evaluate to a singleton `logical`, `integer`, or `float` value, comprising the result value for the grid square; these types will all be coerced to `float` (T being 1 and F being 0).

Two examples may illustrate the use of `empty` and `operation`. To produce a summary indicating presence/absence, simply use the default of `0.0` for `empty`, and `"1.0; "` (or `"1; "`, or `"T; "`) for `operation`. This will produce `0.0` for empty grid squares, and `1.0` for those that contain at least one individual. Note that the use of `empty` is essential here, because `operation` doesn't even check whether individuals are present or not. To produce a summary with a count of the number of individuals in each grid square, again use the default of `0.0` for `empty`, but now use an `operation` of `"individuals.size();"`, counting the number of individuals in each grid square. In this case, `empty` could be `NULL` instead and `operation` would still produce the correct result; but using `empty` makes `summarizeIndividuals()` more efficient since it allows the execution of `operation` to be skipped for those squares.

Lambdas are not limited in their complexity; they can use `if`, `for`, etc., and can call methods and functions. A typical operation to compute the mean phenotype in a quantitative genetic model that stores phenotype values in `tagF`, for example, would be `"mean(individuals.tagF);"`, and this is still quite simple compared to what is possible. However, keep in mind that the lambda will be evaluated for every grid cell (or at least those that are non-empty), so efficiency can be a concern, and you may wish to pre-calculate values shared by all of the lambda calls, making them available to your lambda code using `defineGlobal()` or `defineConstant()`.

There is one last twist, if `perUnitArea` is T: values are divided by the area (or length, in 1D, or volume, in 3D) that their corresponding grid cell comprises, so that each value is in units of “per unit area” (or “per unit length”, or “per unit volume”). The total area of the grid is defined by the spatial bounds, and the area of a given grid cell is defined by the portion of the spatial bounds that is within that cell. This is not the same for all grid cells; grid cells that fall partially outside `spatialBounds` (because, remember, the *centers* of the edge/corner grid cells are aligned with the limits of `spatialBounds`) will have a smaller area inside the bounds. For an “xy” spatiality summary, for example, corner cells have only a quarter of their area inside `spatialBounds`, while edge elements have half of their area inside `spatialBounds`; for purposes of `perUnitArea`, then, their respective areas are  $\frac{1}{4}$  and  $\frac{1}{2}$  the area of an interior grid cell. By default, `perUnitArea` is F, and no scaling is performed. Whether you want `perUnitArea` to be F or T depends upon whether the summary you are producing is, conceptually, “per unit area”, such as density (individuals per unit area) or local competition strength (total interaction strength per unit area), or is not, such as “mean individual age”, or “maximum `tag` value”. For the previous example of counting individuals with an operation of `"individuals.size();"`, a value of F for `perUnitArea` (the default) will produce a simple *count* of individuals in each grid square, whereas with T it would produce the *density* of individuals in each grid square.

```
(object<Dictionary>$)treeSeqMetadata(string$ filePath, [logical$ userData = T])
```

Returns a `Dictionary` containing top-level metadata from the `.trees` (tree-sequence) file at `filePath`. If `userData` is T (the default), the top-level metadata under the `SLiM/user_metadata` key is returned; this is the same metadata that can optionally be supplied to `treeSeqOutput()` in its `metadata` parameter, so it makes it easy to recover metadata that you attached to the tree sequence when it was saved. If `userData` is F, the entire top-level metadata `Dictionary` object is returned; this can be useful for examining the values of other keys under the `SLiM` key (see section 26.4), or values inside the top-level dictionary itself that might have been placed there by `msprime` or other software.

This function can be used to read in parameter values or other saved state (`tag` property values, for example), in order to resuscitate the complete state of a simulation that was written to a `.trees` file. It could be used for more esoteric purposes too, such as to search through `.trees` files in a directory (with the help of the Eidos function `filesAtPath()`) to find those files that satisfy some metadata criterion.

## 25. Writing Eidos events and callbacks

In the preceding recipes, we have seen many examples of Eidos events and callbacks, but we have not systematically described their syntax and semantics. Eidos events and callbacks are typically used in SLiM simulations by defining them in the SLiM input file; they can also be registered with the simulation dynamically at runtime (see sections 9.5.3 and 17.4, for example).

There are two main ways to use Eidos in the input file. One way is by defining an *Eidos event*, a block of Eidos code that is executed during each tick for which it is active. The other way is by defining an *Eidos callback*, a block of code that is called by SLiM in specific circumstances to extend the functionality of SLiM in particular areas for a particular species. One type of Eidos callback, the `initialize()` callback, was described in section 24.1. The sections below will detail the remaining possibilities.

### 25.1 Defining Eidos events

An Eidos event is a block of Eidos code that is executed every tick, within a tick range, to perform a desired task. The syntax of an Eidos event declaration looks like one of these:

```
[id] [t1 [: t2]] first() { ... }
[id] [t1 [: t2]] early() { ... }
[id] [t1 [: t2]] late() { ... }
```

The first declaration declares a `first()` event that executes first thing in the tick cycle. The second declaration declares an `early()` event that executes relatively early in the tick cycle. The third declaration declares a `late()` event that executes near the end of the tick cycle. Exactly when these events run depends upon whether the model is a WF model (see chapter 22 for details on the tick cycle in WF models) or a nonWF model (see chapter 23 for the same about nonWF models).

The `id` is an optional identifier like `s1` (or more generally, `sX`, where `X` is an integer greater than or equal to 0) that defines an identifier that can be used to refer to the script block. In most situations it can be omitted, in which case the `id` is implicitly defined as `-1`, a placeholder value that essentially represents the lack of an identifier value. Supplying an `id` is only useful if you wish to manipulate your script blocks programmatically (see section 25.11).

Then comes a tick or a range of ticks, and then a block of Eidos code enclosed in braces to form a compound statement. A trivial example might look like this:

```
1000:5000 early() {
    catn(community.tick);
}
```

This would print the tick number in every tick in the specified range, which is obviously not very exciting. The broader point is that the Eidos code in the braces `{}` is executed early in every tick within the specified range of ticks. In this case, the tick range is `1000` to `5000`, and so the Eidos event will be executed 4001 times (not 4000!). A range of ticks can be given, as in the example above, or a single tick can be given with a single integer:

```
100 late() {
    print("Finished tick 100!");
}
```

The tick range may also be incompletely specified, with a somewhat idiosyncratic syntax. A range of `1000:` would specify that the event should run in tick `1000` and every subsequent tick until the model finishes; a range of `:1000` would similarly specify that the event should run in the first



tick executed, and every subsequent tick, up to and including tick **1000**. (You might notice that the grammar shown above for the tick range is not quite correct; see section 27.1 for the correct grammar.)

In fact, you can omit specifying a tick altogether, in which case the Eidos event runs every tick. Since it takes a little time to set up the Eidos interpreter and interpret a script, it is advisable to use the narrowest range of ticks possible; however, that is more of a concern with the callbacks we will look at later in this chapter, since they might be called many time in every tick, whereas `first()`, `early()`, and `late()` events will just be called once per tick.

The ticks specified for a Eidos event block can be any positive integer. All blocks that apply to a given time point will be run in *definition order*; blocks specified higher in the input file will run before those specified lower. Sometimes it is desirable to have a script block execute in a tick which is not fixed, but instead depends upon some parameter, defined constant, or calculation; this may be achieved by rescheduling the script block with the Community method `rescheduleScriptBlock()` (see section 20.2 for an example).

In multispecies models, one can optionally provide a `ticks` specifier before the definition of an Eidos event, specifying that the event should only run in ticks in which a particular species is active. That extended syntax looks like this:

```
[ticks species_name] [id] [t1 [: t2]] first() { ... }
[ticks species_name] [id] [t1 [: t2]] early() { ... }
[ticks species_name] [id] [t1 [: t2]] late() { ... }
```

The `species_name` should be the name of a species that was explicitly declared in the multispecies model, as discussed in section 24.1. If the `ticks` specifier is omitted, the event will run in every tick (within the specified tick range).

When Eidos events are executed, several global variables are defined by SLiM for use by the Eidos code. Here is a summary of those SLiM globals:

<code>community</code>	The Community object for the overall simulation
<code>sim</code>	A Species object for the simulated species (in single-species simulations)
<code>g1, ...</code>	GenomicElementType objects for defined genomic element types
<code>i1, ...</code>	InteractionType objects for defined interaction types
<code>m1, ...</code>	MutationType objects representing defined mutation types
<code>p1, ...</code>	Subpopulation objects for existing subpopulations
<code>s1, ...</code>	SLiMEidosBlock objects for named events and callbacks
<code>self</code>	A SLiMEidosBlock object for the script block currently executing

In multispecies models, symbols for each species will be defined instead of `sim`. Note that species symbols such as `sim` are *not* available in `initialize()` callbacks, since the species objects have not yet been initialized (see section 24.1). Similarly, the globals for subpopulations, mutation types, and genomic element types are only available after the point at which those objects have been defined by an `initialize()` callback.

## 25.2 Defining mutation fitness with a `mutationEffect()` callback

An Eidos callback is a block of Eidos code that is called by SLiM in specific circumstances, to allow the customization of particular actions taken by SLiM while running the simulation of a species. Nine types of callbacks are presently supported (in addition to the `initialize()` callbacks described in section 24.1): `mutationEffect()` callbacks, discussed here, and



`fitnessEffect()`, `mateChoice()`, `modifyChild()`, `recombination()`, `interaction()`, `reproduction()`, `mutation()`, and `survival()` callbacks, discussed in the following sections.

A `mutationEffect()` callback is called by SLiM when it is determining the fitness effect of a mutation carried by an individual. Normally, the fitness effect of a mutation is determined by the selection coefficient  $s$  of the mutation and the dominance coefficient  $h$  of the mutation (the latter used only if the individual is heterozygous for the mutation). More specifically, the standard calculation for the fitness effect of a mutation takes one of two forms. If the individual is homozygous, then the fitness effect is  $(1+s)$ , or:

$$w = w * (1.0 + \text{selectionCoefficient}),$$

where  $w$  is the relative fitness of the individual carrying the mutation. If the individual is heterozygous, then the dominance coefficient enters the picture, and the fitness effect is  $(1+hs)$  or:

$$w = w * (1.0 + \text{dominanceCoeff} * \text{selectionCoeff}).$$

The dominance coefficient usually comes from the `dominanceCoeff` property of the mutation's `MutationType`; if the focal individual has only one non-null genome, however, such that the mutation is paired with a null genome (i.e., is actually hemizygous or haploid, not heterozygous), the `haploidDominanceCoeff` property of the `MutationType` is used instead. See section 22.6 for further discussion of this detail.

That is the standard behavior of SLiM, reviewed here to provide a conceptual baseline. Supplying a `mutationEffect()` callback allows you to substitute any calculation you wish for the relative fitness effect of a mutation; the new relative fitness effect computation becomes:

$$w = w * \text{mutationEffect}()$$

where `mutationEffect()` is the value returned by your callback. This value is a multiplicative fitness effect, so `1.0` is neutral, unlike the selection coefficient scale where `0.0` is neutral; be careful with this distinction!

Like Eidos events, `mutationEffect()` callbacks are defined as script blocks in the input file, but they use a variation of the syntax for defining an Eidos event:

```
[id] [t1 [: t2]] mutationEffect(<mut-type-id> [, <subpop-id>]) { ... }
```

For example, if the callback were defined as:

```
1000:2000 mutationEffect(m2, p3) { 1.0; }
```

then a relative fitness of `1.0` (i.e. neutral) would be used for all mutations of mutation type `m2` in subpopulation `p3` from tick `1000` to tick `2000`. The very same mutations, if also present in individuals in other subpopulations, would preserve their normal selection coefficient and dominance coefficient in those other subpopulations; this callback would therefore establish spatial heterogeneity in selection, in which mutation type `m2` was neutral in subpopulation `p3` but under selection in other subpopulations, for the range of ticks given (see the recipe in section 10.2 for a fuller explication of this idea).

In multispecies models, callbacks must be defined with a species specifier that states the species with which species the callback is associated. Such a definition looks like this:

```
species species_name [id] [t1 [: t2]] mutationEffect(...) { ... }
```

It is the same syntax, in other words, except for the species specifier at the beginning, with the name of the species that the callback will modify. As with the ticks specifier for events, this

means the callback will only be called in ticks when the species is active; but the species specifier goes further, making that species the focal species for the callback.

In addition to the SLiM globals listed in section 25.1, a `mutationEffect()` callback is supplied with some additional information passed through “pseudo-parameters”, variables that are defined by SLiM within the context of the callback’s code to supply the callback with relevant information:

<code>mut</code>	A <code>Mutation</code> object, the mutation whose relative fitness is being evaluated
<code>homozygous</code>	A value of T (the mutation is homozygous), F (heterozygous), or NULL (it is paired with a null chromosome, and is thus hemizygous or haploid)
<code>effect</code>	The default relative fitness value calculated by SLiM
<code>individual</code>	The individual carrying this mutation (an object of class <code>Individual</code> )
<code>subpop</code>	The subpopulation in which that individual lives

These may be used in the `mutationEffect()` callback to compute a fitness value. To implement the standard fitness functions used by SLiM for an autosomal simulation with no null genomes involved, for example, you could do something like this:

```
mutationEffect(m1) {
  if (homozygous)
    return 1.0 + mut.selectionCoeff;
  else
    return 1.0 + mut.mutationType.dominanceCoeff * mut.selectionCoeff;
}
```

As mentioned above, a relative fitness of 1.0 is neutral (whereas a selection coefficient of 0.0 is neutral); the 1.0 + in these calculations converts between the selection coefficient scale and the relative fitness scale, and is therefore essential. However, the `effect` global variable mentioned above would already contain this value, precomputed by SLiM, so you could simply return `effect` to get that behavior when you want it:

```
mutationEffect(m1) {
  if (<conditions>)
    <custom fitness calculations...>;
  else
    return effect;
}
```

This would return a modified fitness value in certain conditions, but would return the standard fitness value otherwise.

More than one `mutationEffect()` callback may be defined to operate in the same tick. As with Eidos events, multiple callbacks will be called in the order in which they were defined in the input file. Furthermore, each callback will be given the `effect` value returned by the previous callback – so the value of `effect` is not necessarily the default value, in fact, but is the result of all previous `mutationEffect()` callbacks for that individual in that tick. In this way, the effects of multiple callbacks can “stack”.

One caveat to be aware of in WF models is that `mutationEffect()` callbacks are called at the end of the tick, just before the next tick begins. If you have a `mutationEffect()` callback defined for tick 10, for example, it will actually be called at the very end of tick 10, after child generation has finished, after the new children have been promoted to be the next parental generation, and after `late()` events have been executed (see chapter 22). The fitness values calculated will thus be used during tick 11; the fitness values used in tick 10 were calculated at the end of tick 9. (This is primarily so that SLiMgui, which refreshes its display in between ticks, has computed fitness values at hand that it can use to display the new parental individuals in the proper colors.) This is not an

issue in nonWF models, since fitness values are used in the same tick in which they are calculated (see chapter 23).

If the `randomizeCallbacks` parameter to `initializeSLiMOptions()` is T (the default), the order in which the fitness of individuals is evaluated will be randomized within each subpopulation. This partially mitigates order-dependency issues, although such issues can still arise whenever the effects of a `mutationEffect()` callback are not independent. If `randomizeCallbacks` is F, the fitness of individuals will be evaluated in sequential order within each subpopulation, greatly increasing the risk of order-dependency problems (see section 24.1).

Many other possibilities can be implemented with `mutationEffect()` callbacks; chapter 10 introduces `mutationEffect()` callbacks with several different recipes, and later chapters also use these techniques extensively. However, since `mutationEffect()` callbacks involve Eidos code being executed for the evaluation of fitness of every mutation of every individual (within the tick range, mutation type, and subpopulation specified), they can slow down a simulation considerably, so use them as sparingly as possible.

### 25.3 Defining individual-level fitness effects with a `fitnessEffect()` callback

The previous section introduced `mutationEffect()` callbacks, which modify the effect of a given mutation in a focal individual. Sometimes it is desirable to model effects upon individual fitness that are not governed by particular mutations (or not directly, at least); fitness effects due to spatial position, or resource acquisition, or behavior such as competitive or altruistic interactions, for example. Another situation of this type is when fitness depends upon the overall phenotype of an individual – the height of a tree, say – which might be influenced by genetics, but also by environmental effects, climate, and so forth. For these sorts of situations, SLiM provides `fitnessEffect()` callbacks, the topic of this section.

A `fitnessEffect()` callback is called by SLiM when it is determining the fitness of an individual – typically, but not always, once per tick during the fitness calculation tick cycle stage. Normally, the fitness of a given individual is determined by multiplying together the fitness effects of all mutations possessed by that individual (see section 25.2 for further discussion). Supplying a `fitnessEffect()` callback allows you to add another multiplicative fitness effect into that calculation. As with `mutationEffect()` callbacks, the value returned by `fitnessEffect()` callbacks is a fitness effect, so 1.0 is neutral.

The syntax for declaring `fitnessEffect()` callbacks is similar to that for `mutationEffect()` callbacks, but simpler since no mutation type is needed:

```
[id] [t1 [: t2]] fitnessEffect([<subpop-id>]) { ... }
```

(In multispecies models, the definition must be preceded by a species specification as usual.)

For example, if the callback were defined as:

```
1000:2000 fitnessEffect(p3) { 0.75; }
```

then a fitness effect of 0.75 would be multiplied into the fitness values of all individuals in subpopulation p3 from tick 1000 to tick 2000.

Much more interesting, of course, are `fitnessEffect()` callbacks that return different fitness effects for different individuals, depending upon their state! In addition to the SLiM globals listed in section 25.1, a `fitnessEffect()` callback is supplied with some additional information passed through “pseudo-parameters”, variables that are defined by SLiM within the context of the callback’s code to supply the callback with relevant information:

<code>individual</code>	The focal individual (an object of class <code>Individual</code> )
<code>subpop</code>	The subpopulation in which that individual lives

These may be used in the `fitnessEffect()` callback to compute a fitness effect that depends upon the state of the focal individual. The fitness effect for the callback is simply returned as a singleton `float` value, as usual.

More than one `fitnessEffect()` callback may be defined to operate in the same tick. Each such callback will provide an independent fitness effect for the focal individual; the results of each `fitnessEffect()` callback will be multiplied in to the individual's fitness. These callbacks will generally be called once per individual in each tick, in an order that is formally undefined, as described in detail in section 22.6.

Beginning in SLiM 3.0, it is also possible to set the `fitnessScaling` property on a subpopulation to scale the fitness values of every individual in the subpopulation by the same constant amount, or to set the `fitnessScaling` property on an individual to scale the fitness value of that specific individual. These scaling factors are multiplied together with all other fitness effects for an individual to produce the individual's final fitness value. The `fitnessScaling` properties of `Subpopulation` and `Individual` can often provide similar functionality to `fitnessEffect()` callbacks with greater efficiency and simplicity. They are reset to `1.0` in every tick for which a given species is active, immediately after fitness values are calculated, so they only need to be set when a value other than `1.0` is desired.

As with `mutationEffect()` callbacks, `fitnessEffect()` callbacks are called at the end of the tick, just before the next tick begins; see section 25.2 for further discussion of this potential pitfall. Also, as with `mutationEffect()` callbacks, the order in which `fitnessEffect()` callbacks are called will be shuffled when `randomizeCallbacks` is enabled, as it is by default, partially mitigating order-dependency issues (see sections 25.2 and 24.1).

The `fitnessEffect()` callback mechanism is quite flexible and useful, although it has been considerably eclipsed by the modern modern and efficient `fitnessScaling` property mentioned above. When efficiency is not at a premium, it remains a clear and expressive paradigm for modeling individual-level fitness effects. The performance penalty paid is often not large, since these callbacks are called only once per individual per tick, whereas a `mutationEffect()` for a type of mutation that is common in the simulation might be called thousands of times per individual per tick (once per mutation of that type possessed by the focal individual). The performance penalty typically becomes severe only when the `fitnessEffect()` callback needs to perform calculations, once per focal individual, that would vectorize well if performed across a whole vector of individuals. In such cases, `fitnessScaling` should be used.

## 25.4 Defining mate choice with a `mateChoice()` callback

Normally, WF models in SLiM regulate mate choice according to fitness; individuals of higher fitness are more likely to be chosen as mates. However, one might wish to simulate more complex mate-choice dynamics such as assortative or disassortative mating, mate search algorithms, and so forth. Such dynamics can be handled in WF models with the `mateChoice()` callback mechanism. (In nonWF models mating is arranged by the script, so there is no need for a callback; see section 1.6).

A `mateChoice()` callback is established in the input file with a syntax very similar to that of `fitnessEffect()` callbacks (section 25.3):

```
[id] [t1 [: t2]] mateChoice([<subpop-id>]) { ... }
```

(In multispecies models, the definition must be preceded by a species specification as usual.)

Note that if a subpopulation is given to which the `mateChoice()` callback is to apply, the callback is used for all matings that will generate a *child* in the stated subpopulation (as opposed to all matings of *parents* in the stated subpopulation); this distinction is important when migration causes children in one subpopulation to be generated by matings of parents in a different subpopulation.

When a `mateChoice()` callback is defined, the first parent in a mating is still chosen proportionally according to fitness (if you wish to influence that choice, you can use a `mutationEffect()` or `fitnessEffect()` callback; see sections 25.2 and 25.3). In a sexual (rather than hermaphroditic) simulation, this will be the female parent; SLiM does not currently support males as the choosy sex. The second parent – the male parent, in a sexual simulation – will then be chosen based upon the results of the `mateChoice()` callback.

More specifically, the callback must return a vector of weights, one for each individual in the subpopulation; SLiM will then choose a parent with probability proportional to weight. The `mateChoice()` callback could therefore modify or replace the standard fitness-based weights depending upon some other criterion such as assortativeness. A singleton object of type `Individual` may be returned instead of a weights vector to indicate that that specific individual has been chosen as the mate (beginning in SLiM 2.3); this could also be achieved by returning a vector of weights in which the chosen mate has a non-zero weight and all other weights are zero, but returning the chosen individual directly is much more efficient. A zero-length return vector – as generated by `float(0)`, for example – indicates that a suitable mate was not found; in that event, a new first parent will be drawn from the subpopulation. Finally, if the callback returns `NULL`, that signifies that SLiM should use the standard fitness-based weights to choose a mate; the `mateChoice()` callback did not wish to alter the standard behavior for the current mating (this is equivalent to returning the unmodified vector of weights, but returning `NULL` is much faster since it allows SLiM to drop into an optimized case). Apart from the special cases described above – a singleton `Individual`, `float(0)`, and `NULL` – the returned vector of weights must contain the same number of values as the size of the subpopulation, and all weights must be non-negative. Note that the vector of weights is not required to sum to 1, however; SLiM will convert relative weights on any scale to probabilities for you.

If the sum of the returned weights vector is zero, SLiM treats it as meaning the same thing as a return of `float(0)` – a suitable mate could not be found, and a new first parent will thus be drawn. (This is a change in policy beginning in SLiM 2.3; prior to that, returning a vector of sum zero was considered a runtime error.) There is a subtle difference in semantics between this and a return of `float(0)`: returning `float(0)` immediately short-circuits mate choice for the current first parent, whereas returning a vector of zeros allows further applicable `mateChoice()` callbacks to be called, one of which might “rescue” the first parent by returning a non-zero weights vector or an individual. In most models this distinction is irrelevant, since chaining `mateChoice()` callbacks is uncommon (see section 25.11). When the choice is otherwise unimportant, returning `float(0)` will be handled more quickly by SLiM.

In addition to the SLiM globals listed in section 25.1, a `mateChoice()` callback is supplied with some additional information passed through “pseudo-parameters”:

<code>individual</code>	The parent already chosen (the female, in sexual simulations)
<code>subpop</code>	The subpopulation into which the offspring will be placed
<code>sourceSubpop</code>	The subpopulation from which the parents are being chosen
<code>weights</code>	The standard fitness-based weights for all individuals

If sex is enabled, the `mateChoice()` callback must ensure that the appropriate weights are zero and nonzero to guarantee that all eligible mates are male (since the first parent chosen is always



female, as explained above). In other words, weights for females must be 0. The weights vector given to the callback is guaranteed to satisfy this constraint. If sex is not enabled – in a hermaphroditic simulation, in other words – this constraint does not apply.

For example, a simple `mateChoice()` callback might look like this:

```
1000:2000 mateChoice(p2) {  
    return weights ^ 2;  
}
```

This defines a `mateChoice()` callback for ticks 1000 to 2000 for subpopulation p2. The callback simply transforms the standard fitness-based probabilities by squaring them. Code like this could represent a situation in which fitness and mate choice proceed normally in one subpopulation (p1, here, presumably), but are altered by the effects of a social dominance hierarchy or male-male competition in another subpopulation (p2, here), such that the highest-fitness individuals tend to be chosen as mates more often than their (perhaps survival-based) fitness values would otherwise suggest. Note that by basing the returned weights on the weights vector supplied by SLiM, the requirement that females be given weights of 0 is finessed; in other situations, care would need to be taken to ensure that.

More than one `mateChoice()` callback may be defined to operate in the same tick. As with Eidos events, multiple callbacks will be called in the order in which they were defined. Furthermore, each callback will be given the weights vector returned by the previous callback – so the value of `weights` is not necessarily the default fitness-based weights, in fact, but is the result of all previous `weights()` callbacks for the current mate-choice event. In this way, the effects of multiple callbacks can “stack”. If any `mateChoice()` callback returns `float(0)`, however – indicating that no eligible mates exist, as described above – then the remainder of the callback chain will be short-circuited and a new first parent will immediately be chosen.

Note that matings in SLiM do not proceed in random order. Offspring are generated for each subpopulation in turn, and within each subpopulation the order of offspring generation is also non-random with respect to both the source subpopulation and the sex of the offspring. It is important, therefore, that `mateChoice()` callbacks are not in any way biased by the offspring generation order; they should not treat matings early in the process any differently than matings late in the process. Any failure to guarantee such invariance could lead to large biases in the simulation outcome. In particular, it is usually dangerous to activate or deactivate `mateChoice()` callbacks while offspring generation is in progress.

A wide variety of mate choice algorithms can easily be implemented with `mateChoice()` callbacks; chapter 11 introduces `mateChoice()` callbacks with several different recipes. However, `mateChoice()` callbacks can be particularly slow since they are called for every proposed mating, and the vector of mating weights can be large and slow to process.

## 25.5 Defining child generation with a `modifyChild()` callback

Normally, a SLiM simulation defines child generation with its rules regarding selfing versus crossing, recombination, mutation, and so forth. However, one might wish to modify these rules in particular circumstances – by preventing particular children from being generated, by modifying the generated children in particular ways, or by generating children oneself. All of these dynamics can be handled in SLiM with the `modifyChild()` callback mechanism.

A `modifyChild()` callback is established in the input file with a syntax very similar to that of other callbacks:

```
[id] [t1 [: t2]] modifyChild([<subpop-id>]) { ... }
```



The `modifyChild()` callback may optionally be restricted to the children generated to occupy a specified subpopulation. (In multispecies models, the definition must be preceded by a species specification as usual.)

When a `modifyChild()` callback is called, a parent or parents have already been chosen, and a candidate child has already been generated. The parent or parents, and their genomes, are provided to the callback, as is the generated child and its genomes. The callback may accept the generated child, modify it, substitute completely different genomic information for it, or reject it (causing a new parent or parents to be selected and a new child to be generated, which will again be passed to the callback).

In addition to the SLiM globals listed in section 25.1, a `modifyChild()` callback is supplied with additional information passed through “pseudo-parameters”:

<code>child</code>	The generated child (an object of class <code>Individual</code> )
<code>parent1</code>	The first parent (an object of class <code>Individual</code> )
<code>isCloning</code>	T if the child is the result of cloning
<code>isSelfing</code>	T if the child is the result of selfing (but see note below)
<code>parent2</code>	The second parent (an object of class <code>Individual</code> )
<code>subpop</code>	The subpopulation in which the child will live
<code>sourceSubpop</code>	The subpopulation of the parents (== <code>subpop</code> if not a migration mating)

These may be used in the `modifyChild()` callback to decide upon a course of action. The genomes of `child` (available as `child.genome1` and `child.genome2`) may be modified by the callback; whatever mutations they contain on exit will be used for the new child. Alternatively, they may be left unmodified (to accept the generated child as is). These variables may be thought of as the two gametes that will fuse to produce the fertilized egg that results in a new offspring; `child.genome1` is the gamete contributed by the first parent (the female, if sex is turned on), and `child.genome2` is the gamete contributed by the second parent (the male, if sex is turned on). The `child` object itself may also be modified – for example, to set the spatial position of the child.

Importantly, a `logical` singleton return value is required from `modifyChild()` callbacks. Normally this should be T, indicating that generation of the child may proceed (with whatever modifications might have been made to the child’s genomes). A return value of F indicates that generation of this child should not continue; this will cause new parent(s) to be drawn, a new child to be generated, and a new call to the `modifyChild()` callback. A `modifyChild()` callback that always returns F can cause SLiM to hang, so be careful that it is guaranteed that your callback has a nonzero probability of returning T for every state your simulation can reach.

