SLiM Workshop Series #11: Callbacks II: `mateChoice()`

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 create this script from scratch:

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);     // neutral
   initializeMutationType("m2", 0.5, "f", -0.025);  // ornamental
   initializeGenomicElementType("g1", c(m1, m2), c(1.0, 0.01));
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
1 { sim.addSubpop("p1", 500); }
1:10001 early() {
   if (sim.generation % 1000 == 1) {
      fixedMuts = sum(sim.substitutions.mutationType == m2);
      osize = fixedMuts * 2 + p1.individuals.countOfMutationsOfType(m2);
      catn(sim.generation + ": Mean ornament size == " + mean(osize));
   }
}
```

This model creates a single subpopulation that undergoes mostly neutral mutations. About one out of every hundred mutations is an `m2` mutation that affects the "ornament size" of the individual in a simple, linear fashion: the more `m2` mutations the individual has, the larger its ornament is. (Think of a peacock tail.) The `m2` mutations continue to affect ornament size even after they fix, because the model counts the number of `m2` substitutions, adding them in to the total. Segregating mutations are tallied up by the Individual method `countOfMutationsOfType()`, which counts each occurrence on either of the individual's genomes (i.e., a homozygous mutation counts for `2`).

Every thousand generations, as assessed by the module operator `%` (the remainder after division, remember, so `sim.generation % 1000` equals `1` in generations `1001`, `2001`, …), the mean ornament size across the population is calculated and logged to the output. This is done with a vectorized method call to `countOfMutationsOfType()`, across all of the individuals in `p1.individuals`, producing a vector of results, per individual, that is then averaged with `mean()`.

As the model stands, however, the ornament mutations are mildly deleterious (s = `-0.025`). When the model is recycled and run, ornament mutations arise but don't generally fix:

```
1: Mean ornament size == 0
1001: Mean ornament size == 0.006
2001: Mean ornament size == 0.002
3001: Mean ornament size == 0
4001: Mean ornament size == 0
5001: Mean ornament size == 0.002
6001: Mean ornament size == 0.018
7001: Mean ornament size == 0.002
8001: Mean ornament size == 0.012
9001: Mean ornament size == 0
10001: Mean ornament size == 0
```

2. Adding a `mateChoice()` callback

We would like ornaments to work like peacock tails: they decrease your fitness a little bit with respect to ecological performance (being large and energetically costly), but they make you more likely to be chosen by other individuals looking for a mate, because they're eye-catching and attractive. The first part – the direct fitness cost – is already implemented. We need to implement the second part – the indirect fitness benefit due to being chosen as a mate.

We will do that with a `mateChoice()` callback. This callback type influences how a focal *first parent* – an individual that is in the process of generating a new offspring – chooses its mate (referred to as the *second parent*). By default, SLiM chooses both first parents and second parents according to fitness: each individual is chosen with a probability that is proportional to its relative fitness (assessed within its subpopulation). But a `mateChoice()` callback can modify or replace this mating weights vector, influencing the probability that each individual will be chosen; or it can simply make the mate choice and return the chosen individual; or it can veto the first parent completely (to model some sort of infertility, for example); or it can tell SLiM to go with its default choice.

Here we will use the second strategy: our script will choose the mate for SLiM and return it. Add this callback code to the model (at the bottom is fine, it doesn't matter; and don't forget about code completion to help with typing these long code blocks):

```
mateChoice() {
   fixedMuts = sum(sim.substitutions.mutationType == m2);
   for (attempt in 1:5)
   {
      mate = sample(p1.individuals, 1, T, weights);
      osize = fixedMuts * 2 + mate.countOfMutationsOfType(m2);

