Supplementary Materials

for

SLiM 3: Forward genetic simulations beyond the Wright–Fisher model

Benjamin C. Haller, Philipp W. Messer

Dept. of Biological Statistics and Computational Biology
Cornell University, Ithaca, NY 14853, USA

This supplement will present four example models that illustrate the new features in SLiM 3 described in the main paper. The first two examples are of non-Wright–Fisher or "nonWF" models; the third and fourth examples are of models utilizing continuous space (which are also nonWF models, as it happens). The main paper should probably be read before these example models, since it explains the concepts underlying nonWF models and continuous-space models in SLiM. Readers who are completely unfamiliar with SLiM may wish to read chapter 1 of the SLiM manual (Haller and Messer, 2016) even before that, since it lays out the fundamental concepts of modeling in SLiM.

The example models shown here are also available as separate supplemental text files. We recommend that users download those supplemental text files, open them in SLiMgui on a Mac OS X machine if possible, and experiment with them there. Users on Linux or Un*x can instead use SLiM at the command line, but working in the graphical modeling environment of SLiMgui is much more conducive to learning and experimentation. Information on downloading and installing SLiM on various platforms may be found in the SLiM manual (Haller and Messer, 2016), in chapter 2.

See the SLiM manual (Haller and Messer, 2016) for many more examples, as well as reference documentation on SLiM.

After the examples, this supplement will present the results of performance comparisons between SLiM 2 and SLiM 3.2, including comparisons involving new features of SLiM 3 such as nonWF models, continuous space, and tree-sequence recording.

Supplementary Example 1: A simple nonWF model

Our first example is of a simple nonWF model that illustrates the basic structure and generation cycle involved. A `reproduction()` callback chooses a random mate and generates a single offspring. Density-dependent selection imposes a fitness cost upon individuals that depends on the population size, leading to an emergent population size that fluctuates stochastically around the carrying capacity of the model. Age structure is emergent too, with some individuals surviving through multiple generations (which might be thought of as years, if we assume one litter per year), while others perish as juveniles.

```
// initialize() callbacks configure the overall properties
// of the simulation; they are called first by SLiM
initialize() {
  // declare to SLiM that this is a nonWF model
  initializeSLiMModelType("nonWF");

  // define a constant K that we will use as a carrying capacity below
  defineConstant("K", 500);

  // set up a mutation type m1 to represent neutral mutations
  initializeMutationType("m1", 0.5, "f", 0.0);

  // tell SLiM to convert m1 mutations to substitutions upon fixation
  m1.convertToSubstitution = T;

  // define a genomic element type g1 representing neutral regions
  initializeGenomicElementType("g1", m1, 1.0);

  // define the chromosome as a single g1 region of length 1e5
  initializeGenomicElement(g1, 0, 99999);

  // define the mutation and recombination rates
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);
}

// reproduction() callbacks are called once per individual per generation
// to provide an opportunity for each individual to reproduce
reproduction() {
  // here the focal individual chooses a mate from p1 with a call to
  // sampleIndividuals(1), and then an offspring is generated by crossing;
  // the new offspring is added to the subpop of the focal individual
  subpop.addCrossed(individual, p1.sampleIndividuals(1));
}

// this event runs early in generation 1; it adds a subpopulation named
// p1 to the simulation, with initial size 10, to start the simulation
1 early() {
  sim.addSubpop("p1", 10);
}

// this event runs early in every generation; it scales the fitness of
// every individual in p1 according to the population size versus the
// carrying capacity, to implement density-dependent population regulation
early() {
  p1.fitnessScaling = K / p1.individualCount;
}

// in generation 2000, output all fixed mutations and stop (since there
// is no more to do, since no events are defined further in the future)
2000 late() {
  sim.outputFixedMutations();
}
```

First of all, the script sets up the overall structure of the model in its `initialize()` callback. The model is declared to be a nonWF model with the call to `initializeSLiMModelType()`; this tells SLiM to use the nonWF generation cycle and so forth. Next, a constant `K` is defined, representing the nominal carrying capacity of the model as we will discuss further below. The remaining lines of the initialize() callback will be familiar to users of SLiM 2 as standard simulation configuration steps. They declare that the simulation uses only a single mutation type, representing neutral mutations; that it uses only a single genomic element type, representing genomic regions that undergo mutations of that mutation type; that the chromosome is of length 10^5 and is composed of a single such genomic element type; that mutations occur at a rate of 10^{-7} per base position per generation; and that recombination occurs at a rate of 10^{-8} per base position per generation. These concepts are discussed in more detail in the SLiM manual (Haller and Messer, 2016).

