The Source Code for SYREN: A Connectionist model for Syllable Recognition

By

Erich J. Smythe
Department of Computer Science
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 252

The Source Code for SYREN: A Connectionist model for Syllable Recognition

by

Erich J. Smythe

June, 1988

The Source Code for SYREN:
A Connectionist model for Syllable Recognition

Erich J. Smythe
Computer Science Department
Indiana University
Bloomington, Indiana 47405

Abstract

This report contains the source code for SYREN [1], a connectionist network that performs syllable recognition by identifying formant transitions. The code used for all three phases of recognition is included along with some instructions as to how to put each piece together. This report is intended to serve as an appendix to the full description of the system, and will be useful only to those who need the full implementational details of SYREN.

1. Introduction

SYREN is a connectionist model that identifies the rate and direction of formant transitions and learns to associate these transitions with particular stop consonant–vowel syllables [1]. This report contains all the source code that was used in the three programs that make up the system, as well as the parameters used in the various subnetworks, and is intended as an appendix to the full system description [1] for those that wish to use SYREN on other data. All code has been taken directly from the source files themselves.

SYREN is implemented in two languages. The motion detector network is written in Chez Scheme [2], while the adaptive network and veto recognition network are written in C [3]. All experiments were performed on a DEC VAX 8800 running the ULTRIX operating system.

The system first identifies the rate and direction of the formant transitions from the input data. This information is then provided to the adaptive network that sets the weights of its nodes to associate transition data with particular syllables. After the adaptive nodes are trained, their performance is analyzed and a veto recognition network is constructed. This recognition network uses output from the adaptive network to recognize the syllable. In theory, all of these operations could proceed together, although many operations must be repeated. The adaptive process requires considerable computation time and memory space. For efficiency reasons, SYREN actually operates in phases. First, the formant transitions for each syllable repetition are identified and stored in files for each repetition. The adaptive network uses these transition files as input during the learning and testing phases. Due to limitations in space and time, the adaptive node for each syllable is trained separately. The output of the adaptive nodes during testing is stored, and is used to evaluate the learning process. The veto recognition network uses input that is prepared from the output of the adaptive network during the testing phase.

Input to the system should be a bit matrix containing formant center data for each syllable repetition. The rows of the matrix indicate the presence of a formant center at that frequency, and the columns a time slice. The system requires one file for each repetition, with a '1' indicating a formant center at that time and frequency, and a '0' otherwise.

2. Formant Transition Identification Phase

This is the code for the motion detector network. The code is divided into preliminary material and help functions, and the actual network code. Nodes in the network are implemented as vector cells that store their activation values. Since for each type of detector, the connections and weights are uniform, the actual communication of activation is handled in the node functions themselves.

2.1 Preliminary Code

This section contains the preliminary code and help functions used in the network. The first are the iteration macros.

```
(extend-syntax (for by)
   [(for /var /from /to by /incr /body ...)
    (let ([upper /to])
       (do ([/var /from (+ /incr /var)])
           ((> /var upper) #t)
           /body ...))]
   [(for /var /from /to /body ...)
    (let ([upper /to])
       (do ([/var /from (1+ /var)])
           ((> /var upper) #t)
           /body ...))]
   )

(extend-syntax (iterate)
   [(iterate index bound exp ...)
    (for index 0 (1- bound) exp ...)])

(extend-syntax (iterate1)
   [(iterate1 index ubound step exp ...)
    (for index 0 (1- ubound) by step exp ...)
    ])
```

The following functions compute activation values.

```
(define! act-fn
   (lambda (act sum theta)
      (+ (* act (- 1 theta))
         (* sum (if (> sum 0) (- 1 act) act)))))

(define! veto-act-fn
   (lambda (act veto-sum sum thresh theta)
      (veto-fn (- (act-fn act sum theta)
                  (if (> veto-sum thresh) veto-sum 0)))))

(define! veto-fn (lambda (n) (if (>= n 0) n 0)))

(define! sig-fn
```

```
   (lambda (input theta temp)
      (/ 1 (+ 1 (exp (/ (neg (- input theta)) temp))))))

(define! neg (lambda (n) (- 0 n)))

(define! init-vecs
   (lambda ()
      (for-each (lambda (vec) (vector-fill! vec 0))
         *vec-list*)))
```

These functions get the network ready to run. Vectors are given initial values, input and output files are opened, and various parameters are set.

```
(define! *input-port* ())

(define! *output-port* ())

(define! *cycle* 0)

(define! init
   (lambda (infile outfile)
      (when *input-port* (close-input-port *input-port*))
      (set! *input-port* (open-input-file infile))
      (if (eq? outfile '*)
          (set! *output-port* *standard-output*)
          (if outfile
             (begin
                (when (and *output-port*
                           (not (eq? *output-port* *standard-output*)))
                      (close-output-port *output-port*)
                      )
                (delete-file outfile)
                (set! *output-port* (open-output-file outfile)))
             (set! *output-port* ())))
      (init-vecs)
      (set! *cycle* 0)))

(define! run
   (lambda (ticks)
      (iterate i ticks
         (run-net)
         )))

(define! output
   (lambda ()
      (when (and *output-list* *output-port*)
         (for-each
```

```
            (lambda (vec)
               (iterate i (vector-length vec)
                  (fprintf *output-port* "~a "
                     (clean-num (vector-ref vec i))))
               (newline *output-port*))
            *output-list*)
         (newline *output-port*))))

(define! clean-num
   (lambda (n)
      (float (/ (round (* n 1000)) 1000))))

(define! net-ticks 7)  ; number of update cycles per time slice

(define! close-input
   (lambda () (when *input-port* (close-input-port *input-port*))))

(define! close
   (lambda ()
      (close-input-port *input-port*)
      (close-output-port *output-port*)))

(define! mk-vector-list
   (lambda vecs
      (if (null? vecs)
          ()
          (if (< (vector-length (car vecs)) 10)
              (append (vector->list (car vecs))
                 (apply mk-vector-list (cdr vecs)))
              (cons (car vecs) (apply mk-vector-list (cdr vecs)))))))

(define! eol?
   (lambda (port)
      (let ([ch (read-char port)])
         (unread-char ch port)
         (eq? ch #\newline))))

(define! readline
   (lambda (port)
      (recur loop ()
         (if (eol? port)
             ()
             (cons (read port) (loop))))))

(define! vector-store!
   (lambda (vec index val)
      (vector-set! vec index (clean-num val))))

(define! N 200) ; number of frequency packets
```

```
(define! N-1 (- N 1)) ; some time savers
(define! N-2 (- N 2))
(define! N-3 (- N 3))
(define! N-4 (- N 4))
(define! N-5 (- N 5))
(define! N-6 (- N 6))
(define! N-8 (- N 8))
```

2.2 Motion Detector Code

The following code deals more specifically with the nodes in the motion detector network.

