

MathBio 2008 Evolution Model

Spencer Tipping

November 28, 2009

Abstract

This program is designed to generate populations and then run statistical analysis on the results. Output is to a file specified on the command line. This code was designed to be compiled by the Stalin Scheme compiler, available under the GPL. There is no warranty, express or implied, provided with this code, nor any guarantee of merchantability or fitness for any purpose.

1 Introduction

The purpose of this project is to determine the correlation between Tajima's D statistic and the selection model used.

1.1 Notation

I use a convention of two-letter prefixes to avoid name collision and to keep function names short. Table 1 lists the prefixes and their meanings.

```
#include "scheme-compatibility.sc.in"
```

1.2 Global Assumptions

A sequence is 35 codons long. This number should *not* be used except to create a new sequence. In general, we assume that sequences are of an arbitrary length.

```
(define sequence-length 35)
```

2 Model Framework

2.1 Generally Useful Functions

Integer division is not built into R4RS Scheme. However, it can easily be simulated using flooring, inexact conversion, and normal division:

mt	Mersenne Twister random generator
co	Codon manipulation function
se	Sequence manipulation function
fi	Fitness or selection function
si	Simulation driver function
bm	Benchmark function
ng	Nei-Gojobori statistical function
cl	Combinatorial statistical function
dp	Discrete pathway analysis function
cp	Continuous pathway analysis function
ge	Gaussian elimination function
gs	General statistical function
pi	π statistic calculation
th	θ statistic calculation

Figure 1: Naming conventions in the model source code

```
(define (idiv a b)
  (inexact->exact (floor (/ a b))))
```

2.2 Iteration Extensions

To facilitate simple iteration, we introduce some new functions that make expression of looping constructs very straightforward. Following Ruby's convention, we first introduce the "times" function, which takes a λ -closure expecting a single parameter, the loop index.

```
(define (times n fn)
  ;; Calls fn n times, each time providing the loop index as the
  ;; parameter to fn.
  (do ((i 0 (+ i 1)))
      ((>= i n))
      (fn i)))
```

For convenience and, depending on the nature of the implementation, for optimization, we use a customized vector iterator. One of the things we do frequently in the code is iterate over all of the elements of a vector. To abstract away the details, we use the `each` function. Note that the vector length is expected to be invariant in the loop (as would generally be expected, but in this case it is crucial).

```
(define (each v fn)
  ;; Invokes fn on each element and index in v. Returns v.
  (times (vector-length v) (lambda (i)
    (fn (vector-ref v i) i)))
  v)

(define (each-rev v fn)
  ;; Works like each, but goes backwards.
  (do ((i (- (vector-length v) 1) (- i 1)))
      ((negative? i) v))
```

```
(fn (vector-ref v i) i)))
```

It would also be nice to have an iterator to do pairwise comparisons for the π -calculation. It uses a half-quadratic unique pairwise system.

```
(define (half-pairwise v fn)
  ;; An iterator that calls fn on the set of all unique pairs.
  ;; fn is invoked with four arguments: v1, v2, i1, and i2,
  ;; where vx are values and ix are indices.
  (let ((l (vector-length v)))
    (times (- l 1) (lambda (i)
      (times (- l 1 i) (lambda (j)
        (fn (vector-ref v i) (vector-ref v (+ j i 1))
            i (+ j i 1))))))))))
```

2.3 Vector Functions

In addition to iteration, sometimes simple functions on vectors are needed. The most common ones are retrieving the first and last elements.

```
(define (vector-first v) (vector-ref v 0))
(define (vector-last v) (vector-ref v (- (vector-length v) 1)))
```

Circular access to a vector is sometimes handy.

```
(define (vector-cref v i)
  (vector-ref v (modulo i (vector-length v))))
```

We also could use a quick way to copy or fill a vector:

```
(define (vector-fill! destination source)
  ;; Arguments ordered similarly to set! and other assignment
  ;; operators. Returns the destination vector.
  (times (min (vector-length destination)
              (vector-length source))
    (lambda (i)
      (vector-set! destination i (vector-ref source i))))
  destination)
```

2.4 Matrix Functions

Often we need to create a two-dimensional matrix. A simple way to do this is to invoke this function:

```
(define (matrix rows cols initial)
  ;; Returns a matrix of the specified dimensions filled
  ;; with /initial/.
  (let ((result (make-vector rows)))
    (each result (lambda (v i)
      (vector-set! result i (make-vector cols initial))))))
```

Also, to avoid long strings of `vector-ref` and `vector-set!`, we define matrix accessors:

```
(define (mref m row col) (vector-ref (vector-ref m row) col))
(define (mset! m row col val)
  (vector-set! (vector-ref m row) col val))
```

In addition, it can be useful to have an iterator for matrices. It uses row-major ordering.

```
(define (matrix-each m fn)
  ;; Returns the matrix m.
  (each m (lambda (v i)
    (each v (lambda (u j)
      (fn u i j))))))
```

2.5 Random Number Generator

We use the MT19937 algorithm to generate pseudorandom numbers. The period of this generator is $2^{19937} - 1$. This code was translated from the algorithm on Wikipedia's *Mersenne Twister* article.

```
(define last-31-bit-mask 2147483647) ; 0x7fffffff
(define last-32-bit-mask 4294967295) ; 0xffffffff
(define mt:entropy-length 624)
(define (mt:initialize seed . entropy-array)
  ;; Seed should be a 32-bit number. This function returns an
  ;; entropy array, which may be provided as the optional second
  ;; argument.
  (let ((result (if (and (not (null? entropy-array))
    (vector? (car entropy-array)))
    (car entropy-array)
    (make-vector mt:entropy-length))))
    (vector-set! result 0 seed)
    (each result (lambda (v i)
      (if (not (zero? i))
        (let ((last (vector-ref result (- i 1))))
          (vector-set! result i
            (+ i (* 1812433523 (^ last (>> last 30))))))))))
```

The algorithm to produce random numbers is fairly straightforward, but it relies on there being more numbers available in the entropy array. Thus we have a number generator that automatically maintains the entropy array by refilling it when it becomes empty. For simplicity's sake, the entropy array set is a simple vector $\langle idx, E \rangle$, where idx is the next element from the entropy array E to be used.

```
(define (mt:create-set)
  (vector 0 (make-vector mt:entropy-length 0)))

(define (mt:random entropy-array-set)
  (let* ((idx (vector-ref entropy-array-set 0))
    (a (vector-ref entropy-array-set 1)))
    (if (>= idx (vector-length a))
      (begin
        (mt:initialize (vector-last a) a)
        (set! idx 0))
      (vector-set! entropy-array-set 0 (+ 1 idx))
      (let* ((y0 (vector-ref a idx))
        (y1 (^ y0 (>> y0 11)))
        (y2 (^ y1 (& (<< y1 7) 2636928640)))
        (y3 (^ y2 (& (<< y2 15) 4022730752)))
        (y4 (^ y3 (>> y3 18)))
        (& y4 last-32-bit-mask))))
```

