Selection on **F**inite **S**ites under **CO**mplex **D**emographic **E**vents (SFS_CODE)

Ryan D. Hernandez
Draft date: November 27, 2007

1 Preface

This is the user's guide to SFS_CODE. It outlines how to compile and use the program, but does not delve into many of the details regarding specific algorithms used or the underlying data structures implemented in the C source code. A subsequent verbose version (more of an *owner's manual*) of this document will be made available in the near future, and will delve into the gory details of how SFS_CODE works and is implemented. The verbose version will also include many examples and several unpublished simulation results that have been used to guide my intuition in population genetics, and will hopefully help you understand how to effectively use SFS_CODE. Table 5 on page 29 outlines every option implemented in SFS_CODE, in includes a page reference indicating where each option was described in the text.

2 Overview

The program that this document is dedicated to can be described in a single run-on sentence as follows:

> SFS_CODE is a Wright-Fisher style forward population genetic simulation program for finite-site mutation models with selection, recombination, and demography.

This means that an entire population of individuals (and all their chromosomes) is followed generation by generation, from the beginning of the simulation to the time of sampling. This is contrary to coalescent simulations [such as `ms`; Hudson (2002)], where the history of a sample is simulated backward in time until its founder. SFS_CODE has the ability to simulate finite-site mutation models (meaning that some sites can receive several mutations). Nonetheless, SFS_CODE actually stores all mutations that are either segregating or fixed in at least one of the populations, so it can also act like an infinite-sites simulation program. However, its purpose is to generate a set of DNA sequences (an alignment) that can then be analyzed. This alignment, by the nature of the simulation, can therefore contain sites that have been the target of many mutations (as well as repeatedly being selected upon).

As described in further detail in subsequent sections, `SFS_CODE` allows the user to simulate highly detailed populations, with at least as much flexibility as `ms`. In addition to allowing for fairly complex demographic effects and migration schemes, `SFS_CODE` also allows the user to simulate coding versus non-coding regions, apply a distribution of selective effects to new mutations, generate domesticated populations, assume different male and female population sizes, linked and unlinked loci, sex and autosomal chromosomes, polyploids (haploid, diploid, or tetraploid), as well as a suite of built in or custom mutation models.

The basic algorithm used in this program is as follows:

1. Sample a sequence from the stationary distribution of the mutation model.
2. Burn-in a single population to mutation/selection balance.
3. Perform demographic and other evolutionary events.
4. Sample individuals from populations.

Each generation consists of the following components:

1. Produce each individual by randomly sampling a mother and a father from the previous generation (with replacement according to their relative fitness for their sex, unless simulating haploids, in which case there is no sex).
2. Randomly select individuals to migrate among populations.
3. Distribute a Poisson number of recombination/mutation events.

3 Getting Started

3.1 Compiling the Program

This section is only if you have downloaded the source code and wish to compile the program yourself. If you are using the web-based version of the program, then you can skip this section.

After obtaining and unzipping the distribution of this program, you will have a folder called `SFS_CODE`. Inside this folder, you will find (at least) two subdirectories `src` and `doc`. In the subdirectory `src`, you should find a `makefile`, along with several more subdirectories. The `makefile` will be used to compile all the programs provided with this distribution. It uses GNU's `gcc` compiler. Using your favorite command line terminal (Windows users should download and install `Cygwin` from `http://www.cygwin.com`), change directory to `SFS_CODE/src/`, and type `make`. This will create the directory `SFS_CODE/bin/`, which will contain the executables `sfs_code`, `convertSFS_CODE`, as well as any other programs in the current distribution.

If you get compiling errors, it is likely that either you do not have `gcc` installed, or the optimization flag `-fast` is not implemented for your operating system. If it is the former, then make sure you need to install `gcc`, or change the `makefile` to use your favorite compiler. If you are using `Cygwin`, you may need to update your version, making sure to install `gcc`. If it is the latter (or you get strange "Illegal instruction" errors at runtime), open the `makefile` using your favorite text editor (NOT Word, as you don't want to accidentally add any formatting flags to the file). Scroll down to about line 11, where it says `CFLAGS = ... -fast`. Replace the text "`-fast`" optimization flag with "`-O3`". Now proceed as before. If there are still problems, contact the author, and inform him of the system you are using. Note, if you are planning to use the Intel compiler, you may need to edit the source code `sfs_code.c` by uncommenting the very first line (this enables functions that Intel deems as "safe" but are not part of the standard `C` library, and will get rid of annoying warning messages).

3.2 Usage: Arguments at the Command Line

`SFS_CODE` is a command-line program. If you have already compiled the program, then you should be ready to go. Change directory to `SFS_CODE/bin`. A full list of options can be found in Table 5 on page 29.

The basic command to run `SFS_CODE` is as follows:

```
sfs_code <Npops> <Niter> [<options> [arguments]]
```

Where `<Npops>` is the total number of populations you want to simulate, and `<Niter>` is the total number of iterations you want to run. In this documentation, arguments and options that are enclosed in <angled brackets> are required, and those in [square brackets] are optional. Subsequently, those in both angled and square brackets can be required in some potentially optional instances (e.g. `[<options>...]`, if you include anything after `<Niter>` then they must be options, which may contain required and/or optional arguments).

In `SFS_CODE`, all options have both a **long name** and a **short name**, except for timed events (beginning with '`-T`', described later, and only use the short name). For example, to set the mutation rate, you could use either "`-t` θ" or "`--theta` θ" to achieve the same result. Though both long and short names are case-sensitive, long names are of arbitrary length and tend to be more descriptive of the option. Short names are a single letter. Note that long names are preceded by two dashes ("`--`") while short names are preceded by only a single dash ("`-`"). Both the long and short names of all options are provided in Table 5 on page 29.

In the text of this document, I will provide templates for each option, as well as numbered examples. In option templates, I will first give the long name, then the short name in parenthesis, followed by the format of its arguments, as in the following pattern:

```
--long_name (-short_name) [arguments]
```

As a first example, the help menu can be obtained using the option

```
--help (-h)
```

help menu

This means you would access the help menu by typing "`./sfs_code 1 1 -h`". In this special example, the number of populations and number of iterations do not need to be specified, so you could just type "`./sfs_code --help`" or "`./sfs_code -h`".

4 Running SFS_CODE

The most basic simulation is the following:

Ex. 1. `$ ./sfs_code 1 1`

Typing example 1 (excluding the `$`, which just represents the bash shell; in Windows, you also might not need the "`./`" bit either) into the command prompt will result in running a single iteration of the default simulation. The default parameter values are given in section 10 toward the end of the documentation, and consists of simulating sequences of length 5000 nucleotide base pairs (5kb) from a "standard neutral" population of 500 diploid individuals, where the population scaled mutation rate $\theta = 0.001$/site with no recombination, from which a sample of 6 individuals will be drawn. By "standard neutral" population, I am referring to a population that is devoid of every evolutionary force other than mutation and drift. The full list of default parameter values is given in the Default Parameters section below.

mutation rate

The **mutation rate** per site ($\theta = 4N_e\mu$, for a diploid population) can easily be increased to a value of 0.01 per site using the option `--theta (-t) <`θ`>` as follows:

Ex. 2. `$ ./sfs_code 1 1 -t 0.01`

recombination rate

Recombination is just as easy to incorporate using the `--rho (-r) <`ρ`>` option, where $\rho = 4N_er$ is the population scaled rate of recombination between adjacent sites for a diploid population. For example, the following would simulate a standard neutral population with per site mutation and recombination rates equal to 0.01.

Ex. 3. `$ ./sfs_code 1 1 -t 0.01 -r 0.01`

multiple iterations

In general, you will want to do several (perhaps several thousand) simulations. Doing so requires some patience (this is a forward simulation, after all). However, **multiple simulations** can be performed at once by changing the parameter `<Niter>`. Doing multiple simulations this way is beneficial, as compared to running them all independently, because SFS_CODE is able to take advantage of all the effort that went into all the previous burn-in periods. After an extensive initial burn-in period, the population will be at stationarity. It is much easier to obtain an independent draw from a population at stationarity than it is to reach stationarity. Figure 1 shows how this is done.

The default initial burn-in time is $5\times PN$ generations, while subsequent burn-in periods are only $2\times PN$. You can change the initial burn-in time using

change initial burn-in time

```
--BURN (-B) <burn>
```

and change the subsequent burn-in periods (for iterations > 1) using

change subsequent burn-in times

```
--BURN2 (-b) <burn>
```

This would set the initial or subsequent burn-in times to `<burn>`$\times PN$ generations.

In SFS_CODE, it is also possible to simulate an **arbitrary number of loci** (linked or unlinked) of arbitrary length using the following option.

`--length (-L) <nloci> <L_1> [<L_2>...<L_nloci>] [R]` *number of loci & length*

This option allows you to simulate `<nloci>`. The first locus will have length `<L_1>`. You can stop here to set all loci to the same length. Otherwise, you have two options. You can specify each of `<L_2>...<L_nloci>` to set the lengths of each locus, or if you have a repeating pattern (e.g. a short locus followed by a long one) you can specify a subset of lengths followed by the character 'R'. For example, if you want to simulate 4 loci, with lengths (500bp, 1kb, 500bp, 1kb), then you could use either of the following commands.

Ex. 4.
```
$ ./sfs_code 1 1 -L 4 500 1000 500 1000
$ ./sfs_code 1 1 -L 4 500 1000 R
```

You can **change the linkage among loci** using the next option.

`--linkage (-l) <p/g> <d_1> [<d_2>...<d_nloci-1>] [R]` *linkage among loci*

The first argument to this option must either be 'p' or 'g', indicating whether the distance between loci will be be `<p>`hysical distance (in basepairs) or `<g>`enetic distance (recombination fraction). The second argument is the distance between the first two loci. This is all you need if you want all adjacent loci to have the same distance. Otherwise, (again) you have two options. You can either specify the distance between each pair of adjacent loci (i.e. provide `<nloci>-1` values), or, if you have a repeating linkage structure you can specify a subset of distances followed by the character 'R'. For **independent loci**, you can use "`--linkage p -1`" or "`--linkage g 0.5`". As an example, consider simulating 2 independent genes, each having 4 exons with lengths as in example 4 that are equally spaced with 2kb introns. You could simulate this as follows.