The next set of functions deals with the input nodes. The values for the input nodes are set based on the values of a column of the input matrix, with one column per time slice. The variable *input-port* is a pointer to a file containing 0's and 1's separated by spaces.

```
(define! input-vector (make-vector N 0))

(define! run-input
   (lambda ()
      (do ([i 0 (1+ i)])
          ((>= i N) 'input)
          (vector-set! input-vector i (read *input-port*)))))
```

The following code controls the veto nodes.

```
(define! veto-vector (make-vector N 0))

(define! run-veto
   (lambda ()
      (do ([i 0 (1+ i)])
          ((>= i N) 'veto)
          (vector-set! veto-vector i
             (act-fn
                (vector-ref veto-vector i)
                (vector-ref input-vector i)
                theta-v)))))

(define! theta-v .08)
```

This is the code for the rest of the motion detectors. Various node types use different functions to compute their activations. An S-node computes its activation using the soma-fn. A distal

branch node with an input connection and veto connections uses the `dendr-fn`, with veto inhibition determined by a specific `veto-fn` that is passed as an argument. More proximal branch nodes use the `inter-fn` unless they have an additional input connections, in which case they use the `branch fn`.

Each activation function requires the subnetwork vectors and veto inhibition function to be passed as arguments. This allows the funtions to be used for each type of subnetwork. Subnetwork vectors are two–dimensional, with the most distal node the first row, and the S–node the last. The full subnetwork addes the capture node as the last element.

The function that controls the activation updates, the `slope-`*xx*`fn`, must compute the activation of the capture nodes first, then the S-nodes, and then proximal to distal branch nodes, in order to allow the activaion to flow one node per update cycle.

```
(define! dendr-fn
   (lambda (vecs index veto-fn input-wt thresh-v theta)
      (let ([vec (vector-ref vecs index)])
         (iterate i N
            (vector-set! vec i
               (veto-act-fn
                  (vector-ref vec i)
                  (veto-fn i)
                  (* (vector-ref input-vector i) input-wt)
                  thresh-v
                  theta))))))

(define! inter-fn
   (lambda (vecs index weight theta)
      (let ([invec (vector-ref vecs (1- index))]
            [outvec (vector-ref vecs index)])
         (iterate i N
            (vector-set! outvec i
               (act-fn
                  (vector-ref outvec i)
                  (* (vector-ref invec i) weight)
                  theta))))))

(define! soma-fn
   (lambda (vecs index width wt thresh temp)
      (let ([invec (vector-ref vecs (1- index))]
            [outvec (vector-ref vecs index)]
            [end (- N (1- width))])
         (iterate i end
            (vector-set! outvec i
               (sig-fn
                  (let ([send (+ i width)])
                     (recur loop ([n i])
                        (if (< n send)
                            (+ (* (vector-ref invec n) wt)
                               (loop (1+ n)))
```

```
                        0)))
                thresh
                temp))))))

; this is for "precise" slopes.  Skips over some.

(define! somap-fn
   (lambda (vecs index width step wt thresh temp)
      (let ([invec (vector-ref vecs (1- index))]
            [outvec (vector-ref vecs index)]
            [end (- N (1- width))])
         (iterate i end
            (vector-set! outvec i
               (sig-fn
                  (let ([send (+ i width)])
                     (recur loop ([n i])
                        (if (< n send)
                            (+ (* (vector-ref invec n) wt)
                               (loop (+ n step)))
                            0)))
                  thresh
                  temp))))))

(define! branch-fn
   (lambda (vecs index theta in-wt br-wt)
      (let ([outvec (vector-ref vecs index)]
            [invec (vector-ref vecs (1- index))])
         (iterate i N
            (vector-set! outvec i
               (act-fn
                  (vector-ref outvec i)
                  (+ (* (vector-ref invec i) in-wt)
                     (* (vector-ref input-vector i) br-wt))
                  theta))))))

; this "catches" a value from a soma and keeps it around until
; the expansion net grabs it.

(define! catch-fn
   (lambda (vecs index width)   ; index gives the catch vector
      (let ([outvec (vector-ref vecs index)]
            [invec (vector-ref vecs (1- index))]
            [end (- N (1- width))])
         (if (= (mod *cycle* net-ticks) 1)
             (iterate i end
                (vector-set! outvec i (vector-ref invec i)))
             (iterate i end
                (let ([val (vector-ref invec i)])
                   (when (> val (vector-ref outvec i))
```

```
                    (vector-set! outvec i
                       val)))))))))
```

The following code runs the steady-state detector. It is slightly different from the motion detectors, since it merely counts to see if the same frequency is present for four consecutive time slices.

```
(define! slope-0-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)))

(define! run-slope0 ; set output if counter has seen freq 4 times
   (lambda ()
      (let ([count-vec (vector-ref slope-0-vec 0)])
         (iterate i N
            (vector-set! count-vec i
               (if (= (vector-ref input-vector i) 1)
                   (1+ (vector-ref count-vec i))
                   0))
            (vector-set! (vector-ref slope-0-vec 1) i
               (if (> (vector-ref count-vec i) 3) 1 0))))))
```

