

Using Msprime to Simulate Gene Genealogies with Recombination

Alan R. Rogers

April 11, 2022

1 Introduction

In this project, you will use `msprime` [1] to check for bias in the estimates from your `Legofit` project. This will involve the following steps:

1. Install `msprime` on your computer.
2. Modify the Python script described below, so that it simulates your own model of history.
3. Run the script to generate simulated data, and pipe the output into `simpat`, from the `Legofit` package, to generate site pattern frequencies in `.opf` format.
4. Use `legofit` to analyze the simulated data.
5. Make a graph that compares the true parameter values (the ones used in the simulation) with the estimates obtained by running `legofit` on the simulated data. This will show you whether the `legofit` estimates are biased.

2 Installing `msprime`

If you already have Python 3 on your computer, installation is very simple:

```
python3 -m pip install msprime
```

3 A Python script to run `msprime`

On the class website, you will find a Python script called `msh.py`. This script describes a simple model of history, involving 2 modern populations and 1 archaic. You will modify this code to describe your own model of history. Before explaining how it works, I will show you what it does.

By default, the script does not run a simulation. Instead, it invokes the `msprime` “demography debugger.” To use it in this mode, type

```
python3 msh.py
```

and stare at the output. It’s fairly self explanatory and you can google “`msprime` demography debugger” for details.

To simulate data, you must use the `-r` command-line option. Here’s how to look at the first few lines of simulated data:

```
python3 msh.py -r | head
```

You should get something like

```
npops = 3
pop sample
x 1
y 1
n 1
0 1 1 0
0 0 0 1
0 0 0 1
0 1 1 0
0 1 1 0
```

This will be followed by an error message about a “Broken pipe,” which is generated because the “head” program stopped reading after 10 lines, and so Python had no place to put its output. In this output, the first few lines are a header, which will be read by simpat. It gives the number of populations, their names, and sample sizes. After that (the lines beginning with “0”) comes the data. Each line consists of four numbers: first the chromosome, and then the genotypes (0 or 1) of the haploid samples from “x,” “y,” and “z.”

We now dive into the code, so that you will understand what you need to change to build your own model of history. Here is a listing of “msp.py:”

```
1  #!/usr/bin/python3

3  # Before running this: "python3 -m pip install msprime"
   import msprime as msp
   import os, sys

6

   def usage():
       print("Usage: ./msp.py [options]")
9       print("  where options may include:")
       print("  -r or --run: run simulation. Default: run")
       print("                DemographyDebugger")
12      sys.exit(1)

       do_simulation = False
15      for arg in sys.argv[1:]:
           if arg == "-r" or arg == "--run":
               do_simulation = True
18      else:
           usage()

21      # time parameters in generations
       Txyn = 25920
       Txy = 3788
24      Ta = 1760           # age of Altai fossil
       Talpha = 1897      # time of Neanderthal admixture

27      # population sizes
```

```

Nxyn = 64964.1/2.0 # ancestral population
Nxy = 44869.2/2.0
30 Nn = 9756.8/2.0
Nx = 20000/2.0 # modern Africa
Ny = 20000/2.0 # modern Europe
33
# admixture
mAlpha = 0.05
36
nchromosomes = 10 # number of chromosomes: increase to 1000
basepairs = 2e6 # number of nucleotides per chromosome
39 u_per_site = 1.4e-8 # mutation

# Recombination rate. recomb is the probability of recombination
42 # between sites at opposite ends of the simulated sequence.
c = 1e-8 # rate per base pair per generation

45 dem = msp.Demography()

# Populations
48 dem.add_population(
    name = "X",
    description = "African",
51    initial_size = Nx
)
dem.add_population(
54    name = "Y",
    description = "Eurasian",
    initial_size = Ny
57 )
dem.add_population(
    name = "Y1",
60    description = "Y before delta gene flow",
    initial_size = Ny
)
63 dem.add_population(
    name = "N",
    description = "Neanderthal",
66    initial_size = Nn
)
dem.add_population(
69    name = "XY",
    description = "early modern humans",
    initial_size = Nxy
72 )
dem.add_population(
    name = "XYN",
75    description = "ancestral population",

