Simulations in population genetics

PopGen 2022 — Martin Petr (mp@bodkan.net)

> Many problems in population genetics cannot be solved by a mathematician, no matter how gifted. [It] is already clear that computer methods are very powerful. This is good. It […] **permits people with limited mathematical knowledge to work on important problems** […]

James F. Crow – interview

Why simulate genomic data?

1. Exploring expected behavior of statistics
2. Fitting model parameters (i.e. ABC)
3. Ground truth for method development

Exploring expected behavior of statistics

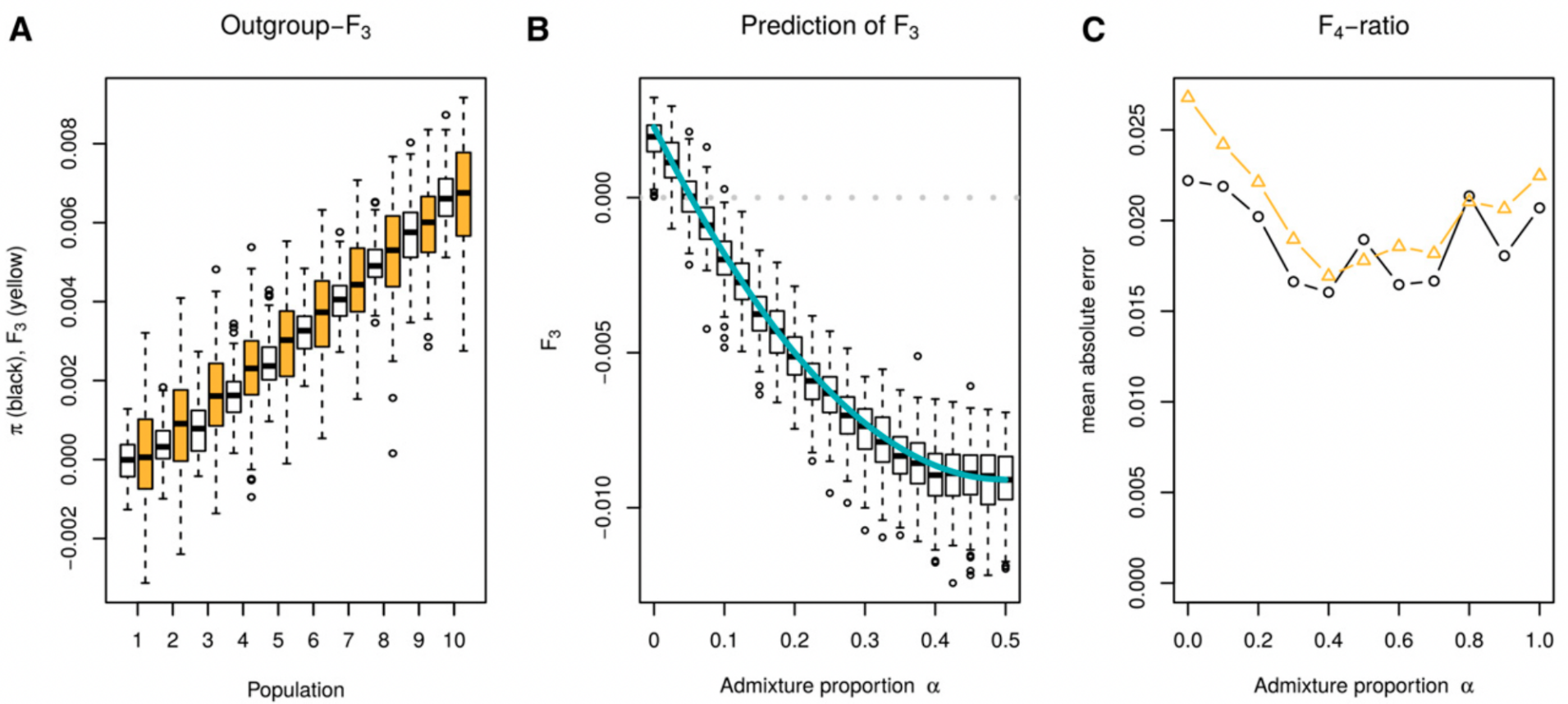


Image from Peter (2016)

Fitting model parameters (i.e. ABC)

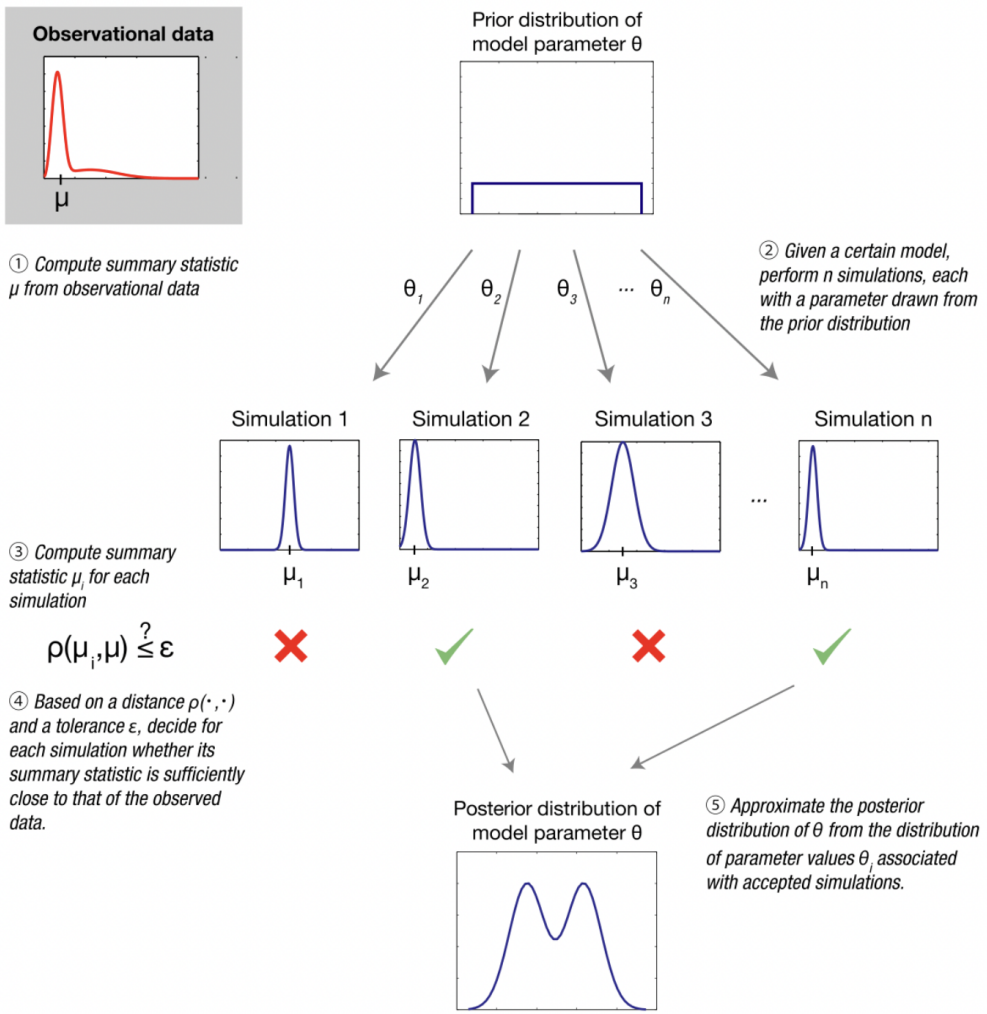


Image from Wikipedia on ABC

Ground truth for testing and method development

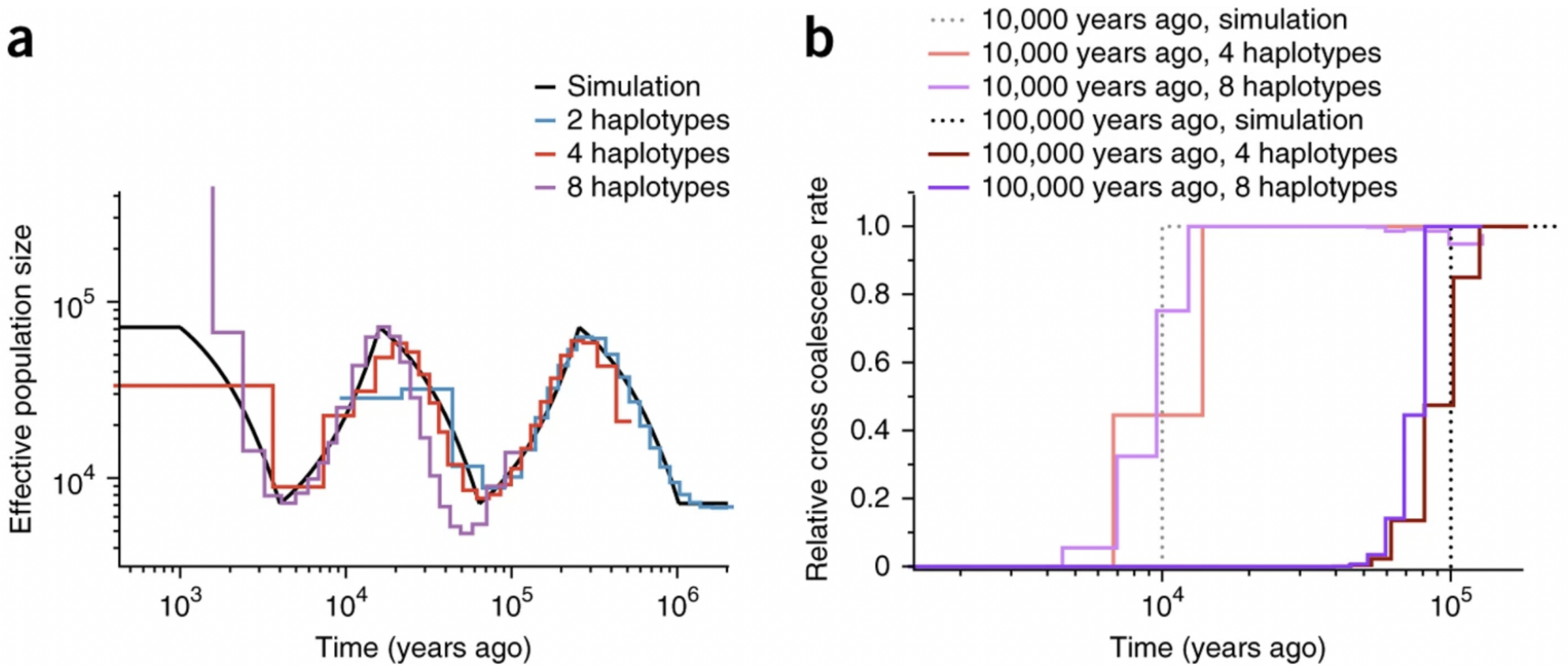


Image from Schiffels and Durbin (2014)

What does it mean to simulate a genome?

How would you design an algorithm for a popgen simulation?

What minimum components are needed for the program to be useful?

If we want to simulate population genetics

We need *populations*.

We need *genetics*.

A chromosome is…

…a linear sequence of nucleotides…

- a list of characters (A/G/C/T nucleotides)
- a list of 0 or 1 values (ancestral/derived allele)

… which accumulates mutations every generation at a given *mutation rate*.

A population is…

A collection of *individuals* at a given point in time.

Each individual carrying two homologous *chromosomes* inherited from two parents in the previous generation.

Chromosomes recombine at a certain *recombination rate*.

Home-brewed single-locus simulations in R

Let’s make the algorithm even more minimal by focusing on the evolution of a single mutation across time (i.e. no mutation process, no recombination).

“What is the expected trajectory of an allele under the influence of genetic drift?”

Some basic setup

If you want to try the following couple of examples yourself, you will need to paste the following lines into your R session (either R in the terminal, or better an RStudio session):

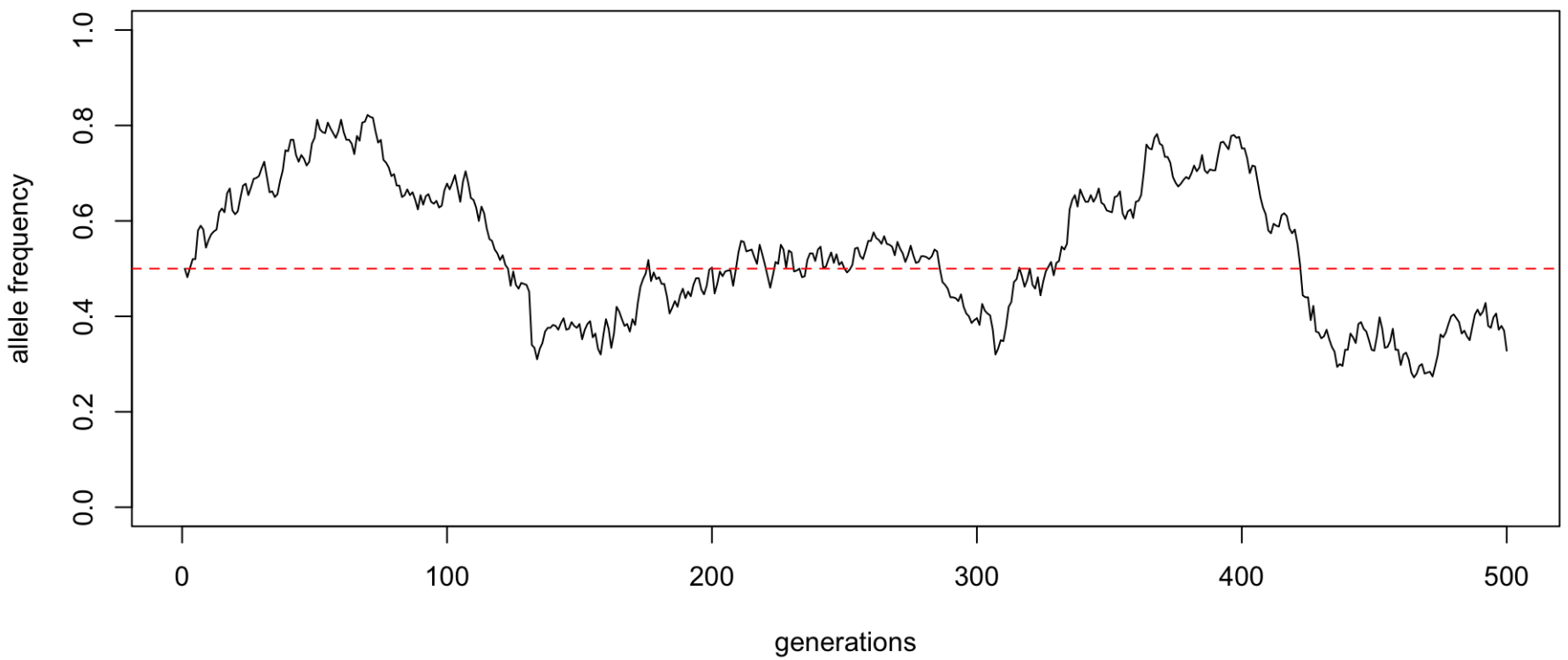
```
library(tidyverse)
library(parallel)
library(MASS)

set.seed(42)
```

Single-locus simulation

```r
N <- 500 # number of (haploid) individuals in the population
generations <- 500 # number of generations to run the simulation for
p_start <- 0.5 # initial allele frequency

# initialize an (empty) trajectory vector of allele frequencies
p_traj<- rep(NA, generations)
p_traj[1] <- p_start

# in each generation...
for (gen_i in 2:generations) {
  p <- p_traj[gen_i - 1] # get the current frequency

  # ... calculate the allele frequency in the next generation ...
  p_next <- rbinom(1, N, p) / N

  # ... and save it to the trajectory vector
  p_traj[gen_i] <- p_next
}
```

$N = 500, p_0 = 0.5$

- Code



Let's make it a function

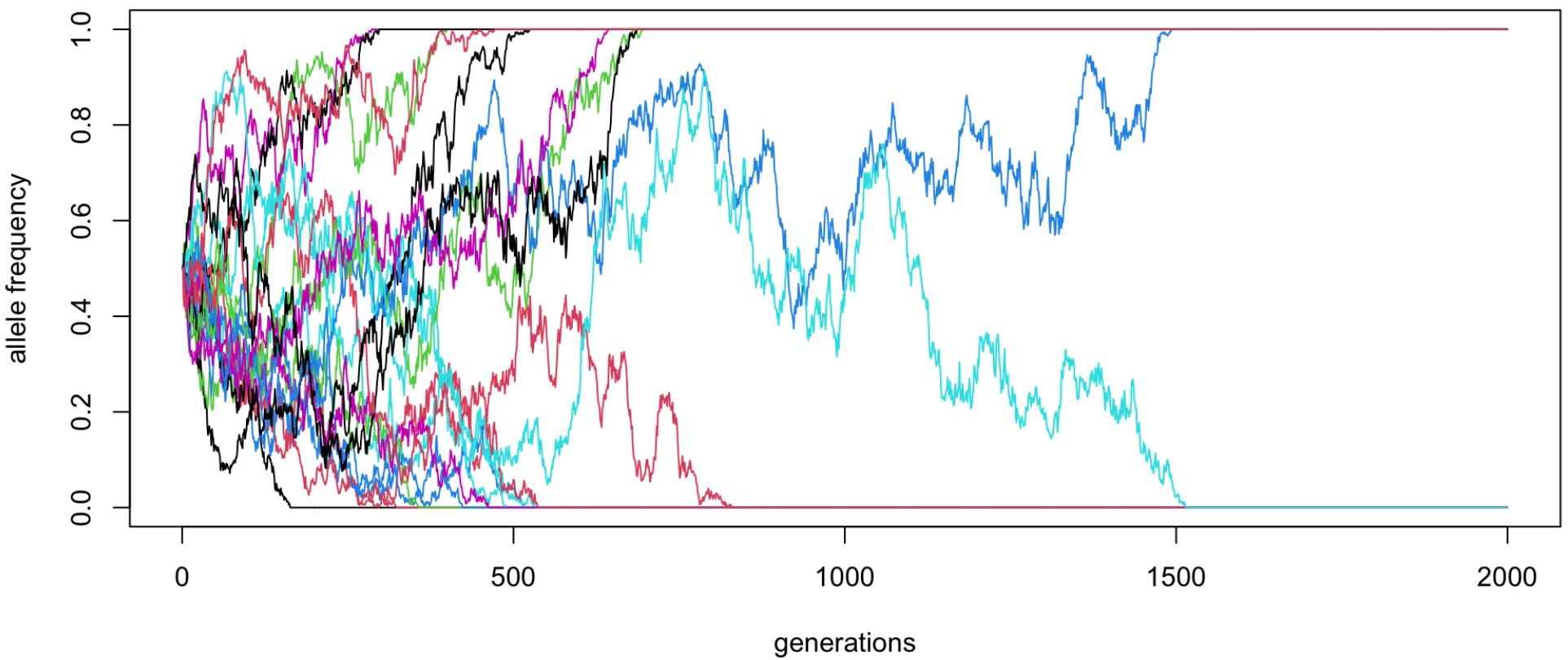
Input: N, p_0 and the number of generations