Finally, we need to create floating-point numbers as well and we should have a convenient, self-contained number generator. The first may be achieved

simply by dividing by the maximum value produced (in this case, $2^{32} - 1$). The second is possible using a λ -closure. Note that for generating floats, we have to jump through some hoops to avoid integer overflow situations. We treat the number as 31 bits, since signed arithmetic is used. Then we convert that to a floating-point value and divide by the maximum that it could be.

```
(define (mt:random-float entropy-array-set)
  (/ (exact->inexact (& last-31-bit-mask
                    (mt:random entropy-array-set)))
     (exact->inexact last-31-bit-mask)))

(define (mt:create-generator
        entropy-array-set . random-function)
  ;; If random-function is specified, then use it instead of
  ;; mt:random.
  (if (null? random-function)
      (lambda () (mt:random entropy-array-set))
      (let ((f (car random-function)))
        ;; We bind f outside of the lambda so that the invocation
        ;; of car is not repeated on a per-call basis.
        (lambda () (f entropy-array-set)))))
```

For convenience, we go ahead and define a default global random number generator and seed.

```
(define global-entropy-array-set (mt:create-set))
(define global-mt-int-generator (mt:create-generator
                                global-entropy-array-set))
(define global-mt-real-generator (mt:create-generator
                                  global-entropy-array-set
                                  mt:random-float))
```

2.6 Statistical Functions

The statistical portion of the code requires some generic math functions that are not provided by Scheme. In particular, we need a fast implementation of a combination function.

```
(define (choose k n)
  ;; Returns n choose k.
  (let ((result n))
    ;; Do a manual partial product here. If we did full
    ;; factorials, we might invoke big-integer handling
    ;; (provided that the Scheme implementation supported
    ;; those), which would be slow.
    (do ((i (- n 1) (- i 1)))
        ((<= i k))
      (set! result (* result i)))

    ;; Now we have the partial factorial. We need to divide by k!.
    (do ((i 1 (+ i 1)))
        ((>= i k) result)
      (set! result (inexact->exact (/ result i))))))
```

We end up doing a lot of things with a given probability, and a fast way to do this is to be able to say (with-probability x (lambda () (...))).

```
(define (event-occurred? p)
  ;; Returns true with probability p.
  (>= p (global-mt-real-generator)))
```

```
(define (with-probability p f)
  ;; Invokes f with probability p.
  (if (event-occurred? p)
      (f)))
```

We also need a binary search. One of the more frequent operations that we perform is randomly selecting an element from a non-uniform distribution. Generally this would involve a linear search, but if we compute the distribution's CDF it becomes $O(\log n)$.

The CDF vector is assumed to be right-handed; that is, for a PDF of $\langle \frac{1}{2}, \frac{1}{2} \rangle$, the CDF should be $\langle \frac{1}{2}, 1 \rangle$, not $\langle 0, \frac{1}{2} \rangle$. This is because a CDF does not necessarily stay inside $[0, 1]$; it may take an arbitrarily large value. Since we know that 0 is the left-hand extreme, we need to have the right-hand extreme encoded into the vector.

```
(define (select-member cdf-vector target)
  (let* ((size (vector-length cdf-vector))
        (location (>> size 1))
        (jump (>> size 2)))
    (do ()
        ((or (>= location size)
             (negative? location)
             (<= (vector-ref cdf-vector location)
                 target)
             (vector-ref cdf-vector (+ location 1))))
      (+ 1 location)) ; Return the location at the end.

    ;; At this point, we know that we have not hit the target,
    ;; so we must be either too far left or right. We can
    ;; easily check left:
    (if (and (>= location 0)
            (< target (vector-ref cdf-vector location)))
        (set! location (- location jump))
        (set! location (+ location jump)))
    ;; Also halve the jump distance per iteration.
    (set! jump (o 1 (>> jump 1))))))
```

2.7 Amino Acid Tables

The table of amino acids needs to exist in two forms. First, each amino acid must be assigned a number. Second, we need a vector that lets us look up amino acids by the index of the six-bit codon (see next section).

```
(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 amino-acid-lookup
  (vector 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))

```

2.8 Codons

We represent codons using bytes. This should be extremely efficient from a memory and implementation standpoint. Each nucleotide fits in two bits, so we define those numeric constants here. Also, since we are working with effectively a base-4 number system, we define the multipliers appropriately.

```

(define co:u 0)
(define co:c 1)
(define co:a 2)
(define co:g 3)

(define nucleotide-mask 3)
(define nucleotide-multiplier 4)
(define codon-multiplier (expt nucleotide-multiplier 3))

```

The `co:create` function creates a byte with `b3` in the most-significant position and `b1` in the least-significant. The two most significant bits are unused. `co:get` retrieves the packed nucleotides created by `co:create`.

```

(define (co:create b1 b2 b3)
  (o (<< b3 4) (<< b2 2) b1))

(define (co:get c n-index)
  (let ((shift-amount (* n-index 2)))
    (>> (& c (<< nucleotide-mask shift-amount))
         shift-amount)))

```

We also need functions to alter nucleotides (inside codons). There are two types of alterations that can occur. Transitions are changes between A and G or C and U, and transversions are everything else. In this model, transitions correspond to flipping the less significant bit, and transversions are either flipping the more significant bit or both.

```
(define (co:create-transformer mask)
  (lambda (c n-index)
    (^ c (<< mask (* n-index 2)))))

(define co:transition (co:create-transformer 1))
(define co:transversion-1 (co:create-transformer 2))
(define co:transversion-2 (co:create-transformer 3))
```

Codons have some properties that we need to be able to find. First, do they map to the same amino acid (i.e. are they synonymous)? Second, how many nucleotides differ between the two? Which nucleotides?

```
(define (co:syn? c1 c2)
  (= (vector-ref amino-acid-lookup c1)
     (vector-ref amino-acid-lookup c2)))
```

