Slendr simulations

Mark Ravinet

Introduction - slendr

slendr is a recently developed and extensively well-managed R package that acts as a front end for demographic simulations with both `msprime` and `SLiM`. It massively simplifies the process of running these programs, making them easy to interface with via `R` - so if you're not great at `python` (like me!) you can easily use `R` instead. It is also very flexible and because it is built on top of the extremely efficient simulation programs, it is very fast and lightweight.

Much of the information in this tutorial is based on the original `slendr`](https://www.slendr.net/) tutorial docs - I would strongly recommend referring to these as they are the definitive source for learning how to start to using the package.

Setting up

First we need to load the packages we will need for this session. They have already been installed, so this will not take long.

```
library(slendr)
library(tidyverse)
```

Note that if you install `slendr` on your own machine, you will need it to setup all the back-end it uses to interact with python etc. We will not do this today because it is already configured for you. However to ensure you know how to do this, you simply run

```
setup_env()
```

Even if `slendr` is setup, it is always a good idea to initiate the environment, to ensure everything is working properly ahead of analyses. Do this like so:

```
init_env()
```

Now we're ready to begin simulating!

A simple non-spatial simulation with two populations

`slendr` allows you to combine some simple and logically named R functions into a complex demographic model. The easiest way to learn how to do this is to simply use these functions! Here we will simulate two populations. The second splits from the first 30,000 generations ago.

```
# Population a with 30,000 individuals, arising 50000 generations ago
a <- population("a", time = 50000, N = 30000)

# Population b with 15,000 individuals, arising 30,000 generations ago
b <- population("b", parent = a, time = 30000, N = 15000)
```

Be sure to check what these populations look like in the R environment. Just call them as objects to see.

Since we are keeping this model very simple in order to learn, all we need to do next is compile the model - all components are combined (i.e. pop size etc) into a single R object.

```
model <- compile_model(
  populations = list(a, b),
  generation_time = 1,
  direction = "backward",
)
```

Note that here we specify the direction of this model is "backwards" - i.e. we are performing a coalescent (backwards-in-time) simulation.

One really nice feature of `slendr` is that it allows you to plot the model to ensure you have specified it correctly.

```
plot_model(model)
```

If everything looks as it should, we are ready to begin simulating.

Running a simulation

Next up, we need to set our sampling scheme. Here we will take 30 individuals from each of our two populations:

```
samples <- schedule_sampling(model, times = 0, list(a, 30), list(b, 30))
```

Note that this function is called `schedule_sampling` - this is because it allows you to set the timing of sampling (i.e. you can sample at different temporal points across the model) and also the location - actually spatially if you wish. But more on that later.

With this setup, we can run the simulation. We will use `msprime` because this is a coalescent simulation over many thousands of generations and therefore it is much more efficient and fast.

```
ts <- msprime(model, samples = samples, sequence_length = 1000, recombination_rate = 0)
```

This should only take a few seconds. Note we have simulated a 1000 bp seuqence here with no recombination, but we could easily alter this if we wanted to. Next we should take a look at the simulation output.

```
ts
```

So far this doesn't show a great deal, other than the fact that it is a tree-sequence stored temporarily on our computer. To get at what the simulation actually shows, we need to process it.

Processing simulations - a simple example

Right now our simulation output is a tree sequence of 60 individuals, 30 from population a and 30 from population b. But we might want to calculate some population genetic statistics on this dataset. So here work towards calculating F_{ST}.

First of all, we need to set up our populations - i.e. let the pipeline know which individuals are in each population. This is easy enough to do with built in functions and some `tidyR`.

```
a_pop <- ts_samples(ts) %>% filter(pop == a) %>% .$name
b_pop <- ts_samples(ts) %>% filter(pop == b) %>% .$name
```

Next we need to add mutations to our tree sequence. One of the reasons msprime is so fast is that it records only the tree sequence or genealogy - mutations can be added after the fact in order to ensure the simulation is extremely efficient. Here we will add simulations using a basic SNP mutation rate.

```
ts_m <- ts_mutate(ts, mutation_rate = 1*10^-4)
```

Now with mutations added and our populations defined, we can use the `tskit` functions (all denoted in the package by starting with `ts_`) to calculate F_{ST}. This is extremly fast and there are a large number of different functions to calculate different statistics.

```
ts_fst(ts_m, sample_sets = list(a = a_pop, b = b_pop))
```

We can also calculate diversity using the `ts_diversity` function - **nb** this is per site - to get `pi` for the sequence, you would need to divide this value by the length (1000 in this case).

```
ts_diversity(ts_m, sample_sets = list(a = a_pop, b = b_pop),
             mode = "site")
```

So there we have it - two population genetic statistic estimates for a simple 2-population model, all calculated in a single R framework at high speed. If we placed all the commands we used together in an single R script, we could have run this all in seconds, using very little hard drive space.

