

# MathBio Yggdrasil Simulation and Analysis

Spencer Tipping

November 28, 2009

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Using the Model</b>	<b>1</b>
2.1	Profiles . . . . .	1
2.1.1	Nomenclatural conventions . . . . .	2
2.1.2	Notation . . . . .	2
2.1.3	Output directives . . . . .	2
2.1.4	Process directives . . . . .	4
2.1.5	General-purpose biological directives . . . . .	5
2.1.6	Model-specific parameters . . . . .	6
2.1.7	Internal directives . . . . .	7
2.1.8	Example profile . . . . .	7
2.2	Rebuilding . . . . .	8
2.2.1	Using <i>make</i> . . . . .	8
2.2.2	Using <i>git</i> . . . . .	9
2.2.3	Todo . . . . .	9
2.3	Required Libraries . . . . .	10
2.4	Conditional Inlining . . . . .	10
2.5	<i>Malloc</i> Wrappers . . . . .	10
2.6	Hierarchical Structures . . . . .	11
2.7	Random Numbers . . . . .	12
<b>3</b>	<b>Model Framework</b>	<b>13</b>
3.1	Codons and Amino Acids . . . . .	13
3.2	Codon Functions . . . . .	14
3.3	Structure Definitions . . . . .	15
3.4	Codon Sets . . . . .	15
3.5	Sequences . . . . .	19
3.5.1	Sequence serialization forms . . . . .	20
3.6	Populations . . . . .	22
3.6.1	Selective fitness . . . . .	22

<b>4</b>	<b>Simulation Parameters</b>	<b>23</b>
4.1	Statistical Structure	24
4.1.1	Variance coefficients	24
4.1.2	Final results	24
4.1.3	Discrete pi data	24
4.1.4	Continuous pi data	24
4.1.5	Parsimonious genetic likelihoods	24
4.1.6	Discrete Nei and Gojobori ratios	25
4.1.7	Theta	25
4.1.8	Traced values	25
4.1.9	Random estimator error	25
4.1.10	Generic initialization methods	25
4.2	Selection and Drift Constants	26
4.3	Simulation Parameter Structures	26
4.3.1	Output parameters	26
4.3.2	Process parameters	27
4.3.3	General-purpose biological parameters	27
4.3.4	Model-specific parameters	27
4.3.5	Internal parameters	28
4.3.6	Simulation parameters	28
<b>5</b>	<b>Statistics</b>	<b>30</b>
5.1	Variance Coefficients	30
5.2	Methods for Sitewise Computation	31
5.3	Parsimonious Genetic Likelihood Ratios	32
5.4	Nei and Gojobori Ratios	32
5.5	Theta Statistic	33
5.5.1	Estimator combination methods	33
5.5.2	Codon ambiguity methods	34
5.6	Discrete Pathways	36
5.7	Pi Statistic	37
5.8	Tajima's D	38
5.9	Full Tracing	38
5.10	Statistical Verification	41
<b>6</b>	<b>Simulation Runtime</b>	<b>42</b>
6.1	Mutation	42
6.2	Model Functions	43
6.2.1	Neutral model	43
6.2.2	Purifying selection	43
6.2.3	Diversifying selection	43
6.2.4	Sharp population bottlenecks	45
6.2.5	Even population subdivision	45
6.3	Parent Selection and Recombination	46
6.4	Runtime Functions	49
6.4.1	Printing	49

6.4.2	Execution . . . . .	50
6.5	Front-end Code . . . . .	52
6.5.1	Profile parser . . . . .	52
6.5.2	Profile sanity check . . . . .	54
6.5.3	Main method delegation . . . . .	56
7	Closing	56

## 1 Introduction

The purpose of this project is to generate data used in determining the correlation between values of Tajima’s  $D$  statistic as computed for a population and the evolutionary and demographic history of that population. Computation proceeds in two major steps: First, the population’s evolution is simulated by applying the forces that would apply in nature. Then, once sufficiently many generations have been simulated, we look at the last generation, take a randomly chosen subset for statistical sampling, and calculate values of  $D_{syn}$  and  $D_{non}$ . This process comprises a single trial.

After many trials, each with a different random seed, we compile distributions of the resulting  $D$  statistics. These distributions allow Bayesian categorization of future  $D$  values.

---

```
#ifndef __MODEL_H
#define __MODEL_H
```

---

## 2 Using the Model

This file contains the core workings of the model, but it does not provide a runtime environment. The runtime is provided by `model-runtime.c`, which includes the header file produced by this TeX document. Using the runtime environment requires a *profile*, which is a set of configuration information for a single simulation run.

### 2.1 Profiles

A profile is a plain text file with a standard format. Each line consists of a series of directives, comment words, and/or values. A directive has the format `directive[: value]`, where `value` may be a number, plain text (not inside quotation marks), or otherwise as dictated by the directive. Comments begin with the directive `begin-comment` and end with the directive `end-comment`. Sections 2.1.3 and following contain a list of the currently supported directives and their functions.

### 2.1.1 Nomenclatural conventions

Profiles generally take on names such as `p01`, `g15`, etc. Currently, I use this convention:

- Standard complete-run profiles begin with `p`.
- Simulation-only profiles (generators) begin with `g`.
- Statistics-only profiles (results) begin with `r`.

The output files are named according to the profile that generated them. The first batch of output from the profile `p12`, for example, is named `p12-001.csv`. Since the profiles serve as the only substantial documentation for the settings used to generate an output file, *a profile is never modified after it is used to generate data*. A new profile is created instead. That way, finding out the configuration that produced a data set is as simple as reading the profile that generated it.

Statistics-only profiles follow the same nomenclature as the output files, except that they begin with `r` instead of `p` or `g`. The same rule applies for the generator profile: It cannot be modified once the output has been produced.

### 2.1.2 Notation

Generally, the options below follow a naming convention that allows the user to infer the type of each option. When any deviations from this convention occur, they are specifically noted. Consider this option definition:

generations: *n*

*n* indicates that the value expected by the *generations* directive should be a positive integer. *x* would indicate a real number, *α*, *β*, or *γ* indicates that the value is a member of a later-defined set, and *s* indicates that the value is a genetic sequence. For an example of how these values are specified, see section 2.1.8.

### 2.1.3 Output directives

These directives determine the format of the output. They do not impact the biological settings of the model in any way. See section 4.3.1 for the storage of these parameters.

trials: *n* Sets the number of trials to run. By default, this is set to 50. Time complexity is  $O(n)$  in the number of trials, and space complexity is  $O(1)$ .

print-sequences Specifying this directive will cause the simulation to print all of the sequences in the statistical sample as well as the resulting statistical computations. This is primarily useful for verifying the statistical code. If this directive is absent, sequences will not be printed.

**make-profile** This directive is available only if `simulation-only` has been specified. Specifying `make-profile` causes the output of the model to be a valid profile instead of just a bunch of sequences. This results in the following being legal:

```
$ ./model-runtime > output_profile <<EOF
simulation-only
make-profile
EOF
[output elided]
$ ./model-runtime < output_profile > output.csv
$
```

Note that if the run is interrupted and a trial is not finished, the output may not be usable without modification because it will not contain as many trials as are indicated by the header. Profiles generated with this option should look like this:

```
begin-comment
[Some descriptive text]
end-comment

trials: 100
population-size: 500
statistics-only
end-of-options

!2naAS+a/GAuga12
!7hSGJas2jtswdug
[more sequences]
```

The blocks of sequences have blank lines to separate the trials for readability, although the amount of whitespace is not relevant to the sequence reader.

**print-each-generation** Specifying this directive will cause the simulation to print statistical measurements at each generation, not just at the end. If this directive is absent, only the final statistics will be printed.

**full-tracing** Specifying this directive will enable full tracing of the evolutionary history as it progresses. This is useful for statistical verification of the estimators used to calculate theta. Enabling this option will make the space complexity of the simulation increase to  $O(n)$  in the number of generations instead of  $O(1)$ . For particulars of what this entails, see section 5.9.

#### 2.1.4 Process directives

These directives alter the nature of the processing done by the program. For example, sometimes statistical computations should be omitted, or they should be performed on sequences that are loaded from a file instead of produced internally. By default, the simulation will be run and statistical analysis will be performed on the sequences from the simulation, however either one of these steps may be deferred.

`simulation-only` Specifying this directive will cause only the simulation portion of the code to be run, and all resulting sequences are output in packed form to the standard output. (See section 3.5.1 for a description of packed formatting.) If `make-profile` is specified, then a valid profile is generated such that running the program on the resulting output produces the statistical measurements for those packed sequences.

`statistics-only` Specifying this directive disables the simulation code and requires that sequences are present in the profile. The purpose of specifying this would be to load sequences from a file instead of generating them. Obviously, this directive is incompatible with `simulation-only`.

`end-of-options` Specifying this directive stops profile parsing. The reason you would want to use this is to provide sequences for statistical analysis after the options. In this scenario, your profile might look like this:

```
trials: 1
population-size: 10
sample-size: 4
statistics-only
end-of-options

!A8bh+/a8sc
!a8sqhAH/as
[...]
```

No comments may appear after the `end-of-options` directive, as they will be treated as sequence data in the event that sequences are to be read.

#### 2.1.5 General-purpose biological directives

These directives alter the fundamental behavior of the model in a significant way, and are commonly used to select different types of simulation. Generally speaking, nothing biologically unrealistic, beyond the known unrealism of our global assumptions, will happen for reasonable values of these parameters.

- population-size:  $n$  Sets the number of individuals per generation. By default, this is set to 500. Time complexity is  $O(n)$  in the number of individuals for the neutral model,  $O(n \log n)$  for purifying selection,  $O(n^2)$  if drift is used, and  $O(n^2)$  if diversifying selection is used. Space complexity is  $O(n)$  in the population size.
- generations:  $n$  Sets the number of generations to simulate. By default, this is set to 1000. Time complexity is  $O(n)$  in the number of generations, and space complexity is  $O(1)$  if full tracing is not used, and  $O(n)$  if full tracing is used.
- sample-size:  $n$  Sets the sample size for statistical calculations. By default, this is set to 20. Time complexity is  $O(n^2)$  if full tracing is not used and  $O(p(n))$  if full tracing is used, and space complexity is  $O(1)$ . In practice, statistical calculations take so little time relative to the simulation that for most purposes the effects are negligible.
- mutations-per-nucleotide:  $x$  Sets the mutation rate on a per-nucleotide basis. This occurs per reproductive cycle (generation) per individual. By default, this is set to  $3.4 \times 10^{-5}$ .
- transitions-per-transversion:  $x$  Sets the ratio of transitions to transversions for the HKY-85 model. By default, this is set to 1.3.
- frequency- $\alpha$ :  $x$  Sets the relative frequency of the  $\alpha$  nucleotide for the HKY-85 model, where  $\alpha$  is one of  $\{a, c, g, u\}$ . Must be between 0 and 1. By default, all nucleotides have the same frequency; thus, this is set to 0.25.
- recombination-rate:  $x$  Sets the recombination rate per individual per generation. By default, this is set to  $2.4 \times 10^{-5}$ .
- ancestral-sequence:  $s$  Sets the ancestral sequence  $s$  used to create the initial population. By default, this is set to the sequence UGU ACA AGA AAC AAC AAU ACA AUA AAA AGU AUA CAU AUG GGA CUA GGG AGG ACA UUU UAU ACA ACA GGA GAA GUA AUA GGA GAU AUA AGA CAA GCA CAU UGU. See section 3.5.1 for a description of valid sequence formatting.
- selection-model:  $\beta$  Sets the selection model to  $\beta$ , where  $\beta$  is one of *neutral*, *purifying*, or *diversifying*. By default, this is set to *neutral*.
- drift-model:  $\gamma$  Sets the drift model to  $\gamma$ , where  $\gamma$  is one of *neutral*, *sharp-bottleneck*, or *even-subdivision*. By default, this is set to *neutral*.

### 2.1.6 Model-specific parameters

These parameters are relevant only if selection or drift are being used. They determine the extent, duration, and other aspects of models. Below is a list of parameters for each model:

**Purifying selection** The parameters below are available to configure purifying selection.

- purifying-selection-coefficient:  $x$  Sets the selection coefficient  $s$  used for purifying selection. By default, this is set to 0.1. This parameter must be in the set  $(0, 1)$ .
- purifying-optimal-sequence:  $s$  Sets the sequence used by purifying selection to represent the optimal individual. Each individual's fitness is determined by scoring against this sequence. By default, this is set to the default ancestral sequence. This sequence and the ancestral sequence must be of equal length. See section 3.5.1 for a description of valid sequence formatting.

**Diversifying selection** The parameters below are available to configure diversifying selection.

- diversifying-selection-coefficient:  $x$  Sets the selection coefficient  $s$  used for diversifying selection. By default, this is set to 0.1. This parameter must be in the set  $(0, 1)$ .

**Sharp bottlenecks** The parameters below are available to configure sharp bottlenecks.

- sharp-bottleneck-start:  $n$  Sets the generation at which the bottleneck begins. By default, this is set to 200. It is an error to begin a bottleneck after the end of a simulation run.
- sharp-bottleneck-duration:  $n$  Sets the number of generations during which the bottleneck is in effect. By default, this is set to 500. A warning is produced if the bottleneck extends past the end of the simulation.
- sharp-bottleneck-size:  $n$  Sets the population size while the bottleneck is in effect. By default, this option is set to 50.

**Even subdivisions** The parameters below are available to configure even subdivisions.

- even-subdivision-start:  $n$  Sets the generation at which the subdivision begins. By default, this is set to 200. It is an error to begin a subdivision after the end of a simulation run.
- even-subdivision-duration:  $n$  Sets the number of generations during which the subdivision is in effect. By default, this is set to 500. A warning is produced if the subdivision extends beyond the end of the simulation.
- even-subdivision-size:  $n$  Sets the number of individuals in each subdivision. By default, this is set to 50. If this number does not evenly divide the population size, individual subdivisions will be rounded down and the rest of the population will occupy a remainder subdivision.

### 2.1.7 Internal directives

These directives alter the functionality of the model's internal subroutines and require implementation-specific knowledge to use effectively. The default values of these parameters are acceptable for all normal use, and specifying values

for these parameters may cause the model to deviate from biological realism. Specifying any internal parameter will produce a warning indicating that the simulation may not be biologically realistic.

`assumed-gl-variance: x` If  $x \neq 0$ , this option enables variance optimization in the theta calculation. This entails taking a linear combination of genetic likelihoods and the Nei and Gojobori ratios to minimize the statistical variance of the non-synonymity estimate (see section 5.5.1). One caveat of enabling variance optimization is that it may not improve accuracy, since neither estimator is known to be unbiased. A sensible nonzero value for this setting is  $\frac{1}{5}$ , indicating that the genetic likelihood estimator is assumed to be as accurate as Nei and Gojobori ratios for a position with 5 singletons.

`random-seed: n` Seeds the random number generator with  $n$  before any trials are run. This facilitates reusability across runs. By default, the random number generator is seeded with the system's clock value at microsecond resolution, so this should not be necessary unless a previous trial is to be rerun.

### 2.1.8 Example profile

Here is an example profile, created using `cat` and saved into the `profiles` directory:

```
$ cat > profiles/my_profile <<EOF
begin-comment
my_profile: An example profile
```

```
We want to run a lot of trials to build a distribution.
Correspondingly, we reduce the number of individuals to
make it go faster.
end-comment
```

```
trials: 10000
population-size: 50
```

```
begin-comment
Create some settings for other parameters as well:
end-comment
```

```
frequency-a: 0.9
frequency-c: 0.09
frequency-g: 0.009
frequency-u: 0.001
```

```
ancestral-sequence: ACA-UCU-GGG
purifying-optimal-sequence: UUU-GGG-AAA
selection-model: purifying
```

```
selection-coefficient: 0.5
EOF
```

(If you were editing this in a normal text editor, you would just type everything between `begin-comment` and `selection-coefficient: 0.5`, inclusive.)

