SLiM Workshop Series #20: Tree-sequence Recording

1. Start with recipe 17.1 in SLiMgui

Start with new Wright–Fisher model in SLiMgui with command-N. Modify the model to contain this script, adding the `initializeTreeSeq()` call and modifying the output event:

```
initialize() {
   initializeTreeSeq();
   initializeMutationRate(0);
   initializeMutationType("m1", 0.5, "f", 0.0);
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 1e8-1);
   initializeRecombinationRate(1e-8);
}
1 {
   sim.addSubpop("p1", 500);
}
5000 late() {
   sim.treeSeqOutput("final.trees");
}
```

Note that the mutation rate is zero; this model does not simulate any mutations. Instead, it just simulates the pattern of mating and recombination that results in the pattern of ancestry along the chromosome present at the end of the run. That pattern of ancestry – the tree sequence – is then output to a `.trees` file at the end of the simulation.

Recycle and run this model. After a couple of seconds, it will finish and a file named `final.trees` should appear on the Desktop (the default output directory for models run under SLiMgui). If the `.trees` suffix of the filename isn't visible, go to the **Finder**, open its **Preferences…**, choose the **Advanced** prefs pane, and check "Show all filename extensions"; this will prevent later confusion.

Next, switch to a plain-text editing app. On macOS, you might have a preferred plain-text editor such as Atom or Sublime Text; if not, you can use TextEdit.app, which might be in the Dock but is certainly in `/Applications`. On Linux, you might prefer Emacs or some such; if not, `gedit` is a reasonable general-purpose plain-text editor that might be installed on your system (if not, a command like `sudo apt-get install gedit` should install it for you).

For those using TextEdit on macOS, some additional instructions are needed; others can skip this paragraph. First, launch TextEdit and then press command-N to make a new empty window if needed. TextEdit can edit in "plain text" and "rich text" modes; make sure you are in plain text mode by going to the **Format** menu. If there is a menu item named **Make Plain Text**, choose it. (If the menu item is **Make Rich Text**, you are in plain text mode already.) Next, we need to make sure TextEdit won't interfere with our code editing. Select **Preferences…** from the TextEdit menu, and find the checkbox titled "Smart quotes and dashes in rich text documents only", and make sure that it is checked. If you don't see that checkbox (at the bottom of the window), look for a checkbox titled "Smart quotes" and make sure it is *not* checked. Also uncheck "Check spelling" and "Check grammar".

Now, in your editor of choice (or in your preferred Python IDE, if you're a regular Python user), enter the following:

```
import msprime, pyslim

ts = pyslim.load("final.trees").simplify()
mutated = msprime.mutate(ts, rate=1e-7, random_seed=1, keep=True)
mutated.dump("final_overlaid.trees")
```

Ensure that the straight quotes have not been made curly by your editor. Save this file to the Desktop folder as `ts_overlay.py`.

This is a Python script that overlays neutral mutations onto the previously generated tree sequence. It begins by importing the `msprime` and `pyslim` packages, then uses `pyslim` to load the `.trees` file. It immediately calls `simplify()` on the loaded tree sequence; this will be discussed in a moment. Next it uses `msprime`'s `mutate()` function to add mutations, given a mutation rate and a random number generator seed (the `keep` option is irrelevant here; it governs whether *existing* mutations on the tree sequence should be kept, but there aren't any anyway). This call overlays neutral mutations along every branch of every tree in the tree sequence, producing a result very similar to what would have been generated by the same neutral mutation rate in SLiM. It can be done much more quickly by `msprime`, however, because now it only needs to be done along the branches of the evolutionary tree that lead to extant individuals in the final generation – typically a very small minority of all of the branches simulated by SLiM. Finally, the new tree sequence with overlaid mutations, named `mutated`, is written out to a new `.trees` file.