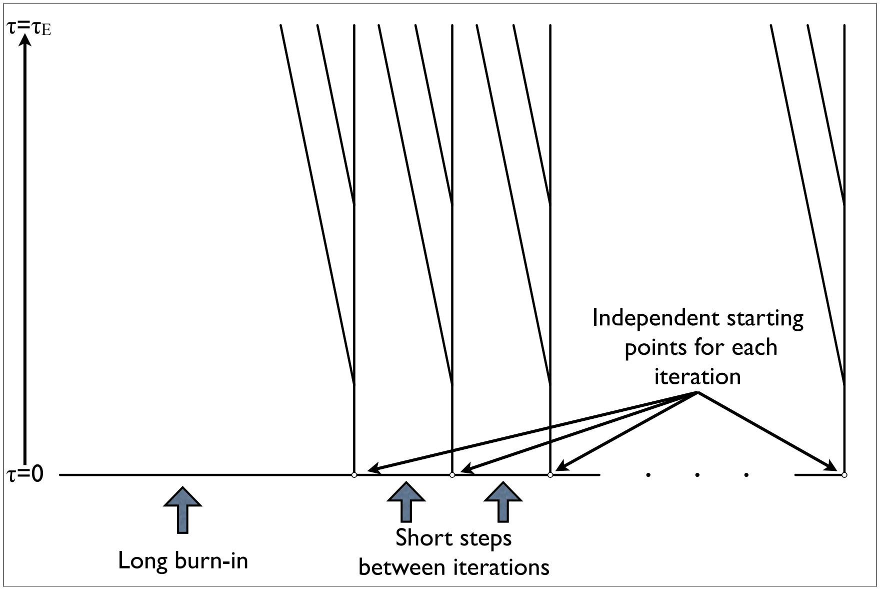
τ=τ_E
τ=0
Independent starting points for each iteration
Long burn-in
Short steps between iterations

Figure 1: Simulating multiple iterations in SFS_CODE begins with a long burn-in time, followed by relatively short steps ($\sim 2PN$ generations) between each iteration. Ancestral information at the beginning of each iteration is stored, such that the each starting point is a random draw of a population at mutation/selection/drift balance (each iteration uses the burn-in of all previous iterations).

Ex. 5. `$ ./sfs_code 1 1 -L 8 500 1000 R -l p 2000 2000 2000 -1 R`

Moreover, you can **annotate** each locus as being either coding or non-coding, and sex or autosomal. By default all loci are autosomal coding regions. If you would like to specify whether each locus is **coding or not**, use the following option:

annotate coding/non-coding

`--annotate (-a) <a`$_1$`> [<a`$_2$`>..<a`$_R$`>] [R]`

where `a`$_i$ = '`C`' or '`N`' to indicate that the ith locus is coding or non-coding (respectively). If you want all loci to have have the same coding/non-coding annotation, just specify `<a`$_1$`>`. Otherwise, you can either specify the annotation of all R loci, or specify the pattern to be repeated followed by the character '`R`'. To specify whether each locus is **sex or autosomal**, use the following option:

annotate sex/autosome

`--sex (-x) <x`$_1$`> [<x`$_2$`>..<x`$_R$`>] [R]`

which has the same structure as option `--annotate`, but `x`$_i$ = '`0`' or '`1`' to represent autosome or X-linked (respectively).

Note that options `--linkage`, `--annotate`, and `--sex` must be specified **after** indicating the number of loci to simulate using option `-L`.

The **ancestral population size** used in a population genetic simulation is not as important as one might imagine (so long as all parameters are population-scaled, the actual size cancels). However, it can be changed from the default of 500 using the following option.

population size

`--popSize (-N) [P <pop>] <size>`

This option would set the ancestral population size to the value `<size>`. For efficiency sake, the value you use should be kept as small as possible (but no smaller!!). The default is 500 diploid individuals, which should be sufficient for most purposes. However, if you are simulating a distribution of selective effects where the mean of the distribution is greater than the population size (in absolute value), then the entire population might go extinct. A realistic distribution inferred from human polymorphism data might induce such an effect.

4.1 Population Expansions and Bottlenecks

Natural populations fluctuate in population size, and any simulation program should accommodate this biological feature. However, it is often not necessary to simulate the exact trajectory of the population size, just the major trends (i.e. the time of an expansion, or the severity of a contraction along with the degree of recovery). `SFS_CODE` implements four **demographic events**:

1. set the population size to a **new value**:

   population size change

   `-TN <`τ`> [P <pop>] <`N_{new}`>`

2. change the population size by a **relative amount** ($\nu = N_{\text{new}}/N_{\text{old}}$):

   relative size change

   `-Td <`τ`> [P <pop>] <`ν`>`

3. allow the population size to start changing **exponentially**:

```
-Tg <τ> [P <pop>] <α>
```

exponential growth

4. or commence **logistic** growth/decay:

```
-Tk <τ> [P <pop>] <K> <r>
```

logistic growth

Each of these options begin with '-T'. This indicates to SFS_CODE that an evolutionary event will occur at a specific time (<τ>, the first argument). The next character (one of 'N', 'd', 'g', or 'k') indicates the type of demographic event (NOTE: only short names are accepted for timed events). The first argument for these options is the time parameter <τ>. Time is scaled by the effective size of the *ancestral population* (essentially the number of generations *since the end of the burn-in* divided by the number of chromosomes in the ancestral population). Next there is an optional parameter that would allow you to specify a specific population. If you want the demographic event to be applied to all populations (or you are only simulating a single population), then this is not necessary. Otherwise, if you only want to apply the demographic effect to population 0 (see description below on how to simulate multiple populations), then you would use 'P 0' here. Using the character 'P' in your command tells SFS_CODE that the next parameter is a population and not the value for the size change effect.

Finally, if you are using '-TN' include the **new size** of the population <$\mathtt{N_{new}}$>. If you are using '-Td' include the **relative size** change <ν> = new size/current size (note that current size is NOT necessarily the ancestral size if you have multiple changes). If you are using '-Tg' include the **exponential** rate of growth/decay <α>. The parameter α determines the size of the population at time t by the equation $N(t) = N_0 e^{\alpha(t-\tau)}$, where time is scaled by PN_A (the number of chromosomes in the ancestral population, n.b. in a diploid population $P = 2$), τ is the time that the population size started changing, and N_0 is the size of the population when it started changing (not necessarily the ancestral size!). This implies that if you want the population to grow from N_0 individuals to N_F individuals in $(t - \tau) \times PN_A$ generations, you would invert the exponential equation to find $\alpha = \ln\left(\frac{N_F}{N_A}\right)/(t - \tau)$. If you are using '-Tk' for **logistic** growth, include the carrying capacity <K> (the final population size) and the rate to approach it <r>. For logistic growth, the size of the population at time t is determined by the equation $N(t) = \frac{KN_A e^{r(t-\tau)}}{K+N_A(e^{r(t-\tau)}-1)}$.

SFS_CODE is a forward simulation program, so it thinks about time going forward. You can think of the burn-in period as "negative time", with the simulation actually starting at time zero (when the burn-in ends), and progressing forward in generations. Rather than referencing a specific number of generations, however, time is referenced in terms of PN_A generations, where N_A is the ancestral (original) population size and P is the **ploidy** (if you are simulating a diploid population, then $P = 2$ [the default], while $P = 1$ for a haploid population and $P = 4$ is a tetraploid population). You can change the ploidy using the following option:

```
--ploidy (-P) <P>
```

ploidy

where P can be 1, 2, or 4. If P=4, you can specify either autotetraploid population or allotetraploid using

```
--tetraType (-p) <0/1>
```

type of tetraploid

where 0 indicates auto- and 1 indicates allotetraploid.

Keep in mind that the time scaling does not change as the population sizes change (though the amount of evolution taking place each generation can be considerably different). This is similar to `ms`, but instead of having a diploid time scaled in units of $4N_0$ generations (with N_0 the size at the time of sampling), `SFS_CODE` would scale time in units of $2N_A$ generations.

In `SFS_CODE`, it is also necessary to tell the simulation program **when to end** using the option

ending simulation

-TE <τ> [pop]

where again, time (τ) is scaled in units of PN_0 generations. In the simple applications above, the simulation actually ended when the burn-in period was over (i.e. at time $\tau = 0$). In general, you can end the simulation for any population at any time (useful for generating samples from now extinct populations, such as neandertal), but in most situations you will terminate the evolution of all populations when you sample at the end of the simulation. To be more specific, the simulation ends when the last evolutionary event takes place. The "`-TE`" option just allows you to put a place holder until a specific generation.

If you want to simulate a model for an *African* population of humans, you might consider a simple 2-epoch model, where there was a constant ancestral population size (N_A) which instantaneously changed by a factor $\nu = N_\mathrm{C}/N_\mathrm{A}$ some time τ ago (in units of $2N_A$ generations). A diagram of this model is shown in figure 2. To implement this model in `SFS_CODE`, you would consider time during which the population has its ancestral size as the burn-in period. At the end of the burn-in period, the population instantaneously grows by a factor ν, and maintains the new size for $2N_\mathrm{A}\tau$ generations, when the simulation ends. Abstractly, this is implemented in `SFS_CODE` as

Ex. 6. `$ ./sfs_code 1 1 -Td 0` ν `-TE` τ

Notice that the demographic event actually occurs at time zero, with the population maintaining it's new size for τ units of time until the simulation ends. The parameters of such a model were inferred by Boyko et al. (2007) using synonymous SNPs across the human genome from an African American (AA) population. Their inferred demographic model is shown in figure 2. Simulating the AA demographic history using their inferred parameters is easy:

Ex. 7. `$ ./sfs_code 1 1 -Td 0 3.3 -TE 0.4377`

The equivalent command in `ms` would be:

```
ms 12 1 -t 16.5 -eN 0.066 0.303.
```

Note that `ms` requires $\theta = 16.5$. This ensures that the ancestral population has $\theta = 5$, which is the case for the `SFS_CODE` simulations (θ/per site = 0.001 across 5kb).

A simple demographic model for European populations is a 3-epoch bottleneck model. This model is also shown in figure 2, and consists of an ancestral population size (N_A), a bottlenecked population size (N_B), and a current population size (N_C). In `SFS_CODE`, generations begin accumulating when the first demographic event occurs (i.e. $\tau = 0$, when the population decreases in size). The second demographic event occurs at the end of the bottleneck ($\tau^{2}_{\rightarrow} = 7703\mathrm{gen.}/(2N_\mathrm{A}) = 0.48$), and the simulation ends at $\tau^{\mathrm{E}}_{\rightarrow} = 8577\mathrm{gen.}/(2N_\mathrm{A}) = 0.54$. Given these parameters, this model is also straightforward to implement:

Ex. 8. `$ ./sfs_code 1 1 -Td 0 0.722 -Td 0.48 5.27 -TE 0.54`

The corresponding command in `ms` would be:

```
ms 12 1 -t 19.02 -eN 0.00728 0.19 -eN 0.0714 0.263.
```

You can **increase the sample size** using the option

--sampSize (-n) [P <pop>] <SS$_1$> [<SS$_2$>...<SS$_{\text{Npops}}$>] *sample size*

If you are only simulating a single population or you want to sample the same number of individuals from each population, then you can simply use "`-n <SS>`". If you want to set a specific sample size for each of n populations, use "-n <SS$_1$>...<SS$_n$>". Alternatively, if you just want to change the sample size of population i, then use "-n P i <SS$_i$>". Note that *individuals* are sampled, so if you simulate a diploid population (P=2), then 2 chromosomes will be printed at each locus for each individual.

