SLiM Workshop Series #21: Nucleotide-based Models

1. Start with the default model in SLiMgui

Close all open windows in SLiMgui, and then open a new window with command-N. We're going to adapt the default script to be nucleotide-based, to cover each of the steps involved.

First of all, at the very top of the `initialize()` callback, add the call:

```
initializeSLiMOptions(nucleotideBased=T);
```

This simply tells SLiM that we're making a nucleotide-based model. That flips various behavioral switches inside SLiM; for example, an ancestral nucleotide squence will now be required. We can supply one by adding these lines, directly below the above call:

```
defineConstant("L", 1e6);
initializeAncestralNucleotides(randomNucleotides(L));
```

We define a constant for `L`, the chromosome length, since we're going to use it in more than one place. Then we call `randomNucleotides(L)` to generate a random ancestral sequence, and pass the result to `initializeAncestralNucleotides()`. This establishes the default nucleotides that exist in all genomes whenever no nucleotide-based mutation is present to override that default; you could think of it as the sequence of the coalescent ancestor that all individuals in the model descend from.

Let's go to the Eidos console for a moment (click the magenta `>` button at the top of the script pane in SLiMgui) to experiment with `randomNucleotides()`. Try the following commands. To type the first command efficiently, you can type just `ran` and then use code completion by pressing the escape key. For the rest, a useful trick is to press the up-arrow key, when at the Eidos console prompt; this gives you the previous command you executed, which you can then edit and execute again. You can also code-complete on `randomNucleotides(10, f` and it will give you `format=`, if you want to be really speedy. Practice history editing and code completion with these:

```
> randomNucleotides(10)
> randomNucleotides(10, format="string")
> randomNucleotides(10, format="char")
> randomNucleotides(10, format="integer")
```

The default format is `"string"`, as you have just seen, and that is the most efficient choice for passing a sequence directly through to `initializeAncestralNucleotides()`. The other representations can be useful if you want to edit the nucleotide sequence in some way (guaranteeing a particular ancestral nucleotide at a particular position, for example). These formats – `"string"`, `"char"`, and `"integer"` – are generally supported across SLiM's interfaces for working with nucleotide-based models.

Incidentally, if you want to read the ancestral sequence from a FASTA file, you can simply pass a filename (or a full file path) to `initializeAncestralNucleotides()`, rather than a sequence. SLiM will read the first sequence defined in the FASTA file and use it.

There's one thing we need to patch up. Where `99999` is given as the last base position, in the call to `initializeGenomicElement()`, change that to `L-1` so we're using `L` everywhere we ought to:

```
initializeGenomicElement(g1, 0, L-1);
```

Next, we should convert mutation type `m1` to be nucleotide-based. This is easy; just change the call to `initializeMutationType()` to call `initializeMutationTypeNuc()` instead. Everything else can stay the same:

```
initializeMutationTypeNuc("m1", 0.5, "f", 0.0);
```

The parameters to `initializeMutationTypeNuc()` are exactly the same; the different function name simply tells SLiM that `m1` mutations should have nucleotides associated with them. You can use non-nucleotide-based mutation types in nucleotide-based models, too, although we won't explore that here.

2. Set up a mutation matrix

If you recycle and run now, you will get an error:

```
ERROR (SLiMSim::ExecuteContextFunction_initializeMutationRate):
initializeMutationRate() may not be called in nucleotide-based models (use
initializeHotspotMap() to vary the mutation rate along the chromosome).
```

Nucleotide-based models use a somewhat different mutational model in SLiM, because it is often desirable to have a sequence-based mutation rate – to have different rates for transitions than for transversions, for example. For this reason, `initializeMutationRate()` is not used in nucleotide-based models; the mutation rate at a given base position depends upon the nucleotide currently present at the location, in the genome that might be mutating. (The mutation rate can even depend upon the surrounding nucleotides, to allow phenomena such as a higher mutation rate at `CG` sites to be modeled.) Delete the call to `initializeMutationRate()`.