Note that `isSelfing` is T only when a mating was explicitly set up to be a selfing event by SLiM; an individual may also mate with itself by chance (by drawing itself as a mate) even when SLiM did not explicitly set up a selfing event, which one might term *incidental* selfing. If you need to know whether a mating event was an incidental selfing event, you can compare the parents; self-fertilization will always entail `parent1==parent2`, even when `isSelfing` is F. See the recipe in section 12.4 for an example of how to use this to suppress incidental selfing. Since selfing is enabled only in non-sexual simulations, `isSelfing` will always be F in sexual simulations (and incidental selfing is also impossible in sexual simulations).

Note that matings in SLiM do not proceed in random order. Offspring are generated for each subpopulation in turn, and within each subpopulation the order of offspring generation is also non-random with respect to the source subpopulation, the sex of the offspring, and the reproductive mode (selfing, cloning, or autogamy). It is important, therefore, that `modifyChild()` callbacks are not in any way biased by the offspring generation order; they should not treat offspring generated early in the process any differently than offspring generated late in the process.

Similar to `mateChoice()` callbacks, any failure to guarantee such invariance could lead to large biases in the simulation outcome. In particular, it is usually dangerous to activate or deactivate `modifyChild()` callbacks while offspring generation is in progress. When SLiM sees that `mateChoice()` or `modifyChild()` callbacks are defined, it randomizes the order of child generation within each subpopulation, so this issue is mitigated somewhat. However, offspring are still generated for each subpopulation in turn. Furthermore, in ticks without active callbacks offspring generation order will not be randomized (making the order of parents nonrandom in the next generation), with possible side effects. In short, order-dependency issues are possible and must be handled very carefully.

As with the other callback types, multiple `modifyChild()` callbacks may be registered and active. In this case, all registered and active callbacks will be called for each child generated, in the order that the callbacks were registered. If a `modifyChild()` callback returns F, however, indicating that the child should not be generated, the remaining callbacks in the chain will not be called.

There are many different ways in which a `modifyChild()` callback could be used in a simulation; see the recipes in chapter 12 for illustrations of the power of this technique. In nonWF models, `modifyChild()` callbacks are often unnecessary since each generated child is available to the script in the models' `reproduction()` callback anyway; but they may be used if desired.

## 25.6 Defining recombination behavior with a `recombination()` callback

Typically, a simulation sets up a recombination map at the beginning of the run with `initializeRecombinationRate()`, and that map is used for the duration of the run. Less commonly, the recombination map is changed dynamically from tick to tick, with `Chromosome`'s method `setRecombinationRate()`; but still, a single recombination map applies for all individuals of a species in a given tick. However, in unusual circumstances a simulation may need to modify the way that recombination works on an individual basis; for this, the `recombination()` callback mechanism is provided. This can be useful for models involving chromosomal inversions that prevent recombination within a region for some individuals (see section 14.4), for example, or for models of the evolution of recombination.

A `recombination()` callback is defined with a syntax much like that of other callbacks:

```
[id] [t1 [: t2]] recombination([<subpop-id>]) { ... }
```

The `recombination()` callback will be called during the generation of every gamete during the tick(s) in which it is active. It may optionally be restricted to apply only to gametes generated by parents in a specified subpopulation, using the `<subpop-id>` specifier. (In multispecies models, the definition must be preceded by a species specification as usual.)

When a `recombination()` callback is called, a parent has already been chosen to generate a gamete, and candidate recombination breakpoints for use in recombining the parental genomes have been drawn. The genomes of the focal parent are provided to the callback, as is the focal parent itself (as an `Individual` object) and the subpopulation in which it resides. Furthermore, the proposed breakpoints are provided to the callback. The callback may modify these breakpoints in order to change the breakpoints used, in which case it must return T to indicate that changes were made, or it may leave the proposed breakpoints unmodified, in which case it must return F. (The behavior of SLiM is undefined if the callback returns the wrong logical value.)

In addition to the SLiM globals listed in section 25.1, then, a `recombination()` callback is supplied with additional information passed through “pseudo-parameters”:

`individual`     The focal parent that is generating a gamete

<code>genome1</code>	One genome of the focal parent; this is the initial copy strand
<code>genome2</code>	The other genome of the focal parent
<code>subpop</code>	The subpopulation to which the focal parent belongs
<code>breakpoints</code>	An integer vector of crossover breakpoints

These may be used in the `recombination()` callback to determine the final recombination breakpoints used by SLiM. If values are set into `breakpoints`, the new values must be of type integer. If `breakpoints` is modified by the callback, `T` should be returned, otherwise `F` should be returned (this is a speed optimization, so that SLiM does not have to spend time checking for changes when no changes have been made).

The positions specified in `breakpoints` mean that a crossover will occur immediately *before* the specified base position (between the preceding base and the specified base, in other words). The genome specified by `genome1` will be used as the initial copy strand when SLiM executes the recombination; this cannot presently be changed by the callback. (Note that `genome1` and `genome2` will be the same objects as `individual.genome1` and `individual.genome2`, but may be swapped, if `individual.genome2` is the initial copy strand!)

In this design, the recombination callback does not specify a custom recombination map. Instead, the callback can add or remove breakpoints at specific locations. To implement a chromosomal inversion, as is done in the recipe in section 14.4, for example, if the parent is heterozygous for the inversion mutation then crossovers within the inversion region are removed by the callback. As another example, to implement a model of the evolution of the overall recombination rate, a model could (1) set the global recombination rate to the highest rate attainable in the simulation, (2) for each individual, within the `recombination()` callback, calculate the fraction of that maximum rate that the focal individual would experience based upon its genetics, and (3) probabilistically remove proposed crossover points based upon random uniform draws compared to that threshold fraction, thus achieving the individual effective recombination rate desired. Other similar treatments could actually vary the effective recombination map, not just the overall rate, by removing proposed crossovers with probabilities that depend upon their position, allowing for the evolution of localized recombination hot-spots and cold-spots. Crossovers may also be added, not just removed, by `recombination()` callbacks.

In SLiM 3.3 the recombination model in SLiM was redesigned (see section 1.5.6 for discussion). This required a corresponding redesign of `recombination()` callbacks. In particular, the `gcStarts` and `gcEnds` pseudo-parameters to `recombination()` callbacks were removed. In the present design, the callback receives “crossover breakpoints” information only, in the `breakpoints` pseudo-parameter; it receives no information about gene conversion. However, `recombination()` callbacks can still be used with the “DSB” recombination model; at the point when the callback is called, the pattern of gene conversion tracts will have been simplified down to a vector of crossover breakpoints. “Complex” gene conversion tracts, however, involving heteroduplex mismatch repair, are not compatible with `recombination()` callbacks, since there is presently no way for them to be specified to the callback.

Note that the positions in `breakpoints` are not, in the general case, guaranteed to be sorted or unique; in other words, positions may appear out of order, and the same position may appear more than once. After all `recombination()` callbacks have completed, the positions from `breakpoints` will be sorted, unique, and used as the crossover points in generating the prospective gamete genome. The essential point here is that if the same position occurs more than once, across breakpoints, the multiple occurrences of the position do not cancel; SLiM does not cross over and then “cross back over” given a pair of identical positions. Instead, the multiple occurrences of the position will simply be unique down to a single occurrence.

As with the other callback types, multiple `recombination()` callbacks may be registered and active. In this case, all registered and active callbacks will be called for each gamete generated, in the order that the callbacks were registered.

## 25.7 Defining interaction behavior with an `interaction()` callback

The `InteractionType` class (section 24.8) provides various built-in interaction functions that translate from distances to interaction strengths. However, it may sometimes be useful to define a custom function for that purpose; for that reason, SLiM allows `interaction()` callbacks to be defined that modify the standard interaction strength calculated by `InteractionType`. In particular, this mechanism allows the strength of interactions to depend upon not only the distance between individuals, but also the genetics and other state of the individuals, the spatial position of the individuals, and other environmental variables.

An `interaction()` callback is called by SLiM when it is determining the strength of the interaction between one individual (the *receiver* of the interaction) and another individual (the *exerter* of the interaction). This generally occurs when an interaction query is made to `InteractionType`, as a side effect of serving that query. This means that `interaction()` callbacks may be called at a variety of points in the tick cycle, unlike the other callback types in SLiM, which are each called at a specific point. If you write an `interaction()` callback, you need to take this into account; assuming that the tick cycle is at a particular stage, or even that the tick or cycle is the same as it was when `evaluate()` was called, may be dangerous.

When an interaction strength is needed, the first thing SLiM does is calculate the default interaction strength using the interaction function that has been defined for the `InteractionType` (see section 24.8). If the receiver is the same as the exerter, the interaction strength is always zero; and in spatial simulations if the distance between the receiver and the exerter is greater than the maximum distance set for the `InteractionType`, the interaction strength is also always zero. In these cases, `interaction()` callbacks will not be called, and there is no way to redefine these interaction strengths.

Otherwise, SLiM will then call `interaction()` callbacks that apply to the interaction type and exerter subpopulation for the interaction being evaluated. An `interaction()` callback is defined with a variation of the syntax used for other callbacks:

```
[id] [t1 [: t2]] interaction(<int-type-id> [, <subpop-id>]) { ... }
```

For example, if the callback were defined as:

```
1000:2000 interaction(i2, p3) { 1.0; }
```

then an interaction strength of `1.0` would be used for all interactions of interaction type `i2`, for exerters in subpopulation `p3`, from tick `1000` to tick `2000`.

Beginning in SLiM 4, the receiver and exerter may be in different subpopulations from each other – or even, in multispecies models, in different species altogether. For the subpopulation `id` in the `interaction()` callback declaration, it does not matter which subpopulation the receiver is in; if the exerter is in `p3`, for the above example, then the `interaction()` callback will be called regardless of the receiver's subpopulation (assuming other preconditions are also met, such as the tick range and the interaction type `id`). This means that `interaction()` callbacks are not species-specific, unlike other callback types; even if an `interaction()` callback is declared to be specific to exerters in `p3`, as above, receivers can still be in a different species. With no subpopulation `id` specified, `interaction()` callbacks are even more general: the `InteractionType` can then be evaluated and queried for receivers and exerters belonging to any species. For this reason, in multispecies models `interaction()` callbacks must be declared using a species specifier of

species all, unlike all other SLiM callback types; it is not legal to declare an `interaction()` callback as species-specific. Note that there is no way to declare an `interaction()` callback as applying only to receivers in a given subpopulation; if that functionality is desired, you can test `receiver.subpopulation` in your callback code and act accordingly.

In addition to the SLiM globals listed in section 25.1, an `interaction()` callback is supplied with some additional information passed through “pseudo-parameters”:

<code>distance</code>	The distance from receiver to exorter, in spatial simulations; <code>NAN</code> otherwise
<code>strength</code>	The default interaction strength calculated by the interaction function
<code>receiver</code>	The individual receiving the interaction (an object of class <code>Individual</code> )
<code>exorter</code>	The individual exerting the interaction (an object of class <code>Individual</code> )

These may be used in the `interaction()` callback to compute an interaction strength. To simply use the default interaction strength that SLiM would use if a callback had not been defined for interaction type `i1`, for example, you could do this:

```
interaction(i1) {
    return strength;
}
```

Usually an `interaction()` callback will modify that default strength based upon factors such as the genetics of the receiver and/or the exorter, the spatial positions of the two individuals, or some other simulation state. Any finite `float` value greater than or equal to `0.0` may be returned. The value returned will not be cached by SLiM; if the interaction strength between the same two individuals is needed again later, the `interaction()` callback will be called again (something to keep in mind if the interaction strength includes a stochastic component). Note that the provided `distance` and `strength` values are based upon the spatial positions of the exorter and receiver when `evaluate()` was called, not their current spatial positions, if they have moved since the interaction was evaluated.

More than one `interaction()` callback may be defined to operate in the same tick. As with other callbacks, multiple callbacks will be called in the order in which they were defined in the input file. Furthermore, each callback will be given the `strength` value returned by the previous callback – so the value of `strength` is not necessarily the default value, in fact, but is the result of all previous `interaction()` callbacks for the interaction in question. In this way, the effects of multiple callbacks can “stack”.

The `interaction()` callback mechanism is extremely powerful and flexible, allowing any sort of user-defined interactions whatsoever to be queried dynamically using the methods of `InteractionType`. However, in the general case a simulation may call for the evaluation of the interaction strength between each individual and every other individual, making the computation of the full interaction network an  $O(N^2)$  problem. Since `interaction()` callbacks may be called for each of those  $N^2$  interaction evaluations, they can slow down a simulation considerably, so it is recommended that they be used sparingly. This is the reason that the various interaction functions of `InteractionType` were provided; when an interaction does not depend upon individual state, the intention is to avoid the necessity of an `interaction()` callback altogether. Furthermore, constraining the number of cases in which interaction strengths need to be calculated – using a short maximum interaction distance, querying the nearest neighbors of the focal individual rather than querying all possible interactions with that individual, and specifying the reciprocity and sex segregation of the `InteractionType`, for example – may greatly decrease the computational overhead of interaction evaluation.



## 25.8 Defining reproduction behavior with a reproduction() callback

In WF models (the default model type in SLiM), the SLiM core manages the reproduction of individuals in each tick. In nonWF models, however, reproduction is managed by the model script, in reproduction() callbacks. These callbacks may only be defined in nonWF models.

A reproduction() callback is defined with a syntax much like that of other callbacks:

```
[id] [t1 [: t2]] reproduction(<subpop-id> [, <sex>]) { ... }
```

The reproduction() callback will be called once for each individual during the tick(s) in which it is active. It may optionally be restricted to apply only to individuals in a specified subpopulation, using the <subpop-id> specifier; this may be a subpopulation specifier such as p1, or NULL indicating no restriction. It may also optionally be restricted to apply only to individuals of a specified sex (in sexual models), using the <sex> specifier; this may be "M" or "F", or NULL indicating no restriction. (In multispecies models, the definition must be preceded by a species specification as usual.)

When a reproduction() callback is called, SLiM's expectation is that the callback will trigger the reproduction of a focal individual by making method calls to add new offspring individuals. Typically the offspring added are the offspring of the focal individual, and typically they are added to the subpopulation to which the focal individual belongs, but neither of these is required; a reproduction() callback may add offspring generated by any parent(s), to any subpopulation in the focal species. The focal individual is provided to the callback (as an Individual object), as are its genomes and the subpopulation in which it resides.

A common alternative pattern is for a reproduction() callback to ignore the focal individual and generate all of the offspring for a species for the current tick, from all parents. The callback then sets self.active to 0, preventing itself from being called again in the current tick (see section 25.11); this callback design therefore executes once per tick. This can be useful if individuals influence each other's offspring generation (as in a monogamous-mating model, for example); it can also simply be more efficient when producing offspring in bulk.

In addition to the SLiM globals listed in section 25.1, then, a reproduction() callback is supplied with additional information passed through global variables:

individual	The focal individual that is expected to reproduce
subpop	The subpopulation to which the focal individual belongs

At present, the return value from reproduction() callbacks is not used, and must be void (i.e., a value may not be returned). It is possible that other return values will be defined in future.

It is possible, of course, to do actions unrelated to reproduction inside reproduction() callbacks, but it is not recommended. The first() event phase of the current tick provides an opportunity for actions immediately before reproduction, and the early() event phase of the current tick provides an opportunity for actions immediately after reproduction, so only actions that are intertwined with reproduction itself should occur in reproduction() callbacks. Besides providing conceptual clarity, following this design principle will also decrease the probability of bugs, since actions that are unrelated to reproduction should usually not influence or be influenced by the dynamics of reproduction.

If the randomizeCallbacks parameter to initializeSLiMOptions() is T (the default), the order in which individuals are given an opportunity to reproduce with a call to reproduction() callbacks will be randomized within each subpopulation. This partially mitigates order-dependency issues, although such issues can still arise whenever the effects of a reproduction() callback are not independent. If randomizeCallbacks is F, individuals will be given their opportunity to reproduce



in sequential order within each subpopulation, greatly increasing the risk of order-dependency problems (see section 24.1).

As with the other callback types, multiple `reproduction()` callbacks may be registered and active. In this case, all registered and active callbacks will be called for each individual, in the order that the callbacks were registered.

## 25.9 Defining mutation behavior with a `mutation()` callback

SLiM auto-generates new mutations according to the current mutation rate (or rate map) and the genetic structure defined by genomic elements, their genomic element types, the mutation types those genomic element types draw from, and the distribution of fitness effects defined by those mutation types (see section 1.5.4). In nucleotide-based models, the nucleotide sequence and the mutation matrix also play a role in determining both the rate of mutation and the nucleotide mutated to (see section 1.8). In some models it can be desirable to modify these dynamics in some way – altering the selection coefficients of new mutations in some way, changing the mutation type used, dictating the nucleotide to be used, replacing the proposed mutation with a pre-existing mutation at the same position, or even suppressing the proposed mutation altogether. To achieve this, one may define a `mutation()` callback.

A `mutation()` callback is defined as:

```
[id] [t1 [: t2]] mutation([<mut-type-id> [, <subpop-id>]]) { ... }
```

The `mutation()` callback will be called once for each new auto-generated mutation during the tick(s) in which the callback is active. It may optionally be restricted to apply only to mutations of a particular mutation type, using the `<mut-type-id>` specifier; this may be a mutation type specifier such as `m1`, or `NULL` indicating no restriction. It may also optionally be restricted to individuals generated by a specified subpopulation (usually – see below for discussion), using the `<subpop-id>` specifier; this should be a subpopulation specifier such as `p1`. (In multispecies models, the definition must be preceded by a species specification as usual.)

When a `mutation()` callback is called, a focal mutation (provided to the callback as an object of type `Mutation`) has just been created by SLiM, referencing a particular position in a parental genome (also provided, as an object of type `Genome`). The mutation will not be added to that parental genome; rather, the parental genome is being copied, during reproduction, to make a gamete or an offspring genome, and the mutation is, conceptually, a copying error made during that process. It will be added to the offspring genome that is the end result of the copying process (which may also involve recombination with another genome). At the point that the `mutation()` callback is called, the offspring genome is not yet created, however, and so it cannot be accessed from within the `mutation()` callback; the `mutation()` callback can affect only the mutation itself, not the genome to which the mutation will be added.

In addition to the SLiM globals listed in section 25.1, then, a `mutation()` callback is supplied with additional information passed through global variables:

<code>mut</code>	The focal mutation that is being modified or reviewed
<code>genome</code>	The parental genome that is being copied
<code>element</code>	The genomic element that controls the mutation site
<code>originalNuc</code>	The nucleotide (0/1/2/3 for A/C/G/T) originally at the mutating position
<code>parent</code>	The parent which is generating the offspring genome
<code>subpop</code>	The subpopulation to which the parent belongs

The `mutation()` callback has three possible returns: `T`, `F`, or (beginning in SLiM 3.5) a singleton object of type `Mutation`. A return of `T` indicates that the proposed mutation should be used in

generating the offspring genome (perhaps with modifications made by the callback). Conversely, a return of `F` indicates that the proposed mutation should be suppressed. If a proposed mutation is suppressed, SLiM will not try again; one fewer mutations will be generated during reproduction than would otherwise have been true. Returning `F` will therefore mean that the realized mutation rate in the model will be lower than the expected mutation rate. Finally, a return of an object of type `Mutation` replaces the proposed mutation (`mut`) with the mutation returned; the offspring genomes being generated will contain the returned mutation. The position of the returned mutation must match that of the proposed mutation. This provides a mechanism for a `mutation()` callback to make SLiM re-use existing mutations instead of generating new mutations, which can be useful.

The callback may perform a variety of actions related to the generated mutation. The selection coefficient of the mutation can be changed with `setSelectionCoefficient()`, and the mutation type of the mutation can be changed with `setMutationType()`; the `drawSelectionCoefficient()` method of `MutationType` may also be useful here. A `tag` property value may be set for the mutation, and named values may be attached to the mutation with `setValue()`. In nucleotide-based models, the `nucleotide` (or `nucleotideValue`) property of the mutation may also be changed; note that the original nucleotide at the focal position in the parental genome is provided through `originalNuc` (it could be retrieved with `genome.nucleotides()`, but SLiM already has it at hand anyway). All of these modifications to the new mutation may be based upon the state of the parent, including its genetic state, or upon any other model state.

It is possible, of course, to do actions unrelated to mutation inside `mutation()` callbacks, but it is not recommended; `first()`, `early()`, and `late()` events should be used for general-purpose scripting. Besides providing conceptual clarity, following this design principle will also decrease the probability of bugs, since actions that are unrelated to mutation should not influence or be influenced by the dynamics of mutation.

The proposed mutation will not appear in the `sim.mutations` vector of segregating mutations until it has been added to a genome; it will therefore not be visible in that vector within its own `mutation()` callback invocation, and indeed, may not be visible in subsequent callbacks during the reproduction tick cycle stage until such time as the offspring individual being generated has been completed. If that offspring is ultimately rejected, in particular by a `modifyChild()` callback, the proposed mutation may not be used by SLiM at all. It may therefore be unwise to assume, in a `mutation()` callback, that the focal mutation will ultimately be added to the simulation, depending upon the rest of the model's script.

There is one subtlety to be mentioned here, having to do with subpopulations. The `subpop` pseudo-parameter discussed above is always the subpopulation of the parent which possesses the genome that is being copied and is mutating; there is no ambiguity about that whatsoever. The `<subpop-id>` specified in the `mutation()` callback declaration, however, is a bit more subtle; above it was said that it restricts the callback "to individuals generated by a specified subpopulation", and that is usually true but requires some explanation. In WF models, recall that migrants are generated in a source subpopulation and placed in a target subpopulation, as a model of juvenile migration (see section 22.2.1); in that context, the `<subpop-id>` specifies the *source* subpopulation to which the `mutation()` callback will be restricted. In nonWF models, offspring are generated by the `add...` family of `Subpopulation` methods (see section 24.15.2), which can cross individuals from two different subpopulations and place the result in a third target subpopulation; in that context, in general, the `<subpop-id>` specifies the source subpopulation that is generating the particular *gamete* that is sustaining a mutation during its production. The exception to this rule is `addRecombinant()`; since there are four different source subpopulations potentially in play there, it was deemed simpler in that case for the `<subpop-id>` to specify the *target* subpopulation to which the `mutation()` callback will be restricted. If restriction to the source subpopulation is needed

with `addRecombinant()`, the `subpop` pseudo-parameter may be consulted rather than using `<subpop-id>`.

Note that `mutation()` callbacks are only called for mutations that are auto-generated by SLiM, as a consequence of the mutation rate and the genetic structure defined (see section 1.5.4). Mutations that are created in script, using `addNewMutation()` or `addNewDrawnMutation()`, will not trigger `mutation()` callbacks; but of course the script may modify or tailor such added mutations in whatever way is desired, so there is no need for callbacks in that situation.

As with the other callback types, multiple `mutation()` callbacks may be registered and active. In this case, all registered and active callbacks will be called for each generated mutation to which they apply, in the order that the callbacks were registered.

## 25.10 Defining survival and mortality with a `survival()` callback

In nonWF models, a selection phase in the tick cycle results in mortality; individuals survive or die based upon their fitness (see section 23.4). In most cases this standard behavior is sufficient; but occasionally it can be useful to observe the survival decisions SLiM makes (to log out information about dying individuals, for example), to modify those decisions (influencing which individuals live and which die, perhaps based upon factors other than genetics), or even to short-circuit mortality completely (moving dead individuals into a “cold storage” subpopulation for later use, perhaps). To accomplish such goals, one can use the `survival()` callback mechanism to override SLiM’s default behavior. Note that in WF models, since they always model non-overlapping generations, the entire parental generation dies in each tick regardless of fitness; `survival()` callbacks therefore apply only to nonWF models.

A `survival()` callback is defined with a syntax much like that of other callbacks:

```
[id] [t1 [: t2]] survival([<subpop-id>]) { ... }
```

The `survival()` callback will be called during the selection phase of the tick cycle of nonWF models, during the tick(s) in which it is active. By default it will be called once per individual in the entire population (whether slated for survival or not); it may optionally be restricted to apply only to individuals in a specified subpopulation, using the `<subpop-id>` specifier. (In multispecies models, the definition must be preceded by a species specification as usual.)

When a `survival()` callback is called, a focal individual has already been evaluated by SLiM regarding its survival; a final fitness value for the individual has been calculated, and a random uniform draw in `[0,1]` has been generated that determines whether the individual is to survive (a draw less than the individual’s fitness) or die (a draw greater than or equal to the individual’s fitness). The focal individual is provided to the callback, as is the subpopulation in which it resides. Furthermore, the preliminary decision (whether the focal individual will survive or not), the focal individual’s fitness, and the random draw made by SLiM to determine survival are also provided to the callback. The callback may return `NULL` to accept SLiM’s decision, or may return `T` to indicate that the individual should survive, or `F` to indicate that it should die, regardless of its fitness and the random deviate drawn. The callback may also return a singleton `Subpopulation` object to indicate the individual should remain alive but should be moved to that subpopulation (note that calling `takeMigrants()` during the survival phase is illegal, because SLiM is busy modifying the population’s internal state).

In addition to the SLiM globals listed in section 25.1, then, a `survival()` callback is supplied with additional information passed through “pseudo-parameters”:

<code>individual</code>	The focal individual that will live or die
<code>subpop</code>	The subpopulation to which the focal individual belongs

<code>surviving</code>	A logical value indicating SLiM's preliminary decision ( <code>T == survival</code> )
<code>fitness</code>	The focal individual's fitness
<code>draw</code>	SLiM's random uniform deviate, which determined the preliminary decision

These may be used in the `survival()` callback to determine the final decision.

While `survival()` callbacks are still being called, no decisions are put into effect; no individuals actually die, and none are moved to a new Subpopulation if that was requested. In effect, SLiM pre-plans the fate of every individual completely without modifying the model state at all. After all `survival()` callbacks have completed for every individual, the planned fates for every individual will then be executed, without any opportunity for further intervention through callbacks. It is therefore legal to inspect subpopulations and individuals inside a `survival()` callback, but it should be understood that previously made decisions about the fates of other individuals will not yet have any visible effect. It is generally a good idea for the decisions rendered by `survival()` callbacks to be independent anyway, to avoid biases due to order-dependency. If the `randomizeCallbacks` parameter to `initializeSLiMOptions()` is `T` (the default), the order in which `survival()` callbacks are called on individuals will be randomized within each subpopulation; nevertheless, order-dependency issues can occur if callback effects are not independent. If `randomizeCallbacks` is `F`, the order in which individuals are evaluated within each subpopulation is not guaranteed to be random, and order-dependency problems are thus even more likely; see section 24.1.

It is worth noting that if `survival()` callbacks are used, “fitness” in the model is then no longer really fitness; the model is making its own decisions about which individuals live and die, and those decisions are the true determinant of fitness in the biological sense. A `survival()` callback that makes its own decisions regarding survival with no regard for SLiM's calculated fitness values can completely alter the pattern of selection in a population, rendering all of SLiM's fitness machinery – selection and dominance coefficients, `fitnessScaling` values, etc. – completely irrelevant. To avoid highly counterintuitive and confusing effects, it is thus generally a good idea to use of `survival()` callbacks only when it is strictly necessary to achieve a desired outcome.

As with the other callback types, multiple `survival()` callbacks may be registered and active. In this case, all registered and active callbacks will be called for each individual evaluated, in the order that the callbacks were registered.

## 25.11 Further details on Eidos events and callbacks

Section 25.1 described Eidos events, and sections 25.2 – 25.10 described several different Eidos callbacks that can be defined to modify the standard behavior of SLiM. This section describes a few additional details that apply to events and callbacks. These details were mentioned previously, but were not detailed, in the interests of simplicity; they are of interest mainly to the most advanced users of SLiM.

Every Eidos block – an event or a callback – is defined in SLiM using a class called `SLiMEidosBlock`. All of the registered instances of this class – all of the Eidos blocks scheduled to run in the simulation – are available through the `allScriptBlocks` property of `Community` (section 24.3.2), as well as the `scriptBlocks` property of `Species` for species-specific callbacks. New script blocks may be added programmatically (rather than in the SLiM input file) using the `Community` and `Species` `register...()` methods; those methods take a string parameter, which is interpreted as Eidos code. Existing script blocks may be deregistered, which removes them from the current simulation permanently, using the `deregisterScriptBlock()` method of `Community`. Similarly, script blocks may be rescheduled to an arbitrary tick or set of ticks using the `rescheduleScriptBlock()` method of `Community`. In this way, the script blocks defined in the SLiM



input file are only the beginning; by adding, removing, and rescheduling script blocks dynamically, SLiM simulations can modify their own code as they run. Obviously this feature would, if used indiscriminately, result in incomprehensible and unmaintainable code; but in some circumstances, it can be extremely useful and powerful.