Output: allele frequency trajectory vector

```
simulate <- function(N, p_start, generations) {
  # initialize an (empty) trajectory vector of allele frequencies
  p_traj<- rep(NA, generations)
  p_traj[1] <- p_start

  # in each generation...
  for (gen_i in 2:generations) {
    p <- p_traj[gen_i - 1] # get the current frequency
    # ... calculate the allele frequency in the next generation ...
    p_next <- rbinom(1, N, p) / N
    # ... and save it to the trajectory vector
    p_traj[gen_i] <- p_next
  }

  p_traj
}
```

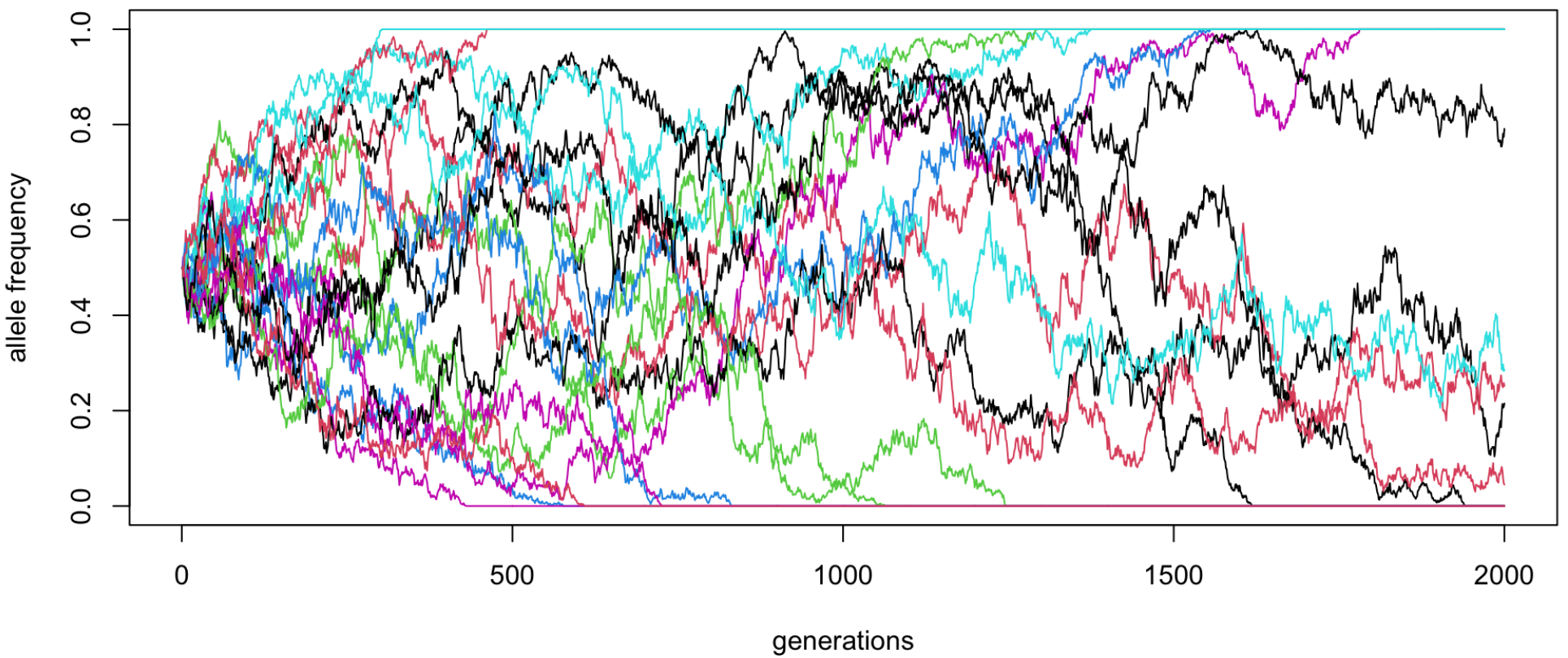
$N = 500, p_0 = 0.5$ (20 replicates)

- Code



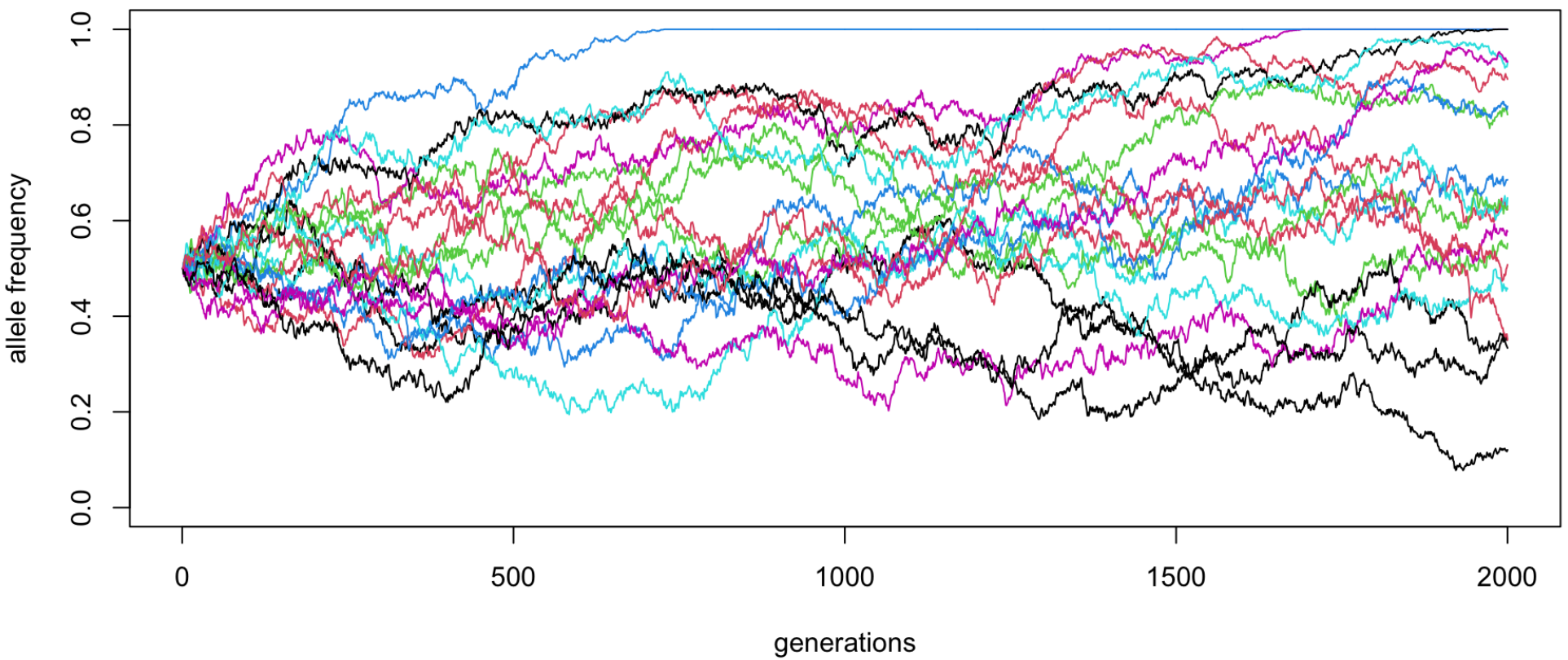
$N = 1000, p_0 = 0.5$ (20 replicates)

- Code



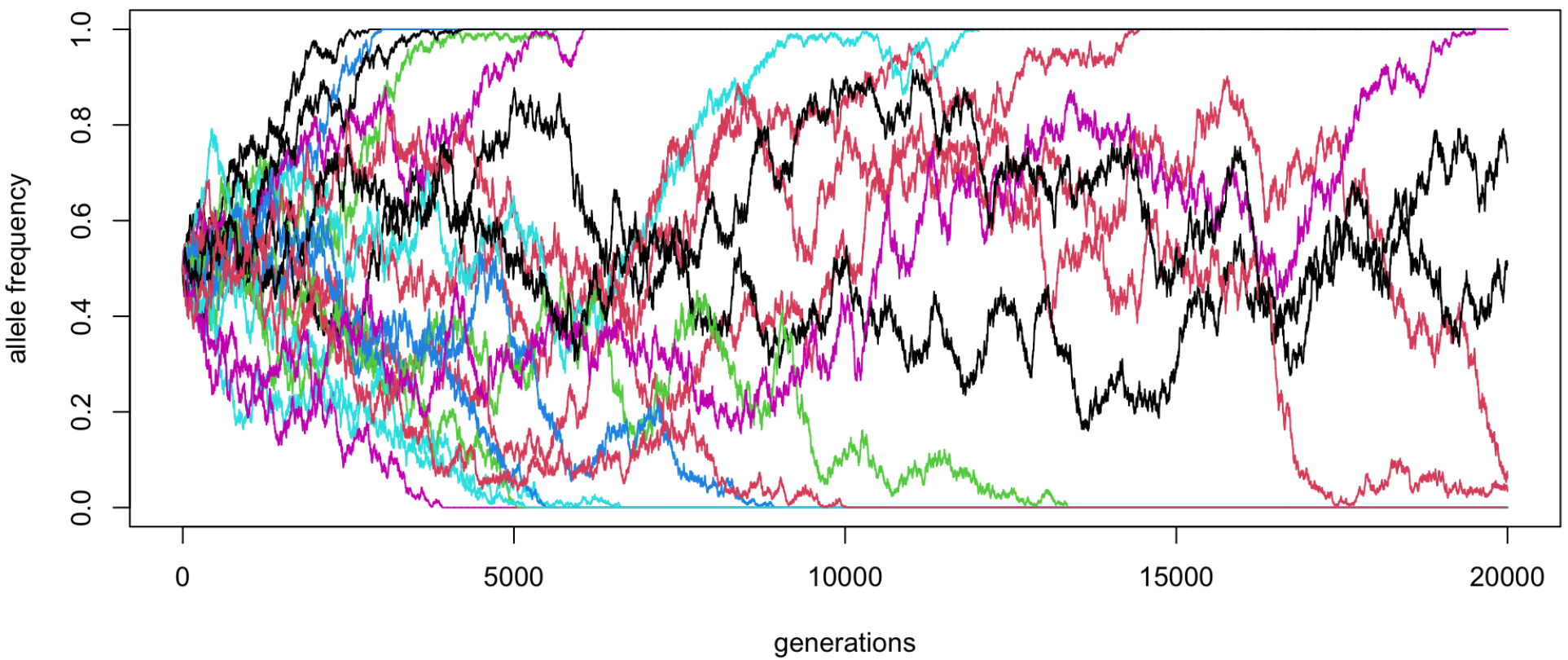
$N = 5000, p_0 = 0.5$ (20 replicates)

- Code



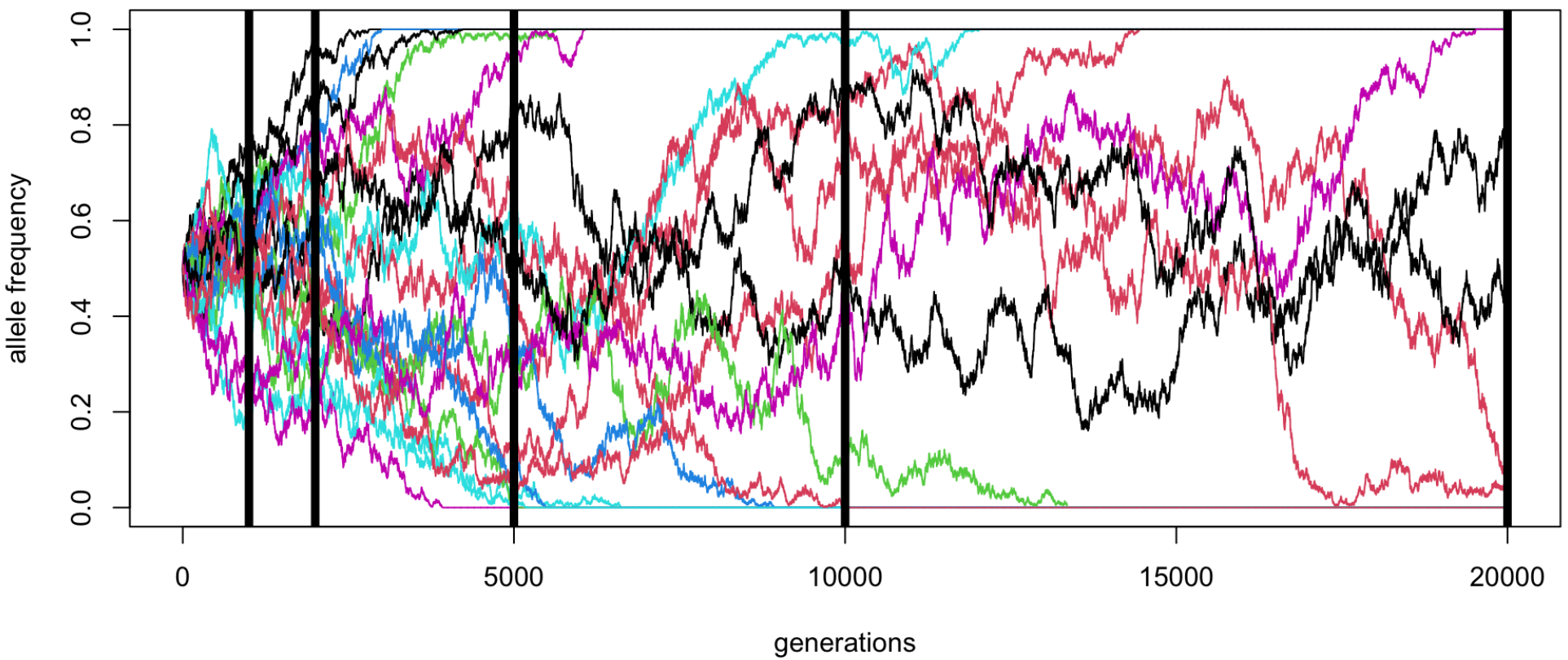
$N = 10000, p_0 = 0.5$ (20 replicates)

- Code



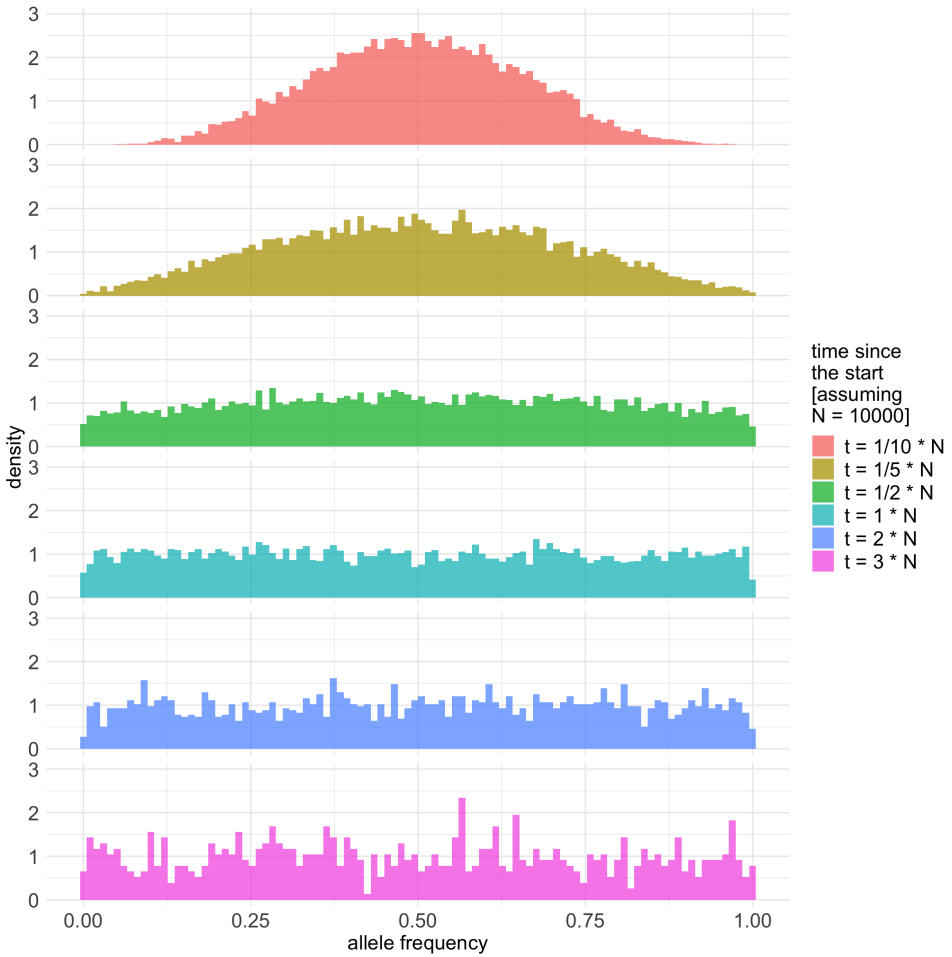
$N = 10000, p_0 = 0.5$ (20 replicates)

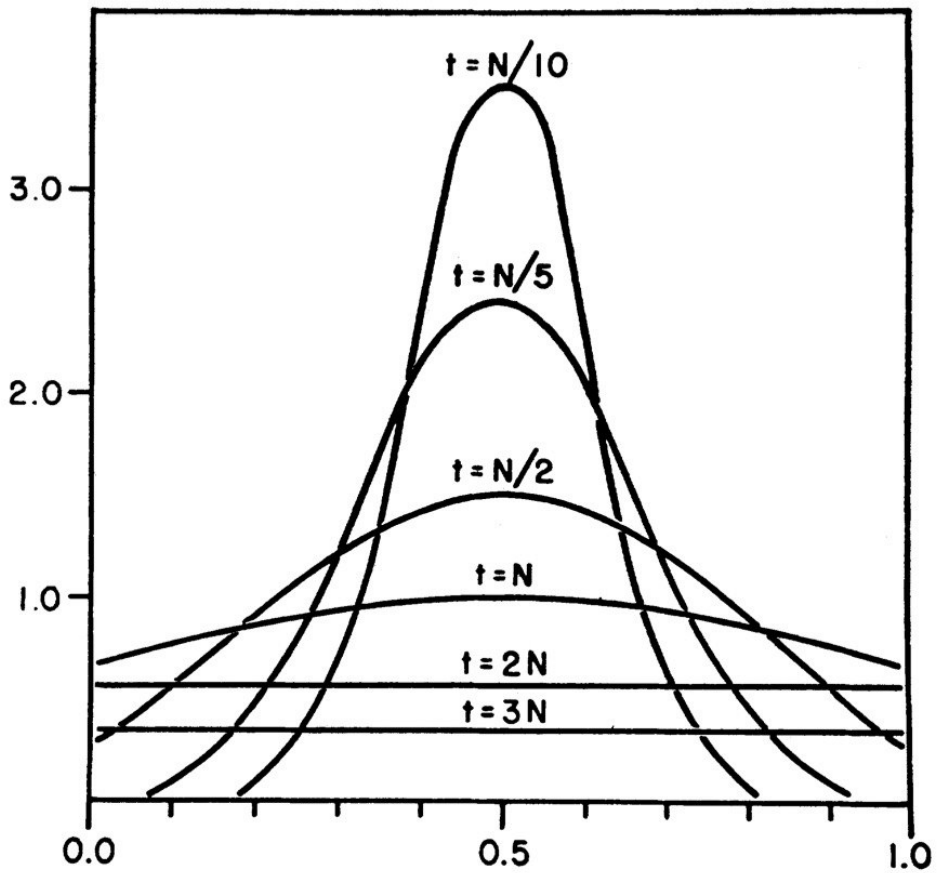
- Code



Expected allele frequency distribution

Code





"Diffusion Models in Population Genetics", Kimura (1964)

Bonus exercise #1

If you're bored later, **try adding selection** to this model!

