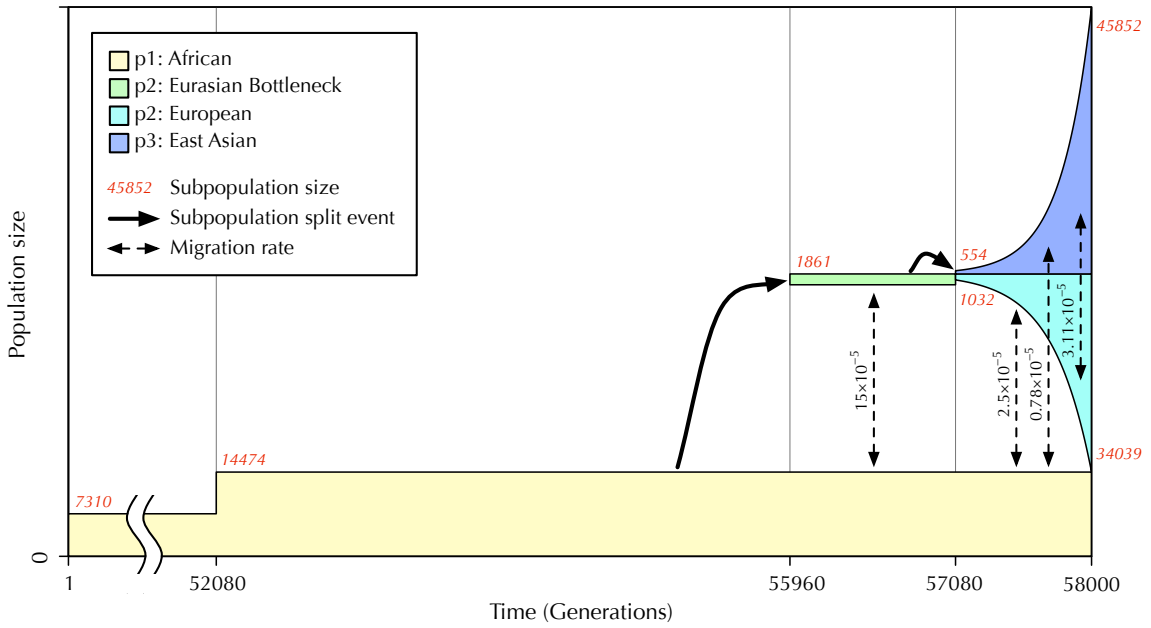
SLiM Workshop Series #5: Demography

1. Start a new model in SLiMgui

Run SLiMgui, and close any existing model windows. Press command-N to make a new model window. Select the script in the script pane, and press backspace to delete it; we will start from scratch in this exercise. To begin with, we will be making the Gravel model of human evolution:



This models the origin of *Homo sapiens* in Africa and subsequent expansion into Eurasia, with low rates of migration, and with exponential population growth near the present. Note that the SLiM manual has an improved version of this recipe, with revised times and sizes that better reflect the original model by Gravel et al.; this worksheet has *not* been updated to reflect those changes.

2. Set up the genetic structure

Add this `initialize()` block to the script pane (don't forget to use code completion to speed up your typing – it's a good habit to develop):

```
initialize() {
   initializeMutationRate(2.36e-8);
   initializeMutationType("m1", 0.5, "f", 0.0);
   initializeGenomicElementType("g1", m1, 1.0);
   initializeGenomicElement(g1, 0, 10000);
   initializeRecombinationRate(1e-8);
}
```

This sets up a chromosome of length 10001 that will undergo neutral mutations of type `m1` in a single genomic element. The mutation rate is an estimate from empirical data.

3. Model the African demography

The African subpopulation demography in the Gravel model is fairly straightforward, as shown in yellow in the diagram above: an initial subpopulation of size 7310, an expansion to size 14474 in

generation 52080 and a steady-state continuation to the present, generation 58000. Note that these are *effective* population sizes given random panmictic mating in a Wright–Fisher model; the actual population sizes in Africa over human history would have been considerably larger. In any case, these sizes are large enough that the model would take a couple of minutes to run, which is inconvenient, so here we will scale all subpopulation sizes down by 10× (the unscaled Gravel model can be found in the SLiM manual). Note that this is not a rigorous rescaling of the model (which would also involve rescaling the mutation rate, recombination rate, selection coefficients, and time; see the manual); we are just artificially clipping the model for experimental purposes.

