SLiM 2: Flexible, Interactive Forward Genetic Simulations

Benjamin C. Haller* and Philipp W. Messer*
Department of Biological Statistics and Computational Biology, Cornell University, Ithaca, NY
*Corresponding authors: E-mails: bhaller@benhaller.com; messer@cornell.edu.
Associate editor: Ryan Hernandez

Abstract

Modern population genomic datasets hold immense promise for revealing the evolutionary processes operating in natural populations, but a crucial prerequisite for this goal is the ability to model realistic evolutionary scenarios and predict their expected patterns in genomic data. To that end, we present SLiM 2: an evolutionary simulation framework that combines a powerful, fast engine for forward population genetic simulations with the capability of modeling a wide variety of complex evolutionary scenarios. SLiM achieves this flexibility through scriptability, which provides control over most aspects of the simulated evolutionary scenarios with a simple R-like scripting language called Eidos. An example SLiM simulation is presented to illustrate the power of this approach. SLiM 2 also includes a graphical user interface for simulation construction, interactive runtime control, and dynamic visualization of simulation output, facilitating easy and fast model development with quick prototyping and visual debugging. We conclude with a performance comparison between SLiM and two other popular forward genetic simulation packages.

Key words: **forward genetic simulation, population genomics, evolutionary modeling, ecological modeling, software.**

Introduction

Modern population genomic datasets are fundamentally changing our ability to study evolutionary processes. As genome sequencing becomes easier and cheaper, patterns of genetic variation can be surveyed in unprecedented detail (Luikart et al. 2003; Ellegren 2014; Schraiber and Akey 2015). In some instances, we can now even observe the essence of evolution directly: how genetic variants change in frequency in a population over time (Jerison and Desai 2015; Levy et al. 2015; Malaspinas 2016; Messer et al. 2016). These advances promise to greatly improve our understanding of how populations evolve, by allowing us to discern and quantify which evolutionary processes are operating, and to link those processes to genetic and ecological factors.

A key prerequisite for achieving these goals is the ability to connect patterns in population genomic data with underlying evolutionary processes. Population genetic theory has been tremendously successful in helping us understand how basic processes such as random genetic drift, classic selective sweeps, and purifying selection are expected to shape these population genomic patterns (Ewens 2004; Charlesworth and Charlesworth 2010). However, evolutionary dynamics in real populations are often far more complex than is captured by these theoretical models (Thompson 2013; Hendry 2016; Messer et al. 2016). Real populations are rarely panmictic; their sizes can fluctuate widely, driven by ecological factors or changes in the genetic makeup of the populations themselves; selection coefficients can vary over both time and space; epistatic interactions can be important; mate choice can be affected by ecological, phenotypic, and genetic factors; reproduction can occur sexually and asexually, with some organisms performing a combination of both; and all of these processes typically operate on highly heterogeneous genomes with varying recombination rates, functional density, and selective constraints.

To study the behavior of such realistic evolutionary scenarios, we increasingly rely on large-scale simulation approaches (Carvajal-Rodríguez 2010; Messer 2013; Bank et al. 2014; Hoban 2014), similar to the situation for other complex dynamical systems, such as climate (Palmer 2014) and the human brain (Markram et al. 2015). A wide variety of evolutionary simulation tools are now available, including idealized models such as the coalescent process (Hudson 2002; Ewing and Hermisson 2010; Excoffier and Foll 2011); command-driven forward simulation packages such as SFS_CODE (Hernandez and Uricchio 2015), Nemo (Guillaume and Rougemont 2006), quantiNemo (Neuenschwander et al. 2008), forqs (Kessner and Novembre 2014), and SLiM 1.8 (Messer 2013); C++-based forward simulation template libraries such as fwdpp (Thornton 2014); and scriptable forward simulation packages such as simuPop (Peng and Kimmel 2005) and fwdpy, a Python front end for fwdpp (Thornton 2014). Indeed, at the time of writing, 110 software packages related to genetic simulation were listed on the National Cancer Institute's Genetic Simulation Resources page (GSR; NCI 2016). However, according to the GSR's database, only a minority of those 110 packages support forward-time simulation, and many of those are special-purpose packages designed for particular narrow problems. For example, only a few support such features as multiple traits, sex-linked traits, epistasis, frequency-dependent selection, gene–environment interactions, selfing, assortative mating, or gene conversion, and almost none support all of those features. Furthermore, there are often trade-offs between performance,

flexibility, and ease of use, and software that embodies all of those qualities is rare.

Here, we present SLiM 2, a general-purpose forward genetic simulation framework that combines a powerful and fast engine for forward population genetic simulations with a high degree of flexibility in specifying complex evolutionary scenarios. SLiM 2 achieves this flexibility through scriptability, which allows most aspects of the simulation to be tailored and customized. Scripting has already proven incredibly powerful in other areas of computational evolutionary biology, such as phylogenetic reconstruction (Pond et al. 2005) and systems biology (Spjuth et al. 2009). The scriptability of SLiM 2, using an R-like scripting language called Eidos, provides this power for forward population genetic simulations with realistic evolutionary dynamics.

Another new feature introduced in SLiM 2 is a graphical user interface, SLiMgui, which provides an interactive scripting experience that makes development and testing of simulations vastly easier. Grimm (2002) emphasized the importance of visual debugging of models; many problems that would be difficult to detect in raw output data are immediately apparent when watching a model run in an interactive environment, leading to models that are more robust and trustworthy. SLiMgui also allows an experimental, curiosity-driven approach to modeling that can enable a different and highly fruitful approach to scientific questions. As such, SLiMgui provides obvious advantages for use in educational contexts, both by instructors and by students.

In the next section, we will discuss the advantages of scriptability in more detail, with examples of the sorts of models that it makes possible to implement in SLiM 2. In subsequent sections, we will then discuss the underlying Wright–Fisher simulation model used in SLiM 2, provide an example of a script for a SLiM 2 simulation, discuss the features of the SLiMgui interactive modeling environment, and show a comparison of SLiM 2 with a few other forward genetic simulation packages.

Why Scriptability?

Many existing forward genetic simulation tools fall into one of two classes. The first class comprises programs that provide a fixed set of options for simulation configuration. These might be specified on the command line, as in SFS_CODE (Hernandez 2008; Hernandez and Uricchio 2015), or via a structured input file, as in SLiM 1.8 (Messer 2013), the previous version of SLiM. While these programs can be easy to use, they provide only limited flexibility since only evolutionary scenarios that have been explicitly incorporated into the program can be simulated; extending these programs to other scenarios typically requires modifying the actual source code.

