SLiM Workshop Series #23: SLiM at the Command Line

1. Start a new model in SLiMgui

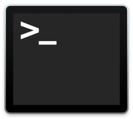
Run SLiMgui, and close any existing model windows. Press command-N to make a new model window. Adapt the model to the following code, or just delete it and enter this model:

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

Save the model by pressing command-S or choosing **Save** from the **File** menu. Save it to the Desktop, with a filename of `rep.slim`. Our goal in this exercise is to see how we can adapt this model to be run at the command line, as a first step toward running it on a computing cluster.

2. Run the model at the command line

We want to run this model at the command line, so switch to **Terminal.app**. 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: `slim rep.slim` and press return. This runs the `slim` command-line tool, which can execute SLiM models on any Unix platform (including macOS and Linux). The model should run in a second or so, producing output directly into Terminal. So far, so good!

3. Adapt the model to use defined parameters

We want to be able to control the parameters of the model with command-line arguments, so we need to change the model to use defined parameters. Modify the script to look like this:

```
initialize() {
   defineConstant("MU", 1e-7);
   defineConstant("R", 1e-8);
   defineConstant("N", 500);

   initializeMutationRate(MU);
   initializeMutationType("m1", 0.5, "f", 0.0);
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(R);
}
```

```
1 {
   sim.addSubpop("p1", N);
}
5000 late() { sim.outputFixedMutations(); }
```

This is the same except that constants `MU`, `R`, and `N` are now defined at the top, and are then used in the script for the mutation rate, recombination rate, and population size.

Switch back to Terminal. Press command-K to clear the old output from the window, to provide a clean slate for working. Now, you will want to run the command `slim rep.slim` again, but you shouldn't need to retype it; if your machine is correctly configured, you should be able to press the up-arrow key to get the previous command in the command history, and then simply press return to execute it again.

4. Pass a random number seed to `slim`

Re-run the model several times by pressing up-arrow and then return. Note that each run produces different output; this is because the random number seed used by `slim` is different each time (based upon the system clock at the start of the run). When doing replicated runs on a computing cluster, we typically want to use known seed values so that we can reproduce any particular run of the model (in case that run does something particularly interesting, or exhibits a bug – or just for scientific reproducibility of our results). We can easily specify a seed value for `slim` to use. In Terminal, type:

```
slim -s 1 rep.slim
```

and hit return. Once that run has completed, hit up-arrow and return a couple of times to run it repeatedly; it should produce the same output every time. Notice that at the top of each run's output, it says:

```
// Initial random seed:
1
```

That tells us the initial random number seed used for that run – a value of 1, as requested by the `-s` command-line option.

5. Adapt the model to accept command-line parameters

We've got the model using the parameters `MU`, `R`, and `N`, but their values are hard-coded in the script. We'd like to be able to change those values at the command line, so that we can run the same model with a variety of parameter values easily. However, we still want to be able to run the model in SLiMgui, too, using default values for the parameters. To achieve this, go back to SLiMgui and change the `initialize()` callback to start this way:

```
initialize() {
   if (exists("slimgui")) {
      defineConstant("MU", 1e-7);
      defineConstant("R", 1e-8);
      defineConstant("N", 500);
   }
   ...
```

The `...` represents the rest of the callback, of course; don't type "…". Now recycle and run in SLiMgui; the model should run as before, using the defined parameter values. How does this work? There is a global variable named `slimgui` that is defined only when a model is running inside SLiMgui; it provides the ability to control SLiMgui from the model script. The `exists()` function checks for the existence of any named symbol; if a symbol named `"slimgui"` exists, we're running under SLiMgui and therefore ought to define the model's parameters.

Save the model in SLiMgui with the change you made above. Now go back to Terminal, and press up-arrow and return the run the model again. You will get an error like this:

```
ERROR (EidosSymbolTable::_GetValue): undefined identifier MU.

Error on script line 8, character 24:

   initializeMutationRate(MU);
                          ^^
```

This is `slim` telling us there's a problem. It tells us the line and character in the script where the problem occurred (line 8, character 24), and shows us the offending line with carets, `^^`, under the spot where the error was encountered. It also tells us what the error was ("`undefined identifier MU`"), and – mostly for use by developers of SLiM itself – the internal function where the error occurred (`EidosSymbolTable::_GetValue`).

So, great – this makes sense, because we've just changed the model to expect parameter values to be defined at the command line. In Terminal, type this command line and hit return:

```
slim -s 1 -d MU=1e-7 -d R=1e-8 -d N=500 rep.slim
```

The model should run nicely. We're still passing a seed of `1`, but now we are also passing definitions of the constants `MU`, `R`, and `N` using the `-d` command-line flag. Each use of `-d` is followed by a symbol name, an equals sign, and a value. Try running the model with different parameters, like:

```
slim -s 1 -d MU=1e-6 -d R=1e-8 -d N=500 rep.slim
```

This run will probably produce quite a few fixed mutations at the end, given the elevated mutation rate.

