# SLiM Workshop Series #19: Continuous Space II

## 1. Start with recipe 15.5 in SLiMgui

Recipe 15.5 provides a starting point that is similar to where we left off: it's a 2D continuous-space model with spatial competition (using `fitnessScaling`), nearest-neighbor mate choice (implemented with a `mateChoice()` callback), and juvenile dispersal. Open recipe 15.5 from the **Open Recipe** submenu of SLiMgui's **File** menu. The model is fairly long, so it won't be reproduced here, but it should be familiar from the previous exercise.

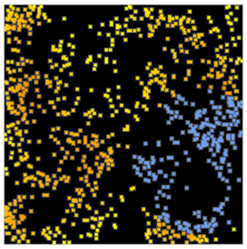
**Increase** the population size from `500` to `1000`; adjust the initial position setting accordingly.

## 2. Add occasional natural disasters

Let's add occasional natural disasters that wipe out a part of the landscape, killing all of the individuals within a given radius of the event epicenter. This is quite easy to do. Let's start by testing the code we intend to use. Add these lines to the end of the `1: late()` event:

```
// occasional natural disasters
if (runif(1) < 0.1) {
   epicenter = p1.pointUniform();
   d = i1.distanceToPoint(inds, epicenter);
   affected = inds[d < 0.3];
   affected.color = "cornflowerblue";
}
```

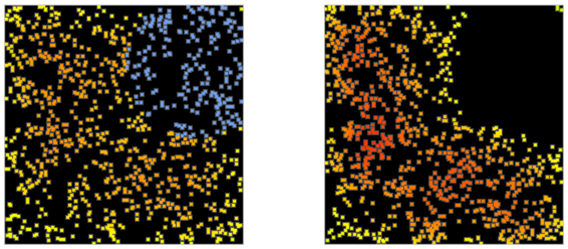
In 10% of generations, this code chooses a disaster epicenter at random with `pointUniform()`. It then uses `i1`, which conveniently has just been evaluated for competition, to measure the distances from the epicenter to every individual (this could be done with script instead, using the Pythagorean theorem, easily enough, but we might as well let `InteractionType` do it for us since it knows how). Then we select the individuals within a radius of `0.3` of the epicenter, and color them blue. If we step forward until we get a disaster, the model looks like this:



Looks good! Let's kill them off afterwards, by setting their `fitnessScaling` to `0.0` in the next line:

```
affected.fitnessScaling = 0.0;
```

Now we see individuals colored blue in one generation, and then a deathly void in the next:



This is the effect we wanted, but it has revealed a deep problem with our model: it doesn't handle population density in a realistic way. Compare the two frames above. The density within the void has gone down (to zero) in the second frame, but the density across the rest of the landscape has gone *up* markedly at the same time. This is because it is a WF model, with a set population size of `1000`; if the density goes down in one area, it *has* to go up elsewhere. We would like the population size to simply drop when a disaster occurs, and then rise gradually as the void is re-colonized.

This is one example of a general phenomenon: because WF models have a fixed population size, they really struggle to behave realistically. The same would be true if a beneficial mutation were introduced at one point, for example: you would probably like the population density to be higher within the range of the spread of that mutation, reflecting lower mortality or higher fecundity, whereas areas that the mutation has not yet spread to should remain at the same density as before. With a WF model, unfortunately, the density will actually drop in areas outside the range of spread; since the density goes up inside the range of spread, it *has* to go down elsewhere.

These sorts of considerations, and others related to other restrictions of the WF model, mean that realistic spatial models should probably almost always be nonWF models.

## 3. A nonWF spatial model

Let's adapt our model to the nonWF framework. Start by adding the nonWF declaration at the top of the `initialize()` callback, with new definitions of `K` and S:

```
initializeSLiMModelType("nonWF");
defineConstant("K", 1000);
defineConstant("S", 0.1);
```

Here `K` is not exactly a carrying capacity; rather, it is what has been called a "carrying-capacity density", which tells us what the carrying capacity of the model would be if individuals were distributed uniformly across the landscape (and without other factors besides density-dependence influencing fitness, as discussed before). Our disasters will cause the model to drop below `K`.