The algorithm for computing the number of changed nucleotides looks peculiar, but is more efficient than looping. The logic is as follows: If you have a codon $C = 00aabbcc$, where aa , bb , and cc are the bits for the nucleotides, then $00aabbcc \wedge 00101010$ gives the high-order bits of each. Similarly, $00aabbcc \wedge 00010101$ gives their low-order bits. Since at least one bit of a nucleotide must have changed for the nucleotide to differ, we can slide the high-order bits over the low-order bits with a single-right-shift and then `or` them together to get a mask of which nucleotides differ. Finally, we can take this number modulo 3 and that will give us the number of differences mod 3.

```
(define (co:mask c1 c2)
  (let ((cdiff (^ c1 c2)))
    (o (>> (& cdiff 42) 1)
       (& cdiff 21))))

(define (co:position c1 c2)
  ;; Returns the nucleotide index at which c1 and c2 differ. (Assuming
  ;; that there is only one difference between them.)
  (case (co:mask c1 c2)
    ((1) 2)
    ((4) 1)
    ((16) 0)))

(define (co:count c1 c2)
  (modulo (co:mask c1 c2) 3))
```

During pathway analysis we will need to change a codon one nucleotide at a time. Since this could occur quite frequently, an optimized version of that function is defined here (note that 63 is 00111111 in binary):

```
(define (co:inherit c1 c2 nucleotide-index)
  ;; Returns c1 with one nucleotide coming from
  ;; nucleotide-index.
  (let ((change-mask (<< nucleotide-mask
                          (<< nucleotide-index 1))))
    (o (& c1 (^ 63 change-mask))
       (& c2 change-mask))))
```

2.9 Codon Sets

We sometimes need a 64-element set to observe codons. An extremely efficient way to implement this is via two 32-bit integers. Since we work with Scheme

distributions that support full 32-bit integers, the algorithm is feasible.

```
(define (co:create-set) (make-vector 2 0))
(define (int-index i)
  ;; Returns the index of the above vector to access given
  ;; the index of the element in the set.
  (if (> i 31) 1 0))

(define (co:add-to-set! i v)
  ;; Performs the addition in-place.
  (let ((idx (int-index i)))
    (vector-set! v idx
      (o (vector-ref v idx) (<< 1 (& i 31))))))
```

We also need to be able to count the number of elements in a set. To do this, we use a bit-counting algorithm that is extremely efficient. The idea derives from group theory: We retrieve the values of all bits whose significance is 1 modulo 63, then those whose significance is 2 modulo 63, etc.

These bits in binary form masks:

```
01000001000001000001000001000001 = 1090785345
10000010000010000010000010000010 = 2181570690
00000100000100000100000100000100 = 68174084
00001000001000001000001000001000 = 136348168
00010000010000010000010000010000 = 272696336
00100000100000100000100000100000 = 545392672
```

For efficiency's sake, we may defer the modulus operation until after the addition.

Two important things to note:

- The `modulo-63` function is important here. We use it because Stalin's 32-bit integers are signed, and the `modulo` function will not correctly compute the value of an unsigned modulus.
- We must defer the `and` operation on the mask with a 1 in the highest bit position because Stalin performs arithmetic right-shifts and all 32-bit quantities are signed. So we first shift the quantity and then take the previous mask, effectively removing the sign.

```
(define (count-ones-in-32-bits v)
  (modulo-63 (+
    (& v 1090785345)
    (& (>> v 1) 1090785345)
    (>> (& v 68174084) 2)
    (>> (& v 136348168) 3)
    (>> (& v 272696336) 4)
    (>> (& v 545392672) 5))))

(define (co:count-set v)
  (+ (count-ones-in-32-bits (vector-ref v 0))
    (count-ones-in-32-bits (vector-ref v 1))))
```

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:

$$100100\dots000$$

This indicates that codons 0 and 3 are in the set. To determine whether the two least-significant bits are the same, we take four bitmasks:

$$\begin{aligned} 0001\dots &= 286331153 \\ 0010\dots &= 572662306 \\ 0100\dots &= 1145324612 \\ 1000\dots &= 2290649224 \end{aligned}$$

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:

$$\begin{aligned} 000000000001111\dots &= 983055 \\ 000000011110000\dots &= 15728880 \\ 000011110000000\dots &= 251662080 \\ 111100000000000\dots &= 4026593280 \end{aligned}$$

The same criterion may be reused. 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 two bitmasks for this:

$$\begin{aligned} 1^{16}0^{16} &= 4294901760 \\ 0^{16}1^{16} &= 65535 \end{aligned}$$

In this case, since no shifting is occurring, we don't have to worry about further `and`-masking to correct for the arithmetic shift.

The `co:singleton-set?` function returns `#f` if the set does not contain a singleton change, and it returns the index of the nucleotide that differs if it is a singleton change.

```
(define (co:three-are-zero? v1 v2 v3 v4)
  ;; Returns true iff at least three of the
  ;; four values are zero.
  (let ((z1 (zero? v1))
        (z2 (zero? v2)))
```

```

(z3 (zero? v3))
(z4 (zero? v4)))
(or (and z1 z2 z3)
    (and z1 z2 z4)
    (and z1 z3 z4)
    (and z2 z3 z4)))

(define (co:singleton-set? v)
  (let* ((v1 (vector-ref v 0))
         (v2 (vector-ref v 1))
         (v12 (o v1 v2))
         (n31 (& v12 286331153))
         (n32 (& v12 572662306))
         (n33 (& v12 1145324612))
         (n34 (& v12 2290649224))
         (n21 (& v12 983055))
         (n22 (& v12 15728880))
         (n23 (& v12 251662080))
         (n24 (& v12 4026593280))
         (z3 (co:three-are-zero?
              n31 n32 n33 n34))
         (z2 (co:three-are-zero?
              n21 n22 n23 n24)))
    ;; Both z2 and z3 nonzero indicates that
    ;; the second and third nucleotides both
    ;; differ, so we can short-circuit out
    ;; of the and.
    (and (or z3 z2)
         (let* ((n11 (& v1 4294901760))
                (n12 (& v1 65535))
                (n13 (& v2 4294901760))
                (n14 (& v2 65535))
                (z1 (co:three-are-zero?
                    n11 n12 n13 n14)))
           ;; z2 and z3 are not both false.
           ;; If either is true, then z1 must
           ;; be false. The formula for implication
           ;; simplifies to what is below.
           (and (or (and z2 z3) z1)
                (cond
                 ((not z1) 0)
                 ((not z2) 1)
                 ((not z3) 2)))))))