Add these events to the script (you can skip the comments if you wish, of course):

```
// Create the ancestral African population
1 { sim.addSubpop("p1", 731); }

// Expand the African population to 14474
// This occurs 148000 years (5920 generations) ago
52080 { p1.setSubpopulationSize(1447); }

// Generation 58000 is the present.  Output and terminate.
58000 late() {
   p1.outputSample(216); // YRI phase 3 sample of size 108
}
```

When you run this, you will see neutral drift, an eventual population expansion, more neutral drift, and then, at the end, the output of the genetic information from a sample of 216 genomes (108 individuals, although we do not ensure here that the sample consists of pairs of genomes from individuals). The sampling here follows the paper by Gravel et al. (2011) from which this model is derived; they compare their model to empirical data sampled from the three subpopulations.

Note that the `setSubpopulationSize()` call is a *future request*: it tells SLiM that `p1` should generate `1447` individuals the next time it generates offspring. This call is done in an `early()` event in generation `52080`, just before offspring generation, so the subpopulation will increase in size later in the same generation, but the effect is not instantaneous – if the script examined `p1` immediately after that method call, it would be unchanged in size. If the same event in generation `52080` were to set a different size first – `10`, say – and then set the size to `1447` as it does now, the first future request would effectively be overridden; the subpopulation would never actually go down to size `10`.

4. Model the non-African colonization event

Next we can add the split of non-Africans from Africans, a colonization event in generation `55960`. Again the subpopulation size is scaled down by a factor of 10×. Since the event declarations specify the generation(s) in which the events will run, it doesn't really matter where this new event is inserted, but for conceptual clarity, insert it in chronological order, above the `58000 late()` event:

```
// Split non-Africans from Africans and set up migration between them
// This occurs 51000 years (2040 generations) ago
55960 {
   sim.addSubpopSplit("p2", 186, p1);
   p1.setMigrationRates(p2, 15e-5);
   p2.setMigrationRates(p1, 15e-5);
}
```

The `addSubpopSplit()` call adds a new subpopulation, `p2`, which derives from `p1`, with an initial size of `186` individuals. Specifically, it immediately creates `p2` (not as a future request), and populates it with individuals that are clonal copies of individuals from `p1`, randomly drawn. This may seem odd, but consider what will happen during the next offspring generation phase: every new offspring will be the result of mating between individuals that are, in effect, drawn randomly from a subsample of `p1`. This generally suffices to produce a "founder effect" as desired.

This event also sets the migration rate between `p1` and `p2` (the same rate in both directions, as it happens), using `setMigrationRates()`.

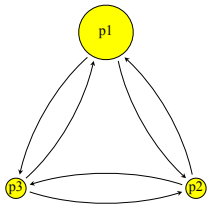
5. Model the European / East Asian split

If you look back to the Gravel model diagram, the next thing to model is the split of the Eurasian subpopulation into separate European and East Asian subpopulations. This is accomplished with another call to `addSubpopSplit()` to create `p3`, followed immediately by a downscaling of `p2`. Migration rates are then set up between all subpopulations, following the model's specification. Again the subpopulation sizes are reduced by a factor of 10:

```
// Split p2 into European and East Asian subpopulations
// This occurs 23000 years (920 generations) ago
57080 {
   sim.addSubpopSplit("p3", 55, p2);
   p2.setSubpopulationSize(103);  // reduce European size

   // Set migration rates for the rest of the simulation
   p1.setMigrationRates(c(p2, p3), c(2.5e-5, 0.78e-5));
   p2.setMigrationRates(c(p1, p3), c(2.5e-5, 3.11e-5));
   p3.setMigrationRates(c(p1, p2), c(0.78e-5, 3.11e-5));
}
```

At this point, let's Recycle the model and open the Graph Population Visualization window, as we did in a previous exercise. Then enter `57080` into the Generation textfield and press return, to run the model forward to the beginning of generation `57080`. As the model runs, you will see p1 created at the beginning, it will grow larger in generation `52080`, and in generation `55960` `p2` will be created. Now use the Step button to step over generation `52080`, executing the new event we have just added. You should see the creation of `p3`, and the new migration rates that were established:



Keep the population visualization window open; we'll continue observing it.

6. Model the European / East Asian population expansion

The next stage of the Gravel model involves exponential expansion of the European and East Asian subpopulations. This is done by calling `setSubpopulationSize()` for `p2` and `p3` in each generation from `57080` on. The size for each generation is calculated by an exponential function:

```
// Set up exponential growth in Europe and East Asia
// Where N(0) is the base subpopulation size and t = gen - 57080:
//    N(Europe) should be int(round(N(0) * e^(0.0038*t)))
//    N(East Asia) should be int(round(N(0) * e^(0.0048*t)))
57080:58000 {
   t = sim.generation - 57080;
   p2_size = round(103 * exp(0.0038 * t));
   p3_size = round(55 * exp(0.0048 * t));

   p2.setSubpopulationSize(asInteger(p2_size));
   p3.setSubpopulationSize(asInteger(p3_size));
}
```

Note that the result from the exponential formulas has type `float`, even though `round()` rounds it off, so we have to convert to the `integer` argument type that `setSubpopulationSize()` expects, using `asInteger()`. Now try recycling and playing the model; the population visualization graph should now display the final expansion quite nicely. Note that the circles representing `p2` and `p3` won't increase in size after they exceed 10,000 individuals; there is an upper bound for the display, to try to keep things fitting within the display window.

