SLiM Workshop Series #10: Callbacks I: `fitness()`

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 default script to the following code, or clear the window and create this script from scratch (feel free to leave out the comments, of course):

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);   // neutral
   initializeMutationType("m2", 0.5, "f", 0.02);
   initializeMutationType("m3", 0.5, "f", 0.02);
   initializeGenomicElementType("g1", c(m1,m2,m3), c(1,0.01,0.01));
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
1 {
   sim.addSubpop("p1", 500);
   sim.addSubpop("p2", 500);
   p1.setMigrationRates(p2, 0.1);   // weak migration p2 -> p1
   p2.setMigrationRates(p1, 0.5);   // strong migration p1 -> p2
}
50000 {
   catn("m2 fixed: " + sum(sim.substitutions.mutationType == m2));
   catn("m3 fixed: " + sum(sim.substitutions.mutationType == m3));
}
```

This sets up a simple two-subpopulation model with weak migration in one direction and strong migration in the other, as can be seen in the Population Visualization Graph window:

If you recycle and run, you will get output like this:

```
m2 fixed: 65
m3 fixed: 67
```

The `m2` and `m3` mutations arise at the same rate (about 1% as often as neutral `m1` mutations), and the model treats them identically, so they fix at the same rate, on average – the difference between `65` and `67` here is just due to chance.

We'd like to introduce spatial variation in selection into this model – we'd like `m2` mutations to be beneficial in `p1`, with their given selection coefficient of s=`0.02`, but to be deleterious in `p2`. Symmetrically, we'd like `m3` mutations to use s=`0.02` in `p2`, but to be deleterious in `p1`. SLiM just provides a single selection coefficient for each mutation, though, and it is in charge of the fitness calculations. How to modify this default behavior? The answer, of course, is to write `fitness()` callbacks.

2. Adding `fitness()` callbacks

Add these lines to the model (it doesn't matter where, at the top level – perhaps between the two events):

```
fitness(m2, p2) { return 0.98; }
fitness(m3, p1) { return 0.98; }
```

The first line is a `fitness()` callback that is declared as applying specifically to mutation type `m2` and subpopulation `p2`. Our goal is to modify the `m2` mutations to be deleterious in `p2`, and that is achieved here by returning `0.98`. A very important but confusing point is: mutations have selection coefficients, which are on a scale where `0.0` is neutral, whereas `fitness()` callbacks return a *fitness effect*, which is on a scale where `1.0` is neutral. The fitness effect of a mutation of selection coefficient s is defined as $(1+s)$ for homozygotes, but as $(1+hs)$ for heterozygotes, where h is the dominance coefficient. This formulation is standard practice, but when the two scales are mixed together it can certainly be confusing. The `0.98` return value represents the fitness effect that would result from a selection coefficient of `-0.02`, if the individual were homozygous (a point we will return to shortly).

The second line above is the symmetric `fitness()` callback to render `m3` mutations deleterious in `p1`; that probably doesn't require any explanation.

What exactly does SLiM do with these callbacks? It calls out to them once for every *focal mutation* within every *focal individual*, and it uses the value returned as the fitness effect for that mutation in that individual, completely overriding the default fitness effect that it would otherwise use. If the same mutation occurs in ten different individuals, the applicable `fitness()` callback (or callbacks) will be called ten times – once for that mutation in each of the ten individuals. This allows the fitness effect of a given mutation to vary from individual to individual, depending upon the focal individual's genetic state (epistasis), or its spatial position (spatially varying selection), or a random draw (environmental variance), or anything else at all. This makes `fitness()` callbacks very powerful, but also potentially a performance bottleneck since they are called so often; beware of making complex, slow calculations inside a frequently called callback.

If we recycle and run this model, we get a very different result than last time:

```
m2 fixed: 55
m3 fixed: 0
```

This difference is certainly *not* due to chance. Rather, it is because of the biased gene flow in this model, as seen in the Population Visualization Graph snapshot above. The strong gene flow from `p1` to `p2` allows it to force `p2` to fix mutations that are deleterious in `p2` – `m2` mutations. The weaker gene flow in the opposite direction, on the other hand, means that `m3` mutations that arise in `p2` will be lost, even though they are beneficial in `p2`; `p2` will not be able to push them over to `p1` effectively, and they will be swamped by the gene flow from `p1`.

So, great – we have a model of spatial variation in selection, and it presents interesting results. We could use it to explore the effects of different levels of gene flow, different strengths of selection, different population sizes, and so forth, in order to fully understand the dynamics it exhibits. Are we done?

We are not, because we have neglected one key issue: heterozygosity. Note that `m2` and `m3` specify a dominance coefficient of `0.5`. This means that in the subpopulation where they are beneficial, they have a fitness effect of $(1+s)$, or `1.02`, for homozygotes, but a fitness effect of $(1+hs)$, or `1.01`, for heterozygotes. The `fitness()` callbacks, however, always return a fitness effect of `0.98`, regardless of zygosity. If we wanted the `m2` mutations to exhibit complete dominance in their deleterious fitness effect when in the "wrong" subpopulation, then this design would be correct; but given that they exhibit partial dominance when beneficial, it feels like this is a bug we need to fix.

3. The `homozygous` pseudo-parameter

As explained above, `fitness()` callbacks are called once per focal individual, for a given focal mutation. To underline that point: whether the focal individual is heterozygous or homozygous, either way, a given callback is called *once* for that focal individual. We therefore need our `fitness()` callbacks to take zygosity into account in the values they return. But how does the callback's code know whether the focal individual is homozygous or heterozygous?

The answer is that it is passed that information in what SLiM calls a *pseudo-parameter*. A pseudo-parameter is like a parameter to a function: a value passed in from outside that provides the called code with information it needs. Pseudo-parameters are "pseudo" because callbacks aren't functions, exactly; and because the pseudo-parameters are implicit in SLiM's design, not declared by the callback. But these are technical distinctions; you can think of them, for all practical purposes, as being parameters passed to the callback.

One such pseudo-parameter is named `homozygous`, and as you might expect, it is a `logical` singleton flag that is `T` if the focal individual is homozygous for the focal mutation, or `F` if the focal individual is heterozygous for the focal mutation. This is all we need to fix our callbacks:

```
fitness(m2, p2) { return homozygous ? 0.98 else 0.99; }
fitness(m3, p1) { return homozygous ? 0.98 else 0.99; }
```

As we saw in a previous exercise, the `?else` operator is like a little `if-else` statement. The logic here thus returns `0.98` if the focal individual is homozygous, `0.99` if it is heterozygous. This implements a selection coefficient of `-0.02` and a dominance coefficient of `0.5`, in effect.

If we run this model, we get this:

```
m2 fixed: 61
m3 fixed: 0
```

The change doesn't seem to have made much difference, but depending upon the various model parameters it might sometimes. Perhaps selection against the `m3` mutations in `p1` is weaker now, because of partial dominance, allowing those mutations to make more progress in `p1` before the selection in `p1` starts to be effective, allowing more `m2` mutations to fix, and perhaps allowing an occasional `m3` mutation to fix as well. It's a bit hard to say without doing lots of replicate runs.

If you option-click on the "fitness" keyword in the script, SLiMgui will show online help regarding `fitness()` callbacks. Scroll down a bit, and you will see a table of the pseudo-parameters passed in to them, among which is `homozygous`. Others include the focal mutation itself, the focal individual, and the focal individual's genomes and subpopulation; these can be useful for implementing various types of fitness effects like epistasis. And then there is one other pseudo-parameter: `relFitness`.

4. The `relFitness` pseudo-parameter

To understand the purpose of `relFitness`, let's change our model a bit: let's draw the selection coefficients for `m2` and `m3` from an exponential distribution, rather than using a fixed selection coefficient, by replacing their declarations with the following:

```
initializeMutationType("m2", 0.5, "e", 0.02);
initializeMutationType("m3", 0.5, "e", 0.02);
```

All that has changed is that the the `"f"` DFE type has been replaced with `"e"`. But this has rendered our `fitness()` callbacks incorrect: presumably a mutation with a very *small* beneficial effect should have a very *small* deleterious effect in the opposite subpopulation, whereas a mutation with a *large* beneficial effect should have a *large* deleterious effect in the opposite subpopulation. Right now, however, we have the deleterious fitness effect hard-coded as `0.98` or `0.99` depending on the individual's zygosity.

What to do? One option would be to get the focal mutation (passed in through pseudo-parameter `mut`), and use it to figure out the correct fitness value. We would make the `fitness()` callback calculate the standard fitness effect, using $(1+s)$ or $(1+hs)$, but with a selection coefficient of the opposite sign: $(1-s)$ or $(1-hs)$, in other words. That would look something like this:

```
fitness(m2, p2) {
   if (homozygous)
      return 1.0 - mut.selectionCoeff;
   else
      return 1.0 - mut.selectionCoeff * mut.mutationType.dominanceCoeff;
}
```

And likewise for the other callback. This strategy works fine, but it's a bit verbose and messy. There is a simpler option, using `relFitness`:

```
fitness(m2, p2) { return 2.0 - relFitness; }
```

And likewise for the other callback. This does precisely the same thing as the more verbose code above, but it is simpler and faster. It works because `relFitness` is SLiM's calculated fitness effect for the focal mutation, given its zygosity in the focal individual, using the standard SLiM fitness machinery. Perhaps this is `1.1`, for a particular focal mutation; the calculated fitness effect returned by the callback with then be `0.9`. Perhaps it is `1.03`; the calculated fitness effect would then be `0.97`. A small beneficial effect becomes a small deleterious effect, while a large beneficial effect becomes a large deleterious effect.

A run with these `fitness()` callbacks produces a similar result:

```
m2 fixed: 60
m3 fixed: 0
```

Incidentally, for this sort of flipping beneficial to deleterious and deleterious to beneficial, an alternative formulation that you will sometimes see looks like this:

```
fitness(m2, p2) { return 1.0 / relFitness; }
```

For small selection coefficients this is almost the same; for `1.1` it returns `0.90909` (not far from `0.9`), and for `1.03` it returns `0.97087` (not far from `0.97`). It might be considered better because it accounts for the fact that the range of fitness effects is $[0, \infty]$: a fitness effect of zero is lethal (and you don't get worse than lethal, so negative values are meaningless), whereas a fitness effect of ∞ is infinitely superior to all non-infinite-fitness individuals (the opposite of lethal). Using `1.0/relFitness` means that a beneficial fitness effect of, say, `5.0` will be converted into a deleterious fitness effect of `0.2`, rather than a meaningless "worse than lethal" value of `-3.0`. Probably that is better, for most purposes; for small selection coefficients it probably makes little difference either way.

EXERCISE: Make `m2` and `m3` mutations be beneficial in one subpopulation but neutral in the other, rather than deleterious. How does this affect the result of the model, and why?

EXERCISE: Convert the model from a hermaphroditic model to a sexual model by adding the call `initializeSex("A");` to the `initialize()` callback (no other modification is needed). In the Wright–Fisher model, this means that females are now selected as the first parent (with probability proportional to the relative fitness of the females), and then each female chosen as a first parent chooses a male to cross with (with probability proportional to the relative fitness of the males). The focal individual whose fitness is being evaluated is provided through the pseudo-parameter `individual`, and the `Individual` class has a property named `sex` that is `"M"` for males, `"F"` for females, so `individual.sex == "M"` is `T` if the focal individual is male, `F` if it is female. Modify the `fitness()` callbacks so that where `m2` and `m3` mutations were neutral, they are now deleterious in

males (but still neutral in females). In the subpopulations where they were beneficial, leave their beneficial effects as they were, for both males and females. To do this, use either `if-else` statements or the `?else` operator. Watch the model run in SLiMgui, and you should notice an interesting visual effect: you can see that as `m2` mutations sweep to fixation, they now have a deleterious effect in only *half* of the individuals in `p2` – the males.

EXERCISE: If you chose `if-else` for the previous exercise, convert the code to `?else`, or *vice versa*, just to get some practice with both ways of handling conditionality in Eidos.

5. Frequency-dependent selection

In the previous sections we developed a two-subpopulation model with spatial variation in selection using `fitness()` callbacks. Let's switch gears completely, to look at a different use of `fitness()` callbacks. Close your existing SLiMgui window, open a new default model with command-N, and modify the model to look like this:

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);    // neutral
   initializeMutationType("m2", 0.5, "f", 0.1);    // balanced
   initializeGenomicElementType("g1", c(m1,m2), c(99999,1));
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
1 { sim.addSubpop("p1", 500); }
100000 { sim.simulationFinished(); }
fitness(m2) {
   return 1.5 - sim.mutationFrequencies(p1, mut);
}
```

