SLiM Workshop Series #18: Continuous Space I

1. Start with recipe 15.1 in SLiMgui

The typing for this model would be tedious, so let's start with recipe 15.1 in SLiMgui. Go to the **File** menu, choose **Open Recipe**, then go to the Chapter 15 recipes and choose recipe 15.1. You should see this model:

```
initialize() {
   initializeSLiMOptions(dimensionality="xy");
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 99999);
   initializeRecombinationRate(1e-8);
}
1 late() {
   sim.addSubpop("p1", 500);

   // initial positions are random in ([0,1], [0,1])
   p1.individuals.x = runif(p1.individualCount);
   p1.individuals.y = runif(p1.individualCount);
}
modifyChild() {
   // draw a child position near the first parent, within bounds
   do child.x = parent1.x + rnorm(1, 0, 0.02);
   while ((child.x < 0.0) | (child.x > 1.0));

   do child.y = parent1.y + rnorm(1, 0, 0.02);
   while ((child.y < 0.0) | (child.y > 1.0));

   return T;
}
2000 late() { sim.outputFixedMutations(); }
```

We've discussed the main features of this model in the lecture already, but to review quickly: (a) the call to `initializeSLiMOptions()` to set the *dimensionality* of the model to `"xy"`; (b) the code in the `1 late()` event to set the initial spatial positions for individuals; and (c) the `modifyChild()` callback, which generates spatial positions for proposed offspring. Remember that `do`-`while` loops execute the statement between `do` and `while` repeatedly (but at least once), until the condition for the loop is `F`. In this case, a new spatial coordinate is generated by the body of the loop until that coordinate is inside the spatial bounds of the landscape (assumed, here, to be [0, 1]).

Change the final generation to 20000 so we get more runtime, and the recycle and step twice. Now the population has been created with spatial positions that are drawn from a uniform distribution for `x` and `y`. Now press play, and notice that the uniform distribution quickly coalesces into a strong pattern of spatial clustering. This is because the position of each offspring is based upon the position of its parent, with a small deviation due to dispersal. Why exactly that leads to clustering is a bit subtle – it is actually related to the process of coalescence of lineages under neutral drift.

EXERCISE: To see more clearly how coalescence leads to clustering, change the standard deviation of the dispersal in the `modifyChild()` callback from `0.02` to `0.002`, then recycle and run. You can see that a large number of independent clusters gets reduced, one by one, to a single cluster as drift leads clusters to go extinct by chance. The individuals within a given cluster all trace their ancestry back to the same maternal ancestor. Once the model coalesces down to a single maternal ancestor for the whole population, one cluster is left.

EXERCISE: Change the dispersal kernel width back to `0.02`. Then make the position of the offspring depend not just on `parent1`, but on `parent2` as well – specifically, upon the midpoint between the two. You could use an expression like `(parent1.x + parent2.x) / 2`, or you could use `mean(c(parent1.x, parent2.x))`. Do this for `y` as well, then recycle and run. What do you think is the cause of the behavior you observe now? Hint: it is no longer related to coalescence. The point of these exercises is to realize that spatial models will often show artifacts that depend strongly upon the specifics of the model; care must be taken to understand the cause of spatial dynamics.

Revert back to using just `parent1`'s position for now.

2. Initial positions

At the moment, the model sets up initial spatial positions with these two lines:

```
p1.individuals.x = runif(p1.individualCount);
p1.individuals.y = runif(p1.individualCount);
```

There is a more idiomatic way to achieve this. Change that code to this:

```
p1.individuals.setSpatialPosition(p1.pointUniform(p1.individualCount));
```

Let's unpack this from the inside out. First we get the number of individuals in `p1`, with `p1.individualCount`. Then we call `p1.pointUniform()` to generate that many points, drawn from a uniform distribution for each spatial dimension; if `p1` has `500` individuals, as it does here, the result is a vector of length `1000`, with alternating `x` and `y` values. Finally, we call `setSpatialPosition()` on `p1.individuals` to set those positions. The `setSpatialPosition()` method is vectorized in a smart way: it sees that the vector it is passed is exactly twice the length of the target vector `p1.individuals`, and it knows that the model has dimensionality `"xy"` (and so there are two values per spatial point), so it assigns each pair of (`x`, `y`) values to the `x` and `y` properties of the corresponding individual. This is just like a vectorized property assignment, except that instead of being a 1-to-1 elementwise assignment, it is a *k*-to-1 assignment, where *k* is the dimensionality.