Code that manipulates `SLiMEidosBlock` objects can find the operand blocks using their defined global symbols such as `s1` or `s13` (see section 25.1), or using the `id` property of `SLiMEidosBlock`. Alternatively, a script block that references itself (to deregister itself, for example, or to set its own active property) can use a global variable called `self` that is defined whenever an Eidos block is executing. The `self` variable refers to the executing `SLiMEidosBlock` object. It may be passed to `deregisterScriptBlock()` in order to deregister the current block; this is safe to do, as the executing block will not actually be deregistered until it has finished executing. It may also be used to change the properties of the currently executing script block.

In particular, `SLiMEidosBlock` defines an integer property, `active`. The `active` property is normally `-1`; this means that script blocks are normally active. If set to `0`, the script block will be inactive for the remainder of the current tick; it will not be called or used in any way (except that if it is currently executing when `active` is set to `0`, that execution will complete). At the beginning of each tick, prior to the execution of any script blocks, the `active` flag of all registered script blocks will be set back to `-1`, activating them all again; if you want a script block to be inactive permanently, you must deregister it rather than just marking it as inactive. (In multispecies models, script blocks will be set to be inactive at the start of a tick if they have an associated species, because of a species or ticks specifier, and that species is inactive in the current tick. Script blocks associated with a given species will also be made inactive if the species is made inactive with the `skipTick()` method.)

Values other than `-1` may be used for `active`; any value other than `0` indicates that the block is active (because `active` is evaluated as a logical value; only `0` is F). This facility is provided to allow script blocks to run a limited number of times in each tick; the block can check whether `active` is `-1` (indicating that it is being called for the first time in a tick), and can set `active` to a counter value. In each call to the script block, the script can decrement the `active` counter by `1`; when it reaches `0`, the block will not be called again in that tick. The `active` property could even be used to implement a more complex state machine, or it could be used as an arbitrary tag value.

The precise way in which SLiM handles the scheduling of `SLiMEidosBlock` objects may be important for some scripts. Because new script blocks can be added dynamically with `register...()`, and existing blocks can be removed with `deregisterScriptBlock()`, the right way to schedule blocks is not entirely clear. If SLiM is partway through generating children, and then a new `modifyChild()` callback is added, for example, should that callback be used for the remaining children generated in the current tick? What if an existing `modifyChild()` callback is removed, partway through the process of child generation – should that callback stop being used immediately? For consistency, SLiM's answer to both of these questions is “no”; a consistent set of scripts are used across each tick cycle stage of each tick. However, if a `modifyChild()` callback is added before generating children begins, then that callback is used in the same tick. In essence, the rule is this: whenever SLiM starts on a new stage of the tick cycle that involves calling a particular kind of Eidos block, SLiM gathers up a list of all of the currently defined script blocks applicable to that stage, and it uses that list throughout the duration of that stage, regardless of what changes are made to the registered script blocks during the stage. The state of the `active` property of each script block is checked immediately before each time that the script block is called, however; the `active` property is specifically intended to change the active status of a script block within a single tick.

## 26. SLiM output formats

In addition to allowing custom output in any format whatsoever, produced with Eidos code, SLiM also has numerous ways to produce output in fixed formats using built-in methods:

- Methods on `Species` (section 24.14.2): `outputFull()`, `outputFixedMutations()`, and `outputMutations()`.
- Methods on `Subpopulation` (section 24.15.2): `outputSample()`, `outputMSSample()`, and `outputVCFSample()`.
- Methods on `Genome` (section 24.4.2): `output()`, `outputMS()`, and `outputVCF()`.

The documentation cited for these classes above summarizes the method calls themselves, but does not document the precise format they produce; that will be covered in this chapter.

Note that these methods, and the format of the output produced by SLiM, changed in various ways in SLiM version 2.1. This documentation will discuss only the format of output for SLiM 2.1 and later, for simplicity.

As will be shown below, all of these output methods can generate a header line beginning with the tag `#OUT:` followed by (1) the tick in which the output was generated, (2) the cycle in which the output was generated, (3) a one- or two-letter output type code, (4) additional values depending upon the output type, and (5) if output was directed to a file, the filename to which output was directed (except in the case of the `Species` method `outputMutations()`). The output code in the header line may be used to detect which type of output follows, which is useful for automated parsing of simulation output files. The codes are as follows:

Species methods:

<code>outputFull()</code>	A
<code>outputFixedMutations()</code>	F
<code>outputMutations()</code>	T

Subpopulation methods:

<code>outputSample()</code>	SS
<code>outputMSSample()</code>	SM
<code>outputVCFSample()</code>	SV

Genome methods:

<code>output()</code>	GS
<code>outputMS()</code>	GM
<code>outputVCF()</code>	GV

All of these methods support output to either the SLiM output stream or to a designated file. When output is sent to a file, all of these methods support either overwriting an existing file at the specified path, or appending to any existing file.

These output methods are some of the more complex methods in SLiM, often with many optional arguments that are typically specified by name. See the Eidos manual for discussion of how to use optional arguments and named arguments, how to interpret complex type-specifiers and method signatures, and so forth; that information is not repeated here.

### 26.1 Species output methods

The output methods of `Species` produce output regarding state that spans the whole population of the species, rather than just a single subpopulation or a selected set of genomes.



### 26.1.1 outputFull()

The `outputFull()` method outputs complete information on all subpopulations, individuals, and genomes, including all currently segregating mutations (but not including mutations that have fixed and been converted into Substitution objects). Sample output for `outputFull()`, abbreviated with ellipses:

```
#OUT: 10000 10000 A
Version: 3
Populations:
p1 50 H
p2 50 H
...
Mutations:
10 387752 m1 1308 0 0.5 p2 9404 130
47 387966 m1 5994 0 0.5 p1 9415 130
...
Individuals:
p1:i0 H p1:0 p1:1
p1:i1 H p1:2 p1:3
...
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19...
p1:1 A 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84...
...
```

The header line begins with `#OUT:` and then gives the tick and cycle in which the output was produced, followed by an A (for “all”). If the output is sent to a file with `outputFull()`’s `filePath` option, this is followed by the full path of the file to which the output was saved; for example:

```
#OUT: 10000 10000 A /Users/bhaller/Desktop/full.txt
```

Beginning in SLiM 2.3, the header line is followed by a single line that indicates the version of the `outputFull()` file format. SLiM 2.3’s format is indicated by a version number of 3 (for internal reasons); if the version line is missing, it may be inferred that the file format is pre-2.3. The reason for this addition is that it allows SLiM (and other software) to more accurately read in output files generated by different versions of SLiM, since the version number indicates what the reader software can expect to find in the file. A version of 4 is used by SLiM 3.0 and later to indicate that age values are included in the Individuals section; see below. Beginning in SLiM 3.5, individual pedigree IDs can optionally be written out to the Individuals section; this is indicated with new version numbers of 5 (without age information) or 6 (with age information), parallel to the existing usage of version numbers 3 (without age information) and 4 (with age information). In other words, beginning with SLiM 3.5 four version numbers are now simultaneously in active use: 3 (no ages, no pedigree ids), 4 (ages, no pedigree ids), 5 (no ages, pedigree ids), and 6 (ages, pedigree ids). This is a bit confusing, but has allowed backward compatibility to be preserved.

After these header lines comes the Populations section, which lists all currently existing subpopulations. First is given the subpopulation’s identifier, such as `p1`. Next is listed its current size, in individuals (not genomes). Finally, an H indicates that the population is composed of hermaphroditic individuals; if sex has been enabled with `initializeSex()`, this will instead be an S followed by the current sex ratio of the subpopulation (i.e., `S 0.5`).

Next is the Mutations section, which lists all of the currently segregating mutations in the population. Each mutation is listed on a separate line. The first field is a within-file numeric identifier for the mutation, beginning at 0 and counting up (although mutations are not listed in sorted order according to this value); see below for a note on why this field exists. Second is the

mutation's `id` property (see section 24.10.1), a within-run unique identifier for mutations that does not change over time, and can thus be used to match up information on the same mutation within multiple output dumps made at different times. Third is the identifier of the mutation's mutation type, such as `m1`. Fourth is the position of the mutation on the chromosome, as a zero-based base position. Fifth is the selection coefficient of the mutation, and sixth is its dominance coefficient (the latter being a property of the mutation type, in fact). Seventh is the identifier for the subpopulation in which the mutation originated, and eighth is the tick in which it originated. Finally, the ninth field gives the mutation's prevalence: an integer count of the number of times that it occurs in any genome in the population.

Following this section is the `Individuals` section. This describes each individual in each subpopulation and specifies which genomes belong to it. The first field is an identifier for the individual, such as `p1:i0` (which indicates the 0th individual in `p1`). Beginning in SLiM 3.5 with version 5, the next field is *optionally* the pedigree ID of the individual (from which the genome pedigree IDs can be derived); in version 4 and earlier this field is not present, and even in SLiM 3.5 it is only present if `pedigreeIDs=T` is passed to `outputFull()` (the default is `F`), so this is not shown in the example output above. Next is the sex of the individual: `H` for a hermaphrodite, or if sex has been enabled, `M` for a male or `F` for a female. Following that come two genome specifiers, of the form `p1:0` (indicating the 0th genome in `p1`).

Beginning in SLiM 2.3, the genome specifiers may be followed by spatial positioning information for each individual. This will be the case if continuous space has been enabled with the `dimensionality` parameter to `initializeSLiMOptions()` and the `spatialPositions` parameter to `outputFull()` is `T` (which is the default). If both of those preconditions are satisfied, then the properties of `Individual` that represent spatial positions (`x`, `y`, and/or `z`) will be output as floating-point values. Only properties that are included in the dimensionality of the simulation will be output. For example, if the simulation has been configured to have dimensionality `"xy"` with `initializeSLiMOptions()`, then the `Individuals` section might look like this:

```
Individuals:
p1:i0 H p1:0 p1:1 0.397687 0.522408
p1:i1 H p1:2 p1:3 0.066159 0.74749
...
```

The first floating-point value on each line is the value of `x` for that individual; the second is the value of `y`. The value of the `z` property is not output because the `z`-coordinate is not included in the dimensionality of the simulation. If `spatialPositions=F` were specified in the call to `outputFull()`, this positional information would not be output and the file format would be identical to that produced by version 2.1 (apart from the addition of the `Version` line in SLiM 2.3, described above).

Beginning in SLiM 3.0, the genome specifiers may be followed by the age of each individual. This will be the case if the simulation is using the `nonWF` model type (which is *not* the default) and the `ages` parameter to `outputFull()` is `T` (which is the default). If both of those preconditions are satisfied, then the `age` property of each individual will be output as an integer at the end of each line in the `Individuals` section, following the genome specifiers and (if present) the optional spatial positioning information controlled by `spatialPositions`. If age information is not output, the output format is just as it was before SLiM 3.0, for backward compatibility. Note that if age information is included in the output, the version number specified in the `Version` line will be 4; if not, the version will remain 3 (as it was before SLiM 3.0).

Last comes the `Genomes` section, which specifies all of the mutations carried by each genome in the population. The first field is a genome specifier, such as `p1:0`, as described above. Second is the type of genome: an `A` for an autosome, or an `X` or `Y` for those types if modeling of sex

chromosomes has been enabled. This is followed by a list of within-file mutation identifiers, as given in the Mutations section described above, that identify all the mutations carried on the genome. Alternatively, if the genome is a “null genome” that is not allowed to carry any mutations in SLiM (such as a Y chromosome if SLiM is modeling the X), the tag <null> will appear instead of any mutation identifiers.

The reader might wonder why the within-file index for mutations (the first field in each mutation’s output line) even exists. Couldn’t the mutation’s id (the second field) be used for that purpose, since it also uniquely identifies mutations? The answer is: yes, in principle it could. In practice, however, id values for mutations are often very large numbers – six, seven, or even more digits long. Because the bulk of a SLiM output file consists of the sequences of mutation identifiers listed in the Genomes section, using small zero-based numbers for these identifiers actually makes SLiM’s output files markedly smaller than they would be if mutation id values were used instead. Keeping output files small is an end in itself, since disk space is limited, but also has the benefit of making file writes and reads faster.

On the topic of file size and read/write speed, note that the `outputFull()` method also provides the option of writing out a binary file. The format of that file is not documented, and is subject to change at any time (although we will try to preserve backward compatibility when possible). Binary files can be smaller, and their read and write times are much faster.

Note that the output from `outputFull()` is the only output format that SLiM can read as well as write. This is done with the `readFromPopulationFile()` method of `Species` (see section 24.14.2).

In SLiM 3.3 and later, `outputFull()` will add a column to the end of the output format of the Mutations section when the mutation being output is nucleotide-based. That column will contain an A, C, G, or T, as in the following example:

```
...
Mutations:
10 387752 m1 1308 0 0.5 p2 9404 130 A
47 387966 m1 5994 0 0.5 p1 9415 130 G
...
```

In nucleotide-based models, an additional output section will also exist, at the end of the output, providing the full ancestral sequence for the model. For example:

```
Ancestral sequence:
CGCTCCTTATCAGGGGCTCAGGCCGCTCAAATATTTGATACGAGGGTAGGTCCGGAAGATCGGATAATGG
AAGTCCTACGGCATCTTTATCCTTAGCCGGTATGTGGTGCATTCTAACTGTACTATAGACCCTGAGA
...
```

The sequence itself is in FASTA format, with 70 characters per line. Note that this output section may be very large, for models with a long chromosome. For a chromosome length of  $10^9$ , the ancestral sequence will be  $10^9$  characters long, which will occupy  $10^9$  bytes, or 1 GB, in the file on disk. If the output is done in binary format instead, the overhead will be about one-quarter of that. This section may be suppressed by supplying `ancestralSequence=F` to the `outputFull()` call; however, the ancestral sequence will then not be correctly restored if the saved output file is read in with `readPopulationFile()`.

### 26.1.2 `outputFixedMutations()`

The `outputFixedMutations()` method outputs information on all mutations that have fixed and been turned into Substitution objects. It therefore complements the information produced by `outputFull()`. Sample output for `outputFixedMutations()`, abbreviated with ellipses:

```
#OUT: 10000 10000 F
Mutations:
0 390 m1 701 0 0.5 p2 20 650
1 1114 m1 957 0 0.5 p1 55 650
...
```

The header line has the standard SLiM output tag `#OUT:` followed by the tick and cycle, and then an F (for “fixed”). If the output is sent to a file with `outputFixedMutations()`’s `filePath` option, this is followed by the full path of the file to which the output was saved.

Following this is one section of output, `Mutations`. This lists every mutation that has fixed and been turned into a `Substitution` object. The first eight fields used are identical to those used in the `Mutations` section of `outputFull()` as described above: (1) a within-file identifier counting upward from 0, (2) the mutation’s `id` property that uniquely identifies in within a run, (3) the identifier for the mutation type, (4) the position on the chromosome, (5) the selection coefficient, (6) the dominance coefficient, (7) the originating subpopulation, and (8) the origination tick. The last field is different, however; instead of being a prevalence (which would be useless since these mutations are, by definition, fixed), this field indicates the tick in which the mutation was converted to a `Substitution` object (which is the same as the tick in which it fixed, unless you are dynamically changing the `convertToSubstitution` flag).

In SLiM 3.3 and later, `outputFixedMutations()` will add a column to the end of this output format when the substitution being output is nucleotide-based. That column will contain an A, C, G, or T, as in the following example:

```
#OUT: 2000 2000 F
Mutations:
0 0 m1 20000 0.1 0.5 p1 1 228
1 1467 m2 65261 0 0.5 p1 201 1418 A
2 3029 m2 68716 0 0.5 p1 411 1418 C
3 4754 m2 295 0 0.5 p1 642 1654 G
4 1026 m2 48460 0 0.5 p1 143 1654 C
...
```

Note that if the substitutions to be output are a mix of nucleotide-based mutations and non-nucleotide-based mutations, some lines will contain the extra nucleotide column and some will not. In the example above, the first substitution (of mutation type `m1`) is non-nucleotide-based, so the nucleotide column is omitted, but the rest of the substitutions (of mutation type `m2`) are nucleotide-based, and so nucleotides (A, C, G, and C) are given.

### 26.1.3 `outputMutations()`

The `outputMutations()` method is intended to be used to output information about particular mutations of interest that are being “tracked” – mutations of a particular mutation type, for example, or perhaps a specific introduced mutation. Sample output for `outputMutations()`:

```
#OUT: 10000 10000 T p1 388376 m1 673 0 0.5 p2 9434 43
#OUT: 10000 10000 T p2 388376 m1 673 0 0.5 p2 9434 27
```

These two lines of output are the result of an `outputMutations()` call requesting output for just a single mutation. The first line gives information about the prevalence of the mutation in subpopulation `p1`, whereas the second line gives the same for `p2`. If you requested output for more than one mutations, you get a line for each mutation requested:

```
#OUT: 10000 10000 T p1 388376 m1 673 0 0.5 p2 9434 43
#OUT: 10000 10000 T p1 388788 m1 9394 0 0.5 p2 9455 57
#OUT: 10000 10000 T p1 390206 m1 6232 0 0.5 p2 9523 57
```

```
...
#OUT: 10000 10000 T p2 388376 m1 673 0 0.5 p2 9434 27
#OUT: 10000 10000 T p2 388788 m1 9394 0 0.5 p2 9455 73
#OUT: 10000 10000 T p2 390206 m1 6232 0 0.5 p2 9523 73
...
```

Note that the output is sorted by subpopulation, not by mutation, so the lines for a particular mutation do not necessarily end up adjacent. If a mutation is not present in a given subpopulation at all, no output line is produced for that mutation in that subpopulation.

The format of each output line follows a similar pattern to other output methods. First comes the #OUT: tag, followed by the tick and cycle, and then a T (for “tracked”, for historical reasons). Next comes the subpopulation identifier, such as p1, for which the line is being produced. The remaining fields are the same mutation information as produced by outputFull(): (1) the mutation’s id property, (2) the identifier of its mutation type, (3) its position, (4) its selection coefficient, (5) its dominance coefficient, (6) origin subpopulation identifier, (7) origin tick, and (8) prevalence. Note that even if outputMutations()’s filePath parameter is used to send the output to a file, the filename is not added at the end of the header line as it is with SLiM’s other output commands, to keep the output from this command concise (since it really consists of nothing but header lines).

In SLiM 3.3 and later, outputMutations() will add a column to the end of this output format when the mutation being output is nucleotide-based. That column will contain an A, C, G, or T, as in the following example:

```
#OUT: 100 100 T p1 0 m1 20000 0.1 0.5 p1 1 270
#OUT: 100 100 T p1 43 m2 1562 0 0.5 p1 6 54 C
#OUT: 100 100 T p1 64 m2 96061 0 0.5 p1 9 52 T
#OUT: 100 100 T p1 83 m2 88245 0 0.5 p1 12 54 C
#OUT: 100 100 T p1 98 m2 93155 0 0.5 p1 14 54 A
...
```

Note that if the vector of mutations to be output is a mix of nucleotide-based mutations and non-nucleotide-based mutations, some lines will contain the extra nucleotide column and some will not. In the example above, the first mutation (of mutation type m1) is non-nucleotide-based, so the nucleotide column is omitted, but the rest of the mutations (of mutation type m2) are nucleotide-based, and so nucleotides (C, T, C, and A) are given.

## 26.2 Subpopulation output methods

The output methods of Subpopulation produce output about the mutations carried by a sampled subset of the Subpopulation. Three different formats of output are presently available: SLiM’s native format, MS, and VCF.

### 26.2.1 outputSample()

The outputSample() method takes a random sample of genomes from the subpopulation as requested (with options regarding sample size, replacement, and sex) and outputs information on them in SLiM’s native format. If the sampling options provided by outputSample() are insufficiently flexible, the output() method of Genome is a more general-purpose method (see section 26.3.1). Sample output for outputSample(), abbreviated with ellipses:

```
#OUT: 10000 10000 SS p1 10
Mutations:
65 587710 m1 1308 0 0.5 p2 9404 5
101 587924 m1 5994 0 0.5 p1 9415 5
```

```

...
Genomes:
p1:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23...
p1:1 A 0 1 2 3 4 5 6 8 9 10 49 50 11 12 13 14 15 16 17 18 51 19 22 23...
...

```

The header line starts with the usual tag, `#OUT:`, followed by the tick and cycle, and then `SS` (representing “sample, SLiM format”). It then gives the identifier for the subpopulation sampled, such as `p1`, and finally the size of the sample (in genomes, not individuals). If the output is sent to a file with `outputSample()`’s `filePath` option, this is followed by the full path of the file to which the output was saved.

This is followed by a Mutations section and then a Genomes section. The formats of these is identical to the same sections in the output of `outputFull()`, described in section 26.1.1, except that the prevalence values given for mutations are their prevalence within the sample of genomes, not in the population as a whole. Note that the Individuals section provided by `outputFull()` is also not present in the output from `outputSample()`, because the sample is of genomes, not complete individuals.

### 26.2.2 `outputMSSample()`

The `outputMSSample()` method takes a random sample of genomes from the subpopulation as requested (with options regarding sample size, replacement, and sex) and outputs information on them in MS format. If the sampling options provided by `outputMSSample()` are insufficiently flexible, the `outputMS()` method of `Genome` is a more general-purpose method (see section 26.3.2). Sample output for `outputMSSample()`, abbreviated with ellipses:

```

#OUT: 10000 10000 SM p1 10
//
segsites: 179
positions: 0.1308131 0.4991499 0.5994599 0.6999700 0.9599960...
00101001000001101011111000000011111000100100111100001000011000100000...
00101001000001101011111000000011111000100100111100001000011000100000...
...

```

The first line is a header in the same format as for `outputSample()`, as described in the previous section. The output type code here, `SM`, represents “sample, MS format”. The `outputMSSample()` method allows output to be sent to a file, with the optional `filePath` argument. In this case, the `#OUT:` header line is not emitted, since it would not be conformant with the MS data format specification.

This is followed by an empty comment line `//`, and then a line stating the total number of segregating sites output. Note that, as with all other output methods in SLiM, these sites are segregating *in the population*, but every genome in the sample may be identical at a given site.

Next comes a line giving the position on the chromosome of each of the segregating sites. These positions have been converted by SLiM from base positions to floating-point positions in the interval `[0,1]` as expected for the MS format. Note that SLiM allows multiple mutations at exactly the same position, so even without roundoff (which may also be an issue for very long chromosomes), two positions in this list may be specified with exactly the same number.

Finally, the output has one line for each genome in the sample. Each line is a simple sequences of 0’s and 1’s, indicating whether the genome in question possesses (1) or does not possess (0) the mutation at the corresponding position in the list of positions.



### 26.2.3 outputVCFSample()

The `outputVCFSample()` method takes a random sample of individuals (*not* genomes!) from the subpopulation as requested (with options regarding sample size, replacement, and sex) and outputs information on them. If the sampling options provided by `outputVCFSample()` are insufficiently flexible, the `outputVCF()` method of `Genome` is a more general-purpose method (see section 26.3.3). Sample output for `outputVCFSample()`, abbreviated with ellipses:

```
#OUT: 10000 10000 SV p1 10
##fileformat=VCFv4.2
##fileDate=20160613
##source=SLiM
##slimGenomePedigreeIDs=4999898,4999899,...
##INFO=<ID=MID,Number=1,Type=Integer,Description="Mutation ID in SLiM">
##INFO=<ID=S,Number=1,Type=Float,Description="Selection Coefficient">
##INFO=<ID=DOM,Number=1,Type=Float,Description="Dominance">
##INFO=<ID=P0,Number=1,Type=Integer,Description="Population of Origin">
##INFO=<ID=T0,Number=1,Type=Integer,Description="Tick of Origin">
##INFO=<ID=MT,Number=1,Type=Integer,Description="Mutation Type">
##INFO=<ID=AC,Number=1,Type=Integer,Description="Allele Count">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=MULTIALLELIC,Number=0,Type=Flag,Description="Multiallelic">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT i0 i1 i2 i3 i4 i5...
1 1309 . A T 1000 PASS
MID=987550;S=0;DOM=0.5;P0=2;T0=9404;MT=1;AC=11;DP=1000 GT 0|1 0|0 0|1...
1 5995 . A T 1000 PASS
MID=987764;S=0;DOM=0.5;P0=1;T0=9415;MT=1;AC=11;DP=1000 GT 0|1 0|0 0|1...
...
```

The first line is a header, similar to that produced by `outputSample()` and `outputMSSample()`. The output code `SV` here represents “sample, VCF format”. The `outputVCFSample()` method allows output to be sent to a file, with the optional `filePath` argument. In this case, the `#OUT:` header line is not emitted, since it would not be conformant with the VCF data format specification.

Following that is the VCF header, which provides various information about the information in the file; the VCF format is quite complex so we will not attempt to document it in detail here. The SLiM `genomePedigreeID` value for each sampled genome will be provided, in order, in the `##slimGenomePedigreeIDs` header line. (Pedigree IDs for sampled individuals are not provided, since each diploid sample in the VCF output might be comprised of two genomes from different individuals; the individual pedigree IDs are therefore not well-defined in the general case. However, in general, the individual pedigree ID for the individual to which a given genome belongs is equal to the genome pedigree ID divided by two, rounded down; in other words, an individual with pedigree ID 10 has genomes with pedigree IDs of 20 and 21.)

Note that the `INFO` fields provided by SLiM include fields for a lot of SLiM-specific information that is not part of the VCF standard itself: the mutation’s `id` property, selection and dominance coefficients, subpopulation of origin and tick of origin, and mutation type (the numeric part of a mutation type identifier like `m1`). These will have no meaning to most VCF tools, but may be useful for filtering or other analysis. The VCF header also describes two standard `INFO` tags: `AC` and `DP`. `AC` gives the “allele count”, the number of occurrences of the given mutation within the sample. `DP` gives the “total depth”, a property of empirical genomic samples that is meaningless for SLiM output; it is supplied, and is always equal to `1000`, in SLiM output to facilitate processing with VCF tools that expect this tag to be present. The last `INFO` tag described in the header is `MULTIALLELIC`; it is discussed below.

Following the VCF header are lines describing each mutation. Because of word-wrapping and line-breaking issues, these lines look a little funny here, but this is actually a single line, with fields separated by tab characters:

```
1 1309 . A T 1000 PASS
MID=987550;S=0;DOM=0.5;P0=2;T0=9404;MT=1;AC=11;DP=1000 GT 0|1 0|0 0|1...
```

These fields correspond to the column headings given in the last line of the VCF header:

```
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT i0 i1 i2 i3 i4 i5...
```

The first field is the chromosome identifier; SLiM always emits 1 for this. Next is the position of the mutation; note that VCF uses 1-based positions, so this is the position used internally by SLiM plus 1. The next five fields have VCF-oriented meanings that are unimportant for SLiM; they will always be . A T 1000 PASS. The next field is very long, and consists of a series of INFO tags separated by semicolons; the meaning of those fields is as specified by the ##INFO lines in the VCF header. Next comes a GT tag indicating the start of genotype data, and finally a sequence of “calls” of the format 0|0, 0|1, 1|0, or 1|1, indicating whether a given individual in the sample possessed (1) or did not possess (0) the mutation in each of its two genomes. This data has essentially the same meaning as MS-format data, but in a notation that groups the 0’s and 1’s into homologous chromosomes in diploid individuals.

There are several things to note about this. First of all, calls are normally diploid (0|0, 1|0, etc.), but if SLiM is modeling sex chromosomes, calls may be haploid instead (just a 0 or a 1). For example, if you are modeling the X chromosome, males will be emitted as haploid while females will be emitted as diploid. There is no way to represent a 0-ploid individual in VCF format, so if you are modeling the Y chromosome you may not include females in your sample; there would be no genetic information to emit for them.

Second, it is important to emphasize that VCF output, unlike all other output from SLiM, is based on individuals, not on genomes. The subpopulation sample taken by `outputVCFsample()` is a sample of individuals, not genomes, and thus a sample of size 10 will include twice as many genomes as a sample of size 10 would include for `outputMSSample()` or `outputSample()`.

Third, note that SLiM designates all mutations as being a change from an A to a T. Since SLiM has no concept of nucleotide sequence (in non-nucleotide-based models), this is simply an arbitrary choice. If you wished to construct a FASTA file for the ancestral sequence, for example, it would simply be the length of the chromosome, filled with A’s.