Let's return to that call to `simplify()`. SLiM produces `.trees` files that contain the first ancestral individuals in each new subpopulation created by `addSubpop()`, as the roots of the ancestry trees. These individuals can be useful for some purposes; if they are desired, you should not call `simplify()`. These first-generation ancestors are not part of the set of nodes (genomes) considered by `msprime` to comprise the "sample", however, and so if you call `simplify()` they will be removed as part of the simplification operation (unless they continue to root an ancestry tree that has not yet coalesced below the first generation). Here we call `simplify()`, which means that mutations will be overlaid only back to the point of coalescence – the root of the simplified tree sequence. If we wanted mutations overlaid all the way back to the first simulated generation, then we should *not* call `simplify()`. Mutations prior to the point of coalescence would be fixed across the whole population, of course, by definition; so the choice is, in some sense, whether we want fixed mutations included in the overlay, or only segregating mutations. There can be other reasons to choose to keep the first-generation individuals or `simplify()` them away, depending upon one's goals; it is an issue that needs to be considered carefully. Later we will see an example of a scenario in which we do not want to call `simplify()`.

We want to run this script, so the next step is to switch to **Terminal.app**. Again, this might be in the Dock; if not, it should be in `/Applications/Utilities`. Its icon should look like this:

Terminal provides you with a command-line prompt where you can run Unix commands. Once it is launched, type `cd ~/Desktop` and press return; this changes the current directory in Terminal to the Desktop. Then type: `python3 ts_overlay.py` and press return. This executes the Python script in Terminal. If all goes well, after a second or so you should get a new Terminal prompt, and a new file named `final_overlaid.trees` should appear on the Desktop.

If you made it through that sequence, then you have run a tree-sequence recording model in SLiM with neutral mutations disabled, and then you have loaded the resulting `.trees` file into Python and overlaid neutral mutations onto the tree sequence after forward simulation was complete. This may be a bit unsatisfying, since we haven't seen any real evidence that the mutations now exist; they can be accessed in Python, as we will see later. For now, you can select `final.trees` and `final_overlaid.trees` in the Finder and choose **Get Info** from the **File** menu (command-I) to compare them; for one test run, `final.trees` was 2.1 MB while `final_overlaid.trees` was 7.6 MB. The extra file size is due to the mutations added to the tree sequence.

EXERCISE: Modify the SLiM model to contain occasional beneficial mutations – add a beneficial mutation type `m2`, with a selection coefficient of `0.01`, and use a very low mutation rate, `1e-12`, since the vast majority of mutations should be neutral mutations (which we are not simulating). Genomic element type `g1` should draw only `m2` mutations; mutation type `m1` should be unused in SLiM. Run this model to produce a new `final.trees` file, and then run the Python script to produce a new `final_overlaid.trees` file. (Now the `keep=True` flag does matter, since there are `m2` mutations already present in the tree sequence.) The lesson here is that tree-sequence recording and neutral mutation overlay works perfectly well with non-neutral dynamics in SLiM; but the *non-neutral* mutations must be simulated in SLiM, rather than overlaid with `msprime`, since they influence the evolutionary dynamics of the simulation.

2. Tracking true local ancestry with the tree sequence

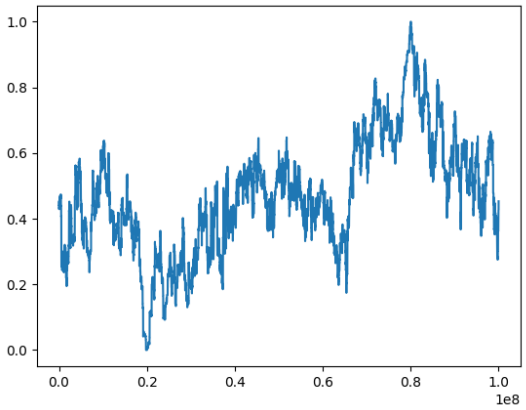
Open recipe 17.5 part I in SLiMgui, from the **Open Recipe** submenu of the **File** menu. This model is quite simple, so we won't reproduce it here. It just sets up two subpopulations, adds a beneficial mutation into every genome of `p1` at one position, adds a beneficial mutation into every genome of `p2` at a different position, merges `p1` and `p2` into a single subpopulation, and runs forward until both of the introduced mutations have fixed. At that point it writes out a `.trees` file and terminates.