```

Sometimes we need to unpack a set into its members. We do this by returning a list. The algorithm below is not very efficient, but it should work.

```

(define (co:unpack s)
  (let ((result '()))
    (sc (vector (vector-ref s 0) (vector-ref s 1)))
    ;; See what leaves the set unchanged.
    (times 64 (lambda (i)
                (let ((count (co:count-set sc)))
                  (co:add-to-set! i sc)
                  (if (= count (co:count-set sc))
                      ;; It was in the set.
                      (set! result (cons i result))))))
    result))

```

Finally, we must be able to determine whether the codons in the set are synonymous. I'm sure there's a very cool number-theoretic way to do this, but it would involve so much special-casing that it's easier to just unpack the set into its two codons and check their synonymy.

```

(define (co:synonymous-set? s)

```

```

;; Assumes that the set contains exactly two elements.
(let* ((r (co:unpack s))
      (c1 (car r))
      (c2 (cadr r)))
  (co:syn? c1 c2))

```

2.10 Sequences

We refer to a fixed-length collection of codons as a sequence. Internally, they are implemented as Scheme vectors of codons. We'll need to do a few management-oriented things with them:

```

(define (se:create . args)
  (let ((result (make-vector sequence-length 0)))
    ;; If an existing sequence was provided, then use that.
    (if (null? args)
        result
        (vector-fill! result (car args)))))

```

A common situation that arises is recombination. When offspring are created, their parents are mixed together, often splitting inside of a codon. Thus we need to specifically handle the case where the split location is not aligned on a codon boundary.

```

(define (se:recombine
        s1 s2 codon-index nucleotide-index . destination)
  (let* ((result (if (null? destination)
                    (se:create)
                    (car destination)))
        (c1 (vector-ref s1 codon-index))
        (c2 (vector-ref s2 codon-index))
        (split-codon (case nucleotide-index
                      ((0) c1)
                      ((1) (co:create (co:get c1 2)
                                       (co:get c1 1)
                                       (co:get c2 0))))
                      ((2) (co:create (co:get c1 2)
                                       (co:get c2 1)
                                       (co:get c2 0)))))
    (times (vector-length s2) (lambda (i)
                               (vector-set! result i
                                             (cond
                                              ((< i codon-index) (vector-ref s1 i))
                                              ((= i codon-index) split-codon)
                                              (else (vector-ref s2 i)))))))
  result))

```

Finally, we need to consider individual fitness. As of yet, we don't have the functions to actually compute fitness, but we delegate that to a lambda passed to this function. This function returns a PDF-vector of the fitness of each member of the population.

`fitness-evaluator` takes the form `(lambda (p i) (...))`, where `p` is the population and `i` is the index of the individual in question. We need to allow the function to analyze the whole population because in some cases an individual's similarities or differences from its peers determine its viability.

```

(define (fi:fitness
        population fitness-evaluator make-cdf? . destination)

```

```

(let* ((result (if (null? destination)
                  (make-vector (vector-length population)
                              (car destination))
                  (total 0.0))
      (each result (lambda (v i)
                    (if make-cdf?
                        (begin
                          (set! total (+ total (fitness-evaluator population i)))
                          (vector-set! result i total))
                          (vector-set! result i
                                      (fitness-evaluator population i)))))))

(define (fi:pdf-to-cdf v)
  ;; Converts a PDF vector into a CDF vector in-place,
  ;; and returns the vector.
  (let ((total (vector-ref v 0)))
    (each v (lambda (v0 i)
              (if (positive? i)
                  (begin
                    (set! total (+ total v0))
                    (vector-set! v i total)))))))

(define (fi:cdf-to-pdf v)
  ;; Reverses the effect of fi:pdf-to-cdf.
  (each-rev v (lambda (v0 i)
                (if (not (zero? i))
                    (vector-set! v i (- v0 (vector-ref v (- i 1)))))))

```

3 Simulation Driver

The simulation-side of the code generates a final population. This population is then sampled for statistical analysis.

3.1 Initialization

```

(define (si:create template size)
  ;; Returns /size/ copies of /template/ as a vector
  ;; of sequences.
  (let ((result (make-vector size)))
    (each result (lambda (v i)
                  (vector-set! result i (se:create template))))))

```

3.2 Mutation and Recombination

Recombination and mutation each occur probabilistically and independently of each other. When we recombine, we take two parents at random and mix them into a new child. There are certain probabilities associated with each event. These should be externally managed for recombination.

```

(define (si:mutate-codon
        c nucleotide-index p-transition p-transversion-type-1)
  ;; Returns the modified codon. Takes two probabilities:
  ;; The probability of a transition and that of a type-1
  ;; transversion (as opposed to a type-2).
  (if (event-occurred? p-transition)
      (co:transition c nucleotide-index)
      (if (event-occurred? p-transversion-type-1)
          (co:transversion c nucleotide-index p-transversion-type-1)
          c)))

```

```
(co:transversion-1 c nucleotide-index)
(co:transversion-2 c nucleotide-index)))
```

Sequence mutation is performed per-nucleotide. The old Perl code used a per-sequence calculation, however here we do a per-nucleotide calculation to ensure that multiple simultaneous mutations are accounted for.

Note: This code can easily be optimized. By using conditional probability, we can isolate regions with no changes. The probability that no changes occur in an entire replication for an individual is on the order of $(1 - 1 \times 10^{-4})^{105} \approx 99\%$, so in only one out of 100 cases do we need to iterate through. However, we then need to figure in the conditional probability. Currently this is not implemented.

```
(define (si:mutate-sequence
  s p-mutation p-transition p-transversion-type-1 site-p)
  ;; Performs mutations along the sequence probabilistically.
  ;; site-p should be a PDF of site mutation
  ;; probability multipliers normalized at 1. (Per nucleotide,
  ;; not per codon.)
  (each s (lambda (v codon-index)
    (times 3 (lambda (nucleotide-index)
      (if (event-occurred? (* p-mutation
        (vector-ref site-p
          (+ (* 3 codon-index)
            nucleotide-index))))
        (vector-set! s codon-index
          (si:mutate-codon v nucleotide-index
            p-transition p-transversion-type-1))))))))))
```

Since `se:recombine` has already been written above, we don't need to worry about it here. The `si:mutate-sequence` function should be invoked on every new sequence, since it performs multiple-site mutation when appropriate and internally applies the mutation probability.

3.3 Population Subdivision & Bottlenecking

A population can be temporarily or permanently partitioned into various groups, and this effect can occur with varying intensity. One place where this could occur in nature is when two groups become isolated. The degree of isolation may be partial or complete.

We model subdivision in this simulation by a crossover matrix whose entries are the probabilities that a given child i will inherit from parent j . For our purposes, we assume that i governs the major axis and j the minor axis. We also implement the matrix as a function of two variables, since as the population grows, the space complexity increases at $O(n^2)$. (Note that the i and j notation is consistent with the nomenclature of the function definition below.)