In each generation, weight the frequencies based on fitness of each of the tree genotypes.

If you need a hint, take a look here.


But now for something completely different.

Let’s do “real” simulations!

There are many pieces of simulation software

The most famous and widely used are SLiM and msprime.

They are both extremely powerful…

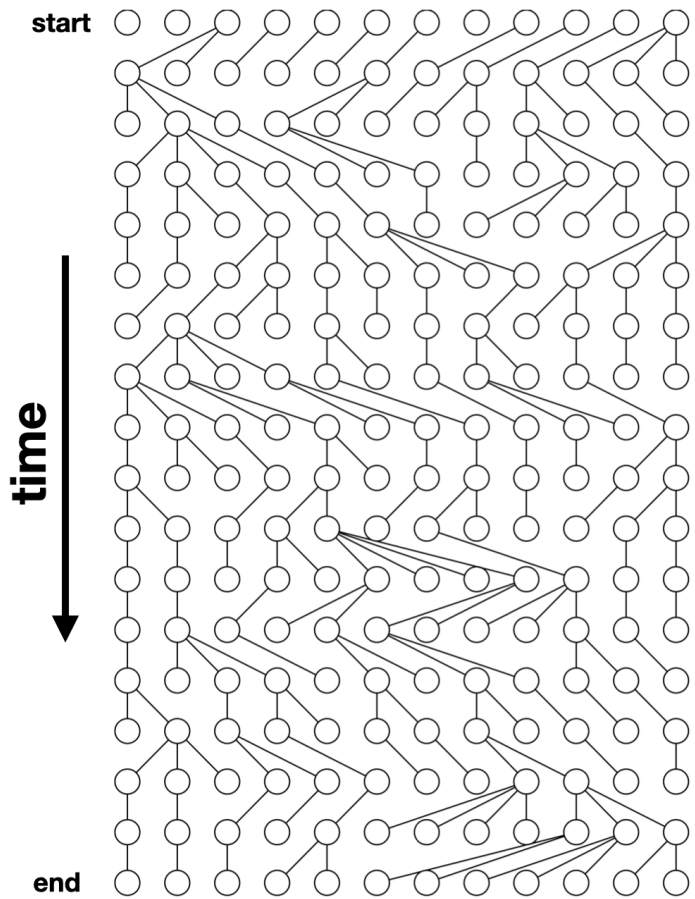
… but they require a lot of programming knowledge and quite a lot of code to write non-trivial simulations (without bugs).

We will learn simulation concepts using *slendr*,
a convenient R interface to both SLiM and msprime.

SLiM

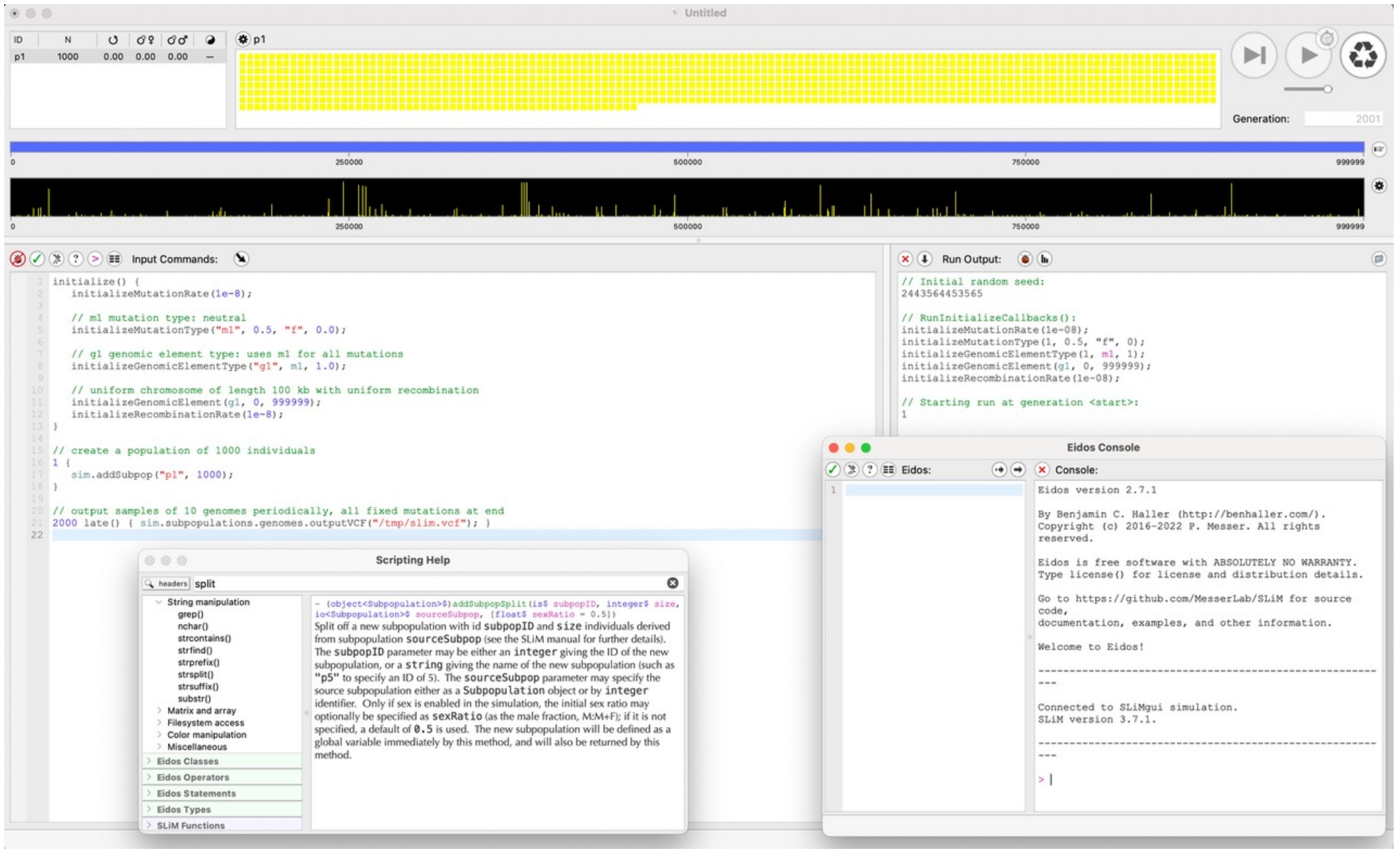
What is SLiM?

- **Forward-time simulator**
- It’s fully programmable!
- Massive library of functions for:
  - Demographic events
  - Various mating systems
  - Selection, quantitative traits, …
- > 700 pages long manual!



Modified from Alex Drummond

SLiMgui – IDE for SLiM

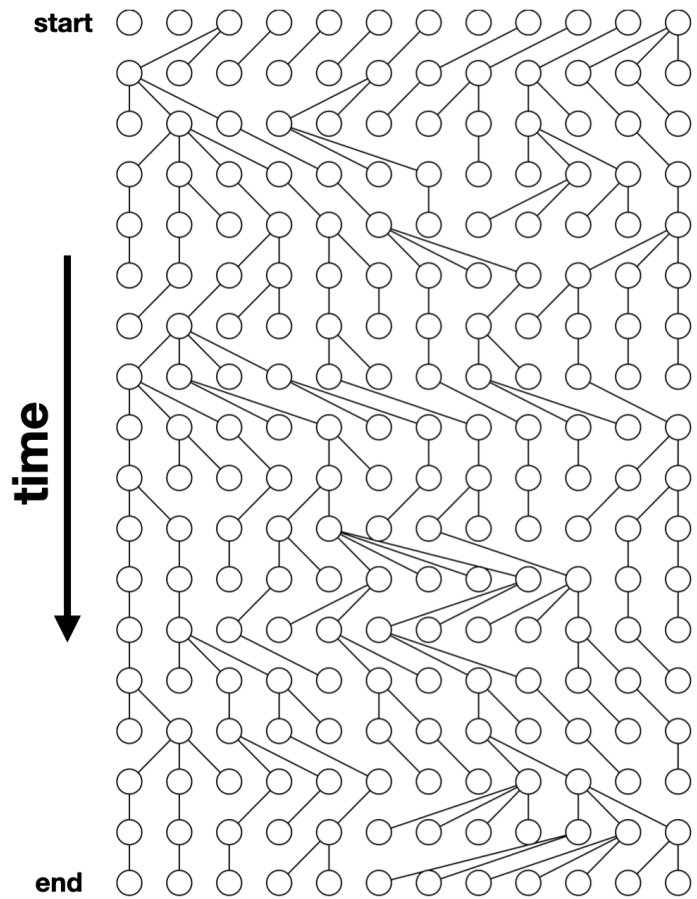


Simple neutral simulation in SLiM

```
initialize() {
    // create a neutral mutation type
    initializeMutationType("m1", 0.5, "f", 0.0);
    // initialize 1Mb segment
    initializeGenomicElementType("g1", m1, 1.0);
    initializeGenomicElement(g1, 0, 999999);
    // set mutation rate and recombination rate of the segment
    initializeMutationRate(1e-8);
    initializeRecombinationRate(1e-8);
}

// create an ancestral population p1 of 10000 diploid individuals
1 early() { sim.addSubpop("p1", 10000); }

// in generation 1000, create two daughter populations p2 and p3
```

msprime

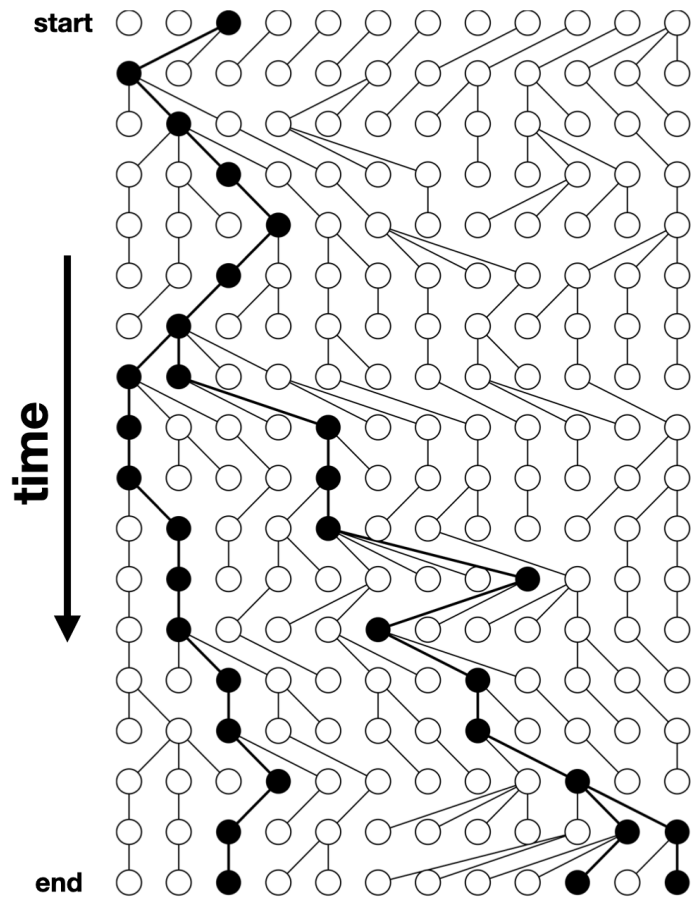
What is msprime?



Modified from Alex Drummond

What is msprime?

- A Python module for writing **coalescent simulations**
- Extremely fast (genome-scale, population-scale data)
- You must know Python fairly well to build complex models



Modified from Alex Drummond

Simple simulation using *msprime*

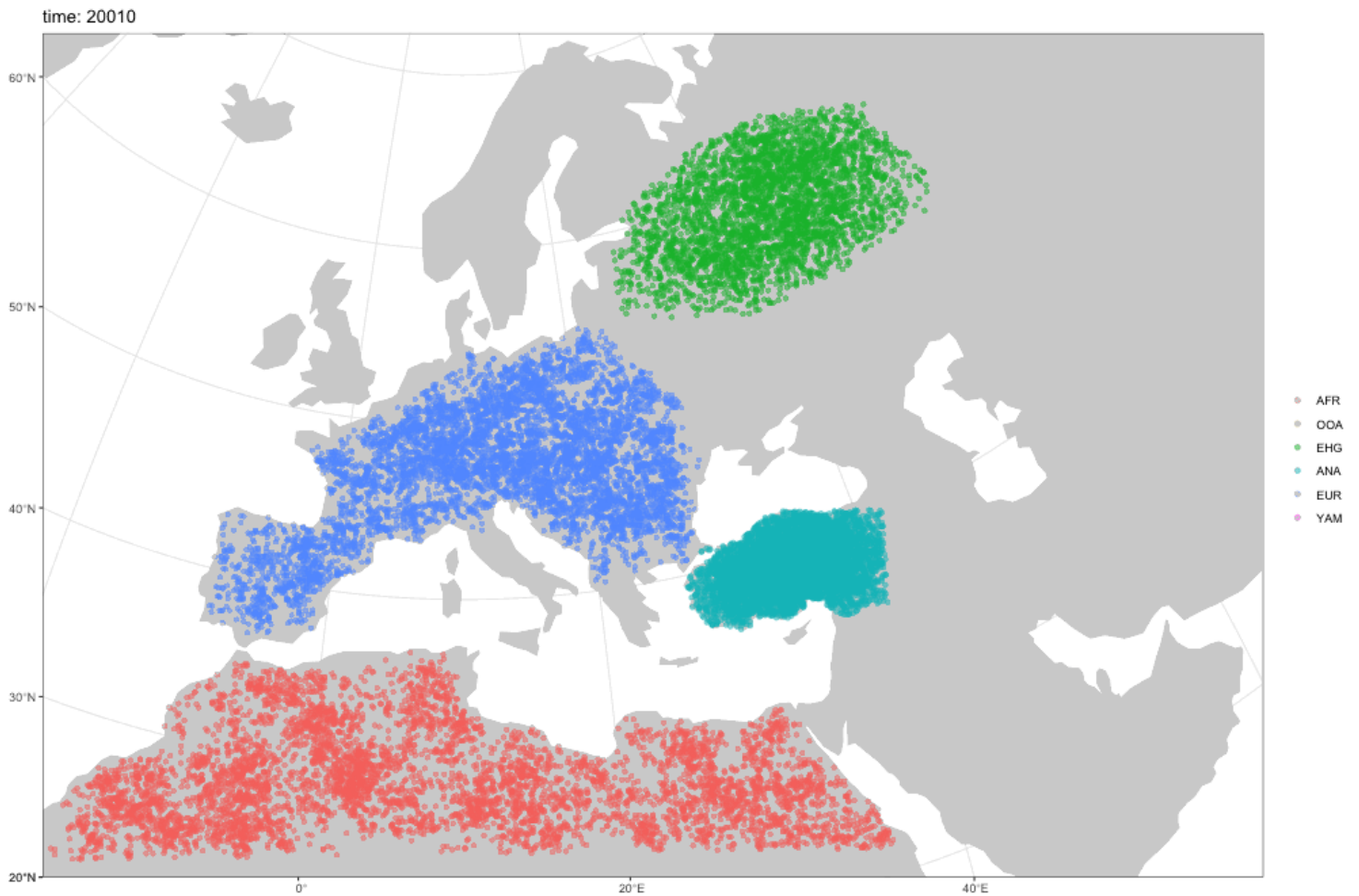
The following is basically the same model as the SLiM script earlier:

```
import msprime

demography = msprime.Demography()
demography.add_population(name="A", initial_size=10_000)
demography.add_population(name="B", initial_size=5_000)
demography.add_population(name="C", initial_size=1_000)
demography.add_population_split(time=1000, derived=["A", "B"], ancestral="C")

ts = msprime.sim_ancestry(
  sequence_length=10e6,
  recombination_rate=1e-8,
  samples={"A": 100, "B": 100},
  demography=demography
)
```

source: link

www.slendr.net

Why a new package? – spatial simulations!



Why a new package?