Finally, there are several subtle points related to the nature of the VCF output produced by SLiM that we have glossed over here. One is the MULTIALLELIC tag defined in the VCF header above. In SLiM, it is often possible for a single individual to have multiple mutations at a given base position. Because the VCF format is an explicit-nucleotide format, this property of SLiM does not fit well into VCF. Since there are only four possible nucleotides at a given base position in VCF, at most one “reference” state and three “alternate” states could be represented at that base position. SLiM, on the other hand, can represent any number of alternative possibilities at a given base; in general, if  $N$  different mutations are segregating at a given position, there are  $2^N$  different allelic states at that position in SLiM. For this reason, SLiM does not attempt to represent multiple mutations at a single site as being alternative alleles in a single output line, as is typical in VCF format. Instead, SLiM produces a separate line of VCF output for each segregating mutation at a given position. In non-nucleotide-based models, SLiM always declares base positions as having a “reference base” of A (representing the state in individuals that do not carry a given mutation) and an “alternate base” of T (representing the state in individuals that do carry the given mutation). Multiallelic positions will thus produce VCF output showing multiple A-to-T changes at the same position, possessed by

different but possibly overlapping sets of individuals. Many programs that process VCF output may not behave correctly with this style of output. SLiM therefore provides a choice, using the `outputMultiallelics` flag; if that flag is T (the default), SLiM will produce multiple lines of output for multiallelic base positions, but will mark those lines with a `MULTIALLELIC` flag in the `INFO` field of the VCF output so that those lines can be filtered or processed in a special manner. The resulting call lines in the VCF output look like this:

```
1 3 . A T 1000 PASS MID=1769;S=0;DOM=0.5;PO=1;T0=10;MT=1;AC=1;DP=1000
GT 0|0 0|0 0|0 0|0 0|0 0|0 0|0 1|0 0|0 0|0
1 5 . A T 1000 PASS MID=1574;S=0;DOM=0.5;PO=1;T0=9;MT=1;AC=1;DP=1000
GT 0|0 0|0 0|0 0|0 0|1 0|0 0|0 0|0 0|0 0|0
1 7 . A T 1000 PASS MID=876;S=0;DOM=0.5;PO=1;T0=6;MT=1;AC=3;
DP=1000;MULTIALLELIC GT 0|0 0|0 1|1 0|0 1|0 0|0 0|0 0|0 0|0 0|0
1 7 . A T 1000 PASS MID=393;S=0;DOM=0.5;PO=1;T0=3;MT=1;AC=2;
DP=1000;MULTIALLELIC GT 1|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 1|0 0|0
1 8 . A T 1000 PASS MID=1705;S=0;DOM=0.5;PO=1;T0=10;MT=1;AC=1;
DP=1000;MULTIALLELIC GT 0|1 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0
1 8 . A T 1000 PASS MID=1236;S=0;DOM=0.5;PO=1;T0=8;MT=1;AC=4;
DP=1000;MULTIALLELIC GT 0|1 0|0 0|0 0|0 0|0 0|0 1|1 0|1 0|0 0|0
1 10 . A T 1000 PASS MID=1154;S=0;DOM=0.5;PO=1;T0=7;MT=1;AC=2;DP=1000
GT 0|0 0|0 0|0 1|0 0|0 0|0 0|0 0|0 0|1 0|0
...
```

Positions 3, 5, and 10 in this output are not multiallelic; there is a single mutation segregating at those positions, shown as an A-to-T mutation as described above. Positions 7 and 8, however, each have two different mutations segregating; those mutations are therefore emitted as separate lines tagged by `MULTIALLELIC`. Note that position 7 is multiallelic simply because more than one mutation is segregating at that position; mutation id 876 is in three genomes, and mutation id 393 is in two different genomes. Position 8, however, is multiallelic in part because of *stacking*: mutation id 1705 is in one genome, mutation id 1236 is in four genomes, and those sets of genomes overlap – the second genome of the first individual contains both mutations (1705 presumably arose on the background of 1236). See section 1.5.3 for discussion of mutation stacking. Both situations are covered by the `MULTIALLELIC` flag.

If `outputMultiallelics` is F, on the other hand, SLiM will completely suppress output of all mutations at multiallelic sites – often the simplest option, if doing so does not lead to bias in the subsequent analysis. This flag has no effect upon the output of sites with only a single mutation present; the output shown above would therefore contain the lines for positions 3, 5, and 10, but the lines for mutations at positions 7 and 8 would be omitted since those positions are multiallelic. Assessment of whether a site is multiallelic is done only within the sample; segregating mutations that are not part of the sample are ignored. The `outputMultiallelics` flag is ignored in nucleotide-based models; see section 26.2.4.

The other more subtle points related to VCF output have to do with output from nucleotide-based models – in particular, the effects of the `simplifyNucleotides` and `outputNonnucleotides` flags. For information on nucleotide-based VCF output, see section 26.2.4.

#### 26.2.4 `outputVCFSample()` in nucleotide-based models

Section 26.2.3 describes VCF output from non-nucleotide-based models. In nucleotide-based models, it is somewhat different. Nucleotide-based mutations will be emitted with their correct nucleotides, of course, rather than always being emitted as A-to-T mutations. The “reference” nucleotide for a given call will be the ancestral nucleotide; it will also be given with the `AA` field in the `INFO` of the call line, redundantly. If `simplifyNucleotides` is F (the default), SLiM’s mutational state will be emitted verbatim; this may include oddities such as an A allele on an ancestral

background of A (if a back-mutation has occurred at a site that had previously mutated away from A), or multiple G alleles with different selection coefficients segregating at the same position (if more than one independent mutation to G has occurred at the site). As a result, the VCF file output produced may not be considered compliant by some software, since these patterns are not something one would typically see in VCF files associated with empirical biological data. Here is an example of VCF output with `simplifyNucleotides` being F:

```
#OUT: 10 10 SV p1 10
##fileformat=VCFv4.2
##fileDate=20190308
##source=SLiM
##slimGenomePedigreeIDs=4999898,4999899,...
##INFO=<ID=MID,Number=.,Type=Integer,Description="Mutation ID in SLiM">
##INFO=<ID=S,Number=.,Type=Float,Description="Selection Coefficient">
##INFO=<ID=DOM,Number=.,Type=Float,Description="Dominance">
##INFO=<ID=PO,Number=.,Type=Integer,Description="Population of Origin">
##INFO=<ID=TO,Number=.,Type=Integer,Description="Tick of Origin">
##INFO=<ID=MT,Number=.,Type=Integer,Description="Mutation Type">
##INFO=<ID=AC,Number=.,Type=Integer,Description="Allele Count">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=NONNUC,Number=0,Type=Flag,Description="Non-nucleotide-based">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT i0 i1 i2 i3 i4 i5 i6 i7 i8
i9
1 4 . G C 1000 PASS MID=1085;S=0;DOM=0.5;PO=1;TO=9;MT=2;AC=2;DP=1000;AA=G
GT 0|0 0|0 0|0 0|0 0|0 1|0 0|0 0|1 0|0 0|0
1 7 . A C,C,C,C 1000 PASS MID=318,1283,1274,1270;S=0,0,0,0;
DOM=0.5,0.5,0.5,0.5;PO=1,1,1,1;TO=3,10,10,10;MT=2,2,2,2;AC=1,1,1,1;
DP=1000;AA=A GT 0|0 0|0 2|0 0|0 0|0 0|1 0|0 4|3 0|0 0|0
1 8 . G T 1000 PASS MID=1236;S=0;DOM=0.5;PO=1;TO=10;MT=2;AC=1;DP=1000;AA=G
GT 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0 1|0
1 9 . G T 1000 PASS MID=316;S=0;DOM=0.5;PO=1;TO=3;MT=2;AC=1;DP=1000;AA=G
GT 0|0 0|0 0|0 0|0 0|0 0|1 0|0 0|0 0|0 0|0
1 11 . G C,G 1000 PASS MID=53,659;S=0,0;DOM=0.5,0.5;PO=1,1;TO=2,6;
MT=2,2;AC=2,1;DP=1000;AA=G GT 0|0 0|0 0|1 0|0 1|0 0|0 0|2 0|0 0|0 0|0
1 13 . C T,T 1000 PASS MID=1280,1067;S=0,0;DOM=0.5,0.5;PO=1,1;TO=10,9;
MT=2,2;AC=1,2;DP=1000;AA=C GT 0|0 0|0 1|0 0|0 0|2 0|0 2|0 0|0 0|0 0|0
1 14 . G A 1000 PASS MID=899;S=0;DOM=0.5;PO=1;TO=7;MT=2;AC=1;DP=1000;AA=G
GT 0|0 0|0 0|0 0|0 0|0 0|1 0|0 0|0 0|0 0|0
1 15 . G A 1000 PASS MID=353;S=0;DOM=0.5;PO=1;TO=4;MT=2;AC=1;DP=1000;AA=G
GT 0|0 0|0 0|0 0|0 0|0 0|1 0|0 0|0 0|0 0|0
1 18 . C T 1000 PASS MID=95;S=0;DOM=0.5;PO=1;TO=2;MT=2;AC=9;DP=1000;AA=C
GT 0|1 1|1 1|0 1|1 0|0 0|0 0|0 1|0 1|1 0|0
1 19 . T G,A 1000 PASS
MID=1307,1241;S=0,0;DOM=0.5,0.5;PO=1,1;TO=10,10;MT=2,2;AC=1,1;DP=1000;AA=T
GT 0|0 0|0 0|0 0|0 0|0 1|0 0|0 0|0 0|0 0|2
...
```

There are several things to note here. The AA and NONNUC tags are now declared in the header as INFO fields (the NONNUC flag will be discussed below). Some call lines, such as position 4, have just a single segregating mutation and so appear similar to the output we have seen before, except that actual nucleotides are specified (G to C, for position 4) and the ancestral allele is given in the AA field of the call line. Position 19 has an ancestral state of T, and two mutations, G and A, are segregating; the call line therefore lists both, and various INFO fields provide values for both of the segregating alleles (two values for S, the selection coefficient, and DOM, the dominance coefficient,

and so forth). Position 7 has an ancestral state of A, and four independent mutations to C are segregating at that site; they are listed separately, with separate values for their relevant state. At position 11 the ancestral state is G, and a mutation to C as well as a back-mutation to G are segregating; the back-mutation is listed as a separate allele rather than being lumped into the ancestral state. Some VCF processing software would probably not handle this output correctly, given these unusual features.

If `simplifyNucleotides` is T instead, SLiM's mutational state will be simplified; an A allele on an ancestral background of A will be reported as the ancestral state, for example, and multiple G alleles with different selection coefficients segregating at the same position will be reported as a single G allele. In this case, the SLiM-specific information about nucleotide-based mutations (such as selection coefficient and subpopulation of origin) will not be emitted, since a single allele in the VCF file might be an aggregate representing more than one mutation in SLiM. The call lines for the same model run as above, but with `simplifyNucleotides` being T, would look like:

```
#OUT: 10 10 SV p1 10
##fileformat=VCFv4.2
##fileDate=20190308
##source=SLiM
##slimGenomePedigreeIDs=4999898,4999899,...
##INFO=<ID=MID,Number=.,Type=Integer,Description="Mutation ID in SLiM">
##INFO=<ID=S,Number=.,Type=Float,Description="Selection Coefficient">
##INFO=<ID=DOM,Number=.,Type=Float,Description="Dominance">
##INFO=<ID=PO,Number=.,Type=Integer,Description="Population of Origin">
##INFO=<ID=TO,Number=.,Type=Integer,Description="Tick of Origin">
##INFO=<ID=MT,Number=.,Type=Integer,Description="Mutation Type">
##INFO=<ID=AC,Number=.,Type=Integer,Description="Allele Count">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=AA,Number=1,Type=String,Description="Ancestral Allele">
##INFO=<ID=NONNUC,Number=0,Type=Flag,Description="Non-nucleotide-based">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT i0 i1 i2 i3 i4 i5 i6 i7 i8
i9
1 4 . G C 1000 PASS AC=2;DP=1000;AA=G GT 0|0 0|0 0|0 0|0 0|0 1|0 0|0 0|1
0|0 0|0
1 7 . A C 1000 PASS AC=4;DP=1000;AA=A GT 0|0 0|0 1|0 0|0 0|0 0|1 0|0 1|1
0|0 0|0
1 8 . G T 1000 PASS AC=1;DP=1000;AA=G GT 0|0 0|0 0|0 0|0 0|0 0|0 0|0 0|0
0|0 1|0
1 9 . G T 1000 PASS AC=1;DP=1000;AA=G GT 0|0 0|0 0|0 0|0 0|0 0|1 0|0 0|0
0|0 0|0
1 11 . G C 1000 PASS AC=2;DP=1000;AA=G GT 0|0 0|0 0|1 0|0 1|0 0|0 0|0 0|0
0|0 0|0
1 13 . C T 1000 PASS AC=3;DP=1000;AA=C GT 0|0 0|0 1|0 0|0 0|1 0|0 1|0 0|0
0|0 0|0
1 14 . G A 1000 PASS AC=1;DP=1000;AA=G GT 0|0 0|0 0|0 0|0 0|0 0|1 0|0 0|0
0|0 0|0
1 15 . G A 1000 PASS AC=1;DP=1000;AA=G GT 0|0 0|0 0|0 0|0 0|0 0|1 0|0 0|0
0|0 0|0
1 18 . C T 1000 PASS AC=9;DP=1000;AA=C GT 0|1 1|1 1|0 1|1 0|0 0|0 0|0 1|0
1|1 0|0
1 19 . T A,G 1000 PASS AC=1,1;DP=1000;AA=T GT 0|0 0|0 0|0 0|0 0|0 2|0 0|0
0|0 0|0 0|1
```

Compared to the previous VCF output, it can be seen that the unusual SLiM-specific mutational states have been simplified. At position 7, for example, the four mutational lineages for independent mutations to C are shown as a single C allele; and at position 11 the back-mutation to



G is just considered to be equivalent to the ancestral state, and so that segregating mutation is not shown at all. The SLiM-specific mutational information such as selection coefficient, subpopulation of origin, etc., is omitted since alleles now represent an aggregation of multiple mutations, as discussed above; if that information is needed, then `simplifyNucleotides=T` cannot be used.

Nucleotide-based models can still use non-nucleotide-based mutations, and for those, the same issue discussed above regarding multiallelic sites still exists, but in nucleotide-based models it is handled differently. Instead, if `outputNonnucleotides` is T (the default), non-nucleotide-based mutations will be emitted as usual, as A-to-T mutations, in separate call lines from any nucleotide-based mutations at the same position, and their call lines will be marked with a `NONNUC` flag in the `INFO` field. For example, such a call line might look like this:

```
1 21 . A T 1000 PASS MID=0;S=0.1;DOM=0.5;PO=1;TO=1;MT=1;AC=7;
DP=1000;NONNUC GT 1|0 0|0 0|1 0|0 1|0 1|0 0|1 0|1 0|0 1|0
```

This call line indicates a non-nucleotide-based mutation at position 21; as usual, the A and T nucleotide states for non-nucleotide-based mutations are fictional. If there were nucleotide-based mutations at position 21 as well, they would be emitted with a separate call line preceding this call line.

If, on the other hand, `outputNonnucleotides` is F, non-nucleotide-based mutations will not be emitted at all, producing a more standard VCF file that contains only nucleotide-based mutations. In either case, the `MULTIALLELIC` flag is never emitted in the VCF output of nucleotide-based models, and the `outputMultiallelics` flag is ignored. (Similarly, the `outputNonnucleotides` flag is ignored in non-nucleotide-based models, as is the `simplifyNucleotides` flag.)

## 26.3 Genome output methods

The output methods of `Genome` produce output about the specific vector of `Genome` objects for which the method is called. Whereas the sampling output methods of `Subpopulation` limit you to a sample drawn from a single subpopulation, and provide only a few options regarding how that sample is conducted, with the `Genome` output methods you can construct your own vector of genomes in whatever way you wish, and produce standardized output from that sample. The `sample()` function of `Eidos` may prove useful for this, providing options such as weighted sampling that `Subpopulation`'s methods don't support. The `Individual` class of SLiM may also be useful; you can get a vector of individuals from the subpopulations you are interested in, use `sample()` to get a sample of individuals from that vector, get the genomes from those individuals using the `genomes` property of `Individual`, and then produce output from the resulting vector of genomes using these methods.

Incidentally, you might wonder why these `Genome` output methods behave differently from most `Eidos` methods – they do not multicast out to all of the objects in the target vector, producing a separate output block for each, but instead produce a single output block for the whole target vector. This is because these methods are designated as class methods, which do not multicast. This is parallel to defining a static member function in a class in C++, taking a parameter that is a `std::vector` containing elements of that same class; that would be the natural way to represent this concept in C++, whereas in `Eidos` such methods are class methods that are called on the target vector but do not multicast. If this is gibberish to you, you can ignore it; the upshot is that it just works.



### 26.3.1 *output()*

The *output()* method is parallel to the *outputSample()* method of *Subpopulation* (see section 26.2.1), but allows output based upon any vector of genomes. Sample output for *output()*, abbreviated with ellipses:

```
#OUT: 10000 10000 GS 10
Mutations:
9 187870 m1 1308 0 0.5 p2 9404 12
47 188084 m1 5994 0 0.5 p1 9415 12
...
Genomes:
p*:0 A 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23...
p*:1 A 0 1 2 3 4 5 6 68 7 8 9 69 10 12 13 14 15 16 17 18 19 20 21 22...
...
```

This is almost identical to the output from *outputSample()*, so see section 26.2.1 for further discussion. The main differences are in the header line. The output type here is *GS* (for “genomes, SLiM format”), and no subpopulation identifier is given since the genome vector being output may originate from more than one subpopulation. If the output is sent to a file with *output()*’s *filePath* option, the last field of the header provides the full path of the file to which the output was saved.

The other difference is in the genome identifiers in the *Genomes* section. Here, since the subpopulation of origin of the genomes is not known to *output()* – since it was just handed a vector of genomes that could have come from anywhere – the genome identifiers are of the form *p\*:0*, *p\*:1*, etc., with the *\** symbolizing an unknown source subpopulation. This syntax is intended to parallel the syntax used by SLiM’s other output functions, to make it easier to share parsing code.

### 26.3.2 *outputMS()*

The *outputMS()* method is parallel to the *outputMSSample()* method of *Subpopulation* (see section 26.2.2), but allows output based upon any vector of genomes. Sample output for *outputMS()*, abbreviated with ellipses:

```
#OUT: 10000 10000 GM 10
//
segsites: 165
positions: 0.1308131 0.4991499 0.5994599 0.6999700 0.9599960...
11010110111110010100000111111100000111011011000011110111100111000010...
11010110111110010100000111111100000111011011000011110111100111000010...
```

This is almost identical to the output from *outputMSSample()*, so see section 26.2.2 for further discussion. The differences are in the header line; output type *GM* is used here (representing “genomes, MS format”), and no subpopulation identifier is given since the genome vector being output may originate from more than one subpopulation.

The *outputMS()* method allows output to be sent to a file, with the optional *filePath* argument. In this case, the *#OUT:* header line is not emitted, since it would not be conformant with the MS data format specification.

### 26.3.3 *outputVCF()*

The *outputVCF()* method is parallel to the *outputVCFSample()* method of *Subpopulation* (see section 26.2.3 and 26.2.4), but allows output based upon any vector of genomes. Sample output for *outputVCF()*, abbreviated with ellipses:

```

#OUT: 10000 10000 GV 20
##fileformat=VCFv4.2
##fileDate=20160613
##source=SLiM
##slimGenomePedigreeIDs=4999898,4999899,...
##INFO=<ID=MID,Number=1,Type=Integer,Description="Mutation ID in SLiM">
##INFO=<ID=S,Number=1,Type=Float,Description="Selection Coefficient">
##INFO=<ID=DOM,Number=1,Type=Float,Description="Dominance">
##INFO=<ID=P0,Number=1,Type=Integer,Description="Population of Origin">
##INFO=<ID=T0,Number=1,Type=Integer,Description="Tick of Origin">
##INFO=<ID=MT,Number=1,Type=Integer,Description="Mutation Type">
##INFO=<ID=AC,Number=1,Type=Integer,Description="Allele Count">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
##INFO=<ID=MULTIALLELIC,Number=0,Type=Flag,Description="Multiallelic">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT i0 i1 i2 i3 i4 i5...
1 1309 . A T 1000 PASS
MID=987550;S=0;DOM=0.5;P0=2;T0=9404;MT=1;AC=11;DP=1000 GT 0|1 0|0 0|1...
1 5995 . A T 1000 PASS
MID=987764;S=0;DOM=0.5;P0=1;T0=9415;MT=1;AC=11;DP=1000 GT 0|1 0|0 0|1...
...

```

This is almost identical to the output from `outputVCFsSample()`, so see section 26.2.3 and 26.2.4 for further discussion. The differences are in the header line; type GV is used here (representing “genomes, VCF format”), and no subpopulation identifier is given since the genome vector being output may originate from more than one subpopulation. Also note that the sample size here is reported in genomes (i.e., 20) whereas with `outputVCFsSample()` it is reported in individuals (i.e., 10). This is because this method operates on a vector of genomes, and thus that is the natural unit for the sample size. Nevertheless, since VCF output naturally groups genomes into diploid individuals, the target genome vector must have an even number of elements, and each pair of elements will be assumed to represent one individual.

The `outputVCF()` method allows output to be sent to a file, with the optional `filePath` argument. In this case, the `#OUT:` header line is not emitted, since it would not be conformant with the VCF data format specification.

## 26.4 SLiM additions to the .trees file format

The `.trees` files produced by the `treeSeqOutput()` method (section 24.14.2) are in a complex binary format, defined at the top level by the `kastore` library and at the next level by `tskit`; it is not documented here. Reading or writing compliant `.trees` files is a topic well beyond the scope of this manual. However, if the `tskit.load()` method is used to load a `.trees` file into Python, the entities defined by the file, such as nodes, edges, and mutations, can then be accessed through the `tskit` APIs in Python. Those entities often have a column for metadata, and this is where SLiM attaches its additional state information. The contents of those metadata fields is documented here. Directly using or generating this metadata information in Python is, again, beyond the scope of this manual, but the information provided here at least documents what you would need to know in order to do so. The `tskit` and `pyslim` packages provide the usual route to accessing and modifying this metadata information, and should suffice for the needs of almost all users.

With `tskit` version 0.3.0, `tskit` has greatly improved metadata handling, introducing *metadata schemas* and adding metadata entries to all tables, as well as to the tree sequence itself (so-called “top-level” metadata). With these improvements, the tree sequence now carries around the information necessary to interpret the metadata it contains (see `tskit`’s documentation for further detail). Metadata schemas, stored as JSON-parseable text, provide the information

necessary to interpret and decode the metadata stored in the tree sequence. (Any equivalent JSON text will work, but for standardization, schemas should be written out without whitespace.) The metadata schemas say what elements are present in the metadata, and in what order, and how they are stored as binary. The actual schemas are lengthy; if you need to see them, they are available in `pyslim` through `pyslim.slim_metadata_schemas`. The metadata itself can be in either a binary format or in JSON, as documented below. For binary metadata the descriptions below will give the number of bytes for each field, their C / C++ type, and a brief description, essentially recapitulating the information in the metadata schemas; for JSON metadata the keys and values will be described.

#### 26.4.1 Metadata for mutations

The derived state field for each mutation entry is actually a comma-separated list of mutation IDs, in ASCII, representing all of the stacked mutations present at the position in question after the addition (or removal) of a mutation; this is rather different from the way the derived state field is used in most mutation models. Note that fixed mutations (i.e., `Substitution` objects in SLiM) will be added on to the end of this derived state list, even in cases such as nucleotide-based models where mutations do not normally stack, because of the way substitutions are reconciled internally between SLiM and the tree sequence.

Each mutation entry's metadata will then consist of a series of 17 byte binary metadata records, corresponding to the mutation IDs listed for the derived state:

- 4 bytes (`int32_t`): the id of the mutation type the mutation belongs to (i.e., 5 for `m5`)
- 4 bytes (`float`): the selection coefficient of the mutation
- 4 bytes (`int32_t`): the id of the subpopulation in which the mutation arose (i.e., 3 for `p3`)
- 4 bytes (`int32_t`): the simulation tick in which the mutation arose
- 1 byte (`int8_t`): the nucleotide for the mutation (0=A, 1=C, 2=G, 3=T, or -1 for none)

The last field, the nucleotide, is only present in file version "0.3" and later (i.e., output generated by SLiM 3.3 and later; see below); in file version "0.2" that field is not present, the mutation metadata record is only 16 bytes in length, and nucleotides cannot be represented.

Note that the same mutation ID may be described by multiple mutation entries, if differences in stacking or other factors lead to it being recorded multiple times. Furthermore, the metadata for these multiple descriptions may not match, since metadata such as the selection coefficient of a mutation can change as a mutation runs. When reading, SLiM will use the metadata associated with the last recorded version of each mutation.

#### 26.4.2 Metadata for nodes

Each node will have 10 bytes of binary metadata attached (*if* the node references an individual that is alive; otherwise, metadata is not required and may be 0 bytes):

- 8 bytes (`int64_t`): the SLiM genome ID for this node, as from `genomePedigreeID`
- 1 byte (`uint8_t`): true (1) if this node represents a null genome, as from `isNullGenome`
- 1 byte (`uint8_t`): the type of the genome (0 for autosome, 1 for X, 2 for Y)

### 26.4.3 Metadata for individuals

Each individual will have 40 bytes of binary metadata attached:

- 8 bytes (int64\_t): the SLiM pedigree ID for this individual, as from `pedigreeID`
- 8 bytes (int64\_t): the SLiM pedigree ID for parent 1, as from `pedigreeParentIDs`
- 8 bytes (int64\_t): the SLiM pedigree ID for parent 2, as from `pedigreeParentIDs`
- 4 bytes (int32\_t): the age of this individual, as from `age`
- 4 bytes (int32\_t): the subpopulation the individual belongs to (i.e., 3 for `p3`)
- 4 bytes (int32\_t): the sex of the individual (0 for female, 1 for male, -1 for hermaphrodite)
- 4 bytes (uint32\_t): flags; see below.

The two parental pedigree ID fields are present only in file version "0.7" and later (i.e., output generated by SLiM 3.7 and later; see below); in file version "0.6" those fields are not present, the individual metadata record is only 24 bytes in length, and parental pedigree IDs are not recorded.

At present, only the low-order bit of the flags metadata, `0x01`, is used; if set, it indicates that this individual migrated between subpopulations during the current tick (following the `migrant` property of `Individual` described in section 24.7.1). Other flag bits are reserved and should be set to 0 until such time as they are defined.

The individual table also has a `flags` field, outside of the metadata record, and SLiM uses some bits in that field; that will be explained below since it is not part of the metadata record itself.

### 26.4.4 Metadata for populations

Beginning in SLiM 3.7, the metadata for populations is in JSON format. The `slim_id` key is required (if metadata is present at all; it may be omitted). The rest of the keys are optional; the keys that apply only to WF models are typically omitted in nonWF metadata, but will be ignored if present. The keys are as follows:

- `bounds_x0`: spatial bounds `x0` value, omitted/unused in non-spatial models
- `bounds_x1`: spatial bounds `x1` value, omitted/unused in non-spatial models
- `bounds_y0`: spatial bounds `y0` value, omitted/unused in non-spatial models
- `bounds_y1`: spatial bounds `y1` value, omitted/unused in non-spatial models
- `bounds_z0`: spatial bounds `z0` value, omitted/unused in non-spatial models
- `bounds_z1`: spatial bounds `z1` value, omitted/unused in non-spatial models
- `description`: a human-readable description of the population, as from `description`
- `female_cloning_fraction`: the cloning fraction for females or hermaphrodites (WF)
- `male_cloning_fraction`: the cloning fraction for males or hermaphrodites (WF)
- `name`: a human-readable name for the population, as from `name`
- `selfing_fraction`: the selfing fraction (WF)

`sex_ratio`: the sex ratio as M:M+F (WF)

`slim_id`: the ID of this subpopulation, as from `id`

In addition to these basic object values, the key `migration_records` contains information about migration rates for the population (WF models only). The value for this key should be an array, made up of object records, each of which has two keys: `source_subpop` with the ID of the source subpopulation, and `migration_rate` with the migration rate from that source.

Prior to SLiM 3.7, this metadata was instead binary. In binary form, each population has 88 bytes of metadata attached at a minimum, followed by a variable number of 12-byte sections (*if* the population is referenced by a node; empty subpopulations are not required to have metadata). The initial 88-byte section is structured as:

4 bytes (`int32_t`): the ID of this subpopulation, as from `id`

8 bytes (`double`): the selfing fraction (WF) or unused (nonWF)

8 bytes (`double`): the cloning fraction for females or hermaphrodites (WF) or unused (nonWF)

8 bytes (`double`): the cloning fraction for males or hermaphrodites (WF) or unused (nonWF)

8 bytes (`double`): the sex ratio as M:M+F (WF) or unused (nonWF)

8 bytes (`double`): spatial bounds `x0` value, unused in non-spatial models

8 bytes (`double`): spatial bounds `x1` value, unused in non-spatial models

8 bytes (`double`): spatial bounds `y0` value, unused in non-spatial models

8 bytes (`double`): spatial bounds `y1` value, unused in non-spatial models

8 bytes (`double`): spatial bounds `z0` value, unused in non-spatial models

8 bytes (`double`): spatial bounds `z1` value, unused in non-spatial models

4 bytes (`int32_t`): the number of migration records, as from `immigrantSubpopFractions`

The value of the last field above dictates the number of 12-byte metadata sections that follow, each of this format:

4 bytes (`int32_t`): the ID of the source subpopulation (i.e., 3 for `p3`)

8 bytes (`double`): the migration rate from the source subpopulation

#### 26.4.5 Top-level metadata

SLiM makes use of the top-level metadata in the tree sequence to store the information necessary to reload a tree sequence generated by SLiM. The top-level metadata is encoded as a JSON string, and SLiM puts its information into the "SLiM" key of this JSON object. This information should be a JSON object with the following keys: "model\_type", "stage", "cycle", "tick", "name", "description", "file\_version", "spatial\_dimensionality", "spatial\_periodicity", "separate\_sexes", and "nucleotide\_based". Note, however, that only "model\_type", "stage", "cycle", "tick", and "file\_version" are actually used by SLiM or `pyslim`; the rest are provided just for informational purposes. Note also that the keys present in the top-level metadata have changed from version to version of SLiM, as described in the notes below; notably, prior to SLiM 4 the "tick" key was not present and the "cycle" key was named

"generation" instead. If backward compatibility with tree-sequence files written by older versions of SLiM is needed, care must be taken.