What we want to know is: what is the pattern of ancestry along the chromosome? At the position where the beneficial mutation was introduced into `p1`, the final ancestry ought to be 100% `p1`; the population fixed for that mutation, so the ancestry must be completely derived from `p1`. Similarly, at the position of the introduced mutation in `p2`, the population should now have 100% `p2` ancestry. But what does the pattern of ancestry look like everywhere else along the chromosome?

Change the filename saved by `treeSeqOutput()` to `ancestry.trees`, then recycle and run. A new file named `ancestry.trees` should appear on the Desktop.

In SLiMgui, select recipe 17.5 part II (with the `.py` extension and the python emoji next to it). This should open a URL in your browser, displaying the Python code for this recipe. Do a **Save As…** to the Desktop as `ancestry.py`, and close the web browser window. Then go to TextEdit (or your editor of choice), select **Open…**, find the file, and open it. (Alternatively, you could copy/paste the code from the browser window into a new editor window and save it as `ancestry.py`.)

With the file open in your editor, delete the line that begins "`subprocess.check_output`"; that runs the SLiM model, which we have already done. Change the filename that the script reads in to be `ancestry.trees`, to match what we did above. Then save the file, and then switch to Terminal. Execute "`cd ~/Desktop`" to go to the Desktop, and then execute `python3 ancestry.py`. It will take a few seconds to run – perhaps as much as a minute – and then will produce a plot like this:



The *y*-axis is the mean ancestry, across all individuals; the *x*-axis is the position along the chromosome. As expected, the positions of the introduced mutations (20% and 80% of the way along the chromosome) are 100% `p1` and 100% `p2`; they have to be, since the introduced mutations fixed across the whole population. The rest of the chromosome is an intergrade between those states, essentially, with quite a bit of stochasticity.

We can't really get into Python programming here; that is beyond the scope of this workshop. But if you look at the Python script, it should be reasonably clear what it's doing. It loops over the trees in the tree sequence; each tree applies across some range of positions along the chromosome, within which the pattern of ancestry is uniform because it has not been broken by a recombination event. For each tree, it calculates the mean ancestry across all of the leaves (the extant individuals), finding the subpopulation represented by each root of the tree using `tree.population()` and weighting that by the number of leaves under that root. Finally, the plot is generated by the `plot()` function of the `matplotlib` package. When you close the plot window, the Python run in Terminal will complete.

The upshot is: the `.trees` file saves the ancestry for every individual at every point along the chromosome, and accessing that information to produce a plot of mean true local ancestry after admixture is quite simple. Doing the same thing in SLiM without tree-sequence recording would be *much* more difficult and time-consuming – the comparable SLiM model without the use of tree-sequence recording would take more than seven days – yes, *days* – to run.

EXERCISE: Let's see what this looks like if the admixture proportions aren't even. Change the size of `p1` to `4500`, but keep the size of `p2` at `500`. Change the size of `p3` to `5000`, and use migration rates of `c(0.9, 0.1)` so that `p3` is a proportional mixture of `p1` and `p2`. To compensate for the larger overall population size, so that the analysis time isn't too long, change the defined value of `L` to `1e7`. Recycle and run; it should still take less than a minute for the model to run. Once it's done, go to Terminal and execute `python3 ancestry.py` again. This will take a bit longer – perhaps 2–3 minutes. The resulting plot should be fairly smooth, though, because of the larger population size, and the pattern it shows should be quite suggestive.

3. Recapitation

Open recipe 17.10 part I in SLiMgui, from the **Open Recipe** submenu of the **File** menu. This recipe just creates a fairly large population (100,000 individuals), introduces a strongly beneficial mutation into a single genome, and runs forward until it fixes or is lost. If it fixes, the tree sequence is written out before termination.

