| RPI Image Processing Laboratory COMPUTER DOCUMENTATION | Bulletin No. U175 | Rev. 1.0 | Page 1 |
|---|---|---|---|

TITLE:

Raster-To-Vector Conversion with A Vector Ordering Post-process

ABSTRACT:

Described are programs for raster-to-vector conversion and vector ordering. Applications exist in the areas of photo-interpretation, cartography, data-processing, raster scanning, image processing, map-data-processing and computer vision.

DESCRIPTION:

To run the vector conversion program on the Prime computer type:
SEG UNSPSOFT>XY2VEC
and
SEG UNSPSOFT>VEC2ORDER

XY2VEC reads in a file of X-Y positions in the following form:

```
I I
I I
.
.
.
```

Where the 'I' is an integer which ranges from 0..511.
The output is a file called VEC which consists of vector data formatted as follows:

```
I I I I
```

Where the first 2 'I's are the X and Y coordinate of the head of the vector and the second 2 'I's are the X and Y coordinate of the tail of the vector.
When VEC2ORDER is run the vectors in 'VEC' are ordered into a file called 'ORDER'. The ordering attempts to minimize the path length traveled by a pen plotter. See the program VEC2TRANS to output the vectors to the pen plotter. VEC2TRANS may be released in user bulletin form at a later date but is currently installed in the UNSPSOFTware.

The following document exists on line in the file:
UNSPSOFT>DOC>XY2VEC

[Use additional sheets if necessary]

| Prepared by: Douglas Lyon | Date: 5/22/87 | Total Pages: 41 |
|---|---|---|

PL-2

## Table of Contents

ABSTRACT....................................................2

ACKNOWLEDGMENT..............................................4

INTRODUCTION................................................5

LITERATURE SURVEY...........................................6

EXPERIMENTAL RESULTS........................................8

RASTER-TO-VECTOR CONVERSION.................................10

ORDERING VECTORS............................................13

TECHNIQUES_FOR_IMPROVEMENT..................................17

CONCLUSION..................................................19

BIBLIOGRAPHY................................................20

XY2VEC.PASCAL...............................................22

VEC2ORDER.PASCAL............................................27

FIGURES.....................................................31

# ABSTRACT

This paper describes real implementations of raster-to-vector and vector-ordering algorithms. The programs have been coded for clarity and portability, they are not optimized for machine efficiency. These programs were designed for processing small numbers of vectors and their implementation shows them to be O(N**2). When N is small, the quadratic nature of the shown implementation can be ignored. Several suggestions have been made for speed up of this class of algorithm for large N [Nagy 1980]. When N is small, however, these suggestion appear to slow down the algorithm. Constant factors in the algorithm must not be overlooked (esp. with small N), and program size and complexity will increase with the incorporation of the more sophisticated techniques. No attempt is made at line thinning or in eliminating redundant points. X-Y coordinates are assumed to appear in scanline order. Techniques for edge detection and line thinning are not addressed.

Keywords:

Photo-interpretation, cartography, data-processing, raster scanning, image processing, map-data-processing, and computer vision.

ACKNOWLEDGMENT

The author is happy to acknowledge the following assitance received during the course of paper creation.

To Professor Randolf Franklin for providing the incentive to write this paper as a term project in his Computational Geometry course.

To Tom DeWitt, who supplied edge detected images for figures one and two.

To Geoff Huebner, who supplied the edge detection algorithm and images for figures three and five.

To all those I have forgotten to mention.

Thanks!

## INTRODUCTION

The purpose of this paper is to demonstrate a raster-to-vector algorithm with a vector-ordering post process. Several examples are shown and data collected from them indicates they run in O(N**2) time. The scope of this paper is limited to descriptions of the algorithms, an attempt to place them in the context of current research, and suggestions for improvement. No noise reductions techniques are used, except the throwing out of single pixels which cannot complete a vector.

# LITERATURE SURVEY

Raster-to-vector conversion is a form of scan conversion and is typically divided into 3 stages:: line thinning, line extraction, and topology reconstruction.