6. Scale the runtime by the population size

We'd like the model to run for a number of generations that is proportional to the population size, `N`. This is often desirable because the expected time for neutral mutations to fix is proportional to `N`; if you increase `N` without scaling, the model will just be getting started when it ends. This is easy to see with our model in Terminal; try:

```
slim -s 1 -d MU=1e-6 -d R=1e-8 -d N=5000 rep.slim
```

Before, when we used a mutation rate of `1e-6`, we got tons of fixed mutations at the end of the run. Now, with the same mutation rate but a population size ten times larger, you probably got none at all; there just wasn't enough time for neutral mutations to drift to fixation.

Go back to SLiMgui and add this line, immediately after the other defined constants:

```
defineConstant("G", N*10);
```

Now G is the generation we want the model to execute to. We'd like to change the final output event to look like this, in order to run in generation `G`:

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

Unfortunately, this is not legal SLiM syntax; the generation(s) in which a script block is scheduled to run must be `integer` literals (as the programming-language jargon says) – they cannot be symbolic. To work around this limitation, we will ask SLiM to dynamically reschedule the script block to the desired generation. First of all, change the event's definition to:

```
s1 5000 late() { sim.outputFixedMutations(); }
```

We've just added "s1" at the beginning. This labels the script block with the symbolic name `s1`, so that we can refer to it elsewhere (just as subpopulations are named `p1`, `p2`, etc., and mutation types are named `m1`, `m2`, etc.).

Next, add the following line to the generation `1` event, just before the call to `addSubpop()`:

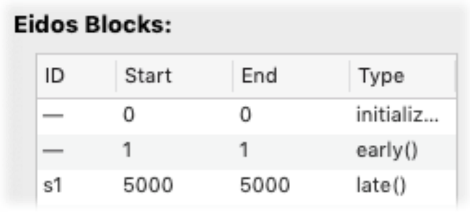
```
sim.rescheduleScriptBlock(s1, start=G, end=G);
```

This command, run in generation `1`, will reschedule the script block named `s1` to run in generation `G`. The original scheduled generation of `5000` for `s1` won't matter at all; the only important thing is that it be large enough that `s1` doesn't get executed accidentally before it gets rescheduled (so it would be wise not to set it to `1` instead of `5000`, although since it's a `late()` event it would work in this case). Since `5000` is fine, let's leave it at that. Save and run the model in SLiMgui, and it should still run through generation `5000`. If you change the definition of `N` to `50`, however – do that now, and recycle and run – it should then run only through generation `500`.

It can be instructive to observe this rescheduling in SLiMgui. With the default value of `N` still defined to be `50`, recycle the model, and then open the tables drawer by clicking this button in SLiMgui:

☞

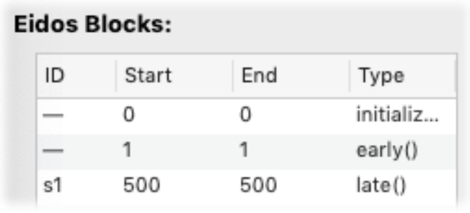
A drawer with several tables in it should open (if there's room – the SLiMgui window needs to have a bit of space on the right-hand side for the drawer to open into):



Eidos Blocks:

ID	Start	End	Type
—	0	0	initializ...
—	1	1	early()
s1	5000	5000	late()

You can see the script block named `s1`, scheduled to run in generation 5000. Now step twice (to get past generation 1, where the reschedule occurs), and you should see:



Eidos Blocks:

ID	Start	End	Type
—	0	0	initializ...
—	1	1	early()
s1	500	500	late()

The rescheduling has occurred as intended.

Complex models with many different events that all need to run at parameter-driven times can name each such event with a different symbol (`s1`, `s2`, `s3`, ...), and then reschedule each to the generation desired. This is a little bit clunky, but it works.

But what about running this model at the command line? Try the previous command again:

```
slim -s 1 -d MU=1e-6 -d R=1e-8 -d N=5000 rep.slim
```

You will get an error: `undefined identifier G`. We've defined `G` inside the `if (exists())` block, so it is only defined when running inside SLiMgui. If we *always* wanted `G` to be `N*10`, we could simply move the definition outside of that block so that G is thusly defined in all cases. However, suppose we sometimes want G to be N*10, and sometimes N*20, so that we can check whether N*10 is long enough to produce equilibrium results. We can actually use arbitrary Eidos expressions in -d defines, so we can do this:

```
slim -s 1 -d MU=1e-6 -d R=1e-8 -d N=2500 -d G="N*10" rep.slim
```

That will take a little while to run – large-ish population size and high mutation rate – but in less than a minute you should get a big dump of fixed mutations.

Notice that we put the expression N*10 in quotes. This is not defining G to be an Eidos string; the quotes actually get stripped off by the Terminal shell before Eidos ever even sees them, so as far as Eidos is concerned G has been defined to be N*10. The reason the quotes are there is that the * character has a meaning in Terminal, as a filename wildcard; we need to quote the definition to avoid confusing Terminal. If you want to be safe, you can simply always quote the definitions of all -d constants so that the Terminal shell never interferes with them.