We need a few changes to this script before we run it. First of all, change the filename saved by `treeSeqOutput()` to be simply `decap.trees`. Second, change the size of `p1` from `1e5` to `5e4` to get a somewhat quicker runtime. Then, at the very top of the `initialize()` callback, add this call:

```
setSeed(2);
```

This tells SLiM to use a random number seed that is known to produce fixation rather than loss (in SLiM 3.3, anyway), so that we don't have to do repeated runs to get the result we want.

Recycle and run the model. You should see that the model runs to generation `163`, and then it sits still for a long time – not finished, but not advancing either. This is because the introduced beneficial mutation has fixed and SLiM is working on writing out the `.trees` file. To do so it first performs a simplification, and that takes a little while with such a large population. It should be done in a couple of minutes, and it should peak at a memory usage of ~3 GB (simplification is memory-hungry). While you're waiting for it to finish, continue reading.

The only interesting thing about this script, really, is the `simplificationRatio=INF` parameter passed to `initializeTreeSeq()`. This tells SLiM not to simplify the tree sequence at all while the model is running; it will not be simplified until the save at the end. This makes the model run somewhat more quickly, since simplification is slow, but at the price of higher peak memory usage.

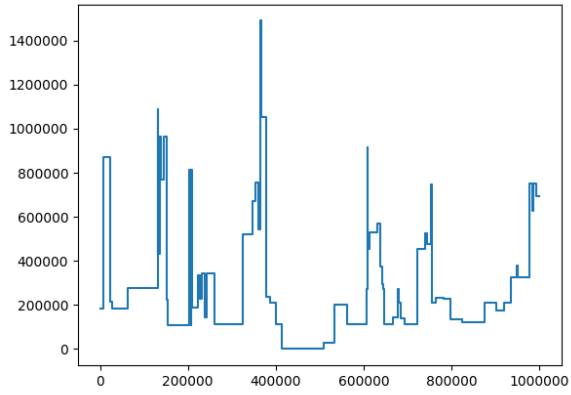
This should be done with care; if the memory usage gets too high the process will probably either crash or bring down the whole machine, so you should generally only do this when you have tested to ensure that the model will remain within safe bounds (as I have tested this model). If a tree-seq model needs to run faster, there are additional tools available to control how often simplification is performed, and this can make a big difference to performance; see the SLiM manual.

The model may still be running, for you, but shouldn't take too much longer. While you're waiting, you might go to the SLiM manual and read section 1.7 (the introductory overview of tree-sequence recording). Once the `decap.trees` file has been saved, the model terminates. Now follow the same procedure as before to get the second part of this example, recipe 17.10 part II, open as a file named `recap.py` in your editor (starting by opening it in your web browser from SLiMgui).

Change the filename that it loads at the top to `decap.trees`, and change the filename it dumps near the end to `recap.trees`. (All this futzing with filenames is somewhat gratuitous, yes; we're just trying to make it more clear what all the clutter we're making on the Desktop is.) Save these changes, and then go to Terminal, do a `cd ~/Desktop` if you're not already there, and then execute `python3 recap.py`.

If all is well, after a couple of seconds it should produce a plot that looks like a squared-off U – high and flat on the outsides, low and flat in the center. This depicts the heights of the ancestry trees along the chromosome prior to recapitation, as the SLiM run left them. The central dip is the area carried along by the selective sweep; it dates back to generation `100`, when the beneficial mutation was introduced. The rest of the chromosome dates back to generation `1`, when subpopulation `p1` was created.

What we are trying to do here is to *recapitate* the simulation run: to provide it with a neutral burn-in history, after the fact, that coalesces the ancestors of the simulation backward in time in a natural manner, using a standard coalescent model. If you close the first graph window, the Python script (which was paused for the plot) will resume, and after a few seconds a second plot will appear that looks something like this:



The lowest "shelf", near the center of the plot, is the same as it was before: the region around the sweep that was already coalesced because of the sweep. The rest of the plot is quite different, though; it shows the new coalescent history for the simulation, which stretches back about 1.5 million generations into the past. That would have taken quite a while to run as a neutral burn-in with SLiM, even without neutral mutations, given the large population size. With recapitation it is even faster than running a coalescent simulation to provide an initial burn-in state for SLiM up front, because only the parts of the ancestral genomes that are still part of the tree sequence at the end of the forward simulation need to be coalesced; the rest can be ignored.

