SLiM Workshop Series #3: The Population Hierarchy

1. Open recipe 5.3.1 in SLiMgui

As before, locate and launch the SLiMgui application if it isn't already running. You may wish to close any open SLiMgui windows. Then open the model for the manual's recipe 5.3.1; all of the manual's recipes are included within SLiMgui, so you can do this by going to SLiMgui's **File** menu, then to the **Open Recipe** submenu, then to the **5 – Demography and population structure** submenu, and finally to recipe 5.3.1.

2. Values, vectors, and variables

Before we even start playing with this model, let's learn a few things about the Eidos language. You can open an interactive Eidos console inside SLiMgui by clicking a button that is in the strip of buttons just above the scripting pane – the one that looks like this:

>

A new window will open, the right-hand half of which contains some output and then a > prompt (which is where the button's icon comes from, of course). The left-hand half of the window is essentially a scratch space where you can type whatever you want. The right-hand half is the interactive Eidos console. Let's try typing something. You should type each of these things at your own Eidos prompt, even if it seems redundant, because doing things yourself, rather than just reading about this, is essential to learning. (And feel free to experiment off-script!) So type:

```
> 6
6
```

We gave Eidos the integer `6`, and it printed it back to us. Let's assign it to a variable, and then fetch the value back again:

```
> x = 6
> x
6
```

In programming languages, variables are essentially named values; when the variable name is used, the associated value is used. In other words, when we use `x` later, the value `6` will be used. Note that this is a bit different from the meaning of a variable in mathematics; the mathematical equation "x=6" would *assert* that x is equal to 6, and if one were then to write "x=7" there would be a contradiction, because both assertions cannot simultaneously be true. But when programming, it is not an assertion but an assignment. We can see the difference by now doing this:

```
> x = 7
> x
7
```

The variable has simply been updated with a new value; it used to be `6`, now it is `7`. Let's define another variable and then do a little math:

```
> y = 3
> x*y
21
```

The `*` is the *multiplication operator* in Eidos (and most programming languages), so the result is `7` times `3`, or `21`. Eidos defines all of the standard mathematical operators you would expect, such as `+`, `-`, `*`, and `/`, and `^` is the *exponentiation operator*; `2^4` is `16`. The `=` symbol we have been using in statements like "`x = 7`" is the *assignment operator*. As a matter of fact, the parentheses `()` that we have been using to contain the arguments passed to functions and method calls are an operator too, the *call operator*. And so forth; there are a number of other operators that we will discuss later.

EXERCISE: Do some playing around with defining variables and using them in expressions, including using the arithmetic operators `+`, `-`, `*`, `/`, and `^`. You can use parentheses to group expressions, just as you would in a mathematical formula.

We now need to discuss the `sequence operator`, which is a colon, `:`. Let's give it a whirl:

```
> 1:10
1 2 3 4 5 6 7 8 9 10
```

For those who have programmed in R, this will be entirely familiar; in fact, Eidos is patterned fairly closely on R in many respects. For everyone else, this requires some explanation. What has been printed out as the value produced by the expression `1:10` is not a single integer, but a *vector* – an ordered series of zero or more elements, in this case the integers from 1 to 10. In fact, all values in Eidos are vectors. That bears repeating: *all values in Eidos are vectors*. The value of `x` that we printed above, `7`, was a vector; it was just a vector that happens to contain exactly one value, termed a *singleton* in Eidos parlance. Let's assign our sequence into a variable:

```
> z = 1:10
> z
1 2 3 4 5 6 7 8 9 10
```

We can get specific elements out of `z` with the subset operator, `[]`, quite easily:

```
> z[0]
1
> z[5]
6
```

Notice that the indexes into vectors in Eidos begin with index `0`, so the element with value `1` is at index `0`. Eidos is a "zero-based" language (like C, C++, Java, and Python); if you're coming from a one-based language (such as R), this will take a little re-wiring of your brain.

Let's try a few experiments to see what Eidos does:

```
> y*z
3 6 9 12 15 18 21 24 27 30
```

This is one type of *vectorized multiplication*: each element in the vector `z` was multiplied by the element in the singleton vector `y`. Let's try something different:

```
> z*z
1 4 9 16 25 36 49 64 81 100
```

This is another type of vectorized multiplication: with two vectors of the same length (`z` and `z`, in this case), pairs of corresponding elements from the two vectors are operated upon. The end result is a vector of the squares of the elements of `z` – each element of `z` has been multiplied by itself.