- Most researchers are not expert programmers
- All but the most trivial simulations require lots of code
- 90% of simulations are basically the same!
  - create populations (splits and N_e changes)
  - specify if/how they should mix (rates and times)
  - save output (VCF, EIGENSTRAT)
- Lot of code duplication across projects

***slendr* makes "standard" simulations trivial (for newbies *and* experts) and unlocks new kinds of spatial simulations**

Let’s get started

We will use `slendr` & `tidyverse`

First run this:

```
library(tidyverse) # table processing/filtering and plotting

library(slendr) # simulation and tree-sequence analysis
```

(ignore the message about missing SLiM)

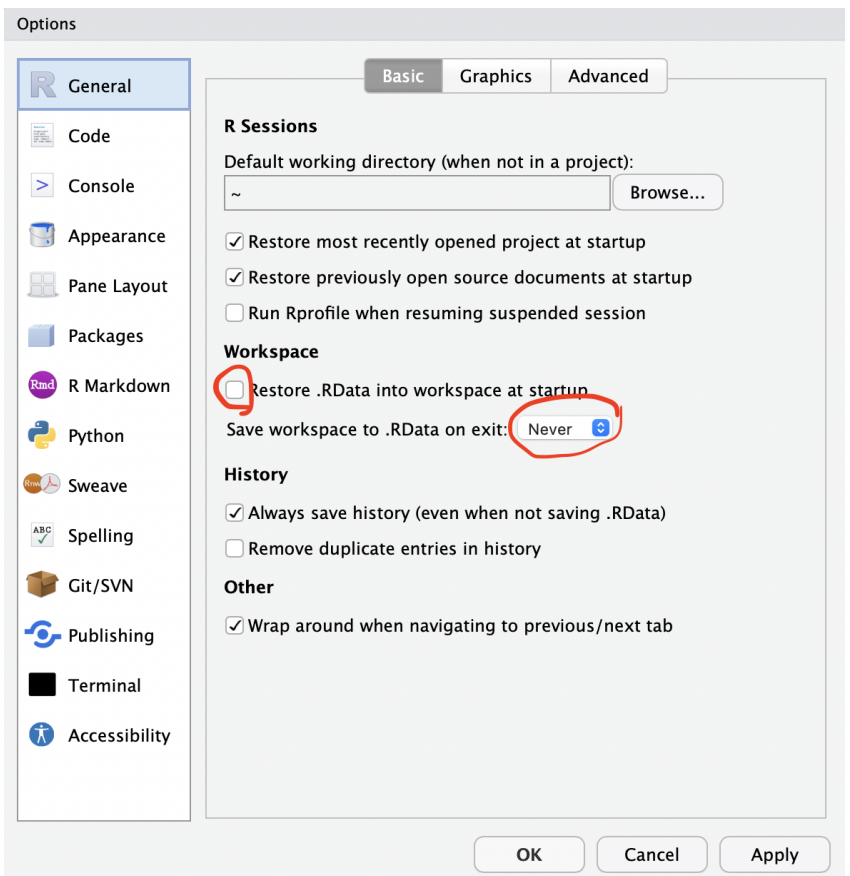
Then run this (and wait for the Python installation to finish!):

```
setup_env()
```

The entire lecture & exercises will be in R!

Workaround for an RStudio bug

RStudio sometimes interferes with Python setup that we need for simulation. To fix this, go to `Tools` -> `Global Options` in your RStudio and set the following options:



slendr workflow:

Build models from simple components

Then simulate data from them

All within a single R script

Typical steps

1. creating populations
2. scheduling population splits
3. programming N_e size changes
4. encoding gene-flow events
5. simulation sequence of a given size
6. computing statistics from simulated outputs

Creating a population

A name, size and the time of appearance must be given:

```
pop1 <- population("pop1", N = 1000, time = 1)
```

Typing an object prints out a summary in the R console:

```
pop1
```

```
slendr 'population' object
--------------------------
name: pop1
non-spatial population
stays until the end of the simulation

population history overview:
  - time 1: created as an ancestral population (N = 1000)
```

Programming population splits

Splits are indicated by the `parent = <pop>` argument:

```
pop2 <- population("pop2", N = 100, time = 50, parent = pop1)
```

The split is reported in the “historical summary”:

```
pop2
```

```
slendr 'population' object
--------------------------
name: pop2
non-spatial population
stays until the end of the simulation

population history overview:
  - time 50: split from pop1
```

Scheduling resize events – `resize()`

Step size decrease:

```
pop1 <- population("pop1", N = 1000, time = 1)
pop1_step <- resize(pop1, N = 100, time = 500, how = "step")
```

Exponential increase:

```
pop2 <- population("pop2", N = 100, time = 50, parent = pop1)
pop2_exp <- resize(pop2, N = 10000, time = 500, end = 2000, how = "exponent
```

Tidyverse-style pipe interface

Step size decrease:

```
pop1 <-
  population("pop1", N = 1000, time = 1) %>%
  resize(N = 100, time = 500, how = "step")
```

Exponential increase:

```
pop2 <-
  population("pop2", N = 1000, time = 1) %>%
  resize(N = 10000, time = 500, end = 2000, how = "exponential")
```

This accomplishes the same thing as the code on the previous slide, but it is a bit more “elegant”.

More complex full model

```
pop1 <- population("pop1", N = 1000, time = 1)

pop2 <-
  population("pop2", N = 1000, time = 300, parent = pop1) %>%
  resize(N = 100, how = "step", time = 1000)

pop3 <-
  population("pop3", N = 1000, time = 400, parent = pop2) %>%
  resize(N = 2500, how = "step", time = 800)

pop4 <-
  population("pop4", N = 1500, time = 500, parent = pop3) %>%
  resize(N = 700, how = "exponential", time = 1200, end = 2000)

pop5 <-
  population("pop5", N = 100, time = 600, parent = pop4) %>%
  resize(N = 50, how = "step", time = 900) %>%
  resize(N = 250, how = "step", time = 1200) %>%
  resize(N = 1000, how = "exponential", time = 1600, end = 2200) %>%
```

Remember: each object carries its history

```
pop5
```

```
slendr 'population' object
--------------------------
name: pop5
non-spatial population
stays until the end of the simulation

population history overview:
  - time 600: split from pop4
  - time 900: resize from 100 to 50 individuals
  - time 1200: resize from 50 to 250 individuals
  - time 1600-2200: exponential resize from 250 to 1000 individuals
  - time 2400: resize from 1000 to 400 individuals
```

Last step before simulation: compilation

```
model <- compile_model(
  list(pop1, pop2, pop3, pop4, pop5),
  generation_time = 1,
  simulation_length = 3000
)
```

Compilation takes a list of model components, performs internal consistency checks, returns a single model object.

Model summary

Typing the compiled model prints a brief summary:

```
model
```

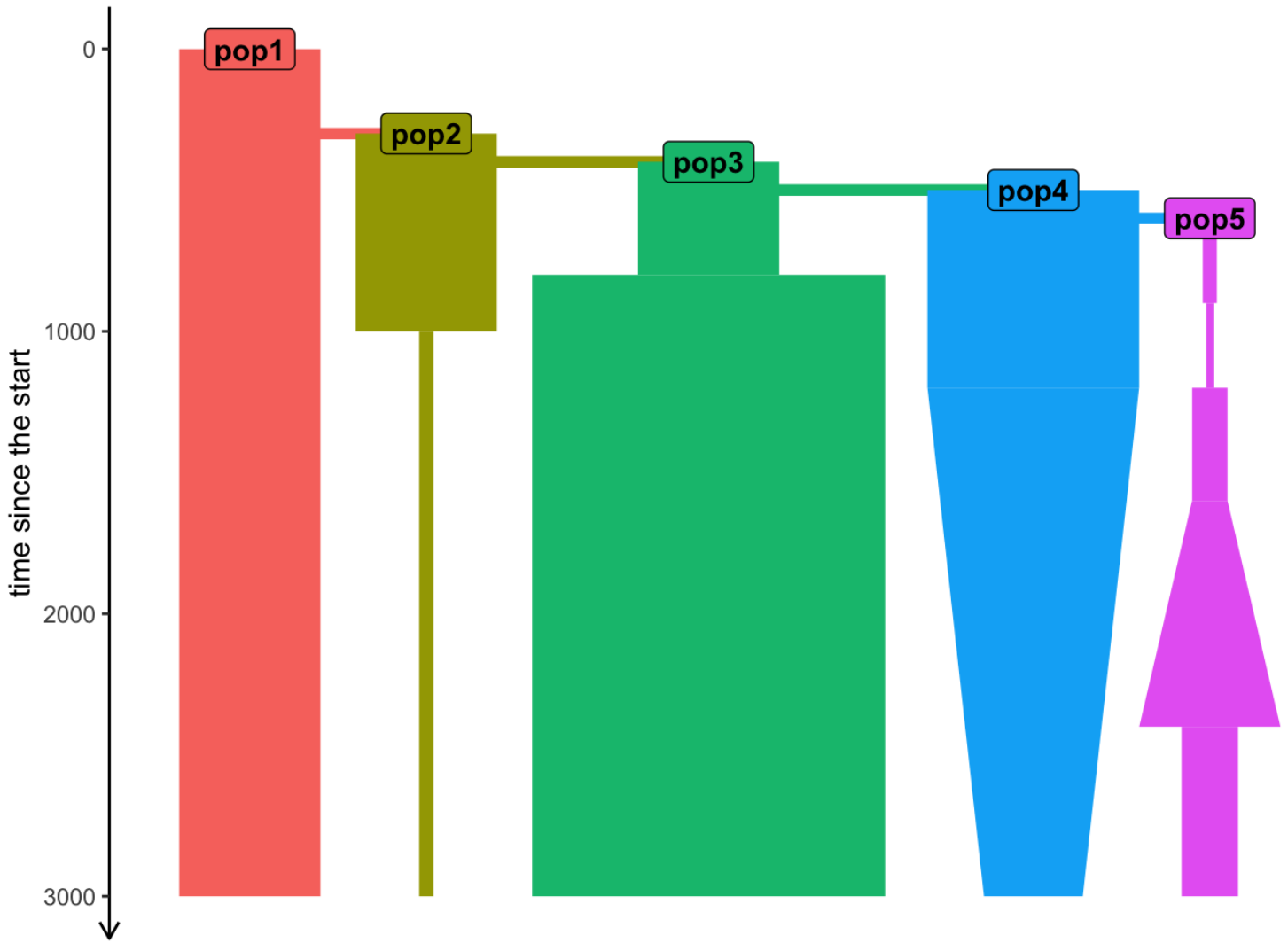
```
slendr 'model' object
---------------------
populations: pop1, pop2, pop3, pop4, pop5
geneflow events: [no geneflow]
generation time: 1
time direction: forward
total running length: 3000 model time units
model type: non-spatial

configuration files in:
/private/var/folders/d_/hblb15pd3b94rg0v35920wd80000gn/T/Rtmpa2Azea/file3874105
```

The model is also saved to disk! (The location can be specified via `path =` argument to `compile_model()`):

```
 [1] "checksums.tsv"       "description.txt"     "direction.txt"
 [4] "generation_time.txt" "length.txt"          "orig_length.txt"
```

Model visualization

```
plot_model(model)
```



Simulating data (finally...)

We have the compiled `model`, how do we simulate data?

slendr has two simulation engines already built-in:

- SLiM engine
- msprime engine

You don't have to write any msprime/SLiM code!

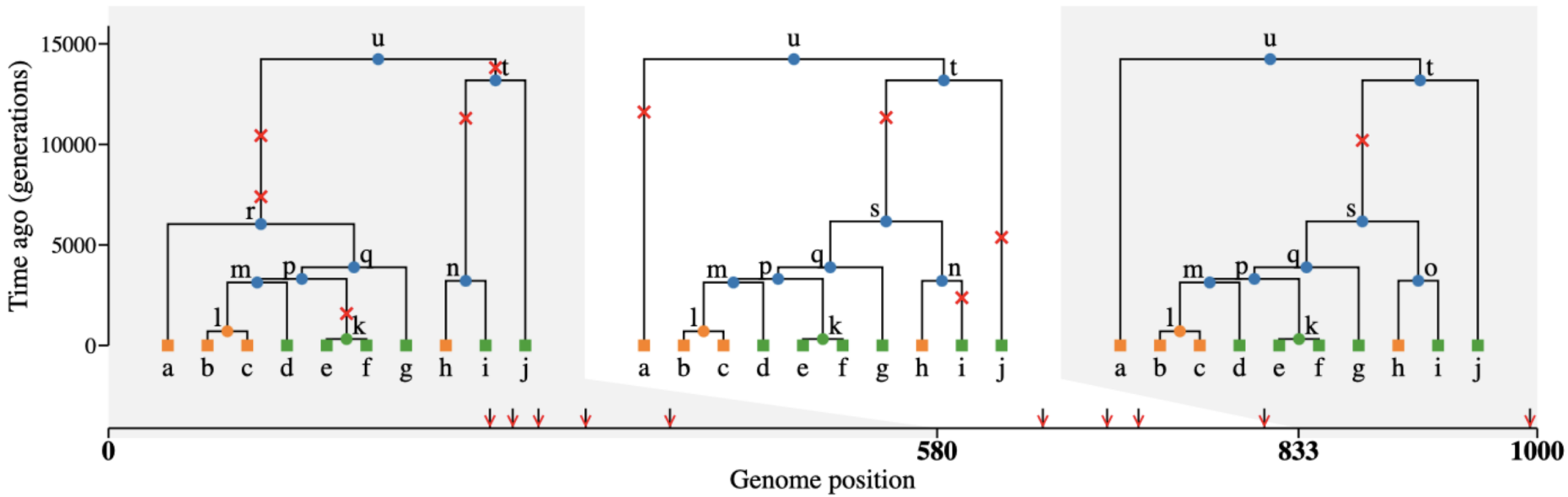
Take a model object and use the built-in simulation engine:

```
ts <- msprime(model, sequence_length = 100e6, recombination_rate = 1e-8)
```

`ts` is a so-called "tree sequence"

The output of a *slendr* simulation is a tree sequence

What is a tree sequence?



- A record of full genetic ancestry of a set of samples
- An encoding of DNA sequence carried by those samples
- An efficient analysis framework

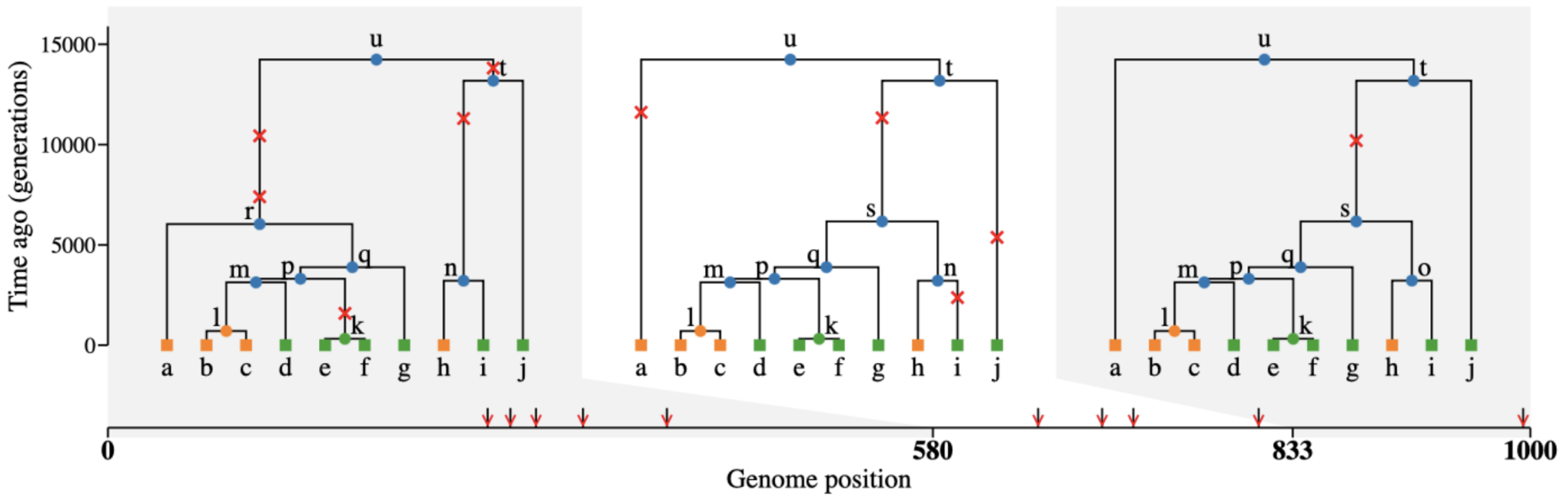
Why a tree sequence?

What we usually have

```
##fileformat=VCFv4.0
##source=BCM:SNPTools:hapfuse
##reference=1000Genomes-NCBI37
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=AP,Number=2,Type=Float,Description="Allelic Probability, P(Allele=1|Haplotype)">
#CHROM  POS     ID      REF     ALT     QUAL    FILTER  INFO    FORMAT  HG00096 HG00097 HG00099 HG00100 HG00101 HG00102
12      60076   .       A       C       100     PASS    .       GT:AP   1|0:1.000,0.000 0|0:0.000,0.005 0|0:0.000,0.005
12      60252   .       A       G       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.300 0|1:0.010,0.590
12      60317   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      60344   .       C       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.020,0.025 0|0:0.005,0.000
12      60383   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      60405   .       T       C       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      60474   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|1:0.015,0.705 0|1:0.020,0.775
12      60614   .       C       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.005 0|0:0.000,0.015
12      60628   .       T       C       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.070 0|0:0.000,0.000
12      60654   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61021   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61107   .       G       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.015 0|0:0.000,0.000
12      61172   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61220   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.005,0.265 0|1:0.020,0.665
12      61258   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.005,0.465 0|1:0.020,0.895
12      61272   .       T       C       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61329   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61341   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|1:0.010,0.560 0|1:0.020,0.855
12      61368   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|1:0.020,0.630 0|1:0.020,0.955
12      61392   .       T       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61405   .       G       C       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61411   .       C       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.015
12      61416   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.025 0|0:0.000,0.010 0|0:0.015,0.075
12      61422   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.105 0|0:0.005,0.010
12      61476   .       C       G       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.005,0.000
12      61510   .       G       A       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61516   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.005 0|0:0.000,0.000
12      61552   .       C       T       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61604   .       T       G       100     PASS    .       GT:AP   0|0:0.000,0.000 0|0:0.000,0.000 0|0:0.000,0.000
12      61687   .       G       A       100     PASS    .       GT:AP   1|0:1.000,0.000 0|1:0.015,0.625 0|1:0.020,0.960
12      61700   .       C       T       100     PASS    .       GT:AP   0|0:0.005,0.000 0|0:0.000,0.035 0|0:0.025,0.060
```

What we usually *want*

(As full as possible) a representation of our samples' history:



And this is exactly what tree sequences give us.

