```
PROGRAM PCR5 (Input, Output, Pad_in, Pad_out, Grid_file, Print_file,
              Vec_file, Mat_file);
```

BROWN UNIVERSITY

Thesis project

CHARACTER RECOGNITION
USING NEURAL NETWORKS, POLAR COORDINATES,
AND LINE ORIENTATION

Michael Weiner
Undergraduate candidate for honors Sc.B. degree in Cognitive Science

18 February 1988

This program is designed to recognize hand-drawn upper-case letters of the English alphabet. The program accepts input from a mouse-controlled BITPAD. It then analyzes the input using polar coordinates, orientation of line segments, and position of endpoints detected in the input. The BSB non-linearity is applied to vectors encoded with this information. The resulting vector is then compared with similarly processed vectors from a model set of data, and a best match is then found and reported.

```
CONST
  Dimensionality    = 206;
  Char_file         = 'GRIDS.DAT';   File of stored model characters
  Test_file         = 'TEST.DAT';    File of stored test characters
  Vecs1_file        = 'VECS1.DAT';   File of stored model vectors
  Vecs2_file        = 'VECS2.DAT';   File of stored test vectors
  Auto_file         = 'AUTO.DAT';    File of the autoassociated model matrix
  BSB_vec_file      = 'BSWV.DAT';    File of BSBed model vectors
  BSB_output_file   = 'BSBO.DAT';    File of BSBed model vectors for output
  Match_file        = 'TEST.VEC';    File of vector matches
  Present           = 30;   Average number of presentations for each character
  Percent_thresh    = 100;   Do not ignore any input points
  N_chars           = 26;   Number of characters to recognize
  Char_start_value  = 65;   ASCII value of first character to recognize
  N_slices          = 18;   Number of pie slices for input
  N_rings           = 5;   Number of rings (subdivisions) of each slice
  Pad_width         = 2200;   Maximum X value of BITPAD
  Pad_height        = 2200;   Maximum Y value of BITPAD
  X                 = 1;   These definitions make using arrays convenient
  Y                 = 2;
  Lo                = 1;
  Hi                = 2;
  Case_difference   = ORD ('a') - ORD ('A');   For case conversion
  Pi                = 3.141592654;
  D_theta           = 2 * Pi / N_slices;   Angular size of each pie slice
  Comma             = ',';
  Screen_height     = 23;
  Screen_width      = 79;
  Grid_width        = Screen_width;   Highest X-value for input
  Grid_height       = Screen_height;   Highest Y-value for input
```



```
  CSI               = CHR(27) + '[';   Screen-control code
  Yellow            = 1;   Buttons of the BITPAD mouse
  White             = 2;
  Blue              = 4;
  Green             = 8;
  Bit_rate          = 'C';   Switched-stream mode for BITPAD
  Right             = 0;   Used for tracing; this routine was not implemented,
  Up                = 1;     and is incomplete
  Left              = 2;
  Down              = 3;
  Connects_max      = 8;   Used for tracing; this routine was not implemented,
                           and is incomplete

TYPE
  Vector              = ARRAY [1..Dimensionality] OF REAL;
  Matrix              = ARRAY [1..Dimensionality, 1..Dimensionality] OF REAL;
  Connectivity_matrix = ARRAY [1..Connects_max, 1..Connects_max] OF BOOLEAN;
    Used for tracing; this routine was not implemented, and is incomplete
  Vector_set          = ARRAY [1..N_chars] OF Vector;
  Matrix_set          = ARRAY [1..N_chars] OF Matrix;
  History_type = (NEW,OLD,READONLY,UNKNOWN);
  Slice_code          = ARRAY [1..2*N_rings] OF INTEGER;
    Bits of encoded input for one slice
  Grid                = ARRAY [0..Grid_width,0..Grid_height] OF BOOLEAN;
  Two_things          = ARRAY [1..2] OF INTEGER;
  Point               = Two_things;
  Line                = ARRAY [1..2] OF Point;
  RSlice              = ARRAY [1..2] OF Line;   Corners of ring-slice
  Slice_bits          = ARRAY [1..N_rings] OF INTEGER;
    Encoded input for one intersection of one slice--half of Slice_code
  Ring_radii    = ARRAY [0..N_rings] OF REAL;   Radii of ring-slices
  String              = VARYING [80] OF CHAR;
  Direction           = 0..3;   0 = right; 1 = up; 2 = left; 3 = down
    Used for tracing; this routine was not implemented, and is incomplete
  Seg_bits            = ARRAY [1..3] OF INTEGER;
    For 1 unit of line-based information
  End_bits            = ARRAY [1..4] OF INTEGER;
    For 4 units of information about endpoints (1 unit for each quadrant)

VAR
  Seed                : INTEGER := 22;   For random-number generation
  Group               : INTEGER := 5;   Number of screen groupings
  Pad_in, Pad_out     : TEXT;   BITPAD
  Grid_file           : FILE OF Grid;
  Vec_file            : FILE OF Vector;
  Mat_file            : FILE OF REAL;   Autoassociation matrix
  Print_file          : TEXT;
  Out_char            : ARRAY [0..1] OF CHAR;   Screen characters for Grid
  BSB_done            : BOOLEAN := FALSE;
    TRUE if model vectors have been BSBed
  BSB_vec_set         : Vector_set;   BSBed model vectors
  Auto_matrix         : Matrix;
  VT220               : CHAR := 'N';   Current terminal a VT220 ?

PROCEDURE Clear_screen;
```

```
  Clear the screen

  BEGIN
    IF (VT220 = 'Y') THEN WRITE(CSI,'2J');
  END;

PROCEDURE Cursor_posn (Col, Row : INTEGER);
  Position the cursor on the screen.  (0,0) is the lower left-hand corner.

  VAR
    X_coord, Y_coord : INTEGER;

  BEGIN
    IF (VT220 = 'Y') THEN
      BEGIN
        X_coord := Col;
        Y_coord := Row;
        IF Col > Screen_width THEN X_coord := Screen_width ELSE
        IF Col < 0 THEN X_coord := 0;
        IF Row > Screen_height THEN Y_coord := Screen_height ELSE
        IF Row < 0 THEN Y_coord := 0;
        WRITE (CSI,Screen_height-Y_coord:1,';',X_coord:1,'H');
      END;
  END;

PROCEDURE Clear_line (Row : INTEGER);
  Clear the entire line (row given by Row)

  BEGIN
    IF (VT220 = 'Y') THEN
      BEGIN
        Cursor_posn (0,Row);
        WRITE (CSI,'2K');
      END;
  END;

FUNCTION Direc (Num : INTEGER) : INTEGER;
  Standardize numerically the current direction; used for tracing; this
  routine was not implemented, and is incomplete

  BEGIN
    Direc := Num MOD 4;
  END;

FUNCTION Scrunch (Number, Scrunch_from, Scrunch_to : INTEGER) : INTEGER;
  Map one scale to another

  BEGIN
    Scrunch := ROUND (Number * Scrunch_to / Scrunch_from);
  END;

FUNCTION Instr (Start          : INTEGER;
                Object, Search : String) : INTEGER;
  Position of Search in Object.  Search will be found if it is located at
  position Start or after.
```



```
  BEGIN
    Instr := INDEX ( SUBSTR (Object,Start,LENGTH(Object)-Start+1),Search)
             + Start - 1;
  END;

FUNCTION MTH$RANDOM (Seed: INTEGER): REAL; EXTERN;  Set up for random nos.

FUNCTION Random : REAL;
  Get a random number between 0 and 1

  BEGIN
    Random := MTH$RANDOM (Seed);
  END;

FUNCTION Rand_int (Max : INTEGER) : INTEGER;
  Pick a random integer from 1 to Max

  BEGIN
    Rand_int := TRUNC (Random*Max+1);
  END;

PROCEDURE Outer_product (A, B  : Vector;
                         VAR C     : Matrix);
  Compute the outer product (ABtranspose) of two vectors (A & B)

  VAR
    I, J : INTEGER;

  BEGIN
    FOR I := 1 to Dimensionality DO
      FOR J := 1 to Dimensionality DO C [I,J] := B[J] * A[I];
  END;

PROCEDURE Initialize_vector (VAR Vec : Vector);
  Zero vector elements

  VAR Bin : INTEGER;

  BEGIN
    FOR Bin := 1 to Dimensionality DO
      Vec [Bin] := 0;
  END;

PROCEDURE Matrix_times_vector (A : Matrix;
                               F : Vector;
                           VAR G : Vector);
  Multiply matrix times vector

  VAR
    I, J : INTEGER;

  BEGIN
    Initialize_vector (G);
    FOR I := 1 TO Dimensionality DO
```

```
      FOR J := 1 TO Dimensionality DO G[I] := G[I] + A[I,J] * F[J];
  END;

PROCEDURE Print_vec (Vec : Vector);
  Print a vector

  VAR
    Slice, Bit : INTEGER;
    Counter    : INTEGER;

  BEGIN
    FOR Slice := 1 TO N_slices DO
      BEGIN
        Counter := 0;
        FOR Bit := 1 to 2 * N_rings DO
          BEGIN
            WRITE (' ', Vec[(Slice-1)*2*N_rings+Bit]:5:2);
            Counter := Counter + 1;
            IF (Counter >= Group) THEN
              BEGIN
                WRITELN;
                Counter := 0;
              END;
          END;
        WRITELN;
      END;  Slice
  END;

PROCEDURE Add_vectors (Vector_1, Vector_2 : Vector;
                   VAR Vector_sum         : Vector);
  Add two vectors

  VAR I : INTEGER;

  BEGIN
    FOR I := 1 to Dimensionality DO
      Vector_sum [I] := Vector_1 [I] + Vector_2 [I];
  END;

PROCEDURE Add_matrices (M1, M2 : Matrix;
                    VAR M_sum  : Matrix);
  Add two matrices

  VAR I, J : INTEGER;

  BEGIN
    FOR I := 1 to Dimensionality DO
      FOR J := 1 to Dimensionality DO
        M_sum [I,J] := M1 [I,J] + M2 [I,J];
  END;

PROCEDURE Subtract_vectors (Vector_1, Vector_2 : Vector;
                        VAR Vector_difference  : Vector);
  Subtract 2 vectors
```



```
  VAR I: INTEGER;

  BEGIN
    FOR I:= 1 to Dimensionality DO
      Vector_difference [I] := Vector_1 [I] - Vector_2 [I];
  END;

FUNCTION Vector_length (Vector_1 : Vector) : REAL;
  Find length of vector

  VAR
    Sum_of_squares : REAL;
    I              : INTEGER;

  BEGIN
    Sum_of_squares:= 0;
    FOR I:= 1 TO Dimensionality DO
      Sum_of_squares := Vector_1 [I]*Vector_1 [I] + Sum_of_squares;
    Vector_length:= SQRT (Sum_of_squares);
  END;

PROCEDURE Scalar_times_vector (Scalar     : REAL;
                               Vector_in  : Vector;
                           VAR Vector_out : Vector);
  Dilate vector

  VAR I : INTEGER;

  BEGIN
    FOR I := 1 to Dimensionality DO
      Vector_out [I] := Scalar * Vector_in [I];
  END;

FUNCTION Inner_product (Vector_1, Vector_2 : Vector) : REAL;
  Take inner product

  VAR
    I              : INTEGER;
    Sum_of_products : REAL;

  BEGIN
    Sum_of_products := 0;
    FOR I := 1 TO Dimensionality DO
      Sum_of_products := Vector_1 [I] * Vector_2 [I] + Sum_of_products;
    Inner_product := Sum_of_products;
  END;

PROCEDURE Normalize (Vector_in  : Vector;
                 VAR Vector_out : Vector);
  Normalize a vector

  VAR
    Length_of_vector_in : REAL;
    I                   : INTEGER;
    Vector_in_is_zero   : BOOLEAN;
```

```
  BEGIN
    Vector_out := Vector_in;
    Length_of_vector_in := Vector_length (Vector_in);
    IF (Length_of_vector_in = 0) THEN Vector_in_is_zero := TRUE
                                 ELSE Vector_in_is_zero := FALSE;
    IF Vector_in_is_zero
      THEN BEGIN
             WRITE ('Normalized zero vector!');
               Vector_out := Vector_in;   Make Vector_out the zero vector
             END   No semicolon here!