Here is an invocation of the model using that profile:

```
$ ./model-runtime < profiles/my_profile > output.csv
```

## 2.2 Rebuilding

Should the model need to be rebuilt for any reason, a makefile with appropriate rules is established to facilitate the process. Also, the model is stored in a Git repository, so after appropriate testing changes should be committed. These aspects of rebuilding are described in detail below.

### 2.2.1 Using *make*

GNU Make is a program that manages build dependencies. In this project, for example, `model.h`, which provides all of the code, depends on `model.tex` in that if `model.tex` is altered then `model.h` must be regenerated. For more information about Make, run `info make` or visit <http://www.gnu.org/software/make>. Below is a list of commands relevant to this project:

- `make all` Rebuilds the entire system, including documentation.
- `make bench` Benchmarks an execution of the neutral model under standard settings. This is used to detect model-independent speed improvements.
- `make bin` Rebuilds all binaries but not documentation. This is especially useful to observe any warnings printed by GCC without scrolling past T<sub>E</sub>X output.
- `make check` Produces a file named `check.csv`, which contains sequences and their resulting statistical output. This is used to verify the accuracy of the statistical computation code in section 5.
- `make clean` Deletes all generated files. Used to clean the working directory.
- `make doc` Rebuilds this documentation.
- `make mem` Runs the debugging image of the simulation under `valgrind` (requires that Valgrind is installed), which checks for memory access violations and memory leaks. It is run with the profile in `profiles/mem`, which runs with few trials and a small population size to compensate for the slowdown that results from running a program with Valgrind.

`make prof` Profiles the execution of the debugging build. The debugging build is identical to the main build with the exception that function inlining is not performed. This results in more precise location of performance bottlenecks.

### 2.2.2 Using *git*

Git is a self-contained source code control system. For information about the commands supported by Git, see <http://www.kernel.org/pub/software/scm/git/docs/user-manual.html>.

### 2.2.3 Todo

These are some of the objectives our team has for the summer:

1. Generate distributions for various configurations.
  - (a) Neutral model: 10000 trials.
  - (b) Purifying selection with  $s = 0.01$ : 1000 trials.
  - (c) Purifying selection with  $s = 0.08$ : 1000 trials.
  - (d) Purifying selection with  $s = 0.5$ : 1000 trials.
2. Check finite vs. infinite sites.
  - (a) Lengthen sequence to 10000 bp, but preserve per-individual mutation rate.
3. Perform sensitivity analysis.
  - (a) Vary population size. Is  $500 = \infty$ ?
  - (b) Vary sample size. Check 20, 50, 100.
4. Performance improvements.
  - (a) Make the arbitrary matrix support optional. Special-case for subdivisions and bottlenecks?
5. Compartmentalize statistical calculations to provide verification for existing methods.

## 2.3 Required Libraries

The standard libraries will suffice for the majority of the functionality of this program. The only nonstandard library is bundled with the source code in the file `ca-rng.h`. This file provides a cellular automaton-based random number generator that is about three times as fast as the system's built-in random algorithm.

```
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>

#include "ca-rng.h"
```

## 2.4 Conditional Inlining

When we're debugging, we want to have a lot of granularity in profiling runs. However, the main simulation will perform better if more functions are rolled into one. Therefore, we define a preprocessor directive for conditional inlining that will expand to the `inline` keyword when we're compiling a runtime build and will expand to nothing when we're compiling a debugging build.

Some functions are so small that they are always inlined; these are marked with the standard `inline` keyword.

```
#ifndef DEBUG
#define conditional_inline
#else
#define conditional_inline inline
#endif
```

## 2.5 Malloc Wrappers

Sometimes memory allocation fails, and we want to handle this case uniformly. Currently, we just print an error message and quit. We also define a version of `free` that returns true unconditionally. This is to facilitate the short-circuit logic used in section 6.5.1.

```
inline void *safe_malloc (size_t size) {
    void *result = malloc (size);
    if (result)
        return result;
    else {
        fprintf (stderr, "Error:_%d_bytes_of_memory_could_not_be_allocated.\n", size);
        exit (EXIT_FAILURE);
    }
}

inline unsigned int free_return_true (void *v) {
    free (v);
    return ! 0;
}
```

## 2.6 Hierarchical Structures

We end up using a lot of memory management functions for hierarchical data structures. Rather than write each one from scratch, we use the C preprocessor to generate templates for these structures. A structure uses these definitions:

- Ensure function** Ensures that a structure is of the appropriate size. After `ensure_x (px, l)`, `*px` will have the appropriate number of data elements and its length set.
- Init function** Takes a pointer to a structure and initializes the fields to zero. This should be used only immediately after declaring a stack-allocated structure, since any data already in the structure will be lost but not freed.
- Create function** Creates an empty structure and returns it. To completely delete the structure and its children, use the delete function.
- Delete function** Deletes a structure and all of its dependent structures. Leaves the structure itself allocated, since structures are sometimes allocated on the stack instead of on the heap.

```

#define __hierarchical_structure__(parent_type, child_type) \
typedef struct { \
    unsigned int length; \
    child_type *data; \
} parent_type;

#define __ensure_function_simple__(parent_type, child_type) \
inline void ensure_##parent_type (parent_type *s, unsigned int length) { \
    if (s->length != length) { \
        if (s->data) \
            free (s->data); \
        s->data = (child_type*) safe_malloc (length * sizeof (child_type)); \
        s->length = length; \
    } \
}

#define __ensure_function_complex__(parent_type, child_type) \
inline void ensure_##parent_type (parent_type *s, unsigned int length) { \
    if (s->length != length) { \
        for (unsigned int i = 0; i < s->length; i++) \
            delete_##child_type (s->data + i); \
        free (s->data); \
        s->data = (child_type*) safe_malloc (length * sizeof (child_type)); \
        s->length = length; \
        for (unsigned int i = 0; i < s->length; i++) { \
            s->data[i].data = NULL; \
            s->data[i].length = 0; \
        } \
    } \
}

#define __init_function__(parent_type, child_type) \
inline void init_##parent_type (parent_type *s) { \
    s->length = 0; \
    s->data = NULL; \
}

#define __create_function__(parent_type, child_type) \
inline parent_type *create_##parent_type () { \
    parent_type* result = (parent_type*) safe_malloc (sizeof (parent_type)); \
    init_##parent_type (result); \
    return result; \
}

#define __delete_function_simple__(parent_type, child_type) \
inline void delete_##parent_type (parent_type *s) { \
    if (s) { \
        free (s->data); \
        s->data = NULL; \
    } \
}

```

```

    s->length = 0; \
  } \
}

#define __delete_function_complex__(parent_type, child_type) \
inline void delete_##parent_type (parent_type *s) { \
  if (s) { \
    for (unsigned int i = 0; i < s->length; i++) \
      delete_##child_type (s->data + i); \
    free (s->data); \
    s->data = NULL; \
    s->length = 0; \
  } \
}

#define __functions_complex__(parent_type, child_type) \
__ensure_function_complex__(parent_type, child_type) \
__init_function__(parent_type, child_type) \
__create_function__(parent_type, child_type) \
__delete_function_complex__(parent_type, child_type)

#define __functions_simple__(parent_type, child_type) \
__ensure_function_simple__(parent_type, child_type) \
__init_function__(parent_type, child_type) \
__create_function__(parent_type, child_type) \
__delete_function_simple__(parent_type, child_type)

#define __define_structure_complex__(parent_type, child_type) \
__hierarchical_structure__(parent_type, child_type) \
__functions_complex__(parent_type, child_type)

#define __define_structure_simple__(parent_type, child_type) \
__hierarchical_structure__(parent_type, child_type) \
__functions_simple__(parent_type, child_type)

```

## 2.7 Random Numbers

At the moment I am using a cellular automaton-based random number generator implemented by Tony Pasqualoni. It outperforms more traditional random number generators such as the Mersenne Twister algorithm by a factor of three, and as far as is known produces results of comparable quality. The source code for this generator may be found in `ca_rng.h`. These wrapper functions facilitate easy replacement of the random number generator being used.

```

inline void seed_rng (unsigned int seed)
{ca_rng_initialize (seed);}

conditional_inline unsigned int random_integer ()
{return ca_rng_get_int ();}

conditional_inline double normalized_random ()
{return (double) random_integer () / (double) 0xfffffffful;}

```

## 3 Model Framework

The model framework involves defining types and functions for convenient manipulation of biological structures. First, we define the data types that rep-

resent the biological structures, and then we define the functions that manipulate those structures.

```
typedef uint8_t      amino_acid;
typedef uint8_t      codon;
typedef uint8_t      nucleotide;
typedef uint64_t     codon_set;
typedef uint8_t      bool;

#define FALSE (0 == 1)
#define TRUE (! FALSE)
```

### 3.1 Codons and Amino Acids

Codons are represented by 8-bit integers, as are amino acids. Each codon takes on values between 0 and 63, inclusive. Amino acids take on values between 0 and 20, inclusive. Nucleotide values are defined according to some biological properties. Flipping the lower-order bit performs a transition, for instance. If  $U = 0$ ,  $C = 1$ ,  $A = 2$ , and  $G = 3$ , then the numerical representation of the codon UCA is the binary number 000110. So the least-significant nucleotide is the third one. This must be strictly adhered to throughout the code for consistency. In terms of ordering, the nucleotides are ordered increasing by significance; so nucleotide 0 is A, nucleotide 1 is C, and nucleotide 2 is U. This facilitates the bitshifting required to extract or replace individual nucleotides.

Also, the structure of codons is purely numerical at the moment, but that could change. When we want to use a codon as a number, we should abstract that through the `_idx` macro, which, should the structure of a codon ever change in the future, will return the codon as an index and not as a structure.

```
#define _idx(c) (c)

#define _u 0
#define _c 1
#define _a 2
#define _g 3
#define _co(n1, n2, n3) (((n1) << 4) | ((n2) << 2) | (n3))

#define ser 0
#define phe 1
#define leu 2
#define tyr 3
#define sto 4
#define cys 5
#define trp 6
#define pro 7
#define his 8
#define gln 9
#define arg 10
#define ile 11
#define met 12
#define thr 13
#define asn 14
#define lys 15
#define val 16
#define ala 17
#define asp 18
#define glu 19
```

```
#define gly 20
#define n_acids 21
```

We implement a lookup table to translate codons into amino acids.

```
static const amino_acid amino_lookup[64] = {
    phe, phe, leu, leu,
    ser, ser, ser, ser,
    tyr, tyr, sto, sto,
    cys, cys, sto, trp,

    leu, leu, leu, leu,
    pro, pro, pro, pro,
    his, his, gln, gln,
    arg, arg, arg, arg,

    ile, ile, ile, met,
    thr, thr, thr, thr,
    asn, asn, lys, lys,
    ser, ser, arg, arg,

    val, val, val, val,
    ala, ala, ala, ala,
    asp, asp, glu, glu,
    gly, gly, gly, gly};
```

### 3.2 Codon Functions

We define a number of other accessor functions for working with codons.

```
inline codon inherit (codon c1, codon c2, unsigned int ni) {
    codon mask = (codon) (3 << (ni * 2));
    return (c1 & (0x3f ^ mask)) | (c2 & mask);
}

inline codon inherit_nuc (codon c, nucleotide n, unsigned int ni)
    {return inherit (c, _co(n, n, n), ni);}

inline nucleotide pick (codon c, unsigned int ni) {
    unsigned int shift_amount = ni * 2;
    return (codon) ((c & (3 << shift_amount)) >> shift_amount);
}

inline codon mask (codon c1, codon c2) {
    codon diff = c1 ^ c2;
    return (diff & 0x15) | ((diff & 0x2a) >> 1);
}

inline unsigned int changes (codon c1, codon c2)
    {return (unsigned) (mask (c1, c2) % 3);}

inline bool synonymous (codon c1, codon c2)
    {return amino_lookup[(unsigned int) _idx(c1)] == amino_lookup[(unsigned int) _idx(c2)];}

inline bool is_purine (nucleotide n)
    {return !! (n & 2);}
```

### 3.3 Structure Definitions

Later on, we'll need a full hierarchy of structures, including sequences and populations. We define those here so that we can define all dependent struc-

tures. For more information about how these `define_structure` macros work, see section 2.6.

```
__define_structure_simple__(sequence, codon)
__define_structure_complex__(population, sequence)
__define_structure_simple__(fitness_vector, double)
__define_structure_simple__(site_statistic, double)
```

We also define structures to trace the evolutionary history of an individual. These are arranged in a binary lattice that for the most part is a tree. The only real exception to the tree structure is a recombination event, which splits the evolutionary history.

```
typedef struct {
    unsigned int parent_index;
    unsigned int recombination_nucleotide; // = 0 if no recombination, = 3 if position 0.
    unsigned int other_parent_index;

    codon state_before_mutation;
    codon state_after_mutation;
} individual_history_node;

__define_structure_simple__(individual_history, individual_history_node)
__define_structure_complex__(generation_history, individual_history)
__define_structure_complex__(population_history, generation_history)
```

### 3.4 Codon Sets

During theta calculations, we maintain a set of the codons that have appeared in a column. Since the number of codons is constant, there is a shortcut that can be taken to optimize the code. By defining a codon set as a 64-bit integer, we can use number-theoretic operations to yield extremely fast and compact code. An element is a member of the set if the bit with that element's index is set to a 1. This representation allows a convenient way to count the number of elements in a set with relatively few operations. We retrieve the values of all bits whose significance is 1 modulo 63, then those that are 2 modulo 63, etc. until all bits are accounted for. (For convenience, we split the number into its two 32-bit halves and then add the halves together after separation.) Taking the whole number modulo 63 gives the number of ones that were present in the original quantity. The bit masks used are these:

```
01000001000001000001000001000001 = 0x41041041
10000010000010000010000010000010 = 0x82082082
00000100000100000100000100000100 = 0x04104104
00001000001000001000001000001000 = 0x08208208
00010000010000010000010000010000 = 0x10410410
00100000100000100000100000100000 = 0x20820820
```

Next, we must also ascertain whether a set containing exactly two codons represents a singleton difference for Nei-Gojobori calculations. This means that

the two codons must differ by exactly one nucleotide. An efficient way to test this is to reconstruct the bits that make up their numerical value by taking binary masks over the codon set. If they fall into the same mask, then they are the same; otherwise, they differ.