How does it work?

This simulates 20 × 10000 chromosomes of 100 Mb.

In less than 30 seconds.

That's a crazy amount of data!

And it only requires 66.1 Mb of memory!

```
ts <-
  population("pop", time = 100000, N = 10000) %>%
  compile_model(generation_time = 1, direction = "backward") %>%
  msprime(sequence_length = 100e6, recombination_rate = 1e-8)

ts
```

TreeSequence	
Trees	393469
Sequence Length	100000000
Time Units	generations

Sample Nodes	20000
Total Size	66.4 MiB

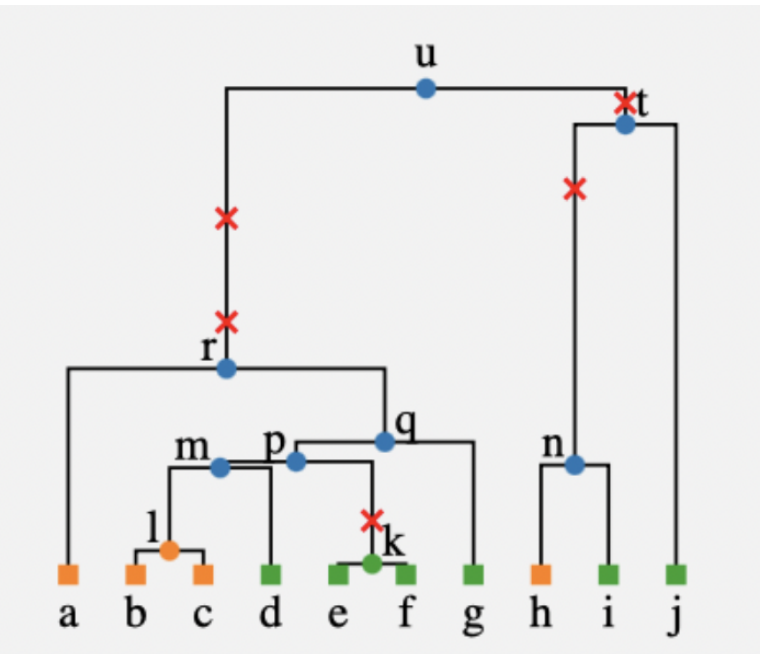
Table	Rows	Size	Has Metadata

How does this work?!

Tree-sequence tables (tskit docs)

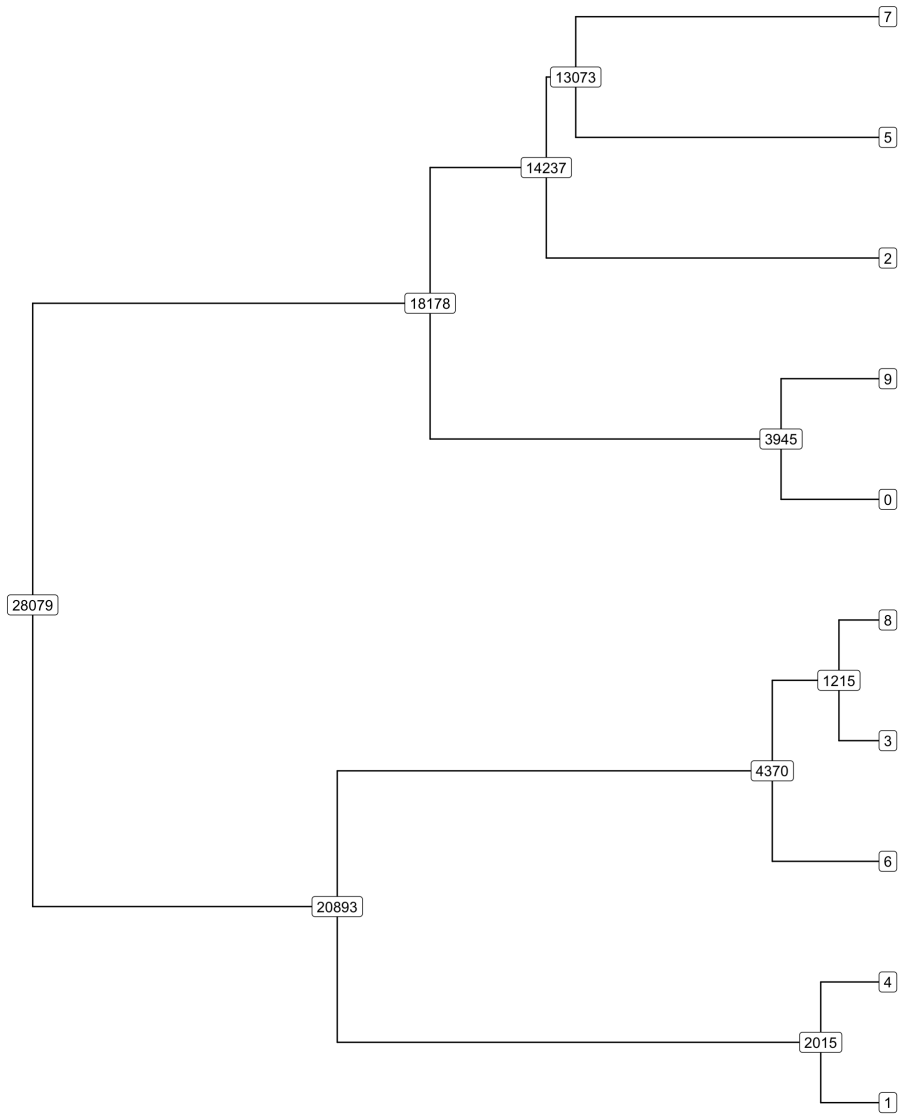
A tree (sequence) can be represented by tables of:

- nodes (“chromosomes”)
- edges (branches) between nodes
- mutations on edges
- individuals (who carry nodes)
- populations



A set of such tables is a tree sequence.

Tree-sequence tables in practice



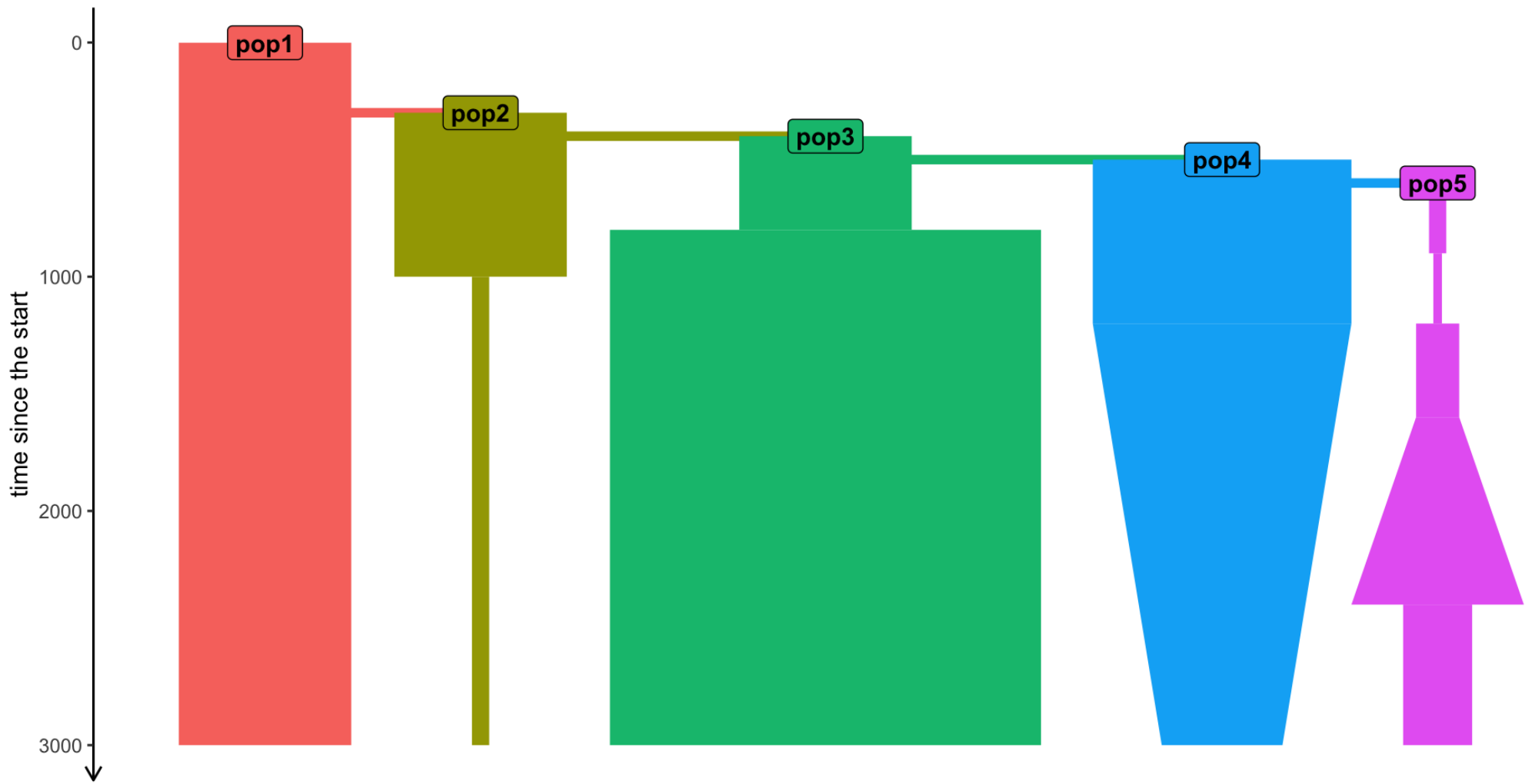
▶ Code

```
# A tibble: 3 × 4
  name  pop   node_id  time
  <chr> <fct>   <int> <dbl>
1 pop_1 pop         0     0
2 pop_1 pop         1     0
3 pop_2 pop         2     0
```

▶ Code

```
# A tibble: 3 × 2
  child_node_id parent_node_id
          <int>          <int>
1             0          29163
2             0          30352
3             0          36004
```

Let’s get back to the model we defined earlier



Simulating a tree-sequence output

By default, `msprime` function automatically loads the tree-sequence that the simulation produced in the background:

```
ts <- msprime(
  model,
  sequence_length = 100e6,
  recombination_rate = 1e-8
)
```

If we type `ts` into an R console, we get…

… a tree-sequence content summary

```
ts
```

TreeSequence	
Trees	140813
Sequence Length	100000000
Time Units	generations
Sample Nodes	9400
Total Size	24.6 MiB

Table	Rows	Size	Has Metadata

What can we do with this?

R interface to tskit

Tree sequence loading and processing

`ts_load()`	Load a tree sequence file produced by a given model
`ts_save()`	Save a tree sequence to a file
`ts_recapitate()`	Recapitate the tree sequence
`ts_simplify()`	Simplify the tree sequence down to a given set of individuals
`ts_mutate()`	Add mutations to the given tree sequence
`ts_coalesced()`	Check that all trees in the tree sequence are fully coalesced
`ts_samples()`	Extract names and times of individuals of interest in the current tree sequence (either all sampled individuals or those that the user simplified to)

Tree sequence format conversion

`ts_genotypes()`	Extract genotype table from the tree sequence
`ts_eigenstrat()`	Convert genotypes to the EIGENSTRAT file format
`ts_vcf()`	Save genotypes from the tree sequence as a VCF file

Accessing tree sequence components

`ts_nodes()`	Extract combined annotated table of individuals and nodes
`ts_edges()`	Extract spatio-temporal edge annotation table from a given tree or tree sequence
`ts_table()`	Get the table of individuals/nodes/edges/mutations from the tree sequence
`ts_phylo()`	Convert a tree in the tree sequence to an object of the class `phylo`
`ts_tree()`	Get a tree from a given tree sequence
`ts_draw()`	Plot a graphical representation of a single tree
`ts_metadata()`	Extract list with tree sequence metadata saved by SLiM
`ts_ancestors()`	Extract (spatio-)temporal ancestral history for given nodes/individuals
`ts_descendants()`	Extract all descendants of a given tree-sequence node

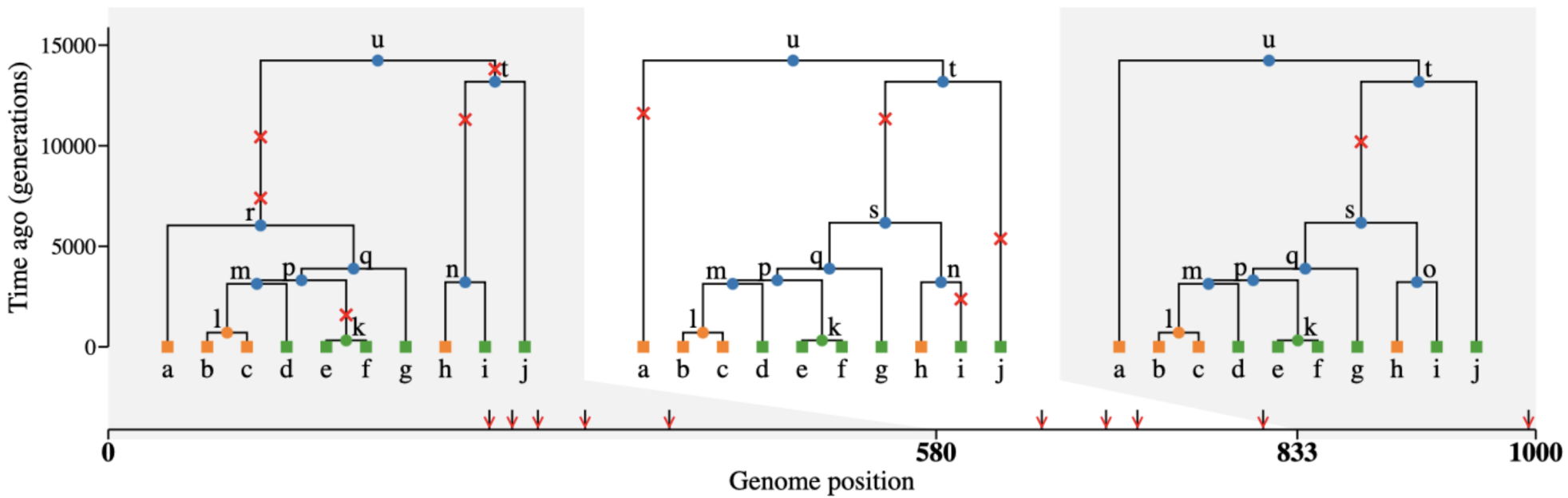
Tree sequence statistics

`ts_f2()` `ts_f3()` `ts_f4()` `ts_f4ratio()`	Calculate the f2, f3, f4, and f4-ratio statistics
`ts_afs()`	Compute the allele frequency spectrum (AFS)
`ts_divergence()`	Calculate pairwise divergence between sets of individuals
`ts_diversity()`	Calculate diversity in given sets of individuals
`ts_fst()`	Calculate pairwise statistics between sets of individuals
`ts_tajima()`	Calculate Tajima's D for given sets of individuals
`ts_segregating()`	Calculate the density of segregating sites for the given sets of individuals

This R interface links to Python methods implemented in *tskit*.

Here is the magic

Tree sequences make it possible to directly compute many quantities of interest *without going via conversion to a genotype table/VCF*!



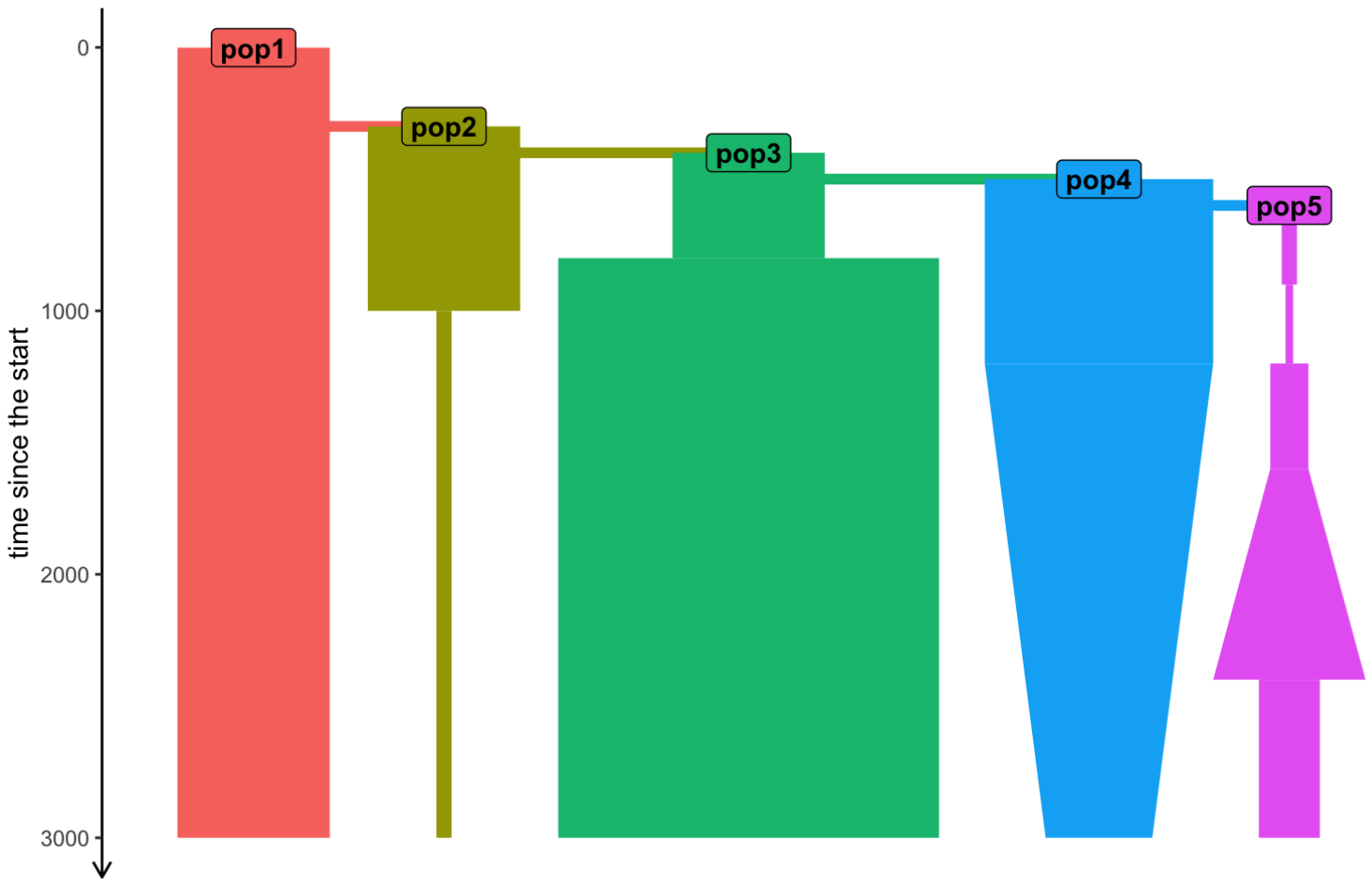
Ralph et al. (2020)

How do we use this in practice?