```

```

        initial_size = Nxyn
    )
78
    # N->Y gene flow
    dem.add_admixture(
81         time = Talpha,
            derived = "Y",
            ancestral = ["Y1", "N"],
84         proportions = [1-mAlpha, mAlpha]
    )

87 # X-Y split
    dem.add_population_split(
        time = Txy,
90         derived = ["X", "Y1"],
            ancestral = "XY"
    )
93
    # XY-N split
    dem.add_population_split(
96         time = Txyn,
            derived = ["XY", "N"],
            ancestral = "XYN"
99     )

    # One haploid sample from each of 3 populations: 2 modern (X,Y),
102 # and 1 archaic (N).
    samples = [
        msp.SampleSet(num_samples=1, population="X", time=0, ploidy=1),
105     msp.SampleSet(num_samples=1, population="Y", time=0, ploidy=1),
        msp.SampleSet(num_samples=1, population="N", time=Ta, ploidy=1),
    ]
108
    if do_simulation:
        # Seed for random number generator. Uses 4 bytes (32 bits) from
111 # /dev/urandom.
        seed = int.from_bytes(os.urandom(4), "big")

114 # Simulate gene genealogy. When sim_ancestry is called without
        # num_replicates, it returns a value of class
        # tskit.trees.TreeSequence. But when it's called with
117 # num_replicates, it returns a "generator", which can be
        # used in a loop to deal with each replicate in turn.
        chr_generator = msp.sim_ancestry(samples = samples,
120                                     demography = dem,
                                         sequence_length = basepairs,
                                         random_seed = seed,
123                                     recombination_rate = c,

```

```

                                num_replicates = nchromosomes
    )
126
    # header
    lbl = ["x", "y", "n"]
129    print("npops = %d" % len(lbl))
    print("%s %s" % ("pop", "sampsiz"))
    for s in lbl:
132        print("%s %d" % (s, 1))

    # Put mutations onto the gene genealogy
135    for i, chr in enumerate(chr_generator):
        sim = msp.sim_mutations(chr, rate = u_per_site)

138        for variant in sim.variants():

            # Use only biallelic sites
141            if len(variant.alleles) != 2:
                continue

            print(i, end="")
            for g in variant.genotypes:
                print("", g, end="")
147            print()
    else:
        # Run demography debugger and quit
150        print(dem.debug())
        print("Use \"/sim -r\" to run simulation")

```

In this program, line 4 imports `msprime` and abbreviates its name as “`msp.`” Lines 7–12 define a function that prints a usage message and then aborts. It is used if some problem is detected in parsing the command-line arguments. Line 14 sets `do_simulation` equal to `false`. Unless this is changed by a command-line argument, the program will not execute a computer simulation. Lines 15–19 parse command-line arguments and set `do_simulation` equal to `true` if the user specifies the `-r` argument. Lines 21–35 define population history parameters. These values should reflect the fitted parameters of your own model of history. Lines 37–43 define parameters relating to the simulation. In this code, I have set `nchromosomes` equal to 10. You will eventually want to increase it to 1000. The `basepairs` parameter is the length of a chromosome. I have set it to `2e6`, because there isn’t much linkage disequilibrium between sites that are farther apart than that. Using a large number of relatively short chromosomes makes the simulations faster.

Line 45 defines a variable called `dem`, which will hold our model of demographic history. It is an instance of the `Demography` class, which is defined by `msprime`. Lines 48–77 define the populations used in the model. Each population has a short name, a description, and an initial size. This size represents the diploid size of the population at the recent end of its temporal extent. My model of history does not include growth within populations, so the initial size of a population is its size throughout its history.

Lines 79–85 define an episode of admixture, giving the time at which it occurs, the name of the derived population, those of the two ancestral populations, and the proportions of the derived

population that came from each of the two parents. Msprime does not allow you to specify the same population as both derived and ancestral within the same episode of admixture. That is why I define Y1 to represent the pre-admixture portion of population Y.

Lines 87–99 define two population splits, one separating XY into X and Y, and the other separating XYN into XY and N. A population split is similar to admixture, except that there are two derived populations and only one derived one.

Lines 103–107 define a list called `samples`, each element of which is an object of class `SampleSet`. The arguments are self explanatory. I set `ploidy=1` for consistency with Legofit.

Lines 109–147 execute only if we are doing a simulation. The only line you will need to change is 128, which defines the labels of the populations. These should be consistent with the `population` parameters specified in lines 103–107. There is presumably a way to read these labels directly from the `samples` array, but I haven’t yet figured out how to do that. This is why I define a separate `lbl` array on line 128.

Even though you won’t need to change the other lines, let me explain what is going on in lines 109–147. The variable `seed` is set using random data provided by the operating system. This value is used to initialize the random number generator so that each run of the program will use a different sequence of pseudo-random numbers.