You can define other constants with expressions too; many population-genetics models want to use RHO and THETA as parameters, and so such models might calculate SLiM's per-base-position mutation and recombination rates based upon RHO, THETA, and N. Definitions are processed in the order in which they are supplied on the command line, so later definitions can depend upon earlier definitions (as G's definition depends upon N here).

7. Running a batch of simulations with a shell script

Now we'd like to run this model multiple times, with different parameter values. We're going to do this with a shell script that runs in the Unix shell Bash. Getting things set up will take a few steps. First of all, go to Terminal, execute cd ~/Desktop to go to the Desktop directory if you're not already there, and type the command:

```
touch run_reps.sh
```

This creates a new file named run_reps.sh, which will become our sublaunch shell script. Now, still in Terminal, on macOS type:

```
open -e run_reps.sh
```

This asks Terminal to open the file run_reps.sh in TextEdit, the default text editor in macOS (of course feel free to use a different plain-text editor instead, such as Atom or Sublime Text). Linux users might try gedit instead of open -e; it's a decent plain-text editor, but you may need to install it with a command like sudo apt-get install gedit. Your machine should switch to TextEdit (or gedit, or whatever), launching it if it isn't already running, and should then open a new empty window titled run_reps.sh. In that window, enter the following text:

```
echo "foo"
```

(Make sure the straight quotes haven't been replaced with curly quotes; if they have, find and turn off the "smart quotes" pref in your editor.) Save the file. This rudimentary shell script simply prints "foo" to the output, using the Bash command echo.

Let's go back to Terminal now and run the script, by executing:

```
/bin/bash run_reps.sh
```

If all has gone according to plan, you should get the expected output:

```
foo
```

Now we're ready to expand this script into a proper sublaunch script. Go back to the script file in your editor and replace the script with the following:

```
slim -s 1 -d MU=1e-7 -d R=1e-8 -d N=100 -d G="N*10" rep.slim
```

Now, executing `/bin/bash run_reps.sh` should result in that SLiM run being sublaunched; try that now to make sure it works.

We'd like to do multiple runs with different values for `N`. To achieve that, add a `for` loop:

```
for N in 100 200 500 1000 ; do
   echo "Running with N ==" ${N}:
   slim -s 1 -d MU=1e-7 -d R=1e-8 -d N=${N} -d G="N*10" rep.slim
   echo
done
```

This is a bit different from Eidos, but it should be clear enough what's going on. For help with the syntax of Bash shell scripts, there's lots of information online including free tutorials and such, so we won't go into any detail here. Suffice to say that we're looping over multiple values for `N`, printing a line above each run's output, and then sublaunching the run with the current loop iteration value of `N` using the Bash syntax `${N}` to refer to the value of variable `N`.

Execute this, with `/bin/bash run_reps.sh` as usual, and you should see the four model runs occur in succession.

If you would prefer to sublaunch your simulations using R or Python, the SLiM-Extras repository on GitHub (at https://github.com/MesserLab/SLiM-Extras) has examples of sublaunch scripts in those languages; you should be able to adapt them without much trouble.

8. Redirecting output to separate files

It's not really very convenient having the output from all the runs dumping right into Terminal; we'd like it to be placed into separate files instead. To achieve this, we just need to add a redirection operator to the `slim` command, like so:

```
slim -s 1 -d MU=1e-7 -d R=1e-8 -d N=${N} -d G="N*10" rep.slim > out_${N}.txt
```

The `>` symbol tells Bash to redirect the output from slim to the following file, where the filename `out_${N}.txt` produces files named `out_100.txt`, `out_200.txt`, `out_500.txt`, and `out_1000.txt`. Try running this script now, again with `/bin/bash run_reps.sh`, and verify that the output files have been created as expected.

EXERCISE: One often wants to run multiple replicates for each parameter value, so that you don't get just a single result for a given parameter value (which might be idiosyncratic), but instead a distribution of results. First, create a new folder on your Desktop, named `RepWork`, drag the `rep.slim` script and `run_reps.sh` script inside it, and `cd ~/Desktop/RepWork` in Terminal so you don't clog up the Desktop with too many files. Then, modify the shell script to do three replicates for each value of `N`, by adding a nested `for` loop over values `1 2 3` with loop index variable SEED. Pass the value of `SEED` to `slim` as the random number seed with `-s`, and make the redirected output go to files with filenames like `out_Nxxxxx_SEEDxxxxx.txt`, where `xxxxx` should be value of the relevant variable. Run the script to confirm that it does all the replicate runs expected.

At this point you've got a sublaunch script that you could use on our local machine, perhaps running tasks simultaneously in the background by putting " &" at the end of the sublaunch line (another Bash feature – try it!). To run on a computing cluster, you would typically not run `slim` directly, but would instead run some kind of job-submission command like `qsub` to schedule jobs on cores.

9. BEEP!

With extra time, do some Googling about Bash shell script syntax to find out more.