This code is for a slope of 1 in either direction.

```
(define! slope-1-f-vec          ; for rising transitions
   (vector (make-vector N 0)
           (make-vector N 0)
           (make-vector N 0)
           (make-vector N 0)))

(define! slope-1-b-vec          ; for falling transitions
   (vector (make-vector N 0)
           (make-vector N 0)
           (make-vector N 0)
           (make-vector N 0)))

(define! slope-1-f-veto-fn
   (lambda (i)
      (if (<= i N-6)
          (* (+ (vector-ref veto-vector (1+ i))
                (+ (vector-ref veto-vector (+ i 2))
                   (+ (vector-ref veto-vector (+ i 3))
                      (+ (vector-ref veto-vector (+ i 4))
                         (+ (vector-ref veto-vector (+ i 5))
                            (vector-ref veto-vector (+ i 6)))))))
             slope-1-veto-wt)
```

```
            0)))

    (define! slope-1-b-veto-fn
       (lambda (i)
          (if (>= i 6)
              (* (+ (vector-ref veto-vector (1- i))
                    (+ (vector-ref veto-vector (- i 2))
                       (+ (vector-ref veto-vector (- i 3))
                          (+ (vector-ref veto-vector (- i 4))
                             (+ (vector-ref veto-vector (- i 5))
                                (+ (vector-ref veto-vector (- i 6))))))))
                 slope-1-veto-wt)
              0)))

    (define! slope-1-veto-wt 1)
    (define! slope-1-input-wt .3)
    (define! slope-1-thresh-v .1)
    (define! slope-1-theta-d .05)
    (define! slope-1-inter-wt .4)
    (define! slope-1-inter-theta .08)
    (define! slope-1-soma-wt .5)
    (define! slope-1-soma-thresh .77)
    (define! slope-1-soma-temp .01)
    (define! slope-1-catch-thresh .05)

    (define! run-slope1
       (lambda ()
          (slope-1-fn slope-1-f-vec slope-1-f-veto-fn)
          (slope-1-fn slope-1-b-vec slope-1-b-veto-fn)))

    (define! slope-1-fn
       (lambda (vecs veto-fn)
          (catch-fn vecs 3 4)
          (soma-fn vecs 2 4 slope-1-soma-wt slope-1-soma-thresh slope-1-soma-temp)
          (inter-fn vecs 1 slope-1-inter-wt slope-1-inter-theta)
          (dendr-fn vecs 0 veto-fn slope-1-input-wt slope-1-thresh-v
             slope-1-theta-d)))
```

This is for a slope of 2, i.e. a transition that skips a frequency unit each time slice.

```
    (define! slope-2-f-vec
       (vector
          (make-vector N 0)
          (make-vector N 0)
          (make-vector N 0)
          (make-vector N 0)))
```

```
(define! slope-2-b-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)))
```

slope-2p refers to the precise detector

```
(define! slope-2p-f-soma-vec (make-vector N 0))

(define! slope-2p-b-soma-vec (make-vector N 0))

(define! slope-2p-f-vec
   (vector
      (vector-ref slope-2-f-vec 0)
      (vector-ref slope-2-f-vec 1)
      slope-2p-f-soma-vec
      (make-vector N 0)))

(define! slope-2p-b-vec
   (vector
      (vector-ref slope-2-b-vec 0)
      (vector-ref slope-2-b-vec 1)
      slope-2p-b-soma-vec
      (make-vector N 0)))

(define! slope-2-f-veto-fn
   (lambda (i)
      (if (and (<= i N-6) (> i 0))
          (* (+ (vector-ref veto-vector (+ 4 i))
                (+ (vector-ref veto-vector (+ 5 i))
                   (+ (vector-ref veto-vector (+ 6 i))
                      (+ (vector-ref veto-vector (+ 3 i))
                         (+ (vector-ref veto-vector (+ 2 i))
                            (+ (vector-ref veto-vector (+ 1 i))
                               (vector-ref veto-vector (1- i))))))))
             slope-2-veto-wt)
          0)))

(define! slope-2-b-veto-fn
   (lambda (i)
      (if (and (>= i 6) (< i N-1))
          (* (+ (vector-ref veto-vector (- i 4))
                (+ (vector-ref veto-vector (- i 5))
                   (+ (vector-ref veto-vector (- i 6))
                      (+ (vector-ref veto-vector (- i 3))
```

```
                        (+ (vector-ref veto-vector (- i 2))
                           (+ (vector-ref veto-vector (1- i))
                              (vector-ref veto-vector (1+ i))))))))
               slope-2-veto-wt)
            0)))

(define! slope-2-input-wt .3)
(define! slope-2-theta-d .05)
(define! slope-2-veto-wt 1)
(define! slope-2-inter-wt .3)
(define! slope-2-inter-theta .1) ;used to be .1
(define! slope-2-soma-wt .6)
(define! slope-2-soma-thresh .65)
(define! slope-2-soma-temp .01)

(define! run-slope2
   (lambda ()
      (slope-2-fn slope-2-f-vec slope-2-f-veto-fn slope-2p-f-vec)
      (slope-2-fn slope-2-b-vec slope-2-b-veto-fn slope-2p-b-vec)))

(define! slope-2-fn
   (lambda (vecs veto-fn othervecs)
      (catch-fn vecs 3 5)
      (catch-fn othervecs 3 5)
      (soma-fn vecs 2 6 slope-2-soma-wt slope-2-soma-thresh slope-2-soma-temp)
      (somap-fn othervecs 2 5 2
         slope-2-soma-wt slope-2-soma-thresh slope-2-soma-temp)
      (inter-fn vecs 1 slope-2-inter-wt slope-2-inter-theta)
      (dendr-fn vecs 0 veto-fn slope-2-input-wt slope-1-thresh-v
         slope-2-theta-d)))
```