Lines 119–125 simulate the population history, using msprime’s `sim_ancestry` function. The type of the value returned by this function depends on whether the `num_replicates` parameter is used. Without `num_replicates`, `sim_ancestry` returns an object of type `TreeSequence`, which is defined by msprime. With `num_replicates`, it returns a Python “generator,” which can be used to iterate over the `TreeSequence` objects of the replicates.

That is the purpose of the loop beginning at line 135. Each pass through that loop handles a different replicate, which I think of as a different chromosome. Within the loop `i` is the number of the current chromosome, and `chr` is an object (defined by msprime) that describes the current chromosome. Line 136 creates `sim`, which holds all the data about genetic variation on the current chromosome. This variable can be thought of as a list of polymorphic loci, or “variants.” We loop across these variants beginning at line 138, ignoring those that aren’t biallelic (lines 141–142). Each variant includes a list of genotypes, which we loop over in lines 146–147.

4 Generate site pattern frequencies

The output of “msp.py” is designed for use with the “sitepat” program, which is part of Legofit. To run a simulation and tabulate site pattern frequencies, type

```
python3 msp.py -r | simpat
```

You should get something like this:

```
# simpat version 2.3.8-12-gf1f00c9
# Including singleton site patterns.
# Number of site patterns: 6
Doing single pass through data to tabulate patterns..
# Nucleotide sites: 66238
# Sites used: 66238
#      SitePat          E[count]
#      x          13779.0000000
#      y          12838.0000000
#      n          20232.0000000
```

```
x:y      11146.0000000
x:n      3535.0000000
y:n      4679.0000000
```

simpat is finished

As you can see, this run only generated 66,238 polymorphic sites. You'll need more sites than this in your analysis. To produce them, edit `msp.py` and increase the value of `nchromosomes` to 1000.

In the scaled-up analysis, you'll need an additional file called "sim.sh." This is a shell script that runs the simulation and puts the output into files whose names are determined by `$1`, its first command-line argument. This file looks like this:

```
# msp.py is a Python script that executes msprime and generates output
# in the "sim" format, which is described in the document of simpat,
# within the Legofit package.  simpat is a program that reads sim
# format and tabulates site patterns.
ofile=sim${1}.opf # standard output
efile=sim${1}.err # error output
python3 msp.py -r | simpat 1>${ofile} 2>${efile}
```

To generate 50 sets of simulated data, you will use a command I describe below, but don't do this on your laptop. The command launches 28 parallel processes, and your laptop probably doesn't have that many cores. Do it on the CHPC server, but not on one of the login nodes. (Users are not allowed to use these for substantial calculations.) Instead, start a job on one of my owner nodes by typing

```
salloc -t 2:00:00 -n 28 -N 1 -A rogersa-kp -p rogersa-kp
```

This starts an interactive session on an owner node. The command above will work on the kingspeak cluster. On notchpeak, change `rogersa-kp` to `rogersa-np`. Once you're in that interactive session, type:

```
seq 0 49 | xargs -n 1 -P 28 bash sim.sh
```

Here, "seq 0 49" prints the integers from 0 through 49. The "xargs" command reads this input, and launches processes that look like "bash sim.sh 0," "bash sim.sh 1," and so on up to "bash sim.sh 49." These jobs run in parallel, 28 at a time.

When these jobs finish, you will have files with names like `sim0.opf`, `sim1.opf`, and so on up to `sim49.opf`. You can use these in Legofit analyses similar to the ones you have already done.

5 Using Legofit to study simulated data

The analysis pipeline for simulated data hardly differs from the one you used earlier on a real data set and 50 bootstrap replicates. The only difference is that now there is no real data set, and you have simulation replicates instead of bootstrap replicates.

Here is the slurm script, `a1.slr`, for stage 1 of the analysis:

```
#!/bin/bash
#SBATCH -J a1
#SBATCH --account=rogersa-np
#SBATCH --partition=rogersa-np
```

```

#SBATCH --time=0:10:00
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH -o a1-%a.legofit # stdout
#SBATCH -e a1-%a.err # stderr

i=${SLURM_ARRAY_TASK_ID}
ifile='printf "../sim/sim%d.opf" $i' # input file
stateout='printf "a1-%d.state" $i'

time legofit -1 -d 0 --stateOut ${stateout} --tol 3e-6 -S 5000 a.lgo ${ifile}

```

On the server, you would launch this command like this:

```

sbatch --array=0-49 a1.slr

```

6 Graphing estimates from simulated data

Figure 1 is copied from a recent preprint [2]. It compares estimates from 50 simulated data sets with the true parameter values, as specified in the input to `msprime`. The horizontal spread of the blue values measures the uncertainty of each estimated parameter. The horizontal distance between the red crosses and the center of the blue distribution measures bias.