Specifically, suppose we have a set that looks like this:

0000...0001001

This indicates that codons 0 and 3 are in the set (we go from right to left). To determine whether the two least-significant bits of the codons are the same, we take four bitmasks:

0001... = 0x11111111  
 0010... = 0x22222222  
 0100... = 0x44444444  
 1000... = 0x88888888

A change occurred in the lower-order two bits iff two of these masks returned nonzero values when anded with the set. Similarly, we can determine whether a change occurred within the two middle bits by these four bitmasks:

0000000000001111... = 0x000f000f  
 0000000011110000... = 0x00f000f0  
 0000111100000000... = 0x0f000f00  
 1111000000000000... = 0xf000f000

The same criterion may be reused for the first nucleotide. Lastly, we must determine whether the codons differ in their most-significant bits by breaking the 64-bit set into four 16-bit regions. We use these four bitmasks for this:

$1^{16}0^{48} = 0xffff000000000000$   
 $0^{16}1^{16}0^{32} = 0x0000ffff00000000$   
 $0^{32}1^{16}0^{16} = 0x00000000ffff0000$   
 $0^{48}1^{16} = 0x000000000000ffff$

```

inline codon_set codon_set_add (codon_set s, codon c)
{return s | (0x1ull << _idx(c));}

inline bool codon_set_member (codon_set s, codon c)
{return !! (s & (0x1ull << _idx(c)));}

inline codon_set codon_set_union (codon_set s1, codon_set s2)
{return s1 | s2;}

```

```

inline codon_set codon_set_count_half (codon_set _s) {
    return (_s & 0x41041041ull) +
        ((_s & 0x82082082ull) >> 1) +
        ((_s & 0x04104104ull) >> 2) +
        ((_s & 0x08208208ull) >> 3) +
        ((_s & 0x10410410ull) >> 4) +
        ((_s & 0x20820820ull) >> 5);
}

conditional_inline unsigned int codon_set_count (codon_set s) {
    // Count by halves. This code uses a mod-63 algorithm described in the
    // writeup.
    return codon_set_count_half ((s >> 32) & 0xffffffffull) % 63 +
        codon_set_count_half (s & 0xffffffffull) % 63;
}

inline bool changes_in_group (codon_set a, codon_set b, codon_set c, codon_set d) {
    // If a, b, c, and d are bitmasks, then this method returns true if all of the
    // bitmasks but one are zero. In the function codon_set_singleton, this has
    // the effect of determining whether all of the set bits belong to the same
    // bitgroup.
    return ! ((a == 0 && b == 0 && c == 0) ||
        (a == 0 && b == 0 && d == 0) ||
        (a == 0 && c == 0 && d == 0) ||
        (b == 0 && c == 0 && d == 0));
}

conditional_inline int codon_set_singleton (codon_set s) {
    // Returns -1 if the set is not a singleton, and 0, 1, or 2 if the set is a
    // singleton. The 0, 1, or 2 will indicate whether the least significant,
    // middle, or most significant nucleotide changed, respectively.
    // Check the two lower nucleotides first.
    codon_set _s = ((s >> 32) & 0xffffffffull) | (s & 0xffffffffull);
    bool c0 = changes_in_group (_s & 0x11111111ull,
        _s & 0x22222222ull,
        _s & 0x44444444ull,
        _s & 0x88888888ull);
    bool c1 = changes_in_group (_s & 0x00f000full,
        _s & 0x00f000f0ull,
        _s & 0x0f000f00ull,
        _s & 0xf000f000ull);

    if (c0 && c1)
        return -1;
    else {
        bool c2 = changes_in_group (s & 0xffff000000000000ull,
            s & 0x0000ffff00000000ull,
            s & 0x00000000ffff0000ull,
            s & 0x000000000000ffffull);

        if (c2 && (c1 || c0))
            return -1;
        else if (c0)
            return 0;
        else if (c1)
            return 1;
        else
            return 2;
    }
}

```

Here's where it gets even scarier. We want to determine whether a set of codons is synonymous or nonsynonymous. Specifically, do the codons in the set code for exactly one amino acid or more than one? To do this with the minimal number of operations requires bitmasks, which we use below. The guiding idea is that there are a lot of patterns in the genetic code, and by exploiting those we can handle many cases at once. Later I'll write up the actual

process for deriving the hexadecimal numbers that appear here.

```

inline bool codon_set_inside (codon_set s, codon_set mask)
{return s == (s & mask);}

conditional_inline bool codon_set_synonymous (codon_set s) {
// Determines whether exactly two codons in a codon set are synonymous with
// one another.
if (codon_set_inside (s, 0x00f000f000f000f0ull))
return codon_set_inside (s, 0x00f0000000000000ull) ||
codon_set_inside (s, 0x000000f000000000ull) ||
codon_set_inside (s, 0x0000000000f00000ull) ||
codon_set_inside (s, 0x00000000000000f0ull);

if (codon_set_inside (s, 0xf00f0000f00f0000ull))
return codon_set_inside (s, 0xf000000000000000ull) ||
codon_set_inside (s, 0x000f000000000000ull) ||
codon_set_inside (s, 0x0000000f00000000ull) ||
codon_set_inside (s, 0x000000000000f000ull);

if (codon_set_inside (s, 0x0300030003000300ull) !=
codon_set_inside (s, 0x0c000c000c000c00ull))
return codon_set_inside (s, 0x0f00000000000000ull) ||
codon_set_inside (s, 0x00000f0000000000ull) ||
codon_set_inside (s, 0x00000000f0000000ull) ||
codon_set_inside (s, 0x00000000000000f0ull);

// Handle cases where only single-coding values are present. Since we assume
// that multiple distinct values are present in this codon set, if an amino
// acid that has only one coding option is found, then the set must be
// nonsynonymous.
if (codon_set_inside (s, 0x0000000800008000ull))
return FALSE;

// Below are the more irregular cases. These include the bridging of serine,
// leucine, and arginine across groups, and rows containing just one deviant,
// such as isoleucine-methionine at the beginning of the third group.
return codon_set_inside (s, 0x0000000000004c00ull) ||
codon_set_inside (s, 0x0000000000f000cull) ||
codon_set_inside (s, 0x0000c000f0000000ull) ||
codon_set_inside (s, 0x00003000000000f0ull) ||
codon_set_inside (s, 0x0000000700000000ull) ||
codon_set_inside (s, 0x000000000000003ull) ||
codon_set_inside (s, 0x0000000000003000ull);
}

```

We also will need to know which codons changed. The simplest format for this is to return the codon mask that would be returned when comparing two individual codons.

```

conditional_inline codon codon_set_mask (codon_set s) {
codon result = 0;
if (! (codon_set_inside (s, 0xffff000000000000ull) ||
codon_set_inside (s, 0x0000ffff00000000ull) ||
codon_set_inside (s, 0x00000000ffff0000ull) ||
codon_set_inside (s, 0x000000000000ffffull)))
result |= 0x10;

if (! (codon_set_inside (s, 0xf000f000f000f000ull) ||
codon_set_inside (s, 0x0f000f000f000f00ull) ||
codon_set_inside (s, 0x00f000f000f000f0ull) ||
codon_set_inside (s, 0x000f000f000f000full)))
result |= 0x4;

if (! (codon_set_inside (s, 0x8888888888888888ull) ||
codon_set_inside (s, 0x4444444444444444ull) ||
codon_set_inside (s, 0x2222222222222222ull) ||

```

```

        codon_set_inside (s, 0x11111111111111111111))
    result |= 0x1;
return result;
}

```

### 3.5 Sequences

A sequence is an array of codons. We define a sequence and some methods for sequences in general here. We also define the notion of a population, which is an array of sequences. In practice, we use hierarchical structures to represent these, since we want access to the size of the data as well as the data itself. Recombination involves splicing two sequence pieces together somehow. Since a recombination split point can be within a single codon, we have these three scenarios:

1. The split point is a nucleotide 0 within a codon, or between codons. Then the nucleotides whose indices are strictly less than the split point come from sequence 1, and all others come from sequence 2.
2. The split point is at nucleotide 1, or after the first nucleotide in the codon. Then the first nucleotide of the codon after the split point is from sequence 1, and all after it are from sequence 2. To get the intermediate codon, we simply inherit nucleotide 2 from the codon in sequence 1.
3. The split point is at nucleotide 2, or after the second nucleotide of the codon. To get the intermediate codon, we inherit nucleotide 0 from the codon in sequence 2.

```

conditional_inline sequence *copy_sequence (sequence *source, sequence *dest) {
    // This function is called a lot, so any optimizations here are going to make
    // a big difference. We use memcpy to copy the actual data, since that is
    // going to be the fastest way to copy a continuous segment of memory.
    ensure_sequence (dest, source->length);
    memcpy (dest->data, source->data, source->length * sizeof (codon));
    return dest;
}

conditional_inline sequence *recombine_sequences (
    sequence *s1, sequence *s2, sequence *dest, unsigned int nucleotide_index) {
    unsigned int ci = nucleotide_index / 3;
    unsigned int ni = nucleotide_index % 3;

    ensure_sequence (dest, s2->length);

    // Copy in the codons from the first sequence.
    memcpy (dest->data, s1->data, sizeof (codon) * ci);

    // During the split, copy the partial codon.
    if (ni == 0)
        dest->data[ci] = s1->data[ci];
    else
        dest->data[ci] = (ni == 1) ? inherit (s2->data[ci], s1->data[ci], 2) :
            inherit (s1->data[ci], s2->data[ci], 0);

    // Copy the rest.
}

```

```

memcpy (dest->data + ci + 1, s2->data + ci + 1, sizeof (codon) * (s2->length - (ci + 1)));
return dest;
}

```

---

### 3.5.1 Sequence serialization forms

Sequences can be serialized into two different forms. One form, the *nucleotide form*, has the following specifications:

1. The output consists only of the characters A, C, G, T, U, and a dash (-), which is ignored.
2. If a codon in the sequence has nucleotide  $n_0$  at position 0,  $n_1$  at position 1, and  $n_2$  at position 2, then it is output as the characters  $n_2n_1n_0$ .
3. No whitespace is allowed inside a sequence.

An example of a sequence in nucleotide form is ACA-UCU-GGG-TAG. The other form, *packed form*, has these specifications:

1. The first character of output is an exclamation point (!).
2. Each codon is represented as a single character from the string "ABCDEFGHIJKLMN O PQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/", such that the codon whose numerical index is  $i$  is represented by the  $i$  character in that string.
3. A dash is allowed at any point in the sequence. Dashes are ignored but allowed for readability.<sup>1</sup>
4. No whitespace is allowed inside a sequence.

An example of a sequence in packed form is !6cA+5gnG9/r. This encodes the human-readable sequence beginning with AAA-CGU-UUU-GGA.

The `print_sequence` function writes a sequence to a file in either nucleotide or packed form. The `read_sequence` function does the opposite; given a string containing a nucleotide-form or packed-form sequence, it returns a newly constructed sequence object.

```

inline void set_nucleotide_at_position (sequence *s, nucleotide n, unsigned int position) {
    // Position is the position in nucleotides, not codons.
    s->data[position / 3] = inherit_nuc (s->data[position / 3], n, 2 - (position % 3));
}

conditional_inline void print_sequence (sequence *s, FILE *f, bool packed) {
    // Prints a serialized representation of the sequence s to the file f. If
    // packed is true, then packed form is used; otherwise nucleotide form is
    // used.

    const char *packed_chars      = "ABCDEFGHIJKLMN O PQRSTUVWXYZ"
                                   "abcdefghijklmnopqrstuvwxyz"
                                   "0123456789+/" ;
}

```

<sup>1</sup>Though all bets are off about readability of packed sequences.

```

const char *nucleotide_chars = "TCAG";

if (packed) {
    fprintf (f, "!");

    for (unsigned int i = 0; i < s->length; i++)
        fprintf (f, "%c", packed_chars[s->data[i]]);
} else
    for (unsigned int i = 0; i < s->length; i++)
        fprintf (f, "%c%c%c", nucleotide_chars[pick (s->data[i], 2)],
                nucleotide_chars[pick (s->data[i], 1)],
                nucleotide_chars[pick (s->data[i], 0)]);
}

conditional_inline sequence *read_sequence (const char *input, sequence *dest) {
    unsigned int position = 0; // In nucleotides if reading a nucleotide-form
                             // sequence, and codons if packed-form.
    unsigned int l = 0; // Same here.
    sequence *result = dest;

    if (result == NULL)
        result = create_sequence ();

    if (input[0] == '!') {
        // Read a packed-form sequence. We start at one to skip over the leading
        // exclamation point.
        for (unsigned int i = 1; i < strlen (input); i++)
            if ((input[i] >= 'A' && input[i] <= 'Z') ||
                (input[i] >= 'a' && input[i] <= 'z') ||
                (input[i] >= '0' && input[i] <= '9') ||
                input[i] == '+' || input[i] == '/')
                l++;

        ensure_sequence (result, l);

        for (unsigned int i = 1; i < strlen (input); i++)
            if (input[i] >= 'A' && input[i] <= 'Z')
                result->data[position++] = input[i] - 'A';
            else if (input[i] >= 'a' && input[i] <= 'z')
                result->data[position++] = input[i] - 'a' + 26;
            else if (input[i] >= '0' && input[i] <= '9')
                result->data[position++] = input[i] - '0' + 52;
            else if (input[i] == '+')
                result->data[position++] = 62;
            else if (input[i] == '/')
                result->data[position++] = 63;
        } else {
            // Read a nucleotide-form sequence. We start at zero in this case because
            // there was no leading exclamation point.
            for (unsigned int i = 0; i < strlen (input); i++)
                if (input[i] == 'A' || input[i] == 'C' || input[i] == 'G' ||
                    input[i] == 'T' || input[i] == 'U')
                    l++;

            if (l % 3) {
                fprintf (stderr, "Error:_Nucleotide-form_sequence_being_read_(%s)_has_an_incomplete_codon.\n", input);
                exit (EXIT_FAILURE);
            }

            ensure_sequence (result, l / 3);

            for (unsigned int i = 0; i < l / 3; i++)
                result->data[i] = 0;

            for (unsigned int i = 0; i < strlen (input); i++)
                switch (input[i]) {
                    case 'A': set_nucleotide_at_position (result, _a, position++); break;
                    case 'C': set_nucleotide_at_position (result, _c, position++); break;
                }
        }
    }
}

```

```

        case 'G': set_nucleotide_at_position (result, _g, position++); break;
        case 'T':
        case 'U': set_nucleotide_at_position (result, _u, position++); break;
    }
}

return result;
}

```

## 3.6 Populations

Populations are indexed collections of sequences. We work mainly with populations, so we define several functions that take a population and return given results.

### 3.6.1 Selective fitness

Natural selection prefers some individuals to others for reproductive fitness. Thus some individuals are more likely to be parents than others, and that bias is reflected by assigning a coefficient to each individual within a population. When we consider the fitness coefficients of all of the individuals, then we have a fitness vector, keyed just as the individuals are.

While the population functions provide the ability to compute the fitnesses of each individual, they do not make decisions about parentage. This is to facilitate options such as using an isolation matrix, which is implemented later.

Fitness vectors have one important peculiarity, and that is that they are stored cumulatively. So a fitness vector of all ones would look like  $\langle 1, 2, 3, 4, \dots, n \rangle$ . The reason for this is to enable a binary search instead of a linear one for parent selection. For a population size of 500, this makes it about 25 times faster. Algorithmically speaking, it makes a neutral model run  $O(n \log n)$  instead of  $O(n^2)$ . The fitness vector structure is defined in section 3.3.

```

#define __fitness_vector_binary_operation__(function_name, op) \
conditional_inline fitness_vector *function_name (fitness_vector *v1, fitness_vector *v2, \
fitness_vector *dest) { \
    double last_1 = 0.0; \
    double last_2 = 0.0; \
    ensure_fitness_vector (dest, v1->length); \
    for (unsigned int i = 0; i < v1->length; i++) { \
        dest->data[i] = (v1->data[i] - last_1) op (v2->data[i] - last_2) + \
            ((i > 0) ? dest->data[i - 1] : 0.0); \
        last_1 = v1->data[i]; \
        last_2 = v2->data[i]; \
    } \
    return dest; \
}

__fitness_vector_binary_operation__(fitness_vector_add, +)
__fitness_vector_binary_operation__(fitness_vector_mul, *)

inline double fitness_vector_magnitude (fitness_vector *v)
{return v->data[v->length - 1];}

conditional_inline unsigned int choose_individual (fitness_vector *v, population *p) {
    // Returns the index of a randomly chosen individual.
    // This function is  $O(n \log n)$  in the size of the fitness vector, since a

```

```

// binary search is used.
if (v != NULL) {
    // Binary-search the population.
    double      point   = normalized_random () * fitness_vector_magnitude (v);
    unsigned int shift   = (v->length / 4) | 1;
    unsigned int result  = v->length / 2;

    if (fitness_vector_magnitude (v) == 0.0) {
        fprintf (stderr, "Internal_error:_The_magnitude_of_the_fitness_vector_is_zero.\n");
        exit (EXIT_FAILURE);
    }

    while (TRUE) {
        if (result >= v->length) result = v->length - 1;
        if (result < 0)          result = 0;

        if (v->data[result] < point)
            result += shift;
        else if (result > 0 && v->data[result - 1] > point)
            result -= shift;
        else
            return result;

        shift = (shift / 2) | 1;
    }
} else
    // Simply choose a random element, since they all carry the same weight.
    return random_integer () % p->length;
}

```

## 4 Simulation Parameters

The simulation parameter structure stores the global data for a single program run. The data here corresponds strongly to the profile directives defined in section 2.1.