The next code deals with two faster slopes. The subnetworks are similar to the slope-2 networks, but they cover a larger number of vector cells.

```
(define! slope-3-f-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)))

(define! slope-3-b-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)))
```

```
(define! slope-3p-f-soma-vec (make-vector N 0))

(define! slope-3p-b-soma-vec (make-vector N 0))

(define! slope-3p-f-vec
   (vector
      (vector-ref slope-3-f-vec 0)
      (vector-ref slope-3-f-vec 1)
      slope-3p-f-soma-vec
      (make-vector N 0)))

(define! slope-3p-b-vec
   (vector
      (vector-ref slope-3-b-vec 0)
      (vector-ref slope-3-b-vec 1)
      slope-3p-b-soma-vec
      (make-vector N 0)))

(define! slope-3-f-veto-fn
   (lambda (i)
      (if (and (< i N-6) (> i 3))
          (* (+ (vector-ref veto-vector (+ i 3))
                (+ (vector-ref veto-vector (+ i 2))
                   (+ (vector-ref veto-vector (+ i 1))
                      (+ (vector-ref veto-vector (+ i 4))
                         (+ (vector-ref veto-vector (+ i 5))
                            (+ (vector-ref veto-vector (+ i 6))
                               (+ (vector-ref veto-vector (- i 1))
                                  (vector-ref veto-vector (- i 2)))))))))
             slope-3-veto-wt)
          0)))

(define! slope-3-b-veto-fn
   (lambda (i)
      (if (and (> i 6) (< i N-2))
          (* (+ (vector-ref veto-vector (- i 4))
                (+ (vector-ref veto-vector (- i 5))
                   (+ (vector-ref veto-vector (- i 6))
                      (+ (vector-ref veto-vector (- i 3))
                         (+ (vector-ref veto-vector (- i 2))
                            (+ (vector-ref veto-vector (- i 1))
                               (+ (vector-ref veto-vector (+ i 1))
                                  (vector-ref veto-vector (+ i 2)))))))))
             slope-3-veto-wt)
          0)))


(define! slope-3-input-wt .3)
```

```
(define! slope-3-theta-d .01)
(define! slope-3-veto-wt 1)
(define! slope-3-inter-wt .3)
(define! slope-3-inter-theta .1)
(define! slope-3-soma-wt .6)
(define! slope-3-soma-thresh .65)
(define! slope-3-soma-temp .01)

(define! run-slope3
   (lambda ()
      (slope-3-fn slope-3-f-vec slope-3-f-veto-fn slope-3p-f-vec)
      (slope-3-fn slope-3-b-vec slope-3-b-veto-fn slope-3p-b-vec)))

(define! slope-3-fn
   (lambda (vecs veto-fn othervecs)
      (catch-fn vecs 3 7)
      (catch-fn othervecs 3 7)
      (soma-fn vecs 2 8 slope-3-soma-wt slope-3-soma-thresh slope-3-soma-temp)
      (somap-fn othervecs 2 7 3
         slope-3-soma-wt slope-3-soma-thresh slope-3-soma-temp)
      (inter-fn vecs 1 slope-3-inter-wt slope-3-inter-theta)
      (dendr-fn vecs 0 veto-fn slope-3-input-wt slope-1-thresh-v
         slope-3-theta-d)))

; slope 4 (skip 3 cells)

(define! slope-4-f-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)))

(define! slope-4-b-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)))

(define! slope-4p-f-soma-vec (make-vector N 0))

(define! slope-4p-b-soma-vec (make-vector N 0))

(define! slope-4p-f-vec
   (vector
      (vector-ref slope-4-f-vec 0)
      (vector-ref slope-4-f-vec 1)
      slope-4p-f-soma-vec
```

```
      (make-vector N 0)))

(define! slope-4p-b-vec
   (vector
      (vector-ref slope-4-b-vec 0)
      (vector-ref slope-4-b-vec 1)
      slope-4p-b-soma-vec
      (make-vector N 0)))

(define! slope-4-f-veto-fn
  (lambda (i)
    (if (and (< i N-8) (> i 4))
        (* (+ (vector-ref veto-vector (+ i 4))
              (+ (vector-ref veto-vector (+ i 3))
                 (+ (vector-ref veto-vector (+ i 2))
                    (+ (vector-ref veto-vector (+ i 1))
                       (+ (vector-ref veto-vector (- i 1))
                          (+ (vector-ref veto-vector (- i 2))
                             (vector-ref veto-vector (- i 3)))))))))
           slope-4-veto-wt)
      0)))

(define! slope-4-b-veto-fn
   (lambda (i)
      (if (and (> i 7) (< i N-4))
          (* (+ (vector-ref veto-vector (- i 4))
                (+ (vector-ref veto-vector (- i 3))
                   (+ (vector-ref veto-vector (- i 2))
                      (+ (vector-ref veto-vector (- i 1))
                         (+ (vector-ref veto-vector (+ i 1))
                            (+ (vector-ref veto-vector (+ i 2))
                               (vector-ref veto-vector (+ i 3)))))))))
             slope-4-veto-wt)
          0)))


(define! slope-4-input-wt .3)
(define! slope-4-theta-d .01)
(define! slope-4-veto-wt 1)
(define! slope-4-inter-wt .3)
(define! slope-4-inter-theta .1)
(define! slope-4-soma-wt .6)
(define! slope-4-soma-thresh .65)
(define! slope-4-soma-temp .01)

(define! run-slope4
   (lambda ()
      (slope-4-fn slope-4-f-vec slope-4-f-veto-fn slope-4p-f-vec)
      (slope-4-fn slope-4-b-vec slope-4-b-veto-fn slope-4p-b-vec)))
```

```
(define! slope-4-fn
   (lambda (vecs veto-fn othervecs)
      (catch-fn vecs 3 9)
      (catch-fn othervecs 3 9)
      (soma-fn vecs 2 10 slope-4-soma-wt slope-4-soma-thresh slope-4-soma-temp)
      (somap-fn othervecs 2 9 4
         slope-4-soma-wt slope-4-soma-thresh slope-4-soma-temp)
      (inter-fn vecs 1 slope-4-inter-wt slope-4-inter-theta)
      (dendr-fn vecs 0 veto-fn slope-4-input-wt slope-1-thresh-v
         slope-4-theta-d)))
```