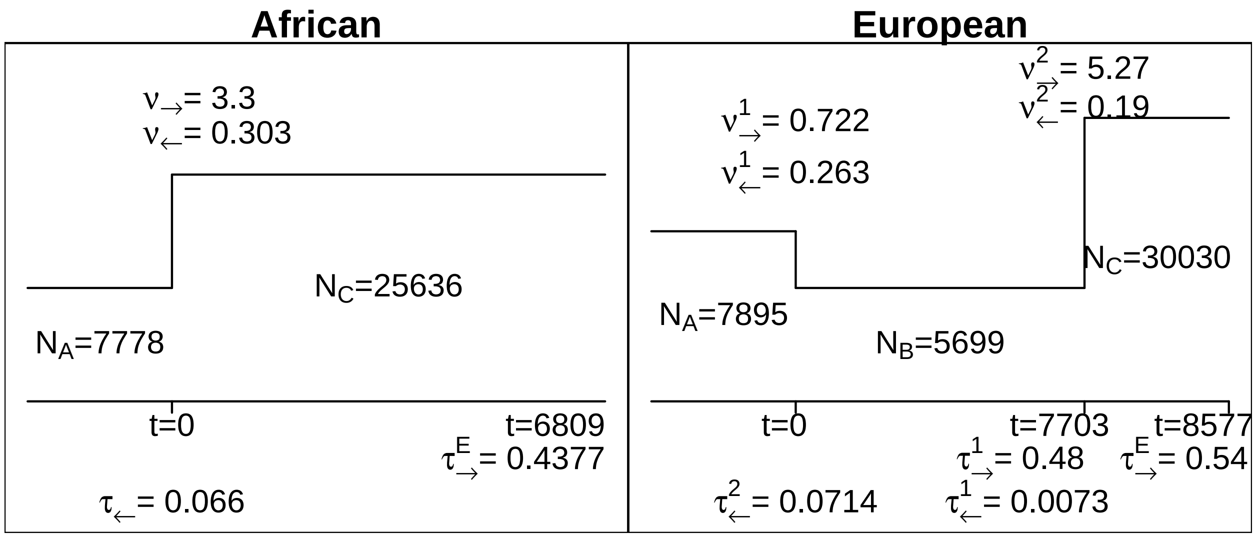
African
European
$\nu_{\rightarrow}$= 3.3
$\nu_{\leftarrow}$= 0.303
N_C=25636
N_A=7778
t=0
t=6809
$\tau^E_{\rightarrow}$= 0.4377
$\tau_{\leftarrow}$= 0.066
$\nu^2_{\rightarrow}$= 5.27
$\nu^2_{\leftarrow}$= 0.19
$\nu^1_{\rightarrow}$= 0.722
$\nu^1_{\leftarrow}$= 0.263
N_C=30030
N_A=7895
N_B=5699
t=0
t=7703
t=8577
$\tau^1_{\rightarrow}$= 0.48
$\tau^E_{\rightarrow}$= 0.54
$\tau^2_{\leftarrow}$= 0.0714
$\tau^1_{\leftarrow}$= 0.0073

Figure 2: The simple demographic scenarios considered considered in section 4.1. Parameters (τ and ν) with subscript $\rightarrow$ are for `SFS_CODE` (forward time), while those with subscript $\leftarrow$ are for `ms` (pastward time). The horizontal axis represents time in generations (with $t = 0$ at the first demographic event). To obtain $\tau_{\rightarrow}$, divide the accumulated number of generations by $(2 \times N_A)$. To obtain $\nu_{\rightarrow}$ divide the new population size by the current population size at each transition. This methodology differs from `ms`, where the population size at time of sampling is generally the base. The number of generations and the effective population sizes for both populations were inferred by Boyko et al. (2007).

4.2 Distribution of Selective Effects

One of the many important components of a forward population genetic simulation program is natural selection. `SFS_CODE` assumes a simple multiplicative model of genic selection. This means that the fitness of an individual is just the product of the fitness effects of each mutation they carry. In general, a new mutation will have fitness $1+s$, where s is the selective effect ($s>0$ indicates positive selection, $s<0$ indicates negative selection, and $s=0$ indicates neutrality). An individual that is homozygous for such a mutation would then have fitness $(1+s)^2$. The selection coefficient is related to the population scaled selection coefficient $\gamma = 2N_e s$. Because population genetic theory is generally based on inference of γ, `SFS_CODE` draws γ from a specified distribution (discussed below), then divides it by PN_{C}, the number of chromosomes in the population when the mutation arises (note that P is the ploidy, which is 2 for the default diploid population). `SFS_CODE` then uses s to determine the fitness of each individual, and normalizes by the mean fitness in the population.

It is important to note that `SFS_CODE` only implements *shift* models of selection. This means that as soon as a selected mutation is fixed in the population, the fitness effect of the site returns to 1. This avoids problems such as Muller's Ratchet, where the accumulation of deleterious mutations drives the population into the ground. Shift models are also in contrast to models such as the *House of Cards* model that was developed by T. Ohta in the 1960s (whereby assuming a normal distribution of selective effects will eventually lead to the fixation of an allele with selective effect ≥ 8 standard deviations above the mean, at which point evolution nearly halts).

You can **specify the distribution of selective effects** using the following option:

selective effects

```
--selDistType (-W) [P <pop>] [L <locus>] <type> [args]
```

where `<type> [args]` are outlined in Table 1 on page 10, and the optional flags '`P`' and '`L`' allow you to specify a single population or locus (respectively, if simulating more than one population or locus). For example, to simulate rampant positive selection, where all new nonsynonymous mutations have $\gamma = 5.0$, you would use

Ex. 9. `$ ./sfs_code 1 1 -W 1 5.0 1.0 0.0`

To simulate a situation in which 70% of new nonsynonymous mutations are deleterious with $\gamma = -5$, 10% are advantageous with $\gamma = 5$, and the remainder are neutral, you would use:

Ex. 10. `$ ./sfs_code 1 1 -W 1 5.0 0.1 0.7`

Table 1: Selection: arguments for option `--selDistType` (`-W`)

`<type>`	`[args]`	description
0	$\emptyset$	**Neutral** (gamma = 0 for all mutations).
1	`<GAMMA> <p_pos> <p_neg>`	**3-point mass model**. Single γ (>0) for both deleterious and advantageous mutations. With probability `<p_pos>` the sign is positive, with probability `<p_neg>` it is negative, otherwise with probability `1-<p_pos>-<p_neg>`, $\gamma = 0$.
2	`<p_pos> <aP> <lP> <aN> <lN>`	**Gamma (Γ) distributions**. With probability `<p_pos>` $\gamma \sim \Gamma($`<aP>`,`<lP>`$)$ (mean = `aP/lP`, var. = `aP/lP`^{2}), otherwise $\gamma \sim -\Gamma($`<aN>`, `<lN>`$)$.
3	`<mean> <var>`	**Normal distribution**. Mean = `<mean>` and variance = `<var>`.
4	$\emptyset$	**Advanced option**. Predefine distribution in file gencontextfreq.c, see text.

For a more complicated scenario, in which you want a distribution of positive and negative selection, we have `<type>=2`, which implements a mixture of Gamma distributions ($\Gamma(\cdot)$), one that corresponds to positive values of γ and one that has been reflected across the y-axis to capture a distribution of negative values. For example, if you want to assume that 90% of new nonsynonymous mutations are deleterious with a selection coefficient drawn from $\Gamma(1,1)$ (a simple exponential distribution) and the remaining 10% are advantageous and drawn from $\Gamma(50,10)$ (having mean = 5 and variance = 0.5), then you could use the following example.

Ex. 11. `$ ./sfs_code 1 1 -W 2 0.1 50.0 10.0 1.0 1.0`

Note that for example 11, the distribution of deleterious effects reduces to an exponential distribution, while the distribution of advantageous effects has a mean and mode at 5. This mixture distribution is shown in figure 3. Of course if you simply want a Γ-distribution of negative selection (assuming no positive selection), then you can simply set `<p_pos>` = 0.

The fourth `<type>` (number 3), is a simple normal distribution. With a mean of zero, Cutler (2000) refers to the normal distribution of selective effects a model of positive selection. This is because on average, half of the new mutations will be advantageous, and a majority of the deleterious mutations will be eliminated.

The final `<type>` of model for the distribution of selective effects is an "advanced" option. For this option, you can create as complicated a distribution as you'd like in another statistical package (`R`, for example). This distribution can be discretized into 100 bins of equal density (using the `quantile` function in `R`, for example). These 100 bins are then copied into the vector `fitQuant` that is stored in the file `gencontextrate.c`. After changing this vector, the program must be recompiled (this is the only reason that it is referred to as an "advanced" option... more realistically, it is a rudimentary option that requires more work, but provides the ultimate flexibility). This model is actually preferred to `<type>=2`, as it is much quicker to randomly sample from a discretized distribution than it is to draw from a mixture of Γ-distributions. However, population size changes cannot be accommodated with this option.

It is also possible to specify that one population remain a neutral population. This can be useful if you want to specify a common distribution of selective effects for all populations but one. This is done using

```
--neutPop (-w) <pop>
```

neutral population

Selective Effects with Demography

When population sizes change, the relative effect of selection changes (selection is stronger in a larger population). This is accommodated by altering the distribution of the population scaled selection coefficient. For constant (type 1) and normal distributions (type 3) this is easily accommodated by adjusting the mean. For a mixture of Gamma distributions (type 2), the λ_N and λ_P parameters are adjusted such that the new means correspond to the change in population size. This also affects the variance, in accordance with the Gamma distribution. However, for the custom distribution of selection coefficients (type 4), population demography cannot be accommodated. If a custom distribution of selection coefficients is used and the population sizes change, then the same distribution of γ will be used (thereby inflating/deflating s to maintain a constant value of γ).

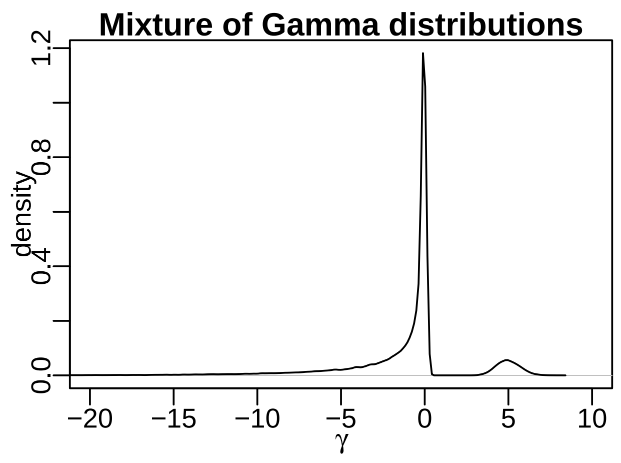

Figure 3: A distribution of selection coefficients, where 90% of new mutations would be deleterious with $\gamma \sim -\Gamma(0.231, 0.1279)$, and the remaining are drawn from $\gamma \sim \Gamma(50, 10)$.

Selective Constraint

In the way that Kimura outlined the *neutral model of evolution*, some proportion of nonsynonymous mutations are completely lethal, and never contribute to polymorphism. All other nonsynonymous mutations were completely neutral, and had no selective effect at all. As a result, it is often of interest to simulate data under such a neutral model (or allowing some proportion of nonsynonymous mutations to be lethal in general while the remaining nonsynonymous mutations follow the specified distribution of selective effects). This also generalizes to non-coding regions, where some proportion of mutations can be lethal. In Kimura's model, the parameter f_0 represents the proportion of neutral mutations. In SFS_CODE, you can adjust the **non-lethality parameter** using the following command.

non-lethal mutation rate

```
--constraint (-c) [P <pop>] [L <locus>] <f0>
```

This option can even be used when simulating non-neutral models of evolution, as a way of signifying that only some mutations will contribute to polymorphism.