The algorithm presented here makes no assumptions about line thinning (the process of reducing lines to unit thickness); therefore, the topic of line thinning is not addressed. When the algorithm presented is given lines which are not unit thickness, many parallel vectors result.

Line extraction is the process of building vector information from X-Y pixel positions. This is what the raster-to-vector algorithm does. Line intersections cause multiple vectors to result with this algorithm.

Vector ordering is a form of topology reconstruction. In topology resonstruction, line segments are joined to minimize plot time and to approximate the original image.

Line-following algorithms tend to be a simpler class of vector ordering algorithms but are usually slower [Boyle, 1984] [Gibson and Lenzmeir, 1981].

It is hard to evaluate other systems for their raster-to-vector software. At an ASP/ASSM convention in March 1982, Interaction Systems Corporation demonstrated a product which appeared promising, but no quantitative details are known. Intergraph is also known as a developer of this type of software, also proprietary. Other vendors known to be working in the field are, Laser Scan (a line following approach) and Environmental Research Technology, Inc. [Crane 1981]. Again no details are available.

# EXPERIMENTAL RESULTS

The following tables summarize experimental results obtained from real data processed by the raster-to-vector and vector-ordering algorithms. A graphical summary can be seen in Figure 7.

In the summary of results below, N is the number of vectors, raster-to-vector and ordering are conversion times in CPU seconds, Ordered Plot and Unordered Plot are plot times in real seconds.

| Figure | N | Raster-to-Vector | Ordering | Ordered Plot | Unordered Plot |
|---|---|---|---|---|---|
| 1 | 212 | 67 | 10 | 40 | 44 |
| 2 | 387 | 177 | 25 | 56 | 56 |
| 3 | 1273 | 1293 | 100 | 128 | 168 |
| 4 | 1179 | 4039 | 189 | 208 | 252 |
| 5 | 2417 | 5060 | 448 | 246 | 392 |
| 6 | 912 | 1231 | 94 | 124 | 198 |

These result, when plotted against a normalized N**2 result, seem to ver
the N**2 nature of the proposed algorithms.

## RASTER-TO-VECTOR CONVERSION

The following is a description of an algorithm for raster-to-vector conversion. The input to the program is a file of pixel positions (X-Y coordinates). The output from the program is a file of vectors.

The main portion of the program is divided into 3 parts: initialization (garbage in), point_processing (garbage compaction) and printing data (garbage out). During the initialization phase an array (list_array) of vector type (more on this later) is initialized to be empty. During the point processing phase, a point is read in and compared to the list_array (initially empty). In each comparison an attempt is made to establish the point as a member of an existing vector. A point is a member of an existing vector if it can adopt the position of being at the head or tail of that vector without causing the vector to change slope beyond a slope tolerance. Slope tolerance is established by the user of

the program and is a critical parameter. Slope tolerance affects how far a vector can deviate from the initial slope of the vector before a new point is not accepted.

What follows is a p-code summary of the above description.

```
for each_point in the_list_of_points do
  stash_point
```

What stash_point does is it tries to stash a point in an existing list of vectors and if it can't it starts a new vector.

```
point_not_stashed := true;
try_to_stash_a_point
if point_not_stashed then new_point
```

Try_to_stash_a_point checks for the number of vectors being equal to zero, if so, it tries to place the point in the list of vectors.

```
if number_of_vectors = 0 then new_point
   else
     try_to_put_in_vectors
```

Try_to_put_in_vectors is the place where optimization techniques seem most promising. It examines every vector and checks to see if the point can be placed there.

```
for each_vector in the_list_of_vectors do
   if point_not_stashed then try_to_put_in_head
   if point_not_stashed then try_to_put_in_tail
```

The act of trying to place a point in a head or tail of a vector is performed in constant time by checking to see if the point is adjacent and the new slope is acceptable. If this is true then the looping over the vectors can be ceased.

If there are N points to place and N vectors in which to place them, the algorithm can take O(N**2) time if all the points must be placed in N vectors (worse case).

