SLiM Workshop Series #13: Quantitative Traits

1. Start a new model in SLiMgui

Make a new SLiMgui model window, and adapt the default script to the following code:

```
initialize() {
   initializeMutationRate(1e-7);
   initializeMutationType("m1", 0.5, "f", 0.0);       // neutral
   initializeMutationType("m2", 0.5, "n", 0.0, 0.5);  // QTLs
   m2.convertToSubstitution = F;
   initializeGenomicElementType("g1", m1, 1);
   initializeGenomicElementType("g2", c(m1,m2), c(0.9,0.1));
   initializeGenomicElement(g1, 0, 20000);
   initializeGenomicElement(g2, 20001, 30000);
   initializeGenomicElement(g1, 30001, 99999);
   initializeRecombinationRate(1e-8);
}
fitness(m2) { return 1.0; }
1 { sim.addSubpop("p1", 500); }
20000 late() {
   catn("\nDID NOT REACH OPTIMUM");
   dumpFrequencies(sim.mutationsOfType(m2));
}
```

This is the starting point for our QTL-based model (don't try to run it yet, it's incomplete). Mutation type `m2` represents QTLs (quantitative trait loci) that will determine the phenotypes of individuals. The chromosome consists of a "gene" in the middle, from `20001` to `30000`, using genomic element type `g2`, within which 10% of mutations are QTL mutations (the rest being neutral `m1` mutations). Outside of that, the chromosome consists of genomic elements of type `g1`, which is entirely neutral. The model creates `p1`, runs for `20000` generations, dumps a little output, and ends.

Mutation type `m2` is defined as using a DFE of type `"n"`, which we haven't seen before. This draws selection coefficients from a normal distribution, and the next two parameters are the mean and standard deviation of that distribution – `0.0` and `0.5`, respectively. Fixation will not cause substitution for `m2` mutations, since `convertToSubstitution` is set to `F`; after all, QTL mutations continue to influence phenotype even after fixation. Finally, a `fitness(m2)` callback tells SLiM to ignore the selection coefficients of `m2` mutations and simply treat them as neutral. This is crucial when implementing QTL-based models in SLiM, because of the switcheroo we need to pull: `m2` mutations will, as far as SLiM is concerned, have no intrinsic fitness effect at all (i.e., will be neutral), and their selection coefficients will instead by used by the model's script as QTL effect sizes.

A major point we've glossed over is the call to `dumpFrequencies()`. We haven't seen that function before, and for a good reason: it doesn't exist. We will now define it:

```
function (void)dumpFrequencies(object<Mutation> muts) {
   if (muts.size())
   {
      freqs = sim.mutationFrequencies(NULL, muts);
      coeffs = muts.selectionCoeff;
      catn();
      print(cbind(freqs, coeffs));
   }
}
```

This is the first time we've created a user-defined function (add it anywhere – perhaps at the very top). The declaration begins with the keyword `function`, and then provides the function signature

for the new function, just as we have seen before for built-in Eidos functions. This declaration says that the function returns no value (`void`), but takes a parameter named `muts` that is a vector of type `object`, with class `Mutation`. Eidos uses the signature to type-check calls to the function at runtime, so there is no need for the function to check the type/class of `muts`; Eidos guarantees it.

The code of the function gets the frequencies of the mutations in `muts`, and fetches their selection coefficients (which are QTL effect sizes, in this model). It emits a blank line with `catn()` for padding, and then prints a matrix that results from binding the frequencies and the selection coefficients together as columns with `cbind()`. We haven't seen matrices in Eidos before, and we're not going to go into any detail about them here since we're only using them for purposes of output; the Eidos manual has a good introduction to them. For our purposes here, it is easier to show than to explain. When this model is run, typical output looks like this:

```
            [,0]       [,1]
[0,]       0.003   0.877338
[1,]           1  -0.541942
[2,]           1   0.270283
[3,]       0.047    0.38477
[4,]       0.151  -0.234286
```

As given to the `cbind()` function, the first column of the matrix is the frequency of an `m2` mutation, and the second column is that mutation's selection coefficient (i.e., QTL effect size). There is no particular pattern here; because of the `fitness(m2)` callback, `m2` mutations are effectively neutral, so they fix by chance and their selection coefficients are unused by the model.