### 4.1 Statistical Structure

We compute a set of statistics for each population. Since we may wish to add more information later, we encapsulate this information into a large structure whose fields are filled in by several different functions (see section 5).

```
typedef struct {
```

#### 4.1.1 Variance coefficients

We need to compute some coefficients to find the variance of Tajima's  $D$ . These are described in more detail in section 5.1.

```

    unsigned int    n;                // Number of sequences per population
    double          a1, a2, b1, b2,
    double          c1, c2, e1, e2; // Coefficients for variance calculation

```

### 4.1.2 Final results

We compile the results below into final measurements such as  $s_{syn}$ ,  $s_{non}$ ,  $\theta_{syn}$ ,  $\theta_{non}$ , and  $D_{syn}$  and  $D_{non}$ .

<b>double</b>	s_syn,	s_non;
<b>double</b>	theta_syn,	theta_non;
<b>double</b>	pi_syn,	pi_non;
<b>double</b>	d_syn,	d_non;

### 4.1.3 Discrete pi data

The discrete pi statistical estimator uses discrete pathways. For this, we don't collect any data ahead of time, so we just use a site statistic to store the results.

site_statistic	discrete_pi_seg;
site_statistic	discrete_pi_non;
<b>double</b>	discrete_pi_seg_bias, discrete_pi_seg_var;
<b>double</b>	discrete_pi_non_bias, discrete_pi_non_var;

### 4.1.4 Continuous pi data

If we create a continuous pairwise estimator, then we'll need to solve a  $64 \times 64$  system of linear equations and store the results here. For now, however, this is not implemented.

### 4.1.5 Parsimonious genetic likelihoods

Genetic likelihood data consists of three 64-element vectors, where each entry represents the number of nonsynonymous changes present at the given codon at the given position.

<b>double</b>	gl_data[3][64];
site_statistic	gl_site_non;
<b>double</b>	gl_non_bias, gl_non_var;

### 4.1.6 Discrete Nei and Gojobori ratios

Nei and Gojobori ratios are computed ahead of time for a sample. We store the number of singletons at each site and the ratio of those singletons that were nonsynonymous. We also keep track of the number of times the singleton data was used to estimate the nonsynonymity of a multipliton (for auditing purposes).

<b>double</b>	dng_data[3];
<b>unsigned int</b>	dng_size[3];
<b>unsigned int</b>	dng_multiplitions[3];

#### 4.1.7 Theta

The results for theta are stored sitewise so that we can measure the effectiveness of various computing methods. Right now, we want to see whether minimizing the estimator variance for theta is an improvement over using whole selection.

```
site_statistic      theta_site_seg;
site_statistic      theta_site_non;
double              theta_site_seg_bias, theta_site_seg_var;
double              theta_site_non_bias, theta_site_non_var;
```

#### 4.1.8 Traced values

During the simulation process, we trace the actual mutation paths of the individual codon sites if the `full-tracing` option is specified. Then we use backtracing to verify the accuracy of theta.

```
site_statistic      traced_site_seg;
site_statistic      traced_site_non;
```

#### 4.1.9 Random estimator error

To provide a benchmark for our statistical estimators, we use a random number generator and measure its bias and variance.

```
double              random_seg_bias, random_seg_var;
double              random_non_bias, random_non_var;
```

#### 4.1.10 Generic initialization methods

This concludes the data elements of the statistical structure. We also define methods to initialize and destroy statistics.

```
} statistics;

void initialize_statistics (statistics *s) {
  init_site_statistic (&s->discrete_pi_seg);
  init_site_statistic (&s->discrete_pi_non);
  init_site_statistic (&s->gl_site_non);
  init_site_statistic (&s->theta_site_seg);
  init_site_statistic (&s->theta_site_non);
  init_site_statistic (&s->traced_site_seg);
  init_site_statistic (&s->traced_site_non);
}

void delete_statistics (statistics *s) {
  delete_site_statistic (&s->discrete_pi_seg);
  delete_site_statistic (&s->discrete_pi_non);
  delete_site_statistic (&s->gl_site_non);
  delete_site_statistic (&s->theta_site_seg);
  delete_site_statistic (&s->theta_site_non);
  delete_site_statistic (&s->traced_site_seg);
  delete_site_statistic (&s->traced_site_non);
}
```

## 4.2 Selection and Drift Constants

Each model of selection and drift has an associated identification constant. We define those here for easy reference later.

```
#define NEUTRAL_SELECTION 0
#define PURIFYING_SELECTION 1
#define DIVERSIFYING_SELECTION 2

#define NEUTRAL_DRIFT 0
#define SHARP_BOTTLENECK_DRIFT 1
#define EVEN_SUBDIVISION_DRIFT 2
```

## 4.3 Simulation Parameter Structures

Simulation parameters include the initial population size, the ancestral sequence, HKY-85 parameters, recombination rate, etc. Fitness functions take a population and the index of a sequence and should return a double between 0 and 1 indicating the relative fitness of that sequence.

By default, we assume the neutral model and add in selective or drift influences. An example of such an influence is purifying selection, where individuals who deviate from a predefined optimum are less fit. The next step is to define the rules for mutation under the neutral model. Mutation is generally simple, as is recombination.

These structures define various aspects of a running simulation's behavior. We break them down symbolically, although the in-memory representation of the `simulation_parameters` structure will be contiguous since most of the child structures are instantiated in-place and not pointed to.

### 4.3.1 Output parameters

These parameters govern the format of the output. They correspond to the directives defined in section 2.1.3.

```
typedef struct {
    unsigned int    trials;
    bool            print_sequences;
    bool            make_profile;
    bool            print_each_generation;
    bool            full_tracing;
    bool            trace_multiplifitons_only;
} output_parameters;
```

### 4.3.2 Process parameters

These parameters govern the processing phase in the general sense. They correspond to the directives defined in section 2.1.4.

```
typedef struct {
    bool            simulation_only;
    bool            statistics_only;
} process_parameters;
```

### 4.3.3 General-purpose biological parameters

These parameters govern the general biological operation of the model. They correspond to the directives defined in section 2.1.5.

```
typedef struct {
    double mutations_per_nucleotide;
    double transitions_per_transversion;
    double frequency_a;
    double frequency_c;
    double frequency_g;
    double frequency_u;
} hky85_parameters;

typedef struct {
    unsigned int population_size;
    unsigned int generations;
    unsigned int sample_size;
    hky85_parameters mutation;
    double recombination_rate;
    sequence *ancestral_sequence;
    unsigned int selection_model;
    unsigned int drift_model;
} general_parameters;
```

### 4.3.4 Model-specific parameters

These parameters govern model-specific behavior. They correspond to the directives defined in section 2.1.6.

```
typedef struct {
    double selection_coefficient;
    sequence *optimal_sequence;
} purifying_selection_parameters;

typedef struct {
    double selection_coefficient;
} diversifying_selection_parameters;

typedef struct {
    unsigned int start;
    unsigned int duration;
    unsigned int size;
} sharp_bottleneck_parameters;

typedef struct {
    unsigned int start;
    unsigned int duration;
    unsigned int size;
} even_subdivision_parameters;

typedef struct {
    purifying_selection_parameters purifying;
    diversifying_selection_parameters diversifying;
    sharp_bottleneck_parameters bottleneck;
    even_subdivision_parameters subdivision;
} model_parameters;
```

### 4.3.5 Internal parameters

Parameters internal to the simulation are used as a workspace as the simulation is running. These are primarily here for optimization, although full tracing

requires the history to be persistent across generations.

Of note is the `diversifying_selection_cache` parameter, which stores a list of codons at each site and the number of each. Since diversifying selection requires a comparison with each codon in the column and the mutation rate per codon site is fairly low, we can optimize this by caching the count of each codon in the site and comparing just against the listed codons.

We can take this another step as well. Since the amino acids are what we compare against, we can just store the amino acid translations and their frequencies. This should end up reducing the  $O(n^2)$  algorithm to  $O(n)$ , a 400-fold speed improvement for the standard parameters.

```
typedef struct {
    amino_acid      freqs[21];
    unsigned int    total;
} diversifying_aa_freq_site;

__define_structure_simple__(diversifying_aa_freq, diversifying_aa_freq_site)

typedef struct {
    // These fields are used for internal purposes. We put them here to be
    // effectively global to the thread being run.
    fitness_vector *parent_fitness_vector;
    fitness_vector *child_fitness_vector;
    fitness_vector *result_fitness_vector;
    population_history *history;
    double assumed_gl_variance;
    unsigned int random_seed;
    statistics s;
    diversifying_aa_freq *diversifying_cache;
    double diversifying_cached_pi;
} simulation_internal_data;
```

#### 4.3.6 Simulation parameters

We define the full simulation parameter structure here. The variable names are shortened to facilitate easy access to the fields.

```
typedef struct {
    output_parameters      o;
    process_parameters     p;
    general_parameters     g;
    model_parameters      m;
    simulation_internal_data i;
    char *image_name;
} simulation_parameters;

void initialize_simulation_parameters (simulation_parameters *sp) {
    struct timeval tv;
    gettimeofday (&tv, NULL);

    sp->i.random_seed      = tv.tv_sec ^ ((uint32_t) tv.tv_usec << 11);
    sp->o.trials            = 50;
    sp->g.population_size  = 500;
    sp->g.generations      = 1000;
    sp->g.sample_size      = 20;

    sp->g.selection_model  = NEUTRAL_SELECTION;
    sp->g.drift_model      = NEUTRAL_DRIFT;

    sp->g.mutation.frequency_a = 0.25;
```

```

sp->g.mutation.frequency_c          = 0.25;
sp->g.mutation.frequency_g          = 0.25;
sp->g.mutation.frequency_u          = 0.25;
sp->g.mutation.mutations_per_nucleotide = 3.4e-5;
sp->g.mutation.transitions_per_transversion = 1.3;

sp->g.recombination_rate             = 2.5e-5;
sp->i.assumed_gl_variance             = 0.0;

sp->g.ancestral_sequence = read_sequence (
    "UGU-ACA-AGA-CCC-AAC-AAC-AAU-ACA-AUA-AAA-AGU-AUA-CAU-AUG-GGA-CUA-GGG-AGG-"
    "ACA-UUU-UAU-ACA-ACA-GGA-GAA-GUA-AUA-GGA-GAU-AUA-AGA-CAA-GCA-CAU-UGU", NULL);
sp->m.purifying.optimal_sequence = read_sequence (
    "UGU-ACA-AGA-CCC-AAC-AAC-AAU-ACA-AUA-AAA-AGU-AUA-CAU-AUG-GGA-CUA-GGG-AGG-"
    "ACA-UUU-UAU-ACA-ACA-GGA-GAA-GUA-AUA-GGA-GAU-AUA-AGA-CAA-GCA-CAU-UGU", NULL);

sp->m.purifying.selection_coefficient = 0.1;
sp->m.diversifying.selection_coefficient = 0.1;
sp->o.print_sequences                  = FALSE;
sp->o.print_each_generation            = FALSE;
sp->o.full_tracing                     = FALSE;
sp->o.trace_multiplifitons_only        = FALSE;
sp->o.make_profile                     = FALSE;

sp->p.simulation_only                  = FALSE;
sp->p.statistics_only                  = FALSE;

sp->m.bottleneck.start                 = 200;
sp->m.bottleneck.duration               = 500;
sp->m.bottleneck.size                  = 50;

sp->m.subdivision.start                 = 200;
sp->m.subdivision.duration              = 500;
sp->m.subdivision.size                 = 50;

sp->i.parent_fitness_vector             = NULL;
sp->i.child_fitness_vector              = NULL;
sp->i.result_fitness_vector             = NULL;
sp->i.history                           = NULL;
sp->i.diversifying_cache                = create_diversifying_aa_freq ();

    initialize_statistics (&sp->i.s);
}

void delete_simulation_parameters (simulation_parameters *sp) {
    // We do not free the image name because the memory for this is usually
    // stack-allocated, not heap-allocated.

    delete_sequence (sp->g.ancestral_sequence);
    free (sp->g.ancestral_sequence);
    sp->g.ancestral_sequence = NULL;

    delete_sequence (sp->m.purifying.optimal_sequence);
    free (sp->m.purifying.optimal_sequence);
    sp->m.purifying.optimal_sequence = NULL;

    delete_fitness_vector (sp->i.parent_fitness_vector);
    free (sp->i.parent_fitness_vector);
    sp->i.parent_fitness_vector = NULL;

    delete_fitness_vector (sp->i.child_fitness_vector);
    free (sp->i.child_fitness_vector);
    sp->i.child_fitness_vector = NULL;

    delete_fitness_vector (sp->i.result_fitness_vector);
    free (sp->i.result_fitness_vector);
    sp->i.result_fitness_vector = NULL;
}

```

```

delete_population_history (sp->i.history);
free (sp->i.history);
sp->i.history = NULL;

delete_diversifying_aa_freq (sp->i.diversifying_cache);
free (sp->i.diversifying_cache);
sp->i.diversifying_cache = NULL;

delete_statistics (&sp->i.s);
}

```

## 5 Statistics

The statistical analysis is the most dynamic part of this model, in the sense that the analyses performed here are under a lot of debate at any given time. Thus they are coded for maximum flexibility, often at some expense to performance.