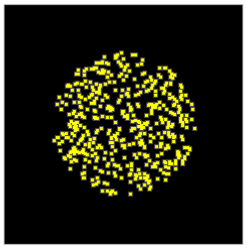
Why is this better? The main reason is that the `pointUniform()` method correctly accounts for the spatial bounds of the subpopulation. The default spatial bounds span the interval [0, 1] in each dimension, and `runif()` generates values in [0, 1] by default, so that worked out nicely. However, the spatial bounds of the subpopulation can be changed to whatever you wish, using the `setSpatialBounds()` method of `Subpopulation`. The `pointUniform()` method will always generate points uniformly distributed within the subpopulation's bounds, whatever they might be.

Of course, one will not always want initial positions to be uniform. Let's distribute the initial individuals within a circle instead. To do this, we will loop over the individuals and set each one's position individually, since it's hard to come up with a vectorized way to do this:

```
for (ind in p1.individuals) {
   do {
      p = p1.pointUniform(1);
      d = sqrt(sum((p - 0.5)^2));
   } while (d > 0.3);
   ind.setSpatialPosition(p);
}
```

We create a new uniform point, `p`, and then calculate the distance from `p` to the center at (`0.5`, `0.5`) – doing it with vector operations just to be fancy and keep you on your toes. Make sure you understand how that expression works. The do-while loop does that repeatedly until the distance is less than or equal to `0.3`; then the loop terminates and the position `p` is set on the individual.

If you now recycle and step twice, you should see that the initial distribution of individuals lies entirely within a circle of radius `0.3` in the center of the landscape:



The resemblance to Pac-Man is striking.

EXERCISE: Modify the code so that the initial distribution of individuals lies within a square, rather than a circle, of the same size – i.e., with `x` and `y` coordinates within [0.2, 0.8]. If you want a hint as to the easy way to do this, read on.

Hint: don't change the loop condition `(d > 0.3)` at all. Instead, change the formula for `d`. You will want to use the `abs()` and `max()` functions.

3. Boundary conditions

The `modifyChild()` callback we're using assumes that the bounds of the landscape span [0, 1], as the initial position code used to before we switched to `pointUniform()`. Let's fix that problem by upgrading the callback to a better algorithm:

```
modifyChild() {
   // Reprising boundary conditions
   do pos = parent1.spatialPosition + rnorm(2, 0, 0.02);
   while (!p1.pointInBounds(pos));
   child.setSpatialPosition(pos);

   return T;
}
```

This does essentially the same thing as the previous code, but it will work for any spatial bounds, and it's considerably easier to read. The spatial position of `parent1` is fetched (a vector of length `2`, given the `"xy"` dimensionality of the model), and a vector of two normal deviates is added to represent dispersal in `x` and `y`. The `do-while` loop calls `p1.pointInBounds()` to bounds-check the new point, and loops until it is within bounds. Finally, the position is set on the proposed child.

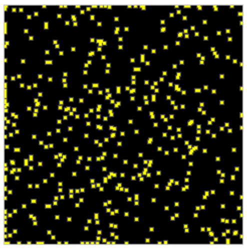
As the comment says, this implements *reprising* boundary conditions, meaning that points outside of bounds are "reprised" – revisited, drawn again – until within bounds. Several other boundary conditions are supported by SLiM, but first let's increase the dispersal kernel's width to `0.1`, recycle, and run. Look closely at the distribution of points within the space. It looks pretty uniform, averaged over time, right? That's because it is – reprising boundary conditions do not exhibit any bias toward or away from the edges.