The two slower slope detectors are shown below. In each case the last branch node has an additional connection to an input node. The subnetworks for the two slower slopes differ only in the parameters used in the activation functions.

```
(define! slope-05-f-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
            ))

(define! slope-05-b-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
            ))

; vec 0 is dendr, 1-5 is inter, 6 is soma

(define! slope-05-f-veto-fn
   (lambda (i)
      (if (<= i N-4)
          (* (+ (vector-ref veto-vector (+ i 3))
                (+ (vector-ref veto-vector (+ i 2))
```

```
                (vector-ref veto-vector (1+ i))))
            slope-05-veto-wt)
         0)))

(define! slope-05-b-veto-fn
   (lambda (i)
      (if (> i 3)
          (* (+ (vector-ref veto-vector (- i 3))
                (+ (vector-ref veto-vector (- i 2))
                   (vector-ref veto-vector (1- i))))
            slope-05-veto-wt)
          0)))

(define! slope-05-input-wt .3)
(define! slope-05-thresh-v slope-1-thresh-v)
(define! slope-05-theta-d .05)
(define! slope-05-veto-wt 1)
(define! slope-05-soma-wt 2.3)
(define! slope-05-soma-thresh .733)
(define! slope-05-soma-temp .008)

(define! run-slope05
   (lambda ()
      (slope-05-fn slope-05-f-vec slope-05-f-veto-fn)
      (slope-05-fn slope-05-b-vec slope-05-b-veto-fn)
      ))

(define! slope-05-fn
   (lambda (vec veto-fn)
      (catch-fn vec 7 3)
      (soma-fn vec 6 3 slope-05-soma-wt slope-05-soma-thresh
         slope-05-soma-temp)
      (branch-fn vec 5
         slope-05-5-inter-theta slope-05-feat-wt slope-05-br-wt)
      (branch-fn vec 4
         slope-05-4-inter-theta slope-05-feat-wt 0)
      (branch-fn vec 3
         slope-05-3-inter-theta slope-05-feat-wt 0)
      (branch-fn vec 2
         slope-05-2-inter-theta slope-05-feat-wt 0)
      (branch-fn vec 1
         slope-05-1-inter-theta slope-05-feat-wt 0)
      (dendr-fn vec 0 veto-fn slope-05-input-wt slope-1-thresh-v
         slope-05-theta-d)))


(define! slope-05-feat-wt .235) ; this was working at .23 with thresh .66
(define! slope-05-br-wt .05)
(define! slope-05-1-inter-theta .26)
```

```
(define! slope-05-2-inter-theta .26)
(define! slope-05-3-inter-theta .26)
(define! slope-05-4-inter-theta .26)
(define! slope-05-5-inter-theta .26)

(define! slope-03-f-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
            ))

(define! slope-03-b-vec
   (vector
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
      (make-vector N 0)
            ))

(define! slope-03-f-veto-fn
   (lambda (i)
      (if (<= i N-4)
          (* (+ (vector-ref veto-vector (+ i 3))
                (+ (vector-ref veto-vector (+ i 2))
                   (vector-ref veto-vector (1+ i))))
             slope-03-veto-wt)
          0)))

(define! slope-03-b-veto-fn
   (lambda (i)
      (if (> i 3)
          (* (+ (vector-ref veto-vector (- i 3))
                (+ (vector-ref veto-vector (- i 2))
                   (vector-ref veto-vector (1- i))))
             slope-03-veto-wt)
          0)))

(define! slope-03-input-wt .3)
(define! slope-03-thresh-v slope-1-thresh-v)
```

```
(define! slope-03-theta-d .05)
(define! slope-03-veto-wt 1)
(define! slope-03-soma-wt .6)
(define! slope-03-soma-thresh .793)
(define! slope-03-soma-temp .007)

(define! run-slope03
   (lambda ()
      (slope-03-fn slope-03-f-vec slope-03-f-veto-fn)
      (slope-03-fn slope-03-b-vec slope-03-b-veto-fn)
      ))

(define! slope-03-fn
   (lambda (vec veto-fn)
      (catch-fn vec 7 3)
      (soma-fn vec 6 3 slope-03-soma-wt slope-03-soma-thresh
         slope-03-soma-temp)
      (branch-fn vec 5
         slope-03-5-inter-theta slope-03-feat-wt slope-03-br-wt)
      (branch-fn vec 4
         slope-03-4-inter-theta slope-03-feat-wt 0)
      (branch-fn vec 3
         slope-03-3-inter-theta slope-03-feat-wt 0)
      (branch-fn vec 2
         slope-03-2-inter-theta slope-03-feat-wt 0)
      (branch-fn vec 1
         slope-03-1-inter-theta slope-03-feat-wt 0)
      (dendr-fn vec 0 veto-fn slope-03-input-wt slope-1-thresh-v
         slope-03-theta-d)))


(define! slope-03-feat-wt .18)
(define! slope-03-br-wt .1)
(define! slope-03-1-inter-theta .1)
(define! slope-03-2-inter-theta .1)
(define! slope-03-3-inter-theta .1)
(define! slope-03-4-inter-theta .1)
(define! slope-03-5-inter-theta .1)
```

2.3 Running the Motion Detector Network

This is the code that is actually used to run the network. For each syllable repetition, it expects a bit matrix of formant center frequencies. In its current state, the matrices of each syllable are in a different directory for each repetition, in files whose names are consist of the syllable name prepended by the file prefix "`out.`". This program then writes formant transition data in files with a prefix "`mv.`".

The program is invoked with the procedure call (doit ph-list), which identifies formant transitions for each syllable in ph-list.

```
(define! doit
  (lambda (phs)
    (for-each
     (lambda (ph)
       (fprintf *standard-output* "~a~n" ph)
       (init (string-append "out." ph) (string-append "mv." ph))
       (run 350)
       )
     phs)
    (exit 0)))


(define! run-net
   (lambda ()
      (set! *cycle* (1+ *cycle*))
      (when (zero? (modulo *cycle* net-ticks))
            (dump-catch-vecs catch-vecs *cycle*))
      (run-slope1)
      (run-slope2)
      (run-slope3)
      (run-slope4)
      (run-slope05)
      (run-slope03)
      (run-veto)
      (when (zero? (modulo *cycle* net-ticks))
            (run-slope0)
            (run-input)
            )
     ))

(define! dump-catch-vecs
   (lambda (c-vecs cycle)
      (fprintf *output-port* "~a:" (/ cycle net-ticks))
      (let ([c-end (vector-length c-vecs)])
         (iterate catch c-end
            (let ([c-vec (vector-ref c-vecs catch)])
               (iterate freq N
                  (let ([val (clean-num (vector-ref c-vec freq))])
                     (when (> val 0)
                           (fprintf *output-port* " ~a ~a ~a"
                              catch freq val)))))))
      (newline *output-port*)))

(define! mk-phon-list
   (lambda phons
      (recur loop ([ph phons])
```

```
        (if ph
           (append (mk-vowel-list (car ph))
              (loop (cdr ph)))
           ())))))

(define! mk-vowel-list
   (lambda (v)
      (list (string-append "b" v)
            (string-append "d" v)
            (string-append "g" v))))

(set! phs '("ee"
            "ih"
            "ei"
            "eh"
            "ae"
            "ah"
            "ou"
            "uu"))

(set! ph-list (apply mk-phon-list phs))
```

3. The Adaptive Syllable Detection Network

The code for the adaptive network was written in C [3], in an attempt to squeeze some more speed out of the machine. The adaptive network reads formant transition data placed in files from the motion detector network. Training and tesing is done one adaptive node at a time due to implementational constraints.