The way this option works, is that for each nonsynonymous or non-coding mutation, with probability $1 - f_0$, the fitness effect will be -1. This effectively sets the fitness of the individual to zero (as the fitness of the individual is defined as $1 + s$). This means that any mutations that are unique to this individual will also be lost in the next generation, as it will not pass on any of its gametes. All synonymous mutations are assumed to be neutral (i.e. none are considered lethal).

4.3 Multiple Populations

In the above examples, we have used exclusively a single population, with `<Npops> = 1` as the first parameter into SFS_CODE. If we change this parameter, then we can simulate multiple populations. Note that populations are numbered from 0 through `<Npops>`-1.

There are two ways to create new populations. You can either have a speciation event or a domestication-style event. For a **speciation event**, one population will be split into two identical populations (equal size, etc.). To split population `i` into two populations (`i` and `j`) at time τ, you use the following template.

speciation events

```
-TS <τ> <i> <j>
```

For a **domestication event**, one population (`i`) will be split into two (`i` and `j`), but the second population will primarily be composed of individuals that carry a particular derived allele, chosen at random from all the alleles that have a specified frequency (within 5% of `<allele_freq>`). After choosing a particular allele from the founding population, `SFS_CODE` will randomly sample individuals that are homozygous for the allele. If there are not enough homozygous individuals, then it will choose from the heterozygous individuals. If there are still insufficient individuals, then it will randomly choose non-carriers, until the specified population size, `<N>` is reached (note that `<N>` must be less than the size of the parent population `i`). The template for this option is as follows.

domestication events

```
-TD <τ> <i> <j> <allele_freq> <N> [locus]
```

If a `locus` is specified, then `SFS_CODE` will try to find an allele in that particular locus (not necessary if only simulating a single locus). If `locus` is not specified, then `SFS_CODE` will start at the center-most locus that is simulated. If there isn't an allele near the specified `allele_freq`, `SFS_CODE` will search adjacent loci until one is found. Failing to find any mutations at the specified frequency, `SFS_CODE` will select the allele that is closest in frequency.

Now that multiple populations have been initialized, it is essential to tell `SFS_CODE` when to **end the simulation**. This was mentioned above with regards to demographic effects, but is worth mentioning again. This is done using the familiar option `-TE <τ> [pop]`. As an example, say you wanted to simulate human polymorphism data with a chimpanzee outgroup (assuming a population scaled divergence time of $\tau = 10$ and an allopatric speciation event). You could use the following:

Ex. 12. `$ ./sfs_code 2 1 -TS 0 0 1 -TE 10`

This example would first generate a single population at stationarity during the burn-in. At the end of the burn-in ($\tau = 0$), the population would be split into two identical populations, which would evolve independently until the end of the simulation ($\tau = 10$).

As an example of a domestication event, consider a model for dog breed formation, where you also want to simulate the ancestral dog population. This model is characterized by a major bottleneck in the ancestral population followed by rapid growth. Then, after growing for some time, 2 new breeds (of size 100 and 10) are formed using alleles at frequency 0.1 and 0.01 (respectively) in the ancestral population. These new breeds are then simulated for $0.1 \times 2 \times 500 = 100$ generations.

Ex. 13.
```
$ sfs_code 3 1 -Td 0.0 P 0 .1 -Tg 0 P 0 2 -TD 2.5 0 1 0.1 100 \
  -TD 2.5 0 2 0.01 10 -Tg 2.5 P 1 10 -Tg 2.5 P 2 15 -TE 2.6
```

Let's walk through this example step by step. First, `sfs_code 3 1` indicates that we are going to simulate a total of 3 populations for 1 iteration. Next `-Td 0.0 P 0 .1` indicates that there is going to be a demographic event at the end of the burn-in period for population 0. This demographic event will shrink the population to 1/10th its size. After the major contraction, `-Tg 0 P 0 2` indicates that population 0 will start exponentially growing at a rate of 2 per generation (the backslash '\' indicates that the command stretches onto the next line and can be ignored). Then, after 2.5 units of time, two new breeds are formed from this ancestral breed. Population 1 is created by `-TD 2.5 0 1 0.1 100`, indicating that an allele at frequency 0.1 in the parental population was used to form a population of 100 individuals. Population 2 is created by `-TD 2.5 0 2 0.01 10`, indicating that an allele at frequency 0.01 is used to form a population of size 10. Both breeds then start growing at an exponential rate (population 1 at a rate of 10, while population 2 grows at a rate of 15). Then, after another 0.1 units of time (100 generations, or approximately 200-300 years), the simulation ends and we draw the default of 6 individuals from each population. This simulation takes less than 5 seconds on a 2.33 GHz Intel Core 2 Duo MacBook Pro.

Migration

Individuals are free to migrate to any extant populations. The migration rate matrix indicates the average number of individuals in each population that are composed of individuals from each of the other populations. For the migration matrix $\mathbb{M}$, the (i, j) entry $m_{i,j}$ represents the expected number of individuals in population i that came from population j (this is also referred to as the "backward migration rate matrix"). To set the migration rate, you would use the command `--migMat` (`-m`). There are three ways to set the values of the migration matrix, indicated by the first argument to the option being either '`A`', '`P`', or '`L`'. You can set **A̲ll entries** to be the same value `M`:

migration rates

all rates equal

```
--migMat (-m) A <M>
```

Note that this option specifies a symmetric island model, where the number of migrants into population i is M. So, for `<NPOP>`=3, there would be $M/2$ migrants from both of the other two populations. You can also set the migration rates explicitly **from one P̲opulation to another**:

population specific

--migMat (-m) P <$\mathtt{P_{to}}$> <$\mathtt{P_{from}}$> <M>

which would specify that the average number of migrants into population $\mathtt{P_{to}}$ from $\mathtt{P_{from}}$ is `M`. Finally, you can **L̲ist the entire migration matrix**:

list all matrix entries

--migMat (-m) L <$M_{0,1}$>...<$\mathtt{M_{NPOP,NPOP-1}}$>

which would set each entry of the matrix. Note that the *diagonal entries are not specified.* For example, if you have 3 populations and want to use option '`L`', you should specify all 6 entries: $M_{0,1}, M_{0,2}, M_{1,0}, M_{1,2}, M_{2,0}, M_{2,1}$.

In `SFS_CODE`, a Poisson number of individuals are chosen to migrate from population j to population i each generation with expected value $M_{i,j}$. Each migrant out of population j will be male with probability `pMaleMig`. You can set the **male migration rate** using

male migration rate

--pMaleMig (-y) [P <pop>] <$\mathtt{p_{male}}$>

By default, $\mathtt{p_{male}}$=`1-propFemale`, corresponding to the proportion of males in the originating population. By default, this is 0.5, but you can **change the proportion of females in a population** using

proportion of females

```
--propFemale (-f) [P <pop>] <pf>
```

This can be set for all populations simultaneously, or for a given population explicitly.

4.4 Mutation Models

There are 6 mutation models built into `SFS_CODE`. The basic initiation of a mutation model is as follows.

```
--substMod (-M) <mod> [args]
```

substitution model

Table 2 outlines the models and arguments for this option. The most basic mutation model (`<mod>` = 0) was proposed by Jukes and Cantor (1969), and referred to as **JC69**. This model assumes that the rate of mutation is equal among all nucleotides. A simple modification of this model was proposed by Kimura (1980) to account for the observation that most mutations tend to be transitions (A$\leftrightarrow$G or C$\leftrightarrow$T). This model (`<mod>` = 2) adds another parameter (the transition/transversion bias, κ), and is referred to as the Kimura 2-parameter model (or just **K2P**). An extension of the K2P model would be to allow a transition/transversion bias for each nucleotide (i.e. the rate of A$\rightarrow$G is not equal to the rate of C$\rightarrow$T). Zhang and Gerstein (2003) fit the parameters of such a model to human data. This model has been implemented in `SFS_CODE` as `<mod>` = 4.

One feature of mammalian genomes is the presence of hypermutable CpGs (due to the deamination of methylated C's that are immediately 5' of a G). `SFS_CODE` implements a CpG extension to both the JC69 model and the K2P model (`<mod>` = 1 and 3, respectively). This is implemented by rejecting mutations at non-CpG sites with probability `<PSI>`. Given a non-CpG site is rejected, a new site will be picked to mutate until either finding a CpG or accepting a non-CpG site. Once accepting a site to mutate, it will either mutate to a new nucleotide randomly (in the case of `<mod>`=1) or to a transitional nucleotide at a rate equal to <κ> (in the case of `<mod>`=3). For substitution models 1, 2, and 3, the mutation parameters (ψ and κ) can also be set for a single population using the following option.

--KAPPA (-K) [P <pop>] <κ>

set the value of κ

--PSI (-C) [P <pop>] <ψ>

set the value of ψ

The most detailed model that is implemented in `SFS_CODE` is `<mod>`=5. This is a full context-dependent substitution model, where the site-specific rate of mutation depends on both of its adjacent nucleotides. This accounts for mutation rate variation due to CpGs as well as other context-effects found by Hwang and Green (2004). Conditional on picking a site to mutate, the replacement nucleotide will also depend on the flanking nucleotides. Choosing a new site to mutate is done using an inverse-CDF method, where relative hit-probabilities are defined by the cumulative site-specific mutation rates.

More generally, any trinucleotide substitution model can be used by updating the 64$\times$4 rate matrix Q in the file `gencontextrate.h` and recompiling the program.

While `SFS_CODE` is based on simulating finitely many sites, it is also possible to simulate data under a pseudo-infinitely many sites model. It is pseudo because multiple hits can occur, but no more than one mutation will be segregating at a site at any given time. This is specified using the following option.

```
--INF_SITES (-I)
```

infinite sites model

Table 2: Mutation models: arguments for option `--substMod`

`<mod>`	`[args]`	description
0	$\emptyset$	**JC69** model of equal mutation rates to and from all nucleotides.
1	`<`ψ`>`	**JC69+CpG** Simple model of hypermutable CpGs, where `<`ψ`>` is the non-CpG rejection rate.
2	`<`κ`>`	**Kimura 2-parameter** model, with `<KAPPA>` the transition-transversion bias.
3	`<`κ`>` `<`ψ`>`	**K2P+CpG** combining model 1 and 2.
4	$\emptyset$	**ZG2003** the generalized K2P model, where each nucleotide has its own transition/transversion bias (all parameters inferred by Zhang and Gerstein (2003)).
5	$\emptyset$	**Context-Dependent** model, where the mutation rate at each nucleotide depends on both of its adjacent neighbors (all parameters inferred by Hwang and Green (2004)). This is the model `SFS_CODE` was named after.

Mutation Rate Variation Across Sites and Loci

Context-dependent mutation models impose mutation rate variation along a sequence. However, not all species show evidence for such a mutation process (e.g. *Drosophila*), despite having mutation rate variation. For such species, mutation rate variation has in the past been modeled as a discretized Γ distribution across sites. `SFS_CODE` allows you to simulate under such a model, allowing both sites as well as loci to have a mutation rate scaled by a discretized Γ distribution (with mean 1). These are implemented in the following options.

mutation rate variation across sites & loci

`--rateClassSites (-V) [P <pop>] <n`$_{\texttt{classes}}$`> <`α`>`

`--rateClassLoci (-v) [P <pop>] <n`$_{\texttt{classes}}$`> <`α`>`

These options allow you to specify a certain number of mutation rate classes ($\texttt{n}_{\texttt{classes}}$), which will be drawn from a $\Gamma(\alpha, \alpha)$ distribution (having mean 1 and variance $1/\alpha$).