The code that produced this graph is on the class website in a file called `b2.r`. For your project, modify this code to reflect the parameter names and the true parameter values of your own project. You may want to remove the code that graphs $m_\alpha + m_\epsilon$ and $m_\alpha - m_\epsilon$. These were relevant to a point I was making, but may not be of interest in your project.

Once you have edited `b2.r` as needed, you can run it from the command line by typing `Rscript b2.r`. This will produce a file called `b2dot.pdf`.

Here is a listing of `b2.r`:

```

1 # File b2.r: This script reads file "b2.flat", which contains
  # parameter estimates from several sets of simulated data. It then
3 # makes a 1-dimensional scatter plot for each parameter, which
  # includes the estimate from each simulation replicate along with the
  # true parameter value, as specified in the input to msprime. The true
6 # parameter values are hard coded into this file, rather than being
  # read as input.

9 library(ggplot2)
  library(cowplot)
  library(tidyr)
12
  # Adjust text size. Default is 11
  mytheme = theme_get()
15 mytheme$text$size = 18
  theme_set(mytheme)

18 # Input file

```

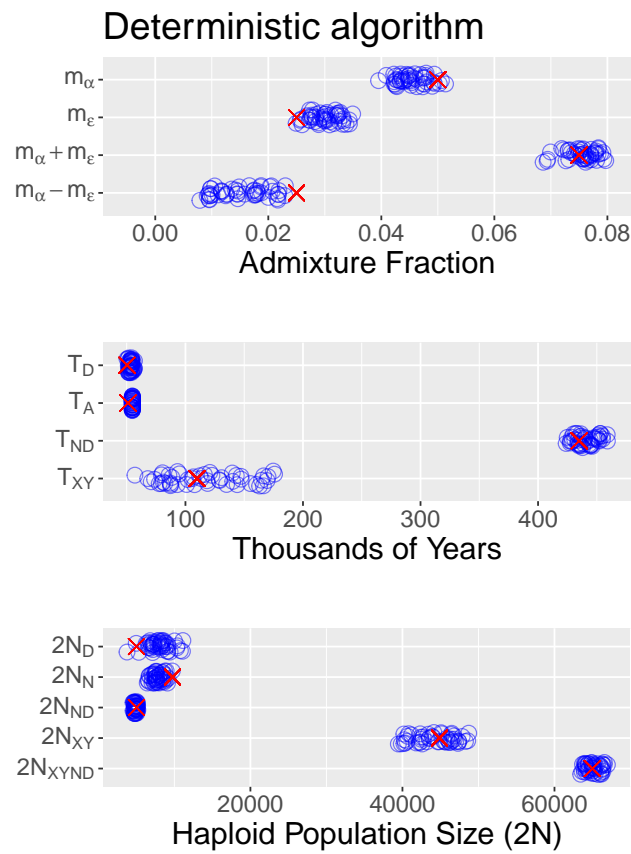



Figure 1: Parameter estimates from 50 simulated data sets. Blue circles are estimates and red crosses are the true parameter values. Symbols for simulation estimates have been jittered vertically.

```

wide <- read.table("b2.flat", header=T)

21 # parameter names
parnames <- c("mAlpha", "mEpsilon", "Txy", "Tnd", "Ta", "Td",
              "twoNxynd", "twoNxy", "twoNnd", "twoNn", "twoNd")
24
# Subset data, keeping only the parameters we wish to plot.
wide <- wide[,parnames]
27
# Parameter labels, using R's "expression" syntax to make subscripts.
lbl <- expression("m"[alpha], "m"[epsilon], "T"["XY"], "T"["ND"],
30   "T"["A"], "T"["D"],
   "2N"["XYND"], "2N"["XY"], "2N"["ND"], "2N"["N"], "2N"["D"])

33 # Make graph of migration parameters.
mpar <- c("mAlpha", "mEpsilon")
mdat <- subset(wide, select=mpar)
36
## Create two new parameters
mdat$msum <- mdat$mAlpha + mdat$mEpsilon
39 mdat$mdif <- mdat$mAlpha - mdat$mEpsilon

## Labels of migration parameters
42 mlbl <- expression("m"[alpha], "m"[epsilon], "m"[alpha]+"m"[epsilon],
   "m"[alpha]-"m"[epsilon])

45 ## Convert "wide" data format to "long".
mdat <- gather(mdat, par, value, mAlpha, mEpsilon, msum, mdif)

48 ## Create column of true parameter values.
mdat$tru <- rep(NA, nrow(mdat))
mdat$tru[mdat$par == "mAlpha"] <- 0.05
51 mdat$tru[mdat$par == "mEpsilon"] <- 0.025
mdat$tru[mdat$par == "msum"] <- 0.075
mdat$tru[mdat$par == "mdif"] <- 0.025
54 mpar <- c("mAlpha", "mEpsilon", "msum", "mdif")

## This controls the order in which parameters appear on the graph.
57 mdat$par <- ordered(mdat$par, levels=rev(mpar))

## Plot migration rate
60 mplt <- ggplot(mdat, aes(value, par)) +
  xlab("Admixture Fraction") +
  theme(aspect.ratio=0.2,
63   axis.title.y = element_blank()) +
  scale_x_continuous(limits=c(-0.005,0.08)) +
  scale_y_discrete(labels=rev(mlbl)) +
66   geom_jitter(height=0.2, width=0, shape=1, alpha=0.5, size=4, color="blue") +