The network reads transition information one time slice at a time, propagates activation through the delay and training matrices, computes the adaptive node's activation, and, in the case of a learning cycle, adjusts the weights of the adaptive node. During testing, the activation value of the node is written at each time slice.

The program that follows is a version of the Raw–Average Experiment.

3.1 Declarations and Definitions

The code that follows contains global variable declarations, defaults, and macro definitions. This is the so–called ".h" file common to C programs. It is used for both the adaptive network and the veto recognition network.

```
#include "math.h"
#include "stdio.h"
#include "sys/file.h"

#define N 200                    /* The number of frequency units */
```

```
#define W 5                        /* Length of the delay matrix */
#define C 19                       /* Number of motion detectors */
#define Learning_constant 0.01
#define Length_to_reward 1
#define Reward 2.3                 /* Set to 1.5 for Avg-only experiment */
#define Penalty -0.8               /* Set to -1.0 for Avg-only experiment */
#define trace_decay 0.8
#define rec_thresh 1.0
#define rec_temp 0.1
#define default_cycle_length 20
#define default_training_cycles 4
#define TESTLEN 1
#define TRAINLEN 5
#define PHONL 24
#define FILESLEN 44
#define WFTYP O_TRUNC  O_WRONLY  O_CREAT
#define RFTYP O_RDONLY

double catchvecs[C][N];
int Training_cycle;
double warpmats[C][W][N];          /* The delay matrix */
double tracemats[C][W][N];         /* Eligibility trace matrix */
double recognizer_vec;             /* The value of the recognizer */
double recognizer_sum;
double recognizer_weight_vec[C][W][N];
double reward;
double learning_constant;
char mother[] = "/u/smythe/Res/move/data/kpdat/0dat/";
char testdirs[TESTLEN][3] = {"A/"};
char traindirs[TRAINLEN][3] = {"1/", "2/", "3/", "4/", "5/"};
int test_len = TESTLEN, train_len = TRAINLEN, subd_len = TRAINLEN;
char phons[PHONL][4] = {"bae", "bah", "bee", "beh", "bei", "bih", "bou", "buu",
                        "dae", "dah", "dee", "deh", "dei", "dih", "dou", "duu",
                        "gae", "gah", "gee", "geh", "gei", "gih", "gou", "guu"};
char fprefix[] = "mv.";
char filestr[FILESLEN];
FILE *inp, *outp, *fopen();
int do_dump, do_read, do_test;
char *dump_file = "dump";
int phonset, cycle_length, train_cycles, start_phon;
int row, col; /* for the debugger */
double value;
int c_table[PHONL];
    l_table[PHONL];
int config_set;
char *config_ptr;
int testphon_i;
```