Instead, we need to give a mutation rate matrix to `initializeGenomicElementType()`. Each genomic element type can have a different mutation matrix, if desired, allowing the sequence-based mutation rates to vary along the chromosome. Change the `initializeGenomicElementType()` call to have an additional parameter:

```
initializeGenomicElementType("g1", m1, 1.0, mmJukesCantor(1e-7 / 3));
```

The Jukes–Cantor mutational model is the simplest mutational model: each nucleotide will mutate into each other nucleotide at the same given rate. Add the above and recycle, but don't run yet. Then go back to the Eidos console and type:

```
> mmJukesCantor(1e-7 / 3)
            [,0]        [,1]        [,2]        [,3]
[0,]           0 3.33333e-08 3.33333e-08 3.33333e-08
[1,] 3.33333e-08           0 3.33333e-08 3.33333e-08
[2,] 3.33333e-08 3.33333e-08           0 3.33333e-08
[3,] 3.33333e-08 3.33333e-08 3.33333e-08           0
```

This is a mutation matrix (Eidos supports matrices – two-dimensional arrays of values – as well as higher-dimensional arrays, much as R does, but we haven't had much cause to use them). For the Jukes–Cantor model, it follows a very simple form:

$$\begin{pmatrix} 0 & \alpha & \alpha & \alpha \\ \alpha & 0 & \alpha & \alpha \\ \alpha & \alpha & 0 & \alpha \\ \alpha & \alpha & \alpha & 0 \end{pmatrix}$$

Each row of the matrix represents one possible nucleotide state prior to mutation; there could be an `A`, `C`, `G`, or `T` present at the mutating site, represented by rows 0 through 3 respectively. Each column provides the corresponding mutation rates for the mutation of a given state to a new state: `A`, `C`, `G`, or `T`, as represented by columns 0 through 3 respectively. For a mutating site that presently has a `C` in it (row 1), for example, the probabilities of mutating to an `A`, `G`, or `T` are all given as `~3.3e-8` (columns 0, 2, and 3). The probability of an "identity" mutation, of a nucleotide to itself, must conventionally

be specified as zero, as it is here (the zero values along the matrix diagonal). The parameter α is here `1e-7 / 3`; this results in a mutation rate of `1e-7` (per base position per generation, as usual), since for any given nucleotide there are three possible mutations to undergo, each with the rate of `1e-7 / 3`. The overall mutation rate in the model is therefore unchanged in our adaptation of the model to use nucleotides.

Now close the Eidos console and run the model. It will look like a normal SLiM model when running, but it is in fact keeping track of the nucleotide sequence of every genome as it runs. The only evidence of this in SLiMgui is in the output generated by the model. For example, `outputFixedMutations()` now generates something like this:

```
#OUT: 2000 F
Mutations:
0 14891 m1 432658 0 0.5 p1 151 1121 A
1 50461 m1 847086 0 0.5 p1 504 1348 T
2 25507 m1 854116 0 0.5 p1 256 1348 A
3 2700 m1 935085 0 0.5 p1 28 1348 C
...
```

The final column of the method's output gives the nucleotide associated with each fixed mutation, as a letter `A` / `C` / `G` / `T`. The same is true of other SLiM output formats, such as `outputFull()` and `outputSample()`.

SLiM is happy to use any mutation matrix you give it, more or less, as long as the "identity" rates are all zero. The `mmKimura()` function provides a mutation matrix for the Kimura (1980) model, in which the rates for transitions and transversions differ. More complex matrices can be built in script. One can even supply a 64×4 matrix, instead of a 4×4 matrix, to provide different mutation rates for every possible trinucleotide sequence (for the mutation of the central base of each possible trinucleotide). The SLiM manual has recipes and discussion presenting such models in detail; we won't get into it further here.

3. Nucleotides for mutations and substitutions

Change the definition of `L` to `1e2`. This gives us a nice short chromosome, which will be more convenient for what we're about to do. Then change the parameter passed to `mmJukesCantor()` from `1e-7 / 3` to `1e-4`, so that we'll still get some genetic diversity despite the very short chromosome modeled. Recycle and run to the end.

What we want to explore now is the ways that SLiM lets you look at the nucleotides present in the model. First let's look at the nucleotides associated with mutations. Open the Eidos console and try this:

```
> sim.mutations.nucleotide
"G" "T" "T" "A" "T" "A" "C" "C" "C" "G" "G" "A" ...
```

Every mutation of type `m1` has an associated nucleotide (because `m1` is a nucleotide-based mutation type), and the `nucleotide` property accesses it as a character. Now try this:

```
> sim.mutations.nucleotideValue
2 3 3 0 3 0 1 1 1 2 2 0
```

These values represent the same nucleotides, but as integers; in SLiM, `"A"` is `0`, `"C"` is `1`, `"G"` is `2`, and `"T"` is `3`, by convention. You can work with characters or with integers, as you see fit; integers will be more efficient if you're manipulating large numbers of nucleotide values.