The second class of tools comes in the form of libraries of functions that can be used to implement custom evolutionary simulations in a programming language like C++, such as fwdpp (Thornton 2014). This approach provides a lot of flexibility, but it requires users to be proficient in the programming language in which the toolkit is written.

As a result, it is often simpler for researchers to write their own purpose-built models from scratch, without using any pre-existing simulation framework or library, but this entails substantial drawbacks as well: it involves reinventing the wheel over and over, it limits the level of complexity attainable because each project begins again from scratch, and it is more bug-prone because each newly developed model does not benefit from the debugging and testing that went into previous models.

However, there is a third class of forward genetic simulation tools: scriptable frameworks. Scriptability bridges the gap between ease of use and power by allowing enormous flexibility (since the user can write custom script to build their own functionality) while still providing a simple and intuitive platform for defining evolutionary models (since a great deal of standard functionality is provided by the framework). Scriptability does not require extensive programming skills from the user, since it is not necessary to understand or modify the underlying code that implements the framework itself. Even complex scenarios such as epistasis or sequential mate-choice can often be expressed in a few lines of script—a much simpler proposition than trying to add such a feature to a framework's actual sources (typically thousands of lines of complex code), and a much simpler proposition than writing a custom model from scratch. The first such scriptable forward genetic simulation package was simuPOP (Peng and Kimmel 2005). Other examples include the fwdpy Python-based front end for fwdpp (Thornton 2014), and now, SLiM 2.

SLiM 2 implements scripting via the Eidos language, which is quite similar to R (even to the point of sharing many function names); users familiar with R should find Eidos very easy to learn. The underlying simulation engine contains a full Eidos interpreter, and is thus quite complex, but in practice it can be treated as a black box that never needs to be modified or understood by the user. The Eidos script that drives a particular simulation, on the other hand, is usually trivially short, easily understood, and quickly modified.

Scripting with Eidos allows the structure of a simulation to be expressed with high-level concepts such as "for" loops, conditional "if" statements, and calls to functions and methods. In addition, script blocks called "callbacks" can be defined that are called by SLiM's simulation engine to modify the dynamics of the simulation. The simplest callbacks are so-called `early()` and `late()` events, which are called by SLiM at the beginning and end of generations, respectively. A `fitness()` callback can be used to change the fitness effects of mutations, on a per-individual and per-mutation basis; this allows mutational effects to depend upon genetic background, the subpopulation in which a mutation is expressed, or any other simulation state. A `mateChoice()` callback can redefine mating preferences (which are proportional to relative fitness by default), allowing mate choice to depend on individual genetics or on environmental factors. Finally, a `modifyChild()` callback can customize offspring generation in a wide variety of ways, from adding or removing mutations in particular offspring, to suppressing the generation of particular individuals completely because of genetic or environmental factors.

SLiM's manual provides dozens of "recipes" showing how to implement a wide range of evolutionary models using this scriptability (Haller and Messer 2016). A few possibilities include:

- **Dynamic population size, structure, and patterns of migration.** Loops can be used to create complex spatial structure such as "island" models, grids, and networks in just a few lines of Eidos code. Population dynamics can depend upon the simulation itself; e.g., the size of a subpopulation could be set proportionally to mean absolute fitness in each generation using a `late()` event to produce a "hard selection" model, or the probability of migration of individuals could depend upon their genetics (using a `modifyChild()` callback to suppress some proposed offspring) to produce a model of dispersal evolution.
- **Variation in mating systems.** Aspects of reproduction such as selfing, cloning, hermaphroditism versus separate sexes, and sex ratio variation are all controlled by script. The sex ratio and the cloning rate of a particular subpopulation could be made to vary through time with a simple `late()` callback, e.g., or the probability of selfing could be made to depend upon other simulation dynamics such as the current population size, reflecting a propensity to self when rare but to outcross when common.
- **Genomic complexity**. Scriptability can govern aspects of genetic structure (introns, exons, and noncoding regions, e.g.), recombination hot/cold spots, different mutational distributions of fitness effects, and gene conversion. With a few lines, a particular genetic structure could be generated based upon model parameters such as the degree of linkage between loci or the average length of exons versus introns in the genome. Alternatively, a map of the genetic structure for a simulation could be read in from a file, since Eidos supports file input and output. Mutations could be introduced in script depending upon the state of the simulation; e.g., an introduced mutation could be triggered by a particular demographic event.
- **Context-dependent selection**. Models can include complex, context-dependent selection such as temporal and spatial variation in fitness, epistasis, polygenic selection, frequency-dependent selection, and kin selection. For example, an introduced mutation can be made beneficial in one subpopulation but deleterious in another using a `fitness()` callback that calculates the fitness effect of the mutation depending upon the subpopulation of the mutation's carrier. Lethal epistasis could be implemented using a `modifyChild()` callback that suppresses the generation of an offspring if it inherits both lethally epistatic alleles from its parents. Selection could even be made to depend upon interactions between individuals, or upon nongenetic state such as nutrition or social status.
- **Selective sweeps**. Script can be used to initiate and track phenomena such as hard and soft selective sweeps, partial sweeps, and adaptive introgression. Script can trigger further simulation events to occur depending upon the state of a sweep, which allows a simulation to be made conditional on the establishment or fixation of a sweep by resetting the simulation to a previous (saved) state if the sweep mutation is lost.
- **Complex mating schemes**. Mating schemes such as assortative mating and sequential mate search can be implemented easily using a `mateChoice()` callback. Mate choice could depend upon the genetics of the parents (for assortative mating, for example), upon nongenetic factors such as health or social interactions (which can be tracked using user-defined "tags" on individuals), or upon simulation state such as the mean relative fitness of the population as a whole (to simulate mate choice that becomes more "choosy" when there are more potential mates from which to choose, for example).

Note that this is not a list of hard-coded features available in SLiM; it is merely a list of some ideas for models that one could write in Eidos. If you can implement it in Eidos, you can do it in SLiM. To illustrate that, we also include a few fairly unusual recipes in the SLiM manual, such as a model of social learning of culturally inherited traits that influence fitness, and a model of gametophytic self-incompatibility based upon an S-allele system that produces balancing selection.

The SLiM 2 Simulation Model

At the simplest level SLiM is based upon a Wright–Fisher-type model of evolution (Ewens 2004); in particular, (1) generations are nonoverlapping 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, but this is the foundational model.

SLiM allows arbitrary population structure; any number of subpopulations may exist, connected by any pattern of migration, and subpopulations may come into existence, change size, and be removed at any time. Migration occurs at the juvenile stage, and mate choice occurs within subpopulations.

