SLiM Workshop Series #9: Selective Sweeps

1. Start a new model in SLiMgui

Run SLiMgui, and close any existing model windows. Press command-N to make a new model window. Adapt the default script – which is close to what we want – to the following code:

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
1 { sim.addSubpop("p1", 500); }
100000 late() {
   catn("TIME EXPIRED");
   sim.simulationFinished();
}
```

Mostly you will need to just delete the output events and instead add the generation `100000` termination event shown above. You can also delete the comments in the default model if you wish.

What we're left with is a trivial neutral model that runs to generation 100000 and then prints a message and stops. The `catn()` function's name is short for "con**cat**enate to output with **n**ewline"; it prints what it is given to SLiM's output stream, adding a final newline character. (The `cat()` function may be used when a final newline is not desired.) We'll use `catn()` to produce more custom output in this model; it's a very useful tool.

2. Introduce a sweep mutation

Add this line to the `initialize()` callback, just below the definition of `m1` (it doesn't actually matter where):

```
initializeMutationType("m2", 1.0, "f", 0.5);  // introduced mutation
```

This creates mutation type `m2`, which will be used for the sweep mutation. This mutation type uses a fixed DFE with a selection coefficient of `0.5` (quite strongly beneficial) and a dominance coefficient of `1.0` (making it fully dominant). This will make it less likely that it will be lost due to drift when it is still at low frequency.

Next let's add an event that adds the sweep mutation (add it in chronological order in the script):

```
1000 late() {
   target = sample(p1.genomes, 1);
   target.addNewDrawnMutation(m2, 10000);
}
```

This first chooses a target genome, into which the sweep mutation will be introduced, by using `sample()` to take a sample of size `1` from `p1.genomes`. Second, it calls `addNewDrawnMutation()` to add a new `m2` mutation to the target genome at position `10000`. The selection coefficient will be drawn from the (fixed) DFE of `m2`; if a specific selection coefficient were desired instead (perhaps one not even normally within the mutation type's DFE), `addNewMutation()` could be used instead. If a random position along the chromosome were desired, it would be trivial to draw the position using `rdunif()`. (You might wish to look up `rdunif()` in SLiMgui's online help now; don't forget that that help panel is there!)

Note that it is important that this is done in a `late()` event; to see why, look at the WF generation cycle diagram on the SLiM reference sheet. Since it is introduced in a `late()` event, its fitness effect is evaluated by SLiM immediately after, in the fitness calculation phase, and so it will have the correct effect during the next offspring generation phase. If it were introduced in an `early()` event instead, SLiM would not recalculate fitness values prior to offspring generation, and so the carrier of the sweep mutation would not receive the fitness benefit it ought to in the first round of mating after introduction. This would make it much more likely that the introduced mutation would immediately be lost due to drift. Indeed, this is so likely to represent a bug that SLiM will emit a warning message if you add a new mutation in an `early()` event in a WF model.

3. Tracking the sweep mutation

If you Recycle and Play the model now, it will probably run so quickly that you will have trouble seeing whether the sweep mutation fixed or was lost; and in any case we'd like the outcome to be recorded in the model's output, and we'd like the model to stop when either fixation or loss occurs.

To achieve this, let's add a new event to the model that will track the sweep mutation's progress. Add this event immediately after the `1000 late()` event, so that it runs for the first time immediately after the sweep mutation is introduced by that event:

```
1000:100000 late() {
   sweep = sim.mutationsOfType(m2);