### 5.1 Variance Coefficients

The variance coefficients used in Tajima's  $D$  calculation are described in the background literature. The formulas for the various constants are implemented as follows:

$$\begin{aligned}
a_1 &= \sum_{i=1}^{n-1} \frac{1}{i} \\
a_2 &= \sum_{i=1}^{n-1} \frac{1}{i^2} \\
b_1 &= \frac{n+1}{3(n-1)} \\
b_2 &= 2 \cdot \frac{n^2+n+3}{9n(n-1)} \\
c_1 &= b_1 - \frac{1}{a_1} \\
c_2 &= b_2 - \frac{n+2}{na_1} + \frac{a_2}{a_1^2} \\
e_1 &= \frac{c_1}{a_1} \\
e_2 &= \frac{c_2}{a_1^2 + a_2}
\end{aligned}$$

```

void compute_variance_coefficients (population *p, simulation_parameters *sp, statistics *result) {
// Populate the sample size and then compute the variance coefficients.
result->n          = p->length;
result->a1         = 0.0;
result->a2         = 0.0;
}

```

```

for (unsigned int i = 1; i < result->n; i++) {
    result->a1 += 1.0 / (double) i;
    result->a2 += 1.0 / (double) (i * i);
}

result->b1 = (double) (result->n + 1) / (3.0 * (double) (result->n - 1));
result->b2 = 2.0 * (double) (result->n * result->n + result->n + 3) /
           (double) (9 * result->n * (result->n - 1));

result->c1 = result->b1 - 1.0 / result->a1;
result->c2 = result->b2 -
           (double) (result->n + 2) / (result->a1 * (double) result->n) +
           result->a2 / (result->a1 * result->a1);

result->e1 = result->c1 / result->a1;
result->e2 = result->c2 / (result->a1 * result->a1 + result->a2);
}

```

## 5.2 Methods for Sitewise Computation

These methods make it simpler to write sitewise statistical analysis functions.

```

conditional_inline codon_set codon_site_set (population *p, unsigned int ci) {
    // Builds and returns the codon set constructed from codon site ci.
    codon_set result = 0ull;
    for (unsigned int i = 0; i < p->length; i++)
        result = codon_set_add (result, _idx(p->data[i].data[ci]));
    return result;
}

inline bool codon_set_multipliton (codon_set cs) {
    // Returns true iff cs contains a multipliton.
    return (codon_set_singleton (cs) == -1) && (codon_set_count (cs) > 2);
}

```

## 5.3 Parsimonious Genetic Likelihood Ratios

We build a table of numbers, each representing the likelihood that a change in nucleotide  $i$  of codon  $j$  will be nonsynonymous assuming equal nucleotide proportions. We then use this table to estimate the nonsynonymity of the variance at each codon site.

```

void compute_genetic_likelihoods (population *p, simulation_parameters *sp, statistics *result) {
    for (codon c = 0; c < 64; c++)
        for (unsigned int p = 0; p < 3; p++) {
            result->gl_data[p][c] = 0.0;
            for (nucleotide n = 0; n < 4; n++)
                if (!synonymous (c, inherit_nuc (c, n, p)))
                    result->gl_data[p][c] += 1.0 / 3.0;
        }

    ensure_site_statistic (&result->gl_site_non, p->data[0].length);

    // Now estimate these likelihoods across sites.
    for (unsigned int i = 0; i < p->data[0].length; i++) {
        result->gl_site_non.data[i] = 0.0;
        codon_set cs = codon_site_set (p, i);
        codon cs_mask = codon_set_mask (cs);

        for (unsigned int j = 0; j < 3; j++)

```

```

// Check for variance at position j.
if (cs_mask & (1 << (2 * j))) {
// If there's variance, then add up the average of all of the codons in
// this column.
double total = 0.0;
for (unsigned int k = 0; k < p->length; k++)
total += result->gl_data[j][_idx(p->data[k].data[i])];
result->gl_site_non.data[i] += total / (double) p->length;
}
}
}

```

## 5.4 Nei and Gojobori Ratios

Using the Nei and Gojobori method, the likelihood of a change in a particular nucleotide position being synonymous or nonsynonymous can be estimated for a given population. Computing these ratios involves scanning for codon loci within a population such that all of the codons across sequences at that locus differ only once, called a singleton change. The ratio of singleton changes whose effects are nonsynonymous to those whose effects are synonymous is the Nei and Gojobori ratio for that nucleotide index.

In practice, we need to know more than just the ratios because sometimes the ratios will be based on many observations and sometimes they will be based on just a few. The strength of the data plays a role in whether or not it is used, and if it is used, how much weight it receives. The alternative to using Nei and Gojobori data is using a combinatorial estimate of how likely each nucleotide is to cause a change. This assumes neutral nucleotide preference, which is often not true in nature.

```

void compute_nei_gojobori (population *p, simulation_parameters *sp, statistics *result) {
for (unsigned int i = 0; i < 3; i++) {
result->dng_data[i] = 0.0;
result->dng_size[i] = 0;
}

// Run through the data building codon sets and searching for singletons.
for (unsigned int i = 0; i < p->data[0].length; i++) {
codon_set cs = 0ull;
for (unsigned int j = 0; j < p->length; j++)
cs = codon_set_add (cs, p->data[j].data[i]);

// Check to see if it is a segregating singleton site. If not, then pass it up.
if (codon_set_count (cs) == 2) {
signed int singleton_position = codon_set_singleton (cs);
if (singleton_position != -1) {
// See the definition of codon_set_singleton if the usage of
// singleton_position looks peculiar.
result->dng_size[singleton_position]++;
if (! codon_set_synonymous (cs))
result->dng_data[singleton_position] += 1.0;
}
}
}

// Since the site data slots contain the number of nonsynonymous observations,
// all we need to do is divide by the total number of samples taken.
for (unsigned int i = 0; i < 3; i++)
if (result->dng_size[i] != 0)
result->dng_data[i] /= (double) result->dng_size[i];
}

```

## 5.5 Theta Statistic

Theta measures the proportion of sites that show variance. To do this, we look at each codon locus across individuals and consider the set of distinct values observed. If this set falls into the same equivalence class of amino acid coding, then it is synonymous; otherwise, it is nonsynonymous.

In practice, we establish a gradient that blurs this distinction somewhat. If there is variance among a nucleotide site, then we add that site's nonsynonymity value to the nonsynonymity of the site. (Correspondingly, we add the difference to 1 to the synonymity of the site).

### 5.5.1 Estimator combination methods

There are two ways to combine the Nei and Gojobori estimator  $X$  and the genetic likelihood estimator  $Y$ . Currently, we use a cutoff method by default; for sufficiently small data sets ( $n = 0$ ), we let our estimate be  $Y$ ; otherwise, we let the estimate be  $X$ . A proposed modification of this method involves estimating the relative variances of  $X$  and  $Y$ , assuming that neither is biased, and solving for a linear combination to minimize the combined variance. The derivation is below, assuming that  $cov(X, Y) = 0$ :

$$\begin{aligned} V(X) &= \frac{1}{n} \\ V(Y) &= c \\ a + b &= 1 \\ V(aX + bY) &= a^2V(X) + b^2V(Y) \end{aligned}$$

Minimizing the last equation involves zeroing the derivative:

$$\begin{aligned} \frac{d}{da} \left[ \frac{a^2}{n} + (1 - a^2)c \right] &= \frac{2a}{n} + (-2 + 2a)c \\ 0 &= \frac{2a}{n} + (-2 + 2a)c \\ c &= \frac{a}{n} + 2ac \\ a &= \frac{nc}{1 + nc} \end{aligned}$$

Here,  $n$  is the number of singleton instances in a given position, and  $c$  is the constant variance assumed of the genetic likelihood method. The linear combination that minimizes error, assuming that each estimator is unbiased, is:

$$\left(\frac{nc}{1+nc}\right)X + \left(1 - \frac{nc}{1+nc}\right)Y$$

When considering the value of the genetic likelihood estimator, we average across all values that appear in a column.

## 5.5.2 Codon ambiguity methods

In the population given by sequences AAATTT, GACTTA, and CACTTA, there is a Nei and Gojobori singleton at position 3. However, the first codon is also variant at position 1, which has no Nei and Gojobori singleton. During the summer of 2007, three different ways of handling this situation were documented. Two of them are these:

1. Flag the first codon in its entirety as being ambiguous, since it has at least one nucleotide site that has no Nei and Gojobori data. Then use genetic likelihoods, averaged across all codon values in the first codon, on the first and third positions.
2. Use the available Nei and Gojobori data on the third position of the first codon and use genetic likelihoods on the first position.

We currently use option 2. That's why we have a loop in this code that's very similar to the site-wise computation of genetic likelihood data in section 5.3.

```

void compute_theta_site (population *p, simulation_parameters *sp, statistics *result,
                        unsigned int codon_site, codon_set cs) {
    unsigned int cs_size = codon_set_count (cs);

    // If the site segregates, then we compute theta for it. Otherwise, we pass
    // it up.
    result->theta_site_seg.data[codon_site] = 0.0;

    codon cs_mask = codon_set_mask (cs);

    for (unsigned int i = 0; i < 3; i++)
        if (cs_mask & (1 << (i * 2)))
            result->theta_site_seg.data[codon_site] += 1.0;

    if (cs_size == 2 && codon_set_singleton (cs) != -1) {
        // It's a singleton. In this case, it's either synonymous or not.
        if (! codon_set_synonymous (cs))
            result->theta_site_non.data[codon_site] = 1.0;
    } else if (cs_size > 1)
        // Determine which nucleotides segregate.
        for (unsigned int i = 0; i < 3; i++)
            if (cs_mask & (1 << (i * 2))) {
                // The condition on this if statement is just a way of determining
                // whether we see mutation at a given nucleotide site.
                double nc = sp->i.assumed_gl_variance * (double) result->dng_size[i];
                double ng_factor = (sp->i.assumed_gl_variance == 0.0) ?
                    ((result->dng_size[i] > 0) ? 1.0 : 0.0) :
                    nc / (1.0 + nc);
                double co_factor = 1.0 - ng_factor;
                double ng_est = result->dng_data[i];
                double co_est = 0.0;
            }
    }
}

```

```

        double non          = ng_factor * ng_est;

        // Count the occurrence of the multipliton.
        result->dng_multiplitons[i]++;

        if (co_factor > 0.0) {
            // Average out all of the combinatorial numbers for the given
            // codons. Weight by the frequency of each, so we actually get the
            // values from the population itself.
            double total = 0.0;
            for (unsigned int j = 0; j < p->length; j++)
                total += result->gl_data[i][_idx(p->data[j].data[codon_site])];
            co_est = total / (double) p->length;
            non += co_factor * co_est;
        }

        result->theta_site_non.data[codon_site] += non;
    }
}

void compute_theta (population *p, simulation_parameters *sp, statistics *result) {
    // Populates the theta portion of the statistics.
    compute_genetic_likelihoods (p, sp, result);
    compute_nei_gojobori (p, sp, result);

    result->s_syn = result->s_non = 0.0;

    for (unsigned int i = 0; i < 3; i++)
        result->dng_multiplitons[i] = 0;

    ensure_site_statistic (&result->theta_site_seg, p->data[0].length);
    ensure_site_statistic (&result->theta_site_non, p->data[0].length);

    // Now go through the columns and compute theta.
    for (unsigned int i = 0; i < p->data[0].length; i++) {
        result->theta_site_seg.data[i] = result->theta_site_non.data[i] = 0.0;
        compute_theta_site (p, sp, result, i, codon_site_set (p, i));

        result->s_syn += result->theta_site_seg.data[i] - result->theta_site_non.data[i];
        result->s_non += result->theta_site_non.data[i];
    }

    // Divide by the correctional factor.
    result->theta_syn = result->s_syn / result->a1;
    result->theta_non = result->s_non / result->a1;
}

```

## 5.6 Discrete Pathways

Our modification separates the synonymous from the nonsynonymous changes. For singleton changes, it is a simple distinction; we simply check the synonymity or nonsynonymity of the change. However, some changes are ambiguous. For example, if AAA changes to CAC, did it change by first mutating to CAA or by first mutating to AAC? We cannot find out, but we can take the average of all of the possibilities by using discrete estimation.

```

inline double non_path_1 (codon _c1, codon _c2)
    {return synonymous (_c1, _c2) ? 0.0 : 1.0;}

inline double non_path_2 (codon _c1, codon i, codon _c2)
    {return non_path_1 (_c1, i) + non_path_1 (i, _c2);}

inline double non_path_3 (codon _c1, codon i1, codon i2, codon _c2)

```

```

    {return non_path_2 (_c1, i1, i2) + non_path_1 (i2, _c2);}

inline double discrete_pathways_nonsynonymity (codon c1, codon c2, simulation_parameters *sp) {
    unsigned int nc = changes (c1, c2);

    if (nc == 0) {
        // No or three changes. We wouldn't be calling this procedure if there
        // weren't any changes, so there must be three. (Actually, everything
        // works for zero changes as well, but it's a slow way to come up with
        // zero changes.)

        codon i11 = inherit (c1, c2, 0), i12 = inherit (i11, c2, 1);
        codon i21 = inherit (c1, c2, 0), i22 = inherit (i21, c2, 2);
        codon i31 = inherit (c1, c2, 1), i32 = inherit (i31, c2, 0);
        codon i41 = inherit (c1, c2, 1), i42 = inherit (i41, c2, 2);
        codon i51 = inherit (c1, c2, 2), i52 = inherit (i51, c2, 0);
        codon i61 = inherit (c1, c2, 2), i62 = inherit (i61, c2, 1);

        return (non_path_3 (c1, i11, i12, c2) + non_path_3 (c1, i21, i22, c2) +
                non_path_3 (c1, i31, i32, c2) + non_path_3 (c1, i41, i42, c2) +
                non_path_3 (c1, i51, i52, c2) + non_path_3 (c1, i61, i62, c2)) / 6.0;
    } else if (nc == 1) {
        // One change.
        return non_path_1 (c1, c2);
    } else {
        // Two changes. In a way, this is the most tricky since we have to
        // specifically exclude the configuration that doesn't change anything.

        codon i1 = inherit (c1, c2, 0);
        codon i2 = inherit (c1, c2, 1);
        codon i3 = inherit (c1, c2, 2);

        return (((i1 != c1) ? non_path_2 (c1, i1, c2) : 0.0) +
                ((i2 != c1) ? non_path_2 (c1, i2, c2) : 0.0) +
                ((i3 != c1) ? non_path_2 (c1, i3, c2) : 0.0)) / 2.0;
    }
}
}

```

## 5.7 Pi Statistic

The pi statistic gives the average amount of variation within each site, averaged across sites. It is computed by counting the number of differences within each codon locus across sequences and dividing by the number of pairwise comparisons made.

Similarly to diversifying selection, pi may be linearized by pre-counting the list of codons. This imposes some unnecessary overhead for low sequence counts, but for high sequences it reduces a quadratic to a linear operation.

```

void compute_pi (population *p, simulation_parameters *sp, statistics *result) {
    // Fills in the pi computations of the statistical structure.
    double multiplier = 1.0 / (double) (p->length * (p->length - 1) / 2);
    result->pi_syn = result->pi_non = 0.0;

    ensure_site_statistic (&result->discrete_pi_seg, p->data[0].length);
    ensure_site_statistic (&result->discrete_pi_non, p->data[0].length);

    // Go through each site looking for codon differences. When we find them, then
    // we use discrete pathways.
    for (unsigned int pos = 0; pos < p->data[0].length; pos++) {
        result->discrete_pi_seg.data[pos] =
            result->discrete_pi_non.data[pos] = 0.0;
    }
}

```

```

unsigned int codon_counts[64] = {0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0,
                                0, 0, 0, 0, 0, 0, 0, 0};

// Count up all of the codon occurrences. This is linear in the number of
// sequences.
for (unsigned int i = 0; i < p->length; i++)
    codon_counts[_idx(p->data[i].data[pos])]++;

// This is a constant-time algorithm in the number of sequences. The
// constant is large (2048), but better than something like 100,000,000.
for (unsigned int c1 = 0; c1 < 63; c1++)
    for (unsigned int c2 = c1 + 1; c2 < 64; c2++) {
        // Count the difference.
        unsigned int    n_changes    = changes (c1, c2);
        double         non_factor   = discrete_pathways_nonsynonymity (c1, c2, sp);
        unsigned int    bias        = codon_counts[c1] * codon_counts[c2];

        if (n_changes == 0) n_changes = 3;

        result->discrete_pi_seg.data[pos] += (double) n_changes * (double) bias;
        result->discrete_pi_non.data[pos] += non_factor * (double) bias;
    }

result->discrete_pi_seg.data[pos] *= multiplier;
result->discrete_pi_non.data[pos] *= multiplier;

result->pi_syn += result->discrete_pi_seg.data[pos] - result->discrete_pi_non.data[pos];
result->pi_non += result->discrete_pi_non.data[pos];
}
}

```

## 5.8 Tajima's D

Tajima's  $D$  statistic is calculated after  $\theta$  and  $\pi$  are known. We use the variance coefficients,  $\pi$ , and  $\theta$  for each component of  $D$ . Specifically,

$$D_{syn} = \frac{\pi_{syn} - \theta_{syn}}{\sqrt{e_1 s_{syn} + e_2 s_{syn} (s_{syn} - 1)}}$$

$$D_{non} = \frac{\pi_{non} - \theta_{non}}{\sqrt{e_1 s_{non} + e_2 s_{non} (s_{non} - 1)}}$$