# ORDERING VECTORS

The following is a description of the vector ordering algorithm. This algorithm takes as input a file of vectors (called vec) and produces as output a file of ordered vectors (called order).

The vectors are ordered with an eye towards minimizing the plot time. The algorithm takes the form of a distance minimization algorithm; that is, for a given vector to be plotted (called the current output vector), the list of input vectors is searched for a vector with an end point a minimum distance away from the tail of the current output vector. This next vector is then copied from the input vector list to the next position on the output vector list and becomes the new output vector. This approach requires that an input list of N vectors be searched N times. This means that the time of a search for N vectors is O(N**2). Literature on the subject seems to indicate that the problem is analogous to the Traveling Salesman problem (identifying the problem as being NP-complete).

Evaluation must not stop at running time, however, and techniques for speeding up such an algorithm seem moot if the algorithm does not yield a desirable result in the first place. The vectors are sorted to minimize total path length of a steered drawing device. For a constant speed plotter, we can thus conclude that the total plot time should see some reduction. If it does not then it is not worth even considering a method for running time reduction of the ordering algorithm. Evaluation of plot time can be performed theoretically, but in the areas of experimental computer science it seems fashonable to test an implementation on experimental data. Both techniques are seemingly appropriate, but the appearance is deceiving when the input data is coming in scan line order (as the xy2vec algorithm produces them). Thus, plot time in scanline order wastes a "flyback" of the pen (much the same as a carriage return on a printer) and may be sped up using the obvious technique of reversing the order of every other scanline. Thus the plotting order would no longer be left to right, top to bottom but right to left, left to right, in top to bottom order. Such fixes seem like experimental hacks at best (such is the nature of experimental computer science). The trade off seems to be between CPU time and plotting time (with the cost of software development being a one time capital intensive venture). Since the CPU time costs are spiralling downward (outpacing plotting speed increases) it seems only fair that the CPU should pitch in a bit to help the plotter along.

How does the Program work?

The main program is divided into 3 stages: "garbage in" (initialize), processing (process_vectors), and "garbage out" (write_vectors). The "garbage in" phase (initialize) opens a file for read access (called vec) and opens a file for write access (called order). It reads in all the vectors into an array called 'input_array' and sets the number_of_vectors to the number of vectors in the input array. The input_array is an array of input_type records which contains the locations of the vector end points and a boolean flag which indicates whether the vector is eligible (more on this later).

Process_vectors checks to see if there is only 1 vector in the input_array. If there is only 1 vector, it is a singularity; the vector is moved to the first position in the output_array, and the procedure terminates. If there is more than one vector in the input_array, the process_many_vectors procedure is called.

Process_vectors keeps two pointers: one called current_input_vector, another called next_output_vector. Current_input_vector is a pointer into the input_array and identifies the current input vector to be considered. Next_output_vector is a pointer to the output_array and moves down the output_array in an incremental fashion. The procedure is best described by the P-code which follows:

```
For next_output_vector := 1 TO number_of_vectors DO
  BEGIN
    move(current_input_vector, next_output_vector);
    current_input_vector := next_closest_vector_head(next_out
put_vector);
  END
```

The idea here is that a move(1,2) places vector 1 on the input_array into position 2 of the output_array. In addition, move marks the vector in the input_array as being ineligible for further processing (hence the boolean flag). The next_closest_vector head is a function which returns the vector in the input_array which has an end point nearest the tail of the current output vector. Next_closest_vector_head looks at the distance from the output vectors tail to both the head and tail of the current_input_vector and uses the minimum. If the tail is the minimum of the 2 end-points, then next_closest_vector_head flips the vector around so that the head and tail of the vector (its orientation) are reversed.

Next_closest_vector head relies on 2 helping functions for calculating distance: distance_from_input_vectors_head_to_output_vectors_tail and distance_from_input_vectors_tail_to_output_vectors_tail. These actually calculate the distance squared (rather than the square root of the sum of the differences) since these formulas yield the same criterion.

## TECHNIQUES FOR IMPROVEMENT

The following techniques for improvement seem promising but have not been tried by the author.