Note that for consistency's sake, a population crossover matrix A must be identical to A' . Thus, the function must reflect this relationship in order to be mathematically consistent.

The way this fits into the model is fairly simple: We take the individual fitnesses of the sequences in the population and multiply each by the corre-

α	Probability of mutation per base pair per individual per generation
β	Probability of recombination per individual per generation
τ_0	Probability of transition given a mutation
τ_1	Probability of transversion type-1 given a transversion

Figure 2: Coefficients for trial runs

sponding entry in the matrix. Since normally probabilities are in PDF-form, this is a simple operation to perform.¹

Bottlenecks are also possible using this generalized notion of isolation. If the matrix has a row of zeroes, then the individual corresponding to that row has no possibility of reproduction. Thus, that individual is not considered for that generation. By isolating the mixing to an upper-left square, the effective population is reduced.

```
(define (si:select-parent
  i parent-fitness-vector matrix-function target)
  ;; The target parameter is a random number between 0 and 1.
  ;; This function returns the index of the chosen parent.
  (if (null? matrix-function)
      ;; If no matrix function, then assume that the fitness vector
      ;; is in CDF form and do a quick search.
      (select-member parent-fitness-vector
        (* target (vector-last parent-fitness-vector)))

      ;; Otherwise, build a CDF (we assume that the fitness
      ;; vector is in PDF form).
      (let ((temp-cdf (vector-copy parent-fitness-vector))
            (total 0.0))
        (each parent-fitness-vector (lambda (v j)
          (set! total (+ total (* v (matrix-function i j))))
          (vector-set! temp-cdf j total)))
        ;; Multiply the unscaled target to the proportion of
        ;; the CDF.
        (select-member temp-cdf (* target total)))))
```

3.4 Automation

Here we define procedures to automate the execution of a trial. Notice the curried function definition for the matrix function in particular. We do this so that the matrix may change depending on the generation.

There are several coefficients used (see table 2).

To avoid an excessively long parameter list, we pass these in as a vector. The vector is ordered this way: $\langle \alpha, \beta, \tau_0, \tau_1 \rangle$.

The `site-mutation-probabilities` parameter determines whether individual nucleotides are more or less likely to mutate. It is normalized at 1 and must be supplied. If in doubt, just use `(make-vector 105 1.0)`.

¹Currently, the implementation uses a binary search that is possible only for a CDF. This reduces the algorithmic complexity of choosing a parent to $O(\log n)$. However, taking a product of two vectors is linear, as is conversion to or from a CDF, so it may be more feasible to do a simple linear search instead.

A slight oddity here: If the matrix function is null, then we can apply an optimization by just building the CDF and reducing a linear search to a binary search for fitness calculations. This should speed up the model by about 25 times for a population size of 500.

```
(define (si:generation
  parent-vector child-vector p-vector matrix-function
  fitness-function site-mutation-probabilities)
  (let ((alpha (vector-ref p-vector 0))
        (beta (vector-ref p-vector 1))
        (tau-0 (vector-ref p-vector 2))
        (tau-1 (vector-ref p-vector 3))
        (pv (fi:fitness parent-vector fitness-function
            (null? matrix-function))))
    ;; Populate the child generation from randomly chosen parents.
    (each child-vector (lambda (v i)
      (if (event-occurred? beta)
        ;; A recombination occurs
        (se:recombine
          (vector-ref parent-vector
            (si:select-parent i pv matrix-function
              (global-mt-real-generator)))
          (vector-ref parent-vector
            (si:select-parent i pv matrix-function
              (global-mt-real-generator)))
          (modulo (global-mt-int-generator) (vector-length v))
          (modulo (global-mt-int-generator) 3)
          v)
        ;; A recombination doesn't occur, so just select a parent at random.
        (vector-fill! (vector-ref child-vector i)
          (vector-ref parent-vector
            (si:select-parent i pv matrix-function
              (global-mt-real-generator))))))
    ;; Perform obligatory sequence mutation independently of the origin of the
    ;; child.
    (si:mutate-sequence v alpha tau-0 tau-1 site-mutation-probabilities))))
```

The matrix function m should be curried such that $m(g)$, where g is the number of the current generation, returns a function $m_0 : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{R}$, which gives the matrix value $m_0(i, j)$ as described in the previous section. The same currying is applied to the fitness function f . $f(g)$ should return a function $f_0(p, i)$, which in turn returns the viability of individual i in population p .

Trial parameters are arranged in a vector $\langle |P|, g \rangle$, where P is the initial population (thus $|P|$ is the size of the initial population) and g is the number of generations to run.

A trial, formally speaking, consists of a number of generations to be run. Specifically, we create an initial population and a child population, run that for some number of generations, and then return the final child population. This definition of a trial does not include statistical analysis; that will be covered in the next section.

The following code will run a trial under the neutral model and return the ending population:

```
(si:trial (vector 500 1000) ; 500 individuals, 1000 generations
  (vector 0.0025 0.0315 0.56521 0.5)
  (lambda (g) '())
  (lambda (g) (lambda (p i) 1.0))
```

```
(make-vector 105 1.0)
(make-vector sequence-length 0))
```

```
(define (si:trial
  trial-parameters p-vector matrix-function
  fitness-function site-mutation-probabilities
  original-sequence)
  (let* ((p-size (vector-ref trial-parameters 0))
        (generations (vector-ref trial-parameters 1))
        (parent-population (si:create original-sequence p-size))
        (child-population (si:create original-sequence p-size)))
    (times generations (lambda (i)
      (si:generation parent-population child-population p-vector
        (matrix-function i) (fitness-function i)
        site-mutation-probabilities)
      ;; Convert the previous child population to the
      ;; next set of parents.
      (each child-population (lambda (v j)
        (vector-fill! (vector-ref parent-population j) v))))
      child-population))
```

For simplicity, we include some shortcut functions specialized for the pre-defined models. First, however, we must define the original sequence and a sequence reader function:

```
(define (se:read-sequence l)
  (let ((len (length l)))
    (let loop ((result (make-vector (idiv len 3) 0))
              (current l)
              (frame 0))
      (let ((codon-position (idiv frame 3)))
        (if (< frame len)
            (begin
              (vector-set! result codon-position
                (o (vector-ref result codon-position)
                  (<< (case (car current) ((u) co:u)
                                         ((c) co:c)
                                         ((a) co:a)
                                         ((g) co:g)
                                         (<< (modulo frame 3) 1))))
                (loop result (cdr current) (+ 1 frame)))
            result))))))

(define (se:original)
  '(u g u a c a a g a c c c a a c a a c a a u a c a a u a
    a a a a g u a u a c a u a u g g g a c u a g g g a g g
    a c a u u u u a u a c a a c a g g a g a a g u a a u a
    g g a g a u a u a a g a c a a g c a c a u u g u))
```