The Python script for this example is somewhat complex, and the SLiM manual discusses it, so we won't go into detail about it here. Most of the complexity is in a function it defines named `tree_heights()`, which is a little bit tricky because it has to find the right node to measure tree height at. The recapitation operation itself is very easy – a single line of code. The other thing to note about this Python script is that it does *not* call `simplify()` on the loaded tree sequence. This is because we specifically want to recapitate backward from the first-generation ancestors of the forward simulation – we need those ancestors, so we certainly don't want to `simplify()` them away.

Recapitation can allow extremely large simulations to be run, as long as the non-neutral portion of the simulation that is still run in SLiM is manageable in size and duration. For large models, the burn-in can often take much longer than the actual simulation of interest – but recapitation allows the overhead of neutral burn-in to be reduced almost to zero. There are limitations, of course; in particular, if the burn-in period itself needs to be non-neutral then recapitation might not be usable.

4. Examining mutations with `pyslim`

We're going to finish with a very quick look at the facilities provided by `pyslim`, which we have only used, so far, to load SLiM tree sequences.

In SLiMgui, open recipe 17.7 part I from the **Open Recipe** submenu of the **File** menu. This simulates a population undergoing beneficial and deleterious mutations in equal proportions, both drawn from gamma distributions of the same shape but opposite sign – the beneficial mutation DFE has a mean of `0.1`, while the deleterious mutation DFE has a mean of `-0.1`. The expectation is that the beneficial mutations that actually fix will tend to have a mean selection coefficient larger than `0.1`, because the more beneficial they are the more likely they are to fix. Similarly, we expect that the deleterious mutations that actually fix will tend to have a mean selection coefficient smaller than `-0.1` (i.e., closer to zero). Finally, we can expect that fewer deleterious mutations will fix.

Change the filename that is saved by `treeSeqOutput()` to `muts.trees`. Then recycle and run the model, which should take a minute or so. Once the `muts.trees` file has been created, open recipe 17.7 part II, and get it into an editor file named `muts.py` following the same procedure as before. Change it to load `muts.trees`, save that change, and then switch to Terminal. Do `cd ~/Desktop` if needed, then execute `python3 muts.py`. For a test run, I got:

```
Beneficial: 3594, mean 1.0713634433442913
Deleterious: 101, mean -0.00802324793824068
```

It looks like there is at least a weak upward bias for beneficial mutations, and a strong bias for deleterious mutations; and certainly fewer deleterious mutations fixed, as predicted.

How does the analysis in Python work? The `mutations()` method provides all of the mutations in the tree sequence, and the code loops through them one by one. Some information about mutations is kept by `msprime`, but the information we want – their SLiM selection coefficients – is stored in a special SLiM-specific "metadata" buffer. We therefore use `pyslim` to decode the mutation metadata for us, with `decode_mutation()`. The result is a list, not a single object, because of the possibility of stacked mutations at a position; stacked mutations are going to be very uncommon in this model, though, so let's not worry about that detail (the SLiM manual discusses it further). We get the selection coefficients from that list, and add the ones that aren't neutral to a vector named `coeffs`. We can access all sorts of other SLiM information through `pyslim` too; see the `pyslim` manual.

EXERCISE: Try changing the model in some way that you're curious about – perhaps decreasing the means of the DFEs to `0.01` and `-0.01`, or increasing the population size to `2000`, or reducing the recombination rate to `1e-9`. Then recycle, run, and then re-run the Python analysis.

5. BEEP! With extra time, look at SLiM manual section 17.4, an interesting tree-seq model that we will not look at today since it takes several hours to run.