A simple simulation pipeline

Running a single simulation is useful for learning but it isn't that helpful as a standalone tool. Instead, we can combine the code we've learned to examine how the variaton in population parameters might influence the statistics we calculate. Here we will run the model we created above for different population sizes for population B (from 1000 to 15000) - how does it alter our estimate of F_{ST}?

This code might look complex - but it is almost everything we covered above!

```
# set our sequence to simulate across
pop_sizes <- seq(1000, 15000, by = 1000)

# run our model within an sapply command
fst_i <- sapply(pop_sizes, function(x){
  # set the sampling scheme
  samples <- schedule_sampling(model, times = 0, list(a, 30), list(b, 30))
  # Population a
  a <- population("a", time = 50000, N = 30000)
  # Population b
  b <- population("b", parent = a, time = 30000, N = x)

  # compile model
  model <- compile_model(
    populations = list(a, b),
    generation_time = 1,
    direction = "backward",
  )
  # run the model
```

```
  ts <- msprime(model, samples = samples, sequence_length = 1000, recombination_rate = 0)
  # add the mutations
  ts_m <- ts_mutate(ts, mutation_rate = 1*10^-4)
  # set the populations
  a_pop <- ts_samples(ts) %>% filter(pop == a) %>% .$name
  b_pop <- ts_samples(ts) %>% filter(pop == b) %>% .$name
  # calculate fst
  y <- ts_fst(ts_m, sample_sets = list(a = a_pop, b = b_pop))
  y$Fst
})
```

We can then combine our results into a `data.frame` and plot the results using `ggplot` to see how they vary across the parameters.

```
# create a data.frame or tibble
sims <- as_tibble(data.frame(pop_sizes = pop_sizes, fst = fst_i))

# plot the output
ggplot(sims, aes(pop_sizes, fst)) + geom_point() + geom_line()
```

Quite clearly from this simple simulation we can see that as the populations size of B increases, so does the magnitude of F_{ST} between the two populations.

Adding to the model: gene flow

The model we have been using so far is very simple - it is basically a 2-deme isolation model. But what if we want to add gene flow between our two populations? We can do this very easily using the `gene_flow` function.

```
gf <-
  list(gene_flow(from = a, to = b, rate = 0.2, start = 10000, end = 0),
       gene_flow(from = b, to = a, rate = 0.2, start = 10000, end = 0)
  )
```

Here we have ensured that we are simulating gene flow that is symmetric between our two populations at a rate of 0.2 between populations. This starts 10,000 generations in the past and continues until the present day. We have defined these two events in and combined them into a `list` called `gf`

Next we compile our model, but this time include a gene flow term.

```
model <- compile_model(
  populations = list(a, b),
  gene_flow = gf,
  generation_time = 1,
  direction = "backward",
)
```

And as before, we can plot the model to see what it looks like:

```
plot_model(model)
```

As before, we can run our model and see how it alters our estimates of F_{ST} and diversity. We need to rerun our model, populate it with mutations, define our populations and then calculate the statistics.

```
  # run the model
  ts <- msprime(model, samples = samples, sequence_length = 1000, recombination_rate = 0)
  # add the mutations
  ts_m <- ts_mutate(ts, mutation_rate = 1*10^-4)
  # set the populations
  a_pop <- ts_samples(ts) %>% filter(pop == a) %>% .$name
  b_pop <- ts_samples(ts) %>% filter(pop == b) %>% .$name
  # calculate fst
  ts_fst(ts_m, sample_sets = list(a = a_pop, b = b_pop))
  # calculate diversity
  ts_diversity(ts_m, sample_sets = list(a = a_pop, b = b_pop),
               mode = "site")
```

