1. Open recipe 5.3.1 in SLiMgui

You should still have recipe 5.3.1 open in SLiMgui, played all the way to the end. If not, open it now, click Play, and wait until it is done running. This will give us a simulation state we can explore.

2. Exploring mutations with type logical and logical subsetting of vectors

Open the Eidos console if it isn't already open, and press command-K to clear any existing output. Now execute:

```
> muts = sim.mutations
```

This creates a new variable, `muts`, that is a vector of all of the mutations segregating in the simulation at the point when it stopped. There are probably quite a few; try doing:

```
> size(muts)
924
```

There are 924 in the run I did when preparing this exercise. As we saw in the previous exercise, we can print all of the properties of one of these mutations using the `str()` method:

```
> muts[0].str()
```

One of the properties listed is `position`, which is the base position of the mutation along the chromosome. Let's get the positions of all of the mutations:

```
> positions = muts.position
> size(positions)
924
> positions
46113 17216 88540 80769 19411 ...
```

We have just done a *vectorized property access*. For each of the 924 mutations in `muts`, the `position` property was accessed, fetching an integer value, and all 924 values were stuck together to form a single result vector of type `integer`. The result vector is guaranteed to be in the same order as `muts`; in other words, the position of `muts[0]` is `46113`, the position of `muts[1]` is `17216`, etc.

Suppose we'd like to find all mutations with a position less than 10000. This is very simple, but there are some important concepts to develop along the way, so we're going to walk through it very slowly. First of all, let's try using a *comparison operator*, `<`, with the positions vector:

```
> positions < 10000
F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F F
T F F F F F F F F F F F F F F F F F F F F F F F F F F T T F F ...
```

The result is a vector, again of length 924, of type `logical`: true (`T`) and false (`F`) values. Mostly the values are `F`, because most mutations have positions greater than 10000, but there are a few `T` values. Again, the same order is guaranteed; these `T`/`F` values specify whether the corresponding position in `positions` is less than 10000.

Let's put this logical vector into a variable so we can play with it a bit:

```
> small = positions < 10000
```

We can do some useful things with logical vectors in Eidos. Try each of these examples:

```
> all(small)
> any(small)
> sum(small)
> mean(small)
```

The first tells you whether *all* of the values in `small` are `T`; that is probably `F` for you. The second tells whether *any* of the values are `T`; that is probably `T` for you. The third tells you *how many* values are `T`; for my test run this is 93. The `sum()` function works for this purpose because in Eidos, as in many programming languages, `T` is considered to have an integer value of `1`, and `F` is considered to have an integer value of `0`, so `sum(small)` is the number of `T` values. Finally, the fourth statement tells you the *fraction* of `T` values in `small`, again relying on the fact that `T` is `1` and `F` is `0`; for my run this is `0.100649`.

Although we have been working with intermediate variables to make each step simple and clear, there is no need for those variables; we can obtain the same results directly in Eidos with:

```
> mean(sim.mutations.position < 10000)
```

This fetches the global variable `sim` (giving a value of type object and class `SLiMSim`), gets the `mutations` property from it (producing a result vector of type `object` and class `Mutation`), gets the `position` property from that result (producing a result vector of type `integer`), compares that result to `10000` (producing a result vector of type `logical`) and takes the mean of the result (producing a vector of length 1 – a singleton – of type `float`). The vectorization of Eidos makes it extremely concise, and highly efficient as well, so this sort of expression is common in Eidos code; make sure you understand how it works.

But we've gone off on a tangent; we were trying to get the mutations with position less than 10000. We now have everything we need to do that, so we can just write:

```
> muts[small]
Mutation<998309:0> Mutation<997388:0> Mutation<999102:0> ...
```

The result is a vector of specifically the mutations with a position less than 10000. (The first value printed for each mutation is its id, a unique identifying integer, and the second value is its selection coefficient). Remember that `small` has a `T`/`F` value that tells us whether each corresponding mutation in `muts` has a small position (`T`) or not (`F`). We saw the subset operator, `[]`, before with an integer index like `muts[0]`, but it can also take a logical vector as an operand, in which case it selects specifically the values for which the logical vector is `T`. To make that more clear, here's a simple example:

```
> x = 1:10
> x[c(T, F, T, T, F, F, F, T, F, T)]
1 3 4 8 10
```

This makes `x` be a sequence from `1` to `10`, with `:`, and then selects the first, third, fourth, eighth, and tenth values out of that sequence using `[]` with a logical vector as the operand. Here we constructed the logical vector using the `c()` function, which simply concatenates its arguments to form a single vector.

Again, we could have selected the mutations of interest with a single expression instead of using temporary variables, with:

```
> muts[muts.position < 10000]
```

or even:

```
> sim.mutations[sim.mutations.position < 10000]
```

The second version is less efficient, however, since it fetches the whole vector of mutations from `sim.mutations` twice. Using the temporary variable `muts` prevents that double-fetch, and that version is also a bit easier to read, so it is better. In general, it is often good practice to introduce a temporary variable if your code would otherwise have to calculate the same sub-expression twice.