      if (runif(1) < log(osize + 1) * 0.1 + attempt * 0.1)
         return mate;
   }
   return float(0);
}
```

This callback is called once for each focal first parent, during offspring generation in each generation. It begins by counting the number of fixed `m2` mutations, since it will need that information to calculate ornament sizes. Then it runs a `for` loop from `1` to `5`, representing five separate attempts by the focal first parent to choose a mate. This is a mating strategy commonly called "sequential mate search": look at a series of prospective mates, and if one is found that is of sufficient quality, choose it, otherwise continue looking.

Each time through the loop, a candidate mate is chosen from the subpopulation with `sample()`. The standard SLiM mating weights vector, provided to the callback as a pseudo-parameter named `weights`, is passed to the `sample()` function as weights for it; because of this, individuals with a large ornament will be somewhat less likely to be selected as a candidate (they are too busy preening their splendid tails, perhaps). Note that for simplicity we allow individuals to choose themselves as mates, in this hermaphroditic model; a more realistic model would likely be sexual anyway.

Next, the callback calculates the ornament size of the candidate mate, just as we saw before. It then does a probabilistic draw, using `runif()` (a draw from a uniform distribution), to determine whether the candidate mate is deemed adequate. The probability of acceptance increases with increasing ornament size, but on a log scale, so there are diminishing returns to increasingly large ornaments. The probability of acceptance also increased with the number of attempts made in the sequential search: the first parent becomes less choosy as time goes by, perhaps concluding that the pool of potential mates simply isn't very high quality. If the draw indicates that the mate is acceptable, it is returned to SLiM, which will proceed to generate a new offspring between the parents.

If all five attempts fail, and the `for` loop finishes unsuccessfully, the callback returns `float(0)`. This is Eidos syntax for a zero-length vector of type `float`, and it is a special signal to SLiM that indicates that the focal first parent is rejected as a potential parent. In this case, SLiM will go back to square one, choosing a new first parent and calling the `mateChoice()` callback again.

This callback, then, completely encapsulates a sequential mate choice strategy in which up to five candidate mates are examined for suitability, based upon their ornament size. If this model is run, the results are quite different than they were without the callback:

```
1: Mean ornament size == 0
1001: Mean ornament size == 5.454
2001: Mean ornament size == 6
3001: Mean ornament size == 6.002
4001: Mean ornament size == 6.462
5001: Mean ornament size == 8.058
6001: Mean ornament size == 8.006
7001: Mean ornament size == 8.044
8001: Mean ornament size == 8.04
9001: Mean ornament size == 8.466
10001: Mean ornament size == 9.04
```

The ornament size has grown to about `9` units, and in fact that is more or less the equilibrium of the model as it stands – somewhere around `8` or `9`. Ornament size has grown despite the direct negative fitness consequences of a large ornament, which manifest in two ways in this model: (1) a lower probability of being chosen as a first parent by SLiM, and (2) a lower probability of being chosen as a candidate mate by the callback code. Those two negative effects are more than compensated for by the positive effect: that once chosen as a candidate mate, one is more likely to be accepted by the probabilistic draw. However, the log function used in the preference equation means that the rewards of large ornament size extend only so far; as with peacocks, presumably, having *too* large a tail simply renders one clumsy and ridiculous.

3. Optimizing the callback

You may have noticed that this model runs fairly slowly. This is partly because `mateChoice()` callbacks just tend to be slow; the Eidos code here is doing a lot of work, and it's called for every mating event in every generation, so there's a certain amount of unavoidable overhead. But it's also because the callback does a lot of redundant work that could be optimized away. Let's try a SLiMgui feature we haven't seen yet: profiling. Recycle the model, and then click the small stopwatch button that overlaps the Play button; this is the Profile button. You might let the model run all the way to the end, to get a total time for its execution – it should take only a minute or so – but you can click the Profile button again to stop profiling after 1000 generations or so if you get impatient. Either way, a profile report will open in a new window, summarizing the performance of the model in many different ways. The SLiM manual has a detailed discussion of this feature, so we won't go into it in depth here. For our purposes, the salient points are that, in a test run I just did, (a) the "elapsed CPU time inside SLiM core" is 54.48 seconds, (b) the offspring generation life cycle stage is listed as taking 97.66% of total runtime, and (c) `mateChoice()` callbacks are listed as taking 95.34% of total runtime among callback types. The callback is clearly slow. And the profile report also provides some information about why, with this heat-map-colored summary of its code:

```
mateChoice() {
   fixedMuts = sum(sim.substitutions.mutationType == m2);
   for (attempt in 1:5)
   {
      mate = sample(p1.individuals, 1, T, weights);
      osize = fixedMuts * 2 + mate.countOfMutationsOfType(m2);
      
      if (runif(1) < log(osize + 1) * 0.1 + attempt * 0.1)
         return mate;
   }
   return float(0);
}
```

The most time-consuming part of the callback, according to the heat colors, is the call to `sample()` to choose a candidate mate, and this is difficult to optimize since it is rather intrinsic to the functioning of the callback; each first parent needs to choose candidate mates and examine them.

The rest of the callback can be optimized, though, if we realize that the calculations are likely to be quite redundant. Each first parent doing a mate search will examine up to five candidate mates, so it is probably quite likely that each individual will be looked at as a candidate more than once (especially early on, when ornament sizes are small and many candidates are rejected). It is also noteworthy that each time a candidate mate is looked at here, the calculation of its ornament size and the resulting acceptance probability is done in a one-off, non-vectorized fashion; since Eidos is an interpreted language, that means there is a great deal of language overhead involved in doing the same operations over and over, one at a time, rather than doing them once using vectorized operations.

How to improve this? The key – and this is a strategy that can apply to other callback types too, especially `fitness()` callbacks – is to move calculations out of the callback entirely, caching their results beforehand. This lets us calculate the ornament size of each individual just once, and it also lets us do the calculations with vectorization. The `1:10001 early()` event was already calculating ornament sizes, in fact – but only every thousand generations – so just a bit of reorganization is needed to make it do that work in every generation and cache the results:

```
1:10001 early() {
   fixedMuts = sum(sim.substitutions.mutationType == m2);
   inds = p1.individuals;
   osize = fixedMuts * 2 + inds.countOfMutationsOfType(m2);
   inds.tagF = log(osize + 1) * 0.1;

   if (sim.generation % 1000 == 1)
      catn(sim.generation + ": Mean ornament size == " + mean(osize));
}
```

Notice the vectorized calculations; `osize` is a vector of ornament sizes for each individual, and the code then calculates a vector of attractiveness values with `log()`, as the `mateChoice()` callback did. The next step is conceptually quite new, however: it does a *vectorized property assignment* into a property named `tagF`, on the vector of individuals. The `tagF` property is a user-defined property that can be used to store any sort of model state you wish; SLiM does not use it in any way, so it is entirely free for your own use. In this model we will use it to store attractiveness values for every individual. The `tagF` property is a singleton property of type `float` (there is a similar property, `tag`, that is a singleton of type `integer`). Since it is a singleton property, Eidos is able to understand that the assignment of a vector of values into it must be a vectorized assignment, and so it will assign one element of the calculated value into the `tagF` property of each corresponding individual. The result is that each individual's `tagF` property contains that individual's attractiveness value.

Since `early()` events occur just before offspring generation, in WF models, these cached `tagF` values are all precalculated and available for use by the callback. Modifying it to use them is very simple:

```
mateChoice() {
   for (attempt in 1:5)
   {
      mate = sample(p1.individuals, 1, T, weights);

      if (runif(1) < mate.tagF + attempt * 0.1)
         return mate;
   }
   return float(0);
}
```

This is the same logic as before, but it looks up the attractiveness of each candidate mate using mate.tagF, rather than calculating it from scratch.

If we run a new profile on this model, we can see that the model is much faster – the "elapsed CPU time inside SLiM core" is now 38.28 seconds, so we've cut about 30% off the runtime. The script block profiles are revealing as well: even though the 1:10001 early() event is now pre-calculating all of the attractiveness values in each generation, it it still takes only 2.82% of total runtime (in my test run) – a negligible amount. That is the power of vectorization: vectorizing those calculations has reduced their overhead almost to zero. More than 75% of total runtime is still spent in the mateChoice() callback, which now looks like this:

```
mateChoice() {
   for (attempt in 1:5)
   {
      mate = sample(p1.individuals, 1, T, weights);

      if (runif(1) < mate.tagF + attempt * 0.1)
         return mate;
   }
   return float(0);
}
```

The sample() call has gotten even darker orange, indicating that we are spending an even larger fraction of total time in it; this is to be expected, since the time spent doing other things has been reduced. Most of the rest of the time is taken calculating the final attractiveness value for the candidate mate, and then using runif() to decide whether the candidate is accepted or not. It isn't obvious how that work could easily be optimized further, since it depends upon the attempt number, so our optimization is done. (If one were really strapped for speed, though, one could probably make further progress by pre-drawing candidate males in bulk – 100 at a time, say – in an attempt to vectorize the sample() call. At that point it might be easier and better to recast the model as a nonWF model anyway, which makes vectorization of mating operations much easier.)

4. Returning mating weights instead of mates

The model we just developed uses a mateChoice() callback that explicitly chooses one individual as a mate and returns that individual to SLiM. That is often a good design because it's simple and often fairly efficient, but there is an alternative: returning a vector of mating weights. A mateChoice() callback that follows this design should return a vector of float values that is equal in length to the number of individuals in the source subpopulation from which parents are being chosen. Those weights are then used by SLiM to choose the mate. This follows SLiM's default mating behavior, in which the probability of each individual being chosen as a mate is proportional to that individual's relative fitness; but here the mateChoice() callback can override that default mating weights vector with its own weights.

As it happens, our optimized design above makes this very easy, because the attractiveness values of every individual have been pre-calculated. The mateChoice() implementation *seems* trivial:

```
s1 mateChoice() {
   return weights * p1.individuals.tagF;
}
```

First of all, the s1 designation at the beginning gives this script block a symbolic name, just like p1 for a subpopulation, m1 for a mutation type, etc.; script blocks can also be named so that they can be referred to later. We will use this symbolic name in a moment.

This callback returns a vector of weights that is the result of multiplying SLiM's default weights (passed in to the callback through the pseudo-parameter weights, as we saw previously) with the cached attractiveness values in p1.individuals.tagF. Both of these vectors are of the same size – one value per individual – so this works nicely. Notice that there is no need to ensure that the weights vector sums to 1.0; we are not supplying probabilities for each individual, but merely relative weights, and SLiM will normalize them for us. We use weights as well as the attractiveness values because we want the deleterious effects of large ornament size to be included in the final mating weights – just as we did in the original design, where the sample() call uses weights to draw a candidate mate. Notice that we have gotten rid of the "sequential mate search" aspect of the model; that doesn't work so well when returning a weights vector, for obvious reasons.

If we recycle and run this model, we hit a snag: it gives us an error:

```
ERROR (Population::EvolveSubpopulation): failed to generate child after 1
million attempts; terminating to avoid infinite loop.
```

The new mateChoice() callback is somehow failing us. Why? A moment of thought reveals the answer: in the very first generation there are no ornament mutations in any individual, and so the calculated attractiveness of every individual is zero. The vector returned by the mateChoice() callback is therefore a vector of all zeros. SLiM interprets that as meaning the same as a return of float(0), as we used before: the first parent cannot find any suitable mate, so pick a new first parent. So SLiM goes back, picks a new first parent, and gets the same result again – all zero weights. After a million attempts, SLiM gives up to avoid hanging in an infinite loop.

We would like to fix this problem with a simple rule: if every individual has an attractiveness of zero, then just use SLiM's default mating weights unmodified by attractiveness. Implementing this just requires the following lines in the 1:10001 early() event, right after inds.tagF is assigned:

```
if (sum(inds.tagF) == 0.0)
   s1.active = 0;
```

If the attractiveness values sum to zero, we deactivate the mateChoice() callback – referring to it by its symbolic name s1 – by setting its active property to 0. The active property is a property of script blocks (which are objects of class SLiMEidosBlock). If active is 0, the script block is deactivated; it simply does not get used or called, as if it didn't even exist. Any non-zero value means that the script block is active and will be used. The active value of every script block gets reset to 1 at the beginning of every generation, so script blocks are always active in every generation by default, and must be deactivated in any generation in which they are not needed. (Of course blocks also only run in the generations they are declared for, and so forth; active is an *additional* restriction on the applicability of callbacks.)

With this tweak, the model runs quite nicely, and ornament sizes increase to an equilibrium of about 16 in this formulation of mate choice without sequential search. Ornament size increases much more rapidly in this version of the model, because if one individual has a particularly large ornament then *every other individual* sees that large ornament and is likely to choose the attractive individual as a mate, whereas with sequential mate search only five candidates would be examined, maximum, before making a choice, and so an outlier with a large ornament would be unlikely to even be seen.

EXERCISE: This might take a little while, and is not essential to finish, so please do your **BEEP!** now, and then explore this exercise in the time remaining. Feel free to look at the solution!

Profile the new model in SLiMgui. The mateChoice() callback's one line is taking quite a bit of time, because it is having to do a vectorized property access to re-fetch p1.individuals.tagF every time the callback is called. Look at the documentation for the setValue() and getValue() methods on SLiMSim, and try caching the whole attractiveness vector using sim.setValue("a", ...). How much faster is the model then?

5. BEEP! With extra time, look at SLiM manual section 11.1, or section 20.5 on profiling.