Yet again, a single value is interesting but it doesn't tell us too much. So we will do what we did previously - rerunning our model but altering the population size of population B. **However**, this time we will see how gene flow influences our inference!

```
# set our sequence to simulate across
pop_sizes <- seq(1000, 15000, by = 1000)

# run our model within an sapply command
fst_g <- sapply(pop_sizes, function(x){
  # set the sampling scheme
  samples <- schedule_sampling(model, times = 0, list(a, 30), list(b, 30))
  # Population a
  a <- population("a", time = 50000, N = 30000)
  # Population b
```

```
  b <- population("b", parent = a, time = 30000, N = x)

  # compile model - note the inclusion of gene flow
 model <- compile_model(
  populations = list(a, b),
  gene_flow = gf,
  generation_time = 1,
  direction = "backward",
)

  # run the model
  ts <- msprime(model, samples = samples, sequence_length = 1000, recombination_rate = 0)
  # add the mutations
  ts_m <- ts_mutate(ts, mutation_rate = 1*10^-4)
  # set the populations
  a_pop <- ts_samples(ts) %>% filter(pop == a) %>% .$name
  b_pop <- ts_samples(ts) %>% filter(pop == b) %>% .$name
  # calculate fst
  y <- ts_fst(ts_m, sample_sets = list(a = a_pop, b = b_pop))
  y$Fst
})
```

Now we can add the output of the simulations from this run to our previous simulations (those without gene flow) and see the difference

```
# combine everything into a tibble
sims <- as_tibble(data.frame(sims, fst_g))
# alter names! you will see why shortly
colnames(sims) <- c("pop_sizes", "isolation", "gene_flow")
# pivot to allow easy plotting
sims_p <- pivot_longer(sims, -pop_sizes, names_to = "model", values_to = "fst")
# plot the output
ggplot(sims_p, aes(pop_sizes, fst, colour = model)) + geom_point() + geom_line()
```

This shows that changing the population size of b has the same result in both models (i.e. F_{ST} decreases with increasing pop size) but that F_{ST} is lower in the gene flow model - as we would expect!

Adding to the model: resizing a population

As well as gene flow events, we can also resize populations so that they experience bottlenecks or growth over time. Yet again, `slendr` makes this very straightforward. All we need to do is

pipe our population declaration to a `resize` function when we declare populations.

```
# Population a with 30,000 individuals, arising 50000 generations ago
a <- population("a", time = 50000, N = 30000)

# Population b with 15,000 individuals, arising 30,000 generations ago
b <- population("b", parent = a, time = 30000, N = 15000) %>%
  resize(N = 2000, how = "step", time = 5000, end = 0)
```

Remember because we are working with coalescent models, we are working backwards in time. So here we set population b to start with a population size of 2000 at time 0 and this then increases to the ancestral population size 5000 generations in the past. Importantly we set the `how` for this resize to `step` - i.e. it will just change suddenly.

Then we need to just declare our model again. For simplicity here, we'll do this without gene flow.

```
model <- compile_model(
  populations = list(a, b),
  generation_time = 1,
  direction = "backward",
)
```

Then we can plot it to see how this looks!

```
plot_model(model)
```

And what if we had set this to happen exponentially, rather than a sudden step?

```
# Population a with 30,000 individuals, arising 50000 generations ago
a <- population("a", time = 50000, N = 30000)

# Population b with 15,000 individuals, arising 30,000 generations ago
b <- population("b", parent = a, time = 30000, N = 15000) %>%
  resize(N = 2000, how = "exponential", time = 5000, end = 0)

# recompile the model
model <- compile_model(
  populations = list(a, b),
  generation_time = 1,
  direction = "backward",
)
```

```
# plot the model
plot_model(model)
```

We won't include the resizing in our simulation pipeline above because next we will try to develop a simple spatial simulation - this will be a very useful tool for landscape genomics!

Spatial models

So far, our models have all been relatively basic with no spatial context. But what if adding a spatial context helped make them more realistic? This is a very difficult and complex topic but it is the main motivation behind the development of `slendr` - again another reason to refer to its excellent and extensive website.

Setting up the spatial context

We will try to take our basic model and put it in a spatial context in order to get a flavour of the kind of things you can do with `slendr`. The very first thing we will do is define a map that we will use - we use the `world` function to do this - and we simply set the longitude (`xrange`) and latitude (`yrange`) to do this.

```
map <- world(
  xrange = c(-13, 70), # min-max longitude
  yrange = c(18, 65),  # min-max latitude
  crs = "EPSG:3035"    # coordinate reference system (CRS) for West Eurasia
)
```

With this set, we can then plot the map using the `plot_map` function from `slendr` to make an easy map as a background for simulations.

```
plot_map(map)
```

Next we will create two regions on our map - one over the UK, the other over Europe.

```
# anatolia
anatolia <- region(
  "Anatolia", map,
  polygon = list(c(28, 35), c(40, 35), c(42, 40),
                 c(30, 43), c(27, 40), c(25, 38))
)
# europe
```

```
europe <- region(
  "Europe", map,
  polygon = list(
    c(-10, 35), c(-5, 36), c(10, 38), c(20, 35), c(23, 35),
    c(30, 45), c(20, 52), c(0, 50), c(-10, 48)
  )
)
plot_map(anatolia, europe)
```

So here we have a map with two polygons imposed on the top that define regions. With this spatial structure set up, we can now start to build a model that is anchored in this geographical context.