To analyze tree sequences, we need to refer to “samples”

Extracting sample information

Each “sampled” individual in *slendr* has a symbolic name, a sampling time, and a population assignment:



Extracting sample information

Each “sampled” individual in *slendr* has a symbolic name, a sampling time, and a population assignment:

```
ts_samples(ts)
```

```
# A tibble: 4,700 × 3
   name      time pop
   <chr>    <dbl> <chr>
 1 pop1_1       0 pop1
 2 pop1_2       0 pop1
 3 pop1_3       0 pop1
 4 pop1_4       0 pop1
 5 pop1_5       0 pop1
 6 pop1_6       0 pop1
 7 pop1_7       0 pop1
 8 pop1_8       0 pop1
 9 pop1_9       0 pop1
10 pop1_10      0 pop1
# … with 4,690 more rows
# ℹ Use `print(n = ...)` to see more rows
```

```
ts_samples(ts) %>% count(pop)
```

```
# A tibble: 5 × 2
  pop       n
  <chr> <int>
1 pop1   1000
2 pop2    100
3 pop3   2500
4 pop4    700
5 pop5    400
```

Analyzing tree sequences with *slendr*

Let's say we have the following model and we simulate a tree sequence from it.

```
pop <- population("pop", N = 10000, time = 1)
model <- compile_model(pop, generation_time = 1, simulation_length = 10000)

ts <- msprime(model, sequence_length = 100e6, recombination_rate = 1e-8)
```

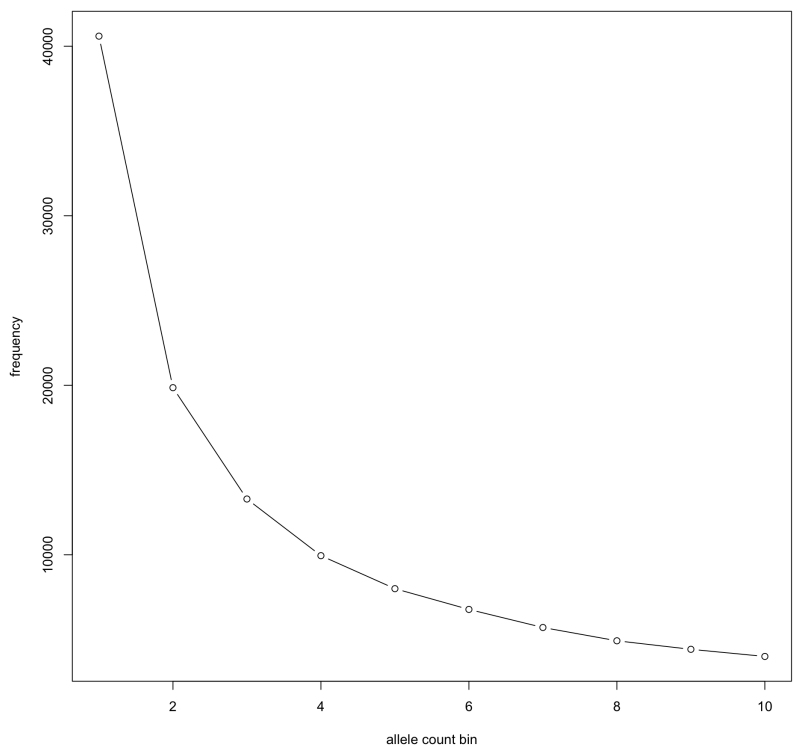
***slendr* provides a large library of functions for computing population genetic statistics on tree sequences**

Example: allele frequency spectrum

```
# sample 5 individuals
# (i.e. 10 chromosomes)
samples <-
  ts_samples(ts) %>%
  sample_n(5) %>%
  pull(name)

# compute allele frequency
# spectrum from the given set
# of individuals
afs1 <- ts_afs(
  ts, list(samples),
  mode = "branch",
  polarised = TRUE,
  span_normalise = TRUE
)

afs1
```

```
 [1] 40595.361 19858.677 13290.490
9948.410  8001.611  6777.533  5711.430
 [8]  4926.715  4424.256  4003.344
```

```
plot(afs1, type = "b",
     xlab = "allele count bin",
     ylab = "frequency")
```



But wait, we don’t have any mutations!

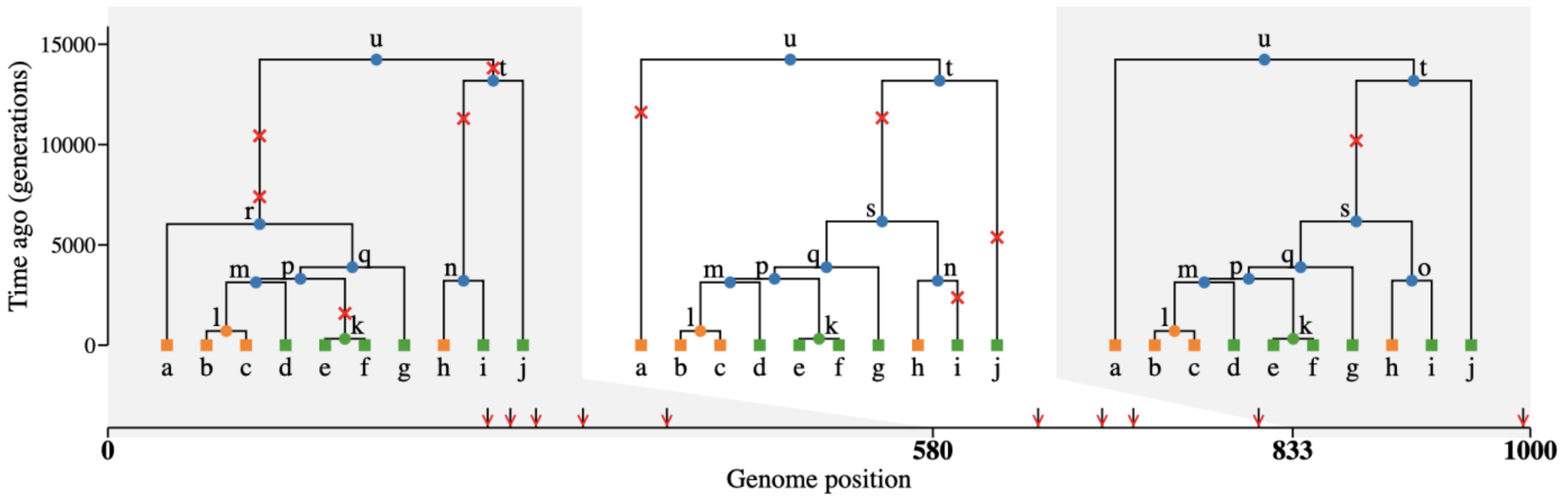
```
ts
```

TreeSequence	
Trees	390608
Sequence Length	100000000
Time Units	generations
Sample Nodes	20000
Total Size	65.9 MiB

Table	Rows	Size	Has Metadata

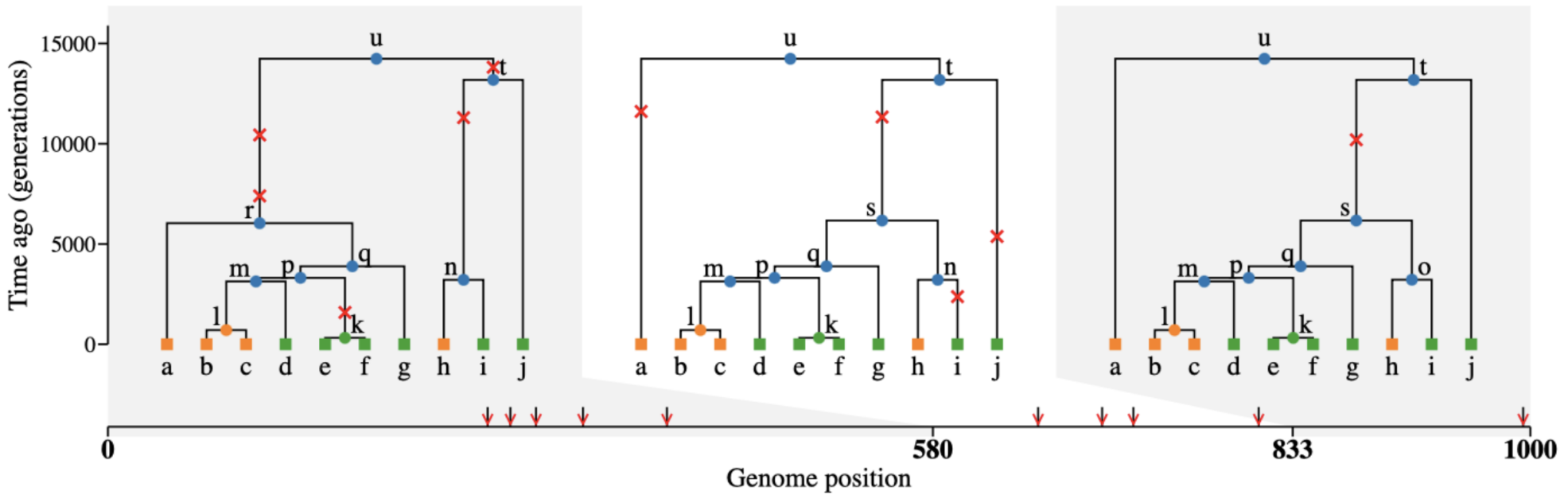
How can we compute statistics?

There is a duality between mutations and branch lengths in trees (more here).



But what if we want mutations?

Coalescent and mutation processes can be decoupled!



This means we can add mutations *after* the simulation.

This allows efficient, massive simulations (especially with SLiM)

```
ts_mutated <- ts_mutate(
  ts,
  mutation_rate = 1e-8
)
```

Or, with a shortcut:

```
ts <- msprime(
  model,
  sequence_length = 100e6,
  recombination_rate = 1e-8
) %>%
  ts_mutate(
    mutation_rate = 1e-8
  )
```

TreeSequence	
Trees	391911
Sequence Length	100000000
Time Units	generations
Sample Nodes	20000
Total Size	90.9 MiB

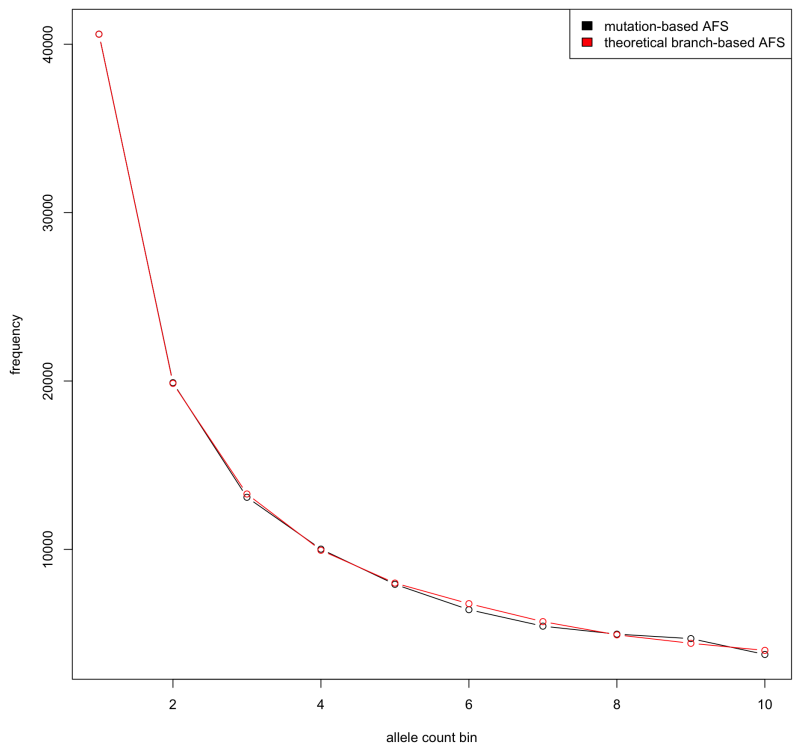
Table	Rows	Size	Has Metadata

Example: allele frequency spectrum

```
# sample 5 individuals
# (i.e. 10 chromosomes)
samples <-
  ts_samples(ts) %>%
  sample_n(5) %>%
  pull(name)

# compute allele frequency
# spectrum from the given set
# of individuals
afs2 <- ts_afs(
  ts, list(samples),
  polarised = TRUE
)

afs2
```

```
 [1] 40599 19901 13096 10017  7927  6422
5443  4968  4703  3763
```

```
plot(afs2, type = "b",
     xlab = "allele count bin",
     ylab = "frequency")
lines(afs1, type = "b", col = "r
legend("topright", legend = c("m
       fill = c("black", "red"))
```



What we have learned so far

1. creating populations – `population()`
2. compiling models – `compile_model()`
3. simulating tree sequences – `msprime()`
4. extracting samples – `ts_samples()`
5. computing AFS – `ts_afs()`

Let's put this to use!

Exercise #1

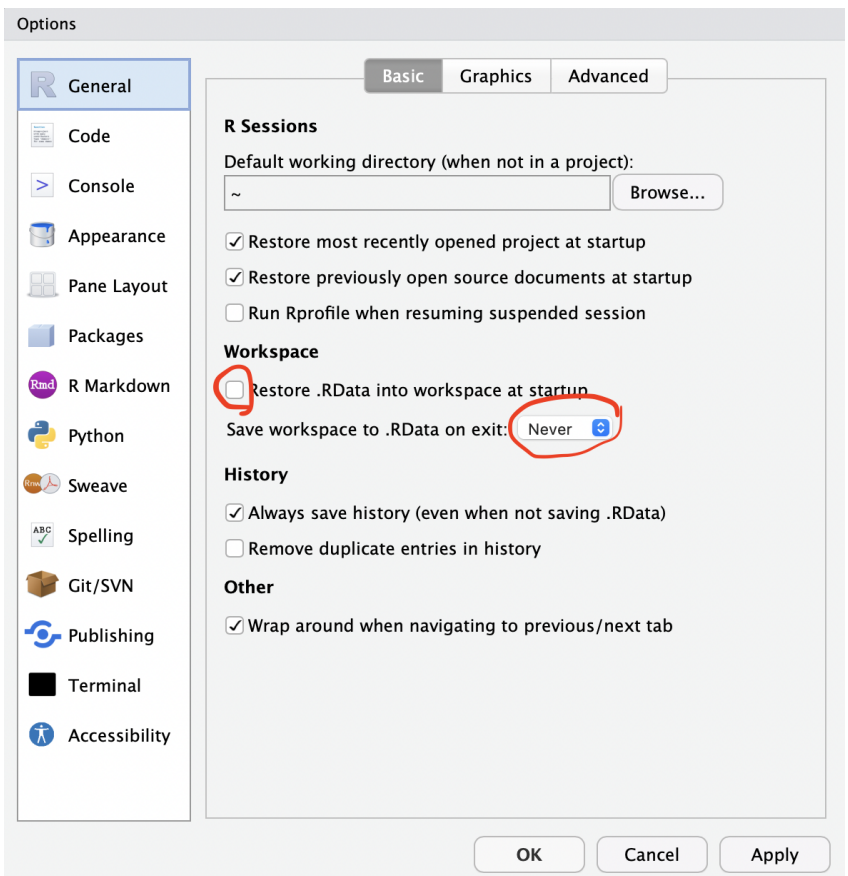
Open your RStudio session:

- **On-site participants**: http://ricco.popgen.dk:3838
- **Remote participants**: http://cloud.popgen.dk:8787

If you feel the servers are slow, you can also use your own laptop (setup instructions are here).

Workaround for an RStudio bug

RStudio sometimes interferes with Python setup that we need for simulation. To fix this, go to `Tools` -> `Global Options` in your RStudio and set the following options:



Before you start

1. Type `library(slendr)` into the R console

slendr is pre-installed for the browser RStudio. If you use RStudio on your own computer (not in the browser), you can get it by `install.packages("slendr")`.

2. Type `setup_env()`

When prompted to setup Python, answer “Yes”!

Exercise #1

Collaborator Hestu gave you AFS computed from 10 individuals of a sub-species of the *bushy-tailed squirrel* discovered in the Forest of Spirits in the land of Hyrule:

```
afs_observed <- c(2520, 1449, 855, 622, 530, 446, 365, 334, 349, 244, 264, 218,
133, 173, 159, 142, 167, 129, 125, 143)
```

Fossil evidence is consistent with constant size of the population over 100,000 generations of its history. An Oracle you met in the Temple of Time said that the true squirrel N_e has been between 1000 and 30000.

Use *slendr* to simulate history of this species. Use this to guess the likely value of squirrel's N_e given the observed AFS.

Exercise #1 – hints

1. Write a function that gets N_e as input and returns the AFS.
2. Find the N_e value that will give the closest AFS to the one you got from Hestu. Use whatever method you're comfortable with based on your programming experience:
   - i) Plot simulated AFS for different N_e with the AFS and just eye-ball N_e value that looks correct.
   - ii) Simulate AFS across a grid of N_e values and find the closest matching one (maybe use mean-squared error?)
   - iii) Run a mini-Approximate Bayesian Computation, using the Oracle's range of [10000, 30000] as a uniform prior.