SLiM can model either hermaphrodites or 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 supported instead of, or in addition to, biparental mating. In the hermaphroditic case, SLiM also supports self-fertilization. When modeling sexual individuals, the sex ratio is controllable, and sex chromosomes may be modeled.

SLiM is genetically explicit in the sense that it models mutations at specific base positions in genomes with an explicit chromosome structure; SLiM does not, however, model nucleotide sequences. The chromosomes are composed of genomic elements (e.g., sections of a gene), each of a particular genomic element type (e.g., intron vs. exon). The genomic element type defines the mutational profile of elements of that type, using a set of mutation types and associated probabilities.

By default, SLiM calculates fitness multiplicatively, based upon all of the mutations possessed by each individual. However, the fitness and dominance effects of mutations may be altered in script, providing 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.

Within each generation, events occur in a fixed order (fig. 1). Each generation begins with the execution of script-defined `early()` events. Offspring are then generated from biparental mating, selfing, and cloning as specified by the script; biparental mating can involve random mate choice (with probabilities proportional to fitness) or specific mating preferences defined by `mateChoice()` callbacks. After offspring have been generated, their genomes can be modified by `modifyChild()` callbacks defined in the simulation script. 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; e.g., this is a good opportunity to generate model output. Fitness values are then calculated, modified by any `fitness()` callbacks defined in script. Finally, the simulation advances to the next generation.

An Example SLiM 2 Simulation

To illustrate how setting up and running a simulation works in practice, we will construct a simple toy model to which we will add successive levels of evolutionary complexity. This will demonstrate several key concepts regarding how evolutionary scenarios are specified in Eidos, including population structure and demography, fitness and selection, mate choice, and explicit genetic modification of individuals.

We will begin by modeling a genomic region with uniform mutation and recombination rates, evolving in a six-population island model with a predominant directionality of gene flow. The Eidos code for this simulation (Code Sample 1) can be entered directly in SLiMgui, or can be provided to SLiM on the command line as a text file.

SLiM's manual spells out the details of the various calls made here; for our purposes, it suffices to say that an `initialize()` callback (the first half of the script) sets up basic simulation parameters like the mutation rate, the recombination rate, the structure of the genomic region, and the fitness characteristics of the mutations that can occur. In our case, mutation type `m1` specifies neutral mutations that occur uniformly over a 100-kb-long genomic region. An event that runs in generation 1 (the bottom half of the script) sets up an "island chain" of six subpopulations with 500 individuals each, and initializes the migration rates along the chain to be greater in one direction than in the other. Finally, an event at generation 10000 outputs the state of the population, including information on all segregating mutations; since nothing is defined as happening after generation 10000, the simulation ends at the end of that generation. Note that SLiM simulations begin with empty genomes (i.e., with no pre-existing mutational variation) unless you write script that explicitly adds such variation to the initial population. When studying patterns of genetic diversity, simulations should therefore include a "burn-in" period to establish diversity

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

  // Will be used for a driver mutation in a subsequent example
  initializeMutationType("m2", 0.5, "f", 0.5);
}

// Add subpops and set up stepping-stone migration
1 {
  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);
}

// Output the population state at the end of the simulation
10000 late() {
  sim.outputFull();
}
```

Code Sample 1

from new mutations. In the above model, for instance, levels of neutral diversity will have fully equilibrated by generation 10000.

Already we have a useful model, and it would be straightforward to set up a more complex genetic or population structure by modifying this script. Note that this script did not make a call to activate a built-in "island model" population structure feature. Instead, whatever features are needed are specified in just a few lines of Eidos code, as above. This means that SLiM does not restrict you to a particular feature set built into the package; you can do literally anything that you can write in Eidos, without having to understand or modify the C++ code underlying SLiM's implementation. Similarly, although the code above uses a pre-existing output method, `outputFull()`, to generate output at the end of the simulation, you can write Eidos code to produce whatever custom output you wish, as will be shown below.

The script above sets up a mutation type named `m2`, but does not yet use it. We can add some code to our script (Code Sample 2) to introduce a mutation of type `m2` in generation 5000, and to then monitor it for either fixation or loss.

The top code block adds the new `m2` mutation at the center of our region to 50 genomes within the population. The bottom block checks for fixation or loss of that mutation; if either occurs, it prints out a message (`FIXED` or `LOST`), outputs 10-genome samples from each subpopulation, and terminates the simulation. This is very simple, but it illustrates the fact that SLiM simulations can generate whatever customized output is desired based upon the current simulation state; example recipes in the SLiM manual produce more complex output, such as calculating and reporting the level of F_{ST} between subpopulations.

Since mutation type `m2` is configured (in the initialization code) to be beneficial, with a selection coefficient of 0.5, the introduced `m2` mutation will often sweep through all subpopulations and reach fixation. We can change that (Code Sample 3) by introducing spatial variation in the fitness effects of the `m2` mutation type.

The fitness effect of mutations of type `m2` is now calculated, on a per-individual basis, by this Eidos callback, rather than being determined by the fixed selection coefficient of 0.5 specified in the initialization code for the `m2` mutation type. For a given individual carrying an `m2` mutation, this code uses the identifier (from 0 to 5) of the subpopulation the individual is from to determine the fitness effect, making the mutation somewhat beneficial in the first subpopulation but strongly deleterious in the last. When the model is now run, the introduced mutation will typically reach migration-selection balance within the first subpopulation

```
// Add an m2 mutation to 50 genomes, at the center of the genomic region
5000 late() {
  mut = p0.genomes[0].addNewDrawnMutation(m2, 50000);
  p0.genomes[1:49].addMutations(mut);
}

// Print a message on fixation/loss of the m2 mutation
5000:10000 late() {
  if (sim.countOfMutationsOfType(m2) == 0)
  {
    fixed = any(sim.substitutions.mutationType == m2);
    cat(ifelse(fixed, "FIXED\n", "LOST\n"));
    for (i in 0:5)
    {
      cat("\n// sample for subpop " + i + ":\n");
      sim.subpopulations[i].outputSample(10);
    }
    sim.simulationFinished();
  }
}
```

Code Sample 2

