SLiM Workshop Series #14: Live Plotting in R

1. Start with recipe 13.2 in SLiMgui

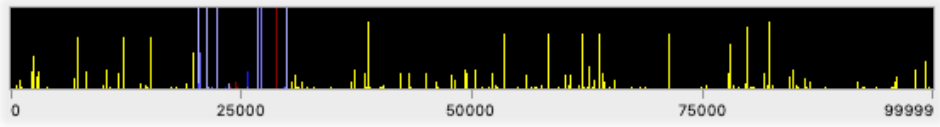
Close all open windows in SLiMgui, and then start with recipe 13.2 by choosing it from the **Open Recipe** submenu of the **File** menu. This is quite similar to the model we built in the previous exercise:

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);       // neutral
   initializeMutationType("m2", 0.5, "n", 0.0, 0.5);  // QTLs
   m2.convertToSubstitution = F;
   initializeGenomicElementType("g1", m1, 1);
   initializeGenomicElementType("g2", m2, 1);
   initializeGenomicElement(g1, 0, 20000);
   initializeGenomicElement(g2, 20001, 30000);
   initializeGenomicElement(g1, 30001, 99999);
   initializeRecombinationRate(1e-8);
}
fitness(m2) { return 1.0; }
1 { sim.addSubpop("p1", 500); }
1: late() {
   inds = sim.subpopulations.individuals;
   phenotypes = inds.sumOfMutationsOfType(m2);
   inds.fitnessScaling = 1.5 - (phenotypes - 10.0)^2 * 0.005;

   if (sim.generation % 100 == 0)
      catn(sim.generation + ": Mean phenotype == " + mean(phenotypes));
}
5000 late() {
   m2muts = sim.mutationsOfType(m2);
   freqs = sim.mutationFrequencies(NULL, m2muts);
   effects = m2muts.selectionCoeff;
   catn();
   print(cbind(freqs, effects));
}
```

The chromosome has a neutral region flanking a central region that undergoes QTL mutations. The QTL mutations are drawn from a normal distribution of standard deviation `0.5`, centered on `0.0`. These mutations are made neutral for SLiM with a `fitness(m2)` callback that returns `1.0`; instead, their additive effects are computed with `sumOfMutationsOfType()` and used as phenotypes in a quadratic fitness function that defines stabilizing selection toward an optimum phenotypic trait value of `10.0`. The fitness effect of phenotype for each individual is placed in the `fitnessScaling` property of the individuals. Every `100` generations the mean phenotype is printed to the output.

If you recycle and run, you might see a final state something like this:



The QTLs that produced adaptation to the fitness peak are clearly visible.

All of this should be review, so let's move on to something new.

2. Keeping a history of phenotypic means

Expand the generation `1` event that sets up the initial generation to this code:

```
1 {
   sim.addSubpop("p1", 500);
   sim.setValue("history", NULL);
}
```

The `setValue()` call associates the value `NULL` with the named key `"history"` in the object `sim`. Most SLiM objects maintain a key-value dictionary that can hold arbitrary values for you, because they are subclasses of the Eidos class `Dictionary`. Such dictionaries are called by various names in other programming languages, such as "associative arrays", "unordered maps", or "symbol tables"; these concepts are all basically the same. Values can be set (and overwritten with new values) using `setValue()`, and read with `getValue()` as we will see momentarily. This, like the `tag` property on many SLiM objects, is a convenient place to stash values that you want to keep long-term. Since we want our phenotype history to be simulation-wide, we'll keep it on the `sim` object, but you can define keys on subpopulations, individuals, mutations, and various other SLiM objects as well.

So far we have just defined an empty initial history with the constant `NULL`. Now we'd like to add new entries to the history in each generation. To do so, replace the current output code:

```
if (sim.generation % 100 == 0)
   catn(sim.generation + ": Mean phenotype == " + mean(phenotypes));
```

with some history logging code:

```
m = mean(phenotypes);
h = c(sim.getValue("history"), m);
sim.setValue("history", h);
```

This just gets the old value of the history key, appends the current phenotypic mean to it with `c()`, and sets the new vector back into the history key. To make sure that this is working, replace the existing `5000 late()` event with this:

```
5000 late() {
   print(sim.getValue("history"));
}
```

If you recycle and run, at the end of the run a complete history should be dumped to the output. That's a bit hard to interpret visually, though – we'd like to see a plot. That's what we'll do next.

3. Open a new plot window in SLiMgui

SLiMgui is capable of displaying PNG image files as well as SLiM models, specifically to support this live-plotting feature. First of all, add a line to the end of the generation `1` event to create a new temporary PNG file:

```
defineConstant("pngPath", writeTempFile("plot_", ".png", ""));
```

The `writeTempFile()` call creates a new temporary file inside the `/tmp` directory, a standard place on Unix systems (including macOS) where temporary scratch files can be placed. This temporary file will use the supplied prefix and suffix, with a random string in between; an example temporary file created by this call might be `/tmp/plot_WKNVUI.png`. The file generated is guaranteed not to collide with any existing files, so this is a safe way to generate temporary files even when multiple SLiM processes are running simultaneously on the same system.

The path of the temporary file is returned by `writeTempFile()`, and saved as a defined constant named `pngPath`. The new file is empty, since we passed "" for its contents.

Next, we'd like SLiMgui to automatically open the plot. Add these lines just below the last one:

```
if (exists("slimgui"))
   slimgui.openDocument(pngPath);
```

When running under SLiMgui, a global object named `slimgui` is defined, which can be used to communicate from your Eidos code to SLiMgui. If that object is defined, as tested with the built-in Eidos function `exists()`, then we make a call to the `openDocument()` method on it to request that `pngPath` be opened in a new window. (If we're running at the command line instead of inside SLiMgui, `slimgui` will not exist, and this code will do nothing.)

Recycle, and then step twice – once for the `initialize()` phase and then once to step over generation 1. A new window should open for the temporary PNG file. Resize/reposition your windows so you can see this window while you're working on your model.

4. Update the plot

Now we want to update the PNG with a new plot every `100` generations. We will use a command-line tool called `Rscript` as a part of this; `Rscript` is a Unix command that is installed as part of the standard R installation on most systems, and it just runs an R script that is provided to it as a file path. The first step is to look up the location of `Rscript`. To do this, add this code to the end of your `initialize()` callback:

```
if (fileExists("/usr/bin/Rscript"))
   defineConstant("RSCRIPT", "/usr/bin/Rscript");
else if (fileExists("/usr/local/bin/Rscript"))
   defineConstant("RSCRIPT", "/usr/local/bin/Rscript");
else
   stop("Couldn't find Rscript.");
```

This searches for the location of `Rscript` by checking two likely locations, and defines a global constant, `RSCRIPT`, with the path found. Alternatively, you can simply go to Terminal and execute the command `which Rscript`, and then add a single line at the end of `initialize()`:

```
defineConstant("RSCRIPT", <path-to-Rscript-as-a-quoted-string>);
```

That's what you'll need to do if `Rscript` is installed at some other location than the two we try above. If `which Rscript` cannot find its location, you may not have R installed on your machine, in which case you will need to install it now. On macOS you can download a double-click installer from CRAN; on Linux, depending on your distro, executing `sudo apt-get install r-base-core` might install it for you.

Next, add this code to the end of the `1: late()` event, after the history is updated (you might copy/paste it from the solution instead of typing it out, as it's a rather icky chunk of code):

```
if (sim.generation % 100 == 0)
{
   rstr = paste('{',
      'x <- (1:' + size(h) + ')',
      'y <- c(' + paste(h, sep=", ") + ')',
      'png("' + pngPath + '", width=4, height=4, units="in", res=72)',
      'plot(x=x, y=y, xlim=c(0, 5000), ylim=c(-1, 11), type="l")',
      'dev.off()',
      '}', sep="\n");