The constant `S` will be used as a spatial competition kernel width. In the declaration of interaction type `i1`, **change** the maximum distance to `S*3`, and **change** the interaction function width for `i1` from `0.1` to `S`. While you're at it, **change** `i1`'s maximum strength from `3.0` to `1.0`. The reason for these changes will become clear below. (I've put these changes in **bold** because people miss them!)

We want the neutral mutations in the model to substitute when fixed, which they won't by default as a nonWF model, so add this line to the `initialize()` callback:

```
m1.convertToSubstitution = T;
```

Next we need to fix reproduction. Remove the `mateChoice()` callback; those are not supported in nonWF models at all, since the `reproduction()` callback is responsible for choosing mates and generating offspring. Add the following `reproduction()` callback instead, after `initialize()`:

```
reproduction() {
   mate = i2.nearestNeighbors(individual, 3);
   if (!mate.size())
      return;
   mate = sample(mate, 1);

   for (i in seqLen(rpois(1, 0.1)))
   {
      pos = individual.spatialPosition + rnorm(2, 0, 0.02);
      offspring = subpop.addCrossed(individual, mate);
      offspring.setSpatialPosition(p1.pointReflected(pos));
   }
}
```

As the `mateChoice()` callback did, this begins by finding the three nearest neighbors of the focal individual. If no neighbors are found within the maximum interaction distance, the individual doesn't mate, so the callback returns. Otherwise, we draw one mate from the vector. Next, we draw a number of offspring from a Poisson distribution with a mean of just 0.1; we will model highly overlapping generations, for variety. We use `seqLen()` to a sequence of length equal to the litter size, which we loop over. (Remember that `seqLen()` produces a zero-length vector if given a length of zero, as we want here.)

For each offspring that we're generating, we generate a spatial position based upon the first parent's position with a bit of random dispersal, just as we did in the `modifyChild()` callback before. We've switched to reflecting boundaries, using `pointReflected()`, for no particular reason. This code replaces the `modifyChild()` callback, in fact, so that can be removed. In general, `modifyChild()` callbacks that only modify proposed children (rather than sometimes rejecting them) can usually be implemented directly in the `reproduction()` callback in nonWF models.

Next, the `1 late()` and `1: late()` events should both be changed to `early()` events. Because of the different generation cycle in nonWF models, we want these events to run after offspring generation and before fitness recalculation (the generation cycle comparison in the SLiM reference sheet may help with visualizing this). This is better for the `addSubpop()` call because then the new subpopulation will have fitness values calculated before it reproduces; if we called `addSubpop()` in a `late()` event, the next generation's offspring generation would occur without fitness values ever having been computed, a small inaccuracy. The `1: late()` event should be a `1: early()` event because it sets `fitnessScaling` values that should take effect immediately; the model really wouldn't work properly at all if those `fitnessScaling` values were set immediately *after* fitness recalculation.

The only problem with that rearrangement is the call to `i2.evaluate()`. We want that to happen in a `late()` event, so that mating interactions are evaluated after survival/viability, just before offspring generation in the following generation. Remove the call from the `1: early()` event, and add this:

```
1: late() {
   i2.evaluate();
}
```

Problem solved. There's one more fix to be made: the code doesn't actually enforce `K` anywhere. There is competition in the model, but it is not scaled to produce the carrying-capacity density we want. Replace the line:

```
inds.fitnessScaling = 1.1 - competition / size(inds);
```

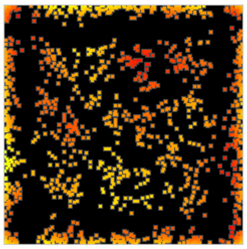
with these lines:

```
competition = (competition + 1) / (2 * PI * S^2);
inds.fitnessScaling = K / competition;
```

Instead of a relative fitness effect based on the strength of competition, we now compute an absolute fitness effect based on the local population density (as represented by the strength of competition) compared to the desired carrying-capacity density. The `+ 1` in the formula counts the focal individual itself – it is one of the individuals that count toward local population density, but it exerts no competitive interaction toward itself. The division by `(2 * PI * S^2)` basically rescales by the density of the (Gaussian) competition kernel; the SLiM manual has a long discussion of this technical point, but for our purposes here it can just be assumed. Note that these calculations are all vectorized, as usual. Given that rescaling, the ratio `K / competition` is the ratio between the desired carrying-capacity density and the actual local population density; it can be used for density-dependent selection, analogously to density-dependence in non-spatial nonWF models.