Building a spatial model

As with our simpler, non-spatial models, we can start to specify our model using the same functions as previously - i.e. `population`, except this time we actually incorporate the map data.

```
# european population
eur <- population(
  name = "eur", time = 6000, N = 2000,
  polygon = europe, map = map
)
# check it by plotting!
plot_map(eur)
# anatolian population
ana <- population( # Anatolian pop
  name = "ana", time = 6000, N = 3000,
  center = c(34, 38), radius = 500e3, polygon = anatolia, map = map
) %>%
  expand_range( # expand the range by 2.500 km
    by = 2500e3, start = 5000, end = 3000, overlap = 0.5,
    polygon = join(europe, anatolia)
  )
```

Note that the arguments `map` and `polygon` allow us to specify the map and polygon we have already defined - it is these arguments which give our population its spatial rooting.

With this done, we can now replot these polygons and you will see we have defined the ranges of the populations within the polygons - here they are explicitly bounded to the landscape. We will see why this is important shortly.

```r
plot_map(ana)
# plot both together
plot_map(eur, ana) # showing an expansion into europe
```

With our populations defined, we can also set out some gene flow events. We will do this exactly the same way we did with our non-spatial model earlier.

```r
# this will not work
gf <- gene_flow(from = ana, to = eur, rate = 0.1, start = 5000, end = 4000, overlap = T)
# this will
gf <- gene_flow(from = ana, to = eur, rate = 0.1, start = 3000, end = 2000, overlap = T)
```

The main difference here however is that we have now added an `overlap` argument. This is basically a requirement that populations must spatially overlap in order to to exchange genes.

With this done, we can then compile our model. The principle here is the same as with our non-spatial model but with some additional arguments. We will learn about these after we have run the command. Also, ensure population names are correct!

```r
# compile model
model_dir <- paste0(tempfile(), "_tutorial-model")


model <- compile_model(
  populations = list(eur, ana), # populations defined above
  gene_flow = gf, # gene-flow events defined above
  generation_time = 30,
  resolution = 100e3, # resolution in meters per pixel
  competition = 130e3, mating = 100e3, # spatial interaction in SLiM
  dispersal = 700e3, # how far will offspring end up from their parents
  path = model_dir
)

plot_model(model, proportions = T) # to check
```

So what additional arguments have we added here that differs from our previous model compliation in the non-spatial examples?

- Firstly we have the `resolution` argument. This is simply the resolution of the map to simulate on and how much a single pixel represents. Here we have set it to 100,000 units.
- Next we have `competition` - this is the maximum distance between two individuals where they can influence each others fitness via competition. Here it is set to 130,000

which basically menas individuals influence each other only if the occur right next to one another on the map.
- We also have `mating` - this sets the mating choice distance, i.e. the maximum distance an individual will find a mate over; set to 100,000 here, it means individuals look for mates in close proximity.
- Last we have `dispersal` which is fairly self-explanatory as dispersal distance. In the context of our model, it determines how far an individual can move before contributing to next generation.

Note that we also specify a `path` to set a model directory, just so we can look at the files SLiM writes to the directory should we need to.

Finally we use `plot_model` to make sure the model is doing what we expect. If we are satisfied with this, we can now run it using `slim`. Note that this is a large difference from our non-spatial models which used `msprime`. SliM is a forward in time simulator which allows it to incorporate selection and spatial dynamics. It is extremely powerful and I would strongly recommend you investigate it in more detail!

However one disadvantage of SLiM is that it is slower than `msprime` - this menas our simulation will take a bit longer to complete… but not too long!

```
# set locations file
locations_file <- tempfile(fileext = ".gz")
# run the simulations
ts <- slim(model, sequence_length = 1000, recombination_rate = 0, method = "batch",
           locations = locations_file)
# look at the tree sequence
ts
```

And with that, we're done. Hopefully this has given you a taste of what is possible with `slendr`. There is so much more you could do and many ways to extend and expand these models. It is well worth spending some time with this excellent R package!

Optional extra - animating your model

This actually doesn't work on our RStudio Server **but** it should work locally. This is just an optional example that allows you to see some of the ways `slendr` allows you to explore your simulations and models.

```
# animate the model
animate_model(model = model, file = locations_file, steps = 50, width = 700, height = 400)
```