The neutral model takes no parameters. It simply runs and produces output.

```
(define si:neutral-matrix (matrix 500 500 1.0))

(define (si:hiv-trial-neutral)
  (si:trial (vector 500 1000) ; 500 individuals, 1000 generations
    (vector 0.00034 0.00034 0.56521 0.5)
    (lambda (g) (lambda (i j) 1.0))
    (lambda (g) (lambda (p i) 1.0))
    (make-vector 105 1.0)
    (se:read-sequence (se:original))))
```

4 Benchmarks

This section will need major revision, since the benchmarks now must run through the statistical methods as well as invoking the simulation.

To facilitate refinement and test hypotheses about statistical methods, it is important to be able to measure their effectiveness. There are several tests that could be used for this purpose, but for now we assume a simple percentage of correctness. This step constitutes a generalization of the problem to be adaptable to various statistical estimators, not just Tajima's D or modifications thereof. The significant benefit of doing things this way is that we can have a very good idea of the accuracy of a statistical method as it applies to inferring the simulation parameters used.

In practice, benchmarking the statistical estimators will provide an excellent way to determine whether or not a compromise may be made. For instance, we currently do not know how much difference it will make to assume finite sites instead of infinite sites. This assumption would require a lot of computational overhead if done properly, but the benchmarks might indicate that the difference between the two assumptions does not affect the accuracy of the model whatsoever; in that case, we could choose to continue using infinite sites for speed reasons.

4.1 Discrete Estimator Specifications

In order to be benchmarked, an estimator must conform to a particular specification. Due to the way these are specified, often an estimator will end up being an anonymous function, since the format is largely dependent on the application in question.

An estimator function e is one that maps a population p to a parameter vector v . The tuple of available parameters is denoted η , the corresponding model runners are κ , and a distance metric $\delta(\hat{v}, v)$ is defined that determines the distance between an estimated \hat{v} and the actual v . So, for example, let's take the simple case of just determining whether a given selection model was directional or not:

$$\begin{aligned}\eta &= \langle \text{directional}, \text{neutral} \rangle \\ \kappa &= \langle \lambda : () \rightarrow p, \lambda : () \rightarrow p \rangle \\ \delta(\hat{v}, v) &= \text{dim}(v) - (\hat{v} \cdot v) \\ e(p) &= \begin{cases} \langle \text{directional} \rangle & \pi(p) > \theta(p) \\ \langle \text{neutral} \rangle & \pi(p) < \theta(p) \end{cases}\end{aligned}$$

Here, we have η defined to include directional and neutral simulation, κ loaded with random variables that return populations generated by those simulation models, and an estimator $e(p)$ that attempts to determine the model used.

4.2 Continuous Estimator Specifications

4.3 Benchmarking Framework

Running a discrete benchmark involves simply summing all of the δ -based distances between the estimated and actual results for a trial.

```
(define (bm:run-discrete-benchmark eta kappa delta e n)
  ;; Performs benchmark estimation given the parameters
  ;; as outlined above. The additional parameter n governs
  ;; the number of trials that are executed. Since the
  ;; estimator is stateless, we simply iterate through
  ;; eta and kappa without worrying about randomizing per
  ;; iteration. Returns the total distance of estimates
  ;; from actual values.
  (let ((distance 0.0))
    (times n (lambda (i)
      ;; Notice the invocation of the function
      ;; (vector-cref kappa i) by double-paren
      ;; enclosure.
      (set! distance (+ distance (delta
        (e ((vector-cref kappa i)) (vector-cref eta i))))))
      distance)))
```

5 Tajima's D Statistic

Analysis of the results that we generate is crucial to this project. We use two main statistical measures that together comprise Tajima's D statistic. The first is a pairwise comparison between the sample sequences (π), in which we look at the average variance between sites. The second is a whole-stack comparison between sites (θ), where we simply figure out the number of sites that vary. In each case, we separate the differences into synonymous and nonsynonymous.

We can use several different statistical methods, depending on the type of input that we have. Those are:

- Nei-Gojobori
- Combinatorial
- Discrete pathways
- Continuous pathways

In practice, we can unify some code. Combinatorial-style data is all of the same form (that is, indexed by nucleotide and then codon), so we can abstract it into a common "linear" estimator form:

```
(define (gs:linear-nonsynonymity c1 c2 initial-data)
  (if (co:syn? c1 c2)
      0.0
      ;; Otherwise, check along each nucleotide.
      (let ((result-1 0.0)
            (result-2 0.0))
        (diff (co:mask c1 c2)))
        (if (not (zero? (& diff 1)))
```

```

(begin
  (set! result-1 (+ result-1 (mref initial-data 2 c1)))
  (set! result-2 (+ result-2 (mref initial-data 2 c2))))
(if (not (zero? (& diff 4)))
  (begin
    (set! result-1 (+ result-1 (mref initial-data 1 c1)))
    (set! result-2 (+ result-2 (mref initial-data 1 c2))))
  (if (not (zero? (& diff 16)))
    (begin
      (set! result-1 (+ result-1 (mref initial-data 0 c1)))
      (set! result-2 (+ result-2 (mref initial-data 0 c2))))
    (/ (+ result-1 result-2) 2.0))))

```

This system may be used with the combinatorial and discrete pathways estimators. For discrete pathways, the data is in a square matrix M , where $M_{i,j}$ is the nonsynonymity estimate of an observed change from codon i to codon j . The same is true of continuous pathways. That function is defined this way:

```

(define (gs:matrix-nonsynonymity c1 c2 initial-data)
  (/ (+ (mref initial-data c1 c2) (mref initial-data c2 c1)) 2.0))

```

Note the calculation of an average. In an ideal world, we would not have to do this, but since it is possible to have directional preference (e.g. A mutates to G more readily than the reverse), we must explicitly remove all assumptions of directionality by averaging both directions. In most calculations, however, it will not be an issue.