3.2 The Actual Program

The following code is for the learning algorithm and one adaptive node. Nodes are trained one at a time for a number of efficiency and safety reasons. If all were trained at once, the weight matrices would have to be periodically dumped in case the computer would crash during training. Due to file space restrictions only one matrix could be dumped at any one time. This saves the amount of computation that must be repeated in the event of a crash.

```
char *testphon;
int dirin, phonin;

main(argc, argv)
int argc;
char **argv;
{
    int     i;

    phonset = 0;
    config_set = 0;                 /* set defaults and set arguments */
    cycle_length = 20;
    train_cycles = 4;
    outp = stdout;
    start_phon = 0;
    do_dump = 0;
    do_read = 0;
    do_test = 1;

    for (; (argc > 1) && (argv[1][0] == '-'); argc--, argv++) {
        switch (argv[1][1]) {
                                /* The program can be started to run
                                   either for all the syllables, or
                                   for a single syllable specified
                                   with the -p option                */
            case 'p':
                argc--;
                argv++;
                testphon = &argv[1][0];
                phonset = 1;
                continue;
                                /*  The -c option sets a training
                                    length that is different from
                                    the default.                     */
            case 'c':
                argc--;
                argv++;
                train_cycles = atoi (argv[1]);
```

```
        continue;
                        /* The -l option sets an exposure
                           window other than the default  */
    case 'l':
        argc--;
        argv++;
        cycle_length = atoi (argv[1]);
        continue;
                        /* The -s option starts the program
                           at a syllable other than where it
                           normally begins.  This is to
                           recover from a crash           */
    case 's':
        argc--;
        argv++;
        start_phon = atoi (argv[1]);
        continue;
                        /* The -d option dumps the weight
                           matrix to a file called ''dump''
                           at the end of the run          */
    case 'd':
        do_dump = 1;
        continue;
                        /* The -r option reads the weight
                           matrix from the dump file.     */
    case 'r':
        do_read = 1;
        continue;
                        /* The -t option does not run a
                           testing phase.  You had jolly
                           well better use it in conjunction
                           with the -d option             */
    case 't':
        do_test = 0;
        continue;
                        /* The -C option, given a file name
                           will read configuration parameters
                           from the file.  These are in the
                           form of
                      <syl> <exposure window> <learning constant>
                         */
    case 'C':
        argc--;
        argv++;
        config_set = 1;
        config_ptr = &argv[1][0];
        config(config_ptr);
        continue;
}
```

```
    }

    if (phonset >= 1) {
        exec ();
    }
    else {
        for (i = start_phon; i < PHONL; i++) {
            testphon = &phons[i][0];
            if (config_set) {
              testphon_i = findsyl(testphon);
              cycle_length = l_table[testphon_i];
              train_cycles = c_table[testphon_i];
            }
            outp = fopen (testphon, "w");
            exec ();
            fclose (outp);
        }
    }
}

exec ()
{
    int     i,
            j,
            dumptr;

    if (do_read == 0)
        InitMat (&recognizer_weight_vec[0][0][0]);
    else {
        dumptr = open (dump_file, RFTYP);
        i = read (dumptr, recognizer_weight_vec, C * W * N * sizeof (double));
        if (i != C * W * N * sizeof (double)) {
            fprintf (stderr, "read of dumpfile %s, failed: %d\n",
                    dump_file, i);
            exit (1);
        }
        close (dumptr);
    }

    for (j = 0; j < train_cycles; j++) {
        for (dirin = 0; dirin < train_len; dirin++) {
            for (phonin = 0; phonin < PHONL; phonin++) {

                InitTask (&traindirs[dirin][0], phonin);
                for (i = 0; i < cycle_length; i++) {
                    LearnCycle ();
                }
                fclose (inp);
            }
```

```
        }
    }

    if (do_test == 1) {
        for (dirin = 0; dirin < test_len; dirin++) {
            for (phonin = 0; phonin < PHONL; phonin++) {

                InitTask (&testdirs[dirin][0], phonin);
                fprintf (outp, "%s %g %d\n", filestr, reward, train_cycles);
                learning_constant = 0;

                for (i = 0; i < cycle_length; i++) {
                    LearnCycle ();
                    fprintf (outp, "%1.2e ", recognizer_vec);
                }
                fclose (inp);
                fprintf (outp, "\n");
            }
        }
    }
    if (do_dump == 1)
        Dump ();
}


InitTask(dirptr, phonin)
int phonin;
char *dirptr;
{
    makefilestr (dirptr, &phons[phonin][0]);
    inp = fopen (filestr, "r");
    if (Streq (testphon, &phons[phonin][0]))
        reward = Reward;
    else
        reward = Penalty;

    InitLearnCycle ();
}

LearnCycle()                        /* This drives a learning cycle */
{
    Training_cycle++;
    RunMotionNet();
    RunRecognizers();
    RunTrace();
    AdjustWeights();
}

RunMotionNet()                        /* This runs the motion net */
```

```
{
    int i, j;
    double v, *cvp;

    RunWarp();

    cvp = &catchvecs[0][0];
    for (i=0; i < C * N; i++) {
        *(cvp + i) = 0;
    }
    while (getc(inp) != ':');

    while(getc(inp) != '\n')  {
        fscanf(inp, "%d %d %f", &i, &j, &v);
        catchvecs[i][j] = v;
    }
}

InitLearnCycle()
{
    int     i;
    double *cvp;

    cvp = &catchvecs[0][0];
    for (i = 0; i < C * N; i++) {
        *(cvp + i) = 0;
    }
    InitMat (&warpmats[0][0][0]);
    InitMat (&tracemats[0][0][0]);
    learning_constant = Learning_constant;
}

RunWarp()                          /* This runs the delay matricies */
{
    int     i,
            j;
    register    k;
    register double *kp,
                    *kp1;

    for (i = 0; i < C; i++) {
        for (j = W - 1; j > 0; j--) {
            kp = &warpmats[i][j][0];
            kp1 = &warpmats[i][j - 1][0];
            for (k = 0; k < N; k++) {
                *kp = *kp1;
                kp++;
                kp1++;
            }
```

```
        }
        kp = &warpmats[i][0][0];
        kp1 = &catchvecs[i][0];
        for (k = 0; k < N; k++) {
            *kp = *(kp1 + k);
            kp++;
        }
    }
}

RunRecognizers()                /* runs the recognizer nodes */
{
    register int    i;
    register double sum,
                    *weightp,
                    *warpp;

    weightp = &recognizer_weight_vec[0][0][0];
    warpp = &warpmats[0][0][0];
    sum = 0.0;
    for (i = 0; i < C * W * N; i++) {
        sum = sum + (*weightp * *warpp);
        weightp++;
        warpp++;
    }
    recognizer_sum = sum;
    recognizer_vec = 1.0 / (1.0 + exp (-(sum - 1.0) /
                0.1));
}

RunTrace()                          /* runs the trace matricies */
{
    register int    i;
    register double *wm,
                    *tm;

    wm = &warpmats[0][0][0];
    tm = &tracemats[0][0][0];
    for (i = 0; i < C * W * N; i++) {
        *tm =
            (*wm >= *tm) ?
            *wm :
            ((trace_decay * *tm) +
                ((trace_decay - 1) * *wm));
        wm++;
        tm++;
    }
}
```

```
AdjustWeights()
{
    register int    i;
    register double *tracep,
                    *weightp;

    if (learning_constant != 0) {
        tracep = &tracemats[0][0][0];
        weightp = &recognizer_weight_vec[0][0][0];
        for (i = 0; i < C * W * N; i++) {
            *weightp = *weightp +
                (learning_constant * ((reward - recognizer_sum) *
                        *tracep));;
            weightp++;
            tracep++;
        }
    }
}

makefilestr(subdir, phon)
char *subdir, *phon;
{
    char *c, *c1;

    c = filestr;
    for (c1 = mother; *c1 != '\0'; c1++) {
        *c++ = *c1;
    }
    for (c1 = subdir; *c1 != '\0'; c1++) {
        *c++ = *c1;
    }
    for (c1 = fprefix; *c1 != '\0'; c1++) {
        *c++ = *c1;
    }
    for (c1 = phon; *c1 != '\0'; c1++) {
        *c++ = *c1;
    }
    *c = '\0';
}

InitMat(dp)
double *dp;
{
    int     i;

    for (i = 0; i < C * W * N; i++) {
        *(dp + i) = 0.0;
    }
}
```

```
Streq(s1, s2)
char *s1, *s2;
{
    for( ; *s1 == *s2 ; s1++, s2++) {
        if (*s1 == '\0')
           return(1);
       }
       return(0);
}

Dump()                                  /* temporary */
{
    int     dumptr,
            i;

    dumptr = open (dump_file, WFTYP, 0622);
    i = write (dumptr, recognizer_weight_vec, C * W * N * sizeof (double));
    if (i != (C * W * N * sizeof (double)))
        fprintf (stderr, "weight dump failed\n");

    close (dumptr);
}

config(ptr)
char *ptr;
{
    int     index,
            val,
            j;
    char    phon[5];

    for (index = 0; index < PHONL; index++) {
        c_table[index] = default_training_cycles;
        l_table[index] = default_cycle_length;
    }
    inp = fopen (ptr, "r");
    while (!eof (inp)) {        /* set the recognition weight */
        fscanf (inp, "%s", phon);
        index = findsyl (phon);
        fscanf (inp, "%d", &val);
        l_table[index] = val;
        if (getc(inp) != '\n') {
          fscanf(inp, "%d", &val);
          c_table[index] = val;
          getc(inp);
        }
    }
    fclose (inp);
```

```
}

findsyl(phon)
char *phon;
{
    int     index;

    for (index = 0; !Streq (phon, &phons[index][0]) && (index < PHONL);
            index++);
    return (index);
}

eof(ptr)
FILE *ptr;
{
    char    c;

    if ((c = getc (ptr)) != EOF) {
        ungetc (c, ptr);
        return (0);
    }
    else
        return (1);
}

skipline(ptr)
FILE *ptr;
{
    char    c;
    while ((c = getc (ptr)) != '\n');
}
```

4. Veto Recognition Network Code

After the training phase is completed, the network performance is analyzed to construct the veto recognition network. The network is constructed by providing it with another type of configuration file the sets the excitatory weight for each veto node's adaptive node. The configuration file also contains the veto connections and thresholds for each syllable.

