An Msprime Tutorial

Alan R. Rogers

December 26, 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 msp.py. This script describes a simple model of history, involving 3 populations: 2 modern 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, in invokes the msprime "demography debugger." To use it in this mode, type

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

You should get something like

```
npops = 3
pop sampsize
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:”

```
#!/usr/bin/python3

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

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

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

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

# population sizes
```

```
Nxyn = 64964.1/2.0  # ancestral population
Nxy = 44869.2/2.0
Nn = 9756.8/2.0
Nx = 20000/2.0      # modern Africa
Ny = 20000/2.0      # modern Europe

# admixture
mAlpha = 0.05

nchromosomes = 10   # number of chromosomes: increase to 1000
basepairs = 2e6     # number of nucleotides per chromosome
u_per_site = 1.4e-8 # mutation

# Recombination rate per base pair per generation
c = 1e-8

dem = msp.Demography()

# Populations
dem.add_population(
    name = "X",
    description = "African",
    initial_size = Nx
)
dem.add_population(
    name = "Y",
    description = "Eurasian",
    initial_size = Ny
)
dem.add_population(
    name = "Y1",
    description = "Y before delta gene flow",
    initial_size = Ny
)
dem.add_population(
    name = "N",
    description = "Neanderthal",
    initial_size = Nn
)
dem.add_population(
    name = "XY",
    description = "early modern humans",
    initial_size = Nxy
)
dem.add_population(
    name = "XYN",
    description = "ancestral population",
    initial_size = Nxyn
```

```
)

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

# X-Y split
dem.add_population_split(
    time = Txy,
    derived = ["X", "Y1"],
    ancestral = "XY"
)

# XY-N split
dem.add_population_split(
    time = Txyn,
    derived = ["XY", "N"],
    ancestral = "XYN"
)

dem.sort_events()

if dem == None:
    raise ValueError("dem = None after sort_evenrts")

# One haploid sample from each of 3 populations: 2 modern (X,Y),
# and 1 archaic (N).
samples = [
    msp.SampleSet(num_samples=1, population="X", time=0, ploidy=1),
    msp.SampleSet(num_samples=1, population="Y", time=0, ploidy=1),
    msp.SampleSet(num_samples=1, population="N", time=Ta, ploidy=1),
]

if do_simulation:
    # Seed for random number generator. Uses 4 bytes (32 bits) from
    # /dev/urandom.
    seed = int.from_bytes(os.urandom(4), "big")

    # 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
    # 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,
                                            demography = dem,
                                            sequence_length = basepairs,
126                                         random_seed = seed,
                                            recombination_rate = c,
                                            num_replicates = nchromosomes
129         )

            # header
132         lbl = ["x", "y", "n"]
            print("npops = %d" % len(lbl))
            print("%s %s" % ("pop", "sampsize"))
135         for s in lbl:
                print("%s %d" % (s, 1))

138         # Put mutations onto the gene genealogy
            for i, chr in enumerate(chr_generator):
                sim = msp.sim_mutations(chr, rate = u_per_site)
141
                for variant in sim.variants():

144                 # Use only biallelic sites
                    if len(variant.alleles) != 2:
                        continue
147
                    print(i, end="")
                    for g in variant.genotypes:
150                     print("", g, end="")
                    print()
        else:
153         # Run demography debugger and quit
            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 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 44 defines a variable called `dem`, which will hold our model of demographic history. It is in instance of the `Demography` class, which is defined by msprime. Lines 47–76 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 78–84 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 86–98 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 105–111 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 114–151 execute only if we are doing a simulation. The only line you will need to change is 132, 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.

Even though you won't need to change the other lines, let me explain what is going on. 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 123–129 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 139. 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 that describes the current chromosome. Line 140 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 142, ignoring those that aren't biallelic. Each variant includes a list of genotypes, which we loop over in lines 149–150.

4 Generate site pattern frequencies

The output of "msp.py" is designed for use with the "simpat" 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 dosn'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 Rogers [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`:

```
# File b2.r: This script reads file "b2.flat", which contains
# parameter estimates from several sets of simulated data. It then
# 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
# parameter values are hard coded into this file, rather than being
# read as input.

library(ggplot2)
library(cowplot)
library(tidyr)

# Adjust text size. Default is 11
mytheme = theme_get()
```