Here is an example, from the top-level metadata generated by the recipe in section 17.1 (with whitespace added for readability):

```
{
  "SLiM": {
    "file_version": "0.8",
    "description": "",
    "cycle": 5000,
    "model_type": "WF",
    "name": "sim",
    "nucleotide_based": false,
    "separate_sexes": false,
    "spatial_dimensionality": "",
    "spatial_periodicity": "",
    "stage": "late"
    "tick": 5000
  }
}
```

These keys have the following meanings:

**file\_version:** the overall version of the SLiM-specific information in the file, such as metadata and provenance entry information. The current value, for SLiM 4.0, is "0.8". See below for further details.

**cycle:** the cycle count at which the file was written, for the focal species, so that that cycle count can be restored when the file is read.

**tick:** the tick at which the file was written, for the community, so that that tick can be restored when the file is read. For single-species models, this will generally have the same value as **cycle**; for multispecies models it can be different, if for example the focal species has been inactive in some ticks.

**name:** the name of the species persisted in the `.trees` file ("sim" for single-species models); this is the name property of the species, as set by the explicit species declaration in the model script.

**description:** the description of the species persisted in the `.trees` file, for annotation; this comes from the `description` property of the species.

**stage:** the tick cycle stage ("first", "early", or "late") at which the file was written; this is used by `pyslim` for time calculations.

**model\_type:** this should be either "WF" or "nonWF", depending upon the type of model that generated the file. This has some implications for other metadata; in particular, some of the population metadata is required for WF models but unused in nonWF models, and individual ages in WF model data are expected to be -1.

**nucleotide\_based:** this is true if the model that generated the file was nucleotide-based, false otherwise.

**separate\_sexes:** this is true if the model that generated the file was sexual (i.e., `initializeSex()` was called), false otherwise.



`spatial_dimensionality`: the spatial dimensionality of the model that generated the file; this will be "", "x", "xy", or "xyz" for non-spatial, 1D, 2D, and 3D models respectively.

`spatial_periodicity`: the spatial periodicity of the model that generated the file; this will be "", "x", "y", "z", "xy", "xz", "yz", or "xyz", indicating which spatial dimensions are periodic (if any).

A key named `user_metadata` may also be present, containing optional user-level metadata that was supplied to SLiM with the `metadata` parameter of `treeSeqOutput()`, as described in section 24.14.2. If present, the value for this key should be a JSON-serialized dictionary. This key is not used by SLiM (or by `pyslim`, `tskit`, or `msprime`); it may contain any information desired.

Finally, a top-level key (not inside the "SLiM" JSON object) named `reference_sequence` will be present for nucleotide-based models that were saved with `file_version` "0.7" or later (SLiM 3.7 or later). It contains a string value representing the ancestral ("reference") nucleotide sequence for the model, as a sequence of ACGT characters in ASCII. Prior to `file_version` "0.7" this string was placed in a `kastore` key instead, as noted below.

The `file_version` value for SLiM 3.3.x is "0.3"; it extends the mutation metadata to include nucleotide information, as described above (section 26.4.1). Version "0.3" also includes the ancestral nucleotide sequence, written out to the `.trees` files under the `kastore` key `reference_sequence/data`; this sequence is provided for nucleotide-based models, and will not be present for non-nucleotide-based models (note that it moved to top-level metadata in `file_version` "0.7", see below). These are the only changes in version "0.3" compared to "0.2"; the provenance entry format is unchanged. Version "0.2" files can still be read by SLiM 3.3 and later (but since they do not contain nucleotide information, they cannot be read into a nucleotide-based model).

The `file_version` for SLiM 3.4.x is "0.4"; it adds a `model_hash` subkey to the `parameters` top-level key, providing a SHA-256 hash (as a 64-character ASCII string containing the lowercase hexadecimal representation of the hash) for the script provided in the `model` subkey. This allows the script to be easily compared/validated against known script strings. With this extension, the `model` subkey is no longer guaranteed to be present; if the `includeModel` option to `treeSeqOutput()` is F (T is the default), the `model` subkey will be omitted from `parameters` (see section 24.14.2). This allows one to avoid writing out extremely large scripts to the provenance, which was a problem for certain usage cases. This is the only change in version "0.4" compared to "0.3".

The `file_version` for SLiM 3.5.x is "0.5"; it added top-level metadata and metadata schemas as previously discussed. It also removed the `SLIM_TSK_INDIVIDUAL_FIRST_GEN` flag from the `flags` column of the `individuals` table, since first-generation individuals no longer need to be explicitly marked and retained due to improvements in `tskit`'s simplification algorithm.

The `file_version` for SLiM 3.6.x is "0.6"; it added the `SLIM_TSK_INDIVIDUAL_RETAINED` flag to the `flags` column of the `individuals` table.

The `file_version` for SLiM 3.7 and later is "0.7"; it added the two parental pedigree ID fields to the individual metadata record, and moved the ancestral nucleotide sequence from `kastore` to top-level metadata, as described above.

The `file_version` for SLiM 4.0 and later is "0.8"; it added the `tick` key to top-level metadata, renamed the `generation` key to `cycle` in the top-level metadata, changed the semantics of the `slim_time` metadata property of mutations from generations to ticks, and added `name` and `description` metadata values, from the corresponding properties of the focal species, to the top-level metadata.

#### 26.4.6 The SLiM provenance table entry format

The provenance table is designed to hold an entry for each software program that has been involved in the creation of the file, providing a sort of “chain of custody” for the data in the file. A SLiM provenance entry is an ASCII string, with no terminating NULL (the end of the string is dictated by the length of the entry itself, as tracked by kastore). A typical provenance table entry from SLiM 3.0 (file\_version of “0.1”) looked like:

```
{
  "program": "SLiM", "version": "3.0", "file_version": "0.1",
  "model_type": "WF", "generation": 1000, "remembered_node_count": 0}
}
```

In SLiM 3.1 this was extended to include more information (file\_version of “0.2”). The new provenance table entry format provided a superset of the information in file\_version “0.1” format, but some keys were renamed or moved within the JSON hierarchy. Such changes have continued over time; as of SLiM 4.0 (file\_version “0.8”), a provenance entry looks like this (with a few ellipses for long lines):

```
{
  "environment": {
    "os": {
      "machine": "x86_64",
      "node": "lanois.local",
      "release": "19.6.0",
      "system": "Darwin",
      "version": "Darwin Kernel Version 19.6.0: Tue Feb 15 21:39:11 PST 2022; ..."
    }
  },
  "metadata": {
    "individuals": {
      "flags": {
        "16": {
          "description": "the individual was alive at the time the file was written",
          "name": "SLIM_TSK_INDIVIDUAL_ALIVE"
        },
        "17": {
          "description": "the individual was requested by the user to be...",
          "name": "SLIM_TSK_INDIVIDUAL_REMEMBERED"
        },
        "18": {
          "description": "the individual was requested by the user to be...",
          "name": "SLIM_TSK_INDIVIDUAL_RETAINED"
        }
      }
    }
  },
  "parameters": {
    "command": [],
    "model": "initialize() {\n\tinitializeTreeSeq();\n\tinitializeMutationRate(1e-7);\n\tinitializeMutationType(\"m1\", 0.5, \"f\", 0.0);\n\tinitializeGenomicElementType(\"g1\", m1,\n1.0);\n\tinitializeGenomicElement(g1, 0, 99999);\n\tinitializeRecombinationRate(1e-8);\n}\n\nearly() {\n\tsim.addSubpop(\"p1\", 500);\n}\n\n2000 late() { sim.treeSeqOutput(\"~/Desktop/\njunk.trees\"); }\n",
    "model_hash": "eb8f55ebb263a3a42c20aed3d3cb23d44e63ad80365a252408a7126d402c1d6e",
    "model_type": "WF",
    "nucleotide_based": false,
    "seed": 2297719191923,
    "separate_sexes": false,
    "spatial_dimensionality": "",
    "spatial_periodicity": "",
    "stage": "late"
  },
  "schema_version": "1.0.0",
  "slim": {
    "cycle": 2000,
    "file_version": "0.8",
    "name": "sim",
    "tick": 2000
  }
}
```

```

    "software": {
      "name": "SLiM",
      "version": "4.0"
    }
  }
}

```

These provenance strings are JSON strings (<https://www.json.org>). For SLiM 3.0, the keys must be provided in exactly the order given in the `file_version` "0.1" example above, including the exact positions of spaces and punctuation. For SLiM 3.1 and later, any JSON-compliant string supplying the expected keys will work, as a proper JSON reader has been incorporated into SLiM.

The top-level keys have the following meanings:

**environment:** This has information about the environment in which the simulation was executed. Right now it has an `os` key under it, with `machine`, `node`, `release`, `system`, and `version` keys under that, providing information vended by the POSIX function `uname()`. These keys are for diagnostic purposes, and are not particularly standardized, and should not be relied upon by reading software.

**metadata:** This has information about the metadata annotations used in the file. Right now it describes only the three bits that are used by SLiM in the `flags` column of the `individuals` table (*not* the `flags` field inside the metadata record for each individual). Further entries may be added describing all the metadata documented above.

**parameters:** This provides information that would be necessary to recreate the model run. The `command` key provides the command-line parameters (beginning with `slim` itself) that were used to run the model; for models run in SLiMgui this will be an empty array. The `model` key provides the script that was run by SLiM to generate the saved file; this is archived in the `.trees` file for easier identification and reproducibility of runs. Note that other files that may be sourced or read by the script are not archived; for full reproducibility it would be necessary to archive such auxiliary files alongside the `.trees` file. The script is a JSON-quoted string, so it should be unJSON-quoted to reproduce the original script. In SLiM 3.4 and later (`file_version` "0.4" and later), a `model_hash` key will always be present, providing a SHA-256 hash of the script, and the `model` key may optionally be omitted; see below for details. The `file_version` key gives the version of the SLiM annotations in the file (provenance and metadata). The `model_type` key should be either "WF" or "nonWF", depending upon the type of model that generated the file; this is the same as the top-level metadata key, provided here also since it is relevant to provenance. Finally, the `seed` key provides the original random number generator seed. If a seed is supplied at the command line with the `-s` option, that will be given here. Otherwise, the seed will be a randomly generated seed provided by SLiM. Note that this is only the original seed; if the seed is set in script using `setSeed()`, that will not be reflected here (but might be reflected in the script supplied by the `model` key).

**schema\_version:** The version of the JSON schema used. The schema for provenance entries is documented at <https://tskit.dev/tskit/docs/stable/provenance.html>. Note that the schema is fairly minimal; most of the information emitted by SLiM is not described.

**slim:** This provides SLiM-specific information, but is not used by default; it retains a record of the history of a simulation (e.g., one that has multiple stages of SLiM simulation), and can be used to recover (some of) the top-level metadata if the top-level metadata is somehow lost. The `file_version`, `cycle`, `tick`, and `name` keys provide

the same information as those keys in the top-level metadata (and note that prior to SLiM 4 the `tick` and `name` keys were not present and the `cycle` key was named `generation`, as in the top-level metadata, as discussed above). There was a key named `remembered_node_count` key in `file_version "0.1"` which specified how many rows at the top of the nodes table were “remembered” by the user with the `treeSeqRememberIndividuals()` method; in `file_version "0.2"` and later, remembered individuals and genomes are controlled by the flags set on the rows of the individual table.

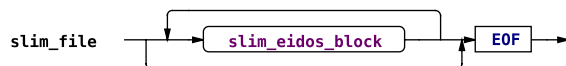
**software:** This provides information about the software program that produced this provenance entry. The `name` key gives the name of the software, and the `version` key gives the version number of the software. Note that in `file_version "0.1"` these were top-level keys instead, and the `name` key was called `program`, as illustrated by the `file_version "0.1"` example earlier.

The hope is that this existing provenance information will now be fairly stable, although further fields may be added.

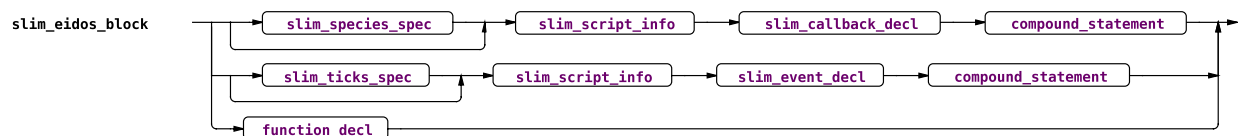
## 27. SLiM extensions to the Eidos language

### 27.1 Extensions to the Eidos grammar

The Eidos language itself is defined by the grammar given in the Eidos manual. However, SLiM defines the grammar of a *SLiM input file*, which defines a small extension to the Eidos language – in particular, by using a different start rule than Eidos’s interpreter normally uses:

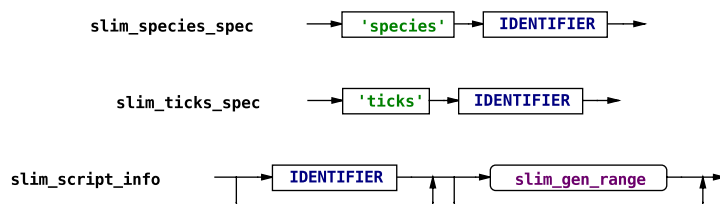


This start rule defines a SLiM input file as a series of zero or more *SLiM Eidos blocks*:



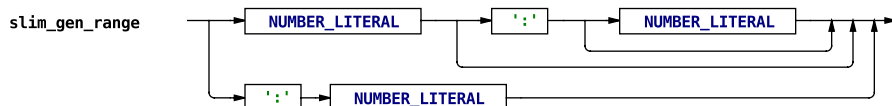
As shown above, each `slim_eidos_block` is a callback, an event, or a user-defined function; callbacks and events can be preceded by some informational tokens. Note that ordinary Eidos statements are *not* allowed at the top level of the SLiM input file; they must be within the body of a `slim_eidos_block`. The SLiM input file therefore structures Eidos statements into encapsulated blocks: *events* and *callbacks* (as well as, perhaps, user-defined functions).

In the definition of a `slim_eidos_block`, there are a couple of optional components:



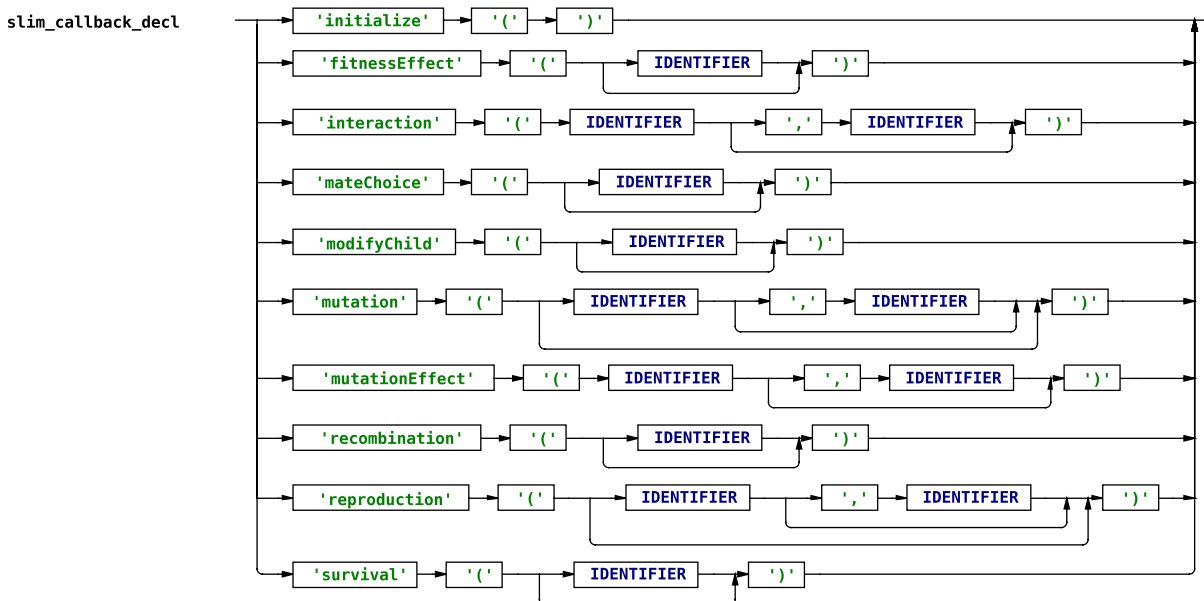
The `slim_species_spec`, called a *species specifier*, is used in multispecies models to indicate the species to which a callback block applies; in single-species models it must be omitted. Similarly, the `slim_ticks_spec`, called a *ticks specifier*, is used in multispecies models to indicate a species to which an event block should be synchronized (executing only in ticks for which that species is active); in single-species models it must be omitted.

The `slim_script_info` begins with an optional identifier that can be used to later identify the script block programmatically. If supplied, it should be an identifier like "s1", or more generally, "sX" where X is an integer greater than or equal to 0. The rest of the informational section comprises an optional tick or range of ticks in which the script block will be used by SLiM:

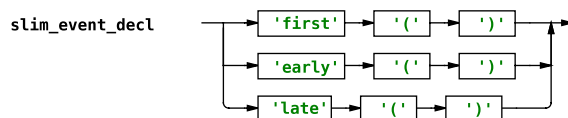


The tick numbers are defined syntactically by the grammar as numeric literals, but semantically, there are further restrictions (see section 25.1). The railroad diagram above describes that the tick range may take one of several forms: `NUMBER_LITERAL`, `NUMBER_LITERAL :`, `NUMBER_LITERAL :NUMBER_LITERAL`, or `:NUMBER_LITERAL`. See section 25.1 for discussion of these forms.

After these optional components of the informational section comes either a SLiM callback declaration section, declaring an *Eidos* callback (`initialize()`, `mutationEffect()`, `fitnessEffect()`, `interaction()`, `mateChoice()`, `modifyChild()`, `mutation()`, `recombination()`, `reproduction()`, or `survival()`):



or a SLiM event declaration section, declaring an *Eidos* event (`first()`, `early()`, or `late()`):



or the block is a user-defined function, the grammar for which is defined in the Eidos language (see the Eidos manual for details).

For Eidos callbacks, there may be required or optional identifier tokens following the callback type declaration (in parentheses), as shown above; these identifiers specify restrictions on the circumstances in which the callback will be used by SLiM. See chapter 25 for further details regarding the possible restrictions supported by each callback type.

In all other respects the grammar of Eidos is unmodified in SLiM.

## 27.2 SLiM scoping rules

The term “scope”, in computer science, refers to the semantic rules about where defined variables are visible, and how long defined variables continue to exist. Languages vary widely in their scoping rules.

Eidos is not a strongly scoped language; variables do not exist until they are defined, and once defined they exist in the *global scope*, and continue to exist forever (unless/until they are subsequently undefined with `rm()`). There is no *expression scope* or *block scope*; variables do not become undefined at the end of the expression or block in which they were defined. The only exception to this, in pure Eidos code, is that user-defined functions run within their own private scope; in other words, Eidos does employ *function scope*. This means that when code inside a user-defined function is executing, variables in the global scope are still visible (unless they are



masked by a local variable with the same name), but assignment occurs in the local scope unless `defineConstant()` or `defineGlobal()` are used to explicitly put a constant or variable into the global scope. (This is true even if a given variable has already been defined in the global scope; assigning to that variable name with operator `=` inside a user-defined function will create a new local variable with the same name, masking the global variable!) Local variables defined inside a function's private scope will go out of scope, and thus cease to exist, when the function returns. Finally, if one function calls out to a second function, the local variables of the first function are not visible to the code of the second function (in other words, function scoping is *lexical*, not *dynamic*). (See the Eidos manual for further discussion of scope in Eidos.)

SLiM follows these scoping rules, in general, but there are two twists. The first twist is that SLiM events and callbacks are treated much like Eidos user-defined functions, in that they have their own private scope. This is desirable, in general, because it prevents the global namespace from becoming cluttered up with all of the local, temporary variables defined by particular events and callbacks. It also provides a clean way for SLiM to, in effect, pass values in to callbacks (so-called “pseudo-parameters”), as described in chapter 25. For all intents and purposes, then, SLiM events and callbacks can actually be thought of as a special kind of user-defined function that is called automatically by SLiM, with a non-standard syntax for their declaration.

The second twist in SLiM is that the global scope is de-emphasized. It is not possible to write top-level Eidos code in a SLiM script that gets executed in the global scope; *all* SLiM code exists inside an event or callback, and therefore *all* SLiM code gets executed inside a private scope. However, the global scope does still exist, and can be used, as described below. Again, this design is actually advantageous, because it keeps the global namespace free of clutter, and automatically disposes of the context-specific variables used in events and callbacks.

Two kinds of entities can live in the global scope (and this is true in pure Eidos as well as in SLiM): *constants* and *variables*. Constants can be defined in the global scope with the `defineConstant()` function; there is no way to define a constant with local scope. Variables can be defined in the global scope with the `defineGlobal()` function (and in the local scope with operator `=`). The most obvious difference between constants and variables is that variables can be redefined with a new value, whereas constants cannot. This is primarily a safety measure; using a constant ensures that the value will be what you intended it to be, because it cannot be replaced accidentally. The other important difference is that global variables can be *masked*; a local variable with the same name can hide the global variable, which can cause difficult-to-find bugs if you intend your code to be referencing the global variable instead of the local variable. Constants, on the other hand, cannot be masked, ever; if a global constant has been defined, it is illegal for a variable to be defined, in any scope, with the same name. Again, this is a safety measure; it ensures that when you use the name of a constant you will always get the value you intend, because masking of constants is impossible.

When writing SLiM code, you will usually want your values to live in the local scope, as variables do by default. Sometimes, however, you will want to keep a value around beyond the lifetime of the local scope, for future reference. Given the scoping rules above, there are a number of strategies for this:

- You could define a **global constant** with `defineConstant()`. This is the best approach if you want the value to be permanent and unchanging, and is a common way to handle model parameters.
- You could define a **global variable** with `defineGlobal()`. This is often the best approach for a value that you want to exist more or less forever, but that might change over time –

especially if the value is not conceptually associated with a particular SLiM object. Note that you will need to update the value of such a global variable by calling `defineGlobal()` again; if you try to just assign into the variable after defining it, you will create a new variable with the same name as the global, in the local scope. It is perfectly legal to “mask” a global variable with a local variable of the same name, but it is usually not what you want.

- You could put the value into a **tag property** on a SLiM object. A property named `tag` exists on many of SLiM’s classes, which can hold singleton integer values persistently (and the `tagF` property on `Individual` can hold singleton float values, as well). If you want to keep a single number associated with an object (something about the state of an individual, a genome, or a mutation, for example), the `tag` property is a good way to do it.
- You could put the value into a **SLiM object’s dictionary**. Most SLiM classes inherit from the Eidos class `Dictionary` (documented in the Eidos manual) and thus provide a key-value dictionary that you can use to attach arbitrary values to them. You attach a value with the `setValue()` method, and fetch values back with the `getValue()` method, using a string key to look up the value you want. This mechanism is much more flexible than `tag` and `tagF`; values of any type can be kept, and the values don’t have to be singletons. Indeed, you could even attach a whole `Dictionary` object to a SLiM object using `setValue()`.
- You could put the value into a **global dictionary**. You can use the Eidos class `Dictionary` for a variety of purposes. One that can be convenient is to define a global variable that contains a dictionary, with a call like `defineGlobal("D", Dictionary());`. Now you can throw key-value pairs into that dictionary, using the `Dictionary` methods `setValue()` and `getValue()`, keeping your global namespace free of clutter and organizing your data by string keys. You can even nest `Dictionary()` objects to create tree structures if you wish, which might be useful for storing complex data.

One thing that it is impossible to do in SLiM is to persistently keep *most* values of object type (see below for the exceptions to this rule). *Most* values of object type cannot be set as named values with `setValue()`, nor can they be defined as global constants with `defineConstant()`, or as global variables with `defineGlobal()`. The object values that are persistent, in SLiM, are *mostly* those set up by SLiM itself, such as `sim`. This is deliberate and necessary, because otherwise stale references to objects that no longer exist in the SLiM simulation could persist in variables or constants. Since no persistent reference to most objects can be created, stale references can never exist in a SLiM model, which greatly simplifies object lifetime management and error-checking issues. Most SLiM objects are only disposed of in between calls out to Eidos, at points in time when no Eidos reference to them can possibly exist because of the design of SLiM’s scoping and persistence rules. This does prove inconvenient when one wants to keep an object value persistently; in such cases, the object generally needs to be looked up when it is needed, by its `id` or some other property. The Eidos manual has further discussion regarding this philosophy.

However, in SLiM 3.5 (Eidos 2.5) this policy has been relaxed somewhat. It is now possible to keep references to Eidos and SLiM objects, with `defineConstant()`, `defineGlobal()`, or `setValue()`, *if* they follow an internal memory-management strategy called “retain-release”. The only SLiM classes that follow this strategy, at present, are `Chromosome`, `Mutation`, and `Substitution`; but this means that you can now keep permanent references to objects of those

classes, which can be quite convenient for keeping track of a sweep mutation you have introduced, for example (see section 9.9). It is important to understand the consequences of this, however; in particular, this means that you can keep mutation objects past their normal lifetime in SLiM, after they have fixed (and been converted to a `Substitution` object), or have been lost. If you want to know whether a mutation that you have held onto is still actually segregating in the population, or has been fixed or lost, there are now properties on `Mutation` that provide this information (see section 24.10.1).

## 28. SLiM reference sheet

This reference sheet may be downloaded as a separate PDF from <http://messerlab.org/slim/>.