Let's turn off natural disasters temporarily. Select the lines that impose natural disasters, and then press command-/ (forward slash). SLiMgui will comment out the selected lines.

Now recycle and run. If you made all of the above changes correctly, you should see something like this:



This seems very odd; and the population size at equilibrium seems to be around `1450` or so, as well, not `K`. What's going on?

**4. Periodic boundaries**

What's going on is that the individuals really like hanging out at the edge of the landscape, because they feel less competition there – the empty space outside the bounds of the landscape counts as zero density, letting the individuals pack up at higher-than-uniform density right around the edge. This high density around the edge has the knock-on effect of causing a band of low density just inside the edge area; nobody in the interior wants to be near the edge individuals because their density is so high. The echoes of the high edge density probably continue all the way to the center, although they're harder to see.

This is also the reason for the equilibrium population size exceeding `K`: the extra individuals that can be squeezed in at the edge account for the difference. In a sense, the population density of the model at equilibrium really is `K` – if you consider the landscape as extending outward with a sort of shelf of diminishing carrying-capacity density, following the shape of the competition kernel as it extends past the edge.

Anyway, suffice to say that SLiM is doing exactly what we told it to do, but perhaps we're not happy with the result. We could fix this in a variety of ways. We could decrease the target carrying-capacity density for positions near the edge of the landscape (basically we want to calculate the integral of the portion of the competition kernel that lies inside bounds), but the math for that is pretty messy. We could make the landscape very large, so that these edge effects would make little difference to the average behavior of the model (maybe). We could increase the dispersal distance enough that these patterns would be erased by stochastic long-range dispersal (but then the model loses much of its spatiality and moves toward panmixia). There is a better solution, for many models: use periodic boundaries that wrap around at the edges to create a uniform space with no edge effects.

The change is trivial. First, declare the periodicity by modifying the `initializeSLiMOptions()` call:

```
initializeSLiMOptions(dimensionality="xy", periodicity="xy");
```

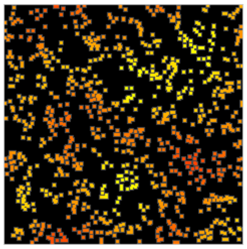
SLiM will now consider both the x and y spatial dimensions to be periodic, wrapping around at the edges to form a landscape that is, topologically speaking, a torus.

Next, change the call to `pointReflected()` into a call to `pointPeriodic()` instead:

```
offspring.setSpatialPosition(p1.pointPeriodic(pos));
```

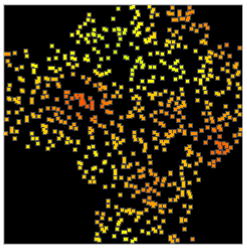
The `pointPeriodic()` method takes a spatial point that might lie outside of bounds, and brings it into bounds by wrapping it around the edges of the periodic space.

That is all that needs to be done; all spatial positions and interactions will now occur on a landscape with periodic coordinates. Recycle and run, and you should see something like this:



Much better! The equilibrium population size now varies around `1000`, as desired, too, so it looks like our problems are all solved.

Select the "natural disaster" lines and press command-/ again to uncomment them. Recycle, and then step carefully so you can observe individual natural disasters as they occur. You should notice that they now wrap around the edges of the periodic space. For example, this is a single disaster:



The blue coloring is no longer visible; since the viability/selection generation stage follows directly on the heels of fitness recalculation, in nonWF models, SLiMgui never has an opportunity to show us blue individuals before they are dead and gone. Now click Play. The model might hang on by its fingernails for a little while, but it will soon go extinct; the natural disasters are frequent enough that they overwhelm the (rather low) fecundity of the model.