      ELSE FOR I := 1 to Dimensionality DO
       Vector_out [I] := Vector_in [I] / Length_of_vector_in;
  END;

FUNCTION Vector_cosine (Vector_1, Vector_2 : Vector): REAL;
   Find the cosine between two vectors

  VAR
    Normal_vector_1, Normal_vector_2 : Vector;

  BEGIN
    Normalize (Vector_1, Normal_vector_1);
    Normalize (Vector_2, Normal_vector_2);
    Vector_cosine := Inner_product (Normal_vector_1, Normal_vector_2);
  END;

PROCEDURE Initialize_grid (VAR Data_grid : Grid);
   Set grid elements to FALSE

  VAR
    X_coord, Y_coord : INTEGER;

  BEGIN
    FOR X_coord := 1 TO Grid_width DO
      FOR Y_coord := 1 TO Grid_height DO
        Data_grid [X_coord,Y_coord] := FALSE;
  END;

PROCEDURE Initialize_matrix (VAR Mat : Matrix);
   Set matrix elements to zero

  VAR I, J : INTEGER;

  BEGIN
    FOR I := 1 to Dimensionality DO
      FOR J := 1 to Dimensionality DO Mat [I,J] := 0;
  END;

 PROCEDURE Initialize_connect_matrix (VAR Mat : Connectivity_matrix);
   Set connectivity-matrix elements to FALSE; used for tracing; this routine
   was not implemented, and is incomplete

  VAR I, J : INTEGER;
```



```
  BEGIN
    FOR I := 1 TO Connects_max DO
      FOR J := 1 TO Connects_max DO Mat [I,J] := FALSE;
  END;

PROCEDURE Message (Row            : INTEGER;
                  Message_string : String);
   Display a message at row given by Row.

  BEGIN
    Clear_line (Row);
    WRITELN (Message_string);
  END;

PROCEDURE Get_input (Row    : INTEGER;
                     Prompt : String;
                 VAR Return : String);
   Get input in response to a prompt.

  BEGIN
    Clear_line (Row);
    WRITE (Prompt);
    IF (EOLN = TRUE) THEN READLN;
    READLN (Return);
  END;

FUNCTION Upper_case (Character : CHAR) : CHAR;
   Convert letter to upper-case

  VAR
    Offset : INTEGER;

  BEGIN
    IF (ORD (Character) >= ORD ('a')) AND
       (ORD (Character) <= ORD ('z'))
     THEN Offset := Case_difference
     ELSE Offset := 0;
    Upper_case := CHR ( ORD (Character) - Offset );
  END;

PROCEDURE Get_char (Row    : INTEGER;
                    Prompt : String;
                VAR Return : CHAR);
   Get a single character in response to a prompt.  Return the upper-case
   equivalent of the input.

  BEGIN
    Clear_line (Row);
    WRITE (Prompt);
    IF (EOLN = TRUE) THEN READLN;
    READ (Return);
    Return := Upper_case (Return);
  END;

PROCEDURE Wait_for_return;
```

```
  Get any character.

  VAR
    Dummy : CHAR;

  BEGIN
    Get_char (1,'Press <RETURN> to continue ',Dummy);
  END;

FUNCTION Rec_num (Character : CHAR) : INTEGER;
  Return the file's record number corresponding to Character

  BEGIN
    Rec_num := ORD (Character) - Char_start_value + 1;
  END;

FUNCTION Rec_to_char (Rec_num : INTEGER) : CHAR;
  Return the character corresponding to a record number

  BEGIN
    Rec_to_char := CHR ( Rec_num + Char_start_value - 1 );
  END;

PROCEDURE Store_char (Char_to_store  : CHAR;
                      Grid_to_store  : Grid;
                      Grid_file_name : String);   Filename of Grid_file
  Store a character grid in a file

  BEGIN
    OPEN (FILE_NAME       := Grid_file_name,
          FILE_VARIABLE   := Grid_file,
          HISTORY         := UNKNOWN,
          ACCESS_METHOD   := DIRECT,   Random access
          RECORD_TYPE     := FIXED,    Each grid has the same length
          CARRIAGE_CONTROL := NONE,    No special control codes
          ORGANIZATION    := SEQUENTIAL,
          DISPOSITION     := SAVE);  Retain the file when it is closed
    LOCATE (Grid_file, Rec_num (Char_to_store));   Position file pointer
    Grid_file := Grid_to_store;   Put grid into file buffer
    PUT (Grid_file);   Store the grid
    CLOSE (Grid_file);
  END;

PROCEDURE Read_char (Char_to_read   : CHAR;
                     Grid_file_name : String;   Filename of Grid_file
                 VAR Char_found     : BOOLEAN;
                 VAR Stored_grid    : Grid);
  Read a character from a grid file

  BEGIN
    Initialize_grid (Stored_grid);
    OPEN (FILE_NAME       := Grid_file_name,
          FILE_VARIABLE   := Grid_file,
          HISTORY         := UNKNOWN,
          ACCESS_METHOD   := DIRECT,   Random access
```



```
          RECORD_TYPE     := FIXED,    Each grid has the same length
          CARRIAGE_CONTROL := NONE,    No special control codes
          ORGANIZATION    := SEQUENTIAL,
          DISPOSITION     := SAVE);   Retain the file when it is closed
    FIND (Grid_file, Rec_num (Char_to_read));   Search for character
    IF UFB (Grid_file)
      THEN Char_found := FALSE
      ELSE
        BEGIN
          Char_found := TRUE;
          Stored_grid := Grid_file;
        END;
    CLOSE (Grid_file);
  END;

PROCEDURE Read_vec (Char_to_read  : CHAR;
                    Vec_file_name : String;   Filename of Vec_file
                VAR Char_found    : BOOLEAN;
                VAR Stored_vec    : Vector);
  Read a character from a grid file

  BEGIN
    Initialize_vector (Stored_vec);
    OPEN (FILE_NAME       := Vec_file_name,
          FILE_VARIABLE   := Vec_file,
          HISTORY         := READONLY,
          ACCESS_METHOD   := DIRECT,   Random access
          RECORD_TYPE     := FIXED,    Each grid has the same length
          CARRIAGE_CONTROL := NONE,    No special control codes
          ORGANIZATION    := SEQUENTIAL,
          DISPOSITION     := SAVE);   Retain the file when it is closed
    FIND (Vec_file, Rec_num (Char_to_read));
    IF UFB (Vec_file)
      THEN Char_found := FALSE
      ELSE
        BEGIN
          Char_found := TRUE;
          Stored_vec := Vec_file;
        END;
    CLOSE (Vec_file);
  END;

PROCEDURE Find_grid_traits (Data_grid       : Grid;
                        VAR Center          : Point;
                        VAR Num_of_grid_pts : INTEGER);
  Return number of grid points & center of mass

  VAR
    X_coord, Y_coord    : INTEGER;
    Sum_of_coord_values : Two_things;
    XY                  : INTEGER;
     LH                  : INTEGER;   This was used to assign center to the
                                      average of extremes; not currently being
                                      implemented
    Box                 : ARRAY [Lo..Hi] OF Two_things;
```

```
  BEGIN

    FOR XY := X TO Y DO Sum_of_coord_values [XY] := 0;   Initialize sums
    Num_of_grid_pts := 0;
     FOR LH := Lo TO Hi DO
      FOR XY := X TO Y DO Box [LH] [XY] := 0;

    FOR X_coord := 1 TO Grid_width DO
      FOR Y_coord := 1 to Grid_height DO
        IF (Data_grid [X_coord,Y_coord] = TRUE) THEN   If there's a point then
          BEGIN
            Num_of_grid_pts := Num_of_grid_pts + 1;
            Sum_of_coord_values [X] := Sum_of_coord_values [X] + X_coord;
            Sum_of_coord_values [Y] := Sum_of_coord_values [Y] + Y_coord;

             FOR LH := Lo TO Hi DO
              BEGIN
                IF (Box [LH] [X] = 0) THEN Box [LH] [X] := X_coord;
                IF (Box [LH] [Y] = 0) THEN Box [LH] [Y] := Y_coord;
              END;

            Box [Hi] [X] := MAX (Box[Hi][X],X_coord);
            Box [Lo] [X] := MIN (Box[Lo][X],X_coord);
            Box [Hi] [Y] := MAX (Box[Hi][Y],Y_coord);
            Box [Lo] [Y] := MIN (Box[Lo][Y],Y_coord);

          END;
    FOR XY := X TO Y DO
      Center [XY] := ROUND (Sum_of_coord_values [XY] / Num_of_grid_pts);
         This is the formula for center of mass of particle systems

        Center [XY] := ROUND (0.5 * (Box[Lo][XY] + Box[Hi][XY]));
           Take average of extremes; not currently being implemented
  END;

PROCEDURE Widrow_hoff (Correct_g, Actual_g, F  : Vector;
                       N                      : REAL;   Const = 1/N
                   VAR Delta_a                : Matrix);
   Error correction procedure:

   Correc. factor Delta_a := ((Learning const.) * (Correct_g - Actual_g)) x F

  VAR
    Difference_vector, Weighted_vector : Vector;
    Learning_constant                  : REAL;

  BEGIN
    Learning_constant := 1/N;
    Subtract_vectors (Correct_g, Actual_g, Difference_vector);
    Scalar_times_vector (Learning_constant, Difference_vector, Weighted_vector);
    Outer_product (Weighted_vector, F, Delta_a);
  END;

FUNCTION Bool_to_int (Bool : BOOLEAN) : INTEGER;
```



```
   Convert BOOLEAN to INTEGER value (1 or 0)

  BEGIN
    IF Bool = TRUE
      THEN Bool_to_int := 1
      ELSE Bool_to_int := 0;
  END;

FUNCTION Circ_dist (One_dist, Radius : INTEGER) : INTEGER;
  Distance to the circle on a line parallel to the other axis

  BEGIN
    IF (One_dist > Radius)
      THEN Circ_dist := 0
      ELSE Circ_dist := ROUND (SQRT (Radius ** 2 - One_dist ** 2));
  END;

FUNCTION Rad_of_points (Goal_percentage : INTEGER;
                        Data_grid       : Grid;
                        Center          : Point;
                        Num_of_grid_pts : INTEGER) : INTEGER;
  The radius inside of which there are Goal_percentage% of the total number
   of points (Num_of_grid_pts) in the grid (Data_grid)

  VAR
    Goal_pts, Cur_pts, Radius, X_coord, Y_dist, Y_coord : INTEGER;
    Scratch_grid                                        : Grid;

  BEGIN
    Initialize_grid (Scratch_grid);  Grid to mark what's been checked
    Goal_pts := ROUND (Goal_percentage * Num_of_grid_pts / 100);
       Number of points desired
    Radius := 0;   Radius being examined
    Cur_pts := 0;   Number of points inside the current radius
    WHILE (Cur_pts < Goal_pts) DO
      BEGIN
        FOR X_coord := MAX (Center[X]-Radius,0) TO
                       MIN (Center[X]+Radius,Grid_width) DO
          BEGIN
            Y_dist := Circ_dist (X_coord-Center [X], Radius);
            FOR Y_coord := MAX (Center[Y]-Y_dist,0) TO
                           MIN (Center[Y]+Y_dist,Grid_height) DO
              IF Scratch_grid [X_coord,Y_coord] = FALSE THEN
                 If this coordinate has not been checked yet then
                BEGIN
                  Cur_pts := Cur_pts +
                   Bool_to_int (Data_grid [X_coord,Y_coord]);
                  Scratch_grid [X_coord,Y_coord] := TRUE;  Mark as checked
                END;  IF Scratch_grid [X_coord,Y_coord] = FALSE
          END;   X_coord
        Radius := Radius + 1;
      END;   While (Cur_pts < Goal_pts)
    Rad_of_points := Radius;
  END;
```

```
FUNCTION Grid_code (Data_grid : Grid;