```
// Spatial variation in the fitness effects of m2 mutations
fitness(m2) {
  return 1.2 - subpop.id * 0.15;
}
```

Code Sample 3

(limited by strong gene flow from the second), with almost no penetration into the other subpopulations.

We can modify this migration-selection balance by adding some code (Code Sample 4) to make the introduced m2 mutation a CRISPR gene drive allele, which converts heterozygotes for the allele into homozygotes, to create a so-called "mutagenic chain reaction" (Gantz and Bier 2015).

This code tells SLiM that whenever an offspring carries the m2 mutation in one of its two genomes, the mutation should be copied into the other (homologous) genome as well. Now that the m2 mutation is a driver allele, it will typically sweep through the first five subpopulations, even though it is deleterious in the later subpopulations. However, it is still unable to fix in the sixth subpopulation because it is so strongly deleterious there (with a relative fitness of only 0.45, according to the `fitness()` callback defined above).

In order to get the m2 driver allele to sweep all the way to fixation in the whole population, we will add a mating preference for m2 carriers. Biologically, this would reflect some phenotypic trait of m2 carriers—an attractive pheromone, perhaps—that causes them to be preferred as mates. Implementing this in our model is very simple (Code Sample 5).

This callback detects m2 carriers by leveraging the fact that their relative fitness is not equal to 1.0, whereas noncarriers have only neutral mutations and thus have a relative fitness of 1.0. It would be possible to explicitly base the mate choice on the presence or absence of the m2 mutation in each potential mate, rather than using the fitness values as a proxy, but the code would be slightly more complex (an example of this can be found in SLiM's manual). Carriers get a bonus of 0.1 added to their default mating weight (which is equal to their relative fitness). This small boost is enough, in combination with the conversion activity of the driver allele itself, for the m2 driver allele to now fix—although it can take a while, as it depends upon building a critical mass of carriers in the last subpopulation that overwhelms the selection against the driver.

Interactive Scripting in SLiMgui

The easiest way to explore the example model above is to enter it into SLiM 2's new graphical user interface, SLiMgui (fig. 2). The SLiMgui scripting environment provides a rich set of features to support interactive script development and testing and to ease the learning curve associated with scripting in Eidos (fig. 3). Its Eidos script editor includes features such as syntax coloring, code completion, context-sensitive help, and syntax checking. Script errors, both syntactic and semantic, are reported with highlighting of the specific operator or call that triggered the error. Simulations can be run interactively, typically with only a small speed penalty compared with running at the command line. When running a simulation, the frequency and selection coefficient of every mutation can be visualized, and the relative fitness of every individual is displayed. SLiMgui can help the user add new Eidos script for many common operations, from changing a migration rate to generating model output. For the maximum in flexibility, one can directly browse the state of all

```
// Implement CRISPR gene drive in offspring
5000:10000 modifyChild() {
  mut = sim.mutationsOfType(m2);
  if (size(mut) == 1)
  {
    mutInGenome1 = childGenome1.containsMutations(mut);
    mutInGenome2 = childGenome2.containsMutations(mut);
    if (mutInGenome1 & !mutInGenome2)
      childGenome2.addMutations(mut);
    else if (mutInGenome2 & !mutInGenome1)
      childGenome1.addMutations(mut);
  }
  return T;
}
```

Code Sample 4