Raster-to-vector conversion may be reduced to a constant time algorithm if the conversion is to unit length vectors. A single pass chain generating algorithm has been shown to work in O(S) time, where S is the number of scan-lines in an image [Chakravarty 1981, 1982]. Here the length of each vector is 2 pixels long and arrived at by convolving a 3x3 matrix in raster order to obtain chains in raster order. At this point outlines may be found and quickly removed. This greatly reduces the amount of data to be processed by the ordering phase [Nitzan and Agin 1979].

The vector-ordering algorithm may see excellent speed increase with the introduction of an adaptive grid [Franklin 1984]. Here each vector formed is entered into an array which is indexed by cell. Each cell is an element in a grid which is superimposed on the image frame. This enables access to vectors by relative position within a grid, and this greatly improves vector ordering performance. Starting with a vector in one cell, the ordering of searching for other vectors would be in the same cell first,

adjacent cells next. Since there is strong coherence in the data (not randomly distributed, as Franklin suggests), the conversion time analysis will be case sensitive.

Only a practical implementation with a study of experimental results can result in a conclusive analysis of the proposed algorithm. It seems however, quite promising.

# CONCLUSION

Two algorithms were shown which convert raster information from a scanned picture into ordered vectors. Provisions have been made to obtain conversions which are not exact but which can greatly reduce the number of vectors. Time for conversion has been shown to be O(N**2) by theoretical analysis and by experimental data. The suggested improvements make raster-to-vector conversion constant time and greatly reduce vector-ordering time. While research shows that proprietary algorithms exist, which may well accomplish the tasks described, the algorithms shown here were developed independantly and thus place raster-to-vector and vector-ordering in the public domain.

# BIBLIOGRAPHY


Antell, R.E. 1983. "The Laser-Scan Fastrak Automatic Digitizing System," Proceedings, Auto-Carto V.

Barrett, R.C., and B.W. Jordan. 1974. "Scan Conversion Algorithms for a Cell Orgainized Raster Display." Communications of the ACM. 17 (March):157-63

Chakravarty, I. 1981 "A single-pass chain generating algorithm for region boundaries", Computer Graphics and Image Processing, (15), February 1981, pp. 182 - 193.

Chakravarty, I. 1982. The Use of Characteristic Views As A Basis For Recognition Of Three-Dimensional Objects Ph.D. Thesis for Computer and Systems Engineering Department at Rensselaer Polytechnic Institute, Troy, NY.

Franklin, W.R. 1979. "Evaluation of Algorithms to Display Vector Plots on Raster Devices." Computer Graphics and Image Processing 11: 377-90.

Franklin, W.R. 1984. "Adaptive Grids for Geometric Operations", Auto-Carto Six Edited by David H. Douglas, University of Toronto Press, Canada.

Gibson, L., and C. Lenzmeirer. 1981. "A Hierarchical Pattern Extraction System for Hexagonally Sampled Images." Unpublished report prepared for the Air Force Office of Scientific Research by Interactive Systems Corporation.

Nagy, G. 1980. "What Is a 'Good' Data Structure for 2-D Points?" Map Data Processing Edited by Freeman, H. and Pieroni, G., Academic Press, Inc., NY, NY.

Nitzan, D. and G.J. Agin, "Fast methods for finding object outlines", Computer Graphics and Image Processing, 9, (1), Jan. 1979.

Peuquet, D. and A.R. Boyle. 1984. Raster Scanning, Processing and Plotting of Cartographic Documents. SPAD Systems, Ltd., Williamsville, NY.

Prager, J.M. "Extracting and labelling boundary segments in natural scenes", IEEE Trans. on Pattern Analysis and Machine Intelligence, PAMI-2, (1), Jan. 1980.

Tamura, Hideyuki. 1978. "A Comparison of Line Thinning Algorithms from a Digital Geometry Standpoint." Proceedings of the Fourth Internationsl Joint Conference on Pattern Recognition, Kyoto, Japan. Pages 715-719.