                    Col, Row  : INTEGER) : INTEGER;
   Return code corresponding to status of the grid cells including and
   immediately surrounding (Col,Row).  The following table indicates the
   relation between the position of the cell relative to (Col,Row) and the
   power of two to be raised to generate a number to add to the cumulative
   Grid_code:

       2 5 8
       1 4 7
       0 3 6

  VAR
    X_coord, Y_coord, Code : INTEGER;

  BEGIN
    Code := 0;
    FOR X_coord := Col-1 TO Col+1 DO
      FOR Y_coord := Row-1 TO Row+1 DO
        IF ((X_coord > 0) AND (X_coord <= Grid_width) AND
            (Y_coord > 0) AND (Y_coord <= Grid_height))
          THEN
            IF (Data_grid[X_coord,Y_coord] = TRUE)
              THEN Code := Code + 2 ** (Y_coord-Row+1+3*(X_coord-Col+1));
    Grid_code := Code;
  END;

FUNCTION Filled (Data_grid : Grid) : Grid;
  Fill in some gaps in the grid

  VAR
    X_coord, Y_coord, Code : INTEGER;
    New_grid               : Grid;

  BEGIN
    New_grid := Data_grid;
    FOR X_coord := 2 TO (Grid_width-1) DO
      FOR Y_coord := 2 TO (Grid_height-1) DO
        BEGIN
          Code := Grid_code (Data_grid, X_coord, Y_coord);
          IF ((Code = 257) OR (Code = 130) OR (Code = 40) OR (Code = 68)) THEN
            New_grid [X_coord,Y_coord] := TRUE;
        END;
    Filled := New_grid;
  END;

FUNCTION Endpt (Data_grid : Grid;
                Col, Row  : INTEGER) : BOOLEAN;
  TRUE if this is an endpoint

  VAR
    Code, X_coord, Y_coord : INTEGER;

  BEGIN
    Code := Grid_code (Data_grid, Col, Row);
```



```
    IF ((Code =  17) OR (Code =  18) OR (Code =  20) OR (Code =  24) OR
        (Code =  48) OR (Code =  80) OR (Code = 144) OR (Code = 272) OR
        (Code =  22) OR (Code =  52) OR (Code = 304) OR (Code = 400) OR
        (Code = 208) OR (Code =  88) OR (Code =  25) OR (Code =  19))
      THEN Endpt := TRUE
      ELSE Endpt := FALSE;
  END;

FUNCTION Quad (X_coord, Y_coord : INTEGER;
               Center           : Point) : INTEGER;
  Assign a quadrant number:

          2 1
          3 4

  BEGIN
    IF ((X_coord >= Center [X]) AND (Y_coord >= Center [Y])) THEN Quad := 1 ELSE
    IF ((X_coord <  Center [X]) AND (Y_coord >= Center [Y])) THEN Quad := 2 ELSE
    IF ((X_coord <  Center [X]) AND (Y_coord <  Center [Y])) THEN Quad := 3 ELSE
    IF ((X_coord >= Center [X]) AND (Y_coord <  Center [Y])) THEN Quad := 4;
  END;

FUNCTION Ends (Data_grid : Grid;
               Center    : Point) : End_bits;
  Indicate existence of endpoints in each quadrant in the grid

  VAR
    X_coord, Y_coord, Zero : INTEGER;
    Bits                   : End_bits;

  BEGIN
    FOR Zero := 1 TO 4 DO Bits [Zero] := -1;
    FOR X_coord := 2 TO (Grid_width-1) DO
      FOR Y_coord := 2 TO (Grid_height-1) DO
        IF (Endpt (Data_grid, X_coord, Y_coord) = TRUE)
          THEN Bits [Quad(X_coord,Y_coord,Center)] := 1;
            Mark the quadrant if an endpoint exists in that quadrant
    Ends := Bits;
  END;

PROCEDURE Draw_char (Input_grid : Grid);
  Draw a character on the screen (i.e., show screen representation of Grid)

  VAR
    Row, Col      : INTEGER;
    Screen_coord : Two_things;

  BEGIN
    FOR Col := 1 TO Grid_width DO
      FOR Row := 1 TO Grid_height DO
        IF Input_grid [Col,Row] = TRUE THEN
          BEGIN
            Screen_coord [X] := Scrunch (Col,Grid_width,Screen_width);
            Screen_coord [Y] := Scrunch (Row,Grid_height,Screen_height);
            Cursor_posn (Screen_coord[X],Screen_coord[Y]);
```

```
          WRITELN (Out_char[1]);
        END;
END;

PROCEDURE Find_start;
  Used for tracing; this routine was not implemented, and is incomplete

VAR
  Way         : Direction;
  Start       : Two_things;
  Done        : BOOLEAN;
  Start_found : BOOLEAN;

BEGIN
  Way := Up;
  Start [X] := 0;
  Start [Y] := Grid_height;
  Done := FALSE;
  Start_found := FALSE;
  WHILE (Done = FALSE) DO
    BEGIN
      IF (Data_grid [Start[X],Start[Y]] = TRUE) THEN
        BEGIN
          Start_found := TRUE;
          Done := TRUE;
        END
      ELSE IF (Way = Up) THEN
        BEGIN
          IF (Start[X] = Grid_width) THEN
            BEGIN
              IF (Start[Y] = 0) THEN Done := TRUE
              ELSE
                BEGIN
                  Start[Y] := Start[Y] - 1;
                  Way = Down;
                END;
            END  IF Start[X] = Grid_width
          ELSE IF (Start[Y] = Grid_height) THEN
            BEGIN
              Start[X] := Start[X] + 1;
              Way = Down;
            END   IF Start[Y] = Grid_height
          ELSE
            BEGIN
              Start[X] := Start[X] + 1;
              Start[Y] := Start[Y] + 1;
            END;
        END  IF Way = Up
      ELSE IF (Start[Y] = 0) THEN
        BEGIN
          IF (Start[X] = Grid_width) THEN Done := TRUE
          ELSE
            BEGIN
              Start[X] := Start[X] + 1;
              Way = Up;
```



```
            END;
        END  If Start[Y] = 0
      ELSE IF (Start[X] = 0) THEN
        BEGIN
          Start[Y] := Start[Y] - 1;
          Way = Up;
        END  If Start[X] = 0
      ELSE
        BEGIN
          Start[X] := Start[X] - 1;
          Start[Y] := Start[Y] - 1;
        END;
    END;  While Done = FALSE
END;

PROCEDURE Find_start (Data_grid   : Grid;
                      VAR Start_found : BOOLEAN;
                      VAR Start       : Point);
  Used for tracing; this routine was not implemented, and is incomplete

  VAR
    X_coord, Y_coord : INTEGER;

  BEGIN
    Start_found := FALSE;
    Y_coord := 0;
    WHILE ((Y_coord <= Grid_height) AND (Start_found = FALSE)) DO
      BEGIN
        Y_coord := Y_coord + 1;
        X_coord := 1;
        WHILE ((X_coord <= Grid_width) AND (Start_found = FALSE)) DO
          IF (Data_grid[X_coord,Y_coord] = TRUE)
            THEN Start_found := TRUE
            ELSE X_coord := X_coord + 1;
      END;
    Start [X] := X_coord;
    Start [Y] := Y_coord;
  END;

FUNCTION Coord (XY, Old_coord, Way, Factor : INTEGER) : INTEGER;
  Used for tracing; this routine was not implemented, and is incomplete

  VAR
    Theta : REAL;
    Delta : INTEGER;

  BEGIN
    Theta := Pi / 2 * Way;  Angle of the direction
    CASE XY OF
      1 : Delta := ROUND (COS (Theta));
      2 : Delta := ROUND (SIN (Theta));
    END;  Case XY
    Coord := Old_coord + Factor * Delta;
  END;
```

```
PROCEDURE Trace (Data_grid        : Grid;
                 Draw             : BOOLEAN;
                 Num_of_grid_pts  : INTEGER;
             VAR Verticals        : INTEGER;
             VAR Horizontals      : INTEGER;
             VAR Diagonals        : INTEGER;
             VAR Connects         : Connectivity_matrix;
             VAR Num_of_marked    : INTEGER);
  Used for tracing; this routine was not implemented, and is incomplete

  VAR
    Way, Old_way, Way_temp      : Direction;  0 = right; 1 = up;
                                              2 = left;  3 = down
    Old_point, Start            : Point;
    X_coord, Y_coord, Count     : INTEGER;
    X_coord_temp, Y_coord_temp  : INTEGER;
    Marked                      : Grid;
    Start_found                 : BOOLEAN;

  BEGIN
    Verticals := 0;
    Horizontals := 0;
    Diagonals := 0;
    Num_of_marked := 0;
    Initialize_connect_matrix (Connects);
    Find_start (Data_grid, Start_found, Start);
    IF (Start_found = TRUE) THEN
      BEGIN
        X_coord := Start [X];
        Y_coord := Start [Y];
        Initialize_grid (Marked);
        Count := 0;   Number of consecutive times a false element was detected
        Way := 0;   Initial direction = Right

        WHILE (Num_of_marked < Num_of_grid_pts) DO
          BEGIN
            X_coord := Coord (X, X_coord, Way, 1);
            Y_coord := Coord (Y, Y_coord, Way, 1);

            IF (Data_grid [X_coord,Y_coord] = TRUE)
              THEN
                BEGIN
                  Old_point [X] := X_coord;
                  Old_point [Y] := Y_coord;
                  Old_way := Way;
                  IF (Marked [X_coord,Y_coord] = FALSE) THEN
                    BEGIN
                      IF (Draw = TRUE) THEN
                        BEGIN
                          Cursor_posn (Scrunch (X_coord, Grid_width,
                                                Screen_width),
                                       Scrunch (Y_coord, Grid_height,
                                                Screen_height));
                          WRITELN (Out_char[1]);
                        END;
```



```
                    Marked [X_coord,Y_coord] := TRUE;
                    Num_of_marked := Num_of_marked + 1;
                  END;
                Way := Direc (Way + 1);   Turn left
                Count := 0;
              END
            ELSE
              BEGIN
                Count := Count + 1;
                IF (Count <> 3) THEN Way := Direc (Way - 1);
                 IF ((Count < 3)
                  THEN Way := Direc (Way - 1)   Turn right
                  ELSE
                    IF (Count = 4) THEN
                      BEGIN
                        Way := Way - 1;
                        X_coord := Coord (X, X_coord, Way_temp, 2);
                        Y_coord := Coord (Y, Y_coord, Way_temp, 2);

                        Way_temp := Direc (Way - 2);
                        X_coord_temp := Coord (X, X_coord, Way_temp, 2);
                        Y_coord_temp := Coord (Y, Y_coord, Way_temp, 2);
                        Way_temp := Direc (Way_temp + 1);
                        X_coord_temp := Coord (X, X_coord_temp, Way_temp, 2);
                        Y_coord_temp := Coord (Y, Y_coord_temp, Way_temp, 2);
                        IF (Data_grid[X_coord_temp,Y_coord_temp] = TRUE)
                          THEN
                            BEGIN
                              X_coord := Coord (X, X_coord_temp,
                                  Direc (Way_temp+2), 1);
                              Y_coord := Coord (Y, Y_coord_temp,
                                  Direc (Way_temp+2), 1);
                              Way := Way_temp;
                            END
                          ELSE
                            BEGIN
                              X_coord := Old_point [X];
                              Y_coord := Old_point [Y];
                              Way := Direc (Old_way+2);