```
// Implement a mating preference for driver carriers
5000:10000 mateChoice()
{
  carriers = (sourceSubpop.cachedFitness(NULL) != 1.0);
  return weights + asInteger(carriers) * 0.1;
}
```

Code Sample 5

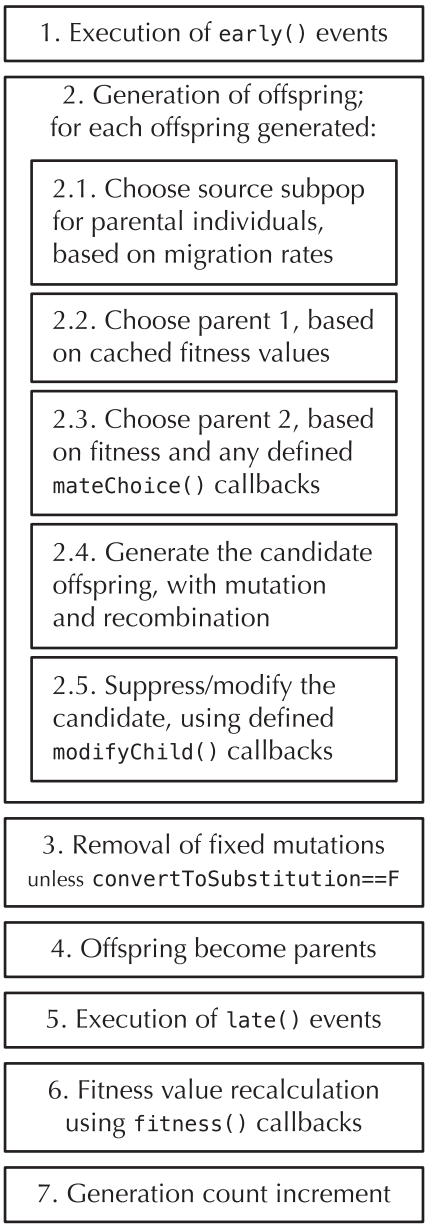

FIG. 1. The sequence of events within one generation in a SLiM simulation.

simulation objects, and can open an Eidos console to run commands that directly modify the simulation's behavior. Searchable help is provided within SLiMgui for all aspects of the Eidos language and all of the Eidos functions and classes defined by SLiM. SLiMgui can also display graphs summarizing various simulation metrics, from mutational trajectories to fixation times, with live updating of those graphs as the simulation runs.

Runtime and Memory-Usage Comparisons with Existing Software

Forward genetic simulations can be computationally intensive, and thus are often run with many replications on high-performance computing clusters. The runtime and memory footprint of a simulation can therefore be crucially important. To evaluate SLiM 2's performance, we compared it to the currently available versions of two other popular simulation packages, SFS_CODE (version 20150910) and fwdpp (version 0.4.6), as well as to its predecessor, SLiM 1.8. All test runs were conducted on 32-core Intel Xeon E5-4620 2.20GHz compute servers. All packages were compiled with g++ version 4.4.7 using makefile settings (if provided) or, in other cases, maximum optimization (-O3).

Figure 4 shows runtime and peak memory usage in simulations of a locus of length $L = 1$ Mbp in a population of size $N = 2{,}000$ diploid individuals, evolving under slightly deleterious mutations occurring at rate $u = 10^{-8}$ per site per generation and a recombination rate of $r = 10^{-7}$ per site per generation. These default parameters were varied individually (but not jointly) across a range of values: for L, {1, 2, 5, 10, 20, 50, 100} Mbp; for N, {500, 1000, 2000, 5000, 10000, 20000} individuals; for r, $\{1, 2, 5, 10, 20, 50, 100\} \times 10^{-8}$; and for u, $\{1, 2, 5, 10, 20, 50, 100\} \times 10^{-9}$. Each parameterization was run for $20N$ generations, in order to emphasize the runtime and memory usage of the packages after diversity had accumulated. All simulations used a fixed selection coefficient of $s = -0.001$ for all mutations, which noticeably reduces fixation probabilities of these mutations in the default scenario, while still allowing for some level of polymorphism to be maintained in the populations. Mutations occurred uniformly along the simulated locus, the population size was constant within each run, and all simulations started with "empty" genomes (i.e., with no segregating mutations). Each parameterization was run in 10 replicates using each software package.

Although there was variation in the memory usage among the packages, the memory usage of all packages was within reasonable limits of ~1 GB or less (fig. 4). Memory usage on our test servers was subject to a 24-MB minimum allocation, as indicated by the presence of a "floor" usage for even the smallest runs.

SLiM 2 was on average 1.8 times faster than SLiM 1.8, 1.3 times slower than the C++ template library fwdpp, and 12.0 times faster than SFS_CODE without its recently introduced "locus optimization" procedure. SLiM 2 was still 2.3 times faster than SFS_CODE when using locus optimization. Locus optimization in SFS_CODE requires the user to first run a Perl script that estimates the optimal subdivision of a long chromosome into a series of linked loci, which improves SFS_CODE's performance due to the internal details of its implementation. Unfortunately, this script can itself take a very long time to complete—more than 50 h for one of the parameterizations used in our comparison. An additional limitation is that the script accepts only a small subset of the configuration parameters supported by SFS_CODE, rendering the locus-optimization approach unusable in many circumstances. We also noticed that the average nucleotide heterozygosity (π) of locus-optimized runs of SFS_CODE diverged from the same metric measured for SFS_CODE runs without locus optimization and runs of SLiM and fwdpp (all of which agreed closely), indicating a potential problem with the optimization procedure that

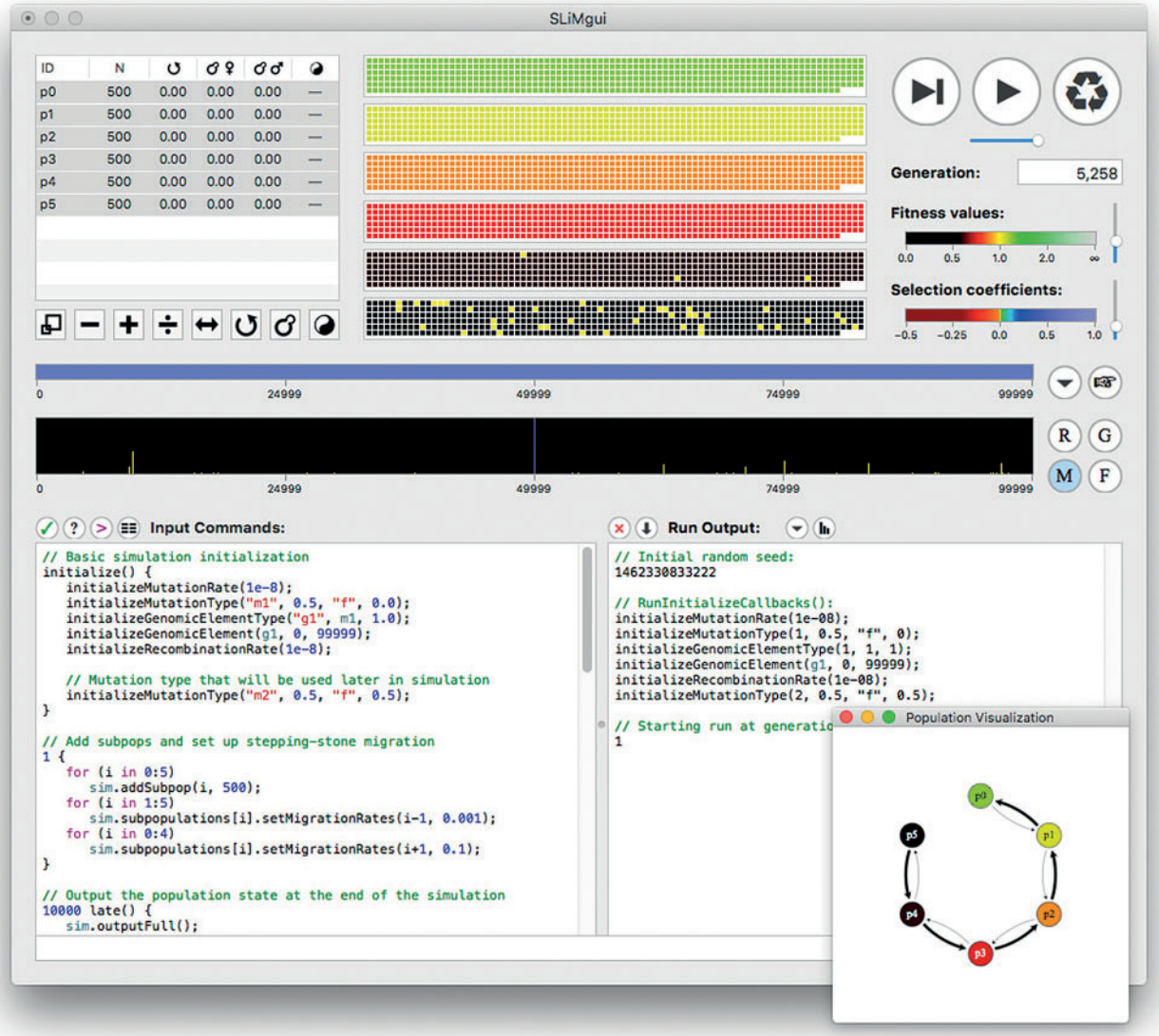