Nucleotides are also available for substitutions, of course:

```
> sim.substitutions.nucleotide
"T" "A" "C"
> sim.substitutions.nucleotideValue
3 0 1
```

4. The ancestral nucleotide sequence

The ancestral sequence is available from the `Chromosome` object:

```
> sim.chromosome.ancestralNucleotides()
"TGTTATGCTGCTGTGGGTGAGTTGTGCTTATTCCATAACTTCACTTACAATGAGCCCATGGAGGTTAGCGGAA
TTGCGCCGGCATGAGCTTGTGAGGGTC"
```

There is an important point to be made here. To see it, we need to modify the model to print out the ancestral sequence at the beginning. Change the call to `initializeAncestralNucleotides()` to the following lines:

```
s = randomNucleotides(L);
print(s);
initializeAncestralNucleotides(s);
```

Also, change the generation for the final output event from `2000` to `5000`; we want to run long enough to be sure to get some fixed mutations. If you recycle and run, you should see an ancestral sequence printed near the beginning of the output, like this:

```
"CAACTGTTCGCCGATCATCTCTAATCCGAAAAGTAGTTGTCTCATAGGGGCGTATAAGAAGTGATAATGTGCA
AAGCGATTCCGGTCAACTCTGGGGGTA"
```

Now go to the Eidos console, and again fetch the ancestral sequence:

```
> sim.chromosome.ancestralNucleotides()
"CAACTGTGCGGCGATCATCTCTAATCCGAAACGTATTTGTCTCATAGGCGCTTATATGAAGTTATAATGTGCA
AAGCGGGTCAGGTCAACTCGGGGGGTA"
```

They are not the same! This is because some mutations have fixed (presumably – if none fixed in your run of the model, just recycle and run again so you can follow along). For example, in the output above, a `T` has changed to a `G` at position 7 (counting from zero, as SLiM does). We can confirm that this makes sense in the Eidos console:

```
> sim.substitutions.nucleotide
"C" "A" "T" "G" "G" "T" "C" "G" "T" "G" "T" "G"
> sim.substitutions.position
31 82 62 78 92 56 48 7 35 79 51 10
```

There is indeed a substitution at position `7`, and it is indeed a fixed `G` mutation. The lesson here is that whenever a nucleotide-based mutation fixes and is converted to a `Substitution` object by SLiM, the ancestral nucleotide sequence will be updated to the new nucleotide to reflect that. Mutations that are prevented from substituting, with the `convertToSubstitution` property of `MutationType`, continue to segregate in the model and so the ancestral sequence is not updated. This all makes sense if you remember that the ancestral sequence provides the default nucleotide at a given position whenever a nucleotide-based mutation is not present in a given genome to override the default.

If you option-click on `ancestralNucleotides()` to bring up the online help for it, you will see that it supports a `format` parameter just as `randomNucleotides()` did:

```
– (is)ancestralNucleotides([Ni$ start = NULL], [Ni$ end = NULL],
      [string$ format = "string"])
```

You can also fetch just the nucleotides within a particular range, using `start` and `end`.

5. Genome nucleotide sequences, codon sequences, and amino acid sequences

Nucleotide sequences are also available for genomes. The nucleotide sequence for a genome is, by definition, the ancestral nucleotide sequence overlaid with the nucleotides for any nucleotide-based mutations contained by the genome. Open the Eidos console and try this:

```
> p1.genomes[0].nucleotides()
"TAATCACCTCTCAACCATACGAGACAGGACAAATATGAGTTTTTGCGAGCCGTATGCGTAATACTCGCATTTT
ACGCATTTCAGCGGGACTCGGGCAGGA"
> p1.genomes[1].nucleotides()
"TAACCACCTCGCAACCATACGAGACCCGACAAGTATGAGTTTTTGAGAGACGTATTCGTAATACTCTCATCTT
ATGCAATTCAGCGTGACTCGGGCAGGA"
```

The sequences are very similar, because they are based upon the ancestral sequence at most positions (or share the same segregating mutation), but they diverge at some points. If you option-click `nucleotides()`, you will see some other documentation hits for things that contain "nucleotides" in their name, but at the end you should see the documentation for `nucleotides()`. It, too, has a `format` parameter that allows you to choose `"char"` or `"integer"` instead of `"string"`. If you read the documentation, it also supports an option, `"codon"`, that provides integer codon values, in [0, 63], for successive trinucleotides. The sequence requested must be a multiple of three in length. Let's try that:

```
> c = p1.genomes[0].nucleotides(0, 29, "codon")
> c
48 52 23 29 1 19 6 8 18 33
```

This option is also supported by `ancestralNucleotides()`, in fact, and the documentation for that method details the encoding used; in short, `AAA` is `0` and `TTT` is `63`, and you might guess the rest.

