SLiM Workshop Series #15: non-Wright–Fisher Models

1. Start a new nonWF model in SLiMgui

We're going to start with SLiMgui's default nonWF model. To get that, press shift-command-N, or go to SLiMgui's File menu and choose **New (nonWF)**. You should have a model window that begins with a comment that says:

```
// set up a simple neutral nonWF simulation
```

2. The `initialize()` callback

Recycle and step. As discussed in the lecture, the `initialize()` callback is quite similar to that for a WF model, except for three things:

```
initialize() {
   initializeSLiMModelType("nonWF");
   defineConstant("K", 500);  // carrying capacity

   // neutral mutations, which are allowed to fix
   initializeMutationType("m1", 0.5, "f", 0.0);
   m1.convertToSubstitution = T;

   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 99999);
   initializeMutationRate(1e-7);
   initializeRecombinationRate(1e-8);
}
```

First, there is the call to `initializeSLiMModelType("nonWF")`. This just tells SLiM that we're making a nonWF model, which switches over some default behaviors (a `reproduction()` callback will now be required, for example).

Second, we use `defineConstant()` to define a constant, `K`, to be the carrying capacity of the model, `500`. SLiM doesn't know anything about `K`; we never give `K` to SLiM as a parameter, as we would in a WF model with `setSubpopulationSize()`. We just use it to implement density-dependent selection.

Third, we set the `convertToSubstitution` property of `m1` to `T`. In WF models this is `T` by default (and sometimes has to be set to `F`, when modeling things like epistasis or QTLs, as we have seen). In nonWF models the policy is reversed; `F` is the default. This is because nonWF models use absolute fitness, not relative fitness, and so the *only* mutations that can safely be removed from the simulation when they reach fixation are neutral mutations that have no effect on fitness at all, even indirectly – like the `m1` mutation type here. It's important to use `T` when you can, for speed.

3. The `reproduction()` callback

Now click Step once more, and the initial subpopulation is created by this event:

```
// create an initial population of 10 individuals
1 early() {
   sim.addSubpop("p1", 10);
}
```

Notice that the initial subpopulation we create is only size `10`; we are far below our carrying capacity of `500`. The population will grow naturally, since births will outnumber deaths until we reach `K`.

Let's step once more. Notice that `p1` has grown to `20` individuals; this is because no individuals died (everybody's fitness is `1.0` or greater, and mortality is based upon fitness), and every individual had exactly one offspring as a first parent (while perhaps also acting as a second parent for one or more offspring, hermaphroditically). The births are a result of the `reproduction()` callback:

```
// each individual reproduces itself once
reproduction() {
   subpop.addCrossed(individual, p1.sampleIndividuals(1));
}
```

This callback is called once for each individual in the model; each individual gets a turn to be the *focal individual*, passed in as the pseudo-parameter `individual`. The callback is expected to do whatever is appropriate to have the focal individual reproduce as a first parent.

Here, we call a method named `addCrossed()` on `subpop` (a pseudo-parameter representing the focal individual's subpopulation). The `addCrossed()` method creates one offspring by biparental sexual reproduction, and adds it to the target subpopulation (`subpop`). We pass it the two parents: the focal individual, `individual`, and a mate chosen randomly from `p1` using `sampleIndividuals()`. We haven't seen `sampleIndividuals()` before; it is essentially the same as calling `sample()` on `p1.individuals`, but it's faster, and provides various options for narrowing down which individuals are candidates to be drawn – very useful for mate choice.

There are a few things to note about this. One is the flexibility; in a WF model SLiM runs the offspring generation process, and so to alter the default behavior you have to make a `mateChoice()` callback, but in nonWF models you are in charge of the process. Your `reproduction()` callback can select any mate, from any subpopulation, based upon any criteria at all, and then generate any number of offspring with that mate. Everything is under the control of the script.

The second thing to note is that we called `addCrossed()` to do a biparental sexual mating. There are other methods – `addCloned()`, `addSelfed()`, and `addRecombinant()` – for other types of offspring generation. In nonWF models these choices are up to your script, on an individual-level basis, allowing each individual to reproduce in its own manner, based upon genetics and environment.