                            END;

                      END;   If Count = 4
                END;   If Data_grid [X_coord,Y_coord] = FALSE
          END;   While

      END;   If (Start_found = TRUE)
  END;

PROCEDURE Draw_trace (Data_file : String);
  Used for tracing; this routine was not implemented, and is incomplete

  VAR
    Char_to_trace           : CHAR;
    Stored_grid             : Grid;
    Char_found              : BOOLEAN;
```

```
    Verticals, Horizontals : INTEGER;
    Diagonals              : INTEGER;
    Connects               : Connectivity_matrix;
    Num_of_marked          : INTEGER;
    Center                 : Point;
    Num_of_grid_pts        : INTEGER;

  BEGIN
    REPEAT
      Clear_screen;
      REPEAT
        Get_char (1,'Enter character to trace (0=exit): ', Char_to_trace);
      UNTIL (Char_to_trace = '0') OR
            (ORD (Char_to_trace) >= ORD ('A')) OR
            (ORD (Char_to_trace) <= ORD ('Z'));
      IF Char_to_trace <> '0' THEN
        BEGIN
          Initialize_grid (Stored_grid);
          Read_char (Char_to_trace, Data_file, Char_found, Stored_grid);
          IF (Char_found = TRUE) THEN
            BEGIN
              Find_grid_traits (Stored_grid, Center, Num_of_grid_pts);
              Trace (Stored_grid, TRUE, Num_of_grid_pts, Verticals,
                  Horizontals, Diagonals, Connects, Num_of_marked);
              Cursor_posn (1,2);
              WRITELN ('Num_of_marked = ', Num_of_marked:1, ' of ',
                  Num_of_grid_pts:1, '.');
              Wait_for_return;
            END;  { If Char_found = TRUE
        END;
    UNTIL Char_to_trace = '0';
  END;

FUNCTION Ver_segs (Data_grid     : Grid;
                   Center        : Point;
                   Grid_radius   : INTEGER;
                   Factor        : REAL) : Two_things;
  Number of vertical segments to the left and right of center; input is
  analyzed in series of bars

  VAR
    Bar, Bar_count, Bar_width, LH, Bar_max   : INTEGER;
    Min_seg_len, Side, X_coord, Y_coord, Gap : INTEGER;
    Segs, Hi_coord                           : Two_things;
    This_row, Row_count                      : BOOLEAN;

  BEGIN
    Bar_width := 2;
     Min_seg_len := ROUND (Factor * Grid_radius);
    Min_seg_len := 6;
          Minimum vertical segment length
    FOR LH := Lo TO Hi DO Segs [LH] := 0;
    Bar_max := MIN (Grid_width, Center[X]+Grid_radius);
      Max. X-value to check
    Bar := MAX (Center[X]-Grid_radius, 1);   Min. X-value to check
```



```
    Hi_coord [Y] := MIN (Center[Y]+Grid_radius, Grid_height);
      Max. Y-value to check

    WHILE (Bar <= Bar_max) DO
      BEGIN
        Bar_count := 0;  Number of data points in the bar
        Hi_coord [X] := MIN (Bar_max, Bar+Bar_width-1);   Max. X-value in bar
        Y_coord := MAX (Center[Y]-Grid_radius, 1);   Min. Y-value to check
        Gap := 0;    Indication of gaps in the input

        WHILE ((Y_coord <= Hi_coord [Y]) AND (Bar_count < Min_seg_len)) DO
          BEGIN
            X_coord := Bar;
            This_row := TRUE;    Look at the current row in this bar
            Row_count := FALSE;  No points in this row for this bar

            WHILE (This_row = TRUE) DO
              BEGIN
                IF (Data_grid [X_coord,Y_coord] = TRUE)
                  THEN
                    BEGIN
                      Row_count := TRUE;    A point was found in the bar-row
                      This_row := FALSE;    Stop looking at this row
                    END
                  ELSE
                    IF (X_coord < Hi_coord [X])
                      THEN X_coord := X_coord + 1
                      ELSE This_row := FALSE;  Nothing found in this bar-row
              END;

            IF (Row_count = TRUE)
              THEN
                BEGIN
                  Bar_count := Bar_count + 1;   Increment number of points
                                                found
                  Gap := 0;.
                END
              ELSE
                BEGIN
                  Gap := Gap + 1;
                  IF (Gap > 0) THEN
                    BEGIN
                      Bar_count := 0;    A gap was found, so ignore points found
                                         previously
                      Gap := 0;    Reset gap marker
                    END;
                END;    IF Row_count = FALSE

            Y_coord := Y_coord + 1;   Go to the next row
          END;  While ((Y_coord <= Hi_coord [Y]) and
              (Bar_count < Min_seg_len))

        IF (Bar_count < Min_seg_len)
          THEN Bar := Bar + 1   Go to the next bar; i.e., increment X-value by
                                only 1
```

```
          ELSE
            BEGIN
              IF (Bar <= Center[X])
                THEN Side := 1  Declare bar location relative to center
                ELSE Side := 2;
              Segs [Side] := Segs [Side] + 1;  Increment number of segs found
              Bar := Bar + Bar_width;   Go to the next bar; i.e., skip over
                                        columns already looked at
            END;
      END;   While Bar <= Bar_max
    Ver_segs := Segs;
  END;

FUNCTION Hor_segs (Data_grid    : Grid;
                   Center       : Point;
                   Grid_radius  : INTEGER;
                   Factor       : REAL) : Two_things;
  Number of horizontal segments above and below center

  VAR
    Bar, Bar_count, Bar_width, LH, Bar_max   : INTEGER;
    Min_seg_len, Side, X_coord, Y_coord, Gap : INTEGER;
    Segs, Hi_coord                           : Two_things;
    This_col, Col_count                      : BOOLEAN;

  BEGIN
    Bar_width := 2;
     Min_seg_len := ROUND (Factor * Grid_radius);
    Min_seg_len := 7;
          Minimum horizontal segment length
    FOR LH := Lo TO Hi DO Segs [LH] := 0;
    Bar_max := MIN (Grid_height, Center[Y]+Grid_radius);   Max. Y-value to
                                                           check
    Bar := MAX (Center[Y]-Grid_radius, 1);   Min. Y-value to check
    Hi_coord [X] := MIN (Center[X]+Grid_radius, Grid_width);

    WHILE (Bar <= Bar_max) DO
      BEGIN
        Bar_count := 0;   Number of data points in the bar
        Hi_coord [Y] := MIN (Bar_max, Bar+Bar_width-1);
        X_coord := MAX (Center[X]-Grid_radius, 1);
        Gap := 0;

        WHILE ((X_coord <= Hi_coord [X]) AND (Bar_count < Min_seg_len)) DO
          BEGIN
            Y_coord := Bar;
            This_col := TRUE;   Continue to look at the current column
            Col_count := FALSE;   Nothing found yet for this column

            WHILE (This_col = TRUE) DO
              BEGIN
                IF (Data_grid [X_coord,Y_coord] = TRUE)
                  THEN
                    BEGIN
                      Col_count := TRUE;
```



```
                      This_col := FALSE;
                    END
                  ELSE
                    IF (Y_coord < Hi_coord [Y])
                      THEN Y_coord := Y_coord + 1
                      ELSE This_col := FALSE;
              END;

            IF (Col_count = TRUE)
              THEN
                BEGIN
                  Bar_count := Bar_count + 1;   Increment number of points
                                                found in the bar-col
                  Gap := 0;   Reset gap marker
                END
              ELSE
                BEGIN
                  Gap := Gap + 1;
                  IF (Gap > 1) THEN
                    BEGIN
                      Bar_count := 0;   There's a gap, so ignore points found
                                        previously
                      Gap := 0;   Reset gap marker
                    END;
                END;   IF Col_count

            X_coord := X_coord + 1;   Go to the next column
          END;   While ((X_coord <= Hi_coord [X]) and
                 (Bar_count < Min_seg_len))

        IF (Bar_count < Min_seg_len)
          THEN Bar := Bar + 1   Go to the next bar; i.e., increment row by
                                only 1
          ELSE
            BEGIN
              IF (Bar >= Center[Y])
                THEN Side := 1   Declare location as above or below center
                ELSE Side := 2;
              Segs [Side] := Segs [Side] + 1;   Increment number of segs found
              Bar := Bar + Bar_width;   Go to the next bar; i.e., skip over
                                        rows already examined
            END;
      END;   While Bar <= Bar_max
    Hor_segs := Segs;
  END;

FUNCTION Y_intercept (Y_coord : INTEGER;
                      Slope   : REAL;
                      X_coord : INTEGER) : INTEGER;
  Calculate the Y-intercept

  BEGIN
    Y_intercept := ROUND (Y_coord - Slope * X_coord);
  END;
```

```
FUNCTION Diagonals (Data_grid   : Grid;
                    Center      : Point;
                    Grid_radius : INTEGER;
                    Factor      : REAL) : Two_things;
  Number of left- and right-slanting diagonals

  VAR
    Bar, Coord, Bar_count, End_bar, Hi_y     : INTEGER;
    Min_seg_len, One_dist, X_coord, Y_coord  : INTEGER;
    XY, Bar_width, Slopes, Gap               : INTEGER;
    Segs, Y_int                              : Two_things;
    Slope                                    : REAL;
    This_col, Col_count                      : BOOLEAN;
    Y_old, Y_reps                            : INTEGER;

  BEGIN
    Bar_width := 2;
    Min_seg_len := ROUND (Factor * Grid_radius);
          Min. diagonal segment length
    Min_seg_len := 7;
          Min. diagonal segment length
    FOR XY := 1 TO 2 DO Segs [XY] := 0;

    FOR XY := 1 TO 2 DO  Left and right diagonals
      FOR Slopes := 1 TO 2 DO  Check 2 slopes for each type of diagonal
        BEGIN
          CASE Slopes OF
            1 : Slope := XY - 1.5;     Slope = +/- 0.5
            2 : Slope := 2 * XY - 3;   Slope = +/- 1.0
          END;  Case Slopes

          CASE XY OF   Set Y-intercepts
            1 :
              BEGIN
                Y_int [Lo] := Y_intercept (1, Slope, 1);
                Y_int [Hi] := Y_intercept (Grid_height, Slope, Grid_width);
              END;
            2 :
              BEGIN
                Y_int [Lo] := Y_intercept (1, Slope, Grid_width);
                Y_int [Hi] := Y_intercept (Grid_height, Slope, 1);
              END;
          END;  Case XY

          Bar := Y_int [Lo];