Second, the constant K is used in the early() event to calculate a density-dependent fitness effect that is set into p1.fitnessScaling, a new property. In SLiM 3, both individuals and subpopulations have fitnessScaling properties, and the final fitness value for each individual is calculated (multiplicatively) from the fitnessScaling of the subpopulation and the individual, as well as the effects (as in SLiM 2) of all mutations and fitness() callbacks. This provides a convenient way of altering fitness values, on either an individual or subpopulation level, without having to implement a fitness() callback. Here, it scales the absolute fitness of every individual so that the population will expand up to the carrying capacity. Importantly, fitness in this model, by default, specifies viability – the probability that an individual survives the viability/survival stage of the generation cycle. Note that the carrying capacity will not be met exactly in every generation; instead, the population size will fluctuate naturally around K, depending upon stochastic variation in fitness-based mortality. The formula in the early() event will result in a dynamic equilibrium around a population size of K. This is because if p1.individualCount is less than K every individual will be assigned a fitnessScaling value greater than 1.0, whereas if p1.individualCount is greater than K then fitnessScaling values will be less than 1.0. However, this scaling only results in equilibrium at size K if other fitness effects are not present; one might think of it as a "baseline carrying capacity", from which the model might depart if other factors influencing survival or reproduction are present.

Third, a reproduction() callback has been defined. This callback is called once for each individual alive at the point of offspring generation, and is expected to generate offspring (if any) for the focal individual. Here, a mate is chosen at random from p1 with sampleIndividuals(), and then biparental mating between the focal individual and the chosen mate is conducted by addCrossed(), adding the resulting offspring to the subpopulation of the focal individual. This code is entirely generalizable; multiple offspring could be generated, multiple mates could be chosen, and offspring could be generated through cloning or selfing instead of crossing.

The last point to note is that convertToSubstitution is set to T for m1, allowing fixed neutral mutations to be converted into Substitution objects by SLiM. In a WF model this would be automatic; the default is convertToSubstitution=T. In nonWF models, the default is convertToSubstitution=F. This is because since fitness is absolute, not relative, it would not be generally safe to remove fixed mutations from the simulation (but since m1 is neutral and has no indirect effects in the model, it is safe in this case). Telling SLiM that it can remove fixed mutations allows the model to run much faster.

Individuals in this model live until they happen to die (rather than being predestined to die at the end of each generation, as in a WF model); the age structure is emergent, determined only by birth and death rates. In the Eidos console in SLiMgui, we can see what age structure happens to emerge by executing p1.individuals.age. This reveals that, at the end of one randomly chosen test run, just one individual was 9 generations old, a few were 8 or 7, and about half the population was 1. This makes sense: in this model one new offspring is generated for every living individual in each generation, and then an equal probability of death for every individual in each generation, combined with the density-dependent selection and the specified carrying capacity, should produce the observed age distribution.

Supplementary Example 2: Age structure and monogamy

This slightly more advanced nonWF model illustrates two features. The first is how the age structure of the model can be controlled, with a minimum age for reproduction and varying survival rates depending upon age. The second is how to implement monogamy with litter size drawn from a distribution.

```
// the initialize() callback is exactly as before
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 for each focal individual is more complex now
reproduction() {
  // only individuals of age >= 3 reproduce
  if (individual.age >= 3)
  {
      // they choose a mate that is also age >= 3
      mate = p1.sampleIndividuals(1, minAge=3);

      // then the draw a litter size from a Poisson distribution, mean 2
      litterSize = rpois(1, 2);

      // loop to create each new individual in the litter by crossing
      for (i in seqLen(litterSize))
          subpop.addCrossed(individual, mate);
  }
}

// we add a new subpop of size 10 exactly as before
1 early() {
  sim.addSubpop("p1", 10);
}

// density-dependence is as before, but we add age-based mortality
early() {
  p1.fitnessScaling = K / p1.individualCount;

  // individuals of age >= 5 get a multiplicative fitness scaling
  // of 0.1 (i.e., a 90% chance of dying due to old age)
  inds = p1.individuals;
  inds.fitnessScaling = ifelse(inds.age < 5, 1.0, 0.1);
}

// output and termination is exactly as before
2000 late() {
  sim.outputFixedMutations();
}
```

The `initialize()` callback is unchanged. The `reproduction()` callback specifies that only individuals of age `3` or older can reproduce, and the same requirement is enforced upon the chosen mate with `sampleIndividuals()`. A litter size is drawn from a Poisson distribution with mean `2`, and then all of the offspring in the litter are generated by calls to `addCrossed()`. Monogamy is modeled here, but any mating scheme could be enforced through the appropriate script.

The other difference is in the `early()` event, where an individual fitness effect, based upon age, is applied to the `fitnessScaling` property of the individuals. Individuals of age less than `5` receive no individual scaling (but still feel the fitness effects of density, mutations, etc.), whereas individuals of age `5` or greater receive a multiplicative fitness penalty of `0.1`; it is still possible to live to a ripe old age, but much less likely than in the previous model. Individuals in this model typically live to be `5` or fewer generations old, although outliers are possible. This age distribution was not specified as a model parameter, however; it is an emergent consequence of the mechanistic biological details specified, such as the minimum age of reproduction, the mean litter size, and the fitness effect of age.