### SLiM Reference Sheet

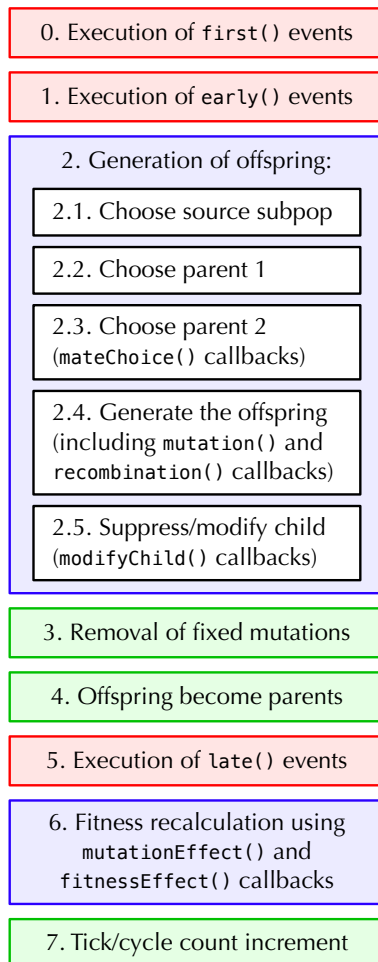
14 September 2022

```
slim -version | -usage | -help | -testEidos | -testSLiM |
[-seed <seed>] [-time] [-mem] [-Memhist] [-long [<L>]] [-x] [-define <def>] <script file>
```

**Types:** **N:**NULL, **l:**logical, **i:**integer, **f:**float, **n:**numeric, **s:**string, **o<X>:**object of class X

(ret-type)functionName(params) { ... }	user-defined function
[id] initialize() { ... }	initialize() callback
[id] [t1 [: t2]] first()   early()   late() { ... }	Eidos events
[id] [t1 [: t2]] fitnessEffect(<subpopId>) { ... }	fitnessEffect() callback
[id] [t1 [: t2]] interaction(<intTypeId> [, <subpopId>]) { ... }	interaction() callback
[id] [t1 [: t2]] mateChoice(<subpopId>) { ... } (WF)	mateChoice() callback
[id] [t1 [: t2]] modifyChild(<subpopId>) { ... }	modifyChild() callback
[id] [t1 [: t2]] mutationEffect(<mutTypeId> [, <subpopId>]) { ... }	mutationEffect() callback
[id] [t1 [: t2]] recombination(<subpopId>) { ... }	recombination() callback
[id] [t1 [: t2]] mutation(<mutTypeId> [, <subpopId>]) { ... }	mutation() callback
[id] [t1 [: t2]] reproduction(<subpopId> [, <sex>]) { ... } (nonWF)	reproduction() callback
[id] [t1 [: t2]] survival(<subpopId>) { ... } (nonWF)	survival() callback

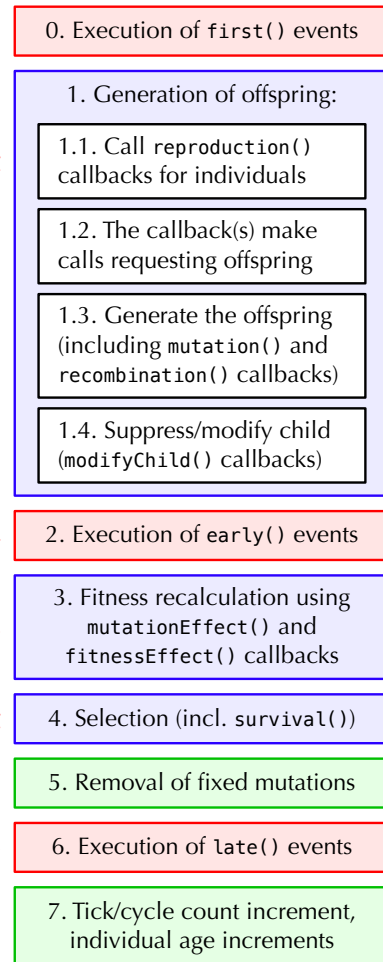
#### The WF model's tick cycle



#### Multispecies execution:

- definition order
- species declaration order
- undefined order

#### The nonWF model's tick cycle



\* : cycle stage present in only one of the model types

### Initialization functions (callable only from initialize() callbacks):

```
(integer$)initializeAncestralNucleotides(is sequence) (nuc)
(void)initializeGeneConversion(n$ nonCrossoverFraction, n$ meanLength,
    n$ simpleConversionFraction, [n$ bias])
(o<GElement>)initializeGenomicElement(io<GEType> genomicElementType, i start, i end)
(o<GEType>)$initializeGenomicElementType(is$ id, io<MutType> mutationTypes, n proportions,
    [Nf mutationMatrix])
(void)initializeHotspotMap(numeric multipliers, [Ni ends], [s$ sex]) (nuc)
(o<IntType>)$initializeInteractionType(is$ id, s$ spatiality, [l$ reciprocal],
    [n$ maxDistance], [s$ sexSegregation])
(void)initializeMutationRate(n rates, [Ni ends], [s$ sex])
(o<MutType>)$initializeMutationType(is$ id, n$ dominanceCoeff, s$ distributionType, ...)
(o<MutType>)$initializeMutationTypeNuc(is$ id, n$ dominanceCoeff, s$ distributType, ...) (nuc)
(void)initializeRecombinationRate(n rates, [Ni ends], [s$ sex])
(void)initializeSex(s$ chromosomeType)
(void)initializeSLiMModelType(s$ modelType)
(void)initializeSLiMOptions([l$ keepPedigrees], [s$ dimensionality], [s$ periodicity],
    [i$ mutationRuns], [l$ preventIncidentalSelfing], [l$ nucleotideBased],
    [l$ randomizeCallbacks])
(void)initializeSpecies([i$ tickModulo], [i$ tickPhase], [s$ avatar], [s$ color])
(void)initializeTreeSeq([l$ recordMutations], [Nif$ simplificationRatio],
    [Ni$ simplificationInterval], [l$ checkCoalescence], [l$ runCrosschecks],
    [l$ retainCoalescentOnly], [Ns$ timeUnit])
```

### SLiM callbacks:

#### mutationEffect():

```
mut (o<Mutation>$)
homozygous (l$)
effect (f$)
individual (o<Ind>$)
subpop (o<Subpop>$)
float$: mutation effect
```

#### reproduction(): (nonWF)

```
individual (o<Ind>$)
subpop (o<Subpop>$)
void: (no return)
```

#### mutation():

```
mut (o<Mutation>$)
genome (o<Genome>$)
element (o<GElement>$)
originalNuc (i$)
parent (o<Ind>$)
subpop (o<Subpop>$)
T: accept the mutation
F: reject the mutation
o<Mut>$: use a substitute
```

#### mateChoice(): (WF)

```
individual (o<Ind>$)
subpop (o<Subpop>$)
sourceSubpop (o<Subpop>$)
weights (f)
NULL: use default mate choice
float: new mating weights
float(0): no suitable mate
o<Ind>$: the chosen mate
```

#### modifyChild():

```
child (o<Ind>$)
parent1 (o<Ind>$)
isCloning (l$)
isSelfing (l$)
parent2 (o<Ind>$)
subpop (o<Subpop>$)
sourceSubpop (o<Subpop>$)
T: accept the proposed child
F: reject the proposed child
```

#### recombination():

```
individual (o<Ind>$)
genome1 (o<Genome>$)
genome2 (o<Genome>$)
subpop (o<Subpop>$)
breakpoints (i)
T: breakpoints changed
F: breakpoints unchanged
```

#### interaction():

```
distance (f$)
strength (f$)
exerter (o<Ind>$)
receiver (o<Ind>$)
float$: interaction strength
```

#### survival(): (nonWF)

```
individual (o<Ind>$)
subpop (o<Subpop>$)
surviving (l$)
fitness (f$)
draw (f$)
T: individual survives
F: individual dies
NULL: use SLiM's decision
o<Subpop>$: move to subpop
```

#### fitnessEffect():

```
individual (o<Ind>$)
subpop (o<Subpop>$)
float$: fitness effect
```

### SLiMEidosBlock (SEBlock):

Superclass: Dictionary

```
active <=> (i$)
end => (i$)
id => (i$)
source => (s$)
speciesSpec => (o<Species>$)
start => (i$)
tag <=> (i$)
ticksSpec => (o<Species>$)
type => (s$)
```

### SLiM globals:

```
community (o<Community>$)
sim, ... (o<Species>$)
g1, ... (o<GEType>$)
i1, ... (o<IntType>$)
m1, ... (o<MutType>$)
p1, ... (o<Subpop>$)
s1, ... (o<SEBlock>$)
self (o<SEBlock>$)
```

## Species:

Superclass: *Dictionary*

```
avatar => (s$)
chromosome => (o<Chromosome>$)
chromosomeType => (s$)
color => (s$)
cycle <=> (i$)
description <=> (s$)
dimensionality => (s$)
genomicElementTypes => (o<GEType>)
id => (i$)
mutationTypes => (o<MutType>)
mutations => (o<Mut>)
name => (s$)
nucleotideBased => (l$) (nuc)
periodicity => (s$)
scriptBlocks => (o<SEBlock>)
sexEnabled => (l$)
subpopulations => (o<Subpop>)
substitutions => (o<Substitution>)
tag <=> (i$)

- (o<Subpop>$)addSubpop(is$ subpopID, i$ size, [f$ sexRatio], [l$ haploid])
- (o<Subpop>$)addSubpopSplit(is$ subpopID, i$ size, io<Subpop>$ sourceSubpop,
    [f$ sexRatio]) (WF)
- (i$)countOfMutationsOfType(io<MutType>$ mutType)
- (o<Ind>)individualsWithPedigreeIDs(i pedigreeIDs, [Nio<Subpop> subpops])
- (void)killIndividuals(o<Individual> individuals) (nonWF)
- (i)mutationCounts(Nio<Subpop> subpops, [No<Mut> mutations])
- (f)mutationFrequencies(Nio<Subpop> subpops, [No<Mut> mutations])
- (o<Mut>)mutationsOfType(io<MutType>$ mutType)
- (void)outputFixedMutations([Ns$ filePath], [l$ append])
- (void)outputFull([Ns$ filePath], [l$ binary], [l$ append], [l$ spatialPositions], [l$ ages],
    [l$ ancestralNucleotides], [l$ pedigreeIDs])
- (void)outputMutations(o<Mut> mutations, [Ns$ filePath], [l$ append])
- (i$)readFromPopulationFile(s$ filePath, [No$ subpopMap])
- (void)recalculateFitness([Ni$ tick])
- (o<SEBlock>$)register[FitnessEffect/MateChoice (WF)/ModifyChild/Recombination/
    Survival]Callback(Nis$ id, s$ source, [Nio<Subpop>$ subpop], [Ni$ start], [Ni$ end])
- (o<SEBlock>$)registerMutationCallback(Nis$ id, s$ source, [Nio<MutType>$ mutType],
    [Nio<Subpop>$ subpop], [Ni$ start], [Ni$ end])
- (o<SEBlock>$)registerMutationEffectCallback(Nis$ id, s$ source, io<MutType>$ mutType,
    [Nio<Subpop>$ subpop], [Ni$ start], [Ni$ end])
- (o<SEBlock>$)registerReproductionCallback(Nis$ id, s$ source, [Nio<Subpop>$ subpop],
    [Ns$ sex], [Ni$ start], [Ni$ end]) (nonWF)
- (void)simulationFinished(void)
- (void)skipTick(void)
- (o<Mut>)subsetMutations([No<Mut>$ exclude], [Nio<MutType>$ mutType], [Ni$ position],
    [Ni$ nucleotide], [Ni$ tag], [Ni$ id])
- (l$)treeSeqCoalesced(void)
- (void)treeSeqOutput(s$ path, [l$ simplify], [l$ includeModel], [No$ metadata])
- (void)treeSeqRememberIndividuals(o<Ind> individuals, [l$ permanent])
- (void)treeSeqSimplify(void)
```

Effects of mutations:

no mutation present	1
heterozygote	$1 + h*s$
homozygote	$1 + s$

$s = \text{mut.selectionCoeff}$   
 $h = \text{mutType.dominanceCoeff}$

## GenomicElement (GELEMENT):

Superclass: *Object*

```
endPosition => (i$)
genomicElementType => (o<GEType>$)
startPosition => (i$)
tag <=> (i$)

- (void)setGenomicElementType(io<GEType>$ genomicElementType)
```



## Subpopulation (Subpop):

Superclass: *Dictionary*

```
cloningRate => (f) (WF)
description <=> (s$)
firstMaleIndex => (i$)
fitnessScaling <=> (f$)
genomes => (o<Genome>)
genomesNonNull => (o<Genome>)
id => (i$)
immigrantSubpopFractions => (f) (WF)
immigrantSubpopIDs => (i) (WF)
individualCount => (i$)
individuals => (o<Ind>)
lifetimeReproductiveOutput => (i$)
lifetimeReproductiveOutputF => (i$)
lifetimeReproductiveOutputM => (i$)
name <=> (s$)
selfingRate => (f$) (WF)
sexRatio => (f$) (WF)
spatialBounds => (f)
species => (o<Species>$)
tag <=> (i$)

- (No<Ind>$)addCloned(o<Ind>$ parent) (nonWF)
- (No<Ind>$)addCrossed(o<Ind>$ parent1, o<Ind>$ parent2, [Nfs$ sex]) (nonWF)
- (No<Ind>$)addEmpty([Nfs$ sex], [Nl$ genome1Null], [Nl$ genome2Null]) (nonWF)
- (No<Ind>$)addRecombinant(No<Genome>$ strand1, No<Genome>$ strand2, Ni breaks1,
    No<Genome>$ strand3, No<Genome>$ strand4, Ni breaks2, [Nfs$ sex]) (nonWF)
- (No<Ind>$)addSelfed(o<Ind>$ parent) (nonWF)
- (f)cachedFitness(Ni indices)
- (void)configureDisplay([Nf center], [Nf scale], [Ns$ color])
- (void)defineSpatialMap(s$ name, s$ spatiality, n values, [l$ interpolate],
    [Nif valueRange], [Ns colors])
- (void)outputMSSample(i$ sampleSize, [l$ replace], [s$ requestedSex],
    [Ns$ filePath], [l$ append], [l$ filterMonomorphic])
- (void)outputSample(i$ sampleSize, [l$ replace], [s$ requestedSex],
    [Ns$ filePath], [l$ append])
- (void)outputVCFSample(i$ sampleSize, [l$ replace], [s$ requestedSex],
    [l$ outputMultiallelics], [Ns$ filePath], [l$ append])
- (l)pointInBounds(f point)
- (f)point[Periodic|Reflected|Stopped](f point)
- (f)pointUniform([i$ n])
- (void)removeSubpopulation(void) (nonWF)
- (void)sampleIndividuals(i$ size, [l$ replace], [No<Ind>$ exclude], [Ns$ sex], [Ni$ tag],
    [Ni$ minAge], [Ni$ maxAge], [Nl$ migrant])
- (void)setCloningRate(n rate) (WF)
- (void)setMigrationRates(io<Subpop> sourceSubpops, n rates) (WF)
- (void)setSelfingRate(n$ rate) (WF)
- (void)setSexRatio(f$ sexRatio) (WF)
- (void)setSpatialBounds(n bounds)
- (void)setSubpopulationSize(i$ size) (WF)
- (s)spatialMapColor(s$ name, n value)
- (o<Image>$) spatialMapImage(s$ name, [Ni$ width], [Ni$ height], [l$ centers], [l$ color])
- (f)spatialMapValue(s$ name, f point)
- (o<Ind>)subsetIndividuals([No<Ind>$ exclude], [Ns$ sex], [Ni$ tag], [Ni$ minAge],
    [Ni$ maxAge], [Nl$ migrant])
- (void)takeMigrants(o<Ind> migrants) (nonWF)
```

## SLiMgui: (in SLiMgui only)

Superclass: *Dictionary*

```
pid => (i$)

- (void)openDocument(s$ filePath)
- (void)pauseExecution(void)
```

SLiMgui quick help:  
opt-click on keyword

Code completion:  
escape (esc)

## Individual (Ind):

Superclass: Dictionary

```
age <=> (i$) (nonWF)
color <=> (s$)
fitnessScaling <=> (f$)
genomes => (o<Genome>)
genomesNonNull => (o<Genome>)
genome1 => (o<Genome>$(
genome2 => (o<Genome>$(
index => (i$)
migrant => (l$)
pedigreeID => (i$)
pedigreeParentIDs => (i)
pedigreeGrandparentIDs => (i)
reproductiveOutput => (i$)
sex => (s$)
spatialPosition => (f)
subpopulation => (o<Subpop>$(
tag <=> (i$)
tagF <=> (f$)
uniqueMutations => (o<Mut>)
x <=> (f$)
y <=> (f$)
z <=> (f$)

- (l)containsMutations(o<Mut> mutations)
- (i$)countOfMutationsOfType(io<MutType>$ mutType)
- (f)relatedness(o<Ind> individuals)
+ (void)setSpatialPosition(f position)
- (f$)sumOfMutationsOfType(io<MutType>$ mutType)
- (o<Mut>)uniqueMutationsOfType(io<MutType>$ mutType)
```

## Chromosome:

Superclass: Dictionary

```
colorSubstitution <=> (s$)
geneConversionEnabled => (l$)
geneConversionGCBias => (f$)
geneConversionNonCrossoverFraction => (f$)
geneConversionMeanLength => (f$)
geneConversionSimpleConversionFraction => (f$)
genomicElements => (o<GElement>)
hotspotEndPositions[F|M] => (i) (nuc)
hotspotMultipliers[F|M] => (f) (nuc)
lastPosition => (i$)
mutationEndPositions[F|M] => (i)
mutationRates[F|M] => (f)
overallMutationRate[F|M] => (f$)
overallRecombinationRate[F|M] => (f$)
recombinationEndPositions[F|M] => (i)
recombinationRates[F|M] => (f)
species => (o<Species>$(
tag <=> (i$)

- (is)ancestralNucleotides([Ni$ start], [Ni$ end], [s$ format]) (nuc)
- (integer)drawBreakpoints([No<Ind>$ parent], [Ni$ n])
- (integer$)setAncestralNucleotides(is sequence) (nuc)
- (void)setGeneConversion(n$ nonCrossoverFraction, n$ meanLength,
  n$ simpleConversionFraction, [n$ bias])
- (void)setHotspotMap(n multipliers, [Ni ends], [s$ sex]) (nuc)
- (void)setMutationRate(n rates, [Ni ends], [s$ sex])
- (void)setRecombinationRate(n rates, [Ni ends], [s$ sex])
```

## Mutation (Mut):

Superclass: Dictionary

```
id => (i$)
isFixed => (l$)
isSegregating => (l$)
mutationType => (o<MutType>$(
nucleotide <=> (s$) (nuc)
nucleotideValue <=> (i$) (nuc)
originTick => (i$)
position => (i$)
selectionCoeff => (f$)
subpopID <=> (i$)
tag <=> (i$)

- (void)setMutationType(io<MutType>$ mutType)
- (void)setSelectionCoeff(f$ selectionCoeff)
```

## Substitution:

Superclass: Dictionary

```
id => (i$)
fixationTick => (i$)
mutationType => (o<MutType>$(
nucleotide <=> (s$) (nuc)
nucleotideValue <=> (i$) (nuc)
originTick => (i$)
position => (i$)
selectionCoeff => (f$)
subpopID <=> (i$)
tag <=> (i$)
```

## MutationType (MutType):

Superclass: Dictionary

```
color <=> (s$)
colorSubstitution <=> (s$)
convertToSubstitution <=> (l$)
distributionParams => (f)
distributionType => (s$)
dominanceCoeff <=> (f$)
id => (i$)
mutationStackGroup <=> (i$)
mutationStackPolicy <=> (s$)
nucleotideBased => (l$) (nuc)
species => (o<Species>$(
tag <=> (i$)

- (float)drawSelectionCoefficient(i$ n)
- (void)setDistribution(s$ distType, ...)
```

## Genome:

Superclass: *Object*

```
genomePedigreeID => (i$)
genomeType => (s$)
individual => (o<Ind>$)
isNullGenome => (l$)
mutations => (o<Mut>)
tag <=> (i$)

+ (void)addMutations(o<Mut> mutations)
+ (o<Mut>)addNewDrawnMutation(io<MutType> mutationType, i position,
    [Nio<Subpop> originSubpop], [Nis nucleotide])
+ (o<Mut>)addNewMutation(io<MutType> mutationType, n selectionCoeff, i position,
    [Nio<Subpop> originSubpop], [Nis nucleotide])
- (l$)containsMarkerMutation(io<MutType>$ mutType, i$ position, [l$ returnMutation])
- (l$)containsMutations(o<Mut> mutations)
- (i$)countOfMutationsOfType(io<MutType>$ mutType)
+ (i)mutationCountsInGenomes(No<Mut> mutations)
+ (f)mutationFrequenciesInGenomes(No<Mut> mutations)
- (o<Mut>)mutationsOfType(io<MutType>$ mutType)
- (is)nucleotides([Ni$ start], [Ni$ end], [s$ format])
+ (void)output([Ns$ filePath], [l$ append])
+ (void)outputMS([Ns$ filePath], [l$ append], [l$ filterMonomorphic])
+ (void)outputVCF([Ns$ filePath], [l$ outputMultiallelics], [l$ append])
- (i)positionsOfMutationsOfType(io<MutType>$ mutType)
+ (o<Mut>)readFromMS(s$ filePath, io<MutType>$ mutationType)
+ (o<Mut>)readFromVCF(s$ filePath, [Nio<MutType>$ mutationType])
+ (void)removeMutations([No<Mut> mutations], [l$ substitute])
- (f$)sumOfMutationsOfType(io<MutType>$ mutType)
```

## InteractionType (IntType):

Superclass: *Dictionary*

```
id => (i$)
maxDistance <=> (f$)
reciprocal => (l$)
sexSegregation => (s$)
spatiality => (s$)
tag <=> (i$)

- (f)clippedIntegral(No<Ind> individuals)
- (f)distance(o<Ind>$ receiver, [No<Ind> exerters])
- (f)distanceFromPoint(f point, o<Ind> exerters)
- (o<Ind>)drawByStrength(o<Ind>$ receiver, [i$ count], [No<Subpop>$ exorterSubpop])
- (void)evaluate(io<Subpop> subpops)
- (i)interactingNeighborCount(o<Ind> receivers, [No<Subpop>$ exorterSubpop])
- (f)interactionDistance(o<Ind> receiver, [No<Ind> exerters])
- (f)localPopulationDensity(o<Ind> receivers, [No<Subpop>$ exorterSubpop])
- (o<Ind>)nearestInteractingNeighbors(o<Ind>$ receiver, [i$ count],
    [No<Subpop>$ exorterSubpop])
- (o<Ind>)nearestNeighbors(o<Ind>$ receiver, [i$ count], [No<Subpop>$ exorterSubpop])
- (o<Ind>)nearestNeighborsOfPoint(f point, io<Subpop>$ exorterSubpop, [i$ count])
- (i)neighborCount(o<Ind> receivers, [No<Subpop>$ exorterSubpop])
- (i)neighborCountOfPoint(o<Ind> receivers, [No<Subpop>$ exorterSubpop])
- (void)setInteractionFunction(s$ functionType, ...)
- (f)strength(o<Ind>$ receiver, [No<Ind> exerters])
- (f)totalOfNeighborStrengths(o<Ind> receivers, [No<Subpop>$ exorterSubpop])
- (void)unevaluate(void)
```

### GenomicElementType (GEType):

Superclass: *Dictionary*

```
color <=> (s$)
id => (i$)
mutationFractions => (f)
mutationMatrix => (f) (nuc)
mutationTypes => (o<MutType>)
species => (o<Species>$)
tag <=> (i$)

- (void)setMutationFractions(io<MutType> mutationTypes, n proportions)
- (void)setMutationMatrix(f mutationMatrix) (nuc)
```

### LogFile:

Superclass: *Dictionary*

```
filePath => (s$)
logInterval => (i$)
precision <=> (i$)
tag <=> (i$)

- (void)addCustomColumn(s$ columnName, s$ source, [* context])
- (void)addCycle(void)
- (void)addCycleStage(void)
- (void)addMeanSDColumns(s$ columnName, s$ source, [* context])
- (void)addPopulationSexRatio(void)
- (void)addPopulationSize(void)
- (void)addSubpopulationSexRatio(io<Subpop>$ subpop)
- (void)addSubpopulationSize(io<Subpop>$ subpop)
- (void)addSuppliedColumn(s$ columnName)
- (void)flush(void)
- (void)logRow(void)
- (void)setFilePath(s$ filePath, [Ns initialContents], [l$ append], [Nl$ compress], [Ns$ sep])
- (void)setLogInterval([Ni$ logInterval])
- (void)setSuppliedValue(s$ columnName, +$ value)
- (void)willAutolog(void)
```

### SLiM population-genetics utilities:

```
- (f$)calcFST(o<Genome> genomes1, o<Genome> genomes2, [No<Mut> muts], [Ni$ start], [Ni$ end])
- (f$)calcHeterozygosity(o<Genome> genomes, [No<Mut> muts], [Ni$ start], [Ni$ end])
- (f$)calcInbreedingLoad(o<Genome> genomes, [No<MutType> mutType])
- (f$)calcPairHeterozygosity(o<Genome>$ genome1, o<Genome>$ genome2, [Ni$ start], [Ni$ end],
    [l$ infiniteSites])
- (f$)calcVA(o<Ind> individuals, io<MutType>$ mutType)
- (f$)calcWattersonsTheta(o<Genome> genomes, [No<Mut> muts], [Ni$ start], [Ni$ end])
```

### SLiM nucleotide-based utilities:

```
- (s)codonsToAminoAcids(i codons, [li$ long], [l$ paste]) (nuc)
- (is)codonsToNucleotides(i codons, [s$ format]) (nuc)
- (f)mm16To256(f mutationMatrix) (nuc)
- (f)mmJukesCantor(f$ alpha) (nuc)
- (f)mmKimura(f$ alpha, f$ beta) (nuc)
- (i)nucleotideCounts(is sequence) (nuc)
- (f)nucleotideFrequencies(is sequence) (nuc)
- (i)nucleotidesToCodons(is sequence) (nuc)
- (is)randomNucleotides(i$ length, [Nif basis], [s$ format]) (nuc)
```

### SLiM other utilities:

```
- (f)summarizeIndividuals(o<Ind> individuals, i dim, n spatialBounds, s$ operation,
    [Nlif$ empty], [l$ perUnitArea], [Ns$ spatiality])
- (o<Dictionary>)treeSeqMetadata(s$ filePath, [l$ userData])
```

## Community:

Superclass: *Dictionary*

```
allGenomicElementTypes => (o<GEType>)
allInteractionTypes => (o<IntType>)
allMutationTypes => (o<MutType>)
allScriptBlocks => (o<SEBlock>)
allSpecies => (o<Species>)
allSubpopulations => (o<Subpop>)
cycleStage <=> (s$)
logFiles => (o<LogFile>)
modelType => (s$)
tag <=> (i$)
tick <=> (i$)
verbosity <=> (i$)

- (o<LogFile>)$createLogFile(s$ filePath, [Ns initialContents], [l$ append], [l$ compress],
    [s$ sep], [Ni$ logInterval], [Ni$ flushInterval])
- (void)deregisterScriptBlock(io<SEBlock> scriptBlocks)
- (o<GEType>)genomicElementTypesWithIDs(i ids)
- (o<IntType>)interactionTypesWithIDs(i ids)
- (o<MutType>)mutationTypesWithIDs(i ids)
- (void)outputUsage(void)
- (o<SEBlock>)$register[First/Early/Late]Event(Nis$ id, s$ source, [Ni$ start], [Ni$ end])
- (o<SEBlock>)$registerInteractionCallback(Nis$ id, s$ source, io<IntType>$ intType,
    [Nio<Subpop>$ subpop], [Ni$ start], [Ni$ end])
- (o<SEBlock>)$rescheduleScriptBlock(io<SEBlock>$ block, [Ni$ start], [Ni$ end],
    [Ni ticks])
- (o<SEBlock>)scriptBlocksWithIDs(i ids)
- (void)simulationFinished(void)
- (o<Species>)speciesWithIDs(i ids)
- (o<Subpop>)subpopulationsWithIDs(i ids)
```

## 29. Revision history

This is a history of the revisions of SLiM since version 2.0. It focuses on new features, not bug fixes (which rapidly become ancient history), and it does not contain every detail. The `VERSIONS` file in the SLiM source code provides more detail, including changes that might not be user-visible at all. Beyond that, the commit history in GitHub is of course the definitive history of the project. For those wanting a quick overview of SLiM's history, however, this is a good place to start.

Recipe numbers have changed over time, as sections of the SLiM manual have been added and reorganized, so the numbers given here may or may not make any sense, unfortunately.

### *Version 4.0.1*

Feature: new `meanParentAge` property on `Individual` for easy generation time calculation

Feature: add `asMatrix()` method to `DataFrame`, making it easy to use `readCSV()` to get a matrix

Tweak: `DataFrame`'s `subset()` method now takes `NULL` for `row/col` to get whole columns/rows

Tweak: `readCSV()` now accepts `sep=""` to use a "whitespace" separator (as in R)

Tweak: recipe 6.1.2 now uses `readCSV()` instead of `readFile()`

Tweak: recipe 16.12 II now uses `asMatrix()` to read mating/death files more elegantly

Bug fix: error/leak when reloading tree-sequence models with `readFromPopulationFile()`

Bug fix: `LogFile` error when given an absolute filesystem path on Windows

Bug fix: error passing a zero-length vector of individuals to `treeSeqRememberIndividuals()`

Bug fix: incorrect autofix in `SLiMgui` for assignment into property (e.g., `sim.generation = 10`)

### *Version 4.0*

Feature: "autofixing" assistance in `SLiMgui`, assisting in transitioning to new versions of SLiM

Feature: numerous extensions to SLiM for simulation of multiple species

- split `SLiMSim` into `Community` and `Species` classes; `Community` has a vector of `Species`

- `Community` gets lots of new APIs for looking up objects associated with species

- add a `species` property to many SLiM Eidos classes, to get the species they belong to

- add `species` and `ticks` specifiers to SLiM's top-level syntax for event/callback declarations

- add `species all` designation for non-species-specific events and callbacks

- add an `initializeSpecies()` function to do species-level configuration

- new "species schedules": `tickModulo` and `tickPhase` parameters for `initializeSpecies()`

- add `skipTick()` method to `Species` for custom tick cycle scheduling

- add "cycle" concept to `Species`: number of times that species has executed its cycle

- change "generations" to either "ticks" or "cycles", as appropriate, across almost all APIs

- remove the `originGeneration` parameter from `addNew[Drawn]Mutation()`

- output format change for most output methods, providing both tick and cycle in header

- `SLiMgui` multispecies support: species tab bar, species avatars, species color lines, etc.

- new multispecies chapter with new recipes: chapter 19

Feature: major design shift in `InteractionType` to support between-species interactions

- interaction types are defined at the `Community` level and may be used with any/all species

- `InteractionType` APIs are now much more explicit about the "receiver" vs. "exerter"

- APIs now allow the receiver and exerter to be in different subpopulations (or species)

- `evaluate()` now requires subpops to be given explicitly, and `immediate` has been removed

- new `neighborCount()` and `neighborCountOfPoint()` methods to fill out the API

- break in backward reproducibility due to internal change in rounding behavior



greatly reduced memory usage for large models, from  $O(N^2)$  to  $O(N)$

Feature: new `killIndividuals()` method for mortality in nonWF models at (almost) any time

Feature: support “no-genetics” species (ecology only) omitting boilerplate null genetics

Feature: randomization of order of processing in tick cycle stages; optional but defaults to T

Feature: add “supplied columns” to `LogFile` for data supplied to the log by script

Feature: add `subpopMap` parameter to `readFromPopulationFile()` for remapping subpop IDs

Feature: add name and description properties to `Species` for better output annotation

Feature: add a `findInterval()` function to `Eidos`, patterned after the same function in R

Feature: new `calcInbreedingLoad()` pop-gen function, thanks to Chris Kyriazis

Feature: add the WF/nonWF tick cycle diagrams to `SLiMgui`, under the Help menu

Feature: add a “Scheduling” tab to the `SLiMgui` output viewer to see script events over time

Feature: add “Focus on Script” and “Focus on Console” keyboard shortcuts in `SLiMgui`

Feature: show the run progress as a progress indicator bar in the Ticks lineedit in `SLiMgui`

Feature: add display of the Git SHA-1 for the `slim/SLiMgui` build, in “about” info

Tweak: `fitness()` renamed to `mutationEffect()`, `fitness(NULL)` renamed to `fitnessEffect()`

Tweak: remove the `removeConstants` parameter from `rm()`; no redefining constants!