Finally, we provide encapsulation so that the initial data may be elided from individual invocations:

```

(define (gs:create-estimator fn initial-data)
  (lambda (c1 c2)
    (fn c1 c2 initial-data)))

```

5.1 π Calculation

The π calculation returns the average number of codon differences in a pairwise context. So for three sequences $AAATTT$, $AAATTA$, and $AAAATT$, $\binom{3}{2}$ comparisons are made:

$AAATTT \leftrightarrow AAATTA$: 1 difference; $AAATTT \leftrightarrow AAAATT$: 1 difference; $AAATTA \leftrightarrow AAAATT$: 2 differences. 4 total differences and 3 comparisons: $\pi = \frac{4}{3}$.

Computation becomes more subtle when we attempt to separate synonymous and nonsynonymous. We use the estimators documented below to find the expected nonsynonymity of each site, and take that out of the total number of differences to find the synonymity. These factors together contribute to π_{SYN} and π_{NON} , irrespectively.

`pi:syn-factor` returns the expected number of synonymous differences between two codons. This is computed by taking the total number of differing nucleotides and subtracting the expected number of nonsynonymous changes.

```

(define (pi:syn-factor c1 c2 non-factor)
  (if (= c1 c2)
    0.0

```

```
(let* ((n-diffs (co:count c1 c2))
      (result (- (exact->inexact n-diffs) non-factor)))
  (if (zero? n-diffs)
      (+ result 3.0) ; Since three codon differences is reported
      result))) ; as zero, we need to compensate.
```

`pi:pair` returns a vector $\langle s, n \rangle$, where s is the total contribution from synonymous factors and n is the contribution from nonsynonymous factors.

```
(define (pi:pair s1 s2 estimator)
  (let ((syn 0.0)
        (non 0.0))
    (each s1 (lambda (c1 i)
              (let* ((c2 (vector-ref s2 i))
                    (non-factor (estimator c1 c2)))
                (set! non (+ non non-factor))
                (set! syn (+ syn (pi:syn-factor c1 c2 non-factor))))))
      (vector syn non)))
```

Now we need to use the half-pairwise iterator and get a total. This is reported as a vector $\langle s, n \rangle$. We compute $\frac{(n-2)(n-1)}{2}$ as a quick computation of the number of comparisons performed, as that is $\sum_{i=1}^{n-1} i$.

```
(define (pi:totals population estimator)
  (let* ((total-syn 0.0)
        (total-non 0.0)
        (l (vector-length population))
        (comparisons (* (- l 2) (- l 1) 0.5)))
    (half-pairwise population (lambda (s1 s2 i1 i2)
                              (let ((immediate (pi:pair s1 s2 estimator)))
                                (set! total-syn (+ total-syn (vector-ref immediate 0)))
                                (set! total-non (+ total-non (vector-ref immediate 1))))))
      (vector (/ total-syn comparisons)
              (/ total-non comparisons))))
```

5.2 θ Calculation

To calculate θ , we first need Nei-Gojobori data. If this is insufficient or not present, then we use combinatorial analysis. In practice, this may mean using a weighted estimator function that takes a biased average of Nei-Gojobori and combinatorial data, depending on the sample size of Nei-Gojobori singleton observations.

A few notes about the way θ is calculated in the old system (primarily for my own edification):

1. Nei-Gojobori ratios are figured for a total of three positions, those being the positions within a codon. Also, Nei-Gojobori ratios do not depend at all on the value of the codon; they are solely positional in nature.
2. When using the combinatorial method (i.e. Nei-Gojobori data is insufficient or not present), since there are multiple codons involved, the average is taken between them so that each side is equally considered. This is because we cannot assume directionality of change.

In the new system, we abstract this selection to a function that automatically selects Nei-Gojobori data or combinatorial data depending on the quantity. This function is the estimator, which, given a column of codons in the form of a population and an index, returns the estimated nonsynonymity of the column.

A potential enhancement in the new system is to perform a weighted average of Nei-Gojobori data and combinatorial data depending on how much Nei-Gojobori data there is. Currently, this is not implemented.

```
(define (th:totals population estimator)
  0.0
)
```

5.3 Nei-Gojobori

The Nei-Gojobori method takes the sample and finds the likelihood that a change in any given nucleotide is nonsynonymous. The reasoning is that the sample as a whole will give accurate ratios of these occurrences, and by multiplying observed occurrences by these average ratios we will obtain a close estimate.

A note about Nei-Gojobori calculation: Only singleton ratios are considered. So for any given codon position, we consider the set of distinct values. Provided that this set has exactly two elements and the two elements differ by a single nucleotide, we then determine whether the change was synonymous or nonsynonymous and add it to the appropriate counter.

For efficiency's sake, we break out of the calculation as soon as we find more than two distinct codons, since we need only singletons. Since the algorithm for counting the number of elements in a codon set is very efficient, this probably saves time.²

This function returns a vector $\langle s_0, s_1, s_2, n_0, n_1, n_2 \rangle$, where s_i is the synonymous ratio of nucleotide position i , and n_i is the number of singleton columns that s_i is based on.

Note that this initial data function assumes that all members of the sample are the same length. Considering the current state of the simulation, this is a very safe assumption, but it will need to be changed if this condition ever changes in the future.

```
(define (ng:initial-data sample)
  (let ((total-syn (make-vector 3 0))
        (total-non (make-vector 3 0))
        (result (make-vector 6 0)))
    (each (vector-ref sample 0) (lambda (c ci) ; ci for codon index
      (let ((observed-codons (co:create-set)))
        (call/cc (lambda (break)
          (each sample (lambda (v i)
            ;; Add the observed entry to the value set.
            (co:add-to-set! (vector-ref v ci) observed-codons)
            (break))
          ))
      (total-syn (vector-set! total-syn ci (+ (vector-ref total-syn ci) 1))
      (total-non (vector-set! total-non ci (+ (vector-ref total-non ci) 1))
      (result (vector-set! result ci (+ (vector-ref result ci) 1))
    ))
  ))
```

²If this use of `call/cc` does not make sense, then just think of the `(break)` line as meaning the same thing as a `break` in Java or C, except that it breaks out of the `each` loop. The `call/cc` and `lambda` are plumbing to make this work.

```

      (if (> (co:count-set observed-codons) 2)
          (break))))))
;; Now check to see whether the codons differ by just one nucleotide.
(let ((d (co:singleton-set? observed-codons)))
    (if (number? d)
        (if (co:synonymous-set? observed-codons)
            (vector-set! total-syn d
                (+ 1 (vector-ref total-syn d)))
            (vector-set! total-non d
                (+ 1 (vector-ref total-non d))))))
    (each total-syn (lambda (v i)
        (let ((total (+ v (vector-ref total-non i))))
            (vector-set! result i
                (if (zero? total)
                    0.0
                    (/ v (+ v (vector-ref total-non i))))))
            (vector-set! result (+ i 3) (+ v (vector-ref total-non i))))))
        result))