Population size is also emergent in this model; we apply a "baseline carrying capacity" using a value of 500 for `K`, as in Example 1, but because fitness values are also influenced by age-based mortality, the actual equilibrium population size is a little lower than `K` (about 480). This is as it should be; the model is doing precisely what we have told it to do in script. If a different model of population regulation were desired, that could be implemented in the script instead; all of the power over population regulation is in the hands of the modeler with nonWF models. Of course, this tremendous flexibility comes at a price: the mechanism of population regulation must be carefully planned in order to prevent unrealistic population dynamics, such as runaway growth or unintended population extinction.

These same concepts apply similarly to selfing rate, cloning rate, sex ratio (in sexual models), and migration rates: in a nonWF model, the mechanistic biological details are specified, and the behavior of the model is emergent from that specification (various recipes illustrating these possibilities are shown in the SLiM manual).

Supplementary Example 3: A spatial mate choice model

We will start with a spatial model of individuals living on a homogeneous two-dimensional landscape. The population size is regulated through local competition, using an `InteractionType` object, `i1`, to tally up its effects on each individual. Individuals prefer nearby individuals as mates, using another `InteractionType` object, `i2`, to find a mate. Each offspring is generated from an independently chosen mate, and lands near its first parent in space. This model uses "periodic" boundary conditions; the landscape wraps around both horizontally and vertically, providing a seamless toroidal space that avoids any boundary effects. A beneficial mutation is introduced in generation `1000`, which might sweep through the population or might be lost. The model:

```
initialize() {
   initializeSLiMModelType("nonWF");

   // tell SLiM that this is a 2D spatial model with dimensions x and y,
   // and that both of those dimensions are periodic – they wrap around
   initializeSLiMOptions(dimensionality="xy", periodicity="xy");

   // K is now a "carrying-capacity density", a spatial analogue of
   // carrying capacity, but based upon local density; see text
   defineConstant("K", 2000);

   // S is the spatial interaction width, the width of the competition kernel
   defineConstant("S", 0.05);

   // m1 represents neutral mutations, as before
   initializeMutationType("m1", 0.5, "f", 0.0);
   m1.convertToSubstitution = T;

   // m2 represents beneficial mutations; note we do not tell SLiM to
   // substitute these, since they continue to affect absolute fitness
   initializeMutationType("m2", 0.5, "f", 0.1);

   // set up the chromosome as before, using only m1 mutations
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 99999);
   initializeMutationRate(1e-7);
   initializeRecombinationRate(1e-8);

   // define interaction type i1 for spatial competition, using a
   // Gaussian (normal or "n") kernel of width S
   initializeInteractionType(1, "xy", reciprocal=T, maxDistance=S * 3);
   i1.setInteractionFunction("n", 1.0, S);

   // define interaction type i2 for spatial mate choice, similarly
   initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.02);
   i2.setInteractionFunction("n", 1.0, 0.01);
}

reproduction() {
   // draw a mate by spatial interaction strength, using i2
   mate = i2.drawByStrength(individual, 1);

   // generate a litter of Poisson size, mean 0.1
   for (i in seqLen(rpois(1, 0.1)))
   {
      // if we found a nearby mate, cross; if not, self
      if (mate.size())
         offspring = subpop.addCrossed(individual, mate);
      else
         offspring = subpop.addSelfed(individual);

      // set up the offspring with a spatial position near the
      // focal parent, wrapping around the periodic boundaries
      pos = individual.spatialPosition + rnorm(2, 0, 0.02);
      offspring.setSpatialPosition(p1.pointPeriodic(pos));
   }
}
```

```
// set up the initial subpopulation as before, but now also set up initial
// positions for all individuals drawn from a uniform distribution
1 early() {
  sim.addSubpop("p1", 1);
  p1.individuals.setSpatialPosition(p1.pointUniform(p1.individualCount));
}

// early in generation 1000, add a new m2 (beneficial) mutation at position
// 20000 in the chromosome, to a single genome chosen at random from p1
1000 early() {
  sample(p1.genomes, 1).addNewDrawnMutation(m2, 20000);
}

// population regulation is now based on local density-dependence
early() {
  // evaluate interaction i1 and use it to get the total interaction
  // strength felt by each individual, exerted by nearby neighbors;
  // that total strength, normalized, scales each individual's fitness
  i1.evaluate();
  inds = p1.individuals;
  competition = (i1.totalOfNeighborStrengths(inds) + 1) / (2 * PI * S^2);
  inds.fitnessScaling = K / competition;

  // color individuals blue if they contain the sweep m2 mutation
  mut = sim.mutationsOfType(m2);
  if (mut.size())
      inds.color = ifelse(inds.containsMutations(mut), "blue", "");
}

// late in every generation, evaluate i2 so reproduction() can use it next generation
late()
{
  i2.evaluate();
}

// finish as before
10000 late() {
  sim.outputFixedMutations();
}
```

This model is complex enough that we can't go into great detail on it, but we can sketch its broad design. In the `initialize()` callback we specify a nonWF model with dimensionality `"xy"`, with both dimensions being periodic. A mutation type is defined for neutral mutations, and another for the beneficial mutation that will be introduced. The model's genetic structure is specified as usual. Finally, two interaction types are set up; `i1` is for spatial competition, using a Gaussian kernel of width `S` (with a maximum distance of `S*3`), whereas `i2` is for spatial mate choice, using a Gaussian kernel of width `0.01` (maximum distance `0.02`).