FIG. 2. A screenshot of SLiMgui running the example model presented in the text, taken at a moment just prior to fixation of the driver allele. The script editor is at lower left, and model output is displayed at lower right. At top center the six subpopulations are displayed, with each colored box representing one individual colored according to relative fitness (toward green in the subpopulations in which the driver allele is beneficial, toward red and then black where it is deleterious). The black box at the center of the window displays a view of the chromosome, showing the frequencies of all mutations in the population; the tall blue bar at its center represents the driver allele, near fixation, whereas the yellow bars represent neutral mutations at various frequencies. The floating subwindow at bottom right shows a visualization of the population structure, with each subpopulation colored according to mean relative fitness, and arrows between the subpopulations showing migration rates according to their thickness (in this case, showing higher migration rates in one direction vs. the other). The buttons at top right allow the user to step, play, or restart the simulation.

can distort results. We have called this issue to the attention of the SFS_CODE developers.

In addition to the above simulations with slightly deleterious mutations, we also conducted neutral simulations for all three packages, using the same parameter values and run lengths except for setting $s = 0$ for all mutations. The results were qualitatively very similar to the results presented in figure 4 (data not shown), the main difference being that the gap in performance between the various packages was slightly larger; in the neutral simulations, SLiM 2 was an average of 3.4 times faster than SLiM 1.8, 37.9 times faster than SFS_CODE, 3.2 times faster than the locus-optimized SFS_CODE runs (with the caveats above), and 1.6 times slower than fwdpp.

After we had completed the performance tests shown in figure 4, we received additional benchmarks from the author of fwdpp (Thornton KR, personal communication). We include those results here because they may shed light on performance tradeoffs between SLiM 2 and fwdpp at large population sizes ($N \geq 10^4$) with high levels of diversity, which may be important in some applications. These tests used a newer fwdpp version (0.5.0) and the current development head version of SLiM 2, and ran on a 2.30 GHz 40-core Intel Xeon E5-2560 machine with 512 GB of RAM. Five

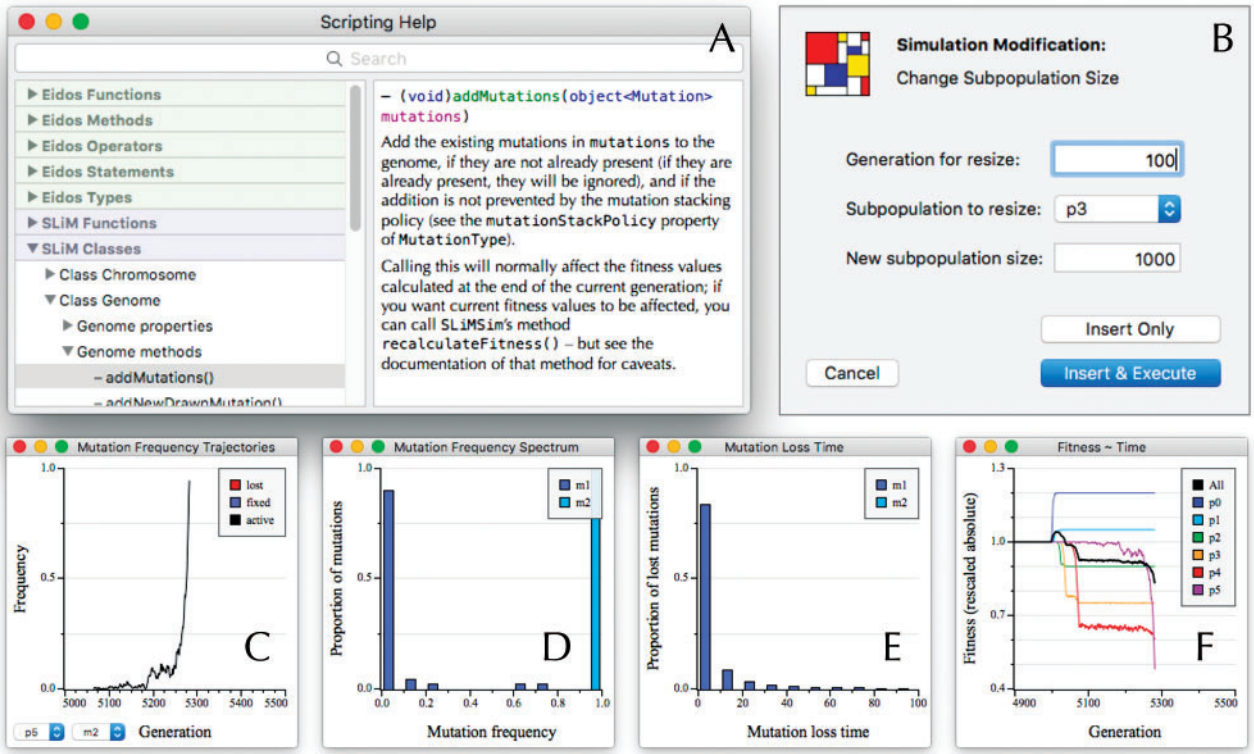

FIG. 3. Screenshots of some of SLiMgui's interactive features. (*A*) SLiMgui's help window, displaying information about one of the methods in the Genome class provided by SLiM. (*B*) A SLiMgui panel assisting the user in adding a new script command to change the size of a subpopulation. (*C*–*F*) Screenshots of SLiMgui graphs for a run of the example model presented in the text, taken just prior to fixation of the driver allele. (*C*) A graph of the frequency trajectory of the driver allele in the last subpopulation, showing that it introgressed into that subpopulation around generation 5070, fluctuated in frequency for about 100 generations, and then accelerated rapidly toward fixation. (*D*) A graph of the mutation frequency spectrum, showing that the driver allele (mutation type `m2`) is near fixation while most neutral mutations (type `m1`) are at low frequency. (*E*) A graph of mutation loss times, showing that most neutral mutations are lost quickly in this model. (*F*) A graph of population mean fitness as a function of time, showing an initial spike in fitness after the introduction of the driver allele as it spread through the first subpopulation, where it was beneficial, followed by a decline in fitness as it propagated through the subpopulations where it was deleterious.

replicates were done for each of two test cases, both involving only neutral mutations, with a chromosome length of 10^7 base positions and a duration of 10^5 generations. In the first test case ($N = 10^4$, $r = 2.5 \times 10^{-9}$, $u = 2.5 \times 10^{-9}$), SLiM had an average runtime of 10,102 s with average peak memory usage of 467 MB, whereas fwdpp was benchmarked at 1,564 s and 90 MB. In the second test case ($N = 10^4$, $r = 2.5 \times 10^{-8}$, $u = 2.5 \times 10^{-8}$) SLiM averaged 189,237 s and 4,677 MB, versus fwdpp's 67,727 s and 3,069 MB. The variation around these mean values was small (CV $< 3.5\%$ in all cases). These results indicate that in the tested parameter regimes, fwdpp was 2.8–6.5 times faster than SLiM (with SLiM having a maximum runtime of ~2.2 days, to fwdpp's 0.78 days), and 1.5–5.2 times more memory-efficient (with SLiM having a maximum memory usage of 4.7 GB, to fwdpp's 3.1 GB).

These performance evaluations provide only a snapshot for a specific hardware configuration and software version, and will likely change with subsequent releases of the various software packages. Nevertheless, the fact that SLiM 2 is faster than previous versions of SLiM, substantially faster than SFS_CODE, and competitive with the highly optimized C++ template library fwdpp, illustrates that the added flexibility and ease of use of SLiM 2 do not come at the price of a performance penalty. Particularly in applications where absolute population sizes cannot be effectively rescaled (as is commonly done), some users will find that fwdpp is somewhat more efficient than SLiM 2 when simulating very large populations. We note, however, that fwdpp is very different in scope from SLiM 2 in that it is a C++ template library, not a full simulation package, and thus requires C++ programming skills to use.