4.5 Selfing and Generation-Effects

`SFS_CODE` generally assumes that all populations are randomly mating (subject to their relative fitnesses). However, in plant species in particular, mating is not random, such that an individual may be more likely to self-fertilize than to mate with another (an ultimate form of inbreeding). To accommodate this, `SFS_CODE` allows the user to specify a selfing rate, `s`, for each population using the following option.

selfing rate

`--self (-i) [P <pop>] <s>`

Moreover, when simulating multiple species, it will not always be the case that they will have the same generation time. For example, today, humans have a longer generation time than most other primates (especially the non-apes). To account for this, `SFS_CODE` provides a generation effect option.

```
--GenEffect (-G) <pop> <G>
```

generation effects

For the generation effect, `G` must be an integer (≥ 1 or ≤ -2). If it is positive, then the indicated population will experience `G` rounds of mating each generation. If `G` is negative, then the indicated population will only have a round of mating every `|G|` generations. For example, setting `G`=2 would shrink the generation time by half (leading to 2 rounds of random mating every generation), while setting `G`=-2 would double the generation time (leading to one round of random mating every second generation). At least one population must have `G`=1.

4.6 Changing Parameters Over Time

In `SFS_CODE`, many of the parameters can be changed at any time during the course of the simulation. Table 5 outlines all the options that have been implemented in `SFS_CODE`, and any option that has an asterisk in the short name (third column) can be changed (or initiated) at any time using the following option.

```
-T<short_name> <τ> [args]
```

timed events

This means that if you are, for example, studying domesticated rice, and want to model the transformation to a primarily selfing organism, you might consider a population which starts with a low selfing rate, but $2N$ generations ago became 99% selfing. This could be achieved as follows.

Ex. 14. `./sfs_code 1 1 -TE 1 -i 0.2 -Ti 0 0.99`

Example 14 would simulate data assuming the selfing rate was 0.2 until the burn-in time ended, at which point the selfing rate would change to 99%. The simulation would then end after another $2N$ generations.

When using `-T*`, the option retains all the functionality as described above and in Table 5. For example, consider simulating 2 populations, that diverged $10{\times}2N$ generations ago (i.e. human-chimp divergence). Suppose you wanted to model recent positive selection (e.g. within the last $2N$ generations) in the human genome while ancestral populations and chimpanzee are completely neutral. You might try the following example.

Ex. 15. `./sfs_code 2 1 -TS 0 0 1 -TE 10 -TW 9 P 0 1 5 1 0`

Example 15 would generate 2 populations, which split at time $\tau = 0$, and evolve independently for $10{\times}2N$ generations (`-TE 10`). However after having diverged for $9{\times}2N$ generations, all new nonsynonymous mutations in population 0 would be advantageous with $\gamma = 5$. In this case, the command "`-TW 9 P 0 1 5 1 0`" literally means change the distribution of selective effects at time $\tau = 9$ for population 0 to `type`=1, γ=5, `p_pos`=1, and `p_neg`=0.

Keep in mind that timed events only work with the short names, so you could not use `-TselDistType`, for example.

5 The Non-Effect of the Effective Population Size

One challenge you will face when running forward simulations, is to pick an effective population size. In coalescent theory, this is a non-issue, as the results have been derived for limiting cases when the population size tends to infinity while parameters tend to zero, such that the product stays constant (isn't it nice that all parameters in population genetics being scaled by the population size?). However, in forward simulations, the actual population size used can become an issue, depending on what you are trying to model.

This section shows you that in most situations, the actual population size doesn't matter. In general, you should do a few simulations with larger and smaller population sizes to show that the population size used does not affect your simulations. Failing to do so could lead to a false interpretation of the results. However, it is always helpful to use the smallest population size possible, as this will make the simulations run quicker.

We will consider **varying the effective population size**. This is done using the following option.

```
--popSize (-N) [P <pop>] <size>
```

We will consider populations of size $N \in \{250, 500, 1000, 5000, 10000\}$. This should give us an idea of what is going on. We will consider just a couple of statistics: the distribution of the total number of polymorphisms and the average SFS. We will evaluate populations of constant size, as well as populations that have recently either grown or shrunk 10-fold (magnitude of change $\nu = 10$ or 0.1, approximately $0.1 \times 2N$ generations ago). We will also consider neutral models as well as selection under both `selDistType` 1 and 2 (`-W 1 5 0.1 0.8` and `-W 2 0.1 50 10 0.23 0.1279`, respectively). Considering just the case of no recombination yields 45 combinations (5 population sizes $\times$ 3 demographic models $\times$ 3 selective effects). The general command line looks like the following example (note that we are sampling 20 diploid individuals, or 40 chromosomes).

Ex. 16.
```
$ ./sfs_code 1 2000 -n 20 -N <N> -Td 0 <ν> -TE <τ> \
  -W <type> [args]
```

Figure 4 summarizes the results. As you can see, for these demographic and selective effects, **the effective population size used has no impact on the distribution of SNPs or the SFS**. For the cases shown here, the only reason the curves do not perfectly overlap is because 2000 simulations is insufficient to capture the true distribution. A very similar figure was generated (but not shown) with recombination ($\rho = 0.01$/site) with identical results.

6 Sampling From an Extinct Lineage

Recent advances in the extraction of ancient DNA has lead to the partial sequencing of the Neanderthal genome. This allows us to gain further insight into homo relatives, and actually learn more about our own species. One of the many questions that gains a lot of interest is whether or not humans mated with Neanderthals, and whether there is any evidence for or against it in our genomes. Understanding whether or not current statistical methods will have the power to detect evidence of such mating (or how much migration there would have had to have been to detect it) lies crucially in the hands of population genetic simulations.

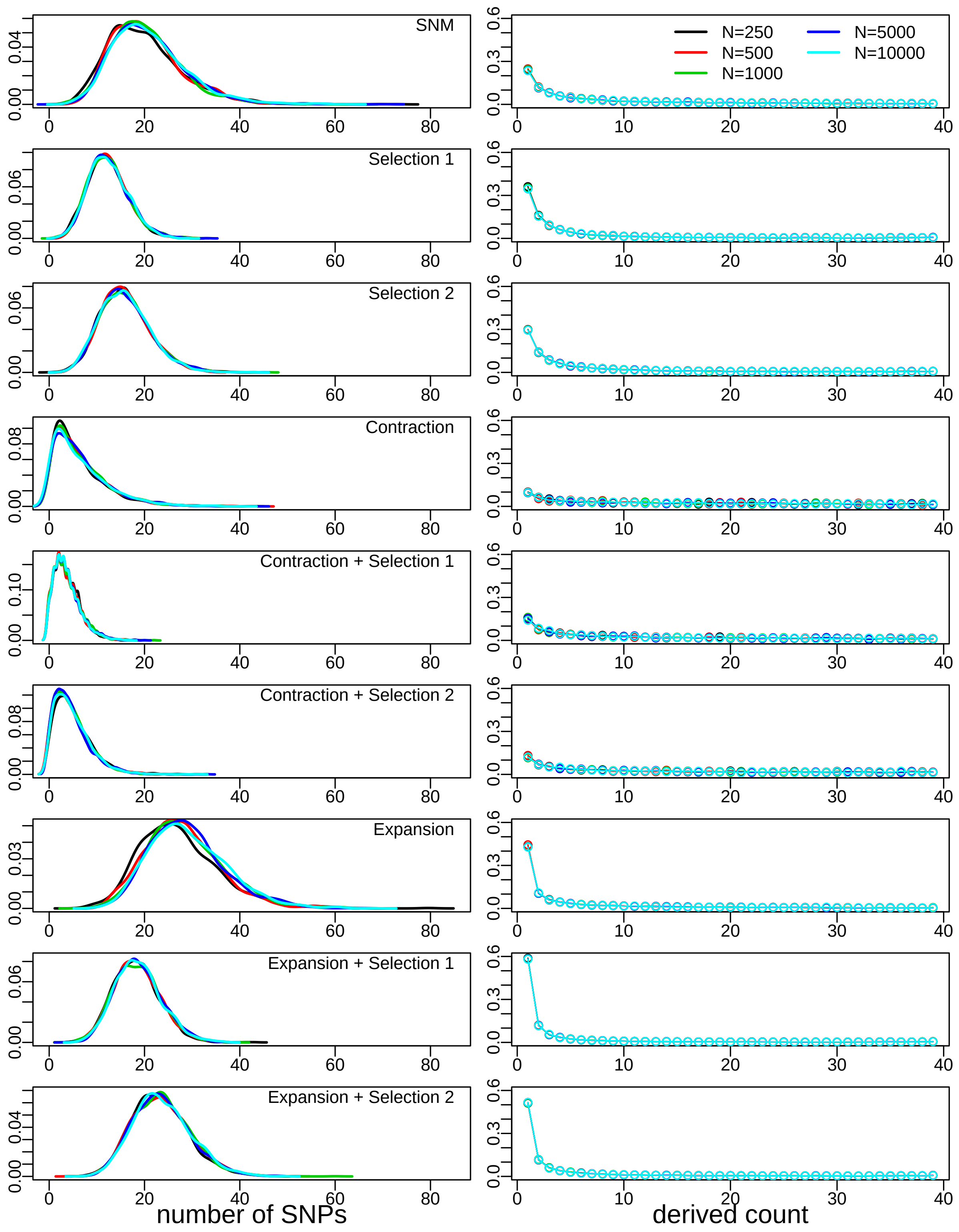

Figure 4: The non-effect of the effective population size in `SFS_CODE`. Each panel has 5 curves, corresponding to different values of N_e (shown in legend in the upper-right plot). Each row corresponds to a different set of assumptions (demography/selection). Left is the distribution of the number of segregating sites, and the right is the average SFS across 2000 simulations.

Assuming that you've been able to simulate more than one population, sampling from an extinct lineage is actually dead easy (pun completely intended). You must simply "kill" one of the populations using the option `-TE <tau> [pop]`. Take the following example.

Ex. 17. `$ ./sfs_code 2 1 -TS 0 0 1 -TE 0.1 0 -TE 0.5`

During the burn-in, a single population would reach stationarity. At the end of the burn-in, the populations would allopatrically split (`-TS 0 0 1`). After $0.1 \times 2N$ generations, population 0 would effectively die (simulations for this population would stop). All the individuals from this population are still in memory, but evolution in this population ceases (all individuals would, quite literally, remain frozen in their prehistoric state). After an additional $0.4 \times 2N$ generations, the simulation completes. At completion, the default of 6 individuals will be sampled from the extinct lineage (population 0) and 6 individuals will be sampled from the other population (you could consider this the human population).

For a slightly more detailed simulation, consider the human-Neanderthal-chimpanzee alignment, where only a single individual is chosen from each species. This could be implemented as follows.