4. Population regulation

Click the Step button a couple more times, and you will see the population size grow to `40`, then `80`, `160`, and `320`. It is doubling in a deterministic fashion because (a) there is no mortality, because everybody's fitness is over `1.0`, and (b) the `reproduction()` callback's behavior is deterministic. With the next click of Step, however, the population size is no longer deterministic, because density-dependent population regulation kicks in courtesy of this event:

```
// provide density-dependent selection
early() {
   p1.fitnessScaling = K / p1.individualCount;
}
```

In the first generation there were only `10` individuals, at that point, the `fitnessScaling` value this calculates would be `500 / 10`, or `50` – quite high! Of course if you wanted to put a cap on the maximum benefit that low population density could confer, that would be easy. In any case, that value of `50` is put, not into `fitnessScaling` on each individual, but into `p1.fitnessScaling`. This is a subpopulation-wide fitness scaling that is applied to every individual in the subpopulation – it is multiplied together with all of the other fitness effects in the model. Nothing else affects fitness in this model, so every individual will end up with an absolute fitness of `50`, and so none will die.

Now let's consider the generation that we just stepped over. The population size began at `320`. Looking at the nonWF generation cycle on the SLiM reference sheet, the first thing to happen is offspring generation: this is deterministic, in this model, and so now we have `640` individuals. Then `early()` events happen, and the code above calculates a value of `500 / 640`, or `0.78125`. That is placed into the `p1.fitnessScaling` property. Next, fitness is recalculated for every individual. Since there are no other factors influencing fitness in this model, every individual gets an absolute fitness of `0.78125`. Next is the viability/survival stage. In this stage, survival occurs with a probability equal to fitness; so every individual has a survival probability of 78.125%. SLiM handles this with random number draws, and so the actual result is unpredictable, but it will probably be somewhere in the ballpark of `500`. We have therefore enforced our chosen carrying capacity, without ever telling SLiM "make the subpopulation have `500` individuals".

Now click Play and let the model run to the end. You can see that the population size varies stochastically, but stays in the vicinity of `500`, guided by this density-dependence equation. It is important to understand, however, that this equation is a *choice*. You can model population regulation however you like – perhaps there is a limited pool of resources, and individuals have to forage and compete with each other to gain enough resources to survive. You could implement that behavior in script, and set the `fitnessScaling` property of each survivor to `1.0` and that of the rest to `0.0`; in that case you are explicitly telling SLiM which individuals die during the viability/survival stage. Or you could model predation, or territorial behavior, or whatever mechanism makes sense for your biological system. But you need to implement *some* kind of population regulation, or the individuals in your model will simply never die, and the population size will grow exponentially.

5. Overlapping generations and age structure

When it is running, this model might not look very different from the equivalent WF model, apart from a little stochasticity in the population size; but in fact it is profoundly different, because it is a model of overlapping generations, which WF models don't support at all. Let's add a little output code to see what's going on with age in this model:

```
late() {
   inds = p1.individuals;
   cat(sim.generation + ": " + size(inds));
   catn(" (" + max(inds.age) + ", " + mean(inds.age) + ")");
}
```

The `age` property of `Individual` is the individual's age, obviously. When recycled and run, the model produces output like this:

```
1: 10 (0, 0)
2: 20 (1, 0.5)
3: 40 (2, 0.75)
4: 80 (3, 0.875)
5: 160 (4, 0.9375)
6: 320 (5, 0.96875)
7: 492 (6, 1.00407)
8: 478 (7, 1.02092)
9: 475 (8, 0.930526)
10: 519 (7, 0.957611)
...
```

In generation `1` all individuals are new, and so their age is `0`, representing juveniles. (You could set the `age` property to other values if you wished.) At the end of generation `2`, half the individuals are age `0` (new offspring) and half are age `1` (the parental generation), so the maximum age is `1` and the mean age is `0.5`. You can follow this logic forward. Once mortality kicks in, in generation `7`, things start to level out, and an equilibrium age structure with a mean age that fluctuates around `1.0` emerges. This makes sense; if the population size is at equilibrium, then there is one birth for each death, and since the model is unbiased in which individuals die, that leads to a pyramidal age structure with a mean of `1.0` (approximately, stochastically).