          WHILE (Bar <= Y_int [Hi]) DO
            BEGIN
              Bar_count := 0;  Number of data points in the bar
              End_bar := MIN (Y_int[Hi], Bar+Bar_width-1);
              Gap := 0;
              Y_reps := 0;
              Y_old := 0;
              X_coord := 1;
              WHILE ((X_coord <= Grid_width) AND (Bar_count < Min_seg_len)) DO
```



```
              BEGIN
                This_col := TRUE;   Look at the current column
                Col_count := FALSE;   No points found yet
                Y_coord := TRUNC (Slope * X_coord + Bar);
                Hi_y := TRUNC (Slope * X_coord + End_bar);

                WHILE (This_col = TRUE) DO
                  BEGIN
                    IF ((Y_coord > 0) AND (Y_coord <= Grid_height)) THEN
                      IF (Data_grid [X_coord,Y_coord] = TRUE) THEN
                        BEGIN
                          IF (Y_coord = Y_old)
                            THEN Y_reps := Y_reps + 1   We looked at this
                                                        Y-value before
                            ELSE
                              BEGIN
                                Y_old := Y_coord;
                                Y_reps := 1;
                              END;
                          IF (Y_reps < 3) THEN  Point is valid only if we
                                                have not looked at this
                                                Y-value more than twice
                            BEGIN
                              Col_count := TRUE;
                              This_col := FALSE;
                            END;
                        END;   If Data_grid [X_coord,Y_coord] = TRUE
                    IF (This_col = TRUE) THEN
                      IF (Y_coord < Hi_y)
                        THEN Y_coord := Y_coord + 1  Check the next Y-value
                        ELSE This_col := FALSE;
                  END;   While This_col = TRUE

                IF (Col_count = TRUE)
                  THEN
                    BEGIN
                      Bar_count := Bar_count + 1;  Increment the number of
                                                   points found
                      Gap := 0;  Reset gap marker
                    END
                  ELSE
                    BEGIN
                      Gap := Gap + 1;
                      IF (Gap > 0) THEN
                        BEGIN
                          Bar_count := 0;   There's a gap, so ignore points
                                            already found
                          Gap := 0;   Reset gap marker
                        END;
                    END;
                X_coord := X_coord + 1;   Go to the next X-value
              END;  While X_coord <= Grid_width AND Bar_count < Min_seg_len

            IF (Bar_count < Min_seg_len)
              THEN Bar := Bar + 1  Go to the next bar
```

```
                ELSE
                  BEGIN
                    Segs [XY] := Segs [XY] + 1;  Increment number of diags
                                                 found
                    Bar := Bar + Bar_width;  Skip to the next bar
                  END;
              END;  While bar <= Y_int [Hi]
          END;  For Slopes := 1 to 2, For XY := 1 to 2
      Diagonals := Segs;
    END;

FUNCTION Complexity_factor (Complexity : INTEGER) : REAL;
   This was used to discriminate between segment criteria of model data and
   segment criteria of test data; no longer in use

  BEGIN
    CASE Complexity OF
      1 : Complexity_factor := 0.7;
      2 : Complexity_factor := 0.5;
    END;  Case Complexity
  END;

PROCEDURE Encode_line_segments (Data_grid           : Grid;
                                Center              : Point;
                                Grid_radius         : INTEGER;
                                VAR Vertical_segs   : Two_things;
                                VAR Horizontal_segs : Two_things;
                                VAR Diagonal_segs   : Two_things;
                                Complexity          : INTEGER);
   Examine grid for vertical, horizontal, and diagonal segments

  VAR
    Factor : REAL;

  BEGIN
    Factor := Complexity_factor (Complexity);
    Vertical_segs := Ver_segs (Data_grid, Center, Grid_radius, Factor);
    Horizontal_segs := Hor_segs (Data_grid, Center, Grid_radius, Factor);
    Diagonal_segs := Diagonals (Data_grid, Center, Grid_radius, Factor);
  END;

PROCEDURE Print_line_segs (Data_file_1, Data_file_2 : String);
   Output information about segments in model and test data

  VAR
    Char_to_test, Char_match       : CHAR;
    Char_num, Num_of_pts           : INTEGER;
    Char_grid                      : Grid;
    Test_vec, Scratch_vec          : Vector;
    Center                         : Point;
    Cos                            : REAL;
    Multiple_match, Char_found     : BOOLEAN;
    XY, Grid_radius, Filenum       : INTEGER;
    Vertical_segs, Horizontal_segs : Two_things;
    Diagonal_segs                  : Two_things;
```



```
    File_name                      : ARRAY [1..2] OF String;

  BEGIN
    WRITELN ('Files "', Data_file_1, '" and "', Data_file_2, '".');
    File_name [1] := Data_file_1;
    File_name [2] := Data_file_2;
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        Char_to_test := Rec_to_char (Char_num);
        FOR Filenum := 1 TO 2 DO  Model and test data files
          BEGIN
            Read_char (Char_to_test, File_name[Filenum], Char_found, Char_grid);
            IF (Char_found = TRUE)
              THEN
                BEGIN
                  Char_grid := Filled (Char_grid);
                  Find_grid_traits (Char_grid, Center, Num_of_pts);
                  Grid_radius := Rad_of_points (Percent_thresh, Char_grid,
                      Center, Num_of_pts);
                  Encode_line_segments (Char_grid, Center, Grid_radius,
                      Vertical_segs, Horizontal_segs, Diagonal_segs, Filenum);
                  CASE Filenum OF
                    1 : WRITE ('"', Char_to_test, '":');
                    2 : WRITE ('    ');
                  END;  Case Filenum
                  FOR XY := 1 TO 2 DO WRITE (' ', Vertical_segs[XY]:1);
                  FOR XY := 1 TO 2 DO WRITE (' ', Horizontal_segs[XY]:1);
                  FOR XY := 1 TO 2 DO WRITE (' ', Diagonal_segs[XY]:1);
                  WRITELN ('  Grid radius = ', Grid_radius:1);
                END  If character was found
              ELSE WRITELN ('"', Char_to_test, '" not found.');
          END;  Filenum
        WRITELN;
      END;  Char_num
  END;

PROCEDURE Print_endpoints (Data_file_1, Data_file_2 : String);
   Output information about endpoints in model and test data sets

  VAR
    Char_to_test, Char_match       : CHAR;
    Char_num, Num_of_pts           : INTEGER;
    Char_grid                      : Grid;
    Test_vec, Scratch_vec          : Vector;
    Center                         : Point;
    Cos                            : REAL;
    Multiple_match, Char_found     : BOOLEAN;
    XY, Grid_radius, Filenum       : INTEGER;
     Quad : INTEGER;
    Vertical_segs, Horizontal_segs : Two_things;
    Diagonal_segs                  : Two_things;
    File_name                      : ARRAY [1..2] OF String;
    Bits                           : End_bits;

  BEGIN
```

```
    WRITELN ('Files "', Data_file_1, '" and "', Data_file_2, '".  Endpts:');
    File_name [1] := Data_file_1;
    File_name [2] := Data_file_2;
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        Char_to_test := Rec_to_char (Char_num);
        FOR Filenum := 1 TO 2 DO   Model and test data sets
          BEGIN
            Read_char (Char_to_test, File_name[Filenum], Char_found, Char_grid);
            IF (Char_found = TRUE)
              THEN
                BEGIN
                  Char_grid := Filled (Char_grid);
                  Find_grid_traits (Char_grid, Center, Num_of_pts);
                   Grid_radius := Rad_of_points (Percent_thresh, Char_grid,
                       Center, Num_of_pts);
                  Encode_line_segments (Char_grid, Center, Grid_radius,
                       Vertical_segs, Horizontal_segs, Diagonal_segs, Filenum);
                  CASE Filenum OF
                    1 : WRITE ('"', Char_to_test, '":');
                    2 : WRITE ('      ');
                  END;   Case Filenum
                  Bits := Ends (Char_grid, Center);
                  FOR XY := 1 TO 4 DO WRITE (' ', Bits[XY]:2);
                  WRITELN;
                END   If character was found
              ELSE WRITELN ('"', Char_to_test, '" not found.');
          END;   Filenum
        WRITELN;
      END;   Char_num
  END;

PROCEDURE Encode_slice (Data_grid         : Grid;
                        Center            : Point;
                        Num_of_grid_pts   : INTEGER;
                        Theta             : REAL;
                        Grid_radius       : INTEGER;
                        R_values          : Ring_radii;
                        Ring_slice_width  : REAL;
                    VAR Slice             : Slice_code);
  Encode the slice of input

  VAR
    Ring, Radius, R, XY       : INTEGER;
    Angles                    : INTEGER;
    Angle                     : ARRAY [1..2] OF REAL;   The 2 bounding angles
    Check_min, Check_max      : Two_things;   RS grid boundaries
    Check_x, Check_y          : INTEGER;
    RS_corner                 : RSlice;   Corners of the ring-slice (RS)
    Dist_from_ctr             : REAL;   Distance from center to grid point
    Set_bits                  : INTEGER;
    Bits                      : Slice_bits;
    X_coord, Y_coord          : INTEGER;

  PROCEDURE Encode_slice_bits (Bits : Slice_bits;
```



```
                           VAR Code : Slice_code);

    VAR
      IO, Seg, Set_bits : INTEGER;

    BEGIN
      FOR Set_bits := 1 TO 2 * N_rings DO Code [Set_bits] := -1;
         Initialize Slice_code: change 0 to -1 everywhere

      FOR IO := 1 to 2 DO   Code for mid-range & outermost 1's
         Set Slice_code: for innermost 1, set bits before and equal to the
         position to 1.  For example, 00100 becomes 11100.  Do the same for
         outermost 1.  Concatenate bits.  Example: 00101 becomes 11100 + 11111 =
         1110011111.