2. Add calculation of phenotypic trait values and fitness effects

Now let's make the `m2` mutations actually act as QTL mutations, rather than just being neutral. To achieve this, we need to calculate individual phenotype from the additive effects of the QTL mutations in the individual's genomes, and then calculate the fitness effect for the individual as a function of the individual's phenotype. This is very simple, and is done with vectorized code so it is also efficient:

```
1: late() {
   inds = sim.subpopulations.individuals;
   phenotypes = inds.sumOfMutationsOfType(m2);
   inds.fitnessScaling = 1.0 + dnorm(phenotypes, 10.0, 5.0) * 10.0;
}
```

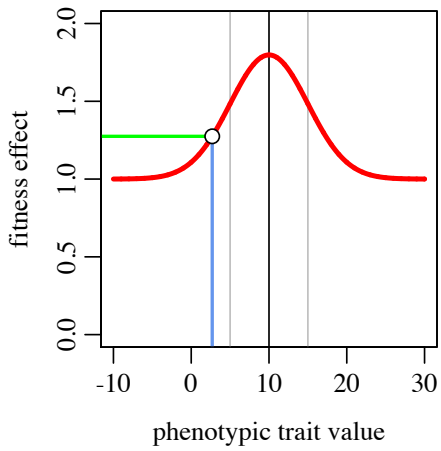
First, this code gets a vector of all individuals in the model, as `inds`, for brevity later on.

Next, it makes a vectorized method call to `sumOfMutationsOfType()`. This method is specially designed for use with QTL-based models. It searches through the target individual's genomes, looking for mutations of the given mutation type (`m2`, here), and keeps a running total of all of the selection coefficients for the mutations it finds. If those values really were selection coefficients, this would be a strange and pointless thing to do; but because they are actually QTL effect sizes, this calculates the additive effect of all of the individual's QTL mutations, and the result is the phenotype of the individual (or at least the additive genetic component of the phenotype). The result of the vectorized call, then, is a vector of phenotypic trait values corresponding to each individual, which is assigned into `phenotypes`.

The third line calculates fitness effects, given those phenotypic trait values. It uses the `dnorm()` function, which returns the probability density for a normal distribution at a given quantile; we're not really thinking of this function as a "normal distribution", though, but just as a Gaussian function. The quantiles are the phenotypic trait values; the mean of the normal distribution is the optimum of stabilizing selection on this phenotypic trait (`10.0` here); and the standard deviation of the normal distribution is often called the "width" of the fitness function (`5.0` here). The result is

multiplied by `10.0` (to produce the desired strength of selection), and then is added to `1.0` (making individuals that are infinitely far from the phenotypic optimum have a fitness of `1.0`). The result of these vectorized operations is a vector containing the fitness effect corresponding to each phenotypic trait value. A vectorized property assignment then places each fitness effect into the `fitnessScaling` property of the corresponding individual. When SLiM calculates fitness values for individuals in the model, these `fitnessScaling` values will be multiplied together with all other fitness effects in the model to produce final fitness values; so by assigning fitness effects into `fitnessScaling`, we are defining the connection between phenotype and fitness.

With this code, then, we have defined the fitness function for the model; but it may seem rather abstract at this point. In practice, the fitness function looks like this:



The red curve is the fitness function itself. The vertical black line at x=10 is the phenotypic optimum, the phenotypic trait value that produces the highest fitness, while the gray lines at x=5 and x=15 enclose one standard deviation, showing the so-called "width" of the fitness function. The minimum value produced by this fitness function is `1.0`, for phenotypes infinitely far from the optimum; this is not required, it's just the choice here. Finally, the blue line at phenotype 2.7 (blue line) is shown producing a fitness of ~1.275 (green line), according to the fitness function.

Fitness functions don't have to be based on a normal function, and indeed they don't have to be unimodal or symmetrical or anything else. They are just a mapping of phenotype to fitness. They are a mathematical expression of the nature of selection exerted by the environment: this phenotype tends to survive, this phenotype tends to die.