Ex. 18. `$ ./sfs_code 3 1 -TS 0 0 1 -TS 8 1 2 -TE 9 2 -TE 10 -n 1`

Step-by-step, this would perform a single simulation of 3 populations (`./sfs_code 3 1`). At the end of the burn-in, the ancestral population splits into two populations (e.g. the chimp and human-Neanderthal ancestor: `-TS 0 0 1`). After $8 \times 2N$ generations, humans and Neanderthals split (`-TS 8 1 2`). After another $2N$ generations, the Neanderthal population suddenly goes extinct (`-TE 9 2`). Finally, after a total of $10 \times 2N$ generations, the simulation ends, so we sample 1 individual from each species (including the extinct one).

Rather than have the Neanderthal population suddenly go extinct, it might be more useful to have them die out at an exponential rate, or something. Just as in section 4.1, we could add a growth (or death) rate of population 2 to -2 at time $\tau = 8.5$ using `-Tg 8.5 P 2 -2`.

After simulating data, one could use `convertSFS_CODE` to analyze patterns of shared haplotypes, patterns of ancestral and derived alleles, etc. One could also consider humans and Neanderthals to be two populations with very high rates of migration, but negative selection acting strongly on the Neanderthal population until they are "bred out".

7 Using `SFS_CODE` on a Cluster

7.1 Your own Cluster

Any large scale simulation study cannot be performed without a large number of processors. Even coalescent simulations require a cluster in some situations, such as when using approximate likelihood techniques [e.g. Hernandez et al. (2007a); Caicedo et al. (2007)]. However, if you do not have access to a cluster, or are unfamiliar with how to run jobs on a cluster, then this section will be insufficient for you (try the next section, which deals with using the CBSU cluster at Cornell University).

Being a forward simulation, `SFS_CODE` works well when identical jobs are sent to multiple processors, as they are all run independently. However, because the burn-in period takes a considerable amount of time, it is often beneficial to run several iterations on each processor (to take advantage of the short successive draws after the burn-in, as shown in figure 1). It therefore becomes a balance between the number of iterations to run on each processor versus the number of processors at your disposal to provide the most efficient results.

Let's consider the case where you want to run 2000 iterations of a single simulation. One way to complete the task is to split it up into 200 jobs, each of which runs 10 iterations. Each of the 200 jobs can then be submitted to a different processor successively. When all the processors have written their output, you could concatenate all 200 output files into a single file for analysis.

Setting the Seed

When running simulations on several processors simultaneously, there is a very nontrivial probability that some jobs will start at the same time. These jobs would then have the same seed for the random number generator, and would therefore produce *exactly* the same results. This is very bad, every simulation must have a different seed. For any given simulation, you can **change the seed** using the following option.

```
--seed (-s) <value>
```

random number seed

Depending on the configuration of your cluster, you may be able to generate a `SEEDFILE` with 200 lines (using `perl` or the system defined `RANDOM` shell variable), each line containing a unique seed that each process can pluck from. Alternatively, you may need to generate a `TASKFILE` with 200 lines, each line containing the entire `SFS_CODE` command, but with a unique seed. In the latter situation, you could use another program (written using `MPI`) to dynamically distribute each of the tasks across the available processors (not provided, but these so called "master-slave" algorithms are commonly used and can be downloaded from other sources).

Setting the seed is also useful if you ever want to reproduce a set of simulation results exactly.

Distributing the Work

Included in the distribution of `SFS_CODE` is an example `perl` script (`genSEEDS.pl`) for generating unique seeds (the `SEEDFILE`), an example `perl` script (`makeTask.pl`) for creating a `TASKFILE`, and an example `shell` file (`run_sfs_code_array.sh`) that could be used to run an *array* of jobs on a Sun Grid Engine Cluster. See comments in these files for more details. If you use something other than the Sun Grid Engine (SGE), then you are on your own (unless you are using the CBSU clusters described in the next section), as I am probably not familiar with it.

Assuming that you have already compiled the program and know where it is located on the cluster, you should be ready to generate some data. Change directory to the folder containing the 3 files mentioned in the previous paragraph. The basic procedure would be to generate the `SEEDFILE`:

```
perl genSEEDS.pl <Nseeds>
```

where `<Nseeds>` is the number of simulations you want to perform. Next, if you want to do several simulations, with each one varying a certain number of parameters, it might be helpful to make a `TASKFILE`:

```
perl makeTask.pl
```

Note that `makeTask.pl` will have to be updated to incorporate the parameters that you are interested in. Next, you will want to update the `shell` file `run_sfs_code_array.sh` to either extract information from the `TASKFILE` or to contain the specific `SFS_CODE` command that you want to simulate. Finally, you will want to submit the `shell` file to the cluster.

7.2 Using SFS_CODE on the CBSU Cluster

`SFS_CODE` has a dedicated webpage at `http://cbsuapps.tc.cornell.edu/ sfscode.aspx`. From here, it is possible to submit your simulation jobs to the cluster hosted by the Cornell University Computation Biology Service Unit (CBSU). The interface is simple. Enter your email address (to have a link with simulation results emailed to you), and a single set of `SFS_CODE` arguments. All you have to do is indicate the total number of populations you want to simulate (`Npop`), the total number of successive simulations that you would like to run on each process (`Niter`), the total number of repetitions you want to run (`Nrep`), and the total number of nodes you want to use. Then, copy-paste your command-line arguments into the text box. A total of `Nrep`$\times$`Niter` simulations will be run automatically, each with a different seed. You can only paste a single line into the text box. Running several different sets of simulations requires submitting several jobs.

The command will then be distributed across the requested number of processors. Upon completion, output will be concatenated into a single file, zipped, and stored on a server until downloaded (a link to the location will be emailed to you). Depending on the number of jobs in the queue, your job may take up to a few days to begin.

The CBSU clusters have a 24 hour time limit. This means that any job that you submit to the cluster must be completed within 24 hours. It is good practice to become familiar with the length of time that your job will take by running a few iterations locally (also helpful to ensure that your command-line works). You could then multiply the average amount of time per job by the total number of tasks you wish to perform and divide by the total number of processors. If this is greater than 24 hours, consider splitting the job into two or more sets.

8 Understanding the Output

A lot of information is stored during the course of an `SFS_CODE` simulation that will be useful in many different situations (yet useless in others). Unfortunately, this makes for a lot of output. In order to make the output file as concise as possible, a fairly complicated format is needed. However, I've also produced `convertSFS_CODE`, a program that will convert the `SFS_CODE` output to a more useful format (see next section). However, it is important to know all of the information that is contained in the output in the event you want to perform a type of analysis that has not yet been implemented in `convertSFS_CODE`. An example of the output looks something like the following:

```
./sfs_code 1 2 -L 2 66
SEED = -382034166
//iteration:1/2
>locus_0
GTTCCAGGAAGCTGGACAGTCTCTTATGGCGACATGGTAAATAAATTTGCGGTCCTGAAATCGGGT
>locus_1
AATGGGTTAGATTATGATTATGTGCATCGTCTTTACACGAGTGGAGTCATTCGACTTTTCGTACTA
Nc:500;
MALES:3;
0,A,24,-237,0,TTA,A,1,Y,N,0.0,2,0.1,0.8;
//iteration:2/2
>locus_0
TTTTCAGTCTGTTTTGTCAAAGATTATTCTTTTGGGCTCCTCACGCACCTTAAGAAGTGTATATAC
>locus_1
TACGCTATCAACTACAATATACATAGTGTGGTTTTCGATGGCCTTAGGTCAGTTGACCTACGTAAC
Nc:500;
MALES:3;
1,A,28,-523,0,GTG,A,1,V,E,0.0,2,0.1,0.3;
```

The first line has the command line used to call SFS_CODE. In this case I've asked SFS_CODE to generate two iterations of a single population, with two loci, each of length 66 basepairs. The second line includes the value of the seed for the random number generator (this can be used to reproduce the results, though the version used to generate this output may be different from the one you are using so you may not be able to reproduce this output exactly). The third line starts the results of the simulations. Each iteration that is simulated starts with "`//iteration:`", followed by the iteration number and the total number of iterations being performed. The fourth line starts a fasta-style representation of the nucleotide sequences at each locus. The next line reports the final population size (N_c) of each population in a comma delimited list terminated with a semi-colon. The next line ("`MALES:`") provides the *index* for the first male that was sampled (*not the number of males in the sample*), in this case indicating that individuals 3, 4, and 5 are male while individuals 0, 1, and 2 are female (note that these are diploid individuals, so chromosomes 0-5 belong to females and chromosomes 6-11 belong to males). The next line starts the information regarding every mutation that contributes information to the sampled sequences in a comma delimited list terminated with a semi-colon. There are at most 20 mutations per line, so mutations can span several lines. The information provided for each mutation are as follows:

1. locus that the mutation arose on (zero-based)
2. 'A', 'X', 'Y', indicating Autosomal, X-, or Y-linked mutation, respectively
3. position of mutation in locus (zero based)
4. generation mutation arose (negative for mutations arising during burn-in)
5. generation mutation fixed in population (or time of sampling if segregating)
6. ancestral trinucleotide (middle nucleotide mutated, NOT CODON)
7. derived nucleotide

8. 0 or 1 for synonymous or nonsynonymous (respectively; 0 for non-coding)
9. ancestral amino acid (single character representation; '`X`' for non-coding)
10. derived amino acid (single character; '`X`' for non-coding)
11. fitness effect (this is s, NOT $\gamma = PNs$)
12. number of chromosomes (n) that carry the mutation

13-... comma delimited list of the n chromosomes carrying mutation

Each mutation event is terminated with a semi-colon. The list of chromosomes carrying each mutation is reported as a decimal: $p.c$, where p is the population number (zero based) and c represents the chromosome number in that population (also zero based). Take the mutation event reported in the first iteration: "`0,A,24,-237,0,TTA,A,1,Y,N,0.0,2,0.1,0.8;`". This indicates that it occurred at the first locus (zero), which is autosomal, at position 24. This mutation arose 237 generations before sampling (i.e. 237 generations before the burn-in period ended), and was segregating at time of sampling (sampled at time 0). This mutation was a nonsynonymous mutation having no fitness effect, and was carried by two chromosomes (1 and 8) in population zero (the only population simulated).

Mutations that fix in the population during the burn-in period are not recorded. If you want to track fixations while simulating a single population, use the `-TE` option. Mutations that are fixed in the sample from population p will be reported as p.-1. It is possible to distinguish mutations that are fixed in the sample but segregating in the population using the 4th entry (the generation it supposedly fixed). If it "fixed" at the end of the simulation, then the fixation time will be the same as segregating polymorphisms. This can only happen if it was still segregating in the population (as random mating would not yet have occurred).

It is generally encouraged to retain the ancestral sequence (just in case you want to go back and re-analyze some previous simulations and need the actual sequences; e.g. to setup the observed McDonald-Kreitman tables). However, they can take up a lot of space in the output, so you can exclude ancestral sequences using the following option.

removing ancestral sequence from output

```
--noSeq (-A)
```

By default, output will be printed to the screen. If you would rather it be written to a file, you can use the following option.

sending output to a file

```
--outfile (-o) [a] <file>
```

The optional character '`a`' would allow you to append to a file rather than overwriting it. When multiple iterations are run, and output is being directed to a file, the progress of the simulation will be printed to the error file. By default, the error file is the screen, but this can be changed using the following option

sending error messages to a file

```
--errfile (-e) [a] <file>
```

This is also useful for keeping track of any error messages that arise (such as when figuring out what might have gone wrong with a set of command line arguments).