Exercise #1 – eye-balling solution

solution

Exercise #1 – simulations on a grid

solution

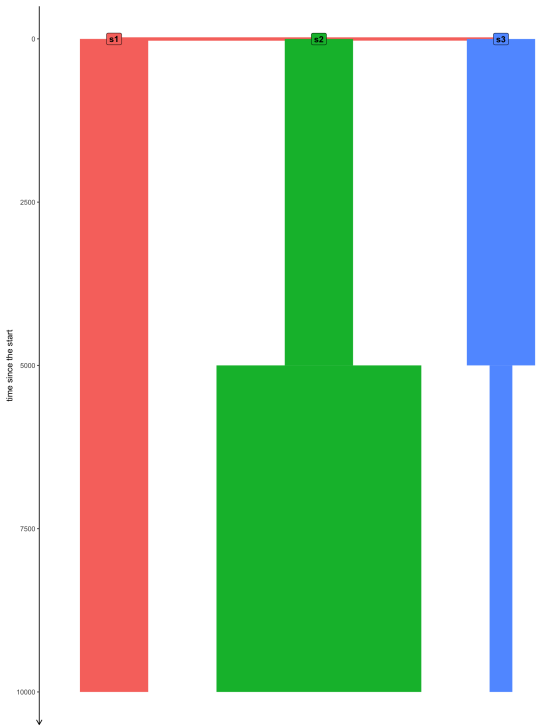
Exercise #1 – ABC inference

solution

Exercise #2

The squirrels have split into three species with different demographic histories (*s1*, *s2*, *s3* – model on the right). Species *s2* and *s3* are daughter populations which split in generation 2 from the original species *s1*.

Help the Oracle predict the future shape of the AFS of the squirrels after 10000 generations, assuming starting N_e = 6543? Species *s1* will remain constant, species *s2* will expand 3X, species *s3* will get 3X smaller.



Exercise #2 – solution

link

What else can *slendr* do?

Gene flow events

Gene flow is programmed using the `gene_flow()` function:

```
gf <- gene_flow(from = pop1, to = pop2, start = 500, end = 600, rate = 0.13)
```

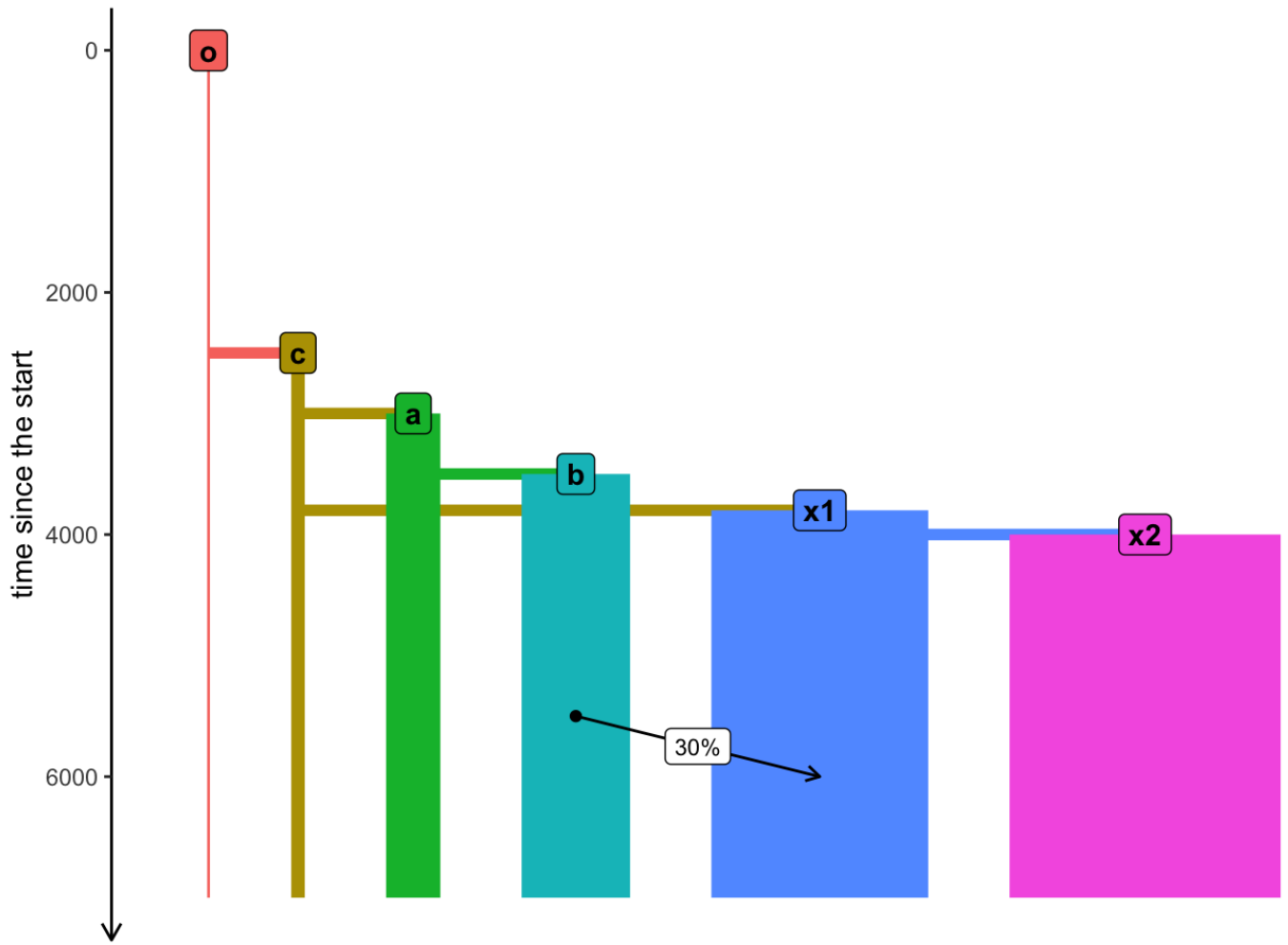
Multiple gene-flow events can be gathered in a list:

```
gf <- list(
  gene_flow(from = pop1, to = pop2, start = 500, end = 600, rate = 0.13),
  gene_flow(from = pop2, to = pop1, start = 700, end = 750, rate = 0.1)
)
```

`gene_flow()` checks admixture events for consistency

Let’s build another toy model

- Code



Diversity – `ts_diversity()`

Extract a list of lists of individuals' names for each population:

```
samples <-
  ts_samples(ts) %>%
  split(., .$pop) %>%
  lapply(pull, "name")
```

Compute diversity in each population:

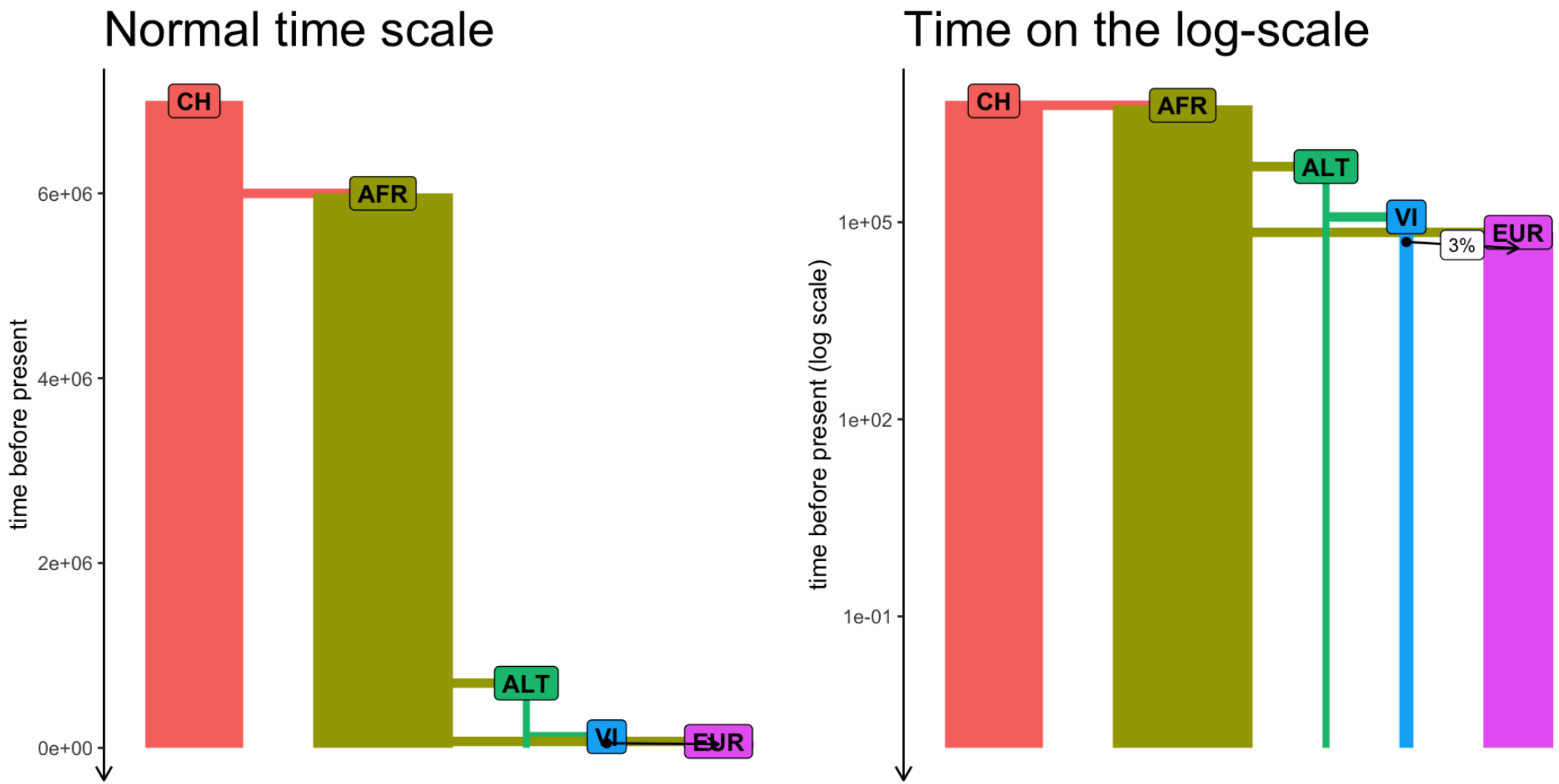
```
ts_diversity(ts, samples) %>%
  arrange(diversity)
```

```
# A tibble: 6 × 2
  set    diversity
  <chr>      <dbl>
1 o      0.00000405
2 c      0.0000212
3 a      0.0000540
4 b      0.0000689
5 x2     0.0000729
6 x1     0.0000758
```

Basically every `ts_<...>()` function of *slendr* accepts a tree-sequence object as its first argument.

Exercise #3

Program the following model in *slendr*



Parameter values

- “CH” outgroup at time 7 Mya (N_e = 7k)
- “AFR” splitting from “CH” at 6 Mya (N_e = 10k)
- “ALT” splitting from “AFR” at 700 kya (N_e = 500)
- “VI” splitting from “ALT” at 120 kya (N_e = 1k)
- “EUR” splitting from “AFR” at 70 kya (N_e = 5k)
- gene flow from “VI” to “EUR” at 3% over 50-40 kya
- generation time 30 years

Then simulate 100Mb sequence with `msprime()`, using recombination rate 1e-8 per bp per generation.

Exercise #3 – solution

link

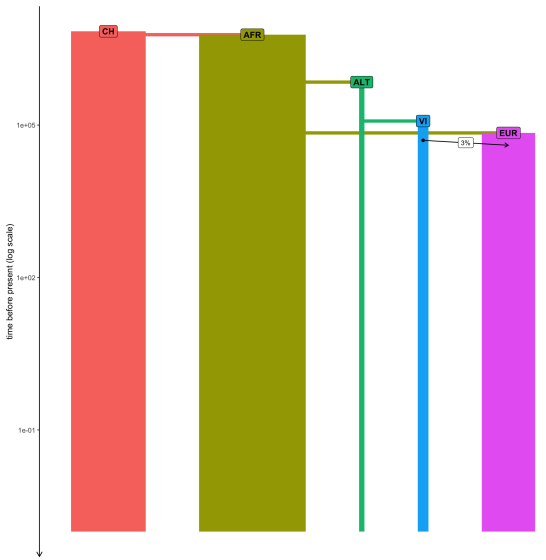
Exercise #4

Diversity and divergence

Use `ts_diversity()` to compute diversity in each simulated population (`CH`, `AFR`, ...).

Does the result match the expectation given demographic history?

What about divergence between all pairs of populations (`ts_divergence()`)? Do your results recapitulate phylogenetic relationships? How about `ts_f3()` or `ts_fst()`?



Exercise #4 – solution (diversity)

▶ Code

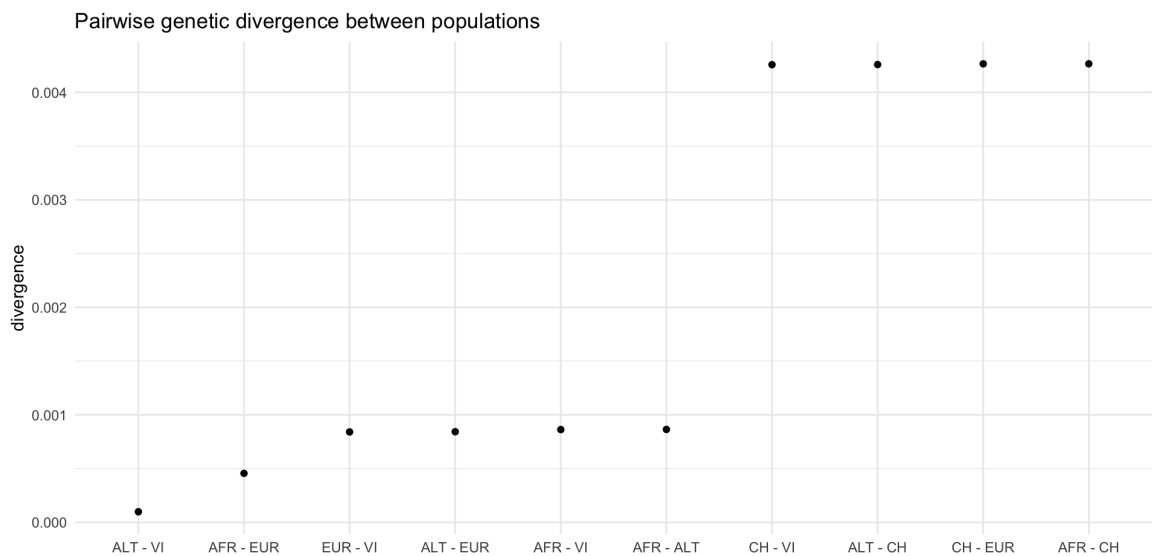
```
# A tibble: 5 × 2
  set   diversity
  <chr>     <dbl>
1 ALT   0.0000204
2 VI    0.0000378
3 CH    0.000277
4 EUR   0.000376
5 AFR   0.000401
```

Exercise #4 – solution (divergence)

▶ Code

```
# A tibble: 10 × 3
   x     y     divergence
   <chr> <chr>      <dbl>
 1 ALT   VI     0.0000980
 2 AFR   EUR    0.000456
 3 EUR   VI     0.000841
 4 ALT   EUR    0.000843
 5 AFR   VI     0.000863
 6 AFR   ALT    0.000864
 7 CH    VI     0.00426
 8 ALT   CH     0.00426
 9 CH    EUR    0.00427
10 AFR   CH     0.00427
```

▶ Code



Sampling “ancient DNA” time-series

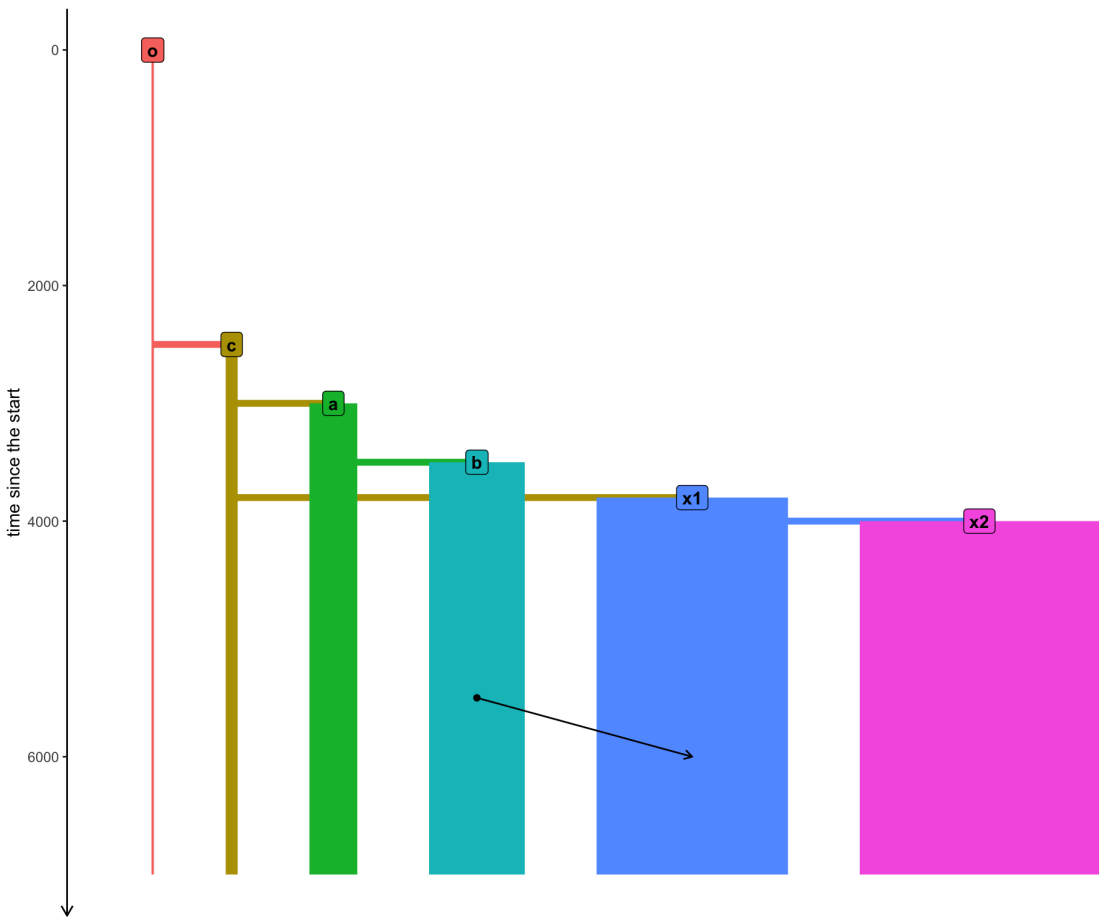
By default, *slendr* records every individual living at the end of the simulation. Sampling can be also scheduled explicitly:

```
x_schedule <- schedule_sampling(
  model, times = seq(from = 7000, to = 4000, by = -100),
  list(x1, 5), list(x2, 5)
)

abco_schedule <- schedule_sampling(
  model, times = 7000,
  list(a, 1), list(b, 1), list(c, 1), list(o, 1)
)

schedule <- rbind(x_schedule, abco_schedule)
```

The schedule can be used in an `msprime()` call like this:

```
ts <- msprime(model, sequence_length = 100e6, recombination_rate = 1e-8,
              samples = schedule)
```

Computing f_4 statistic – `ts_f4()`



Having simulated data from this model, can we detect gene flow from *b* into *x1*?