        BEGIN
          Seg := 2;   Mid-range
          WHILE (Seg < N_rings-1) AND (Bits [Seg] = 0) DO Seg := Seg + 1;
             Seg = Ring-slice number of the innermost 1, excluding the
             outermost ring-slice
          IF Bits [Seg] = 1 THEN
            FOR Set_bits := 1 TO Seg DO Code [Set_bits] := 1;

          Seg := N_rings;   Outermost
          WHILE (Seg > 3) AND (Bits [Seg] = 0) DO Seg := Seg - 1;
             Seg = Ring-slice number of the outermost 1, excluding the
             innermost ring-slice
          IF Bits [Seg] = 1 THEN
            FOR Set_bits := N_rings + 1 TO N_rings + Seg DO
              Code [Set_bits] := 1;
        END;
    END;

  BEGIN

    Angle [1] := Theta;
    Angle [2] := Theta + D_theta;   Bounding angles of the slice to look at
    FOR Set_bits := 1 TO N_rings DO Bits [Set_bits] := 0;   Initialize


    FOR Ring := 1 to N_rings DO
      BEGIN

        FOR Angles := 1 to 2 DO   Find RS-corner offsets from center
          FOR Radius := 1 to 2 DO
            BEGIN
              R := Ring + Radius - 2;
              RS_corner [Angles] [Radius] [X] :=
                ROUND (R_values [R] * COS ( Angle [Angles] ));
              RS_corner [Angles] [Radius] [Y] :=
                ROUND (R_values [R] * SIN ( Angle [Angles] ));
            END;

        FOR XY := X to Y DO   Find RS grid boundaries
          BEGIN
            Check_min [XY] := MIN (RS_corner [1] [1] [XY],
```

```
                                    RS_corner [1] [2] [XY],
                                    RS_corner [2] [1] [XY],
                                    RS_corner [2] [2] [XY]);
          Check_max [XY] := MAX (RS_corner [1] [1] [XY],
                                    RS_corner [1] [2] [XY],
                                    RS_corner [2] [1] [XY],
                                    RS_corner [2] [2] [XY]);
        END;

       FOR Check_x := Check_min[X] TO Check_max[X] DO   Check each grid point
         FOR Check_y := Check_min[Y] TO Check_max[Y] DO
           BEGIN
             Dist_from_ctr := SQRT ( Check_x ** 2 + Check_y ** 2 );
             X_coord := Center [X] + Check_x;
             Y_coord := Center [Y] + Check_y;
             IF (Dist_from_ctr >= R_values [Ring-1])
              AND (Dist_from_ctr <= R_values [Ring])
              AND (X_coord >= 0) AND (X_coord <= Grid_width)
              AND (Y_coord >= 0) AND (Y_coord <= Grid_height) THEN
                 Grid point being checked is in the ring-slice
               IF Data_grid [X_coord,Y_coord] = TRUE THEN
                  There's an input point in the ring-slice
                Bits [Ring] := 1;
           END;   Check each grid point in the ring-slice
      END;   Check each ring
    Encode_slice_bits (Bits,Slice);
  END;

FUNCTION Encode_seg_bits (Segs : INTEGER) : Seg_bits;
   0 becomes [-1,-1,-1]
   1 becomes [ 1,-1,-1]
   2 becomes [ 1, 1,-1]
   3 becomes [ 1, 1, 1]

  VAR
    Bit  : INTEGER;
    Bits : Seg_bits;

  BEGIN
    FOR Bit := 1 TO 3 DO Bits [Bit] := -1;
    IF (Segs > 0) THEN Bits [1] := 1;
    IF (Segs > 1) THEN Bits [2] := 1;
    IF (Segs > 2) THEN Bits [3] := 1;
    Encode_seg_bits := Bits;
  END;

PROCEDURE Encode_input (Data_grid        : Grid;
                        Center           : Point;
                        Num_of_grid_pts  : INTEGER;
                    VAR Vec              : Vector;
                        Complexity       : INTEGER);
   Encode the grid (i.e., produce a vector)

  VAR
    Base, Bits, More : INTEGER;
```



```
    Slice            : INTEGER;
    Theta            : REAL;
    Code             : Slice_code;
    Grid_radius      : INTEGER;
    R_values         : Ring_radii;
    Ring_slice_width : REAL;
    Ring             : INTEGER;
    Vertical_segs    : Two_things;
    Horizontal_segs  : Two_things;
    Diagonal_segs    : Two_things;
    XY, Quad         : INTEGER;
    Segment_bits     : Seg_bits;
    Endpoint_bits    : End_bits;

BEGIN
  Base := 0;
  Initialize_vector (Vec);
  Grid_radius := Rad_of_points (Percent_thresh, Data_grid, Center,
   Num_of_grid_pts);
  R_values [0] := 0;
  Ring_slice_width := Grid_radius / N_rings;
  FOR Ring := 1 TO N_rings DO R_values [Ring] := Ring * Ring_slice_width;
     Set ring radii

  FOR Slice := 1 TO N_slices DO
    BEGIN
      Theta := D_theta * (Slice - 1);
      Encode_slice (Data_grid, Center, Num_of_grid_pts, Theta, Grid_radius,
       R_values, Ring_slice_width, Code);
      FOR Bits := 1 TO 2*N_rings DO Vec [Base+Bits] := Code [Bits];
      Base := Base + 2*N_rings;
    END;

   Encode_line_segments (Data_grid, Center, Grid_radius, Vertical_segs,
    Horizontal_segs, Diagonal_segs, Complexity);

   Endpoint_bits := Ends (Data_grid, Center);

   FOR XY := 1 TO 2 DO
     BEGIN
       Segment_bits := Encode_seg_bits (Vertical_segs [XY]);
       FOR Bits := 1 TO 3 DO Vec [Base+Bits] := Segment_bits [Bits];
       Base := Base + 3;
     END;
   FOR XY := 1 TO 2 DO
     BEGIN
       Segment_bits := Encode_seg_bits (Horizontal_segs [XY]);
       FOR Bits := 1 TO 3 DO Vec [Base+Bits] := Segment_bits [Bits];
       Base := Base + 3;
     END;
    FOR XY := 1 TO 2 DO
      BEGIN
        IF (Diagonal_segs [XY] > 0)
          THEN Vec [Base+1] := 1
          ELSE Vec [Base+1] := -1;
```

```
        base := Base + 1;
      END;

    FOR bits := 1 TO 4 DO
      BEGIN
        FOR More := 1 TO 3 DO Vec [base+More] := Endpoint_bits [Bits];
        base := base + 3;
      END;

    WHILE (Base < Dimensionality) DO
      BEGIN
        WRITELN ('Note: BASE < DIMENSIONALITY');
        Vec [Base+1] := -1;
        base := base + 1;
      END;
  END;

PROCEDURE Store_vec (Char_to_store : CHAR;
                     Vec_to_store  : Vector;
                     Vec_file_name : String);  {Filename of Vec_file}
  {Store a vector in a vector file}

  BEGIN
    OPEN (FILE_NAME         := Vec_file_name,
          FILE_VARIABLE     := Vec_file,
          HISTORY           := UNKNOWN,
          ACCESS_METHOD     := DIRECT,   {Random access}
          RECORD_TYPE       := FIXED,    {Each vector has the same length}
          CARRIAGE_CONTROL  := NONE,     {No special control codes}
          ORGANIZATION      := SEQUENTIAL,
          DISPOSITION       := SAVE);    {Retain the file when it is closed}
    LOCATE (Vec_file, Rec_num (Char_to_store));  {Position file pointer}
    Vec_file^ := Vec_to_store;  {Put vector into file buffer}
    PUT (Vec_file);  {Store the vector}
    CLOSE (Vec_file);
  END;

PROCEDURE Gen_vectors (Data_file    : String;  {Grid file}
                       Store_file   : String);  {Vector file}
  {Generate set of encoded vectors from data set; store the vectors}

  VAR Char_num       : INTEGER;
      Char_to_gen    : CHAR;
      Char_found     : BOOLEAN;
      Char_grid      : Grid;
      Center         : Point;
      Char_vector    : Vector;
      Num_of_points : INTEGER;

  BEGIN
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        WRITELN ('Char_num = ', Char_num:2);
        Char_to_gen := Rec_to_char (Char_num);
        WRITELN ('Char_to_gen = ', Char_to_gen);
```



```
        Read_char (Char_to_gen, Data_file, Char_found, Char_grid);
        WRITELN ('Char_found = ', Char_found);
        IF Char_found = TRUE THEN
          BEGIN
            Char_grid := Filled (Char_grid);
            Find_grid_traits (Char_grid, Center, Num_of_points);
            WRITELN ('Found grid traits.  Center = (', Center[X]:2, ',',
             Center[Y]:2, ')  #pts = ', Num_of_points:2);
            Encode_input (Char_grid, Center, Num_of_points, Char_vector, 1);
            WRITELN ('Storing "', Char_to_gen, '".');
            Store_vec (Char_to_gen, Char_vector, Store_file);
          END;   {If Char_found = TRUE}
      END;   {Char_num}
    WRITELN;
    WRITELN ('DONE WITH Gen_vectors.');
    WRITELN;
  END;

PROCEDURE Read_2_vec_files (Vector_file_1, Vector_file_2    : String;
                        VAR Char_vec_set_1, Char_vec_set_2 : Vector_set1);
  {Read two vector files}

  VAR Char_num                       : INTEGER;
      Char_to_gen                    : CHAR;
      Char_found                     : BOOLEAN;
      Char_vector_1, Char_vector_2 : Vector;

  BEGIN
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        WRITELN ('Char_num = ', Char_num:2);
        Char_to_gen := Rec_to_char (Char_num);
        Read_vec (Char_to_gen, Vector_file_1, Char_found, Char_vector_1);
        IF (Char_found = FALSE) THEN
          BEGIN
            WRITELN ('"', Char_to_gen, '" not found in', Vector_file_1, '.');
            Initialize_vector (Char_vector_1);
          END;
        IF (Vector_file_2 = Vector_file_1)
          THEN Char_vector_2 := Char_vector_1
          ELSE
            BEGIN
              Read_vec (Char_to_gen, Vector_file_2, Char_found, Char_vector_2);
              IF (Char_found = FALSE) THEN
                BEGIN
                  WRITELN ('"', Char_to_gen, '" not found in', Vector_file_2,
                    '.');
                  Initialize_vector (Char_vector_2);
                END;
            END;
        Char_vec_set_1 [Char_num] := Char_vector_1;
        Char_vec_set_2 [Char_num] := Char_vector_2;
      END;   {Char_num}
  END;
```

```
PROCEDURE Gen_outer_prods (Char_vec_set_1, Char_vec_set_2 : Vector_set;
                           VAR Char_mat_set               : Matrix_set);
  Generate outer products; Vector_file_1 as input, Vector_file_2 as output

  VAR Char_num                       : INTEGER;
      Char_vector_1, Char_vector_2 : Vector;

  BEGIN
    WRITELN ('Generating outer products.');
    FOR Char_num := 1 TO N_chars DO
      Outer_product (Char_vec_set_2[Char_num],Char_vec_set_1[Char_num],
       Char_mat_set[Char_num]);
    WRITELN ('Done generating outer products.');
  END;

PROCEDURE Associate_vec_sets (Char_vec_set_1 : Vector_set;
                              Char_vec_set_2 : Vector_set;
                              B              : Matrix;   Old matrix
                          VAR A              : Matrix);  New matrix
  Associate vector sets

  VAR
    Pair, Pikd    : INTEGER;
    Gprime        : Vector;
    Delta_a       : Matrix;
    Char_mat_set : Matrix_set;

  BEGIN
    A := B;   Start with the old matrix
    Gen_outer_prods (Char_vec_set_1, Char_vec_set_2, Char_mat_set);
    WRITELN('Number of presentations per char = ', Present:6, '.');
    WRITELN('Learning constant = 1/[F,G].');
    WRITELN ('Beginning association.');
    FOR Pair := 1 TO (Present * N_chars) DO
      BEGIN
        Pikd := Rand_int (N_chars);  Pick a vector randomly
        Add_Matrices (A,Char_mat_set[Pikd],A);
        Matrix_times_vector (A,Char_vec_set_1[Pikd],Gprime);
        Widrow_hoff (Char_vec_set_2[Pikd],Gprime,Char_vec_set_1[Pikd],
         Inner_product(Char_vec_set_1[Pikd],Char_vec_set_2[Pikd]),Delta_a);
        Add_matrices (A,Delta_a,A);
      END;
  END;

PROCEDURE Associate_vec_files (Vector_file_1, Vector_file_2 : String;
                               Old_matrix                   : Matrix;
                           VAR New_matrix                   : Matrix);
  Associate vectors in vector files

  VAR
    Char_vec_set_1, Char_vec_set_2 : Vector_set;

  BEGIN
    Read_2_vec_files (Vector_file_1, Vector_file_2,
     Char_vec_set_1, Char_vec_set_2);
```



```
    Associate_vec_sets (Char_vec_set_1, Char_vec_set_2, Old_matrix, New_matrix);
  END;

PROCEDURE Store_matrix (Mat_to_store  : Matrix;
                        Mat_file_name : String);
  Store a matrix

  VAR
    Row, Col : INTEGER;

  BEGIN
    WRITELN ('Storing matrix.');
    OPEN (FILE_NAME          := Mat_file_name,
          FILE_VARIABLE      := Mat_file,
          HISTORY            := UNKNOWN,
          ACCESS_METHOD      := SEQUENTIAL,
          RECORD_TYPE        := FIXED,   Each matrix has the same length
          CARRIAGE_CONTROL   := NONE,    No special control codes
          ORGANIZATION       := SEQUENTIAL,
          DISPOSITION        := SAVE);   Retain the file when it is closed
    EXTEND (Mat_file);
    FOR Row := 1 TO Dimensionality DO
      FOR Col := 1 TO Dimensionality DO
        BEGIN
          Mat_file := Mat_to_store [Row,Col];   Put element into file buffer
          PUT (Mat_file);   Store the matrix
        END;
    CLOSE (Mat_file);
    WRITELN ('Done storing matrix.');
  END;

PROCEDURE Read_matrix (Mat_file_name : String;
                   VAR Mat_to_read   : Matrix);
  Read a matrix

  VAR
    Row, Col : INTEGER;

  BEGIN
    WRITELN ('Reading matrix.');
    OPEN (FILE_NAME          := Mat_file_name,
          FILE_VARIABLE      := Mat_file,
          HISTORY            := UNKNOWN,
          ACCESS_METHOD      := SEQUENTIAL,
          RECORD_TYPE        := FIXED,   Each matrix has the same length
          CARRIAGE_CONTROL   := NONE,    No special control codes
          ORGANIZATION       := SEQUENTIAL,
          DISPOSITION        := SAVE);   Retain the file when it is closed
    FOR Row := 1 TO Dimensionality DO
      FOR Col := 1 TO Dimensionality DO
        BEGIN
          IF (Row = 1) AND (Col = 1)
            THEN RESET (Mat_file)
            ELSE GET (Mat_file);
          Mat_to_read [Row,Col] := Mat_file;
```

```
      END;
    CLOSE (Mat_file);
    WRITELN ('Done reading matrix.');
  END;

PROCEDURE BSB (In_vec  : Vector;
               A       : Matrix;
           VAR Out_vec : Vector);
   brain-state-in-a-box processing

  VAR
    Cos, Alpha, Gamma                      : REAL;
    Iterate, X                             : INTEGER;
    Old_vec, Decay, Feedback, Temp, Sum : Vector;

  FUNCTION Limit (B : Vector) : Vector;
    Set maximums for element values

    VAR
      C      : Vector;
      Thresh : REAL;
      X      : INTEGER;

    BEGIN
      C := B;
      Thresh := 1.5;
      FOR X := 1 to Dimensionality DO
        IF C [X] > Thresh THEN
          C [X] := Thresh ELSE
        IF C [X] < -1 * Thresh THEN
          C [X] := -1 * Thresh;
      Limit := C;
    END;

  BEGIN
    Alpha := 0.2;   Feedback
    Gamma := 0.9;   Decay
    Iterate := 25;
    X := 0;
    Cos := 0;
    Out_vec := In_vec;
    WHILE (Cos < 1) AND (X < Iterate) DO
      BEGIN
        X := X + 1;
        Old_vec := Out_vec;
        Scalar_times_vector (Gamma,Out_vec,Decay);
        Matrix_times_vector (A,Out_vec,Temp);
        Scalar_times_vector (Alpha,Temp,Feedback);
        Add_Vectors (Decay,Feedback,Sum);
        Out_vec := Limit (Sum);
        Cos := Vector_cosine (Old_vec,Out_vec);
      END;
  END;

PROCEDURE Find_match (Test_vec      : Vector;
```



```
                      Char_vec_set   : Vector_set;
                  VAR Char_match     : CHAR;
                  VAR Cos            : REAL;
                  VAR Multiple_match : BOOLEAN);
   Find the closest match between the model character set and the test vector
   Test_vec.  Cos is an indicator of the quality of the match.