After running to the end, open the Eidos console and look at the equilibrium age structure that emerged from the model by typing:

```
> sort(p1.individuals.age)
...
> for (a in 0:15) cat(sum(p1.individuals.age == a) + " "); catn()
0 264 127 66 32 13 2 4 1 1 0 0 0 0 0 0
```

Do type this yourself in the Eidos console, and make sure you understand how it works. There are no juveniles because we're running this in SLiMgui's console after the generation has ended; the ages of individuals get incremented at the end of each generation, so every juvenile has just graduated to age `1`. Apart from that, notice that (with a little stochastic noise) there are half as many individuals of age *a*+1 as there are of age *a*, until the numbers get small enough that they trail off to zero. If you ponder how this model works, you will see why that is.

6. Making mortality age-dependent

Suppose we want to modify the model so that there is a maximum age of `2`; all individuals older than `2` should die. That is quite easy to implement with a modification to our `early()` event:

```
early() {
   inds = p1.individuals;
   inds.fitnessScaling = ifelse(inds.age <= 2, 1.0, 0.0);

   p1.fitnessScaling = K / p1.individualCount;
}
```

A quick recycle/run looks promising:

```
1: 10 (0, 0)
2: 20 (1, 0.5)
3: 40 (2, 0.75)
4: 70 (2, 0.571429)
5: 130 (2, 0.615385)
6: 240 (2, 0.625)
7: 440 (2, 0.613636)
8: 470 (2, 0.621277)
9: 467 (2, 0.657388)
10: 435 (2, 0.581609)
...
```

So now we have age-based mortality! One could easily extend this to model specific mortality rates for each age, using a "life table", but that's a trivial extension so we won't belabor the point here (there's an example in the manual). The only problem, as is hinted at in the snippet of output above, is that the equilibrium population size is now below `K`. This makes perfect sense; SLiM is doing exactly what we told it to do. The reason is that we are calculating density-dependent selection based upon `p1.individualCount`, but that count includes some number of individuals that we have already marked for death. Suppose, as we calculated above, the correct survival rate to arrive at `500` individuals is 78.125%. Now suppose that we mark fifty individuals for death – but we still apply that same 78.125% survival rate to the remaining population. We will end up, on average, at less than 500 individuals; we needed to use a higher survival rate, based upon the number of individuals currently projected to survive. We can modify the code accordingly, with this change:

```
early() {
   inds = p1.individuals;
   inds.fitnessScaling = ifelse(inds.age <= 2, 1.0, 0.0);

   p1.fitnessScaling = K / sum(inds.fitnessScaling);
}
```

Given the design of the code, `sum(inds.fitnessScaling)` is the number of individuals expected to survive the age-based mortality stage, so we should use that to govern the density-dependence.

In the more general case, however, these adjustments can be quite difficult to perform, and often it is a good idea to simply let go of an attachment to making the model achieve a particular number (or tune the model heuristically). Population size in a nonWF model is *emergent*: it is a consequence of the balance between birth and death, not an externally imposed parameter, in the end. If a model has segregating beneficial and deleterious mutations influencing fitness, for example, that will cause the population size to rise or fall concomitantly, and it *should*, since fitness in nonWF models is absolute fitness.

7. Adding in beneficial mutations

To illustrate the point just made – that non-neutral mutations will influence fitness and therefore equilibrium population size – let's add beneficial mutations to the model. Add a new beneficial mutation type:

```
initializeMutationType("m2", 0.5, "f", 0.1);
```

And then give every individual in the initial population a beneficial mutation, by adding this right after the `addSubpop()` call:

```
p1.genomes.addNewDrawnMutation(m2, 10000);
```

Recycle and step a couple of times. You should get output like this:

```
1: 10 (0, 0)
2: 20 (1, 0.5)
3: 40 (2, 0.75)
4: 70 (2, 0.571429)
5: 130 (2, 0.615385)
...
```

Notice that the initial population growth rate is the same: `10`, `20`, `40`. The `m2` mutation increases the fitness of every individual, but since fitness values during this initial growth period were already over `1.0`, it makes no difference; survival probability can't be higher than 100%. It also makes no difference to the bite taken by age-dependent mortality starting in generation `4`; we are setting the fitness of individuals older than `2` to `0.0`, so no multiplicative beneficial fitness effect is going to allow them to survive. Now run the model, and observe that the equilibrium population size is ~550. This is because once density-dependent selection kicks in, everybody's fitness is below `1.0`, and so the increase to absolute fitness from the `m2` mutation is meaningful. If you wanted to override this emergent behavior, you might add frequency-dependence or something – but you should first consider whether it is actually a problem, or whether it is in fact more realistic.