Now change the callback to this code:

```
modifyChild() {
   // Stopping boundary conditions
   pos = parent1.spatialPosition + rnorm(2, 0, 0.1);
   child.setSpatialPosition(p1.pointStopped(pos));

   return T;
}
```

This implements *stopping* boundary conditions, which mean that if a spatial coordinate is outside of bounds, it is forced within bounds. There is no need for a loop in this case; whatever point is drawn gets forced within bounds and used. Recycle and run this, and observe closely. You should see that there are a lot of individuals kind of "stuck to the walls", because of the stopping boundaries. If you pause the model in mid-run, you'll see this quite clearly:

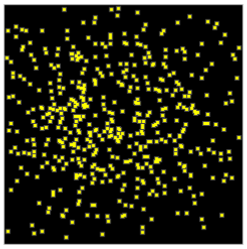


Let's try one more. Change the `modifyChild()` callback code to this:

```
modifyChild() {
   // Absorbing boundary conditions
   pos = parent1.spatialPosition + rnorm(2, 0, 0.1);
   if (!p1.pointInBounds(pos))
      return F;

   child.setSpatialPosition(pos);
   return T;
}
```

This implements *absorbing* boundary conditions. Here, if the generated position is out of bounds that is lethal: the proposed child is rejected by the callback, by returning `F`. Otherwise the position is used and the proposed child is accepted. Recycle and run this model and observe it closely, and you may notice that the spatial distribution is thinner toward the edges, as here:



This is because living near the edges actually entails a fitness cost: some of your offspring get absorbed by the boundary, and so your expected fertility is lower than if you lived at the center. This is, in a sense, the opposite of stopping boundary conditions, which lead to elevated density near the edges. Both boundary conditions are biased at the edges, unlike the reprising boundaries we started with.

SLiM also supports *reflecting* and *periodic* boundary conditions, as we will see later. Periodic boundary conditions are particularly useful and interesting, especially for more theoretical models, but are a bit more complex, too – we won't get into them for now.

4. Spatial competition

Let's shift gears completely. Now that we've developed an understanding of continuous space, positions, and boundaries, let's actually use the spatial positions for something in the model. Specifically, let's introduce *spatial competition*, in which individuals feel competition from other nearby individuals that decreases their fitness. We will implement this using a built-in SLiM facility called an *interaction type*. An interaction type (represented by the Eidos class `InteractionType`) describes a particular interaction between individuals – how the interaction strength varies with distance, what the maximum distance is over which the interaction can be felt, and so forth. Once an interaction type has been defined, it can be *evaluated* to assess the interactions present at one moment in time, and then it can be *queried* to get information about those interactions that the model can use. Under the hood, `InteractionType` uses a sophisticated and highly optimized spatial query engine, based upon advanced data structures called *k*-d trees and sparse arrays, for speed and memory efficiency. To the user, however, it presents a simple interface that makes complex spatial models easy to build.

To set up spatial competition, first add a declaration of the interaction type, at the end of the `initialize()` callback:

```
// Set up an interaction for spatial competition
initializeInteractionType(1, "xy", reciprocal=T, maxDistance=0.3);
i1.setInteractionFunction("n", 3.0, 0.1);
```

The first line tells SLiM that interaction type `i1` (`1`) involves both the `x` and `y` spatial dimensions (`"xy"`), is reciprocal (i.e., the interaction strength from individual A to individual B is the same as from B to A), and has a maximum distance of `0.3`. For any focal individual, then, interactions will be felt from other individuals within a radius of `0.3` of the focal individual.

The second line tells SLiM the shape of the interaction kernel or *interaction function*, with a call to the method `setInteractionFunction()`. An interaction function is essentially a formula describing the strength of interaction as a function of distance. The parameters here set the kernel to be a Gaussian function (`"n"` for "normal", because `"g"` is used for "gamma") with a maximum strength of `3.0` (between two individuals at the same spatial position) and a standard deviation (often called the kernel's "width") of `0.1`.