Conclusions and Availability

In this article, we have provided a brief introduction to SLiM 2, a new version of the SLiM software package for forward population genetic simulation that introduces scriptability for model specification and simulation control. As an illustration of the immense flexibility of SLiM 2, we presented a simulation example of an island model of a CRISPR gene drive, with spatial variation in fitness costs and mating preference for carriers of the driver allele—in just a few lines of Eidos script. We also introduced SLiMgui, a powerful interactive application for building and testing SLiM models. Finally, a comparison between SLiM 2 and the software packages SFS_CODE and fwdpp illustrated that SLiM 2 is highly competitive in speed and memory usage; its increased flexibility relative to SLiM 1.8 does not come at the price of decreased performance. Indeed,

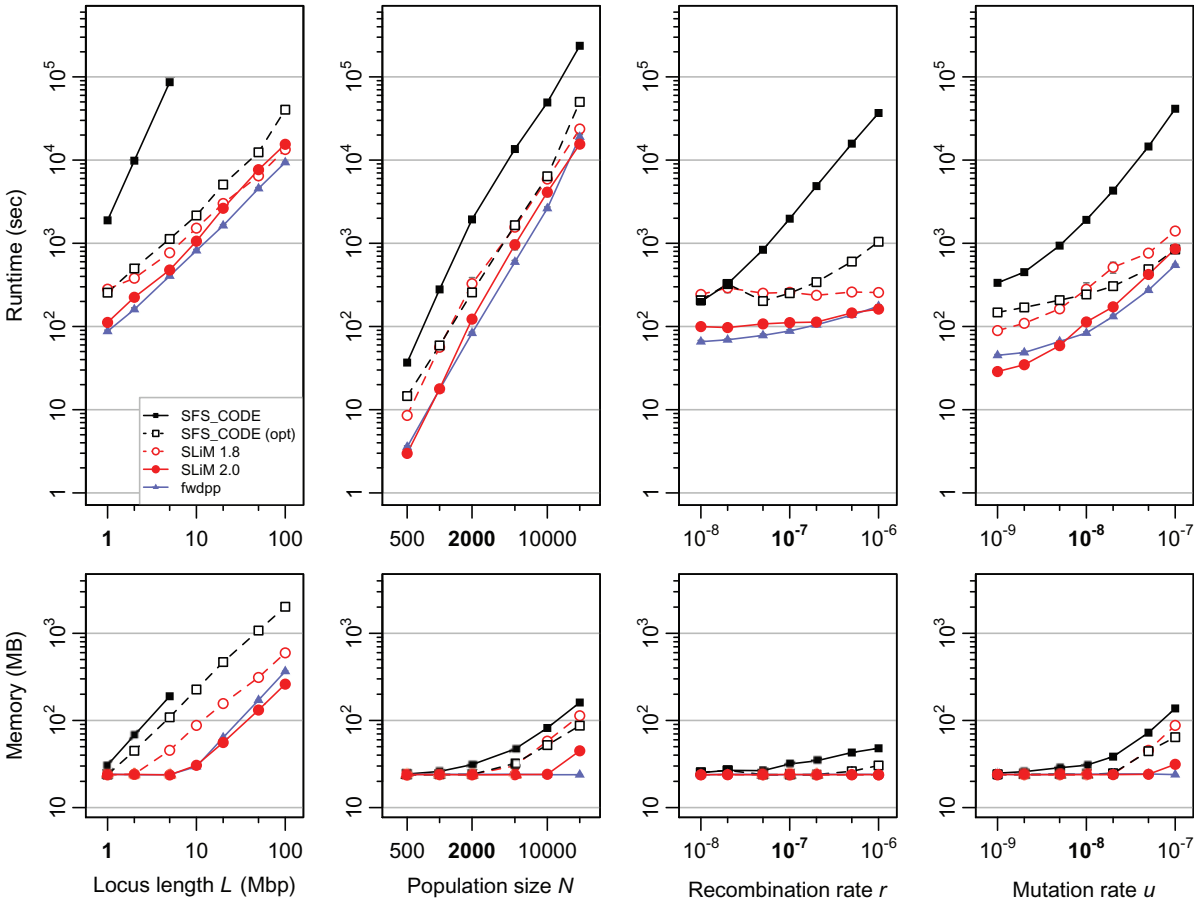

FIG. 4. Speed and memory usage for simulations run with SLiM 2 and other simulation packages (SLiM 1.8, fwdpp, SFS_CODE, and a locus-optimized version of SFS_CODE). The default value of each parameter is shown in bold. Each panel shows variation in one of the four parameters, holding the other parameters to the default values. All points represent 10 replicates, each run for 20*N* generations. All panels are plotted on a log–log scale. Parameterizations with runtimes >5.5 days (5×10^5 s) were terminated and are not shown; this applied only to SFS_CODE runs. Standard error bars are shown but are typically smaller than the plot symbols, and thus are rarely visible.

SLiM 2's performance is sufficient to simulate even full-length chromosomes with realistic population sizes in many scenarios.

The presentation of SLiM 2 here is, by necessity, quite incomplete; the SLiM manual (Haller and Messer 2016) provides complete reference documentation for SLiM, and also contains a large number of example recipes to ease the learning curve. In addition, the Eidos manual (Haller 2016) provides a full introduction to the Eidos scripting language, but it will probably be unnecessary for users familiar with languages such as R, C++, or Java, except as an occasional reference. SLiM itself is freely available on its home page at https://messerlab.org/slim/ as either an installer for Mac OS X, or as a source archive that can be built on Linux. SLiM uses code from the GNU Scientific Library (Galassi et al. 2009) and Boost (Boost 2016), but that code has been integrated into SLiM so that building SLiM does not require the installation of those other libraries; as a result, building SLiM on Linux simply requires a single "make" command. SLiM is free and open-source, licensed under the GNU General Public License version 3, and the current development version of its code is available on GitHub (https://github.com/MesserLab/SLiM), although the downloadable release at the SLiM home page is recommended for most users instead. We provide mailing lists for announcements and discussion regarding SLiM (links provided at the SLiM home page), and we will be happy to receive emails regarding SLiM as well. If you use SLiM, we ask only that you cite this article in your publications.

From the beginning, SLiM was intended to provide both flexibility and efficiency in the same package (Messer 2013); with SLiM 2, that formula has taken a quantum leap forward.

Acknowledgments

We thank Simon Aeschbacher, Chenling Antelope, Tom Booker, Michael DeGiorgio, Ryan Hernandez, Peter Keightley, Emilia Huerta-Sanchez, Stefan Laurent, Kathleen Lotterhos, Andrew Marderstein, Sebastian Matuszewski, Mikhail Matz, Bruno Nevado, Etsuko Nonaka, Dmitri Petrov, Fernando Racimo, Peter Ralph, Aaron Sams, Derek Setter, Kevin Thornton, Rob Unckless, members of the Messer lab, the stackoverflow community, the reviewers, and the editor for their valuable comments and feedback. We further thank Kevin Thornton for providing performance benchmark data included in this article. This work was supported by funds from the College of Agriculture and Life Sciences at Cornell University to P.W.M.;

and by a National Science Foundation Graduate Research Fellowship to B.C.H. (grant number 1038597, 2010–2013).

References

Bank C, Ewing GB, Ferrer-Admettla A, Foll M, Jensen JD. 2014. Thinking too positive? Revisiting current methods of population genetic selection inference. *Trends Genet*. 30(12):540–546.

Boost. 2016. Boost C ++ libraries [version 1.59.0]. Available from: http://www.boost.org/community/index.html.