The network is provided with the activation value of each of the adaptive nodes, one time slice at a time. Normally the performance data from the testing phase is used, since this is composed of the activation value of each adaptive node at each time slice. The output of each adaptive node is written at each time slice.

```
#include "move.h"                /* declarations from adaptive net */
#define CYCLE_LENGTH 33
#define WARPL 15
```

```
double dvec[PHONL],                /* discriminator vector */
       dvec_weights[PHONL],        /* weight vec for dvec */
       veto_thresh[PHONL][PHONL], /* thresholds for veto */
       veto_vec[PHONL],
       rvec[PHONL],                /* recognizer (input) vector */
       wvec[WARPL][PHONL],        /* warp vector */
       veto_theta,                /* threshold for veto sig fn */
       veto_temp;                 /* temperature */

char *phon,
     filehead[] = "/u/smythe/Res/move/data/kpdat/Rin/cycle.",
     filestring[100],
     cfile[] = "config",
     *cfptr;

main(argc, argv)
int argc;
char **argv;
{
    int     cycle;

    cfptr = cfile;
    veto_theta = rec_thresh;
    veto_temp = .01;

    for (; (argc > 1) && (argv[1][0] == '-'); argc--, argv++) {
        switch (argv[1][1]) {
                                /* The program can run for all
                                   syllables or for a single
                                   syllable specified by the -p
                                   option.                      */
            case 'p':
                argc--;
                argv++;
                phon = &argv[1][0];
                continue;
                                /* The -c option specifies a
                                   configuration file for the
                                   network.                     */
            case 'c':
                argc--;
                argv++;
                cfptr = argv[1];
        }
    }
    config (cfptr);
    init();
    for (cycle = 0; cycle < CYCLE_LENGTH + WARPL; cycle++) {
```

```
        run_net(cycle);
        output_net(cycle);
    }
    finish();
}

init()
{
    int     i, j;

    concat_string (filehead, phon);
    inp = fopen (filestring, "r");

    for (i = 0; i < PHONL; i++) {
        dvec[i] = 0;
        rvec[i] = 0;
        for (j = 0; j < WARPL; j++) {
            wvec[j][i] = 0;
        }
    }
}

run_net(cycle)
int cycle;
{
    char    c;
    int     i,
            j,
            vetoed;
    double  sum;

    if (!eof (inp)) {
        skipline (inp);
        for (i = 0; i < PHONL; i++) {
            fscanf (inp, "%f", &rvec[i]);
        }
        skipline (inp);
    }
 /* run the warp */

    for (i = 0; i < PHONL; i++) {
        wvec[0][i] = rvec[i];
    }

    for (i = WARPL - 1; i > 0; i--) {
        for (j = 0; j < PHONL; j++) {
            wvec[i][j] = wvec[i - 1][j];
        }
    }
```

```
/* compute the sum */

    for (i = 0; i < PHONL; i++) {
        vetoed = 0;
        for (j = 0; (j < PHONL) && (vetoed == 0) && (veto_vec[i] == 0); j++) {
            if ((rvec[j] > veto_thresh[i][j]) && (veto_thresh[i][j] > 0))
                vetoed = 1;
        }
        if (vetoed == 1)
           veto_vec[i] = 1;
        if (veto_vec[i] == 1) {
            dvec[i] = 0;
        }
        else {
            sum = wvec[WARPL - 1][i] * dvec_weights[i];
            dvec[i] = 1.0 / (1.0 + exp (-(sum - veto_theta) / veto_temp));
        }
    }
}

output_net(cycle)
int cycle;
{
    int i;

    for (i = 0; i < PHONL; i++) {
        printf("%1.2e ", dvec[i]);
    }
    printf("\n");
}

Streq(s1, s2)
char *s1, *s2;
{
    for( ; *s1 == *s2 ; s1++, s2++) {
        if (*s1 == '\0')
           return(1);
       }
       return(0);
}

config(ptr)
char *ptr;
{
    int     index,
            j;
    double  val;
    char    phon[5];
```

```
    for (index = 0; index < PHONL; index++) {
        dvec_weights[index] = 0;
        for (j = 0; j < PHONL; j++) {
            veto_thresh[index][j] = 0;
        }
    }
    inp = fopen (ptr, "r");
    while (!eof (inp)) {        /* set the recognition weight */
        fscanf (inp, "%s", phon);
        index = findsyl (phon);
        fscanf (inp, "%f", &val);
        dvec_weights[index] = 1 / val;

        while (getc (inp) != '\n') {
            fscanf (inp, "%s", phon);
            j = findsyl (phon);
            fscanf (inp, "%f", &val);
            veto_thresh[index][j] = val;
        }
    }
    fclose (inp);
}

findsyl(phon)
char *phon;
{
    int     index;

    for (index = 0; !Streq (phon, &phons[index][0]) && (index < PHONL);
            index++);
    return (index);
}

eof(ptr)
FILE *ptr;
{
    char    c;

    if ((c = getc (ptr)) != EOF) {
        ungetc (c, ptr);
        return (0);
    }
    else
        return (1);
}

skipline(ptr)
FILE *ptr;
```

```
{
    char    c;
    while ((c = getc (ptr)) != '\n');
}


concat_string(s1, s2)
char *s1, *s2;
{
    int i;

    for (i = 0; *s1 != '\0'; i++) {
        filestring[i] = *s1++;
    }
    for (; *s2 != '\0'; i++) {
        filestring[i] = *s2++;
    }
    filestring[i] = '\0';
}


finish()
{
    fclose(inp);
}
```

References

1. Smythe, E. J., "Temporal Computation in Connectionist Models," Ph.D. thesis, TR #251, Computer Science Department, Indiana University, Bloomington, Indiana, 1988.

2. Dybvig, R. K., *The Scheme Programming Language*, Prentice Hall, 1987.

3. Kernighan, B. W., Ritchie, D. M., *The C Programming Language*, Prentice Hall, 1978.