  VAR
    Test_cos             : REAL;
    Test, Char_match_num : INTEGER;

  BEGIN
    Cos := 0;   The current largest cosine between vectors
    Test := 0;   Vector number
    Char_match_num := 0;
    Char_match := ' ';
    Multiple_match := FALSE;   TRUE if more than one match is found

    Message (1, 'Comparing test vector with model vectors.');
    FOR Test := 1 TO N_Chars DO
      BEGIN
        Test_cos := Vector_cosine (Test_vec,Char_vec_set[Test]);
        IF ((1-Test_cos = 1-Cos) AND (Cos <> 0)) THEN
          If the quality of this match is the same as the quality of some
          previous match, then:
          Multiple_match := TRUE ELSE
        IF ((1-Test_cos) < (1-Cos)) THEN
          If the quality of this match is better than the quality of all
          previous matches, then:
          BEGIN
            Multiple_match := FALSE;
            Char_match_num := Test;
            Cos := Test_cos;
          END;
      END;   Test
    IF (Char_match_num > 0) THEN Char_match := Rec_to_char (Char_match_num);
  END;

PROCEDURE Draw_scales;
   Draw scales for screen editing

  BEGIN
    Message (Screen_height-2, '          1         2         3         4');
    WRITELN ('  1234567890123456789012345678901234567890');
    WRITELN (' 1                1');
    WRITELN (' 2                2');
    WRITELN (' 3                3');
    WRITELN (' 4                4');
    WRITELN (' 5                5');
    WRITELN (' 6                6');
    WRITELN (' 7                7');
    WRITELN (' 8                8');
    WRITELN (' 9                9');
    WRITELN ('1012345678901234 ');
    WRITELN (' 1');
```

```
    WRITELN (' 2');
    WRITELN (' 3');
    WRITELN (' 4');
    WRITELN (' 5');
  END;

PROCEDURE Read_bitpad (VAR Data_grid : Grid);
  Read the BITPAD

  VAR
    B_input                              : String;  Input string
    C1, C2, button                       : INTEGER;
    Junk                                 : CHAR;  Dummy variable
    Screen_coord, Grid_coord, Pad_coord : Two_things;

  BEGIN
    Button := 0;
    Cursor_posn (3,Screen_height-3);
    REPEAT
      READLN (Pad_in, b_input);   Read BITPAD
      IF LENGTH (B_input) > 4 THEN  Min. len. of valid input = 5
        BEGIN
          C1 := Instr ( 2, B_input, Comma);   Locate 2 commas
          C2 := Instr ( C1 + 2, B_input, Comma);
          IF (C1 > 1) AND (C2 > C1+1) THEN   Requirements for valid input
            BEGIN
              READV (B_input, Pad_coord[X], Junk, Pad_coord[Y], Junk, Button);
                Parse input to assign coordinates
              IF Button <> Blue THEN
                BEGIN
                  Grid_coord [X] := Scrunch (Pad_coord[X],Pad_width,Grid_width);
                  Grid_coord [Y] :=
                   Scrunch (Pad_coord[Y],Pad_height,Grid_height);
                  Screen_coord [X] :=
                   Scrunch (Pad_coord[X],Pad_width,Screen_width);
                  Screen_coord [Y] :=
                    Scrunch (Pad_coord[Y],Pad_height,Screen_height);
                  Cursor_posn (Screen_coord[X],Screen_coord[Y]);
                  IF Button = White THEN  Mark
                    BEGIN
                      Data_grid [Grid_coord[X],Grid_coord[Y]] := TRUE;
                      WRITE(Out_char[1])
                    END ELSE
                  IF Button = Green THEN   Erase
                    BEGIN
                      Data_grid [Grid_coord[X],Grid_coord[Y]] := FALSE;
                      WRITE (Out_char[0]);
                    END;
                  Cursor_posn (Screen_coord[X],Screen_coord[Y]);
                END;  IF Button <> Blue
            END;   IF (C1 > 1) AND (C2 > C1+2)
        END;   IF LENGTH(B_input) > 4
    UNTIL Button = Blue;
  END;
```



```
PROCEDURE Get_some_char (Char_to_get    : CHAR;
                         VAR Data_grid  : Grid;
                             Show_scale : BOOLEAN);
  Get a character from the BITPAD

  VAR
    Done       : BOOLEAN;
    Done_input : CHAR;

  PROCEDURE Char_message (Char_to_get : CHAR);

    BEGIN
      IF Char_to_get <> ' ' THEN
        BEGIN
          Cursor_posn (0,Screen_height);
          WRITELN ('This is the "',Char_to_get,'" character.');
        END;
    END;

  BEGIN
    Done := FALSE;
    WRITELN (Pad_out, Bit_rate);   Set up BITPAD for switched-stream input
    REPEAT
      BEGIN
        Message (2, 'Yellow: position cursor; white: mark; green: unmark;');
        Message (1, ' blue: exit.  Begin.');
        IF (Show_scale = TRUE) THEN Draw_scales;
        Char_message (Char_to_get);
        Draw_char (Data_grid);
        Read_bitpad (Data_grid);
        IF (Show_scale = TRUE) THEN
          BEGIN
            Clear_screen;
            Char_message (Char_to_get);
            Draw_char (Data_grid);
          END;
        Clear_line (2);
        REPEAT
          Get_char (1,'Are you done with this letter (y/n)? ',Done_input);
        UNTIL (Done_input = 'Y') OR (Done_input = 'N');
        IF (Done_input = 'Y') THEN Done := TRUE;
      END
    UNTIL Done = TRUE;
    WRITELN (Pad_out, 'S');   Stop reading from BITPAD
    Cursor_posn (0,1);
  END;

PROCEDURE Get_new_char (VAR Data_grid : Grid);
  Get a new character from the BITPAD

  BEGIN
    Clear_screen;
    Get_some_char (' ', Data_grid, FALSE);
  END;
```

```
PROCEDURE Print_vec_to_file (Actual_char : CHAR;
                             Letter      : CHAR;
                             Vec         : Vector;
                             Filename    : String);
  Output vector to file

  VAR
    Slice, Bit     : INTEGER;
    Counter        : INTEGER;

  BEGIN
    OPEN (FILE_NAME         := Filename,
          FILE_VARIABLE     := Print_file,
          HISTORY           := UNKNOWN,
          ACCESS_METHOD     := SEQUENTIAL,
          RECORD_TYPE       := VARIABLE,
          CARRIAGE_CONTROL  := LIST,
          ORGANIZATION      := SEQUENTIAL,
          DISPOSITION       := SAVE);  Retain the file when it is closed
    EXTEND (Print_file);  Append to the file
    WRITELN (Print_file, 'Actual: "', Actual_char, '"; Match: "',
     Letter, '":');
    Counter := 0;
    FOR Slice := 1 TO Dimensionality DO
      BEGIN
        WRITE (Print_file, ' ', Vec[Slice]:5:2);
        Counter := Counter + 1;
        IF (Counter >= Group) THEN
          BEGIN
            WRITELN (Print_file);
            Counter := 0;
          END;
      END;  Slice
    WRITELN (Print_file, '-------------------------------------------------');
    CLOSE (Print_file);
  END;

PROCEDURE BSB_vectors (Vecs_filename : String);
   BSB vectors in file

  VAR
    Char_num    : INTEGER;
    Char_vector : Vector;
    Char_to_gen : CHAR;
    Char_found  : BOOLEAN;
    BSB_vector  : Vector;
    In_vector   : Vector;

  BEGIN
    Read_matrix (Auto_file, Auto_matrix);
    WRITELN ('Reading and BSBing ', Vecs_filename, '.');
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        Char_to_gen := Rec_to_char (Char_num);
        Read_vec (Char_to_gen, Vecs_filename, Char_found, Char_vector);
```



```
        IF (Char_found = FALSE) THEN
          BEGIN
            WRITELN ('"', Char_to_gen, '" not found.');
            Initialize_vector (Char_vector);
          END;
         Matrix_times_vector (Auto_matrix, Char_vector, In_vector);
          This was used to delay the non-linearity (i.e., BSB)
        In_vector := Char_vector;
        BSB (In_vector, Auto_matrix, BSB_vector);
        Store_vec (Char_to_gen, BSB_vector, BSB_vec_file);
        Print_vec_to_file (Char_to_gen, ' ', BSB_vector, BSB_output_file);
      END;  Char_num
  END;

PROCEDURE Read_BSB_vectors (BSB_filename : String);
   Read BSB vectors from a file

  VAR
    Char_num     : INTEGER;
    Char_vector  : Vector;
    Char_to_read : CHAR;
    Char_found   : BOOLEAN;
    BSB_vector   : Vector;

  BEGIN
    WRITELN ('Reading BSB vectors.');
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        Char_to_read := Rec_to_char (Char_num);
        Read_vec (Char_to_read, BSB_filename, Char_found, BSB_vector);
        IF (Char_found = FALSE) THEN
          BEGIN
            WRITELN ('"', Char_to_read, '" not found.');
            Initialize_vector (BSB_vector);
          END;
        BSB_vec_set [Char_num] := BSB_vector;
      END;  Char_num
    BSB_done := TRUE;
  END;

PROCEDURE Print_BSB_vectors;
  Read and print BSB vectors

  VAR
    Char_num : INTEGER;

  BEGIN
    Read_BSB_vectors (BSB_vec_file);
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        WRITELN ('BSB_vec_set[', Char_num:2, ']:');
        Print_vec (BSB_vec_set[Char_num]);
      END;
  END;
```

```
PROCEDURE Test_one_input (Vecs_filename : String);
  Test (i.e., find match for) one new input from the BITPAD

  VAR
    Char_grid       : Grid;
    Test_vec        : Vector;
    Center          : Point;
    Num_of_points   : INTEGER;
    Char_match      : CHAR;
    Cos             : REAL;
    Scratch_vec     : Vector;
    Multiple_match  : BOOLEAN;
    In_vector       : Vector;

  BEGIN
    IF (BSB_done = FALSE) THEN   Read BSB vectors once
      BEGIN
       Read_matrix (Auto_file, Auto_matrix);
       Read_BSB_vectors (Vecs_filename);
      END;
    Initialize_grid (Char_grid);
    Get_new_char (Char_grid);   Read new character from BITPAD
    Clear_line (2);
    Message (1, 'Processing test input.');
    Find_grid_traits (Char_grid, Center, Num_of_points);
    Encode_input (Char_grid, Center, Num_of_points, Test_vec, 2);
    Message (1, 'Finding a match.');
     Matrix_times_vector (Auto_matrix, Test_vec, In_vector);