Tweak: revisions to many recipes to adapt to the above changes, `pyslim` changes, etc.

Tweak: new recipe 14.15 showing how to make a biallelic model in `SLiM`

Tweak: add a second recipe to 16.19 (nonWF range expansion) demonstrating `survival()`

Tweak: new recipe 18.15 and revised 18.14, better showing how to “unique down” mutations

Tweak: remove the deprecated `inSLiMgui` property of `SLiMSim`

Tweak: change G0 (generation of origin) to T0 (tick of origin) in VCF output

Tweak: remove obsolete and unnecessary pseudo-parameters from callbacks, streamlining APIs

Tweak: merge in `tskit` C API 1.0 final, providing improvements under the hood

Policy change: `early()` is no longer a default for events, and must be specified explicitly

Policy change: `calcFST()` now gives “ratio of averages” not “average of ratios” (best practice)

Bug fix: mutation frequency/count could (rarely) be incorrect; see `slim-discuss` announcement

Bug fix: crash in `removeMutations()` when called on a null genome

Bug fix: malformed tree-seq metadata in certain conditions after removal of a subpopulation

#### *Version 3.7.1*

Tweak: Recipe fixes for 17.8, 17.9, 18.13

Bug fix: Preview output in `SLiMgui` was unreadable in dark mode

Bug fix: Occasional self-test error due to a faulty test for `relatedness()`

Bug fix: Fix for a common and reproducible crashing bug on `treeSeqOutput()`

Bug fix: Spurious “subpopulation p0 has already been used” error in certain circumstances

#### *Version 3.7*

Feature: Windows is now supported as a platform, including `SLiMgui`, thanks to Russell Dinnage

Feature: both WF and nonWF models now have a new tick cycle stage, `first()` events

Feature: new `survival()` callbacks in nonWF models now govern mortality (and relocation)

Feature: new `DataFrame` class in `Eidos` for representing data tables, like R’s `data.frame`

Feature: add CSV/TSV serialization for `Dictionary`, and `readCSV()` to read in a `DataFrame`

Feature: `Dictionary` can now serialize to and from JSON, for easy persistence of information

Feature: better support for null genomes with `addRecombinant()`; breaks backward compatibility

Feature: other null genome work: `genomesNonNull`, `addEmpty()` flags, `addSubpop()` flag

Feature: better fitness evaluation in haploids using null genomes, with `haploidDominanceCoeff`

Feature: `summarizeIndividuals()` for creating spatial maps from individuals; new recipe 15.13

Feature: `localPopulationDensity()` and `clippedIntegral()` for local area/density calculations

Feature: add `spatialMapImage()` method to `Subpopulation` to get an `Image` for a spatial map

Feature: can now create a grayscale `Image` from a matrix of values

Feature: add `write()` method to `Image` to write out to PNG

Feature: in tree-seq, parent pedigree ids are now kept and the parent column is maintained

Feature: new name and description properties for `Subpopulation`, used for tree-seq metadata

Feature: `tskit` time unit support, `timeUnit` parameter to `initializeTreeSeq()` to set the name

Feature: new `treeSeqMetadata()` function to get the metadata from a `.trees` file, as a `Dictionary`

Feature: SLiMgui individuals view improvements including action buttons, Unified display

Feature: SLiMgui chromosome view now has an action button and a cleaner UI design

Feature: SLiMgui output viewer window now has tabs for `LogFile` and `writeFile()` output

Feature: added Eidos string functions `strcontains()`, `strfind()`, `strprefix()`, `strsuffix()`

Feature: added an `assert()` function in Eidos, to easily assert that a condition is true

Feature: add `rnbinom()` function to Eidos for the negative binomial distribution

Feature: add `grep()` function to Eidos for regular expression matching

Feature: add `tempdir()` and `sysinfo()` functions for better cross-platform support

Feature: added an `individualsWithPedigreeIDs()` SLiMSim method for fast individual lookup

Tweak: extended recipe 16.15 (WF models in nonWF) to show fitness-based mate choice

Tweak: new recipe 16.22, sperm storage with a `survival()` callback

Tweak: new recipe 16.23, tracking separate sexes in script in nonWF models

Tweak: new recipe 16.24, haplodiploidy

Tweak: revised recipes 9.2, 9.3, 9.6.2, 17.3 I/II, 16.11, 16.12, 16.13, 16.14

Tweak: add a `[logical$ chdir = F]` argument to `source()` to simplify common usage

Tweak: many calls now take an integer `id` in place of a `Subpopulation/MutationType` argument

Tweak: fix `InteractionType` to allow > 2 billion interactions in a model

Tweak: updated SLiM to `tskit` version C API 0.99.15, inheriting various improvements

Tweak: added the concept of deprecated APIs in Eidos, marked some old stuff deprecated

Tweak: remove `dominanceCoeffX` (for `haploidDominanceCoeff`) breaks backward compatibility

Tweak: allow `readFromPopulationFile()` even when tree-seq recording is not enabled

Tweak: `getRowValues()` now allows ragged rows and matrix/array elements; new `[drop=F]` arg

Tweak: remembering large numbers of individuals in tree-seq is now much faster

Tweak: limited Markdown supported in Jump menu comments, improved recipe 5.4 to show it

Policy change: `float` values that are round now output with `".0"` – e.g., `1.0` instead of `1`

Policy change: subpopulation ids can no longer be reused later in time, to prevent collisions

Policy change: `source()` now raises an error if given a non-existent path

Bug fix: completely overhauled recipe 14.4 (chromosomal inversions) fixing several bugs

Bug fix: major bugs in `relatedness()` in SLiM 3.6 fixed; do not use SLiM 3.6 `relatedness()`

Bug fix: subpopulation metadata is now preserved across load/save for untouched subpops

Bug fix: various JSON parsing bugs were fixed

Bug fix: more robust handling of memory allocation failures  
Bug fix: `reproductiveOutput` is now correct after `modifyChild()` rejects a child  
Bug fix: floating-point roundoff issue with `mutationFrequencies()` not being exactly one

#### Version 3.6

Feature: “debug points” and a debug output window in SLiMgui  
Feature: `[permanent=T]` flag for `treeSeqRememberIndividuals()` to “retain” individuals  
Feature: `[retainCoalescentOnly=T]` flag for `initializeTreeSeq()` to modify simplification  
Feature: `metadata=` optional parameter for `treeSeqOutput()`, for user-supplied metadata  
Feature: error stream support in Eidos, with a `[error=F]` parameter for `cat()/catn()/print()`  
Feature: various extensions to the Eidos Dictionary class for dataframe-like usage  
Feature: `rf()` function in Eidos for *F*-distribution draws  
Feature: `precision` property for `LogFile`, to govern the precision of floating-point output  
Feature: dark mode support for SLiMgui, for both macOS (automatic) and Linux (prefs)  
Feature: Eidos and SLiM now allow Unicode variable names (accents, Chinese, emoji, etc.)  
Feature: SLiM and Eidos manuals now have PDF “document outlines” (i.e., TOCs)  
Tweak: new recipe 16.21, “Dynamic population structure in nonWF models” (split, merge)  
Tweak: `tskit` 0.99.10, `.trees` file version 6, supporting various improvements  
Tweak: SLiMgui on macOS no longer quits on last window close (yay!)  
Tweak: `generationStage` property for `SLiMSim`, for (unusual) models that need it  
Tweak: add `-help` command-line option, make `--option` prefixes work  
Tweak: SLiMgui opens `.py` files in the user’s browser, for a better cross-platform experience  
Tweak: `debugIndent()` function in Eidos, to get the current SLiM debugging output indent level  
Tweak: the command-line verbosity level is now available as property `verbosity` of `SLiMSim`  
Policy change: the sequence `<-` is now treated as an illegal Eidos token, for safety versus R  
Bug fix: `takeMigrants()` could cause bad data or crashes, in unusual circumstances  
Bug fix: argument processing crash for function/method calls with more than 256 arguments  
Bug fix: certain recursive calling patterns in Eidos could cause a segfault  
Bug fix: `LogFile` would give an error writing out a compressed file with zero-length data  
Bug fix: SLiMgui would change the directory that `LogFile` was saving into, to its own directory  
Bug fix: incorrect reproductive output tallying for individuals moved by `takeMigrants()`  
Bug fix: memory leak after `removeSubpopulation()`, could trigger an “internal error” message  
Bug fix: spurious error about a “mismatch” with very large dominance coefficients  
Bug fix: 3.5 new regression: `source()` would give a warning and do the wrong thing  
Bug fix: output from nested events/callbacks/functions sometimes printed out of order  
Bug fix: minor SLiMgui bugs: tab stops, syntax coloring, the Jump menu, help info, status bar

#### Version 3.5

Feature: SLiMgui is now Qt-based on all platforms; the old app is renamed `SLiMguiLegacy`  
Feature: SLiMgui can now be run on Windows under the WSL (thanks Bernard Kim)  
Feature: `tabulate()` function in Eidos, parallel to R’s `tabulate()` function  
Feature: `quantile()` function in Eidos, parallel to R’s “type 7” quantile function  
Feature: `colors()` function in Eidos, providing a variety of color palettes

Feature: add new SLiMgui plots for 1D/2D SFS, 2D frequency spectrum, age distribution, population/subpop fitness distributions, population size ~ time

Feature: tskit 0.3.0 with faster simplification, retention of individuals referenced by retained nodes; recipes revised for new tskit/pyslim APIs: 17.2, 17.4, 17.5, 17.7, 17.9, 18.13

Feature: automatic tree-seq simplification is now a separate timing category in SLiMgui profiles

Feature: mutation() callbacks can now return an existing mutation, for easy uniquing

Feature: subsetMutations() method on SLiMSim for fast lookup of specific mutations

Feature: SLiMgui now has line numbers, current line highlight, Jump pop-up button, etc.

Feature: SLiMgui now allows multiple graphs of the same type to be opened

Feature: paste() / paste0() take multiple arguments to avoid paste(c()), recipes 4.2.5, 13.5, 14.8, and 16.12 (I) revised for new explicit sep= usage pattern; **breaks compatibility**

Feature: full relayout option for prettyprinting in SLiMgui, by holding down option/alt

Feature: autosave option in SLiMgui (off by default) whenever you recycle

Feature: new Image class in Eidos can read PNG images, for easy spatial map creation

Feature: defineSpatialMap() now requires a matrix (except for 1D maps), and gridSize is removed; the matrix is read in correct order (working with Image); revise recipes 15.10 and 15.11 for these changes (and a very small bug fix); **breaks compatibility**

Feature: Dictionary class in Eidos is now explicit, public, and independent; new methods

Feature: defineGlobal() function in Eidos, allows global variable definitions; scoping improved

Feature: you can keep references to some objects, including Dictionary, Image, Chromosome, & Mutation, long-term; new recipes 9.9 and 9.10 demonstrate this

Feature: new isFixed and isSegregating properties for Mutation, to diagnose state

Feature: new LogFile class allows easy logging of model statistics; new recipe 4.2.5 demos

Feature: new pop-gen stats functions calcFST(), calcVA(), calcHeterozygosity(), calcPairHeterozygosity(), calcWattersonsTheta(); revise recipes 11.1, 14.1 to use these

Feature: mutation[Counts|Frequencies]InGenomes() methods for easy sample frequency

Feature: reproductive output tracking when pedigree tracking is enabled; new properties reproductiveOutput (Individual) and lifetimeReproductiveOutput [F|M] (Subpopulation); new lifetime reproductive output plot in SLiMgui

Tweak: superclass relationships among Eidos and SLiM classes are explicit and documented

Tweak: new recipe 18.4 demonstrates uniquing of mutations to model identity by state

Tweak: new recipe 10.6 II demonstrates another technique for varying dominance coefficients

Tweak: new recipe 16.19 demonstrates range expansion into empty subpopulations

Tweak: new recipe 16.20 demonstrates logistic growth with the Beverton–Holt model

Tweak: speed up match() for large x vectors, from  $O(N \cdot M)$  to  $O(N+M)$

Tweak: speed up sample() for full shuffles and large N; **breaks reproducibility**

Tweak: faster function/method dispatch by pre-processing of argument lists

Tweak: using “Robin Hood Hashing” to speed up internal algorithms; many models benefit

Tweak: extend codonsToAminoAcids() to be able to return integer codes for amino acids

Tweak: outputFull() can now save pedigree IDs; readFromPopulationFile() can read them

Tweak: recipes 13.5 & 14.8 now emit PNG plots instead of PDF, look for Rscript path

Tweak: recipe 5.4 (Gravel model) better follows the original model (thanks Chase Nelson)

Tweak: the Step button in SLiMgui can be held down to step repeatedly

Tweak: generation play in SLiMgui is now wired to the Play button and can be halted

Tweak: the SLiMgui app icon now darkens while a model is running, for easy monitoring  
 Tweak: SLiMgui shows elapsed CPU time, # of mutations, and # of substitutions in status bar  
 Tweak: buffered compressed append date for `writeFile()` can be flushed with `flushFile()`  
 Tweak: `functionSource()` function in Eidos, to see the source code of Eidos functions  
 Tweak: `slim / eidos` (but not SLiMgui) builds for macOS are now Universal (native for M1)  
 Policy change: disallow adding mutations to individuals of age > 0 to prevent inconsistencies  
 Policy change: mutation positions, generating gametes, are unique; **breaks reproducibility**  
 Policy change: `originGeneration` can no longer be used as scratch space  
 Policy change: errors during `writeFile()` are now errors, not warnings  
 Major bug fix: treatment of NAN in Eidos functions & comparison operators to follow IEEE rules  
 Bug fix: precedence for operator `^` fixed to be higher than unary minus, following convention  
 Bug fix: recipe 14.7 now uses `L` instead of `L-1`, avoiding a minor off-by-one error

#### Version 3.4

Feature: add the QtSLiM graphical modeling environment for Linux (SLiMgui port to Qt)  
 Feature: `writeFile()` and `writeTempFile()` now write .gz-compressed data with `compress=T`  
 Feature: add `qnorm()` function to Eidos (quantile function for the normal distribution)  
 Feature: add `dbeta()`, `dexp()`, and `dgamma()` functions to Eidos (distribution density functions)  
 Feature: add a SHA-256 hash of the model script to SLiM's `.trees` provenance entry  
 Feature: add `includeModel=T` option to `treeSeqOutput()` to allow model script suppression  
 Tweak: new recipe 16.18 (S-I-R epidemiological model in continuous space)  
 Tweak: modified recipes 9.5.2 and 9.5.3 to fix a bug involving fitness calculations  
 Bug fix: reading in a `.trees` file and then writing one out now handles mutation IDs correctly  
 Bug fix: an overflow of the tree-sequence tables in a large model is now caught  
 Bug fix: crash producing a haplotype plot with a large number of genomes  
 Bug fix: bad appearance of haplotype plots on Retina (high-DPI) displays

#### Version 3.3.2

Feature: added `landscape_ac.slim` to SLiM-Extras for autocorrelated landscape generation  
 Tweak: performance improvements for large models  
 Tweak: performance improvements for `sampleIndividuals()` with sample size 1  
 Tweak: the default nonWF model in SLiMgui now uses `subpop`, not `p1`, for generality  
 Tweak: command line option `-l/-long` (verbosity) now takes an optional `<level>` argument  
 Tweak: new recipe 5.2.4 (joining subpopulations)  
 Tweak: add recipe 16.17 (meiotic drive)  
 Tweak: add recipe 13.6 (a variety of fitness functions)  
 Tweak: extend recipe 5.4 (Gravel model) to show output from a multi-subpop sample  
 Tweak: increase MS output precision to 10 decimal places to accommodate large models  
 Bug fix: passing an unsorted positions vector to `addNew[Drawn]Mutation()` did Very Bad Things  
 Bug fix: `InteractionType` now raises if passed new offspring from `reproduction()` callbacks  
 Bug fix: SLiMgui now checks for the fonts it needs on launch, to avoid confusing errors later  
 Bug fix: recipe 13.4 now correctly includes environmental noise in phenotypic values

#### Version 3.3.1

Feature: add clock type option to `clock()` and `executeLambda()` (cpu or monotonic)

Feature: add a `wait=T` optional parameter to `system()`, allowing background execution

Feature: add `individual` property to `Genome`, to get the individual to which a genome belongs

Tweak: allow access to pedigree IDs when tree-seq recording is enabled, and add to VCF output

Tweak: change recipes that set a new seed to use  $2^{62}$  rather than  $2^{32}$ , to avoid repeats

Bug fix: incorrect results with out-of-order genomic elements with a non-uniform mutrate map

Bug fix: crash with clonal nucleotide-based models using a custom mutation matrix

Bug fix: blank lines in nucleotide-based output when output of back-mutations is suppressed

Bug fix: fixes for self-test with non-existent or non-writeable `/tmp`, `/tmp` permissions issues

Bug fix: fix SLiMgui crash with `MutationType` of type "s" and a fixed display color set

Bug fix: fix SLiMgui crash with the variable browser due to inaccessible properties

Bug fix: fix crash with an `InteractionType` whose state was invalidated by `takeMigrants()`

Bug fix: fix crash when `takeMigrants()` tries to migrate the same individual twice

### *Version 3.3*

Feature: nucleotide-based models, ancestral sequence, sequence-based mutation, hotspot map

Feature: nucleotide/codon/amino-acid utility functions (section 24.17.1)

Feature: new "DSB" recombination model, new gene conversion model

Feature: new gBGC support, heteroduplex mismatch repair, GC bias parameter

Feature: intrinsic support for reading from MS and VCF format files

Feature: new `mutation()` callback type for seeing/modifying/vetoing auto-generated mutations

Feature: new `drawSelectionCoefficient()` method on `MutationType`

Feature: new `fileExists()` function in Eidos for doing filesystem existence checks

Tweak: new `simplificationInterval` parameter to tailor automatic tree-seq simplification

Tweak: much faster MS-format output from large models

Tweak: `initializeGenomicElement()` is now vectorized and returns the created object(s)

Tweak: `spatialMapValue()` is now vectorized

Tweak: `sample()` with a weights vector is now optimized

Tweak: operator `[]` with a singleton integer argument is now optimized

Tweak: extensive recipe renumbering, reordering, revisions, rewriting, removal

Tweak: new section 1.8 (nucleotide models) and chapter 18 (nucleotide model recipes)

Tweak: new chapter 13 (QTL-based models)

Tweak: new recipe 6.1.4 (multiple chromosomes)

Tweak: new recipe 10.6 (varying dominance coefficients)

Tweak: new recipe 12.5 (tracking separate sexes in script)

Tweak: new recipe 14.14 (visualizing ancestry and admixture with `mutation()` callbacks)

Tweak: new section 25.9 (`mutation()` callback reference)

Tweak: matches for search keywords now highlighted in the Find Recipe... panel in SLiMgui

Policy change: some methods with `float` parameters now also accept `integer`, for ease of use

Policy change: the selected display mutation types in SLiMgui now apply to substitutions also

Policy change: `recombination()` callbacks no longer supported with gene conversion enabled

Policy change: `.trees` file format incremented to 0.3 for nucleotides, new `pyslim` is required

Policy change: file formats updated to include support for nucleotides in various ways

Policy change: `tag`, `tagF`, pedigree ID properties now inaccessible when they have no set value

Policy change: subpopulation total fitness must now be finite, to catch overflows



Policy change: update to GSL 2.5, JSON for Modern C++ 3.6.1, kastore C\_1.1.0, tskit 0.1.5  
 Major bug fix: incorrect results from `mutationCounts/Frequencies()` in `early()` in nonWF  
 Major bug fix: incorrect calls in MS format output with `filterMonomorphic=T` (not the default)  
 Major bug fix: `max()` with an integer singleton argument, `!= 0` and `!= 1`, was incorrect, ugh  
 Major bug fix: incorrect results from `matrixMult()` with type `float` in some cases  
 Bug fix: imperfect recreation of state after text-format `outputFull()` and `load` (`float` precision)  
 Bug fix: time base issue when `.trees` files were loaded in an `early()` event in a WF model  
 Bug fix: incorrect sex ratio in sexual nonWF models saved to a SLiM format population file  
 Bug fix: various leaks, various minor issues never reported by any user  
 Bug fix: disabling link-time optimization (LTO) (causing build problems for some users)

### *Version 3.2.1*

Feature: `drawBreakpoints()` method on `Chromosome` to draw breakpoints using SLiM's logic  
 Feature: `rbeta()` function in `Eidos` for draws from a beta distribution  
 Feature: `pnorm()` function in `Eidos` for the CDF of the normal distribution  
 Feature: new `SLiMgui` class, with an instance named `slimgui`, available only under `SLiMgui`  
 Feature: `SLiMgui.openDocument()` method to open a file in `SLiMgui` from script  
 Feature: `SLiMgui.pauseExecution()` method to pause a running `SLiMgui` simulation from script  
 Feature: `SLiMgui.pid` property to get the process ID of `SLiMgui` from script  
 Feature: `Subpopulation.configureDisplay()` method to customize `SLiMgui` display of subpops  
 Tweak: revised recipes 14.8 and 14.13 to use the new `SLiMgui` class  
 Tweak: `slim` (the command-line tool) can now read its script from `stdin` (i.e., from a pipe)  
 Tweak: new recipe 16.15 (Implementing a Wright–Fisher model with a nonWF model)  
 Tweak: new recipe 16.16 (Alternation of generations)  
 Tweak: added support for `make install` with `cmake`, thanks to Peter Ralph  
 Tweak: added support for link-time optimization (LTO), thanks to Kevin Thornton  
 Tweak: improved the reseeding procedure in recipes 10.2, 10.3, 10.6.2, and 17.3  
 Tweak: `SLiMgui` should now run on macOS 10.10 and later (formerly 10.11 and later)  
 Policy change: `inSLiMgui` property of `SLiMSim` deprecated; use `exists("slimgui")`  
 Bug fix: `setValue()` did not copy the value, allowing corruption of the set value in rare cases  
 Bug fix: calling `deregisterScriptBlock()` more than once on the same block did bad things  
 Bug fix: the `pedigreeGrandparentIDs` property returned incorrect values  
 Bug fix: loading a `.trees` file with a mismatched chromosome length now gives an error  
 Bug fix: `SLiMgui` crash when displaying a large number of genomic element types  
 Bug fix: code completion bug when completing off a base of `return/else/do/in`  
 Bug fix: fixed poor error-reporting for malformed command-line definitions  
 Bug fix: fixed a raise that would crash `SLiMgui`, triggered by an illegal numeric constant in script

### *Version 3.2*

Feature: keyword-based “Find Recipe...” panel in `SLiMgui` to make recipe lookup easier  
 Feature: code completion in `SLiMgui` is now much smarter (`iTR` -> `initializeTreeSeq()`, e.g.)  
 Feature: memory usage summary in `SLiMgui`'s profile, also from new `outputUsage()` method  
 Feature: `usage()` function in `Eidos` to get the current / peak total memory usage  
 Feature: `addRecombinant()` method for nonWF for horizontal gene transfer, haploid models, etc.

Feature: `rgeom()` function in Eidos for draws from a geometric distribution

Feature: button in SLiMgui's output area to view/change the current working directory

Feature: `outputMS()` and `outputMSSample()` can now filter out in-sample-monomorphic sites

Feature: new color-palette functions `heatColors()`, `rainbow()`, `terrainColors()`, `cmColors()`

Tweak: new recipe 16.13, modeling clonal haploids in nonWF models with `addRecombinant()`

Tweak: new recipe 16.14, modeling clonal haploid bacteria with horizontal gene transfer

Tweak: vectorized the Eidos color-manipulation functions, using matrix arguments/returns

Tweak: add a `[print=T]` flag to `version()` so switching on version can avoid unwanted output

Tweak: vectorized the `exists()` function so a vector of symbols can be checked in one call

Tweak: some minor optimization work, especially on property / method dispatch in Eidos

Policy change: `asString(NULL)` now returns "NULL" for easier output generation

Policy change: string concatenation with `+` now adds "NULL" for NULL, for easier output

Policy change: scheduling an event/callback into the past now produces an error

Bug fix: fixed a major performance issue for large nonWF models with a lot of genetic diversity

Bug fix: recipe 11.1 issue with the `calcFST()` function returning NAN in some marginal cases

Bug fix: code completion in SLiMgui after return, in, else, and do now works correctly

Bug fix: many Eidos functions are now more robust to being passed NAN (e.g., `rpois()` hang)

Bug fix: corrupted spatial location data for some individuals in `.trees` files

Bug fix: minor issue with SLiMgui's population visualization graph's migration rate arrows

Bug fix: symbol table bug that would bite models defining a very large number of symbols

Bug fix: crash in `addEmpty()` that nobody noticed because nobody uses it

Bug fix: incorrect `modifyChild()` source subpop when using `addCloned()/Crossed()/Selfed()`

Bug fix: incorrect `isCloning/isSelfing` in `modifyChild()` when using `addCloned()/addSelfed()`

Bug fix: incorrect `parent2` values in `modifyChild()` in clonal models (shouldn't be used anyway)

### *Version 3.1*

Feature: greatly increased the performance of spatial interactions with large population size

Feature: added `interactionDistance()` method to get interaction-conditional distances

Feature: added `nearestInteractingNeighbors()` to get interaction-conditional neighbors

Feature: added `interactingNeighborCount()` to count interaction-conditional neighbors

Feature: `rememberIndividuals()` now actually remembers individuals (as well as genomes)

Feature: added automatic remembering of the initial generation, for easier recapitation

Feature: `dmvnorm()` added to Eidos to get probability densities from a multivariate normal dist.

Tweak: extended the `.trees` provenance format for better reproducibility / self-documentation

Tweak: new recipe 14.19, biased gene conversion (two recipes actually)

Tweak: improved recipe 14.3 (reading MS format files), with big performance benefits for it

Tweak: updated all chapter 17 recipes (tree-sequence recording) to reflect changes

Policy change: `strength()` now requires a singleton first argument, receiver

Policy change: `.trees` files now have their time rebased to `msprime`-style times on save

Policy change: `initializeInteractionType()` now warns if no maximum distance is given

Bug fix: memory leak with cycling/varying subpopulation size (genome recycling bug)

Bug fix: nodes in `.trees` files occasionally being marked as null genomes incorrectly

Bug fix: incorrect fitness in WF models using only `fitnessScaling` to modify fitness

### Version 3.0

Feature: non-Wright-Fisher (nonWF) models, chosen with `initializeSLiMModelType()`

Feature: nonWF additions (`age`, `addX()` methods, `takeMigrants()`, `removeSubpopulation()`)

Feature: nonWF additions (`reproduction()` callbacks, `registerReproductionCallback()`)

Feature: new `sampleIndividuals()` and `subsetIndividuals()` methods on `Subpopulation`

Feature: new `fitnessScaling` property on `Subpopulation` and `Individual` to alter fitness

Feature: tree-sequence recording, enabled with `initializeTreeSeq()`

Feature: tree-sequence recording additions (`treeSeqX()` methods)

Feature: new `seqLen()` function in Eidos to generate a zero-based sequence of a given length

Feature: new `getwd()` / `setwd()` functions in Eidos to get and change the working directory

Feature: new `var()` / `cov()` / `cor()` stats functions in Eidos for variance / covariance / correlation

Feature: new `rcauchy()` function for Cauchy distribution (also new interaction function, “c”)

Feature: new `genome1` and `genome2` properties on `Individual` for easier haploid models etc.