Next, add this `1: late()` event after the `1 late()` event that initializes `p1`:

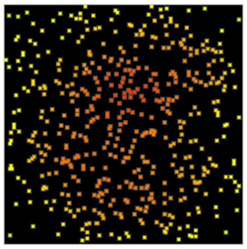
```
1: late() {
   i1.evaluate();

   inds = p1.individuals;
   totalStrengths = i1.totalOfNeighborStrengths(inds);
   inds.fitnessScaling = 1.1 - totalStrengths / inds.size();
}
```

This event begins by evaluating `i1`. This basically takes a snapshot of the current state of the model, by copying the spatial position of every individual. This copied information will be used by `i1` to create the spatial data structures that it uses to answer spatial queries – the *k*-d tree and sparse array mentioned before – without having to worry about the spatial positions of individuals changing out from under it, which would invalidate its work. The details of this aren't really important; suffice to say that you must call `evaluate()` before you try to query an interaction. The cached evaluation will be invalidated by SLiM whenever new offspring are generated, or (in nonWF models) whenever the viability/selection phase occurs.

Next we get the subpopulation's individuals as `inds`, just for shorthand and efficiency. Then we call `i1.totalOfNeighborStrengths()` with `inds`. If called with a single individual A, this method would do a spatial search for all individuals within the maximum interaction distance of A, calculate the interaction strength exerted upon A by each of those nearby individuals (by pushing the distance through the interaction function), and total up those interactions to get a single total interaction strength felt by A. Passed `inds`, this method does all of that work for all of the individuals in the subpopulation, and returns a vector that has the total interaction strength felt by *each* individual. This is a tremendous amount of work, especially in the maximum interaction distance is large; the smaller you can make the maximum interaction distance, the faster your spatial queries will be. Anyway, the last line divides those total interaction strengths by the number of individuals (getting a mean interaction strength), and subtracts that from `1.1` to get a fitness value that is placed into `fitnessScaling`. Note that these computations are all vectorized; with these lines we have just calculated the fitness effect of competition for all individuals in the model.

If you recycle and run, you should see something like this:



The model is no longer neutral, and so individuals are being colored according to fitness. Individuals near the edge have higher fitness because there are fewer other individuals near them – both because of the thinning due to the absorbing boundary conditions, and because no individuals exist in the empty space beyond the edges of the landscape. Switch back to stopping boundaries (use the code above), and you will see that fitness is close to uniform across the landscape; the increased density at the edges due to the stopping boundaries more or less counterbalances the zero density beyond the edge. A better solution to achieve unbiased fitness across the landscape is to use periodic boundaries, as we will explore later.

5. Spatial mate choice

To finish off this model, let's add spatial mate choice: individuals will choose a mate from among their three nearest neighbors. First, add another interaction type:

```
initializeInteractionType(2, "xy", reciprocal=T, maxDistance=0.1);
```

No interaction function is needed, because we won't use this to calculate interaction strengths, just to find nearby individuals.

Next, add an evaluation of `i2` just before offspring generation:

```
early() { i2.evaluate(); }
```

Finally, add a `mateChoice()` callback:

```
1: mateChoice() {
   neighbors = i2.nearestNeighbors(individual, 3);
   return (size(neighbors) ? sample(neighbors, 1) else float(0));
}
```

This uses a new spatial query, `nearestNeighbors()`, to find the three nearest neighbors of the callback's focal individual. Only individuals within the maximum interaction distance are eligible, so `neighbors` could be empty; in that case `float(0)` is returned, telling SLiM that no mate could be found, otherwise a mate is chosen at random from `neighbors` and returned.

Decrease the dispersal kernel's width to `0.01`, to encourage clustering, and then recycle and run. You should see a very dynamic pattern of strong clustering. Notice that neutral mutations typically take a very long time to fix in this model, because it is not panmictic any more; reproductive isolation due to spatial distance is a strong force in this model.

EXERCISE: Play around with various parameters of the model. Do you notice a difference if you set the nearest-neighbor search to `1`, or to `100`? What if you reduce the dispersal kernel width from `0.01` to `0.001`? Try a population size of `2000` instead of `500`. Experiment with different boundary conditions to see their effect with different parameter values. It's a fun model – experiment with it.

6. BEEP! With extra time, look at SLiM manual section 16.10, a nonWF spatial model.