9 Using convertSFS_CODE to Generate Useful Data

The output produced by the program `SFS_CODE` is fairly concise, but is not the easiest file to parse. I have therefore provided the additional program `convertSFS_CODE`, which takes the output from `SFS_CODE`, and converts it to a format that might be easier for you to use. These include various summary statistics, as well as the format required for the program STRUCTURE [e.g. Falush et al. (2003)], and an output analogous to format used by the coalescent simulation program `ms` (Hudson, 2002), among others.The basic usage of `convertSFS_CODE` is as follows:

```
./convertSFS_CODE <input_file> <option [args]>
```

The `<input_file>` to `convertSFS_CODE` is the output file from `SFS_CODE`. The options available are outlined in Table 3. As an example, consider generating a human-neandertal-chimpanzee alignment using a single chromosome from each. You might consider the following slightly modified version of Example 18.

Ex. 19.
```
./sfs_code 3 1 -L 1 66 -TS 0 0 1 -TS 8 1 2 -TE 9 2 \
  -TE 10 -n 1 -o out.txt
```

This would generate a single simulation of 3 individuals for a locus of length 66 basepairs. You could then use `convertSFS_CODE` to generate a **fasta-style alignment** of one chromosome using the following example.

Ex. 20. `./convertSFS_CODE out.txt -a I 1 0`

This would print the alignment of three chromosomes to the screen. It might look like the following:

```
>it0pop0ind0locus0
GTATGTATAGGGCTTGGTATTGAAAATAGGTCCCAGGAAATCTTGACCGGCACCCAAGAGGTCATG
>it0pop1ind0locus0
GTATGTATAGGGCTTTATATTGAAAATAGGTCCCAGGAAATCTTGACCGGCACCCAAGAGGTCATG
>it0pop2ind0locus0
GTATGTATAGGGCTTTATATTGAAAATAGGTCCCAGGAAATCTTGACCGGCACCAAAGAGGTCATG
```

The name of each sequence indicates the iteration ("`it0`"), the population ("`pop0`"), the individual chromosome ("`ind0`"), and the locus ("`locus0`"), followed by the actual sequence on the next line. If you just wanted to extract a single sequence from human and chimpanzee, then you would use the "`P.I`" option, which would allow you to select certain chromosomes from certain populations. In this case, you would use "`P.I 2 0.0 1.0`" to indicate that you want two chromosomes to be printed: chromosome 0 from population 0 and chromosome 0 from population 1.

Ex. 21. `./convertSFS_CODE out.txt -a P.I 2 0.0 1.0`

If you also want to include the true ancestral sequence of each population to be printed, include the character '`A`'. Note that this is the ancestral sequence of a population, and not the ancestral sequence of a pair or group of populations. It will be printed just as any other sequence from a population, but will have "`indA`", such that the chromosome identifier is '`A`'.

To generate the **McDonald-Kreitman table** for the human-chimp simulation, you would use the following example.

Ex. 22. `./convertSFS_CODE out.txt --MK 1 0`

This indicates that you want to use population 1 for polymorphism (the human population) and population 0 as an outgroup to call fixed differences (chimp polymorphism is not included). The output for this particular case is as follows: `0 0 0 2 1.0000`. Not very interesting. There were (in order of appearance) zero synonymous and nonsynonymous polymorphisms, zero synonymous fixed differences, and two nonsynonymous fixed differences. The last value given is the Fisher exact test p-value for the table. You can print the table to a file using the flag "`F <filename>`". If you ran several iterations, you can print each one independently using "`ITS -1`". More generally, if you just want to print the first n iterations, replace "-1" with "n", the -1 just allows you to not worry about the actual number of simulations that were done. By default, the outgroup sample size is 1 (e.g. using just the reference sequence to call fixed differences), but this can be changed to n using the option "`OGSS` n". This is helpful if you want to test the effect of using multiple outgroup sequences on calling fixed differences (many sites that are apparently fixed between populations are actually the result of sampling a common mutation segregating in one of the populations). Finally, if you want to use parsimony to call synonymous and nonsynonymous mutations (instead of using their true classification stored during the mutation), use "`OBS`". As an example, suppose that the file `out.txt` contained several simulations from two populations across 3 loci, but we wanted the observed (i.e. using parsimony) MK table for the first locus using a sample size of 2 for the outgroup. We could use the following example:

Ex. 23.
```
./convertSFS_CODE out.txt --MK 1 0 F mk.txt ITS -1 \
L 1 0 OGSS 2 OBS
```

For printing the **site-frequency spectrum** (**SFS**), use `--SFS`. There are many similar options for this output. However, you can also specify the SFS of just autosomal (`A`), X or Y linked SNPs (`X` or `Y`, respectively). You can also specify the type of mutations to include in the SFS (e.g. synonymous, nonsynonymous, or both). In this case, you would use "`T 0`", "`T 1`", or "`T 2`" (respectively). Non-coding mutations are considered synonymous, and will not be reported for "`T 2`". Sometimes using an outgroup to identify the ancestral state of a polymorphism under parsimony will be wrong (Hernandez et al., 2007b). By default, the true SFS will be generated, but you can also use parsimony (if you've simulated at least two populations) by including "`OBS <ing> <og> [og`$_{\texttt{size}}$`]`", where `ing` is the in-group used for polymorphism, `og` is the outgroup used for identifying the ancestral states of polymorphisms, and the optional argument `og`$_{\texttt{size}}$ indicates the number of sequences to use from the outgroup (default is 1).

To produce output in a similar format as the coalescent simulator **ms** (Hudson, 2002), you can use the option `--ms`. You can print the output to a file using `F <file>`, and specify the type of mutations to output (either synonymous/non-coding, `T 0`; nonsynonymous, `T 1`; or both `T 2`).

Finally, you can produce the input file for the Bayesian clustering algorithm **STRUCTURE** (Falush et al., 2003) using the option `--structure`. For this option, you can specify the centimorgans per megabase (using `CMMB <c>`), and restrict the output to either specific individuals (`I <n> ...`), specific populations (`P <n> ...`) or specific individuals from specific populations (`P.I. <n> ...`).

It is important to note that you can use several options at once. For example, if you want to generate both MK tables and the SFS, you can put them both in the command line. Additionally, if you simulated several species, and want to generate the observed SFS using species with increasing divergence, then you can just concatenate all your `--SFS` commands as follows:

Ex. 24.
```
./convertSFS_CODE out.txt --SFS OBS 0 1 F sfs1.txt \
  --SFS OBS 0 2 F sfs2.txt --SFS OBS 0 3 F sfs3.txt
```

Table 3: `convertSFS_CODE` Options. Any combination of `<arguments>` can be used (in any order) for a given task.

long	short	`<arguments>`	description
`--help`	`-h`	∅	**Display help menu**
`--alignment`	`-a`		**Print sequence alignment in fasta format**
		`[A]`	Print ancestral population sequence
		`[F <file>]`	Print sequences to a `file`
		`[L <n>` $<L_1>\ldots<L_n>$`]`	Only print `n` loci.
		`[I <n>` $<I_1>\ldots<I_n>$`]`	Only print `n` chromosomes (but from all populations).
		`[P <n>` $<P_1>\ldots<P_n>$`]`	Only print specific populations.
		`[P.I <n>` $<p_1.c_1>\ldots$`]`	Only print specific chromosomes from specific populations
		`[ITS <n>]`	Only print first *n* iterations
`--ms`	`-m`		**Produce `ms`-style output**
		`[F <file>]`	Print MK tables to a `file`
		`[T <type>]`	Extract mutations of `<type>` = 0, 1, or 2 (synon., nonsynon., or both)
`--MK`	`-M`		**Print McDonald-Kreitman tables**
		`<ing> <og>`	Ingroup and outgroup (**required**)
		`[F <file>]`	Print MK tables to a `file`
		`[ITS [n]]`	Print each iteration [or just first *n*]
		`[L <n>` $<L_1>\ldots<L_n>$`]`	Only print `n` loci.
		`[OGSS <size>]`	Set the outgroup sample size
		`[N <n>]`	Randomly sample *n* chromosomes from each population
		`[OBS]`	Use parsimony to call synonymous and nonsynonymous mutations
`--structure`	`-s`		**Print structure input**
		`[F <file>]`	Print to a `file`
		`[CMMB <c>]`	set centiMorgans/Megabase to *c*
		`[I <n>` $<I_1>\ldots<I_n>$`]`	Only print `n` chromosomes (but from all populations).
		`[P <n>` $<P_1>\ldots<P_n>$`]`	Only output specific populations.
		`[P.I <n>` $<p_1.c_1>\ldots$`]`	Only print specific chromosomes from specific populations

Continued on next page...

Table 3 – continued from previous page

long	short	`<arguments>`	description
--SFS	-S	**Print site-frequency spectra (SFS)**	
		`[A]`	Extract only autosomal loci
		`[F <file>]`	Print SFS to a `file`
		`[ITS [n]]`	Print each iteration [or just first n]
		`[L <n> <L`$_1$`>...<L`$_n$`>]`	Only print `n` loci.
		`[N <n>]`	Randomly sample n chromosomes from each population
		`[OBS <ing> <og> [og`$_{\texttt{size}}$`]]`	Use parsimony to identify ancestral alleles from an outgroup [using `og`$_{\texttt{size}}$ chromosomes]
		`[I <n> <I`$_1$`>...<I`$_n$`>]`	Only print `n` chromosomes (but from all populations).
		`[P <n> <P`$_1$`>...<P`$_\text{n}$`>]`	Only output specific populations.
		`[T <type>]`	Extract mutations of `<type>` = 0, 1, or 2 (synon., nonsynon., or both)
		`[X]`	Extract only X-linked mutations
		`[Y]`	Extract only Y-linked mutations

10 Default Parameter Values

Running `SFS_CODE` with the default parameters will generate a sample of six diploid individuals (12 chromosomes) under the standard neutral model assumptions of a constant population size, absence of selection, etc. Every parameter discussed below can be changed using the command line options described in Table 5. The full list of default parameter values are given in Table 4.

11 Summary of Options and Arguments

In `SFS_CODE`, an **option** is a feature that allows you to change the characteristics of the simulation. Every option implemented in `SFS_CODE` is summarized in table 5. There are basically five types of options: (1) those that control the output, (2) those those that affect all populations or set foundation for the simulation ("Global Options"), (3) those that may be population specific ("Population Options"), (4) those that describe the effect of natural selection, and (5) those that govern the demographic events ("Evolutionary Events"). All options (except evolutionary events) have both a **long name** (preceded by two dashes) and a **short name** (a single character preceded by a single dash), which can be used interchangeably (e.g. you could either use "`--theta 0.01`" or "`-t 0.01`" to set the population scaled mutation rate to 0.01 for all populations). Most options have **arguments**. Arguments that are **required** are enclosed in `<angled brackets>`. Arguments that are **optional** are enclosed in `[square brackets]`.