This is a single-subpopulation model with two mutation types: one, `m1`, that is very common, and one, `m2`, that is fairly rare according to the weights passed to `initializeGenomicElementType()`. The `m2` mutation type is given a fixed DFE with a dominance coefficient of `0.5` and a selection coefficient of `0.1`, but the `fitness(m2)` callback completely overrides those settings; the dominance coefficient and selection coefficient passed to `initializeMutationType()` end up being irrelevant here. Instead, the `fitness(m2)` callback does something rather different from what we've seen:

```
   return 1.5 - sim.mutationFrequencies(p1, mut);
```

This makes the fitness effect of an `m2` mutation depend on its current frequency in the population. If it is at a frequency near zero, the fitness effect will be close to 1.5 – strongly beneficial. If it is at a frequency near one (near fixation), the fitness effect will be close to 0.5 – strongly deleterious. As a result, `m2` mutations will be under "balancing selection" – favored when at low frequency, disfavored when at high frequency. The particular type of balancing selection implemented here is called "negative frequency-dependent selection"; "negative" because low frequency results in high fitness, and vice versa, so the correlation between frequency and fitness is negative. (With positive frequency-dependent selection, the correlation is positive, causing mutations to tend to be lost when at low frequency, but to sweep rapidly to fixation if they happen to make it to higher frequency.)

If you run this model, you will see (once some `m2` mutations arise) some curious dynamics, with neutral mutations as well as `m2` mutations hovering at a frequency of about 0.5. The whole model segregates into two chromosome-wide haplotypes that are mostly independent. Occasionally this state of affairs will become unstable, and a section of the genome will suddenly sweep to fixation, interestingly. This is likely due to recombination, in which a balanced mutation finds itself on the opposite genetic background, plus some drift-type dynamics that then happen to carry that recombined background upward in frequency. Frequency-dependent selection does strange things.

EXERCISE: This model has the same flaw that the previous one did: the `fitness()` callback does not take zygosity into account. As a result, being heterozygous for a given `m2` mutation produces exactly the same fitness effect as being homozygous for that `m2` mutation; the `m2` mutations exhibit complete dominance. Use the `homozygous` pseudo-parameter to fix this problem, by making the effect of heterozygous mutations half as large: a fitness effect of `1.25` when the mutation is at very low frequency, and a fitness effect of `0.75` when the mutation is at very high frequency. Recycle and run the model; can you see a difference in the dynamics?

6. BEEP! With extra time, look at section 10.3.1 (epistasis) for another use of `fitness()` callbacks. Chapter 10 has several other interesting recipes using `fitness()` callbacks as well.