Let's put the result into a new variable:

```
> small_muts = muts[small]
```

EXERCISE: To review the above concepts, now try to find the mutations that (a) have a position of less than 10000, *and* (b) originated in generation 9900 or later. To do this, you may wish to start with the `small_muts` variable created above, and then narrow down to the subset of mutations that also originated in 9900 or later using the `originGeneration` property of `Mutation`.

Let's now look at an alternative way to subset on more than one condition (position < 10000 *and* origin generation > 9900). Let's work with the vector `x` we constructed above. Suppose we want to subset `x` to get the values in `x` that are even. We can use the *modulo operator*, `%`, to find out which values are even. This operator provides the remainder after division, like so:

```
> x % 2
1 0 1 0 1 0 1 0 1 0
```

Even numbers have a remainder of `0` after division by two, so we can get a logical vector selecting the even values of `x` like so:

```
> x % 2 == 0
F T F T F T F T F T
```

And then we can subset `x` with that vector to get the values of `x` that are even:

```
> x[x % 2 == 0]
2 4 6 8 10
```

Make sure that this makes sense, because we're about to take it up a level. Logical vectors can be combined with the *and operator*, `&` (an ampersand), and the *or operator*, `|` (a vertical bar, typed as a shift-backslash on American English keyboards). For example:

```
> l1 = c(T, F, T, T, F, F, F, T, F, T)
> l2 = x % 2 == 0
> l1
T F T T F F F T F T
> l2
F T F T F T F T F T
> l1 & l2
F F F T F F F T F T
> l1 | l2
T T T T F T F T F T
```

The `&` operator provides a result vector that is `T` only in the positions where its first operand was `T` *and* its second operand was `T`. The `|` operator provides a result vector that is `T` in positions where its first operand was `T` *or* its second operand was `T`.

Knowing this, we can subset the values of `x` that satisfy both criteria:

```
> x[l1 & l2]
4 8 10
```

or, without the temporary variables:

```
> x[c(T, F, T, T, F, F, F, T, F, T) & (x % 2 == 0)]
4 8 10
```

EXERCISE: The `!` operator is the logical not operator, which flips `T` to `F` and `F` to `T`. Try comparing the results of `(x % 2 == 0)` and `!(x % 2 == 0)`. Rewrite the last expression above to select the odd values rather than the even values from `x[c(T, F, T, T, F, F, F, T, F, T)]`.

EXERCISE: Use operator `&` to, again, derive the same subset of mutations as before (position < 10000 *and* origin generation > 9900), but do so with a single-line expression based upon `muts`, not `small_muts`, and using no temporary variables at all. Once you have that, modify it with operator `!` to select the mutations whose position is *not* greater than 9900, instead.

Now you understand vectorized property access and logical subsetting, two of the most important concepts in Eidos programming. Let's move on!

3. Changing selection coefficients

Now we're going to modify the model a bit. We've run it all the way to the end, but we want to change its state in the middle of a run. To do this, first close the Eidos console and Recycle the simulation (which should still be recipe 5.3.1). Go to the Generation field (which should say "initialize()") and enter the value 5000:

Then press return. The simulation will run forward to generation 5000 and stop. Now open the Eidos console again. We'd like to take a sample of mutations from the simulation, and there's a function named `sample()` that can do that for us, but perhaps we don't remember how to call that function. Start typing (but don't press return):

```
> s = sample(
```

Notice that at the bottom of the console window there is a status line that is showing us something:

```
(*)sample(* x, integer$ size, [logical$ replace = F], [Nif weights = NULL])
```

That is the function prototype for `sample()`. We saw these briefly in the lecture; let's dissect this prototype. The `sample()` function takes four parameters, named `x`, `size`, `replace`, and `weights`, and it returns a value of type `*` – remember, that means it could be *any* type. The `x` parameter is also type `*`, whereas `size` and `replace` must be logical singletons (the `$`), and `weights` can be either `NULL` (`N`), `integer` (`i`), or `float` (`f`). The `replace` and `weights` parameters are optional (the brackets around them), with default values of `F` and `NULL`, respectively.