EXERCISE: Change `m2`'s selection coefficient to `-0.1` to make it deleterious. Recycle and run. Make sure you understand the reasons for the initial growth rate and equilibrium population size.

8. Scripting the whole population's reproduction

Let's work on a new problem: monogamous mating. Suppose that, in any given breeding season, we want individuals to pair off monogamously and generate a small litter together, of a somewhat random size. By "monogamously" we mean not only that a given first parent mates with only one chosen second parent, but that that second parent also does not mate with any other individual, so our code as it stands now is *not* monogamous mating. This requires some sort of overall, top-down organization of the mating process. We could imagine marking individuals as having paired off, using the `tag` field perhaps, and then allowing only untagged individuals to be chosen. There's a simpler design, however, and it shows a useful twist on the design of `reproduction()` callbacks.

We said earlier that a reproduction callback is called once for each focal individual, and is expected to generate offspring from that individual as first parent. This is nowhere written in stone, however; SLiM doesn't actually care what you do, that is just the structure of the calls out to your callback that SLiM makes. You can do whatever you want.

With that in mind, let's replace the `reproduction()` callback with this code:

```
reproduction() {
   // randomize the order of p1.individuals
   parents = sample(p1.individuals, p1.individualCount);

   // draw monogamous pairs and generate litters
   for (i in seq(0, p1.individualCount - 2, by=2))
   {
      parent1 = parents[i];
      parent2 = parents[i + 1];
      litterSize = rpois(1, 2.7);

      for (j in seqLen(litterSize))
         p1.addCrossed(parent1, parent2);
   }

   // disable this callback for this generation
   self.active = 0;
}
```

The overall strategy of this callback is to handle the reproduction for all of the individuals in `p1`, in a single callout, so that it can organize them into monogamous pairs easily. When it has done that, it ends by setting `self.active` to `0`. We have seen the `active` flag of script blocks once before; when set to `0`, it deactivates the block within the current generation, preventing SLiM from calling it any more. The variable `self` is implicitly defined by SLiM for all script blocks; it refers to the currently executing script block. So `self` here refers to the `reproduction()` callback itself, and setting `self.active` to `0` deactivates it for the rest of the generation.

Now let's look at the reproduction code. First it draws a sample from `p1.individuals` of size `p1.individualCount`. Sampling without replacement is the default for `sample()`, so this makes `parents` a shuffled copy of `p1.individuals`. Then we step through pairs of individuals in `parents`; each pair will mate monogamously. (Look up the online help on `seq()` if you're not sure how this works.) Next we draw the size of the current pair's litter; this is a draw from a Poisson distribution with mean `2.7`, arbitrarily. The `for` loop with `j` loops to generate the offspring one by one by calling `addCrossed()`. By the way, the `seqLen()` function is a nice convenience function for generating a sequence of a given length; `seqLen(x)` is a lot like `0:(x-1)`, except that `seqLen(0)` produces the result desired – a zero-length vector – whereas `0:(0-1)` generates the sequence `0 -1`. This is important to understand; use `seqLen()` when appropriate to avoid nasty bugs.

So, we have a model of monogamous mating (within breeding season).

EXERCISE: Convert the model to have two subpopulations, `p1` and `p2`. Get the `reproduction()` callback working (you will want to enclose its code in a loop like `for (s in c(p1,p2))` and then reproduce subpopulation `s` inside the loop). Also get the density-dependence code working (use the same approach). Once that's done, add migration using the `takeMigrants()` method, which you can read about in the online help. Try moving 1% of individuals from `p1` to `p2` in each generation, and vice versa, in an `early()` event just before density-dependence. Select individuals to move using `sample()`. Make sure you never move an individual from A to B and back in a single generation.

9. BEEP! With extra time, look at any model in SLiM manual chapter 16 (nonWF recipes). (The spatial models might be opaque since we haven't discussed continuous space yet.)