These formulas are implemented in the `compute_d` function. We have to watch out for divide-by-zero errors because  $S_{syn}$  and  $S_{non}$  are sometimes zero, zeroing the whole denominator. We handle this by saying that if either is zero, then  $D$  is zero as well.

```

void compute_d (population *p, simulation_parameters *sp, statistics *result) {
    double denom_syn = sqrt (result->s_syn * result->e1 + result->e2 * result->s_syn *
                            (result->s_syn - 1.0));
    if (denom_syn != 0.0)
        result->d_syn = (result->pi_syn - result->theta_syn) / denom_syn;
    else

```

```

    result->d_syn = 0.0;

    double denom_non = sqrt (result->s_non * result->e1 + result->e2 * result->s_non *
                             (result->s_non - 1.0));
    if (denom_non != 0.0)
        result->d_non = (result->pi_non - result->theta_non) / denom_non;
    else
        result->d_non = 0.0;
}

```

## 5.9 Full Tracing

Mutations are tracked as they occur during the simulation if the `full-tracing` directive is specified in the profile (see section 2.1). A full trace of the evolutionary history is kept of all of the organisms, and to find the number of mutations we backtrack the tree to find the MRCA (either by state or by descent), totaling the synonymous and nonsynonymous mutations as we go. Then we compare the actual numbers to the estimates given by genetic likelihoods, Nei and G-jobori, and a random number generator to establish relative biases.

```

bool is_mrca_descent (bool *individuals_seen, unsigned int n) {
    // If multiple individuals exist, then we're not at a MRCA yet.
    bool seen_one_yet = FALSE;

    for (unsigned int i = 0; i < n; i++)
        if (individuals_seen[i]) {
            if (seen_one_yet)
                return FALSE;
            else
                seen_one_yet = TRUE;
        }

    return TRUE;
}

bool is_mrca_state (bool *individuals_seen, unsigned int n, generation_history *h,
                   unsigned int codon_index) {
    // Find the first seen current codon state and make sure everything else lines
    // up with it.
    codon first_state = 0xff; // Anything > 128 means we haven't seen anything yet.

    for (unsigned int i = 0; i < n; i++)
        if (individuals_seen[i]) {
            if (first_state & 0x80)
                first_state = h->data[i].data[codon_index].state_after_mutation;
            else if (first_state != h->data[i].data[codon_index].state_after_mutation)
                return FALSE;
        }

    return TRUE;
}

unsigned int history_total_mutations (individual_history_node *child) {
    // Returns the total number of mutations that occurred from parent to child.
    // Since each history node stores not only its own codon but the codon before
    // mutation (whether that arises from recombination or otherwise is not
    // relevant), we just compare those two codons to one another.
    if (child->state_before_mutation == child->state_after_mutation)
        return 0;
    else {
        unsigned int c = changes (child->state_before_mutation, child->state_after_mutation);
        if (c == 0)

```

```

        return 3;
    else
        return c;
    }
}

unsigned int history_non_mutations (individual_history_node *child) {
    // Returns the number of nonsynonymous mutations that occurred from parent to
    // child. In case two or more mutations occurred within the same codon (in one
    // generation), they are counted from left to right.
    codon c1 = child->state_before_mutation;
    codon c2 = child->state_after_mutation;

    if (c1 == c2)
        return 0;
    else if (changes (c1, c2) == 1)
        return synonymous (c1, c2) ? 1 : 0;
    else {
        // Assume all nucleotides are variant. This is OK because we count only
        // nonsynonymous mutations. If a nucleotide was not variant, then the codon
        // will be synonymous.
        codon i1 = inherit (c1, c2, 2);
        codon i2 = inherit (i1, c2, 1);
        codon i3 = inherit (i2, c2, 0);
        return !! synonymous (c1, i1) + !! synonymous (i1, i2) +
            !! synonymous (i2, i3) + !! synonymous (i3, c2);
    }
}

uint64_t count_mutations (population *p, simulation_parameters *sp, bool by_state,
    unsigned int codon_index) {
    // Returns the number of nonsynonymous and the number of total mutations that
    // have occurred since the MRCA of the population. The MRCA may be decided by
    // state or by descent; if by_state is true, then state is used; otherwise,
    // descent is used. This function assumes a maximum population size of
    // sp->g.population_size, since the population is often from the sample and does
    // not represent the entire evolving population. The result is encoded such
    // that the number of non mutations occupies the high 32 bits and the total
    // number of mutations occupies the low 32 bits.

    bool        *individuals_seen      = (bool*) malloc (sizeof (bool) * sp->g.population_size);
    bool        *last_individuals_seen = (bool*) malloc (sizeof (bool) * sp->g.population_size);
    bool        *temp                  = NULL;
    unsigned int non_mutations         = 0;
    unsigned int total_mutations       = 0;
    int         current_generation     = sp->g.generations - 1;

    // Initialize to have seen only the individuals in the population.
    for (unsigned int i = 0; i < sp->g.population_size; i++)
        individuals_seen[i] = (i < p->length);

    // Now backtrace until only one element in individuals_seen is set.
    while (current_generation >= 0 &&
        ! (by_state ?
            is_mrca_state (individuals_seen, sp->g.population_size,
                sp->i.history->data + current_generation, codon_index) :
            is_mrca_descent (individuals_seen, sp->g.population_size))) {

        // Go back one generation. This means rounding up all of the parents and
        // adding them to the last_individuals_seen array. Then we swap the two
        // arrays and decrement the current generation.
        for (unsigned int i = 0; i < sp->g.population_size; i++)
            last_individuals_seen[i] = FALSE;

        for (unsigned int i = 0; i < sp->g.population_size; i++)
            if (individuals_seen[i]) {
                individual_history *h = sp->i.history->data[current_generation].data + i;

```

```

        // Find this guy's parent(s) and mark them as having been seen.
        last_individuals_seen[h->data[codon_index].parent_index] = TRUE;

        if (h->data[codon_index].recombination_nucleotide != 0)
            last_individuals_seen[h->data[codon_index].other_parent_index] = TRUE;

        // Add this individual's mutations to the totals.
        non_mutations += history_non_mutations (h->data + codon_index);
        total_mutations += history_total_mutations (h->data + codon_index);
    }

    // Now swap this and the last individuals seen, and then decrement the
    // current generation.
    temp = individuals_seen;
    individuals_seen = last_individuals_seen;
    last_individuals_seen = temp;
    current_generation--;
}

free (individuals_seen);      individuals_seen      = NULL;
free (last_individuals_seen); last_individuals_seen = NULL;

return ((uint64_t) non_mutations << 32) | total_mutations;
}

void compute_traced_values (population *p, simulation_parameters *sp, statistics *result) {
    // Here we go across codon sites and compare the actual traced nonsynonymity
    // of multipliflons with the estimates made by Nei and Gojobori, genetic
    // likelihoods, and random numbers.

    unsigned int sample_size = p->data[0].length;

    ensure_site_statistic (&result->traced_site_seg, p->data[0].length);
    ensure_site_statistic (&result->traced_site_non, p->data[0].length);

    for (unsigned int ci = 0; ci < p->data[0].length; ci++) {
        // Test the estimators here. We provide actual information on all sites,
        // not just multipliflon sites.

        uint64_t      mutation_counts = count_mutations (p, sp, FALSE, ci);
        unsigned int  non_mutations   = mutation_counts >> 32;
        unsigned int  total_mutations = mutation_counts & 0xfffffffffull;

        result->traced_site_seg.data[ci] = (double) total_mutations /
                                           (double) (sample_size * (sample_size - 1) / 2);

        result->traced_site_non.data[ci] = (total_mutations > 0) ?
                                           (double) non_mutations / (double) total_mutations :
                                           0.0;
    }
}

```

## 5.10 Statistical Verification

We compute the bias and variance of each of the estimators used in calculating statistics. Since this is computationally expensive (that is to say, computing the traced values in section 5.9 is computationally expensive), we enable it only when `full-tracing` has been specified.

```

void compute_estimator_accuracy (population *p, simulation_parameters *sp, statistics *result) {
    compute_traced_values (p, sp, result);

    result->gl_non_bias      = result->gl_non_var      = 0.0;
    result->theta_site_seg_bias = result->theta_site_seg_var = 0.0;
}

```

```

result->theta_site_non_bias = result->theta_site_non_var = 0.0;
result->random_seg_bias    = result->random_seg_var    = 0.0;
result->random_non_bias    = result->random_non_var    = 0.0;

for (unsigned int i = 0; i < result->traced_site_seg.length; i++)
  if (! sp->o.trace_multipliflits_only || codon_set_multipliflits (codon_site_set (p, i))) {
    double gl_non_error      = result->gl_site_non.data[i] - result->traced_site_non.data[i];
    double theta_seg_error   = result->theta_site_seg.data[i] - result->traced_site_seg.data[i];
    double theta_non_error   = result->theta_site_non.data[i] - result->traced_site_non.data[i];
    double random_seg_error  = normalized_random () - result->traced_site_seg.data[i];
    double random_non_error  = normalized_random () - result->traced_site_non.data[i];

    result->gl_non_bias      += gl_non_error;
    result->gl_non_var      += gl_non_error * gl_non_error;

    result->theta_site_seg_bias += theta_seg_error;
    result->theta_site_seg_var += theta_seg_error * theta_seg_error;

    result->theta_site_non_bias += theta_non_error;
    result->theta_site_non_var += theta_non_error * theta_non_error;

    result->random_seg_bias    += random_seg_error;
    result->random_seg_var    += random_seg_error * random_seg_error;

    result->random_non_bias    += random_non_error;
    result->random_non_var    += random_non_error * random_non_error;
  }
}

```

## 6 Simulation Runtime

The simulation is defined in the following sections.

### 6.1 Mutation

We use the HKY-85 model to provide mutation likelihoods. Its parameter set includes a general mutation rate, a transition/transversion ratio, and relative frequencies of each of the four nucleotides. An important shortcut can be taken for mutation simulation. Since the mutation rate per nucleotide is fairly low, it is unlikely that a mutation will occur at all. Instead of checking each site, we can quickly compute the probability that we have no mutations and then check that condition. If it is false, then we adjust all future probabilities accordingly.

```

inline double nucleotide_probability (nucleotide n1, nucleotide n2, hky85_parameters *p) {
  if (n1 == n2)
    return 0.0;
  else {
    double k = p->transitions_per_transversion /
              (p->transitions_per_transversion + 1.0);
    return (is_purine (n1) == is_purine (n2)) ? k : 1.0 - k;
  }
}

inline nucleotide choose_nucleotide (nucleotide existing, hky85_parameters *p) {
  // Returns the nucleotide that is mutated to. It is guaranteed to change.
  double pu = p->frequency_u * nucleotide_probability (_u, existing, p);
  double pc = p->frequency_c * nucleotide_probability (_c, existing, p);
  double pa = p->frequency_a * nucleotide_probability (_a, existing, p);
  double pg = p->frequency_g * nucleotide_probability (_g, existing, p);
}

```

```

double selector = normalized_random () * (pu + pc + pa + pg);

if (selector < pu)           return _u;
else if (selector < pu + pc) return _c;
else if (selector < pu + pc + pa) return _a;
else                        return _g;
}

conditional_inline void mutate_sequence (sequence *s, simulation_parameters *sp,
                                       unsigned int generation, unsigned int individual) {
    hky85_parameters *p = &sp->g.mutation;
    double given_probability = pow (1.0 - p->mutations_per_nucleotide, s->length * 3);

    while (normalized_random () > given_probability) {
        // A mutation actually occurs.
        unsigned int position = random_integer () % (s->length * 3);

        // When we get here, we are guaranteed a change.
        codon new_codon =
            inherit_nuc (s->data[position / 3],
                       choose_nucleotide (pick (s->data[position / 3], position % 3), p),
                       position % 3);

        if (sp->o.full_tracing)
            // Record the new codon state in the appropriate slot.
            // We explicitly don't alter the original codon because its state is lost,
            // and we need to track multiple mutations.
            sp->i.history->data[generation].data[individual].data[position / 3].state_after_mutation = new_codon;

        s->data[position / 3] = new_codon;

        // Each mutation decreases the likelihood that others will happen.
        given_probability /= 1.0 - p->mutations_per_nucleotide;
    }
}

```

## 6.2 Model Functions

To simulate the effects of different evolutionary forces, we add bias to our simulation through the use of these functions.

### 6.2.1 Neutral model

The neutral model is selected by default.

### 6.2.2 Purifying selection

Purifying selection begins with 1 and multiplies by a selection factor for each amino acid difference between the individual and the optimum. This function won't end up being expensive per invocation, but it will be called  $O(ng)$  times, where  $n$  is the number of individuals in a population and  $g$  is the number of generations. For that reason, any optimizations available here are extremely valuable. (I am not aware of any possibilities for optimization at the moment.)

```

conditional_inline double purifying_selection (population *p, unsigned int idx, simulation_parameters *sp) {
    double result = 1.0;
    double drop_factor = 1.0 - sp->m.purifying.selection_coefficient;
    unsigned int i = 0;
}

```

```

for (i = 0; i < p->data[idx].length; i++)
  if (amino_lookup[(unsigned int) _idx(p->data[idx].data[i])] == sto ||
      amino_lookup[(unsigned int) _idx(sp->m.purifying.optimal_sequence->data[i])] == sto)
    break;
  else if (!synonymous (p->data[idx].data[i], sp->m.purifying.optimal_sequence->data[i]))
    result *= drop_factor;

// Take into account stop codons. Anything after the first stop codon is a
// difference.
if (i < p->data[idx].length)
  result *= pow(drop_factor, p->data[idx].length - i);

return result;
}

```

### 6.2.3 Diversifying selection

Diversifying selection begins with 1 and multiplies by a selection factor for each amino acid difference that the organism has at each position.

This algorithm is optimized as follows: Before ascertaining the fitness of each parent, we first run through each codon site and count up the number of amino acids that appear there. (We ignore anything after a stop codon.) So each site ends up with a count of each amino acid and a total number of amino acids translated at that site. Then when we look at a single individual, we can just go through the sites and add the total number of amino acids translated minus the number that were the same as the one for that site to the number of differences. For example, if 40 total amino acids were coded at site 3, the individual whose fitness is to be determined codes for a tryptophan at that site, and 5 others code for a tryptophan at that site, then the number of differences is 35.

This optimization reduces the  $O(nl)$  algorithm, where  $n$  is the number of individuals and  $l$  is the sequence length, to a simple  $O(l)$  algorithm for approximately a 200-factor speed increase on a population size of 500. The only slow part is the cache operation, but that must be performed only once per generation, which is already  $O(nl)$  due to the memory copy operation for sequence parentage.

The `cache_diversifying_selection` method should be called before any fitness assessment is performed, and then fitness assessment may proceed normally with the `diversifying_selection` function.

```

conditional_inline void cache_diversifying_selection (population *p,
                                                    simulation_parameters *sp) {
  // Go through each site and count up the number of each amino acid that is
  // coded.

  // Zero out all of the frequencies.
  for (unsigned int ci = 0; ci < p->data[0].length; ci++)
    for (unsigned int i = 0; i < n_acids; i++)
      sp->i.diversifying_cache->data[ci].freqs[i] = 0;

  // This loop needs to be organized by individuals because we break out of the
  // inner loop upon hitting a stop codon. If we did it the other way, then
  // instead of ignoring the rest of the sequence it would ignore the rest of
  // the site across sequences.
  for (unsigned int i = 0; i < p->length; i++)
    // Now go through each site and total up the amino acid counts.