Finally, we need to track the mean phenotype. Add this to the end of the `1: late()` event:

```
// Output and check for termination
mean_phenotype = mean(phenotypes);

if (abs(mean_phenotype - 10.0) < 0.1)
{
   catn("\n" + sim.generation + ": REACHED OPTIMUM");
   catn("Final phenotype == " + mean_phenotype);
   dumpFrequencies(sim.mutationsOfType(m2));
   sim.simulationFinished();
}
else if (sim.generation % 100 == 0)
{
   catn(sim.generation + ": Mean phenotype == " + mean_phenotype);
}
```

With this code, in every generation the model calculates the mean phenotype and compares it to the optimum at `10.0`. If the optimum has been reached within a tolerance of `0.1`, as shown by a difference with an absolute value of less than `0.1`, then the code prints `"REACHED OPTIMUM"`, prints the mean phenotype reached at the end, calls `dumpFrequencies()` to print a summary of the QTLs in the model, and terminates. If the optimum has not yet been reached, then it prints the mean phenotype every `100` generations.

If we recycle and run the model now, it should produce something like this:

```
100: Mean phenotype == 0
200: Mean phenotype == 0.000674937
300: Mean phenotype == 0
400: Mean phenotype == 0
500: Mean phenotype == 0
600: Mean phenotype == 0
700: Mean phenotype == 0.000261862
800: Mean phenotype == 0
900: Mean phenotype == 0.484062
1000: Mean phenotype == 1.0268
1100: Mean phenotype == 1.27179
1200: Mean phenotype == 1.27739
1300: Mean phenotype == 1.3027
1400: Mean phenotype == 1.30019
1500: Mean phenotype == 1.3013
...
15000: Mean phenotype == 9.72518
15100: Mean phenotype == 9.74396
15200: Mean phenotype == 9.74479

15279: REACHED OPTIMUM
Final phenotype == 9.91894

          [,0]     [,1]
[0,]         1 0.490805
[1,]         1 0.405696
[2,]         1 0.481056
[3,]     0.001 0.549967
[4,]         1 0.832555
[5,]         1  1.31566
[6,]         1 0.650096
[7,]         1 0.396507
[8,]         1 0.300017
[9,]     0.099 0.873996
```

It takes a little while to get going, because the model depends upon new random QTL mutations, but eventually it starts moving toward the optimum, and reaches it in generation `15279`. The final summary produced by `dumpFrequencies()` reveals that almost all of the QTLs in the model fixed, but one QTL in particular, with an effect size of ~`0.87`, is at a frequency of about 10%, and that allowed the mean phenotype to reach the optimum (within the defined tolerance). If that QTL were to actually fix, the population's mean phenotype would probably overshoot the optimum, but then of course QTLs with a negative effect size could arise to compensate.

EXERCISE: Make a copy of the model's code into a new window (so you keep the original code), and play around with the copy. Try things like modifying the phenotypic optimum, the width of the fitness function, and the strength of selection to see how they affect the outcome. Play with the `dnorm()` function in the Eidos console to understand what it does. Also, make sure you understand the vectorized operations used in the fitness function code. We're about to go further, so make sure you've got all the concepts. Close the copy when you're done.

3. A moving phenotypic optimum

So far, the phenotypic optimum has been a constant, `10.0`. Let's extend the model by making the optimum change over time. In a change of pace, we're doing to do this as a series of exercises.

EXERCISE: We don't want the model to terminate when the optimum is reached any more. Remove that code, while keeping the code that outputs the mean phenotype every `100` generations. Also remove the `"DID NOT REACH OPTIMUM"` log, and extend the model to `100000` generations.

EXERCISE: Add a variable, `opt`, at the beginning of the `1: late()` event, and assign it a value that flips back and forth between `0.0` and `10.0` every `10000` generations; in other words, it should be `0.0` for the first `10000` generations, `10.0` for the next `10000` generations, then back to `0.0` for `10000` generations, and so forth. If you want to try working this out for yourself, using the Eidos console to test your solution, feel free to stop reading now. For everyone else:

`opt = (floor(sim.generation / 10000) % 2) * 10.0;` should work.

Alter the call to `dnorm()` so that it uses `opt` as the optimum rather than `10.0`. Also, extend the logging that is done every `100` generations to log the current optimum as well; use the string `+` operator to add additional output to the call.

With these changes, if you recycle and run you should see the optimum flipping between `0` and `10` every `10000` generations (as printed by your modification to the `catn()` call), and the mean phenotype should "chase" the moving optimum. The time between environmental shifts is long enough that the adaptation probably involves fixation of most of the QTLs involved; adapting back in the opposite direction then has to come from new QTL mutations.