In the `reproduction()` callback the focal individual draws a mate from within the maximum interaction distance, weighted by interaction strength so that nearer individuals are more likely to be chosen, using `i2`. A litter is then generated, of a size drawn from a Poisson distribution, using crossing if a mate was found or selfing if not. The position of each offspring lands near its first parent, with accounting for the periodic boundaries of the model.

The next event creates an initial population of a single individual, and sets its initial position randomly. Selfing is thus essential for this model to first get started; but a larger initial population size may be used instead, of course. In generation `1000` a new `m2` mutation is introduced into a randomly chosen genome by the next event. The following `early()` event runs in every generation, and calculates the effect of local density upon the fitness of each individual using `i1` (the interactions of which are evaluated just prior to its use); the mathematical details of this type of local density-dependent regulation are explained in the SLiM manual (Haller and Messer, 2016). This local density dependence generates realistic spatial population dynamics, such as preventing excessive clustering and ensuring that individual fitness is influenced only by nearby individuals. This event also colors carriers of the beneficial mutation blue, for illustration purposes. Finally, `i2` interactions are evaluated in a `late()` event, just in time for them to be used by `reproduction()` callbacks at the beginning of the next generation; and then some token output is generated in generation `10000`.

If the beneficial mutation is not lost early on, it will typically sweep across the population slowly, with a clear spatial pattern in its spread, as shown in Figure S1. The spread is rather slow in this model (compared to a panmictic WF population) partly because it is limited by the dispersal of individuals in this spatial model, and partly because individuals in this model tend to be quite long-lived (with a mean age of approximately `10`).

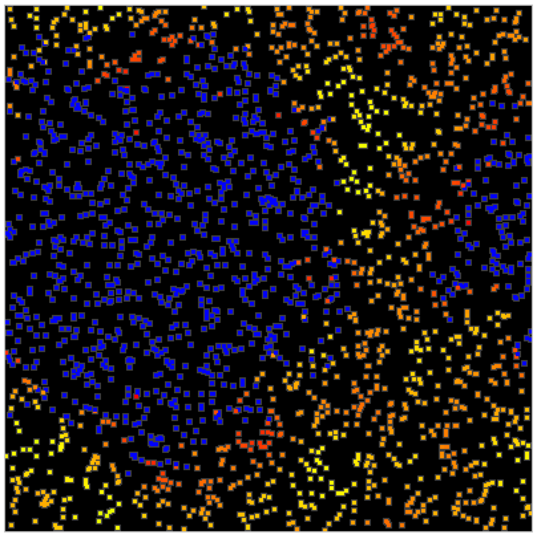

Figure S1. A snapshot of Example 3 at generation 1300, showing the spatial sweep of a mutation (blue) that was introduced in generation 1000. Spatial boundaries in this model are periodic, as seen in the way the sweep bridges the far left and far right. Individuals that do not possess the sweep mutation are colored to indicate their fitness, from yellow (neutral) to red (low); this variation in fitness is due to the effects of spatial competition, which regulates the population density. Only survivors after the selection stage of the generation cycle are visible, explaining what might otherwise appear to be incongruous fitness values.

Supplementary Example 4: Recolonization on a heterogeneous landscape

In this example, we introduce a spatial map that describes the habitability of the landscape. Areas of low habitability decrease the fitness of the individuals that occupy them, thereby decreasing the local population density. Occasional "disasters" will wipe out all individuals in a randomly chosen area of the map, leading to recolonization in a wave limited by habitability. The model:

```
initialize() {
  initializeSLiMModelType("nonWF");
  initializeSLiMOptions(dimensionality="xy");    // not periodic any more
  defineConstant("K", 2000);                     // carrying-capacity density
  defineConstant("S", 0.1);                      // spatial interaction width

  initializeMutationType("m1", 0.5, "f", 0.0);   // neutral only
  m1.convertToSubstitution = T;
  initializeGenomicElementType("g1", m1, 1.0);
  initializeGenomicElement(g1, 0, 99999);
  initializeMutationRate(1e-7);
  initializeRecombinationRate(1e-8);

  // interaction types; note i2 no longer needs an interaction function
  initializeInteractionType(1, "xy", reciprocal=T, maxDistance=S * 3);
  i1.setInteractionFunction("n", 1.0, S);
  initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.1);
}

reproduction() {
  // draw the focal individual's nearest neighbor as a mate, using i2
  mate = i2.nearestNeighbors(individual, 1);

  // if a mate was found, mate with a 10% chance
  if (mate.size() & (runif(1) < 0.1)) {
      offspring = subpop.addCrossed(individual, mate);

      // reprising boundaries: keep drawing positions until we get a legal one
      do pos = individual.spatialPosition + rnorm(2, 0, 0.02);
      while (!p1.pointInBounds(pos));
      offspring.setSpatialPosition(pos);
  }
}

1 early() {
  sim.addSubpop("p1", 100);
  p1.individuals.setSpatialPosition(p1.pointUniform(p1.individualCount));

  // define a habitability map, based on a 10x10 grid of values in [0, K]
  p1.defineSpatialMap("habitability", "xy", c(10,10), runif(100, min=0, max=K),
      interpolate=T, valueRange=c(0.0, K), colors=c("black", "blue"));
}

early() {
  // look up the local carrying-capacity density at each individual's position
  inds = p1.individuals;
  for (ind in inds)
      ind.fitnessScaling = p1.spatialMapValue("habitability", ind.spatialPosition);

  // density-dependent regulation is now based on local carrying-capacity density
  i1.evaluate();
  competition = (i1.totalOfNeighborStrengths(inds) + 1) / (2 * PI * S^2);
  inds.fitnessScaling = inds.fitnessScaling / competition;

  // local extinction events with a probability of 0.001
  if (runif(1) < 0.001) {
      // choose a center, find all individuals in radius 0.4, and give them fitness 0.0
      center = p1.pointUniform(1);
      hit = inds[i1.distanceToPoint(inds, center) < 0.4];
      hit.fitnessScaling = 0.0;
      catn(sim.generation + ": BOOM!");
  }
}
```

```
// evaluate i2 in preparation for reproduction(), as before
late()
{
   i2.evaluate();
}

// finish as before
10000 late() {
   sim.outputFixedMutations();
}
```