7. Model output

We need to add sample output from `p2` and `p3`; put this at the end of the `58000 late()` event:

```
p2.outputSample(198); // CEU phase 3 sample of size 99
p3.outputSample(206); // CHB phase 3 sample of size 103
```

Recycle and play the model one more time. The output generated by the model will contain three sections of sample output, marked off by three header lines that start with "`#OUT`". So far we've avoided talking about SLiM's output; let's take a moment to look at it now. For one run of this model, the first sample output looks like this:

```
#OUT: 58000 SS p1 216
Mutations:
0 19804 m1 4434 0 0.5 p1 54572 216
2 20454 m1 4907 0 0.5 p1 55537 173
3 20470 m1 9185 0 0.5 p1 55560 173
1 23484 m1 1870 0 0.5 p1 57606 33
4 29102 m1 4195 0 0.5 p1 57955 3
Genomes:
p1:0 A 0
p1:1 A 1 0 2 3
p1:2 A 0 2 3
p1:3 A 0 2 3
...
```

The header line says that this section was produced in `58000`, that it is a subpopulation sample generated by `outputSample()`, that `p1` was sampled, and that the sample size was `216` genomes.

Then comes a subsection headed `Mutations:` that contains information about each mutation contained by the sampled genomes. Specifically, each line provides: (1) the index of the mutation within this output (not necessarily in sorted order), (2) the unique identifier for this mutation in SLiM (available through the `id` property of `Mutation`, and constant throughout a given run), (3) the mutation's type (always `m1` here), (4) the base position where this mutation is located, (5) the selection coefficient of the mutation (always zero in this neutral model), (6) the dominance coefficient (always `0.5` here), (7) the subpopulation in which the mutation originated (this happens to be `p1` for the mutations in this sample), (8) the generation in which the mutation originated (this tends to be recent since old mutations tend to have fixed already), and finally (9) the prevalence of the mutation in the sample: the number of sampled genomes that contain the mutation. All 216 sampled genomes contain the first mutation, so it is probably close to fixing; others are lower.

Finally, there is a `Genomes:` section that describes each sampled genome. The first genome line begins with the information that it belongs to `p1`, and is the `0`th genome in the sample. It is marked as an autosome with `A` (when simulating `X` or `Y` chromosomes this might be different). Finally, the line provides a list of indices for the mutations this genome contains; these indices refer back to the indices given in the `Mutations:` section.

Take a very quick skim down through the remainder of the output. The samples from `p2` and `p3` probably contain more mutations that are at very low prevalence. This likely reflects the recent expansion and large size of these subpopulations; they have created a lot of genetic diversity close to the present that has not yet had a chance to drift to higher frequency.

That's probably all we're going to say about SLiM's standard output formats in this workshop, since it's a rather boring topic. The formats produced by other SLiM output methods are similar, and all are documented thoroughly in the reference section of the manual.

8. Merging subpopulations, removing subpopulations

Let's now go beyond the Gravel model to explore some other demographic phenomena that can be modeled in SLiM. Let's extrapolate into a science fiction future. First of all, if generation `58000` is the present, let's model globalization as being, in effect, a merging together of previously genetically distinct subpopulations into a single panmictic world population in generation `58001`:

```
58001 early() {
   sim.addSubpop("p4", 90000);
p4.setMigrationRates(c(p1, p2, p3), c(0.3, 0.4, 0.3));
   }
      58001 late() {
         c(p1,p2,p3).setSubpopulationSize(0);
}
```

You are encouraged to mess up the indentation of the lines, as shown above, just so we can take a momentary detour. Given the messed-up indentation above, press this button in SLiMgui:

This "prettyprints" your script, fixing the indentation of the lines to be standard. Nice, eh?

The `early()` event creates a new subpopulation, `p4`, with a size approximately equal to the total size of `p1`, `p2`, and `p3` at the "present time" in the model. SLiM will, in response to this call, immediately create `p4` and populate it with new individuals with empty genomes, as it did when we created `p1` initially. However, we then set up migration rates from `p1`, `p2`, and `p3` that total `1.0`, so in the offspring generation phase of generation `58001` all of those new empty individuals will be discarded and `p4` will be refilled with "migrants" generated from parents in `p1`, `p2`, and `p3`. They are migrants only in the sense that they are placed by SLiM in `p4` even though their parents are in `p1`, `p2`, or `p3`; `p4` is the new world population.