Codons can be useful if you want to work with the nucleotide sequence in a way that imparts specific meaning to particular codons – giving stop codons within a gene a highly deleterious fitness effect, for example. For that sort of purpose, working at the amino-acid level can be even more useful, and SLiM provides support for that too. Try this:

```
> codonsToAminoAcids(c)
"XSPLNHTRQD"
```

The letters here are the standard letters used to represent amino acids. The standard DNA codon table is used. If you prefer three-letter codes, that option is provided:

```
> codonsToAminoAcids(c, long=T)
"Ter-Ser-Pro-Leu-Asn-His-Thr-Arg-Gln-Asp"
> codonsToAminoAcids(c, long=T, paste=F)
"Ter" "Ser" "Pro" "Leu" "Asn" "His" "Thr" "Arg" "Gln" "Asp"
```

These options might be useful for generating human-readable output from a nucleotide-based model.

6. Codon-based fitness

As a final nucleotide-based adventure, let's look at a model that makes fitness depend upon the codon sequence generated by a particular stretch of the genome. This model is a bit complex, so

we're not going to build it from scratch; instead, close open windows and then open recipe 18.9 from the **Open Recipe** submenu of the **File** menu.

The model begins with some bookkeeping work:

```
defineConstant("TAA", nucleotidesToCodons("TAA"));
defineConstant("TAG", nucleotidesToCodons("TAG"));
defineConstant("TGA", nucleotidesToCodons("TGA"));
defineConstant("STOP", c(TAA, TAG, TGA));
defineConstant("NONSTOP", (0:63)[match(0:63, STOP) < 0]);
```

This looks up the codons for `TAA` (48), `TAG` (50), and `TGA` (56); these three codons are stop codons in the standard DNA codon table. It defines a constant, `STOP`, for those three, and a constant, `NONSTOP`, for all the rest. (The `match()` function is a very useful function that you ought to acquaint yourself with.)

Next come some lines that construct an ancestral sequence. This model is of a single gene, with exons spanning base positions [253, 670] and [871, 1034]. Outside of those exons, we want the ancestral sequence to use completely random nucleotides; inside the exons, we ensure that the sequence does not contain a stop codon, since we want to start with a functional gene. You might want to examine that code until you understand how it works.

The rest of the model is not very interesting – fairly standard initialization and output – except this:

```
fitness(NULL) {
   for (g in individual.genomes)
   {
      seq = g.nucleotides(253, 670) + g.nucleotides(871, 1034);
      codons = nucleotidesToCodons(seq);
      if (sum(match(codons, STOP) >= 0))
         return 0.0;
   }

   return 1.0;
}
```

We haven't seen `fitness(NULL)` callbacks before; they're rarely used in SLiM nowadays, because they have largely been replaced by the use of the `fitnessScaling` property (and indeed, we could easily reshape this code to use `fitnessScaling` instead). But they're very simple: `fitness(NULL)` callbacks are a special kind of `fitness()` callback that gets called, not once per mutation per focal individual, but simply once per focal individual. They let you define a per-individual fitness effect that can depend upon whatever individual state or model state you wish.

Here, we loop over the two genomes in the focal individual. For each genome, we get the exonic nucleotides, and paste them together – we're basically splicing the mRNA transcript here, although we didn't transcribe the DNA into RNA. Then, given that nucleotide sequence – the inverse of the mRNA – we get the codons it represents. If any codon is a stop codon – again using the very useful `match()` function – a nonsense mutation has occurred and we return a fitness effect of `0.0`. Otherwise, this is a neutral model and so we return `1.0`.

This model takes quite a while to run, because we want it to undergo a large number of mutations; you probably don't want to run it to the end. But if you run it briefly, you'll see that individuals are occasionally shown as black; those individuals have a nonsense mutation and a fitness of `0.0`.

EXERCISE: Modify the `fitness(NULL)` callback to translate the codons into amino acids, and then check whether any amino acid is a stop (`"X"`). You will no longer use `STOP` or `match()`, but the `strsplit()` function will likely come in useful.

7. BEEP! With extra time, look at SLiM manual section 18.4, which covers some useful concepts.