The `initialize()` callback is similar to the previous model, but without periodic boundaries or beneficial mutations. The `reproduction()` callback now simply chooses the nearest individual as a mate, within the maximum distance of `i2`; `i2`'s kernel is unused and so was omitted. If a mate is found, there is a 10% chance of generating an offspring through crossing. "Reprising" boundaries are used here, meaning that new candidate positions for the offspring are drawn until a position within bounds is obtained.

The population initialization now sets up a spatial map, named `"hab"`, generated by 100 random values forming a 10×10 grid. Random values are between `0` and `K`; the local habitability value from this map will take the place of the so-called "carrying-capacity density" from the previous model. The spatial map is interpolated, and we tell SLiM – purely for purposes of display in SLiMgui – that values from `0.0` to `K` should be given colors from black to blue. The next `early()` event then uses this spatial map to look up the local carrying-capacity density for each individual from the spatial map, and competition scales with that value rather than with a global carrying-capacity density. Finally, with a probability of 0.001 in each generation, disasters occur: a random point is chosen, and all individuals within a radius of 0.4 of the epicenter are given a `fitnessScaling` value of `0.0`, which is lethal.

Recolonization of disaster areas will occur by spatial spread from the surviving areas. However, dispersal will be largely restricted to high-habitability corridors, preventing recolonization from some areas that are nearby but disconnected. A snapshot of these dynamics soon after a disaster is shown in Figure S2.

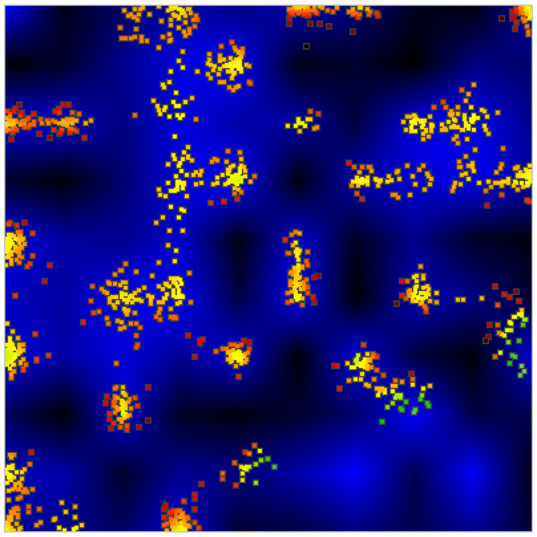

Figure S2. A snapshot of Example 4 at generation 657, after a disaster in generation 361 wiped out the individuals at lower right. Habitability, defined by a random spatial map with interpolation, is shown in background shades from black (uninhabitable) to blue (fully habitable). Recolonization of the disaster area is proceeding on three fronts, through corridors of relatively high habitability. Individuals at the three recolonization fronts are green, indicating their exceptionally high fitness because they are experiencing little competition from other nearby individuals.

Spatial maps, such as the habitability map defined in this model, can represent anything: real elevation values from an empirical study area, or levels of an environmental contaminant, or even calculated values related to the dynamics of the model, such as the mean age of individuals in the local area. Multiple spatial maps can be defined if desired; each is accessed by name. The possibilities are quite open-ended, and many more examples are shown in the SLiM manual (Haller and Messer, 2016).