So when this event finishes executing, we have (a) a new subpopulation `p4`, filled with new empty individuals, and (b) migration rates set for `p4` that will cause its offspring to come entirely from the other subpopulations. This is an `early()` event, so the next thing that happens in generation `58001` is offspring generation (see the WF model generation cycle diagram on the SLiM reference sheet). That puts into effect what we have requested: the *offspring* generation of `p4` is generated from scratch using offspring generated by the *parental* generation of `p1`, `p2`, and `p3`. This will also generate an offspring generation for `p1`, `p2`, and `p3` as usual, but those subpopulations conceptually no longer exist once they have generated `p4`, and we will remove them momentarily.

If you look at the life cycle diagram, the next thing that happens is "Removal of fixed mutations" (ok, doesn't matter to us), and then "Offspring become parents". It is at this point, in WF models, that the *offspring* generation of the subpopulations becomes the *parental* generation; so it is at this point that the new empty individuals in `p4` are discarded in favor of the new "migrant" offspring. (The offspring generations of `p1`, `p2`, and `p3` will also become the parental generation.)

Next comes execution of `late()` events, which will execute the event defined above and set the size of `p1`, `p2`, and `p3` to zero. Note how we do that: we construct a vector that contains all three, using `c()`, and then make a single vectorized method call on that vector. That's a nice quick way to "broadcast" a method call to multiple target objects. Setting a size of zero on a subpopulation is an exception to the "future request" rule; it takes effect immediately, and the effect it has is to not only set the size of the subpopulation to zero, but to actually remove it from the subpopulation

altogether. (In nonWF models it is legal to have a subpopulation with a size of zero, but in WF models it is not.) This, then, is how you remove subpopulations in WF models.

Let's recycle and run again. You will see that at the end of generation `58000` we have a single subpopulation, `p4`; `p1`, `p2`, and `p3` are gone. We can see in the chromosome view that `p4` does contain some genetic diversity; but is it from the original subpopulations, or is it just new mutations that occurred in the final generation, perhaps? To find out, open the Eidos console and type:

```
> sim.mutations.originGeneration
54685 56034 52508 57968 57995 57978 57333 57980 57732 57415 57937 58001
57467 57992 57717 57388 57998 57340 58001 57288 57999 57976 ...
```

This shows that the mutations originated in past generations; we have successfully merged the subpopulations. We could also look at their subpopulation of origin:

```
> sim.mutations.subpopID
1 1 1 3 3 3 1 3 3 2 3 3 3 3 3 1 3 2 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 ...
```

None of the mutations have an origin subpopulation of 4, because all of the individuals in `p4` were generated by parents in either `p1`, `p2`, or `p3`, in this generation. In WF models, furthermore, individuals are always generated by parents drawn from the same subpopulation, so none of the individuals in `p4` are crosses between subpopulations; such crosses will occur in the *next* generation.

9. Mating systems

The other thing covered by the last lecture was different mating systems. So far we have actually been modeling *hermaphrodites*, where any given individual can mate with any other individual because individuals can produce both sperm and eggs. There are lots of other mating systems, of course, including reproduction by *cloning* (mitotic replication of parental genomes, with no recombination), reproduction by *selfing* (self-fertilization, which is sexual reproduction via gametes produced, with recombination, by the same individual), and sexual models in which separate sexes are modeled, a sex ratio is defined, and biparental mating occurs between opposite sexes (as opposed to the hermaphroditic models we have been using thus far). Let's explore some of those by extending our Gravel model into a science-fiction future.

EXERCISE: In generation `58010`, humanity establishes a Mars colony. Model this with a new subpopulation, `p5`, that splits from `p4` with an initial size of `10`. The colony struggles for a little while, but beginning in generation `58012` have its size double in each generation until generation `58020`, when it stabilizes. The twist: on Mars, sexual reproduction is difficult (the spacesuits get in the way), so they reproduce clonally instead. You can use the call `p5.setCloningRate(1.0)` to put this into effect immediately after `p5` is created.

EXERCISE: In generation `58030`, terraforming of Mars is complete and they ditch the spacesuits. Set the cloning rate to `0.0` to reflect the new, happier Martian lifestyle. However, sexual reproduction had become unfashionable in the meantime, and so some Martians decide to reproduce by selfing rather than returning to biparental mating. In the same generation, introduce a selfing rate of `0.2` for `p5` with the `setSelfingRate()` method.

EXERCISE: Let's convert the whole model to model separate sexes instead of hermaphrodites, for greater realism. Add a new call, `initializeSex("A")`, to the `initialize()` callback (`"A"` says we are modeling autosomes, not sex chromosomes). You will find that the `setSelfingRate()` call now produces an error – only hermaphrodites can self – so remove it from the model. (This limitation could be overcome in a nonWF model, where you can model almost any mating system.)

10. BEEP! With extra time, you might read any topic in SLiM manual chapter 5 that interests you.