Feature: new `migrant` property on `Individual`, usable in both WF and nonWF models

Feature: display interaction types in SLiMgui’s drawer, with hover preview for the function

Feature: improved subpopulation display options in SLiMgui, including spatial map choice

Feature: increase maximum chromosome length from `1e9` to `1e15`

Feature: code completion in SLiMgui can now supply argument names when in a call

Feature: added `getValue()` / `setValue()` functionality to `Substitution`

Feature: optimization work across SLiM and Eidos

Tweak: new recipes for nonWF (chapter 16) and tree-sequence recording (chapter 17)

Tweak: new recipe 14.2 III (mortality-based fitness using `fitnessScaling`)

Tweak: new recipe 14.18 (modeling opposite ends of a chromosome)

Tweak: recipe fixes to conform with changes (13.3, 14.1, 14.7, 14.10, 11.2)

Tweak: `mean()` now works with a logical vector argument

Tweak: vectorize point-processing methods (`pointX()`, `setSpatialPosition()`)

Tweak: add `removeMutations(NULL)` option to quickly remove all mutations from a `Genome`

Tweak: add `[returnMutation = F]` argument to `containsMarkerMutation()` to get the mutation

Tweak: new `suppressWarnings()` function in Eidos to... suppress warnings

Policy change: return values are no longer implied by the value of the last statement

Policy change: `void` is now a true value type in Eidos, not a synonym for `NULL`

Policy change: property/method accesses on zero-length vectors now raise if ambiguous in type

Policy change: `cachedFitness()` may no longer be called from a `late()` event in WF models

Policy change: the default working directory when running in SLiMgui is now `~/Desktop`

Bug fix: incorrect strengths from `InteractionType` with periodic boundaries

Bug fix: gene conversion rate of exactly 1.0 would be treated as 0.0

Bug fix: crash when `setSubpopulationSize(0)` was called twice on the same subpop

Bug fix: crash with the “fitness over time” graph in SLiMgui with an invalid simulation

Bug fix: rare crash on quit in SLiMgui

Bug fix: crash due to stack overflow with large population size and callbacks

Bug fix: display glitch in SLiMgui on Retina displays

Bug fix: fix incorrect actual recombination rate for specified rate of 0.5, impose `<= 0.5` limit

Bug fix: `rdunif()` can now generate draws over the full 64-bit range, not limited to 32-bit

Bug fix: crash in SLiMgui displaying haplotypes in a model with null genomes (X/Y models)

Bug fix: dropped model output from just before a simulation terminates due to an error

#### Version 2.6

Feature: Eidos now supports matrices (2-D) and arrays ( $n$ -D), with new related functions

Feature: haplotype plots and haplotype-based chromosome view display mode

Feature: periodic spatial boundary conditions now supported, including in `InteractionType`

Feature: visualize `MutationType` DFEs in the info drawer in SLiMgui (hover the mouse to see)

Feature: new `rdunif()` Eidos function for draws from discrete uniform distributions

Feature: new `rmvnorm()` Eidos function for draws from multivariate normal distributions

Feature: optimizations of the Eidos core may lead to ~20% speedup for Eidos-intensive models

Tweak: the `apply()` function is now renamed `sapply()`, following R

Tweak: increased display flexibility of individual mutation types in the chromosome view

Tweak: the number of bins in the frequency spectrum plot in SLiMgui can now be changed

Tweak: improved display of mutation/recombination maps in SLiMgui, with a broader range

Tweak: `addNew[Drawn]Mutation()` are now vectorized for speed when adding many mutations

Tweak: new `positionsOfMutationsOfType()` method on `Genome` for speed of some models

Tweak: extended the `integer()` function of Eidos to be able to create two-valued filled vectors

Tweak: `defineSpatialMap()` now accepts a matrix/array of values (backward compatible)

Tweak: `sapply()` (which was `apply()`) has a new `simplify` parameter to get a matrix/array result

Tweak: extended recipe 14.4 to show off the new haplotype display options, check it out

Tweak: revised recipe 14.7 (tracking true local ancestry) for much greater speed

Tweak: recipe 14.9 (modeling nucleotides) is now much faster, with optimizations in SLiM

Tweak: added recipes 14.11 (modeling microsatellites) and 14.12 (modeling transposons)

Tweak: added recipe 14.13: multiple QTL-based quantitative phenotypic traits with pleiotropy

Tweak: added recipe 15.12 to demonstrate periodic boundary conditions in spatial models

Tweak: revised recipes to use `sapply()` instead of `apply()`, other minor recipe tweaks

Policy change: assignment into a subset of a property is no longer legal in Eidos (you don't care)

Policy change: `version()` now returns a numeric version number, for runtime version checking

Policy change: Chromosome properties that are undefined now raise when accessed

Policy change: `addNew[Drawn]Mutation()` returns requested mutations whether added or not

Policy change: `periodicity` inserted in `initializeSLiMOptions()`; use named arguments there!

Bug fix: possible incorrect mutation freqs/counts after `addMutations()` / `removeMutations()`

Bug fix: fixed a long-standing display bug in SLiMgui with added/removed mutations

Bug fix: very minor bug fixes in `doCall()`, SLiMgui, etc.; very unlikely impact on model results

#### Version 2.5

Feature: add support for user-defined Eidos functions in SLiM – make your own functions

Feature: support for variable mutation rate along the chromosome (i.e., “hot” and “cold” spots)

Feature: display of the mutation-rate map in SLiMgui with the R button

Feature: `Mutation` now supports the `getValue()/setValue()` mechanism

Feature: SLiMgui can do script prettyprinting, for nice code formatting

Feature: new ternary conditional operator, `?else`, added to Eidos, like `?:` in C/C++

Feature: support for block-style `/* */` comments added to Eidos, as in C/C++

Feature: added a function `source()` to read in and execute Eidos code from a given file  
 Feature: the SLiM and Eidos documentation now has hyperlinks in its table of contents  
 Major bug fix: new mutations were not added correctly in clonal reproduction, in 2.4–2.4.2  
 Tweak: improved `pmax()`, `pmin()`, `max()`, `min()`, `range()`, `seq()`, `any()`, `all()`, `unique()`  
 Tweak: added a `sumExact()` function to Eidos for exact summation of floating-point numbers  
 Tweak: fixed a bug in recipe 11.1's  $F_{ST}$  calculation code, and split it into a standalone function  
 Tweak: added recipe 14.10, illustrating how to make a simple haploid clonal model in SLiM  
 Tweak: added recipe 14.11, modeling variation in functional density along the chromosome  
 Policy change: removed the `mutationRate` property of `Chromosome` in favor of new API  
 Policy change: `function()`, `method()`, `property()` renamed to `functionSignature()`, etc.  
 Policy change: argument function for `doCall()` renamed to `functionName`  
 Bug fix: improved numerical accuracy for complex recombination maps  
 Bug fix: incorrect results from `InteractionType` if individuals had an identical coordinate value

#### *Version 2.4.2*

Major bug fix: fix non-unique mutation id bug (incorrect output from many/most models)

#### *Version 2.4.1*

Bug fix: fix stale subpop pointer bug (possible crash or incorrect data in multi-subpop sims)

#### *Version 2.4*

Feature: extensive internal optimizations for better performance of many models  
 Feature: model runtime profiling added in SLiMgui for performance evaluation  
 Feature: `sumOfMutationsOfType()` method added for fast QTL effect addition  
 Feature: `preventIncidentalSelfing` option added to `initializeSLiMOptions()`  
 Feature: optionally configure the mutation run count in `initializeSLiMOptions()`  
 Feature: `system()` function to call out to Unix to run commands  
 Feature: added PDF viewing to SLiMgui for R plotting integration (see recipe 14.8)  
 Feature: chromosome view display customization via right-click in SLiMgui  
 Feature: population view display customization via right-click in SLiMgui  
 Feature: `writeTempFile()` function for creating randomly named temporary files  
 Feature: new `catn()` and `paste0()` functions for simpler output generation  
 Feature: `MutationType`, `GenomicElementType`, `InteractionType` now support `get/setValue()`  
 Feature: add `mutationStackGroup` property to `MutationType` and improve stacking options  
 Major bug fix: significant bug in the Eidos function `setDifference()` (no impact if not using it)  
 Tweak: added top/bottom splitview in SLiMgui to allow greater display flexibility  
 Tweak: the play speed slider in SLiMgui now shows the chosen play speed in a tooltip  
 Tweak: added font size preference in SLiMgui for presentations, etc.  
 Tweak: new `inSLiMgui` property on `SLiMSim` to tell whether the model is running in SLiMgui  
 Tweak: `rescheduleScriptBlock()` method added on `SLiMSim` for easier block rescheduling  
 Tweak: `ttest()` function added for performing  $t$ -tests in Eidos  
 Tweak: added `-l` (`-long`) command-line option for more verbose output  
 Policy change: scripted (type “s”) DFEs in `MutationType` now have access to SLiM constants  
 Bug fix: for loops on `seqAlong()` vectors with a zero-length parameter were incorrect

### *Version 2.3*

Feature: continuous space (1D, 2D, or 3D) and spatial positions for `Individual` (x, y, and z)  
Feature: landscape maps that can define environmental values across space  
Feature: interactions (both non-spatial and spatial), including `interaction()` callbacks  
Feature: “global” `fitness()` callbacks that are called once for every individual  
Tweak: `outputFull()` and `readFromPopulationFile()` now save/restore spatial positions  
Tweak: `mateChoice()` callbacks can now return a vector of all zero to reject the first parent  
Tweak: `mateChoice()` callbacks can now return a singleton `Individual` as the chosen parent  
Tweak: added several color-conversion functions to `Eidos` (RGB to/from HSV, etc.)  
Policy change: `readFromPopulationFile()` no longer recalculates fitness values  
Policy change: `readFromPopulationFile()` now warns if called outside a `late()` event

### *Version 2.2.1*

Feature: `SLiMgui` is now a proper document-based app using the `.slim` file extension  
Feature: `tagF` property on `Individual` for storage of float custom state  
Feature: `order()` function that provides sorting indices, like the R `order()` function  
Feature: custom coloring of individuals, etc., in `SLiMgui` using new color properties  
Tweak: `SLiMgui`’s recycle button now highlights green when the script has been changed  
Tweak: arbitrary `Eidos` expressions are now legal in command-line constant definitions  
Policy change: multiple calls to `initializeRecombinationRate()` is now explicitly illegal

### *Version 2.2*

Feature: `recombination()` callbacks allow scripted alteration of recombination breakpoints  
Feature: `containsMarkerMutation()` method allows fast checking for “marker” mutations  
Feature: `clock()` function gives the CPU usage of the process, for measuring performance  
Feature: `setValue()` / `getValue()` methods provide several classes with dictionary-like values  
Feature: command-line constant definition using a `-define` or `-d` flag passed to `slim`  
Tweak: “tips & tricks” panel for `SLiMgui` that introduces some obscure features of the app

### *Version 2.1.1*

Feature: `mutationCounts()` method provides raw mutation reference counts

### *Version 2.1*

Feature: `Individual` class added to represent individuals, with many associated changes  
Feature: optional pedigree-based relatedness tracking for individuals  
Feature: overhaul of `SLiM` output methods to provide file-based output and appending  
Feature: new `Genome` output methods allow output of custom samples  
Feature: VCF format output added to the existing `SLiM` and `MS` formats  
Feature: `Mutation` now has a unique identifier usable for tracking mutations through time  
Feature: `Mutation` and `Substitution` now have a `tag` property like many other `SLiM` classes  
Feature: DFE type “s” now allows scripted DFEs for `MutationType`  
Feature: sex-specific recombination rates and/or recombination maps  
Feature: default arguments and named arguments supported for `Eidos` functions and methods  
Feature: `deleteFile()` and `createDirectory()` functions added to `Eidos`  
Feature: set operation functions (union, intersection, etc.) added to `Eidos`



Tweak: `size()` method added to the Eidos object base class  
Tweak: recipes now openable directly from the Recipes submenu in SLiMgui  
Policy change: output of mutations now includes a mutation identifier field (format change)  
Policy change: `readFromPopulationFile()` now sets the generation as a side effect  
Policy change: class methods in Eidos now provide non-multicasted method invocation  
Policy change: `addNewMutation()` and `addNewDrawnMutation()` changed to class methods

#### *Version 2.0.4*

Minor bug fixes.

#### *Version 2.0.3*

Minor bug fixes.

#### *Version 2.0.2*

Feature: binary format for `outputFull()`, for smaller files and much faster save/load  
Feature: `setMutationType()` added to allow mutations to be assigned to a different type  
Feature: `beep()` function allows audible output from scripts  
Tweak: `readFromPopulationFile()` can now read the new binary file format

#### *Version 2.0.1*

Feature: `format()` function added to Eidos for customized formatting of output

#### *Version 2.0*

This was the original version of SLiM 2, which was an almost complete rewrite of SLiM and introduced such major features as Eidos and SLiMgui. The full list of changes is far too extensive to detail here. Before SLiM 2 came SLiM 1, which was documented in Messer (2013).

## 30. Credits and licenses for incorporated software

SLiM incorporates code from various other software frameworks and packages. This section provides credit and license information for the software thus incorporated, as well as for SLiM itself.

### 30.1 SLiM

SLiM is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

SLiM is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with SLiM. If not, see <http://www.gnu.org/licenses/>.

### 30.2 GNU Scientific Library (GSL)

SLiM contains a modified distribution of the GNU Scientific Library (GSL), which is also licensed under the GNU General Public License under the same terms stated above. Thanks to the authors of the GSL for their very useful software, which can be found in full form at <http://www.gnu.org/software/gsl/>.

### 30.3 tskit / kastore

SLiM contains code from the tskit and kastore software packages (enabling the tree-sequence recording feature of SLiM, and the ability to save and load .trees files). Thanks to the tskit and kastore developers for this code, and to Peter Ralph and Jared Galloway for contributing code to SLiM to integrate it with these software packages. The tskit package is MIT licensed, and is available at <https://github.com/tskit-dev/tskit>. The kastore package is MIT licensed, and is available at <https://github.com/tskit-dev/kastore>.

### 30.4 Boost

SLiM contains modified code from Boost (for a smart pointer implementation), which is licensed under the Boost Software License version 1.0 ([http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt)). The Boost Software License is compatible with the GPL, allowing its use here. Thanks to all of the authors of Boost for their very useful software, which can be downloaded in its full form from their website at <http://www.boost.org/users/download/#live>. Thanks especially to Peter Dimov, who appears to be the author of the class in question.

### 30.5 MT19937-64

SLiM contains a 64-bit Mersenne Twister implementation (a random number generator) by Takuji Nishimura and Makoto Matsumoto; thanks to them for this code, which was obtained from <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/mt19937-64.c>. Following the terms of their license, the following text is provided from them:

Copyright (C) 2004, Makoto Matsumoto and Takuji Nishimura,  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### 30.6 WiggleTools

SLiM contains code from WiggleTools to perform *t*-tests. WiggleTools is licensed under the Apache 2.0 license (<http://www.apache.org/licenses/LICENSE-2.0>), which is compatible with the GPL, allowing its use here. Thanks to EMBL-European Bioinformatics Institute for making this code available.

### 30.7 ObjectPool

SLiM contains a modified version of a C++ object pool class published by Paulo Zemek at <http://www.codeproject.com/Articles/746630/O-Object-Pool-in-Cplusplus>. His code is under the Code Project Open License (CPO), available at <http://www.codeproject.com/info/cpol10.aspx>. The CPO is not compatible with the GPL. Paulo Zemek has explicitly granted permission for his code to be used in Eidos and SLiM, and thus placed under the GPL as an alternative license. An email granting this permission has been archived and can be provided upon request. This code is therefore now under the same GPL license as the rest of SLiM and Eidos, as stated above.

### 30.8 Exact summation

SLiM contains modified (translated to C from Python) code to compute the exact summation (within available precision limits) of a vector of floating-point numbers. This code is adapted from Python's `fsum()` function, as implemented in the file `mathmodule.c` in the `math_fsum()` C function, from Python version 3.6.2, downloaded from <https://www.python.org/getit/source/>. The authors of that code appear to be Raymond Hettinger and Mark Dickinson; thanks to them. The PSF open-source license for Python 3.6.2, which the PSF states is GSL-compatible, may be found on their website at <https://docs.python.org/3.6/license.html>.

### 30.9 JSON for Modern C++

SLiM contains code from the JSON for Modern C++ project (<https://github.com/nlohmann/json>). Thanks to Niels Lohmann and other contributors for this very useful library. This code is under the MIT license, and is governed by the following notice:

The class is licensed under the MIT License:

Copyright © 2013–2019 Niels Lohmann

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

The class contains the UTF-8 Decoder from Bjoern Hoehrmann which is licensed under the MIT License (see above). Copyright © 2008–2009 Björn Hoehrmann [bjoern@hoehrmann.de](mailto:bjoern@hoehrmann.de)

The class contains a slightly modified version of the Grisu2 algorithm from Florian Loitsch which is licensed under the MIT License (see above). Copyright © 2009 Florian Loitsch

### 30.10 SHA-256

SLiM contains code for the computation of SHA-256 hashes, from <https://github.com/amosnier/sha-2>. Thanks to Alain Mosnier for this code. It is provided under the public domain “Unlicense”, which reads as follows:

This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or distribute this software, either in source code form or as a compiled binary, for any purpose, commercial or non-commercial, and by any means.

In jurisdictions that recognize copyright laws, the author or authors of this software dedicate any and all copyright interest in the software to the public domain. We make this dedication for the benefit of the public at large and to the detriment of our heirs and successors. We intend this dedication to be an overt act of relinquishment in perpetuity of all present and future rights to this software under copyright law.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER

IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to <<http://unlicense.org>>

### 30.11 zlib

SLiM contains a modified copy of the zlib compression/decompression library (version 1.2.11) to support the writing of compressed files with `writeFile()` and `writeTempFile()`. The zlib library is written by Jean-loup Gailly and Mark Adler; thanks to them. It is made freely available for reuse, under conditions described in the `eidos_zlib/README` file included in this project; those conditions are GPL-compatible. The original, unmodified sources for zlib may be found at <http://zlib.net/>.

### 30.12 Worst-case Optimal Incremental Sorting

SLiM contains code from <https://github.com/KukyNekoi/magicode> which implements the “Optimal Incremental Sorting” and “Worst-case Optimal Incremental Sorting” algorithms of Paredes & Navarro (2006) and Regla & Paredes (2015). This code is by Erik Regla, and is released under the GPL 3. Thanks very much to Erik Regla for making it available online and for assistance in getting it working within SLiM.

### 30.13 tinycolormap

SLiM contains a modified copy of the tinycolormap library (version 0.7.0) for generation of color maps. The tinycolormap library is written by Yuki Koyama. It is licensed under the MIT License, which is compatible with the GPL 3.0. The original, unmodified sources for tinycolormap may be found at <https://github.com/yuki-koyama/tinycolormap>. That code has been modified, in particular by merging in a pull request by Shot511 (Tomasz Gałaj) that adds support for the Turbo colormap, available at <https://github.com/yuki-koyama/tinycolormap/pull/27>. That code is derived from original code by Anton Mikhailov licensed under the Apache 2.0 license, which is compatible with the GPL 3.0; Anton’s code is available at <https://gist.github.com/mikhailov-work/6a308c20e494d9e0ccc29036b28faa7a>. Copies of the MIT and Apache 2.0 licenses, under which this code was originally released, are provided in `eidos_tinycolormap.h`. Thanks to Yuki, Anton, and Tomasz for their work.

### 30.14 LodePNG

SLiM contains a modified copy of the LodePNG library (cloned 8 October 2020, version 20200306) for reading in PNG images for the Eidos built-in class `Image`. The LodePNG library is written by Lode Vandevenne. It is licensed under the Zlib License, which is compatible with the GPL 3.0. The original, unmodified sources for LodePNG may be found at <https://lodev.org/lodepng/>, or on GitHub at <https://github.com/lvandeve/lodepng>. The code has been modified to get rid of some warnings. Thanks to Lode for this very useful code. The Zlib license used by LodePNG can be found at <https://raw.githubusercontent.com/lvandeve/lodepng/master/lodepng.h>; it reads:

Copyright (c) 2005–2020 Lode Vandevenne

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it

freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

### 30.15 Robin Hood Hashing

SLiM contains an unmodified copy of the Robin Hood Hashing library (cloned 29 October 2020, version 3.9.0), a very fast hash table implementation that provides a drop-in replacement for `std::unordered_map`. Robin Hood Hashing is by Martin Ankerl, and is available on GitHub at <https://github.com/martinus/robin-hood-hashing>; thanks to Martin for this excellent software. It is licensed under the MIT license, which is compatible with the GPL 3.0. The license for Robin Hood Hashing is shown here in full:

MIT License

Copyright (c) 2018–2019 Martin Ankerl

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 30.16 Ryan's CMake Modules Collection

SLiM contains a partial copy of the Ryan's CMake Modules Collection library (cloned 2 August 2022, version 3.9.0), a very useful collection of modules for the CMake build system. This library is by Ryan A. Pavlik, and is available on GitHub at <https://github.com/rpavlik/cmake-modules>; thanks to Ryan for these very useful tools. It is licensed under the Boost Software License version 1.0, which is compatible with the GPL 3.0. The license for it is shown here in full:

Boost Software License – Version 1.0 – August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:



The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### **30.17 Other contributions**

Smaller snippets of code have been gleaned from the web, particularly from stackoverflow. Credit is given in the SLiM source code where this has occurred, and in all cases the code is licensed under terms compatible with the GSL (as far as we, who are not lawyers, can tell). Thanks to everyone whose code has found its way into SLiM.

## 31. References

- Amorim, C.E.G., Gao, Z., Baker, Z., Diesel, J.F., Simons, V.B., Haque, I.S., Pickrell, J., and Przeworski, M. (2017). The population genetics of human disease: The case of recessive, lethal mutations. *PLoS Genetics* 13(9), e1006915.
- Ayala, F.J., and Campbell, C. (1974). Frequency-dependent selection. *Annual Review of Ecology, Evolution, and Systematics* 5, 115–138.
- Bhatia, G., Patterson, N., Sankararaman, S., and Price, A.L. (2013). Estimating and interpreting  $F_{ST}$ : The impact of rare variants. *Genome Research* 23, 1514–1521.
- Boost. (2015). Boost C++ Libraries [version 1.59.0]. URL: <http://www.boost.org>
- Charlesworth, B., Morgan, M.T., and Charlesworth, D. (1993). The effect of deleterious mutations on neutral molecular variation. *Genetics* 134(4), 1289–1303.
- Chen, J.-M., Cooper, D.N., Chuzhanova, N., Férec, C., and Patrinos, G.P. (2007). Gene conversion: Mechanisms, evolution and human disease. *Nature Reviews Genetics* 8(10), 762–775.
- Comeron, J.M., Ratnappan, R., and Bailin, S. (2012). The many landscapes of recombination in *Drosophila melanogaster*. *PLoS Genetics* 8(10), e1002905.
- Cury, J., Haller, B.C., Achaz, G., and Jay, F. (2022). Simulation of bacterial populations with SLiM. *Peer Community Journal* 2(2), e7.
- Dawkins, R. (1976). *The Selfish Gene*. Oxford University Press.
- Deutsch, M., and Long, M. (1999). Intron–exon structures of eukaryotic model organisms. *Nucleic Acids Research* 27(15), 3219–3228.
- Dieckmann, U., and Doebeli, M. (1999). On the origin of species by sympatric speciation. *Nature* 400(6742), 354–357.
- Doebeli, M., and Dieckmann, U. (2003). Speciation along environmental gradients. *Nature* 421(6920), 259–264.
- Doudna, J.A., and Charpentier, E. (2014). The new frontier of genome engineering with CRISPR- Cas9. *Science* 346(6213), 1258096–1–1258096–9.
- Duret, L., and Galtier, N. (2009). Biased gene conversion and the evolution of mammalian genomic landscapes. *Annual Review of Genomics and Human Genetics* 10, 285–311.
- Elyashiv, E., Sattath, S., Hu, T. T., Strutsovsky, A., McVicker, G., Andolfatto, P., Coop, G., and Sella, G. (2016). A genomic map of the effects of linked selection in *Drosophila*. *PLoS Genetics* 12(8), e1006130.
- Faure, M., and Schreiber, S.J. (2014). Quasi-stationary distributions for randomly perturbed dynamical systems. *The Annals of Applied Probability* 24(2), 553–598.
- Fiston-Lavier, A.S., and Petrov, D.A. (2013). *Drosophila melanogaster* recombination rate calculator. URL: [http://petrov.stanford.edu/cgi-bin/recombination-rates\\_updateR5.pl](http://petrov.stanford.edu/cgi-bin/recombination-rates_updateR5.pl). *Drosophila* chromosome map URL: [http://petrov.stanford.edu/RRC\\_scripts/RRCv2.3.tar.gz](http://petrov.stanford.edu/RRC_scripts/RRCv2.3.tar.gz)
- Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., and Rossi, F. (2009). GNU Scientific Library Reference Manual [3rd Ed.]. ISBN 0954612078.
- Gravel, S., Henn, B.M., Gutenkunst, R.N., Indap, A.R., Marth, G.T., Clark, A.G., Yu, F., Gibbs, R.A., Bustamante, C.D., Altshuler, D.L., and Durbin, R.M. (2011). Demographic history and rare allele sharing among human populations. *PNAS* 108(29), 11983–11988.
- Grimm, V. (2002.) Visual debugging: A way of analyzing, understanding and communicating bottom-up simulation models in ecology. *Natural Resource Modeling* 15(1): 23–38.

- Guerrero, R. F., Rousset, F., and Kirkpatrick, M. (2012). Coalescent patterns for chromosomal inversions in divergent populations. *Phil. Trans. R. Soc. B* 367(1587), 430–438.
- Haller, B.C. (2016). Eidos: A Simple Scripting Language. URL: <http://messerlab.org/slim/>
- Haller, B.C., and Hendry, A.P. (2013). Solving the paradox of stasis: Squashed stabilizing selection and the limits of detection. *Evolution* 68(2), 483–500.
- Haller, B.C., Mazzucco, R., and Dieckmann, U. (2013). Evolutionary branching in complex landscapes. *American Naturalist* 182(4), E127–E141.
- Hamilton, W.D. (1964a). The genetical evolution of social behaviour I. *Journal of Theoretical Biology* 7(1), 1–16.
- Hamilton, W.D. (1964b). The genetical evolution of social behaviour II. *Journal of Theoretical Biology* 7(1), 17–52.
- Hill, W. G., and Robertson, A. (1966). The effect of linkage on limits to artificial selection. *Genetical Research* 8(3), 269–294.
- Hudson, R.R. (1994). How can the low levels of DNA sequence variation in regions of the *Drosophila* genome with low recombination rates be explained? *PNAS* 91(15), 6815–6818.
- Jouganous, J., Long, W., Ragsdale, A.P., and Gravel, S. (2017). Inferring the joint demographic history of multiple populations: Beyond the diffusion approximation. *Genetics* 206(3), 1549–1567.
- Kelleher, J., Etheridge, A.M., and McVean, G. (2016). Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS Computational Biology* 12(5): e1004842.
- Kelleher, J., Thornton, K.R., Ashander, J., and Ralph, P.L. (2018). Efficient pedigree recording for fast population genetics simulation. *bioRxiv* 248500. DOI: <https://doi.org/10.1101/248500>
- Kimura, M. (1962). On the probability of fixation of mutant genes in a population. *Genetics* 47(6), 713–719.
- Leimar, O., Doebeli, M., and Dieckmann, U. (2008). Evolution of phenotypic clusters through competition and local adaptation along an environmental gradient. *Evolution* 62(4), 807–822.
- Mazzucco, R., Doebeli, M., and Dieckmann, U. (2018). The influence of habitat boundaries on evolutionary branching along environmental gradients. *Evolutionary Ecology* 32(6), 563–585.
- Messer, P.W. (2013). SLiM: Simulating evolution with selection and linkage. *Genetics* 194(4), 1037–1039.
- Messer, P.W., and Petrov, D.A. (2013). Population genomics of rapid adaptation by soft selective sweeps. *Trends In Ecology & Evolution* 28(1), 659–669.
- Morton, N.E., Crow, J.F., and Muller, H.J. (1956). An estimate of the mutational damage in man from data on consanguineous marriages. *PNAS* 42(11), 855–863.
- Newbigin, E., Anderson, M.A., and Clarke, A.E. (1993). Gametophytic self-incompatibility systems. *The Plant Cell* 5(10), 1315–1324.
- Payne, J.L., Mazzucco, R., and Dieckmann, U. (2011). The evolution of conditional dispersal and reproductive isolation along environmental gradients. *Journal of Theoretical Biology* 273(1), 147–155.
- Philipps, P.C. (2008). Epistasis – the essential role of gene interactions in the structure and evolution of genetic systems. *Nature Reviews Genetics* 9(11), 855–867.
- R Core Team. (2015). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL: <https://www.R-project.org/>

- Sattath, S., Elyashiv, E., Kolodny, O., Rinott, Y., and Sella, G. (2011). Pervasive adaptive protein evolution apparent in diversity patterns around amino acid substitutions in *Drosophila simulans*. *PLoS Genetics* 7(2), e1001302.
- Servedio, M.R., Van Doorn, G.S., Kopp, M., Frame, A.M., and Nosil, P. (2011). Magic traits in speciation: 'Magic' but not rare?. *Trends in Ecology & Evolution* 26(8), 389–397.
- Sjödin, P., Kaj, I., Krone, S., Lascoux, M., and Nordborg, M. (2005). On the meaning and existence of an effective population size. *Genetics* 169(2), 1061–1070.
- Smith, J.M., and Haigh, J. (1974). The hitch-hiking effect of a favorable gene. *Genetical Research* 23(1), 23–25.
- Uricchio, L.H., and Hernandez, R.D. (2014). Robust forward simulations of recurrent hitchhiking. *Genetics* 197(1), 221–236.
- Vincenot, C.E., Carteni, F., Mazzoleni, S., Rietkerk, M., and Giannino, F. (2016). Spatial Self-Organization of Vegetation Subject to Climatic Stress—Insights from a System Dynamics—Individual-Based Hybrid Model. *Frontiers in Plant Science* 7, 636.
- Wang, J., Santiago, E., Caballero, A. (2016). Prediction and estimation of effective population size. *Heredity* 117, 193–206.