       This was used to delay the non-linearity; not currently in use
    In_vector := Test_vec;
    BSB (In_vector, Auto_matrix, Scratch_vec);
    Test_vec := Scratch_vec;
    Find_match (Test_vec, BSB_vec_set, Char_match, Cos, Multiple_match);
    Clear_line (1);
    Clear_line (2);
    WRITELN ('Match is "', Char_match, '"; quality (cos) is ', Cos:6, '.');
    IF (Multiple_match = TRUE) THEN
     WRITELN ('Another match of equal quality was found.');
    Print_vec_to_file ('?', Char_match, Test_vec, 'TEST.VEC');
    Wait_for_Return;
  END;

PROCEDURE Test_all_input (Vecs_filename : String;
                          Grid_filename : String);
  Test (i.e., find matches for) all test data from test file

  VAR
    Char_to_test    : CHAR;
    Char_num        : INTEGER;
    Char_grid       : Grid;
    Test_vec        : Vector;
    Center          : Point;
    Num_of_points   : INTEGER;
    Char_match      : CHAR;
    Cos             : REAL;
```



```
    Scratch_vec     : Vector;
    Multiple_match  : BOOLEAN;
    Char_found      : BOOLEAN;
    In_vector       : Vector;

  BEGIN
    IF (BSB_done = FALSE) THEN
      BEGIN
       Read_matrix (Auto_file, Auto_matrix);
       Read_BSB_vectors (Vecs_filename);
      END;
    Initialize_grid (Char_grid);
    Message (1, 'Processing test characters.');
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        Char_to_test := Rec_to_char (Char_num);
        Read_char (Char_to_test, Grid_filename, Char_found, Char_grid);
        IF (Char_found = TRUE)
          THEN
            BEGIN
              Char_grid := Filled (Char_grid);  Fill in gaps in input
              Find_grid_traits (Char_grid, Center, Num_of_points);
              Encode_input (Char_grid, Center, Num_of_points, Test_vec, 2);
               Matrix_times_vector (Auto_matrix, Test_vec, In_vector);
                 This was used to delay the non-linearity; not currently in
                 use
              In_vector := Test_vec;
              BSB (In_vector, Auto_matrix, Scratch_vec);
              Test_vec := Scratch_vec;
              Find_match (Test_vec, BSB_vec_set, Char_match, Cos,
               Multiple_match);
              WRITE ('Actual: "', Char_to_test, '"; Match is "',
               Char_match, '"; quality (cos) is ', Cos:6, '.');
              IF (Char_to_test = Char_match)
                THEN WRITELN
                ELSE WRITELN (' ***');  Mark the mismatches
              IF (Multiple_match = TRUE) THEN
               WRITELN ('Another match of equal quality was found.');
              Print_vec_to_file (Char_to_test, Char_match, Test_vec,
               Match_file);
            END  If character was found
          ELSE WRITELN ('"', Char_to_test, '" not found.');
      END;    Char_num
  END;

PROCEDURE Print_char_set (Data_file : String);
  Print a character set (visual representation) to a file

  VAR
    Filename        : String;
    Char_num        : INTEGER;
    Char_to_print   : CHAR;
    Char_grid       : Grid;
    Char_found      : BOOLEAN;
    Row, Col        : INTEGER;
```

```
  BEGIN
    Clear_screen;
    Get_input (10,'Enter filename to print to: ', Filename);
    OPEN (FILE_NAME         := Filename,
          FILE_VARIABLE     := Print_file,
          HISTORY           := UNKNOWN,
          ACCESS_METHOD     := SEQUENTIAL,
          RECORD_TYPE       := VARIABLE,
          CARRIAGE_CONTROL  := LIST,
          ORGANIZATION      := SEQUENTIAL,
          DISPOSITION       := SAVE);  { Retain the file when it is closed }
    EXTEND (Print_file);   { Append to the file }
    FOR Char_num := 1 TO N_chars DO
      BEGIN
        Char_to_print := Rec_to_char (Char_num);
        Read_char (Char_to_print, Data_file, Char_found, Char_grid);
        IF Char_found = TRUE THEN
          BEGIN
            FOR Row := Grid_height DOWNTO 1 DO
              BEGIN
                FOR Col := 1 TO Grid_width DO
                  IF Char_grid [Col,Row] = TRUE
                    THEN WRITE (Print_file, Out_char[1])
                    ELSE WRITE (Print_file, Out_char[0]); { Draw row }
                WRITELN (Print_file); { Go to the next row }
              END;   { Row }
            WRITELN (Print_file);   { Leave a blank line between characters }
          END;   { IF Char_found = TRUE }
      END;   { Char_num }
    CLOSE (Print_file);
  END;

PROCEDURE Edit_char_set (Set_type  : String;
                         Data_file : String;
                         Scales    : BOOLEAN);
  { Edit a character set, using the BITPAD }

  VAR
    Char_to_edit : CHAR;
    Stored_grid  : Grid;
    Char_found   : BOOLEAN;

  BEGIN
    REPEAT
      Clear_screen;
      Message (2, Set_type);
      REPEAT
        Get_char (1,'Enter character to edit (0=exit): ', Char_to_edit);
      UNTIL (Char_to_edit = '0') OR
            (ORD (Char_to_edit) >= ORD ('A')) OR
            (ORD (Char_to_edit) <= ORD ('Z'));
      IF Char_to_edit <> '0' THEN
        BEGIN
          Initialize_grid (Stored_grid);
```



```
          Read_char (Char_to_edit, Data_file, Char_found, Stored_grid);
          Get_some_char (Char_to_edit, Stored_grid, Scales);  { Edit }
          Store_char (Char_to_edit, Stored_grid, Data_file);
        END;
    UNTIL Char_to_edit = '0';
  END;

PROCEDURE Initialize;
  { One-time initial routines, like securing BITPAD; get terminal type }

  BEGIN
    Clear_screen;
    REPEAT
      Get_char (10, 'Is output directed to a VT220 (y/n) ?  ', VT220);
    UNTIL (VT220 = 'Y') OR (VT220 = 'N');
    IF (VT220 = 'Y') THEN
      BEGIN
        OPEN (Pad_in, 'bitpad$IT');
        RESET (Pad_in);
        OPEN (Pad_out, 'bitpad$IT');
        REWRITE (Pad_out);
      END;

    Out_char [0] := ' ';  { Screen character representing 0 }
    Out_char [1] := '*';  { Screen character representing 1 }
  END;

PROCEDURE Cleanup;
  { Final procedures, like dumping BITPAD }

  BEGIN
    IF (VT220 = 'Y') THEN
     BEGIN
       CLOSE (Pad_in);
       CLOSE (Pad_out);
     END;
  END;

PROCEDURE Learn_char_sets (Num_of_sets : INTEGER);
  { Learn character sets (either model, or model+test) }

  VAR
    Char_vec_set : Vector_set;
    Char_mat_set : Matrix_set;
    Old_matrix   : Matrix;

  BEGIN
    WRITELN('Dimensionality = ', Dimensionality:1, '.');
    WRITELN;
    Gen_vectors (Char_file, Vecs1_file);   { Encode model characters }
    Initialize_matrix (Old_matrix);
    Associate_vec_files (Vecs1_file, Vecs1_file, Old_matrix, Auto_matrix);
       { Autoassociate model vectors }
    Store_matrix (Auto_matrix, Auto_file);
    BSB_vectors (Vecs1_file);   { In the present implementation, the BSB
```

```
                                non-linearity is introduced here
    IF (Num_of_sets > 1) THEN   Now learn test data (maybe)
      BEGIN
        Read_matrix (Auto_file, Auto_matrix);
        Old_matrix := Auto_matrix;
        Gen_vectors (Test_file, Vecs2_file);  Encode test characters
        Associate_vec_files (Vecs2_file, Vecs1_file, Old_matrix, Auto_matrix);
           Associate test vectors with model vectors
        Store_matrix (Auto_matrix, Auto_file);
      END;
  END;

PROCEDURE Learn_test_set;
  Learn only the test set of data

  VAR
    Char_vec_set : Vector_set;
    Char_mat_set : Matrix_set;
    Old_matrix   : Matrix;

  BEGIN
    WRITELN('Dimensionality = ', Dimensionality:1, '.');
    WRITELN;
    Gen_vectors (Test_file, Vecs2_file);
    Initialize_matrix (Old_matrix);
    Associate_vec_files (Vecs2_file, Vecs2_file, Old_matrix, Auto_matrix);
       Autoassociate test vectors
    Store_matrix (Auto_matrix, Auto_file);
    BSB_vectors (Vecs2_file);
  END;

PROCEDURE Main;
  Process user's commands

  VAR
    Command : CHAR;

  BEGIN
    Initialize;
    REPEAT
      Clear_screen;
      Message (Screen_height, 'Welcome.');
      WRITELN;
      WRITELN ('Commands:');
      WRITELN;
      WRITELN (' M  Edit model character set');
      WRITELN (' P  Print model character set to a file');
      WRITELN (' S  Print line segments for model & test vectors');
      WRITELN (' R  Print endpoints for model & test vectors');
       WRITELN (' C  Draw a trace of a stored test character');
        This is used in conjunction with the tracer; not implemented
      WRITELN (' E  Edit test character set');
      WRITELN (' D  Print test character set to a file');
      WRITELN;
      WRITELN (' T  Test one character for recognition (screen input)');
```



```
      WRITELN (' A  Test all characters for recognition (file input)');
      WRITELN (' 1  Encode input, learn, & BSB for model characters');
      WRITELN (' 2  Encode input, learn, & BSB for model+test characters');
      WRITELN (' X  Encode input & learn for test characters only');
      WRITELN (' Q  Quit this system');
      WRITELN;
      Get_char (2, 'Enter command: ', Command);
      CASE Command OF
        'A' : Test_all_input (BSB_vec_file, Test_file);
        'C' : Draw_trace (Test_file);
        'D' : Print_char_set (Test_file);
        'E' : Edit_char_set ('Test set', Test_file, FALSE);
        'M' : Edit_char_set ('Model set', Char_file, TRUE);
        'P' : Print_char_set (Char_file);
        'R' : Print_endpoints (Char_file, Test_file);
        'S' : Print_line_segs (Char_file, Test_file);
        'T' : Test_one_input (BSB_vec_file);
        '1' : Learn_char_sets (1);
        '2' : Learn_char_sets (2);
        'X' : Learn_test_set;
        'Y' : Test_all_input (Vecs2_file, Test_file);
        'Q' : ;  Do nothing
        OTHERWISE ;  Do nothing
      END;
    UNTIL Command = 'Q';
    Cursor_posn (0,0);
    Cleanup;
  END;

BEGIN
  Main;
END.
```