To compensate, change the probability of a natural disaster from `0.1` to `0.01`, then recycle and run again. Now you should see some very nice spatial dynamics. When disasters occur, the void will slowly be recolonized from the edges as new offspring are generated by the individuals on the edge. The individuals on the edges of voids generally have higher fitness, colored yellow or even green as a result of the low density in their vicinity. The population size drops naturally when a disaster occurs, and then rises as recolonization proceeds, too, as desired.

## 5. Habitability governed by a spatial map

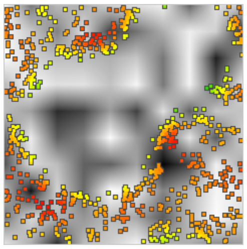
Let's extend the model in one more way. Rather than the landscape being homogenous, we'd like it to vary in its habitability: some areas should support high local density, others low or zero local density. This really means varying `K` over space, instead of using one universal value of `K`.

To achieve this, add the following lines after `p1` is added and initial positions are set:

```
mapValues = matrix(sqrt(runif(100, 0, 1)), nrow=10);
p1.defineSpatialMap("h", "xy", mapValues, interpolate=T,
   valueRange=c(0.0, 1.0), colors=c("black", "white"));
```

We draw `100` random values from a uniform distribution, and take their square root to bias them upward towards `1.0` (making the landscape somewhat more habitable). We convert that vector of `100` values into a 10×10 matrix with `matrix()`. Then we call the `defineSpatialMap()` method on `p1` to define a map using these values; note that spatial maps are specific to particular subpopulations (if you have more than one subpopulation, each on a different landscape). We name the map `"h"` for future reference, and tell SLiM that its grid of values goes across both the *x* and *y* dimensions. We pass `mapValues` in for the values. We request (bilinear) interpolation of the map values between the defined grid points. The last two parameters are solely for SLiMgui's benefit: we tell SLiMgui that for purposes of display, map values should be considered to range from `0.0` to `1.0` (even though the actual range of values might be narrower), and that `0.0` should be displayed as black and `1.0` as white. The colors for intermediate values will be interpolated, but you can supply more colors and color points if you wish to produce a more complex color scheme.

If you recycle and run, you can see the generated map behind the individuals:



The only problem at this point is that the map values themselves are not periodic – they ought to match at the left and right edges, and at the top and bottom edges, following the periodic boundaries of the landscape, but they don't. That's trivial to fix, but we won't do so here. It would probably also be nice if the map values exhibited spatial autocorrelation, rather than each value being independent and random; that is an exercise for the reader.

Next, we want the map to actually influence habitability. This is very easy; replace the line:

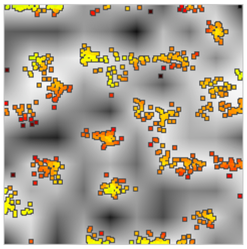
```
inds.fitnessScaling = K / competition;
```

with the lines:

```
K_local = K * p1.spatialMapValue("h", inds.spatialPosition);
inds.fitnessScaling = K_local / competition;
```

This calls `spatialMapValue()` to look up the map values at every individual's location; note that this call is vectorized, interpreting `inds.spatialPosition` correctly as a vector of points and returning the map value for each point. These map values range from `0.0` to `1.0`; we multiply by `K` to get a local carrying-capacity density, `K_local`, for each individual. Then we just use the `K_local` for each individual, instead of `K`, to calculate the fitness effect of local density for each individual.

If we recycle and run, we see clusters inhabiting the more clement parts of the landscape:



They mostly stay in the most habitable locations, but the connectivity of the landscape is usually sufficiently high that they can spread along high-habitability ridgelines to recolonize vacant areas.

**EXERCISE:** Play around with this model and make sure you understand it fully. It's the most complex and sophisticated SLiM model we're going to build in this workshop, so enjoy it!

**6. BEEP!** With extra time, look at SLiM manual section 15.10, a very different use of spatial maps involving loading in empirical raster map data. The matrix of map values in that recipe is provided by the Eidos class `Image`, which reads in a PNG image from disk – an easy way to import map data.