Supplementary Results: Performance comparison

Since many forward genetic simulations are quite large, in terms of both population size and genome size, the runtime performance of software such as SLiM is often a very important consideration. For this reason, we devoted a great deal of effort, during the development of SLiM 3, to performance optimizations. The genetic engine at the heart of SLiM has been improved considerably, using new algorithms that can share stretches of genetic data between genomes for greater efficiency. Tree-sequence recording also offers the possibility of greatly improved performance, particularly by allowing simulation of neutral mutations to be deferred. On the other hand, some of the new features of SLiM 3, such as nonWF models and continuous-space models, are more complex than the vanilla WF models of SLiM 2, and are thus inevitably slower. To provide a better sense of the performance tradeoffs involved in such choices, we conducted a performance comparison between SLiM 2 and several types of SLiM 3 models.

These comparisons were all done with a simple neutral model of a population of *N* diploid individuals with a single uniform locus of length *L*, with recombination rate *r* and mutation rate *u*, run for *T* discrete, non-overlapping generations. Default values of these parameters were $N = 10{,}000$ individuals, $L = 10^7$ base positions (10 Mbp), $r = 10^{-8}$ per site per generation, and $u = 10^{-8}$ per site per generation. These default parameters were varied individually (but not jointly) across a range of values: for *L*, $\{10^5, 10^6, \mathbf{10^7}, 10^8, 10^9\}$ base positions; for *N*, {2000, 5000, **10000**, 20000, 50000} individuals (default values shown in bold). The values of *r* and *u* were not varied in these simulations; see Haller and Messer (2017) for an illustration of their likely qualitative effect. A run length of $T = 20N$ generations was used in all cases so that the measured runtime and memory usage would be dominated by the period after substantial diversity had accumulated. All mutations in these models were neutral, mutations occurred uniformly along the locus, the population size was constant within each run, and all simulations started with no segregating mutations.

For each parameterization, five variants of the model were run: (1) a SLiM 2.0 Wright–Fisher model (SLiM 2.0 WF), (2) a SLiM 3.2 Wright–Fisher model (SLiM 3.2 WF), (3) a SLiM 3.2 Wright-Fisher model with tree-sequence recording enabled and all mutations turned off (SLiM 3.2 WF tree-seq), (4) a SLiM 3.2 non-Wright–Fisher model (SLiM 3.2 nonWF) with non-overlapping generations and random mate choice so that it closely resembles the dynamics of the Wright-Fisher models, and (5) a SLiM 3.2 non-Wright–Fisher model with continuous space (SLiM 3.2 nonWF spatial), with nearest-neighbor mate choice and short-distance dispersal. Example scripts using the default parameter values for each variant are supplied in a supplementary zip archive.

Ten replicates of each parameterization for each variant were run. In total, then, 10 parameterizations × 5 variants × 10 replicates = 500 runs were conducted. However, runs were limited to a maximum runtime of 16 days, which precluded the completion of some parameterizations for some variants (and some of the largest runs were terminated early when extrapolation showed that they would clearly take longer than that time limit). All runs were executed on 32-core Intel Xeon E5-4620 v2 2.60GHz compute servers, with one physical core per instance, using a prerelease build of SLiM 3.2. Peak memory usage was measured with the `-m` command-line option to `slim`, and total runtime was measured with the `-t` option.

Simulation runtimes from this comparison are shown in Figure S3 (panels A and B). The direct comparison between SLiM 2.0 WF and SLiM 3.2 WF shows that SLiM 3.2 was always faster than SLiM 2.0, across the parameter values explored. This speed advantage was most pronounced for larger *N* and *L*; a speedup of slightly over 9× was observed for $L = 10$ Mbp, and of slightly over 10× for $N = 20{,}000$ individuals, the largest runs that can be compared.

Enabling tree-sequence recording – comparing SLiM 3.2 WF to SLiM 3.2 WF tree-seq – resulted in greatly reduced runtimes for the largest models, with a speedup of over 23× at $L = 100$ Mbp, and of slightly over 5× at $N = 20{,}000$ individuals. Given the observed trends, the speedup would probably have been even larger for $L = 1000$ Mbp and for $N = 50{,}000$ individuals, but the SLiM 3.2 WF runs for those parameter values did not complete in 16 days. This speedup from tree-sequence recording is a result of being able to turn off all neutral mutations; with the recorded tree sequence, mutations can be overlaid after forward simulation in a matter of seconds, saving a large amount of runtime. However, simulations that do not involve a large number of neutral mutations will see little benefit (or even a performance decrease) with tree-sequence recording enabled, and small-scale models may not benefit due to the runtime overhead incurred by tree-sequence recording. On the other hand, if "recapitation" can be used to completely eliminate forward simulation for a neutral burn-in period, the performance benefit can be much larger. These considerations are discussed in detail, with more extensive performance comparisons, in Haller et al. (2018).

Switching to a non-Wright–Fisher model – comparing SLiM 3.2 WF to SLiM 3.2 nonWF – entailed a performance penalty, because nonWF models are more general and flexible than WF models. That penalty was smaller for larger *L* (and for larger *N*, to a lesser extent), since the overhead of the nonWF model becomes less relevant as managing genomes demands more processor time. For *L* = 100 Mbp, the runtime of the nonWF model was about 1.7× the WF model, and for *N* = 20,000 individuals, the runtime was about 1.6× (these being the largest *L* and *N* for which these runs completed in 16 days). For large models, then, switching to nonWF should entail a relatively small performance penalty, with runtimes still far faster than with SLiM 2.0; and for small models, runtimes are generally short enough that performance is not a major concern.