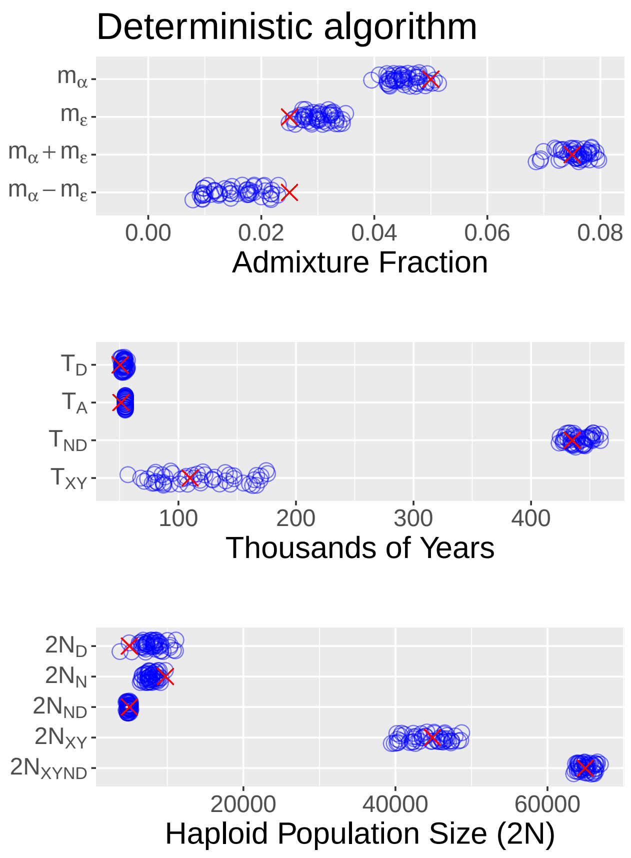

Figure 1: Parameter estimates from 50 simulated data sets. Blue circles are estimates and red crosses are the true parameter values. Symbols for simulation estimates have been jittered vertically.

```
mytheme$text$size = 18
theme_set(mytheme)

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

# parameter names
parnames <- c("mAlpha", "mEpsilon", "Txy", "Tnd", "Ta", "Td",
              "twoNxynd", "twoNxy", "twoNnd", "twoNn", "twoNd")

# Subset data, keeping only the parameters we wish to plot.
wide <- wide[,parnames]

# Parameter labels, using R's "expression" syntax to make subscripts.
lbl <- expression("m"[alpha], "m"[epsilon], "T"["XY"], "T"["ND"],
    "T"["A"], "T"["D"],
    "2N"["XYND"], "2N"["XY"], "2N"["ND"], "2N"["N"], "2N"["D"])

# Make graph of migration parameters.
mpar <- c("mAlpha", "mEpsilon")
mdat <- subset(wide, select=mpar)

## Create two new parameters
mdat$msum <- mdat$mAlpha + mdat$mEpsilon
mdat$mdif <- mdat$mAlpha - mdat$mEpsilon

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

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

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

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

## Plot migration rate
mplt <- ggplot(mdat, aes(value, par)) +
    xlab("Admixture Fraction") +
    theme(aspect.ratio=0.2,
```

```r
          axis.title.y = element_blank()) +
    scale_x_continuous(limits=c(-0.005,0.08)) +
    scale_y_discrete(labels=rev(mlbl)) +
    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")

# Make graph of time parameters
tpar <- c("Txy", "Tnd", "Ta", "Td")
tlbl <- expression("T"["XY"], "T"["ND"], "T"["A"], "T"["D"])
tdat <- subset(wide, select=tpar)
tdat <- gather(tdat, par,value)
tdat$tru <- rep(NA, nrow(tdat))
tdat$tru[tdat$par == "Txy"] <- 3788
tdat$tru[tdat$par == "Tnd"] <- 15000
tdat$tru[tdat$par == "Ta"] <- 1760
tdat$tru[tdat$par == "Td"] <- 1734
tdat$par <- ordered(tdat$par, levels=tpar)

tplt <- ggplot(tdat, aes(29*value/1000, par)) +
    xlab("Thousands of Years") +
    theme(aspect.ratio=0.2,
          axis.title.y = element_blank()) +
    scale_y_discrete(labels=tlbl) +
    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")

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

nplt <- ggplot(ndat, aes(value, par)) +
    xlab("Haploid Population Size (2N)") +
    theme(aspect.ratio=0.3,
          axis.title.y = element_blank()) +
    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")
```

```
# Combine the 3 graphs into a single graph
plot_grid(mplt, tplt, nplt, align="hv", ncol=1)

# 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 (May 2016), pp. 1–22. DOI: `10.1371/journal.pcbi.1004842`.

[2] Alan R. Rogers. “An efficient algorithm for estimating population history from genetic data”. *Peer Community Journal* 2 (2022), e32. DOI: `10.24072/pcjournal.132`.