Carvajal-Rodríguez A. 2010. Simulation of genes and genomes forward in time. *Curr Genomics*. 11(1):58–61.

Charlesworth B, Charlesworth D. 2010. Elements of evolutionary genetics. Greenwood Village (CO): Roberts and Company Publishers.

Ellegren H. 2014. Genome sequencing and population genomics in non-model organisms. *Trends Ecol Evol*. 29(1):51–63.

Ewens WJ. 2004. Mathematical population genetics 1: theoretical introduction. New York: Springer-Verlag.

Ewing G, Hermisson J. 2010. MSMS: a coalescent simulation program including recombination, demographic structure and selection at a single locus. *Bioinformatics* 26(16):2064–2065.

Excoffier L, Foll M. 2011. fastsimcoal: a continuous-time coalescent simulator of genomic diversity under arbitrarily complex evolutionary scenarios. *Bioinformatics* 27(9):1332–1334.

Galassi M, Davies J, Theiler J, Gough B, Jungman G, Alken P, Booth M, Rossi F. 2009. GNU scientific library reference manual 3rd ed. United Kingdom: Network Theory Ltd.

Gantz VM, Bier E. 2015. The mutagenic chain reaction: a method for converting heterozygous to homozygous mutations. *Science* 348(6233):442–444.

Grimm V. 2002. Visual debugging: a way of analyzing, understanding and communicating bottom-up simulation models in ecology. *Nat Resour Model*. 15(1):23–38.

Guillaume F, Rougemont J. 2006. Nemo: an evolutionary and population genetics programming framework. *Bioinformatics* 22(20):2556–2557.

Haller BC. 2016. Eidos: a simple scripting language. Available from: http://benhaller.com/slim/Eidos_Manual.pdf.

Haller BC, Messer PW. 2016. SLiM: an evolutionary simulation framework. Available from: http://benhaller.com/slim/SLiM_Manual.pdf

Hendry AP. 2016. Eco-evolutionary dynamics. Princeton (NJ): Princeton University Press.

Hernandez RD. 2008. A flexible forward simulator for populations subject to selection and demography. *Bioinformatics* 24(23):2786–2787.

Hernandez RD, Uricchio LH. 2015. SFS_CODE: more efficient and flexible forward simulations. BioRxiv. doi: http://dx.doi.org/10.1101/025064.

Hoban S. 2014. An overview of the utility of population simulation software in molecular ecology. *Mol Ecol*. 23(10):2383–2401.

Hudson RR. 2002. Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics* 18(2):337–338.

Jerison ER, Desai MM. 2015. Genomic investigations of evolutionary dynamics and epistasis in microbial evolution experiments. *Curr Opin Genet Dev*. 35:33–39.

Kessner D, Novembre J. 2014. forqs: forward-in-time simulation of recombination, quantitative traits and selection. *Bioinformatics* 30(4):576–577.

Levy SF, Blundell JR, Venkataram S, Petrov DA, Fisher DS, Sherlock G. 2015. Quantitative evolutionary dynamics using high-resolution lineage tracking. *Nature* 519(7542):181–186.

Luikart G, England PR, Tallmon D, Jordan S, Taberlet P. 2003. The power and promise of population genomics: from genotyping to genome typing. *Nat Rev Genet*. 4(12):981–994.

Malaspinas A-S. 2016. Methods to characterize selective sweeps using time serial samples: an ancient DNA perspective. *Mol Ecol*. 25(1):24–41.

Markram H, Muller E, Ramaswamy S, Reimann MW, Abdellah M, Sanchez CA, Ailamaki A, Alonso-Nanclares L, Antille N, Arsever S, et al. 2015. Reconstruction and simulation of neocortical microcircuitry. *Cell* 163(2):456–492.

Messer PW. 2013. SLiM: simulating evolution with selection and linkage. *Genetics* 194(4):1037–1039.

Messer PW, Ellner SP, Hairston NG Jr. 2016. Can population genetics adapt to rapid evolution? *Trends Genet*. 32(7):408–418.

Neuenschwander S, Hospital F, Guillaume F, Goudet J. 2008. quantiNemo: an individual-based program to simulate quantitative traits with explicit genetic architecture in a dynamic metapopulation. *Bioinformatics* 24(13):1552–1553.

NCI (National Cancer Institute). 2016. Genetic simulation resources (GSR). Available from: https://popmodels.cancercontrol.cancer.gov/gsr/about/.

Palmer T. 2014. Climate forecasting: build high-resolution global climate models. *Nature* 515(7527):338–339.

Peng B, Kimmel M. 2005. simuPOP: a forward-time population genetics simulation environment. *Bioinformatics* 21(18):3686–3687.

Pond SL, Frost SD, Muse SV. 2005. HyPhy: hypothesis testing using phylogenies. *Bioinformatics* 21(5):676–679.

Schraiber JG, Akey JM. 2015. Methods and models for unravelling human evolutionary history. *Nat Rev Genet*. 16(12):727–740.

Spjuth O, Alvarsson J, Berg A, Eklund M, Kuhn S, Mäsak C, Torrance G, Wagener J, Willighagen EL, Steinbeck C, Wikberg JE. 2009. Bioclipse 2: a scriptable integration platform for the life sciences. *BMC Bioinformatics* 10(1):397.

Thompson JN. 2013. Relentless evolution. Chicago (IL): University of Chicago Press.

Thornton KR. 2014. A C ++ template library for efficient forward-time population genetic simulation of large populations. *Genetics* 198(1):157–166.