```

```

// Make sure to stop counting after a stop codon.
for (unsigned int ci = 0;
    ci < p->data[i].length &&
    amino_lookup[_idx(p->data[i].data[ci])] != sto;
    ci++)
    sp->i.diversifying_cache->data[ci].freqs[amino_lookup[_idx(p->data[i].data[ci])]]++;

// Count the number of stop codons in each position. Keep any stop codons that
// occurred before, since these sequences should no longer be considered.
unsigned int stops = 0;

for (unsigned int ci = 0; ci < p->data[0].length; ci++) {
    for (unsigned int i = 0; i < p->length; i++)
        if (amino_lookup[_idx(p->data[i].data[ci])] == sto)
            stops++;

    sp->i.diversifying_cache->data[ci].total = p->length - stops;
}

// Finally, cache a computed pi value. This is done linearly as well, which
// is simple because the frequency table has already been computed.
//
// There is some subtle optimization here. I started with this:
//
//     for (ci)
//         for (i)
//             pi += (p->length - f[i]) / p->length
//
// Since p->length is invariant, we can perform the division later. Then we
// have this:
//
//     pi += p->length - f[i]
//
// p->length is still invariant, so we may defer the addition to later as
// well. The only caveat is that we must multiply the deferred term by the
// number of times it was to be added.
sp->i.diversifying_cached_pi = 0.0;

for (unsigned int ci = 0; ci < p->data[0].length; ci++)
    for (unsigned int i = 0; i < n_acids; i++)
        sp->i.diversifying_cached_pi -= (double) sp->i.diversifying_cache->data[ci].freqs[i];

sp->i.diversifying_cached_pi += (double) p->length * n_acids * p->data[0].length;
sp->i.diversifying_cached_pi /= (double) p->length * p->data[0].length;
}

conditional_inline double diversifying_selection (
    population *p, unsigned int idx,
    simulation_parameters *sp) {
    unsigned int differences = 0;

    // Optimized computation! Using the diversifying selection cache, we apply
    // this reasoning: Each amino acid's frequency is recorded in the table. If we
    // see the amino acid X in a sequence, then we subtract the number of
    // occurrences of X in its codon site from the total number of individuals to
    // get the number of differences in that site. This reduces  $O(n^2)$  to  $O(n)$ .

    for (unsigned int ci = 0;
        ci < p->data[idx].length && amino_lookup[_idx(p->data[idx].data[ci])] != sto;
        ci++)
        differences += sp->i.diversifying_cache->data[ci].total -
            sp->i.diversifying_cache->data[ci].freqs[amino_lookup[_idx(p->data[idx].data[ci])]];

    double average_differences_from_others = (double) differences / (double) p->length;

    return pow (1.0 / (1.0 - sp->m.diversifying.selection_coefficient),
        average_differences_from_others - sp->i.diversifying_cached_pi);
}

```

## 6.2.4 Sharp population bottlenecks

We temporarily create a crash in the population during a bottleneck. This means creating an isolation matrix that has an upper-left square of ones for a time, then restoring the matrix to all ones.

```
inline double sharp_population_bottleneck (
    unsigned int generation, unsigned int parent_idx,
    unsigned int child_idx, simulation_parameters *sp) {
    if (generation >= sp->m.bottleneck.start &&
        generation < sp->m.bottleneck.start + sp->m.bottleneck.duration &&
        (parent_idx > sp->m.bottleneck.size ||
         child_idx > sp->m.bottleneck.size))
        // This interaction is subject to a bottleneck.
        return 0.0;
    else
        // The bottleneck is not in effect.
        return 1.0;
}
```

## 6.2.5 Even population subdivision

For even subdivision, we temporarily partition the population into evenly sized groups (minus the fact that the group size might not evenly divide the population size), and then reunite them after some time. This means creating a matrix with multiple squares of ones along the primary diagonal, and everything else zero.

In practice, we do something a little different. Instead of making squares, we make a sparse field of ones throughout the matrix that ends up being equivalent. The only reason we do this is because the modulus operator makes more sense to me personally than the division operator.

```
inline double even_population_subdivision (
    unsigned int generation, unsigned int parent_idx,
    unsigned int child_idx, simulation_parameters *sp) {
    unsigned int denom = sp->g.population_size / sp->m.subdivision.size;
    if (generation >= sp->m.subdivision.start &&
        generation < sp->m.subdivision.start + sp->m.subdivision.duration &&
        parent_idx % denom != child_idx % denom)
        return 0.0;
    else
        return 1.0;
}
```

## 6.3 Parent Selection and Recombination

We cannot modify the population in-place, so we actually have two separate populations. The first represents the parent generation and the second represents the child generation. After each generation, the pointers to the two populations are exchanged and the old child population becomes the parent population.

[More documentation later...]

```

conditional_inline void populate_parent_fitness (
    population *parents, simulation_parameters *sp) {
    // Creates the vector for the parents' fitnesses and stores it in sp.
    for (unsigned int i = 0; i < parents->length; i++) {
        double current = 0.0;

        switch (sp->g.selection_model) {
            case DIVERSIFYING_SELECTION:
                current = diversifying_selection (parents, i, sp);
                break;
            case PURIFYING_SELECTION:
                current = purifying_selection (parents, i, sp);
                break;
            default:
                fprintf (stderr, "Internal_error:_Selection_model_%d_does_not_exist.\n", sp->g.selection_model);
                exit (EXIT_FAILURE);
        }

        sp->i.parent_fitness_vector->data[i] = current +
            ((i > 0) ? sp->i.parent_fitness_vector->data[i - 1] : 0.0);
    }
}

conditional_inline void consider_drift_on_fitness (
    population *parents, unsigned int generation,
    unsigned int child_index, simulation_parameters *sp) {
    // Multiplies drift into the resulting fitness vector.
    if (sp->g.drift_model == EVEN_SUBDIVISION_DRIFT) {
        for (unsigned int i = 0; i < parents->length; i++) {
            double current = 0.0;

            switch (sp->g.drift_model) {
                case EVEN_SUBDIVISION_DRIFT:
                    current = even_population_subdivision (generation, i, child_index, sp);
                    break;
                default:
                    fprintf (stderr, "Internal_error:_Drift_model_%d_does_not_exist.\n", sp->g.drift_model);
                    exit (EXIT_FAILURE);
            }

            sp->i.child_fitness_vector->data[i] = current +
                ((i > 0) ? sp->i.child_fitness_vector->data[i - 1] : 0.0);
        }

        fitness_vector_mul (sp->i.parent_fitness_vector, sp->i.child_fitness_vector,
            sp->i.result_fitness_vector);
    }
}

conditional_inline void do_trace_recombination (
    unsigned int split_point, unsigned int generation,
    unsigned int child_index, unsigned int p1, unsigned int p2,
    population *parents, population *children, simulation_parameters *sp) {
    // Fills in the history for the given individual based on the fact that a
    // recombination event occurred.
    if (sp->o.full_tracing) {
        // Inherit a bunch of codon histories from parent 1 and parent 2.
        for (unsigned int i = 0; i < split_point / 3; i++) {
            individual_history_node *n =
                sp->i.history->data[generation].data[child_index].data + i;
            n->parent_index = p1;
            n->recombination_nucleotide = 0;
            n->other_parent_index = 0xfffffffful;
            n->state_before_mutation = n->state_after_mutation =
                parents->data[p1].data[i];
        }
    }
}

```

```

    for (unsigned int i = split_point / 3 + 1; i < parents->data[p1].length; i++) {
        individual_history_node *n =
            sp->i.history->data[generation].data[child_index].data + i;
        n->parent_index = p2;
        n->recombination_nucleotide = 0;
        n->other_parent_index = 0xfffffffful;
        n->state_before_mutation = n->state_after_mutation =
            parents->data[p2].data[i];
    }

    // Now handle the mixed case.
    individual_history_node *n =
        sp->i.history->data[generation].data[child_index].data + split_point / 3;
    n->parent_index = p1;
    n->recombination_nucleotide = (split_point % 3) ? split_point % 3 : 3;
    n->other_parent_index = p2;
    n->state_before_mutation = n->state_after_mutation =
        children->data[child_index].data[split_point / 3];
}

conditional_inline void do_trace_inheritance (
    unsigned int child_index, unsigned int parent_index,
    unsigned int generation, population *children, simulation_parameters *sp) {
    // Fills in the history for an individual based on the fact that normal
    // inheritance occurred.
    if (sp->o.full_tracing)
        // Trace the individual. If this is the first generation, then every
        // individual came from individual 0.
        for (unsigned int i = 0; i < children->data[child_index].length; i++) {
            individual_history_node *n = sp->i.history->data[generation].data[child_index].data + i;
            n->parent_index = parent_index;
            n->recombination_nucleotide = 0;
            n->other_parent_index = 0xfffffffful;
            n->state_before_mutation = n->state_after_mutation =
                children->data[child_index].data[i];
        }
}

conditional_inline void perform_inheritance (
    population *parents, population *children,
    unsigned int child_index, unsigned int generation,
    fitness_vector *parent_fitness, simulation_parameters *sp) {
    // Inherits from either one or two parents, as appropriate.
    if (normalized_random () < sp->g.recombination_rate) {
        // Perform a recombination. This means that we choose two distinct parents
        // at random.
        unsigned int parent_1 = choose_individual (parent_fitness, parents);
        unsigned int parent_2 = parent_1;

        // In theory, this never has to be false; however, in practice we can
        // choose one or two and it should almost always happen.
        while (parent_2 == parent_1)
            parent_2 = choose_individual (parent_fitness, parents);

        unsigned int split_point = (int)
            (normalized_random () * parents->data[parent_1].length * 3);

        recombine_sequences (parents->data + parent_1,
            parents->data + parent_2,
            children->data + child_index,
            split_point);

        do_trace_recombination (split_point, generation, child_index, parent_1,
            parent_2, parents, children, sp);
    } else {
        unsigned int parent_index = choose_individual (parent_fitness, parents);

```

```

    copy_sequence (parents->data + parent_index, children->data + child_index);
    do_trace_inheritance (child_index, parent_index, generation, children, sp);
}
}

conditional_inline void choose_parents (
    population *parents, population *children,
    simulation_parameters *sp, unsigned int generation) {

    // Run through the population and recombine where necessary, performing simple
    // redirection where possible. There may be places where time could be saved
    // by pointer shuffling rather than data copies, but I'm not seeing them yet.
    fitness_vector *resulting_vector = sp->i.parent_fitness_vector;

    // First, compute the fitness vector for the parents.
    if (sp->g.selection_model != NEUTRAL_SELECTION)
        populate_parent_fitness (parents, sp);
    else if (sp->g.drift_model == NEUTRAL_DRIFT)
        resulting_vector = NULL; // A complete neutral model, so no biasing.
    else
        // No selection, so assume neutral fitness.
        for (unsigned int i = 0; i < parents->length; i++)
            sp->i.parent_fitness_vector->data[i] = (double) (i + 1);

    // If using diversifying selection, precompute the cached amino acid
    // frequencies.
    if (sp->g.selection_model == DIVERSIFYING_SELECTION)
        cache_diversifying_selection (parents, sp);

    unsigned int children_to_simulate = children->length;

    if (sp->g.drift_model == EVEN_SUBDIVISION_DRIFT)
        resulting_vector = sp->i.result_fitness_vector;
    else if (sp->g.drift_model == SHARP_BOTTLENECK_DRIFT) {
        sp->i.parent_fitness_vector->length = (generation < sp->m.bottleneck.start ||
            generation > sp->m.bottleneck.start +
            sp->m.bottleneck.duration) ?
            parents->length :
            sp->m.bottleneck.size;

        // At the ending generation, we simulate the whole child population.
        // This way, we repopulate from the still smaller parent fitness
        // vector without incorporating bottlenecked individuals.
        children_to_simulate = (generation < sp->m.bottleneck.start ||
            generation >= sp->m.bottleneck.start +
            sp->m.bottleneck.duration) ?
            parents->length :
            sp->m.bottleneck.size;
    }

    for (unsigned int i = 0; i < children_to_simulate; i++) {
        consider_drift_on_fitness (parents, generation, i, sp);
        perform_inheritance (parents, children, i, generation, resulting_vector, sp);

        // Now mutate the child sequence.
        mutate_sequence (children->data + i, sp, generation, i);
    }
}
}

```

## 6.4 Runtime Functions

The runtime requirements are fairly straightforward. Primarily, we need to instantiate a set of simulation parameters (which involves defining the appropriate functions for selection and drift) and allocate and then free two popu-

lations. Then we need to perform statistical analysis on the results and print those to `stdout`.

### 6.4.1 Printing

Sequences and statistical output need to be formatted as CSV data. We do this with two functions, `print_sequences` and `print_d_values`. Also, progress needs to be printed as the simulation runs. We achieve this with the `print_trial_indicator` function, which prints both to standard output and standard error. For more information about sequence forms, see section 3.5.1.

```
conditional_inline void print_sequences (population *p, FILE *f,
                                       simulation_parameters *sp, bool packed) {
    // Prints all of the sequences in a population, optionally in packed form.

    for (unsigned int i = 0; i < sp->g.sample_size; i++) {
        if (! packed)
            printf ("Sequence_%d_of_%d:,%", i + 1, sp->g.sample_size);

        print_sequence (p->data + i, f, packed);
        printf ("\n");
    }
}

void print_d_values (population *p, simulation_parameters *sp) {
    // Since the population is completely shuffled, we take the first n
    // individuals. The simplest way to do this is to cheat a little bit: We don't
    // actually need to allocate any new memory. Instead, we just modify the
    // length attribute of the population and then change it back.

    unsigned int old_length = p->length;
    p->length = sp->g.sample_size;

    compute_variance_coefficients (p, sp, &sp->i.s);
    compute_theta (p, sp, &sp->i.s);
    compute_pi (p, sp, &sp->i.s);
    compute_d (p, sp, &sp->i.s);

    // Print a reasonable set of statistics.
    printf ("%f,%f,%", sp->i.s.d_syn, sp->i.s.d_non);
    printf ("%f,%f,%f,%f,%",
            sp->i.s.pi_syn, sp->i.s.pi_non, sp->i.s.theta_syn, sp->i.s.theta_non);
    printf ("%f,%f,%", sp->i.s.s_syn, sp->i.s.s_non);

    printf ("%d,%d,%d,%",
            sp->i.s.dng_size[0], sp->i.s.dng_size[1], sp->i.s.dng_size[2]);
    printf ("%f,%f,%f,%",
            sp->i.s.dng_data[0], sp->i.s.dng_data[1], sp->i.s.dng_data[2]);
    printf ("%d,%d,%d",
            sp->i.s.dng_multiplifitons[0], sp->i.s.dng_multiplifitons[1],
            sp->i.s.dng_multiplifitons[2]);

    if (sp->o.full_tracing) {
        compute_estimator_accuracy (p, sp, &sp->i.s);

        printf (",");

        // Print out additional data about the accuracy of each estimator used.
        printf ("%f,%f,%", sp->i.s.gl_non_bias, sp->i.s.gl_non_var);
        printf ("%f,%f,%f,%f,%",
                sp->i.s.theta_site_seq_bias, sp->i.s.theta_site_seq_var,
                sp->i.s.theta_site_non_bias, sp->i.s.theta_site_non_var);
    }
}
```

```

    printf ("%f, %f, %f, %f",
           sp->i.s.random_seg_bias, sp->i.s.random_seg_var,
           sp->i.s.random_non_bias, sp->i.s.random_non_var);
}

printf ("\n");

p->length = old_length;
}

inline void print_trial_indicator (unsigned int current_trial, simulation_parameters *sp) {
    fprintf (stderr, "Image_%s:_Trial_%d_of_%d\n",
            sp->image_name, current_trial, sp->o.trials);
}

```

## 6.4.2 Execution

We combine all of the runtime code into the `run_trials` method. Given a set of simulation parameters, it will print the results of each trial as it goes. One of the reasons we let `run_trials` run multiple trials is that it is faster to use previously allocated memory than it is to allocate new memory for each trial. Since a trial runs in less than a second, it may make some difference, especially when many trials are executed.