Adding continuous space to a nonWF model – comparing SLiM 3.2 nonWF to SLiM 3.2 nonWF spatial – showed a similar pattern, with a penalty that was again smaller for larger *L* (and smaller *N*, to a lesser extent). For *L* = 100 Mbp, the runtime of the spatial model was about 1.3× the non-spatial model, and for *N* = 20,000, the runtime was about 1.1× (these again being the largest runs that completed in 16 days). This might suggest that the performance impact of adding continuous space to a large model is almost negligible. However, it should be noted that the model used here was quite minimal; an interaction type was used only to choose a mate using the `nearestNeighbors()` method, which requires only the building of the *k*-d tree internally. Interactions that also require the interaction-strength sparse array to be built internally will be slower; and the speed of interaction calculations will depend strongly upon the maximum interaction distances used and the spatial density of individuals on the landscape. It is difficult, then, to make strong predictions regarding the performance penalty involved in continuous space; in general it is necessary to actually build a model and test its performance. The penalty shown here should be regarded as a minimum, not a typical result; but it does show that the impact of adding spatiality to a model in SLiM can potentially be quite small.

Overall, the speedup from SLiM 2.0 to SLiM 3.2 is quite substantial, especially for the largest models, where the performance improvement is as much as an order of magnitude. When tree-sequence recording can be used to avoid modeling many neutral mutations, that can provide more than an order of magnitude of additional performance gain for the largest models. Using a nonWF model instead of the default WF model type entails a penalty, but not a very significant one in many applications. Spatial models entail a less predictable slowdown, but the performance penalty can be minimal if the maximum interaction distances defined by the model are short enough to keep the average number of interactions per individual small.

The patterns of memory usage among these runs are shown in Figure S3 (panels C and D); note that memory usage on the computing cluster we used has a floor of ~35 MB, so the actual memory usage of the smallest runs would be lower than the ~35 MB value shown. SLiM 3.2 WF showed considerably lower memory usage than SLiM 2.0 WF, especially for the largest models, due to its ability to share genomic regions between genomes internally. For smaller *L* and *N*, tree-sequence recording used quite a bit of memory here, but that is because a long interval between simplifications (250 generations) was chosen for better runtime performance; if memory is tight, simplification can be performed more frequently to decrease peak memory usage at the price of longer runtimes. For large *L* and *N*, the models with tree-sequence recording used less memory than all other variants even with this long simplification interval. The impact on memory usage of switching from WF to nonWF was fairly small, since the amount of genetic information being stored is similar. The impact on memory usage of adding continuous space was also small here, but again, that is quite contingent upon model details; models that have to build an interaction-strength sparse array, and that involve many interacting pairs of individuals (i.e., a long maximum interaction distance and/or a high density of individuals on the landscape) may incur a memory penalty that approaches scaling with N^2, so it is essential to keep maximum spatial interaction distances as small as possible for performance.

We did not conduct performance comparisons with non-neutral models. In Haller and Messer (2017) we conducted comparisons with both neutral and non-neutral models, and found that their results were qualitatively similar. We expect that to continue to be the case, since a great deal of SLiM's machinery is used identically by both neutral and selected models. In particular, models in which most, but not all, mutations are neutral should perform quite similarly to the results shown here (in fact, the speedup relative to SLiM 2 should be even larger, since some optimizations were done that specifically targeted this case). However, when very large numbers of non-neutral mutations are present in a model, the time taken to perform fitness calculations may dominate, and the other considerations discussed here may be somewhat overwhelmed. The performance benefit of tree-sequence recording will also be reduced as more of the mutations in a model become non-neutral, since only neutral mutations can be removed and overlaid after forward simulation.

In the end, SLiM's performance can be quite contingent upon a wide array of model details, and these results are just a snapshot of one type of model executed on a particular hardware and software platform with a particular version of SLiM. The best thing to do, if performance is at a premium, is to experiment and explore. The profiling feature in SLiMgui may be quite useful in such endeavors, since it provides a line-by-line breakdown of where the model script is spending its time. The SLiM manual (Haller and Messer, 2016) also provides fairly extensive discussion of model performance, including advanced techniques, such as recapitation and setting the mutation run count, that were not explored here for simplicity.

Nevertheless, the results shown here illustrate that SLiM 3.2 offers substantial performance benefits over SLiM 2.0 even when tree-sequence recording is not used, and that tree-sequence recording can provide large additional performance improvements for many models.