```

5.4 Combinatorial

Combinatorial data is designed to be used with a linear estimator, since it is indexed by nucleotide and then by codon. Since combinatorial data is based solely on the genetic code, no sample or population is needed to initialize it.

```

(define (cl:initial-data)
  (let ((result (matrix 3 64 0.0)))
    (each result (lambda (v i)
        (each v (lambda (v0 j)
            ;; Count the number of synonymous and nonsynonymous neighbors
            ;; that we get by varying the given nucleotide.
            (let ((non-total 0))
                (times 4 (lambda (k)
                    ;; There are four possible values for the modified position
                    ;; to take.
                    (let ((new-codon (co:inherit j (co:create k k k) i)))
                        (if (not (co:syn? j new-codon))
                            (set! non-total (+ 1 non-total))))))
                    ;; 3 because we have three neighbors that are different.
                    (vector-set! v j (/ non-total 3.0))))))
            result))

```

5.5 Discrete Pathways

In discrete pathway analysis, we look at all of the different mutation paths that a codon could have taken. This assumes infinite sites, since each site is assumed to have changed at most once.

This function takes a callback function, which is invoked with each single-difference. So, for instance, if we wanted to do a discrete pathway analysis of the change $AAA \rightarrow AGG$, then the callback would be invoked four times: (AAA, AAG) , (AAG, AGG) , (AAA, AGA) , and (AGA, AGG) . This reflects the fact that we considered all orderings of changes. At the end, the pathways function returns the number of invocations of the callback function.

```

(define (dp:pathways c1 c2 callback)
  ;; Returns the total number of invocations of the callback function.
  (let ((invocations 0))
    (times 3 (lambda (i)

```

```

    (if (not (= (co:get c1 i) (co:get c2 i)))
        (let ((new-codon (co:inherit c1 c2 i)))
            (callback c1 new-codon)
            (set! invocations (+ 1 invocations
                                (dp:pathways new-codon c2 callback))))))
    invocations))

```

To build the initial matrix data, we just look at the total numbers of synonymous and nonsynonymous changes that could have happened.

```

(define (dp:initial-data)
  (let ((result (matrix 64 64 0.0)))
    (each result (lambda (v i)
      (each result (lambda (v0 j)
        (if (= i j)
            ;; There are, of course, no changes between a codon and itself.
            (vector-set! v j 0.0)

            ;; However, other codons are fair game.
            (let* ((non-total 0)
                  (total (dp:pathways i j
                                       (lambda (c1 c2)
                                         (if (not (co:syn? c1 c2))
                                             (set! non-total (+ non-total 1)))))))
              (vector-set! v j (/ non-total total))))))))))

```

5.6 Continuous Pathways

Continuous pathways assumes a finite sites model, so we will end up traversing an infinite tree. To do this efficiently, we solve 64 systems of 64 linear equations. These solutions make up our matrix of codon-to-codon nonsynonymity values.

This operation as a whole is $O(n^4)$ in the number of possible values that a codon can take, however until major changes in biological thought take place, n is a constant 64. Thus this operation, for all practical purposes, runs in constant time (though it is a large constant).

5.6.1 Gaussian Elimination

To perform Gaussian elimination without pivoting, we simply normalize the top row so that the leading coefficient is 1 and add scalar multiples of it to each other row to zero their leading coefficients. Then we do the same with the second row and its leading coefficient until we arrive at the last row. Upon reaching the last row, we then work our way back up, first zeroing all of the coefficients above the only one in the last row, then using the second-to-last row on the ones above it, etc. Essentially, we perform an in-place matrix inversion with the augmented matrix $[M|V]$.

The first operation we need to support is a scaled addition. We assume that the matrix is stored in row-major order and that it is properly rectangular.

```

(define (ge:add-scaled m s d l)
  ;; Adds the multiple l[ambda] of row s[source] to row d[estimation].
  (each (vector-ref m s) (lambda (v i)
    (mset! m d i (+ (mref m d i) (* l v))))))

```

The next piece of logic is a constant multiple of a row. Specifically, we would want to normalize it. The function returns the factor by which the row was multiplied so that the column vector entry can be multiplied by the same factor.

```
(define (ge:normalize-row r i)
  ;; Normalizes the leading nonzero coefficient of r to 1 and returns
  ;; the multiplicative normalization factor. i is the index of the
  ;; leading coefficient.
  (let ((scale-factor (/ 1.0 (vector-ref r i))))
    (each r (lambda (v j)
              (vector-set! r i (* v scale-factor))))
    scale-factor))
```

We need to do pivoting, so we need to first find the largest row and then swap it with the given one. Note that we take the absolute value because the sign doesn't matter for these operations. A cached copy of the largest-so-far value is kept around to minimize out-of-cache hits, which are potentially expensive for this type of calculation.

```
(define (ge:largest-row-index m col)
  (let ((largest-so-far 0.0)
        (largest-index 0))
    (each m (lambda (r i)
              (if (> (abs (vector-ref r col)) largest-so-far)
                  (begin
                     (set! largest-so-far (abs (vector-ref r col)))
                     (set! largest-index i))))
            largest-index))

(define (ge:swap-rows m v r1 r2)
  ;; Swaps the rows in a matrix and an auxiliary vector.
  (let ((tmp-row (vector-ref m r1))
        (tmp-val (vector-ref v r1)))
    (vector-set! m r1 (vector-ref m r2))
    (vector-set! v r1 (vector-ref v r2))
    (vector-set! m r2 tmp-row)
    (vector-set! v r2 tmp-val)))
```

Finally, the Gaussian elimination logic itself:

```
(define (ge:eliminate m v)
  ;; Reduces m to the identity matrix and leaves the solution in v.
  ;; v is expected to be a column vector and m must be in row-major
  ;; order.
  (each m (lambda (r i)
            ;; First, pivot the largest row into place.
            (ge:swap-rows m v i (ge:largest-row-index m i))
            ;; Normalize the row and the entry in the column vector.
            (vector-set! v i (* (vector-ref v i)
                                (ge:normalize-row r i)))
            ;; Add its inverse to each other row.
            (each m (lambda (r0 j)
                      (if (not (= i j))
                          (begin
                             (vector-set! v j (- (vector-ref v j)
                                                    (* (vector-ref v i) (vector-ref r0 i))))
                             (ge:add-scaled m i j (- (vector-ref r0 i))))))))))
```

5.6.2 Methods

(To be written...)