Many of the operators in Eidos are vectorized, generally following these same rules for $N\times1$ and $1\times N$ operations (the first case above) and $N\times N$ operations (the second case above). A full understanding of vector operations is essential knowledge for using Eidos and SLiM, so let's try some more experiments:

```
> x ^ (0:3)
1 7 49 343
```

As mentioned before, `^` is the exponentiation operator, so this makes sense; `x` (which is `7`) is raised to the power of each of the elements in `0:3`. But note the parentheses grouping this expression; try doing this instead:

```
> x ^ 0:3
1 2 3
```

What happened here? The `^` operator has higher *precedence* than the `:` operator, so this is equivalent to writing `(x^0):3`; `x^0` is `1`, so it produces the sequence `1:3`. This is essentially the same as the confusion that results if one writes "3 × 1+2"; the spacing of the expression can mislead the eye, but multiplication has higher precedence than addition, in standard algebra, and so the result is (3×1)+2, not 3×(1+2). It is wise to use parentheses to group expressions whenever one is not sure of the precedence rules that apply.

OK, a few more experiments, which will not be commented on here; just try them yourself and make sure you understand why the result is what it is:

```
> 10:1
> (10:1) + z
> z[4:6]
> z[6:4]
> z[z-1]
> z[10-z]
```

EXERCISE: Try some more experiments with constructing sequences with `:`, doing arithmetic operations that mix vectors and singletons, and subsetting vectors using `[]`. Try to think of cases you can't predict the result of, and see what Eidos does with them.

3. For loops

Let's continue playing in the Eidos console a bit longer, but switch to a new topic: loops, specifically `for` loops. Let's write a simple `for` loop in the console:

```
> for (i in 1:10) print(i)
```

This prints the values from `1` to `10`, but they are on separate lines; this is not a vector containing `1:10`, but separate singleton values `1` through `10`. What happened? The statement above essentially says "Execute the body of the loop (which is `print(i)`) once for each element of `1:10`, assigning the current element each time into a new variable named `i`." So it assigns `1` to `i`, and then executes `print(i)`, and then assigns `2` to `i`, and then executes `print(i)`, and so forth up to assigning `10` to `i`, and then executing `print(i)`. Try doing this:

```
> for (i in 1:10) print(i^3)
```

Make sense? Let's try something a bit trickier; execute all three of these statements, in order:

```
> x = 0
> for (i in 1:10) x = x + i;
> print(x)
```

If it is not clear what happened, ponder it for a while. There is an easier way to do this in Eidos:

```
> sum(1:10)
```

OK, with this understanding of `for` loops we should have all the tools we need to understand this model, so close the Eidos console for now.

4. Explore the model

If you're looking at the model window itself now, you should see this script:

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
1 {
   subpopCount = 10;
   for (i in 1:subpopCount)
      sim.addSubpop(i, 500);
   for (i in 2:subpopCount)
      sim.subpopulations[i-1].setMigrationRates(i-1, 0.2);
   for (i in 1:(subpopCount-1))
      sim.subpopulations[i-1].setMigrationRates(i+1, 0.05);
}
10000 late() { sim.outputFixedMutations(); }
```

Click the Play button in the play controls at upper right, and let the model play to the end. The population view will list ten subpopulations, each with 500 individuals. When the model completes, a chunk of output will be generated (we will talk about these output formats in a later lesson).

There are ten subpopulations because of the structure of the model's script. The `initialize()` callback is essentially the same as we saw before (without the comments, now). In generation 1, however, we are doing some new things!

First we assign `10` to the variable `subpopCount`; this is the number of subpopulations we intend to create. Then we loop, with `for (i in 1:subpopCount)`, and create ten new subpopulations, each with `500` individuals. The only surprising thing about that should be that we can pass an integer `i` to `addSubpop()`; before we passed a string, `"p1"`, which became the name of the new subpopulation. Passing an integer works too; if you pass `1` you create `p1`, if you pass `2` you create `p2`, etc. This is generally true in SLiM; whenever possible you can refer to entities like subpopulations, mutation types, and genomic element types by name (`p1`, `m1`, `g1`) or by their integer identifier.

Next we have a `for` loop:

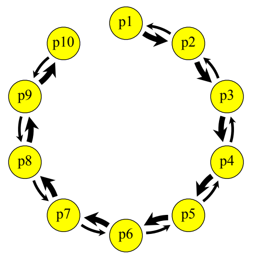
```
   for (i in 2:subpopCount)
      sim.subpopulations[i-1].setMigrationRates(i-1, 0.2);
```

This requires some explanation. We're looping from `2:10`, and the purpose of the loop is to tell SLiM about migration *from* the previous subpopulation *to* that subpopulation: from `p1` to `p2`, from `p2` to `p3`, and so forth. The expression `sim.subpopulations` accesses a *property* named `subpopulations` on the simulation object `sim`. Properties are basically just values contained by an object, and they are accessed with a `.` (the *member access operator*) just like method calls are. The `sim` object manages the subpopulations, and so we can ask it for the subpopulations it is managing; that is all that `sim.subpopulations` means. The result is a vector, of length 10, containing the ten subpopulation objects, in order: `p1 p2 p3 ... p10`. We fetch the particular subpopulation we want by subsetting that vector with `[i-1]`; if `i` is 2, we want `p2`, which is at index `1` in the vector (because Eidos is zero-based, again). Having extracted the subpopulation we want, we then call the `setMigrationRates()` method to tell SLiM about the migration *to* it, *from* the previous subpopulation (the one with identifier `i-1` – for example, `p1`'s identifier is `1`, so when `i` is `2`, migration comes from `p1`). We want migration to occur at a rate of *m*=0.2 (20% migration per generation). Understanding the distinction between indexes (zero-based) and identifiers (one-based, here) is essential; ponder this code until you get it. (To avoid this confusion, one could define `p0`…`p9` instead of `p1`…`p10`.)

And then we do much the same again, but setting migration to the *previous* subpopulation: from `p2` to `p1`, from `p3` to `p2`, and so forth. Work through the logic there until it is completely clear.

Since we have already played this model, we can open a graph window that provides us with a nice visualization of the migration pattern we've set up. Find this button above the output pane:

Click it and a pop-up menu will appear. Choose **Graph Population Visualization** and a new window will appear, showing this:



This shows `p1` through `p10`, with a higher migration rate (thicker arrows) in the "downstream" direction, from `p1` down to `p10`, and a lower migration rate in the "upstream" direction, from `p10` to `p1`. The terms "downstream" and "upstream" are deliberate; this population structure could represent a river system, for example, for a species that has trouble dispersing against the current.

5. Examine the model in Eidos

Now that we understand how this model works, let's look at how it is structured in Eidos. Open the Eidos console again, and then find and click this button above the console pane:

This clears the console, giving you a clean slate on which to work. You could also press command-K (hold down the command key, marked with a ⌘ icon, and press the K key, for you non-Mac folks) to do the same thing. (Again, whenever you're wondering what a particular button in SLiMgui does you can hover the mouse over it to see a tooltip – in this case, "clear console".) Now type this:

```
> sim
SLiMSim
```

This tells us that the variable `sim` is an object of class `SLiMSim`, which is the Eidos class that represents the simulation object. We'd like to know more about it; try this:

```
> sim.str()
```

The method name `str()` is short for "structure"; it prints the properties inside an object. There's a lot there; `sim` keeps track of mutation types and genomic element types, the chromosome object, the segregating mutations, and a good deal more. One thing you will see in there is this:

```
subpopulations => object<Subpopulation> [0:9] Subpopulation<p1> ...
```

This is the `subpopulations` property we saw before. This line tells us that it is an `object` vector of class `Subpopulation` (the kind of object elements it holds), that indices defined for this vector are in `0:9`, and that the first element in the vector is "`Subpopulation<p1>`". We can access this property:

```
> sim.subpopulations
Subpopulation<p1> Subpopulation<p2> Subpopulation<p3> Subpopulation<p4>
Subpopulation<p5> Subpopulation<p6> Subpopulation<p7> Subpopulation<p8>
Subpopulation<p9> Subpopulation<p10>
```

This makes sense. We can also access these subpopulations directly, through global constants:

```
> p1
Subpopulation<p1>
```

Let's look at the structure of `p1`:

```
> p1.str()
```

Various things are in there, but the one we're interested in now is this one:

```
individuals => object<Individual> [0:499] Individual<p1:i0> ...
```

Just as `sim` contains subpopulations, each subpopulation contains individuals. Let's get one individual and put it in a variable:

```
> ind = p1.individuals[0]
```

Now let's look at its structure:

```
> ind.str()
```

Again, lots of stuff (all documented in the SLiM manual, of course), but we're interested in this:

```
genomes => object<Genome> [0:1] Genome<A:45> Genome<A:51>
```

This property, `genomes`, returns a vector of length `2` that contains the two genomes of the individual. Let's get one and look at its structure:

```
> ind.genomes[0].str()
```

The interesting bit, for our purposes, is this:

```
mutations => object<Mutation> [0:44] Mutation<794628:0> ...
```

The genome has a property, `mutations`, that returns a vector of all of the mutations contained by that genome. We have thus traced the whole population hierarchy, from the simulation object `sim`, to subpopulations, individuals, genomes, and finally mutations. You could do:

```
> ind.genomes[0].mutations[0].str()
```

and see that the mutations have properties for their position, their selection coefficient, and the mutation type to which they belong, among other things. As well as understanding the object graph, make sure you understand the syntax of how that last statement works! And finally, note that you can do all this in the Eidos console with a running (paused) simulation too – useful for debugging.

6. BEEP! With extra time, SLiM manual section 5.3 has a variety of population structure models.