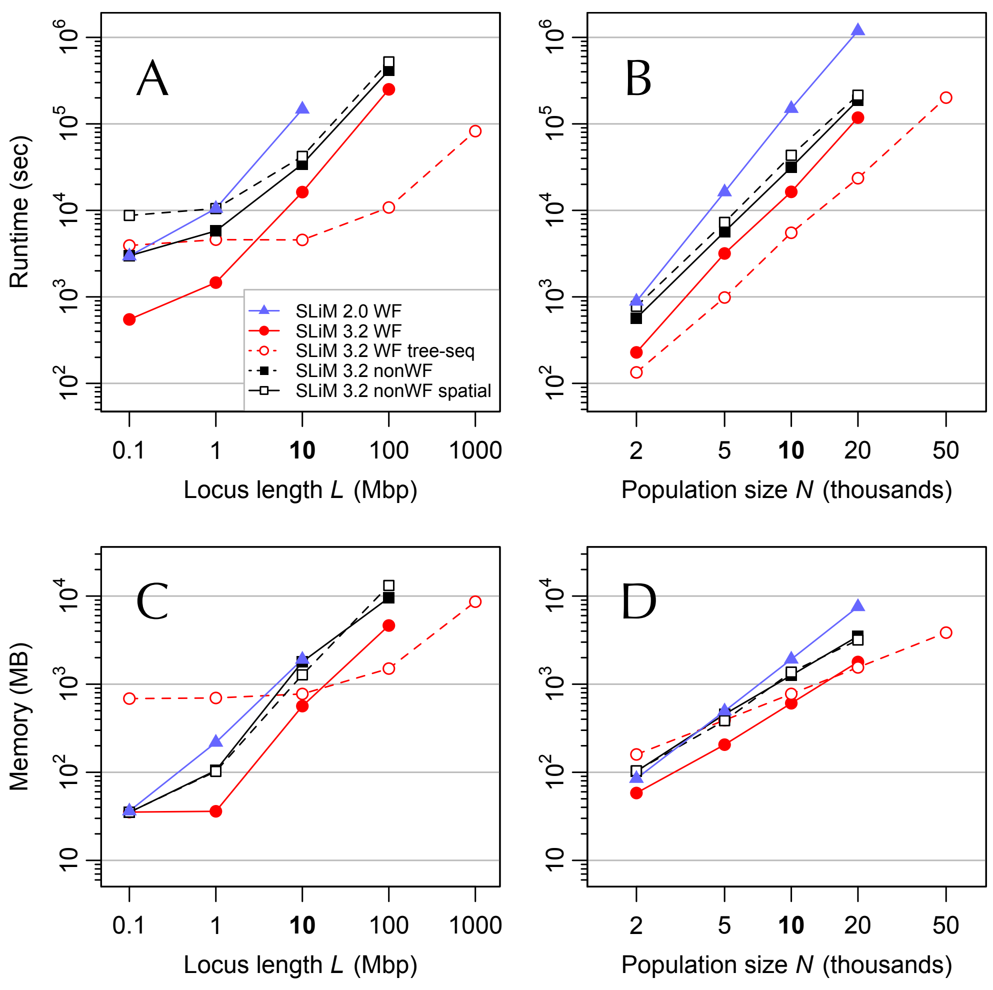

Figure S3. A performance comparison between SLiM 2.0 and various SLiM 3.2 models. Panels A and B show total runtime, whereas panels C and D show peak memory usage; note that a log scale is used for both axes in all panels. SLiM 2.0 WF runs (blue triangles) illustrate the baseline performance of the previous published version of SLiM, described in Haller & Messer (2017). SLiM 3.2 WF runs (filled red circles) provide the most direct performance comparison to the SLiM 2.0 runs. SLiM 3.2 WF runs with tree-sequence recording enabled and neutral mutations removed (hollow red circles) show the speedup available from the use of tree-sequence recording in large models that contain many neutral mutations. SLiM 3.2 nonWF runs (filled black squares) show the performance penalty associated with switching to non-Wright–Fisher dynamics. Finally, SLiM 3.2 nonWF runs with continuous space enabled (hollow black squares) show the additional performance penalty associated with adding continuous spatial dynamics and spatial mate search to a nonWF model. All models are neutral, with non-overlapping generations, and are panmictic (except the continuous-space models, which include local mate choice). Baseline parameter values were chromosome length $L = 10$ Mbp, population size $N = 10{,}000$ diploid individuals, recombination rate $r = 10^{-8}$ per base position per generation, mutation rate $\mu = 10^{-8}$ per base position per generation, and run length $T = 20N$ generations to try to ensure that runtimes predominantly reflect performance after genetic diversity has accumulated. Panels A and C show results for values of L in {0.1, 1, 10, 100, 1000} Mbp, using the baseline values of all other parameters; panels B and D similarly show results for values of N in {2, 5, 10, 20, 50} thousand individuals, using baseline values of all other parameters (but allowing T to vary with N). Each data point is based upon 10 replicate runs using different seeds; standard error bars are smaller than the plot symbols in all cases, and are thus omitted for clarity. Runs were terminated if they exceeded 16 days runtime. See the text for discussion of the patterns shown.

References

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

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

Haller, B.C., Galloway, J., Kelleher, J., Messer, P.W., and Ralph, P.L. (2018). Tree-sequence recording in SLiM opens new horizons for forward-time simulation of whole genomes. bioRxiv, DOI: https://doi.org/10.1101/407783