Sometimes we run only the simulation or the statistics, so I've broken up the code into separate functions to handle these cases.

```

void simulate_trial (simulation_parameters *sp, population *parents, population *children) {
    population *temporary = NULL;

    // Performs the simulation of a single trial.
    for (unsigned int i = 0; i < sp->g.population_size; i++)
        copy_sequence (sp->g.ancestral_sequence, parents->data + i);

    for (unsigned int generation = 0; generation < sp->g.generations; generation++) {
        choose_parents (parents, children, sp, generation);

        // Swap the parents and children out now.
        temporary = parents;
        parents = children;
        children = temporary;

        if (sp->o.print_each_generation)
            print_d_values (parents, sp);
    }
}

conditional_inline void read_trial (simulation_parameters *sp, population *parents, population *children) {
    // Reads a trial's worth of data from stdin into the parent population.
    char s[65536];
    for (unsigned int i = 0; i < sp->g.population_size; i++) {
        scanf ("%65535s", s);
        read_sequence (s, parents->data + i);
    }
}

void allocate_internal_memory (simulation_parameters *sp) {
    // Handles the allocation of any simulation temporary space.
    if (! sp->p.statistics_only) {
        sp->i.parent_fitness_vector = create_fitness_vector ();
        sp->i.child_fitness_vector = create_fitness_vector ();
        sp->i.result_fitness_vector = create_fitness_vector ();
    }
}

```

```

    ensure_fitness_vector (sp->i.parent_fitness_vector, sp->g.population_size);
    ensure_fitness_vector (sp->i.child_fitness_vector, sp->g.population_size);
    ensure_fitness_vector (sp->i.result_fitness_vector, sp->g.population_size);
}

ensure_diversifying_aa_freq (sp->i.diversifying_cache, sp->g.ancestral_sequence->length);

if (sp->o.full_tracing) {
    sp->i.history = create_population_history ();
    ensure_population_history (sp->i.history, sp->g.generations);
    for (unsigned int i = 0; i < sp->g.generations; i++) {
        ensure_generation_history (sp->i.history->data + i, sp->g.population_size);
        for (unsigned int j = 0; j < sp->g.population_size; j++)
            ensure_individual_history (sp->i.history->data[i].data + j, sp->g.ancestral_sequence->length);
    }
}

void make_profile_if_necessary (simulation_parameters *sp) {
    // Builds a profile on stdout that will finish the process.
    if (sp->o.make_profile)
        printf ("begin-comment\n"
            "This_profile_was_automatically_generated_using_the_make-profile\n"
            "directive._When_run,_this_profile_will_perform_statistical\n"
            "analysis_on_the_populations_generated_by_the_simulation_run.\n"
            "The_simulation_run_used_random_seed_%d_and_had_image_name\n"
            "'%s'.\n"
            "end-comment\n\n"
            "trials:%d\n"
            "population-size:%d\n" // Deliberately set to sample size.
            "statistics-only\n"
            "end-of-options\n",
            sp->i.random_seed, sp->image_name, sp->o.trials, sp->g.sample_size);
}

void run_trials (simulation_parameters *sp) {
    population *parents = create_population ();
    population *children = create_population ();

    ensure_population (parents, sp->g.population_size);
    ensure_population (children, sp->g.population_size);

    allocate_internal_memory (sp);
    make_profile_if_necessary (sp);

    seed_rng (sp->i.random_seed);

    for (unsigned int t = 0; t < sp->o.trials; t++) {
        print_trial_indicator (t + 1, sp);

        // We need trial data one way or another. Depending on whether we're
        // supposed to read in existing data or generate new data, we choose
        // the function to call.
        if (sp->p.statistics_only)
            read_trial (sp, parents, children);
        else
            simulate_trial (sp, parents, children);

        // There are two reasons we might print sequences. The first is if it's a
        // normal run and we want to verify things in the Excel checker. The second
        // is if it's a simulation-only run and we want to archive the results for
        // later analysis. Since the Excel checker accepts sequences only in
        // nucleotide form, those are output unpacked. However, for archival
        // purposes we pack the sequences to save on storage space and read/write
        // time.
        if (sp->o.print_sequences)
            print_sequences (parents, stdout, sp, FALSE);
        else if (sp->p.simulation_only) {

```

```

    print_sequences (parents, stdout, sp, TRUE);
    printf ("\n");
}

// Now select a subset and print out D values.
if (! sp->p.simulation_only)
    print_d_values (parents, sp);
}

delete_population (parents);
delete_population (children);

free (parents); parents = NULL;
free (children); children = NULL;
}

```

## 6.5 Front-end Code

The profile files described in section 2.1 are parsed and then a sanity check is run on the settings to ensure that nothing too outlandish is being requested.

### 6.5.1 Profile parser

Parsing the profiles is a single-pass process that is primarily driven by short-circuit logic. We make extensive use of `scanf` to grab words, integers, and doubles, and wrap it using delegate functions. Each delegate function returns nonzero if the directive was successfully read and zero otherwise.

```

unsigned int int_directive (FILE *f, const char *current, const char *key,
                          unsigned int *value)
{return ! strcmp (current, key) && fscanf (f, "%u", value);}

unsigned int double_directive (FILE *f, const char *current, const char *key,
                              double *value)
{return ! strcmp (current, key) && fscanf (f, "%lf", value);}

unsigned int sequence_directive (FILE *f, char *current, const char *key,
                                sequence **value)
{return ! strcmp (current, key) && fscanf (f, "%65535s", current) &&
        free_return_true (*value) && (*value = read_sequence (current, NULL));}

unsigned int boolean_directive (FILE *f, const char *current, const char *key,
                               bool *value)
{return ! strcmp (current, key) && (*value = TRUE);}

unsigned int special_directives (FILE *f, char *current,
                                simulation_parameters *sp) {
    // Take care of keyword directives such as selection and drift.
    return ((! strcmp (current, "selection-model:") && fscanf (f, "%65535s", current) &&
              ((! strcmp (current, "neutral") && ! (sp->g.selection_model = NEUTRAL_SELECTION)) ||
               (! strcmp (current, "purifying") && (sp->g.selection_model = PURIFYING_SELECTION)) ||
               (! strcmp (current, "diversifying") && (sp->g.selection_model = DIVERSIFYING_SELECTION)))) ||

            ((! strcmp (current, "drift-model:") && fscanf (f, "%65535s", current) &&
              ((! strcmp (current, "neutral") && ! (sp->g.drift_model = NEUTRAL_DRIFT)) ||
               (! strcmp (current, "sharp-bottleneck") && (sp->g.drift_model = SHARP_BOTTLENECK_DRIFT)) ||
               (! strcmp (current, "even-subdivision") && (sp->g.drift_model = EVEN_SUBDIVISION_DRIFT))))));
}

unsigned int parse_options (simulation_parameters *sp, FILE *f) {
    char s[65536];
}

```

```

while (! feof (f)) {
    // Read the next whitespace-delimited word and check for known keywords.
    fscanf (f, "%65535s", s);
    if (! strcmp (s, "begin-comment")) {
        while (strcmp (s, "end-comment"))
            fscanf (f, "%65535s", s);

        s[0] = '\0'; // This is to clear the existing memory in case
                    // the read below hits an EOF.

        // Load the next keyword if it exists.
        fscanf (f, "%65535s", s);
    }

    int end = ! strcmp (s, "end-of-options");

    if (s[0] && ! end && !
        (int_directive (f, s, "trials:", &sp->o.trials) ||
         int_directive (f, s, "population-size:", &sp->g.population_size) ||
         int_directive (f, s, "generations:", &sp->g.generations) ||
         int_directive (f, s, "sample-size:", &sp->g.sample_size) ||
         double_directive (f, s, "frequency-a:", &sp->g.mutation.frequency_a) ||
         double_directive (f, s, "frequency-c:", &sp->g.mutation.frequency_c) ||
         double_directive (f, s, "frequency-g:", &sp->g.mutation.frequency_g) ||
         double_directive (f, s, "frequency-u:", &sp->g.mutation.frequency_u) ||
         double_directive (f, s, "purifying-selection-coefficient:",
                          &sp->m.purifying.selection_coefficient) ||
         sequence_directive (f, s, "purifying-optimal-sequence:",
                          &sp->m.purifying.optimal_sequence) ||
         double_directive (f, s, "diversifying-selection-coefficient:",
                          &sp->m.diversifying.selection_coefficient) ||
         int_directive (f, s, "sharp-bottleneck-start:", &sp->m.bottleneck.start) ||
         int_directive (f, s, "sharp-bottleneck-duration:", &sp->m.bottleneck.duration) ||
         int_directive (f, s, "sharp-bottleneck-size:", &sp->m.bottleneck.size) ||
         int_directive (f, s, "even-subdivision-start:", &sp->m.subdivision.start) ||
         int_directive (f, s, "even-subdivision-duration:", &sp->m.subdivision.duration) ||
         int_directive (f, s, "even-subdivision-size:", &sp->m.subdivision.size) ||
         (int_directive (f, s, "random-seed:", &sp->i.random_seed) &&
          fprintf (stderr, "Warning:_Specifying_internal_parameter_random_seed\n")) ||
         (double_directive (f, s, "assumed-gl-variance:", &sp->i.assumed_gl_variance) &&
          fprintf (stderr, "Warning:_Specifying_internal_parameter_assumed-gl-variance\n")) ||
         double_directive (f, s, "mutations-per-nucleotide:",
                          &sp->g.mutation.mutations_per_nucleotide) ||
         double_directive (f, s, "transitions-per-transversion:",
                          &sp->g.mutation.transitions_per_transversion) ||
         double_directive (f, s, "recombination-rate:", &sp->g.recombination_rate) ||
         boolean_directive (f, s, "print-sequences", &sp->o.print_sequences) ||
         boolean_directive (f, s, "print-each-generation", &sp->o.print_each_generation) ||
         boolean_directive (f, s, "full-tracing", &sp->o.full_tracing) ||
         (! sp->p.statistics_only &&
          boolean_directive (f, s, "simulation-only", &sp->p.simulation_only)) ||
         (! sp->p.simulation_only &&
          boolean_directive (f, s, "statistics-only", &sp->p.statistics_only)) ||
         (sp->p.simulation_only &&
          boolean_directive (f, s, "make-profile", &sp->o.make_profile)) ||
         (sp->o.full_tracing &&
          boolean_directive (f, s, "trace-multiplitons-only",
                          &sp->o.trace_multiplitons_only)) ||
         sequence_directive (f, s, "ancestral-sequence:", &sp->g.ancestral_sequence) ||
         special_directives (f, s, sp)) {
        fprintf (stderr, "Error:_Unrecognized_directive_or_invalid_value_for_'s'\n", s);
        return 1;
    } else if (end)
        return 0;

    // Make sure the buffer is empty before the next read. This is to ensure
    // that if an EOF is hit after an empty read, the last word read is not

```

```

    // interpreted as a directive.
    s[0] = '\0';
}

return 0;
}

```

## 6.5.2 Profile sanity check

To prevent common errors, we provide a sanity check of the profile data. This involves making sure that all relevant sequences are the same length, that no quantitative parameters are out of bounds, etc.

```

unsigned int sanity_check (simulation_parameters *sp) {
    // Sequence lengths. Make sure that the purifying optimal sequence and the
    // ancestral sequence have the same length.
    if (sp->g.ancestral_sequence->length != sp->m.purifying.optimal_sequence->length &&
        sp->g.selection_model == PURIFYING_SELECTION) {
        fprintf (stderr, "Error:_Ancestral_sequence_and_optimal_sequence_are_of_"
                 "different_lengths.\n");
        return 1;
    }

    // Check to make sure that the statistical sample size is no larger than the
    // population size.
    if (sp->g.sample_size > sp->g.population_size) {
        fprintf (stderr, "Error:_The_statistical_sample_size_is_larger_than_the_"
                 "population_size.\n");
        return 1;
    }

    // If we're using population bottlenecks, then do a series of sanity checks to
    // make sure that the bottleneck doesn't extend past the end of the
    // generations, etc.
    if (sp->g.drift_model == SHARP_BOTTLENECK_DRIFT) {
        if (sp->m.bottleneck.start > sp->g.generations) {
            fprintf (stderr, "Error:_The_sharp_bottleneck_begins_after_the_simulation_"
                     "ends.\n");
            return 1;
        }

        if (sp->m.bottleneck.start + sp->m.bottleneck.duration > sp->g.generations)
            fprintf (stderr, "Warning:_The_sharp_bottleneck_extends_past_the_end_of_the_simulation.\n");

        if (sp->m.bottleneck.size > sp->g.population_size) {
            fprintf (stderr, "Error:_The_bottleneck_size_is_larger_than_the_total_"
                     "population_size.\n");
            return 1;
        }
    }

    // Same for even subdivision.
    if (sp->g.drift_model == EVEN_SUBDIVISION_DRIFT) {
        if (sp->m.subdivision.start > sp->g.generations) {
            fprintf (stderr, "Error:_The_even_subdivision_begins_after_the_simulation_"
                     "ends.\n");
            return 1;
        }

        if (sp->m.subdivision.start + sp->m.subdivision.duration > sp->g.generations)
            fprintf (stderr, "Warning:_The_even_subdivision_extends_past_the_end_of_the_simulation.\n");

        if (sp->m.subdivision.size > sp->g.population_size) {
            fprintf (stderr, "Error:_The_even_subdivision_size_is_larger_than_the_total_"

```

```

        "population_size.\n");
    return 1;
}
}

// If print_each_generation is set and more than one trial is occurring,
// notify the user that they might have a difficult time separating out the
// runs. This isn't an error, just an advisory.
if (sp->o.print_each_generation && sp->o.trials > 1)
    fprintf (stderr, "Warning: Statistics are being printed at each generation,
                    but more than one trial is being run. This is perfectly
                    legal, but it may be difficult to separate the data once
                    the simulation is finished. (Since multiple trials will
                    be difficult to tell apart.)\n");

// If the selection coefficient is negative or greater than 1, then issue a
// warning indicating that the user has deviated from biological realism.
if (sp->m.purifying.selection_coefficient < 0.0 ||
    sp->m.purifying.selection_coefficient > 1.0) {
    fprintf (stderr, "Error: Selection coefficients not between 0 and 1
                    are not biologically realistic.\n");

    return 1;
}

return 0;
}

```

### 6.5.3 Main method delegation

This main method can be called by the actual main method to provide all of the default functionality of the model. All memory is freed and command options are parsed from stdin. The first command-line argument becomes the image name of this instance.

```

int main_delegate (int argc, char** argv) {
    simulation_parameters sp;

    // Initialize sensible defaults and then parse options.
    initialize_simulation_parameters (&sp);

    if (parse_options (&sp, stdin)) {
        fprintf (stderr, "Error: At least one option was not parsed correctly.\n");
        return EXIT_FAILURE;
    }

    if (sanity_check (&sp)) {
        fprintf (stderr, "Error: At least one parameter had an invalid value.\n");
        return EXIT_FAILURE;
    }

    fprintf (stderr, "Using random seed %d\n", sp.i.random_seed);
    sp.image_name = argv[1];

    // Run the trials.
    run_trials (&sp);
    delete_simulation_parameters (&sp);
    return EXIT_SUCCESS;
}

```

## 7 Closing

This section isn't a proper conclusion so much as a separate section for the necessary `#endif` directive to match the `#ifndef __MODEL_H` at the beginning of the file. In any case, this `#endif` should remain after any code.

```
#endif
```

---