Some options apply to all populations (e.g. the length of the simulated sequence `-L`), and some apply only to a specific population (such as the generation effect `-G`). Others default to all populations, but allow you to specify a specific population. These are denoted by `[P <pop>]` in the argument list. This means that you can add the character '`P`' followed by the number of a specific population to apply the option to that population exclusively. For example, if you are simulating two populations, one of which is twice the size of the other, you might add "`-N P 1 1000`" to your command line.

Table 4: Default parameter values used in SFS_CODE.

Parameter	**value**
Effective population size	500
Ploidy	2
Allo- (0) or Auto-ploidy (1)	0
Relative size of female population	0.5
Sample size for each population	6
Migration between populations	0.0
Probability male migrates (if migration)	0.5
θ/site	0.001
Substitution model	0
Number of mutation rate classes (sites)	1
Number of mutation rate classes (loci)	1
Infinite sites?	0
ρ/site	0.0
Number of loci	1
Length of locus (bp)	5001
Linkage within loci	0.0
Annotation	`C`
Sex chromosome?	no
Self-fertilization rate	0.0
Selection distribution	0
Proportion of loci subject to selection	1.0
Non-lethal mutation rate	1.0
Initial burn-in period (×PN)	5
Burn-in period of subsequent iterations	2
Print ancestral sequence?	yes

Generally speaking, options can be used in any order. The exceptions are `--linkage`, `--annotate`, and `--sex`, which must come after `-L` if `-L` is used (if `-L` is not used, then order really doesn't matter). However, arguments for each option are in a specified order, and must always be used in the proper order.

Table 5: SFS_CODE Options

	long	**short**[a]	`<arguments>`	**description**
Output	`--help`	`-h`	$\emptyset$	Display help menu. (p3)
	`--noSeq`	`-A`	$\emptyset$	DO NOT print ancestral sequences. (p24)
	`--outfile`	`-o`	`[a] <file>`	Write [or `<a>`ppend] output to a `<file>` instead of the screen. (p24)
	`--errfile`	`-e`	`[a] <file>`	Write [or `<a>`ppend] error messages to a `<file>`. (p24)
Selection	`--selDistType`	`-W`*	`[P/L] <type> [arg]`	Distribution of selective effects. (p10, Table 1)
	`--neutPop`	`-w`*	`<pop>`	No selection on population `<pop>`. (p11)
	`--constraint`	`-c`*	`[P/L] <f`$_0$`>`	Set the proportion of non-lethal mutations to `<f`$_0$`>`. (p12)

Continued on next page...

Table 5 – continued from previous page

	long	short[a]	<arguments>	description
Global Options	--BURN	-B	<burn>	Burn-in time for initial population. (p4)
	--BURN2	-b	<burn>	Burn-in time for subsequent simulations. (p4)
	--length	-L	<R> <L_1> [<L_2>..] [R]	Simulate <R> loci (regions) with lengths <L_1> [<L_2>... opt.; add 'R' after a subset of lengths to repeat]. (p5)
	--linkage	-l	<p/g> <d_1> [<d_2>..] [R]	Set linkage between adjacent loci (either <p>hysical, or <g>genetic distance) to <d_1> [<d_2> ..<d_{R-1}> opt.; add 'R' to repeat pattern]. MUST USE AFTER -L. (p5)
	--annotate	-a	<a_1> [<a_2>..<a_R>] [R]	Annotate each locus as C (coding) or N (non-coding) [<a_2>..<a_R> opt; add 'R' to repeat pattern]. Note: If coding (default), length will be rounded to nearest codon. (p6)
	--sex	-x	<x_1> [<x_2>...] [R]	Annotate sex loci, either 0 or 1 (autosomal or sex, resp.). Use only <x_1>, or specify each locus, or put 'R' at end to repeat a subset. Only implemented for the diploid case (P=2). Males do not recombine at sex loci. (p6)
	--ploidy	-P	<ploidy>	Set the ploidy of all populations to <ploidy> (1,2,4 only; i.e. haploid, diploid, or tetraploid). (p7)
	--tetraType	-p	<0/1>	If P=4 (tetraploid) assume auto- or allotetraploid (0 or 1, resp.). (p7)
	--substMod	-M	<mod> [args]	Set the mutation model. (p15, Table 2.)
	--INF_SITES	-I	$\emptyset$	Avoid multiple mutations segregating at the same site concurrently. Multiple hits can still occur for long divergence times. (p15)
	--seed	-s	<int>	Set random number seed to <int>. This should always be set manually if using this program on a cluster!! (p21)
Population Options	--theta	-t*	[P <p>] <θ>	Set the PER SITE population scaled mutation rate to <θ> for ALL populations [or just population <p>]. (p4)
	--rho	-r*	[P <p>] <ρ>	Set the PER SITE population scaled recombination rate to <ρ> for ALL populations [or just population <p>]. (p4)
	--popSize	-N*	[P <p>] <size>	Set all population to size <size> 'P'-ploid individuals [or just population <p>]. (p6)
	--sampSize	-n	<SS_1>..<SS_{NPOP}>	Sample <SS_1> individuals from population 1, ..., <SS_{NPOP}> individuals from population NPOP. Use the value -1 to sample an entire population. (p9)
	--migMat	-m*	A <M>	Set the migration rate to/from all pops to <M>/(NPOP-1). (p14)
			P <P_{to}> <P_{from}> <M>	Set the migration rate entry $m_{\text{to,from}}$ = <M>
			L <$M_{0,1}$>...<$M_{NPOP,NPOP-1}$>	Set all entries of the migration matrix.
	--pMaleMig	-y*	[P <p>] <p_{male}>	Set the proportion of migrants out of each population [or just population <p>] that are male to <p_{male}>. (p14)
	--propFemale	-f	[P <p>] <pf>	Set the proportion of females in each population [or just <pop>] to <pf>, default 0.5. (p14)
	--self	-i*	[P <p>] <s>	Set the selfing rate <s> for ALL populations [or just population <p>]. (p16)

Continued on next page...

Table 5 – continued from previous page

	long	short[a]	`<arguments>`	description
	`--KAPPA`	`-K`*	`[P <p>] <`κ`>`	Set transition/ transversion rate ratio `<`κ`>` (only valid for `--substMod` 2 or 3). (p15)
	`--PSI`	`-C`*	`[P <p>] <`ψ`>`	Set CpG bias parameter (non-CpG rejection rate; only valid for `--substMod` 1 or 3). (p15)
	`--rateClassSites`	`-V`	`[P <p>] <`$\mathtt{n_{classes}}$`> <`α`>`	Mutation rate variation among sites in loci (discrete Gamma model of `<`$\mathtt{n_{classes}}$`>` classes with rate `<`α`>` and mean 1). (p16)
	`--rateClassLoci`	`-v`	`[P <p>] <`$\mathtt{n_{classes}}$`> <`α`>`	Mutation rate variation among loci (discrete Gamma model of `<`$\mathtt{n_{classes}}$`>` classes with rate `<`α`>` and mean 1). (p16)
	`--GenEffect`	`-G`*	`<pop> <G>`	Generation time effect. `<G>` > 1 makes `<pop>` experience `<G>` rounds of mating each generation while if `<G>` < -1 mating occurs only every `|<G>|` generations. `<G>` must be an integer, with `<G>` > 0 or `<G>` < -1. (p17)
Evolutionary Events		`-Td`	`<`τ`> [P <p>] <`ν`>`	Discrete population size change at time `<`τ`>` with magnitude `<`ν`>` $= \mathtt{N_{new}/N_{old}}$, where $\mathtt{N_{old}}$ is the size of `<pop>` prior to the event, NOT ANCESTRAL! (p6)
		`-Tg`	`<`τ`> [P <p>] <`α`>`	Set the exponential growth rate of a population to `<`α`>` at time `<`τ`>`. (p7)
		`-Tk`	`<`τ`> [P <p>] <K> <r>`	Logistic growth at rate `<r>` beginning at time `<`τ`>` until final population size `<K>` is reached. (p7)
		`-TE`	`<`τ`> [p]`	Terminate simulation [or just a population] at time `<`τ`>` $\times PN0$. (p8)
		`-TS`	`<`τ`> <i> <j>`	Split population `<i>` at time `<`τ`>`$\times PN$ generations to found population `<j>`. (p13)
		`-TD`	`<`τ`> <i> <j> <f> <N> [l]`	Domesticate population `<j>` with `<N>` individuals from `<i>` at time `<`τ`>` using a derived allele at frequency `<f>`$\pm 5\%$. (p13)

[a] Asterisk in short name indicates that the parameters can be changed (or option initiated) at any time using `-T<short_name> <`τ`> <args>`. See Section 4.6 on page 17.

References

Boyko, A. R., Williamson, S. H., Indap, A. I., Degenhardt, J. D., Hernandez, R. D., Lohmueller, K. E., Adams, M. D., Schmidt, S., Sninsky, J. J., Sunyaev, S. R., White, T. J., Nielsen, R., Clark, A. G., and Bustamante, C. D. (2007). Quantifying the distribution of selective effects among newly arising amino acid mutations in the human genome. *Submitted.*

Caicedo, A. L., Williamson, S. H., Hernandez, R. D., Boyko, A., Fledel-Alon, A., York, T. L., Polato, N. R., Olsen, K. M., Nielsen, R., McCouch, S. R., Bustamante, C. D., and Purugganan, M. D. (2007). Genome-wide patterns of nucleotide polymorphism in domesticated rice. *PLoS Genetics*, 3(9):1745–1756.

Cutler, D. J. (2000). Understanding the overdispersed molecular clock. *Genetics*, 154(3):1403–1417.

Falush, D., Stephens, M., and Pritchard, J. K. (2003). Inference of population structure using multilocus genotype data: linked loci and correlated allele frequencies. *Genetics*, 164(4):1567–1587.

Hernandez, R. D., Hubisz, M. J., Wheeler, D. A., Smith, D. G., Ferguson, B., Rogers, J., Nazareth, L., Indap, A., Bourquin, T., McPherson, J., Muzny, D., R., Nielsen, R., and Bustamante, C. D. (2007a). Demographic histories and patterns of linkage disequilibrium in Chinese and Indian rhesus macaques. *Science*, 316(5822):240–243.

Hernandez, R. D., Williamson, S. H., and Bustamante, C. D. (2007b). Context dependence, ancestral misidentification, and spurious signatures of natural selection. *Mol Biol Evol*, 24(8):1792–1800.

Hudson, R. R. (2002). Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics*, 18(2):337–338.

Hwang, D. G. and Green, P. (2004). Bayesian Markov chain Monte Carlo sequence analysis reveals varying neutral substitution patterns in mammalian evolution. *Proc Natl Acad Sci U S A*, 101(39):13994–14001.

Jukes, T. H. and Cantor, C. R. (1969). Evolution of protein molecules. In Munro, H. N., editor, *Mammalian protein metabolism*, pages 21–132. Academic Press, New York.

Kimura, M. (1980). A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences. *J Mol Evol*, 16(2):111–120.

Zhang, Z. and Gerstein, M. (2003). Patterns of nucleotide substitution, insertion and deletion in the human genome inferred from pseudogenes. *Nucleic Acids Res*, 31(18):5338–5348.