4. Environmental variance

Thus far, phenotype is a completely deterministic consequence of genetics: for an individual with a particular set of QTL mutations, we can predict its phenotype exactly. Real quantitative traits are generally not so predictable; instead, the phenotype of an individual usually contains a large random element as well. This is often called by names like "environmental variance" or "developmental noise", but it is essentially just a random contribution to phenotype.

EXERCISE: Add environmental variance to the model. To do this, change the name of the `phenotypes` vector, in the line where it is calculated, to `additive` (don't rename its later use, just rename it where it is first assigned). After `additive` is defined, add a new line that defines a variable named `noise`. The `noise` variable should consist of `500` draws from a normal distribution of mean `0.0` and standard deviation `1.0`, using the `rnorm()` function (use the online help as needed). Finally, after the definition of noise, add a new line that simply reads: `phenotypes = additive + noise;`

Do you think the added noise will make the model adapt more effectively or less? Recycle and run, and see if you can see any difference. Try increasing the standard deviation of the noise to `10.0`; do you see a difference now?

5. Heritability

Heritability, in the sense we will use it here (so-called "narrow-sense heritability", often symbolized as *h*), is the fraction of total phenotypic variance that is explained by the additive genetic contribution (as opposed to environmental variance and factors such as dominance and epistasis). It is, loosely speaking, a measure of the degree to which the phenotypes of offspring are predictable from the phenotypes of their parents. Measuring and calculating heritability is an important topic in quantitative genetics, particularly for researchers involved in plant and animal breeding.

Heritability can be influenced by many factors, but one is certainly the amount of environmental variance present; the higher the environmental variance, the lower the heritability is expected to be.

We should be able to explore this, since we added environmental variance to our model in the previous step.

EXERCISE: Add a calculation of *heritability*. Eidos provides a function named `var()` that calculates the variance of a vector, so heritability is then simply `var(additive) / var(phenotypes)`. Add a calculation of heritability after phenotypes are calculated, and add it to the end of the `100`-generation logging line. With a standard deviation of `10.0` for noise, what is the heritability? Is it constant over time? Why do you think it behaves the way it does? If you change the standard deviation to `1.0`, what is the heritability then?

6. Genetic structure and initial variation

One often wants to model QTLs at specific locations in the genome, perhaps following empirical data. The model we've built here gives a nod to that possibility, by establishing a "gene" in the middle of the chromosome which is where QTLs can arise. Obviously this could be taken much further; one could map QTLs down to even single base positions on the chromosome, and could use a different mutation type for different QTLs to represent known effect sizes for QTLs at different positions. One could even go a step further, with a nucleotide-based model, and model specific effect sizes for specific SNPs at the QTL positions modeled.

Often one also wants to start a QTL model, not from empty genomes as shown here, but from some specific starting state that involves particular QTL mutations at particular loci. Given a complete description of the desired initial state, one option is to set up that state with calls to `addNewMutation()` or `addNewDrawnMutation()`. Reading an initial state from a VCF file is also possible using SLiM's `readFromVCF()` method, and easier.

We won't delve into these topics now, but they are certainly possible in SLiM.

7. Phenotypic plasticity

An organism that exhibits *phenotypic plasticity* is able to adapt, to a limited extent, to the environment in which it finds itself when it is a juvenile (or perhaps even pre-natally). To put this more precisely: the intrinsic phenotype encoded for by the additive (and perhaps non-additive) effects of the QTLs it possesses can be modified, to a limited extent, in a direction that would (presumably) make the organism better-adapted and more likely to survive. How might you add phenotypic plasticity to this model?

EXERCISE: Contemplate how you might implement phenotypic plasticity. There may not be time to pursue this relatively advanced idea now, so please do your **BEEP!** now and then have a go at this problem while others are finishing. You can try to figure it out from what you already know, or you can look at the hints that follow.

Hint #1: a nice design would be to assign a developmental bias from the environment in a `modifyChild()` callback.

Hint #2: the callback could calculate what the proposed child's phenotypic trait value would be, without phenotypic plasticity, and compare that to the environment. It could then assign a plastic effect in the direction of greater adaptation.

Hint #3: the developmental bias might be stored in the `tag` field of the proposed child (always an easy place to stick model state), and then the `tagF` values would be added in to the phenotypic trait value calculations. Do this with a single vectorized statement, not with a loop.

8. BEEP! With extra time, work on the last exercise above, or look at SLiM manual section 13.4, which explores quantitative traits and heritability further, with a somewhat different approach.