Computing f_4 statistic – `ts_f4()`

Simulate tree sequence, add mutations on it:

```
ts <- msprime(model, samples = schedule,
              sequence_length = 100e6, recombination_rate = 1e-8) %>%
  ts_mutate(mutation_rate = 1e-8)
```

```
ts_f4(ts, "x1_1", "x2_1", "b_1" , "o_1")
```

```
# A tibble: 1 × 5
  W     X     Y     Z              f4
  <chr> <chr> <chr> <chr>       <dbl>
1 x1_1  x2_1  b_1   o_1   0.000000205
```

```
ts_f4(ts, "x1_100", "x2_100", "b_1" , "o_1")
```

```
# A tibble: 1 × 5
  W      X      Y     Z              f4
  <chr>  <chr>  <chr> <chr>       <dbl>
1 x1_100 x2_100 b_1   o_1   0.00000127
```

Computing f_4 statistic for all x individuals

```
# extract information about samples from populations x1 and x2
x_inds <- ts_samples(ts) %>% filter(pop %in% c("x1", "x2"))

# create a vector of individual's names
# (used as input in ts_<...>() functions
x_names <- x_inds$name

# iterate over all sample names, compute
# f4(X, C; B, O)
f4_result<- map_dfr(
  x_names,
  function(w) ts_f4(ts, W = w, X = "c_1", Y = "b_1" , Z = "o_1")
)

# add the f4 value as a column to the sample information table
x_inds$f4 <- f4_result$f4
```

Computing f_4 statistic for all x individuals

- Code

Exercise #5

Neanderthal ancestry trajectory

1. Take your model of Neanderthal introgression
2. Implement temporal sampling:

- one “European” every 1000 yrs between 40 kya and today
- “Altai” (70 ky old) and “Vindija” individuals (40 ky old)

3. Compute f_4-ratio statistic using:

```
ts_f4ratio(ts, X = '<vector of "European" individuals>',
           A = '<"Altai">', C = '<"African">', O = '<"Chimp">')
```

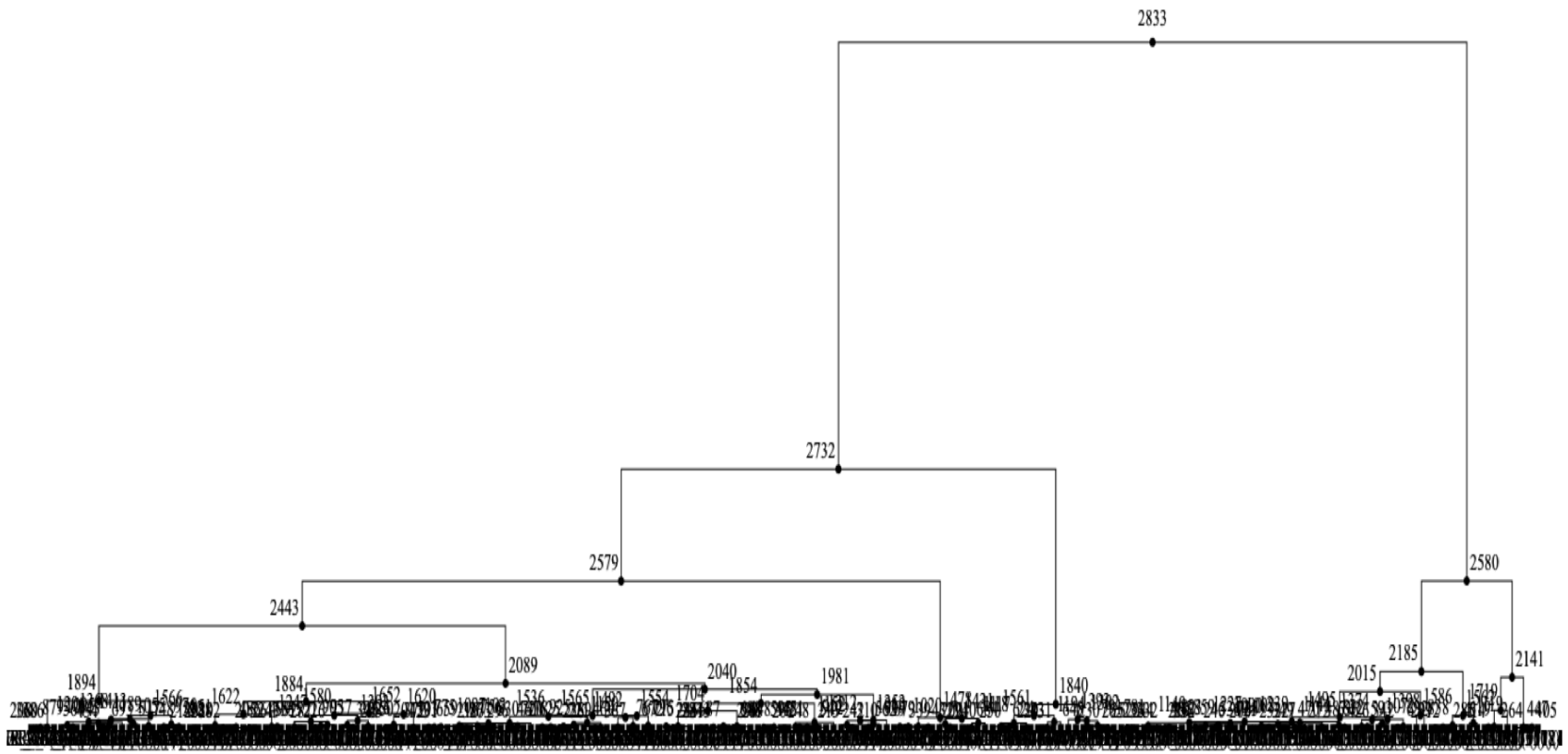
4. Plot the estimated trajectory of Neanderthal ancestry in Europe over time.

Exercise #5 – solution

solution

Tree-sequence simplification

Sometimes a tree sequence is too big



Simplification reduces the genealogies only to those nodes that form ancestors of a desired set of samples.

Meet `ts_simplify()`

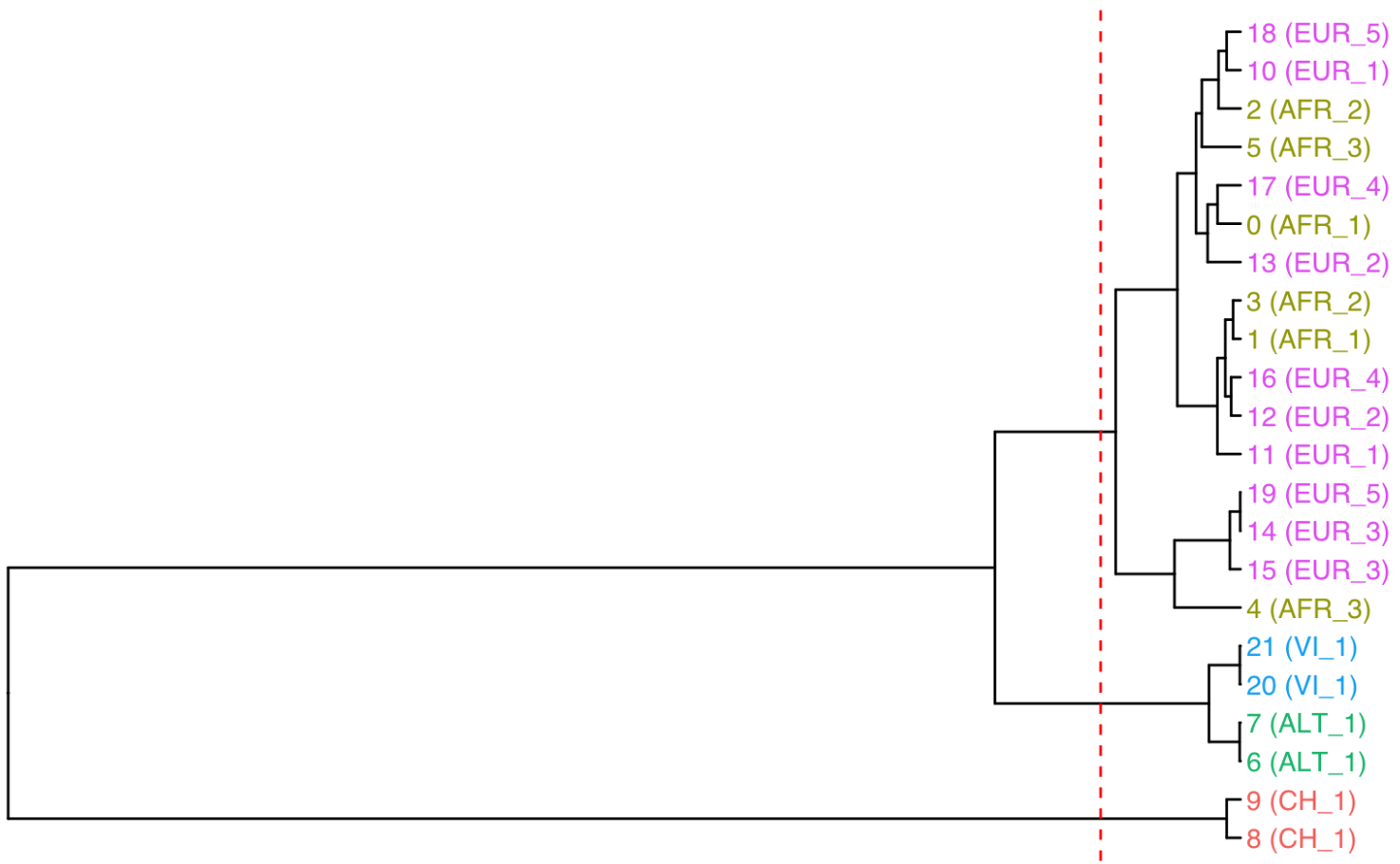
```
ts_small <- ts_simplify(
  ts,
  simplify_to = c(
    "CH_1",
    "AFR_1", "AFR_2", "AFR_3",
    "ALT_1", "VI_1",
    "EUR_1", "EUR_2", "EUR_3", "EUR_4", "EUR_5"
  )
)
```

Only the coalescent nodes of genealogies involving these samples will be retained.

Extracting a tree #1734

```
tree <- ts_phylo(ts_small, 1734, quiet = TRUE)
```

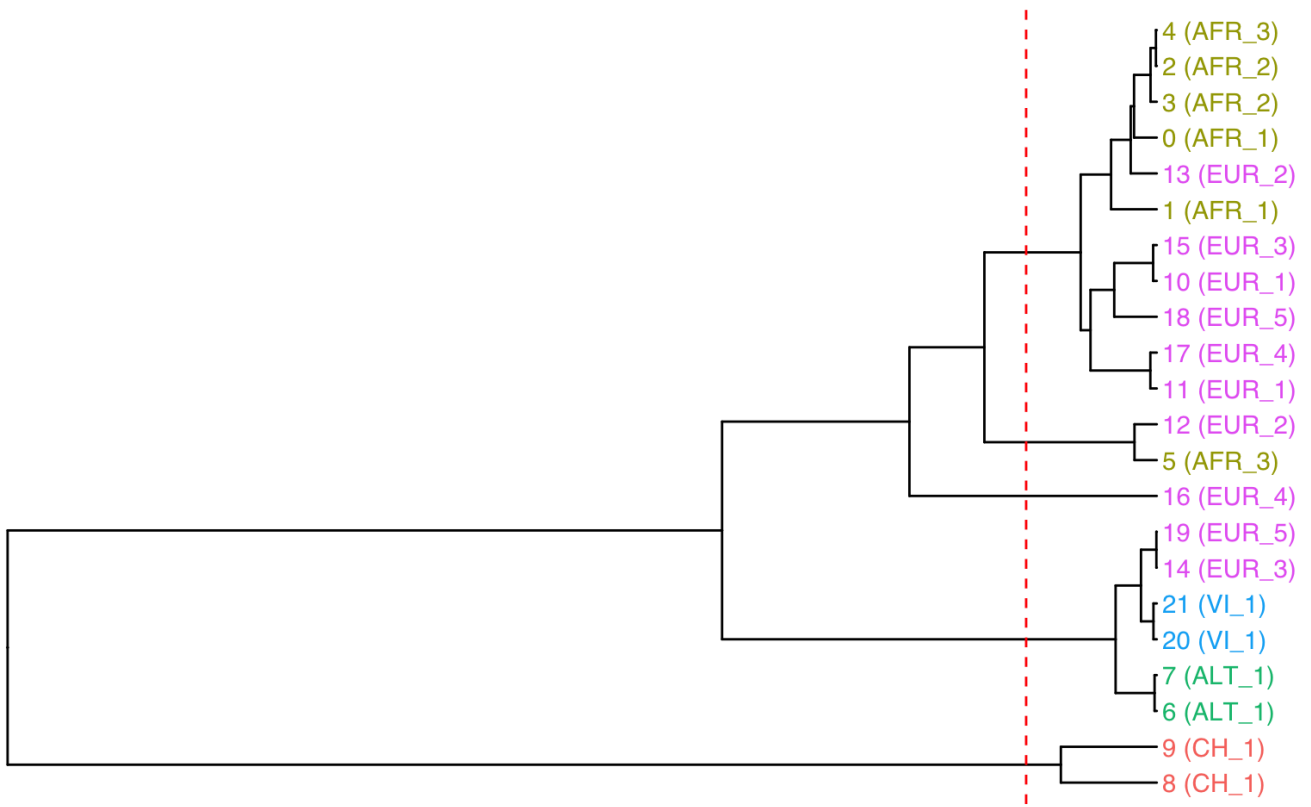
▶ Code



Extracting a tree #1

```
tree <- ts_phylo(ts_small, 1, quiet = TRUE)
```

▶ Code

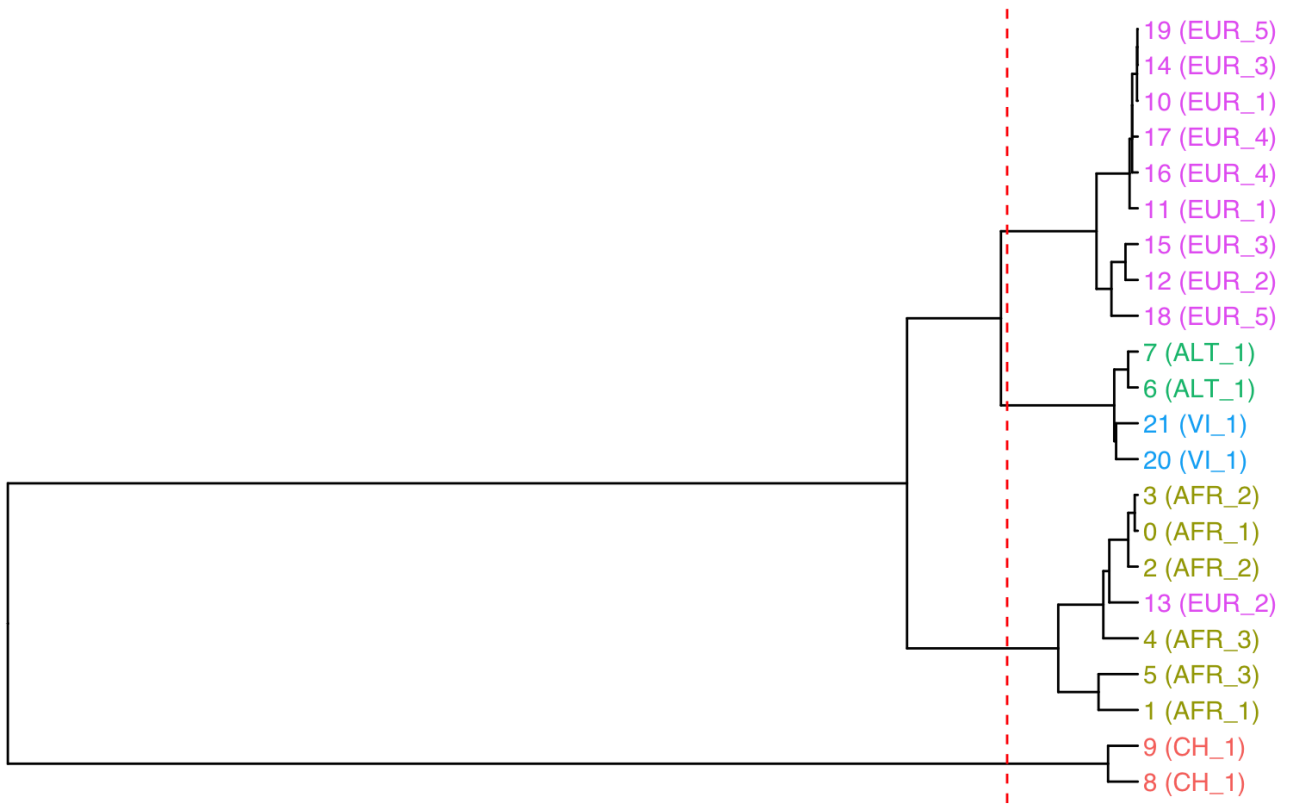


Introgression?

Extracting a tree #903

```
tree <- ts_phylo(ts_small, 903, quiet = TRUE)
```

▶ Code



Introgression?

Bonus exercise #2

Try to implement something relevant for your own work

- Program a demographic model for your species of interest
- Choose a statistic whose value "true value" you know from the literature (divergence? diversity? f4-statistic? ancestry proportion?)
- Try to replicate the statistic in a *slendr* simulation using some of its `ts_<...>()` tree-sequence functions!

Bonus exercise #3

Try to implement your own D-statistic

You can extract data from a tree sequence in the form of a genotype table using the function `ts_genotypes()`.

This function returns a simple R data frame:

- each column for an individual
- each row for a site
- each cell giving the number of derived alleles in that individual at that site

Use the genotype table to compute your own ABBA/BABA statistic on the Neanderthal model data and check if you can recover the same signal as you get from `ts_f4()`.