That might be enough information get us going, but let's suppose it isn't. With the Eidos console still showing the partially complete line

```
> s = sample(
```

hold down the "option" key on your keyboard (sometimes labeled "alt" or "⌥"), and click on the word "sample" in the console. SLiMgui's help window will open, showing the results of a search on

"sample". There are multiple results (there are some SLiM methods that contain "sample" in their name), but the first result is the one we want. Its text should look something like this:

> `(*)sample(* x, integer$ size, [logical$ replace = F], [Nif weights = NULL])`
>
> Returns a vector of `size` containing a **sample from the elements of `x`**. If `replace` is `T`, sampling is conducted with replacement (the same element may be drawn more than once); if it is `F` (the default), then sampling is done without replacement. A vector of weights may be supplied…

That's the information we needed: `x` is a vector of values from which a sample will be drawn, and `size` is the size of the sample to draw. Close the help window and complete the line in the console:

```
> s = sample(sim.mutations, 20)
```

This draws a sample of twenty mutations, at random, from the mutations segregating in the simulation. We could just as well type:

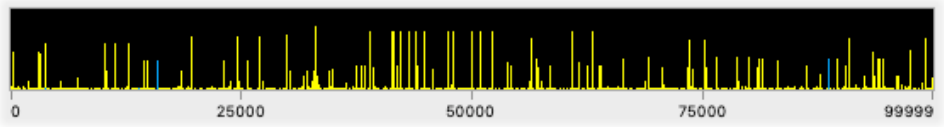
```
> s = sample(x=sim.mutations, size=20)
```

Parameter names are optional; they make code more self-documenting when it would otherwise be unclear. Now let's shake things up a bit, by typing:

```
> s.setSelectionCoeff(0.05)
```

This is a vectorized method call; for every object element in `s`, the `setSelectionCoeff()` method is called with an argument of `0.05`. This is conceptually similar to the vectorized property access we saw before, but instead of simply fetching a property from each element, it makes a whole method call on each element. This change could represent a change in the environment; perhaps there were some mutations segregating in the population that conferred drought resistance, but there was plenty of rainfall and so those mutations were effectively neutral until generation 5000, when a drought began and those mutations suddenly became beneficial.

To see the effect of this change, close the Eidos console now, and in the SLiMgui model window, click Step once. You will probably see that a couple of mutations in the chromosome view have turned blue, as in this screenshot:



This is because they now have a selection coefficient of `0.05` – they aren't neutral any more. SLiMgui colors neutral mutations yellow; other mutations receive other colors, depending on their selection coefficient. The individuals view probably also looks a bit different, like:

Some individuals now have non-neutral fitness, because they carry a beneficial mutation, and so they are colored according to their fitness. Now click Play, and watch the beneficial mutations; they will probably rise in frequency and then fix. When they fix, they disappear from view since they become substitutions, as we have seen before. When this happens, notice that all the individuals also revert to yellow; they all still carry the fixed beneficial mutations, but since they are fixed they produce no relative fitness differences among individuals, so individuals are again effectively neutral.

We modified the simulation while it was running, as an experiment. Since the experiment seemed to go well, we'd like to add it as a standard behavior in the model. To do this, stop the simulation, if it is running, and then add the following code to the model's script:

```
5000 late() {
   s = sample(sim.mutations, 20);
   s.setSelectionCoeff(0.05);
}
```

It makes sense to insert it just above the `10000 late()` event, but as long as you add it at the top level (i.e., not inside some existing block of code), it doesn't actually matter; the generation `5000` in the declaration tells SLiM when it runs, not its position in the script.

Having added that, now Recycle and Play. The event will run in generation 5000, and you can watch the sweep of the beneficial mutations. Note that we changed twenty mutations to be beneficial, but you might only see one or two sweep; the rest might be lost due to drift, or due to the competition from the other beneficial mutations. For more than one of them to fix, they have to end up together in the same genome; that might happen by chance in the initial configuration, otherwise it will have to happen as a result of recombination.

EXERCISE: Add another `late()` event that makes the model revert to neutral in generation 5050. To do it the easy way, just change the selection coefficients of *all* mutations to `0.0`. To do it the hard way, use what you've learned in this exercise to select *only* the modified mutations, and change only their selection coefficients to `0.0`. (Use `> 0.0` as the comparison to subset the relevant mutations; a comparison using the equality operator, `== 0.05`, will not work because of floating-point roundoff issues. Never use == to test for equality between two floating-point values! Also, note that the property of `Mutation` that provides the selection coefficient is named `selectionCoeff`.)

When you've completed this exercise, comment out the generation 5050 event you just wrote. To do this, select its code, and press command-/. This comments out the selection, or uncomments it if you do it with a commented selection. Press command-/ again to uncomment, and command-/ to re-comment. Leave that event commented out, in the end, so that it doesn't interfere with the next exercise.

EXERCISE: Experiment with changing the sample size and and the selection coefficient that is set in generation 5000. What happens with a sample size of `200` and a selection coefficient of `-0.01`? Keep in mind you can use the play speed slider to slow down the model when it gets close to generation 5000, to see what's happening more clearly.

EXERCISE: Click inside the parentheses of various functions and methods in the script, and look at the function signature shown for them at the bottom of SLiMgui's window. Choose one or two that you're curious about, and option-click their name to bring up the documentation for them to learn more.

4. BEEP! With extra time, you might browse the Eidos manual on basic language topics if you want to review them (chapter 2), or you might look at the built-in functions in Eidos (chapter 3).