   if (size(sweep) == 0)
   {
      fixed = (sum(sim.substitutions.mutationType == m2) == 1);
      catn(sim.generation + ": " + (fixed ? "FIXED" else "LOST"));
      sim.simulationFinished();
   }
   else
   {
      freq = sim.mutationFrequencies(NULL, sweep);
      catn(sim.generation + ": " + freq);
   }
}
```

There's a lot going on here, so let's dissect it step by step. First of all, we ask SLiM for a vector of all `m2` mutations, using `sim.mutationsOfType()`. If the sweep mutation is still segregating, `sweep` will now refer to it. If it has been lost, the result will be an empty vector (the sweep mutation no longer exists); and if it has fixed, the result will similarly be an empty vector (the sweep mutation has been converted to a `Substitution` object). We can therefore use `size(sweep)` to distinguish whether it is lost/fixed or is still segregating.

If the size is zero (lost/fixed), we look for a `Substitution` object of type `m2`; if one exists, then the sweep mutation fixed, whereas if none exists it was lost. The strategy here should be familiar; we get `sim.substitutions`, do a vectorized property access to get the mutation type of each substitution, compare each element of that vector to `m2` with a vectorized comparison to get a `logical` vector, use `sum()` to count the number of `T` values, and compare that to 1. (One could devise other ways to do much the same thing, perhaps using `any()` or `all()` instead of `sum()`.)

Now we have a logical value, `fixed`, telling us whether the sweep mutation was fixed or lost. We call `catn()` to print an appropriate message, and then call `simulationFinished()` to terminate the model. The expression inside the `catn()` call does some new things, though! First of all, it uses operator `+` to assemble the output string; whenever either one of the operands to `+` is of type `string`, the operator will construct a result string by concatenation, rather than doing arithmetic addition (which would make no sense with a string operand). This is a very convenient way to assemble complex output strings. Second, there is the expression `(fixed ? "FIXED" else "LOST")`, which uses an operator you have not yet seen, operator `?else`. The formal name for this is a "ternary conditional operator", but it is very simple: it is like a compact little `if-else` statement in a single expression. If `fixed` is `T`, the result of the operator will be `"FIXED"`; if `fixed` is `F`, the result will be `"LOST"`. That result string will then be concatenated together with the rest of the output string, and then `catn()` will concatenate the output string to SLiM's output.

Recycle and Play now. You might get output like this:

```
1000: 0.001
1001: 0.001
1002: LOST
```

The mutation never got off the ground, and was lost almost immediately. Or you might see:

```
1000: 0.001
1001: 0.002
1002: 0.004
1003: 0.005
...
1337: 0.998
1338: 0.999
1339: 0.998
1340: FIXED
```

So now we are tracking the sweep mutation, and producing custom output that allows us to follow the trajectory of the sweep in detail. But many runs of the model result in loss of the sweep mutation; and perhaps we are really only interested in analyzing runs in which the sweep fixes. If the sweep mutation's selection coefficient were smaller (or even neutral, or even negative!), or if the sweep mutation were not completely dominant, the probability of fixation might become quite small, and to get a single run in which the sweep succeeds, we might have to do many, many runs of the model. This model uses a quick burn-in period of 1000 generations, but since a model's burn-in period is usually intended to produce mutation-drift equilibrium as a starting point for further simulation, large models might require a much longer burn-in period (and indeed, even this rather small model probably ought to have a burn-in period five or ten times longer). In such cases, running the burn-in over and over for all of the failed runs that we would ultimately discard would be quite painful. Is there a better way?

4. Making the sweep conditional

There is indeed a better way: writing a *conditional* model. A conditional model runs conditional upon a particular outcome; if the outcome is not attained, it restarts itself at some saved checkpoint and re-runs (with a different sequence from the random number generator), repeatedly if necessary, until the outcome is reached. Here we'd like the outcome to be fixation of the sweep mutation, and we'd like to "checkpoint" the model at the end of the burn-in period, just after the introduction of the sweep mutation.

The first step is to establish that checkpoint, by saving the state of the model at that point. We want the checkpoint to use a filename that is unique to each particular run of the model (that way if we're running multiple instances of the model on multiple cores, such as on a computing cluster, their checkpoint files won't collide with each other). So first, let's remember the random number seed we started with, as a unique identifier. Add this line to the very beginning of the `initialize()` callback:

```
defineConstant("simID", getSeed());
```

The `defineConstant()` function defines a global constant with the given name, giving it the value provided; so now we can use `simID` as a unique identifier, throughout the model. It is often good to define all of the parameters of a model up front, in `initialize()`, using `defineConstant()` as a way of making the parameters visible, collected in one spot, and easy to modify. That's a good habit to cultivate in your modeling work.

Next, modify the `1000 late()` event to have the additional line seen here:

```
1000 late() {
   target = sample(p1.genomes, 1);
   target.addNewDrawnMutation(m2, 10000);
   sim.outputFull("/tmp/slim_" + simID + ".txt");
}
```

This call to `sim.outputFull()` is passed a Unix-style file path, constructed using the string `+` operator to contain the random number seed as a part of the filename. The file is created inside the `/tmp` directory, which on Unix systems is almost universally used as a place where running programs can place temporary files. The contents of `/tmp` are deleted periodically, so don't put anything there that needs to last beyond the next reboot of the machine. For our purposes this is perfect; we can put our checkpoint file there and not have to worry about deleting it later.

The "full" output saved by `outputFull()` is essentially the population's current genetic state. Importantly, it does *not* include information about substitutions, nor does it contain any auxiliary model state that we might have put into defined constants, variables, `tag` values (we'll talk about them soon), `setValue()` keys (again, discussed later), and so forth. However, it is complete enough to serve as a checkpoint for the purposes of this model; it has all the information we need to restore the population to its state at this moment in time – every mutation in every genome in every individual in the population, including the sweep mutation introduced just before the checkpoint.

Finally, modify the sweep tracking event as follows:

```
1000:100000 late() {
   sweep = sim.mutationsOfType(m2);

   if (size(sweep) == 0)
   {
      if (any(sim.substitutions.mutationType == m2))
      {
         cat(simID + ": FIXED\n");
         sim.simulationFinished();
      }
      else
      {
         cat(simID + ": LOST - RESTARTING\n");
         sim.readFromPopulationFile("/tmp/slim_" + simID + ".txt");
      }
   }
   else
   {
      freq = sim.mutationFrequencies(NULL, sweep);
      catn(sim.generation + ": " + freq);
   }
}
```

As suggested above, we have switched to using `any()` to test whether an `m2` substitution exists, for concision. More importantly, we've made it so that whenever the sweep mutation is lost, we restart the model at the checkpoint by reloading the saved state using `readFromPopulationFile()`, with the same filename we saved to before. This call throws out the current state of the model (in which the sweep was lost) and completely replaces it with the saved state (in which the sweep mutation was just introduced). As a side effect of this, the generation counter is reset back to `1000`, when the checkpoint occurred.

Recycle and run the model now, and you should probably see something like this:

```
1000: 0.001
1776741714415: LOST – RESTARTING
1001: 0.001
1776741714415: LOST – RESTARTING
1776741714415: LOST – RESTARTING
1001: 0.001
1002: 0.002
1003: 0.005
...
1276: 0.999
1277: 0.997
1776741714415: FIXED
```

In this run, the model had to restart itself three times before it the sweep fixed.

5. Resetting the seed to allow efficient replication

When we reset the model back to the checkpoint, it takes a different path than it did in the previous attempt because the random number generator is in a different state; the random numbers that determine which individuals mate, where recombination breakpoints occur, etc., will be different after each reset, and so eventually a random sequence will occur that happens to propel the sweep mutation to fixation. If the sweep is unlikely to fix, this could take a large number of attempts.

To see this, try changing the selection coefficient for `m2`'s DFE to `0.0`:

```
initializeMutationType("m2", 1.0, "f", 0.0);  // introduced mutation
```

The sweep mutation will now be neutral, like all the other mutations in the model. Recycle and run, and you will probably see the model reset itself quite a few times before the "sweep" (if one can still call it that) happens to fix; a test run I just did required 1945 resets. This model is small and fast, so that is not so painful – the whole process only takes a couple of seconds – but if the model were larger, one might wish to be able to re-create a successful run, for later purposes of analysis or validation or whatever, without having to re-simulate all of the wrong paths. This is quite straightforward. You can add the following code just after the `readFromPopulationFile()` call:

```
setSeed(rdunif(1, 0, asInteger(2^62) - 1));
catn("# NEW SEED: " + getSeed());
```

This uses `rdunif()` to generate a new random seed, and resets the random number generator to that seed. That doesn't make any substantive difference to the way the model works – since the new seed was generated by the old random number generator state, we're still following a sequence that was set in stone from the very beginning of the model's execution, inexorably dependent upon the original seed value. However it does mean that we can now reset back to exactly this random number generator state later on, if we want to. We print out the chosen seed value with `catn()`, and when the run completes, we know that the last seed value printed is a seed value that, given the checkpoint state, will produce fixation. For the test run I did, the final seed value happened to be `3067997226`, with an initial seed value of `1776742942680`.

To reproduce the run while excluding all of the failed attempts, we can simply load the checkpoint file in generation `1`, and then set the seed to the value that is known to produce fixation. Here is the complete model needed to reproduce such a run from its saved checkpoint file (you do not need to create and run this yourself – it wouldn't run on your machine anyway, unless you created the correct checkpoint file for it to use first):

```
initialize() {
   defineConstant("simID", 1776742942680);
```

```
      initializeMutationRate(1e-7);
      initializeMutationType("m1", 0.5, "f", 0.0);
      initializeMutationType("m2", 1.0, "f", 0.0);  // introduced mutation
      initializeGenomicElementType("g1", m1, 1.0);
      initializeGenomicElement(g1, 0, 99999);
      initializeRecombinationRate(1e-8);
   }
   1 late() {
      sim.readFromPopulationFile("/tmp/slim_" + simID + ".txt");
      setSeed(3067997226);
   }
   1000:100000 late() {
      sweep = sim.mutationsOfType(m2);

      if (size(sweep) == 0)
      {
         fixed = (sum(sim.substitutions.mutationType == m2) == 1);
         catn(sim.generation + ": " + (fixed ? "FIXED" else "LOST"));
         sim.simulationFinished();
      }
      else
      {
         freq = sim.mutationFrequencies(NULL, sweep);
         catn(sim.generation + ": " + freq);
      }
   }
```

We define `simID` with the original seed value. In generation `1` we load the saved checkpoint, and then set the seed to the value known to produce fixation. The model then reproduces the run without any resets; the neutral "sweep" mutation proceeds to fixation almost magically, guided by a sequence of random numbers that happen to favor it. Such is determinism.

EXERCISE: Start with the conditional model that we developed above, and change the `m2` DFE's selection coefficient to `0.01`. Now, change the model to involve the sweep of a *pre-existing* mutation, rather than an introduced mutation. To achieve this, you will need to modify the `1000 late()` event. Instead of choosing a target genome and then adding a new mutation to it, instead:

1. get a vector of all segregating mutations, using `sim.mutations`

2. choose a target mutation from that candidate list with `sample()`

3. change the target mutation's type to `m2` with `setMutationType()` (use the online help)

4. change the target mutation's selection coefficient to `0.01` with `setSelectionCoeff()`

The model should work once these changes are made; try it out. This technique of changing the mutation type of a mutation of interest, to allow it to be tracked easily, is very useful. Note that instead of hard-coding the selection coefficient `0.01`, you could use the `drawSelectionCoeff()` method of `MutationType` instead, to draw a new selection coefficient from `m2`'s DFE.

EXERCISE: If the chosen mutation happens to occur in more than one genome, then you have just written a model of a soft selective sweep from standing variation. How might you *guarantee* that it is a soft sweep? I.e., how could you narrow down the field of candidate mutations, in the `1000 late()` event, to only those mutations with a frequency that is greater than `0.1` but less than `0.3`? You will want to use `mutationFrequencies()` to get the frequencies of all mutations – you can pass a vector of mutations to it, rather than just a single mutation as we did above. You should have all the knowledge you need to do this, so go for it – but ask if you get stuck.

6. BEEP! With extra time, section 9.8 or other chapter 9 sections might be of interest.