   scriptPath = writeTempFile("plot_", ".R", rstr);
   system(RSCRIPT, args=scriptPath);
}
```

Yes, this code is kind of crazy; it is recommended that you copy/paste the code from the solution, rather than attempting to retype the code above.

So what does this code do? Every `100` generations, it generates an R script as an Eidos string, writes it out to a temporary `.R` file using `writeTempFile()`, and then calls `system()` to run the `Rscript` command. The `system()` function runs any Unix-style command line as a forked subprocess, inside a `/bin/sh` shell; here we pass it a command of `Rscript` (the full path to `Rscript`, in fact, to avoid path lookup issues) and the script path returned by `writeTempFile()` as its command-line argument. If that is gibberish, the upshot is: it runs `Rscript` with the R script just saved.

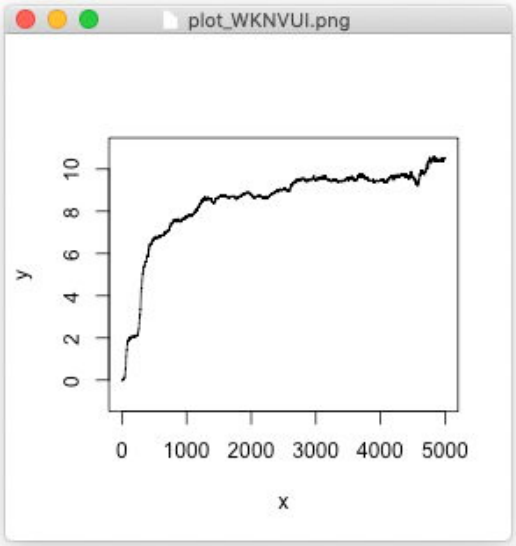
Be careful with the quotes; some are single quotes and some are double, because we're taking advantage of the fact that in Eidos you can represent a string with either single quotes, `'foo'`, or double quotes, `"foo"`; the two are exactly the same string. A single-quoted string can contain double quotes within it, and we take advantage of that fact here to pass double-quoted strings through to R in the script string we're generating. A few dynamic parts of the R script, such as the values to plot, are appended with the string `+` operator. The `paste()` function is used to stick all the R script lines together into a single string separated by newlines (represented by the `"\n"` constant in Eidos).

Triple-check the code you've entered; if there's a problem with it the error will probably occur in R, not in SLiM, and it will be hard to figure it. You can check that you got the quoting correct by making sure that the pattern of black and red syntax coloring matches the code snippet above.

As for the R code generated, it ends up looking something like this:

```
{
x <- (1:200)
y <- c(-0.00204115, -0.0023426, ..., 3.96779)
png("/tmp/plot_WKNVUI.png", width=4, height=4, units="in", res=72)
plot(x=x, y=y, xlim=c(0, 5000), ylim=c(-1, 11), type="l")
dev.off()
}
```

We can't go into a ton of detail here about how plotting in R works, etc., but this is really pretty simple. The `x` and `y` variables are defined as vectors of (x, y) coordinate pairs for the line we want to plot. The `png()` function begins a new PNG plot. Then the `plot()` call makes the actual plot, using type `"l"` to plot a line. Finally, `dev.off()` closes the plot and generates the PNG file at the previously specified path. The result, at the end of a run in SLiMgui, looks something like this:



One could add axis labels, make the layout nicer, etc., with improved R code, but we won't get that deep into the weeds here (the SLiM manual has a similar recipe with nicer plotting code).

This is quite a simple example, but hopefully the potential is clear. You can generate *any* R script on the fly, to produce *any* sort of plot; indeed, there is nothing to stop you from opening and updating more than one plot window simultaneously in your script. Even if you're not interested in live-updating plots in SLiMgui, this facility can be used to generate standardized plot PNG or PDF files at the end of each of your model runs, and it works on computing clusters as well as in SLiMgui as long as R is correctly installed on the target machine. PDF plots are nice because they are resolution-independent, so they're smoother and better for publication; unfortunately, SLiMgui can't display them at present due to limitations of the Qt widget framework it uses. However, you could certainly use an external PDF viewer app instead, if you wished to do so. Examples of this are in the SLiM-Extras repository on GitHub; see the modified `Recipe_14.8` files in the `models` subdirectory.

EXERCISE: If you're conversant with plotting in R, make sure you understand how this code works by adding x and y axis labels to the plot and changing the line to be drawn in red.

5. BEEP! With extra time, you might take a look at section 13.5 of the SLiM manual, which presents a model of adaptation for two pleiotropically linked phenotypic traits, with live plotting of the adaptive walk in two dimensions. It's probably too complex to read in detail now, but you can at least open the recipe in SLiMgui and run it to see what it does.