```

```

    geom_point(mapping=aes(tru, par), shape=4, size=4, color="red") +
    ggtitle("Deterministic algorithm")
69
# Make graph of time parameters
tpar <- c("Txy", "Tnd", "Ta", "Td")
72 tlbl <- expression("T"["XY"], "T"["ND"], "T"["A"], "T"["D"])
tdat <- subset(wide, select=tpar)
tdat <- gather(tdat, par,value)
75 tdat$tru <- rep(NA, nrow(tdat))
tdat$tru[tdat$par == "Txy"] <- 3788
tdat$tru[tdat$par == "Tnd"] <- 15000
78 tdat$tru[tdat$par == "Ta"] <- 1760
tdat$tru[tdat$par == "Td"] <- 1734
tdat$par <- ordered(tdat$par, levels=tpar)
81
tplt <- ggplot(tdat, aes(29*value/1000, par)) +
  xlab("Thousands of Years") +
84   theme(aspect.ratio=0.2,
         axis.title.y = element_blank()) +
  scale_y_discrete(labels=tblbl) +
87   geom_jitter(height=0.2, width=0, shape=1, alpha=0.5, size=4, color="blue") +
  geom_point(mapping=aes(29*tru/1000, par), shape=4, size=4, color="red")

90 # Make graph of population-size parameters
npar <- c("twoNxynd", "twoNxy", "twoNnd", "twoNn", "twoNd")
nlbl <- expression("2N"["XYND"], "2N"["XY"], "2N"["ND"], "2N"["N"],
93   "2N"["D"])
ndat <- subset(wide, select=npar)
ndat <- gather(ndat, par,value)
96 ndat$tru <- rep(NA, nrow(ndat))
ndat$tru[ndat$par == "twoNxynd"] <- 64964.1
ndat$tru[ndat$par == "twoNxy"] <- 44869.2
99 ndat$tru[ndat$par == "twoNnd"] <- 5000.0
ndat$tru[ndat$par == "twoNn"] <- 9756.8
ndat$tru[ndat$par == "twoNd"] <- 5000
102 ndat$par <- ordered(ndat$par, levels=npar)

nplt <- ggplot(ndat, aes(value, par)) +
105   xlab("Haploid Population Size (2N)") +
  theme(aspect.ratio=0.3,
        axis.title.y = element_blank()) +
108   scale_y_discrete(labels=nlbl) +
  geom_jitter(height=0.2, width=0, shape=1, alpha=0.5, size=4, color="blue") +
  geom_point(mapping=aes(tru, par), shape=4, size=4, color="red")
111
# Combine the 3 graphs into a single graph
plot_grid(mplt, tplt, nplt, align="hv", ncol=1)
114

```

```
# Save graph in pdf format.  
ggsave("b2dot.pdf")
```

References

- [1] Jerome Kelleher, Alison M Etheridge, and Gilean McVean. Efficient coalescent simulation and genealogical analysis for large sample sizes. *PLoS Computational Biology*, 12(5):1–22, 5 2016.
- [2] Alan R. Rogers. An efficient algorithm for estimating population history from genetic data. *bioRxiv*, 427922, 2021. Version 5 of this preprint has been peer-reviewed and recommended by *Peer Community In Mathematical and Computational Biology* (<https://doi.org/10.24072/pci.mcb.100003>).