XY2VEC.PASCAL

```
PROGRAM xy2vec(INPUT, OUTPUT);

CONST
  max_number_of_vectors = 2560;

TYPE

   slope_type = RECORD
   {This represents the initial slope of a vector}
    dy,dx : REAL;
   END;

   point_type = RECORD
    x, y : 0 .. 511;
   END;

   vector_type = RECORD
      initial_slope : slope_type; {This is installed when tail is
installed}
      head          : point_type; {Head is installed first}
      tail          : point_type;
   {Both tail an head are subject to change}
      tail_is_empty : BOOLEAN;
   END;



VAR
  list_array          : ARRAY [1..max_number_of_vectors] OF
                                                 vector_type;
  input_file          : TEXT;
  output_file         : TEXT;
  dy_new, dy_old      : REAL;
  dx_new, dx_old      : REAL;
  X                   : INTEGER;
  Y                   : INTEGER;
  index               : INTEGER;
  number_of_vectors   : INTEGER; {number of vectors in list_array}
  point_not_stashed   : BOOLEAN;
                       {flags the placement of a point into store}

PROCEDURE initialize;
VAR
  index : INTEGER;
BEGIN
{write('calling initialize');}
REWRITE(output_file, 'vec');
RESET(input_file, 'xy');
```

```
PROCEDURE find_dy_dx(i : INTEGER);
BEGIN
 WITH list_array[i] DO
  BEGIN
   IF tail_is_empty THEN
     writeln('***ERROR, dx_dy being calculated on an empty tail');
   dy_new := head.y - y;
   dx_new := head.x - x;
   dy_old := initial_slope.dy;
   dx_old := initial_slope.dx;
  END; {with}
END; {find_dy_dx}

FUNCTION slope_close_enough : BOOLEAN;
BEGIN
{writeln('calling slope_close_enough');}
slope_close_enough := FALSE;
  IF (dx_new = 0) AND (dx_old = 0) THEN slope_close_enough := TRUE
      ELSE
        BEGIN
         IF NOT ((dx_new = 0) OR (dx_old = 0)) THEN
           slope_close_enough :=
               (dy_new / dx_new) = (dy_old / dx_old) ;
        END; {else}
END; {slope_close_enough}

FUNCTION slope_acceptable(i :INTEGER) : BOOLEAN;
BEGIN
  slope_acceptable := FALSE;
  IF list_array[i].tail_is_empty THEN slope_acceptable := FALSE
   ELSE
    BEGIN
      find_dy_dx(i);
      IF slope_close_enough THEN  slope_acceptable := TRUE
      ELSE slope_acceptable := FALSE;
    END; {ELSE}
END; {slope_acceptable}

PROCEDURE try_to_put_in_head(i : INTEGER);
BEGIN
  WITH list_array[i] DO
   BEGIN
     IF is_adjacent(head.x,head.y,x,y) AND
         slope_acceptable(i) THEN
        install_head(i);
   END; {with}
END; {try_to_put_in_head}
```

```
PROCEDURE try_to_put_in_tail(i : INTEGER);
BEGIN
  WITH list_array[i] DO
   BEGIN
     IF (is_adjacent(tail.x,tail.y,x,y) AND slope_acceptable(i))
           OR
        (tail_is_empty AND is_adjacent(head.x,head.y,x,y) )
    THEN install_tail(i)
   END; {WITH}
END; {try_to_put_in_tail}

PROCEDURE increment number_of_vectors;
BEGIN
  number_of_vectors := number_of_vectors + 1;
  IF number_of_vectors > MAX_NUMBER_OF_VECTORS THEN
    WRITELN
     ('The number of vectors has been exceeded..run continued.');
END; { increment_number_of_vectors}

PROCEDURE new_point; {this is used for each new vector}
BEGIN
   point_not_stashed := FALSE;
   increment_number_of_vectors;
   WITH list_array [number_of_vectors] DO
   BEGIN
     head.x := x;
     head.y := y;
     tail_is_empty := TRUE;
   END; {with list_array}
END; {new_point}


PROCEDURE try_to_put_in_vectors;  {number_of_vectors <> 0 and not
a new point}
VAR
  index : INTEGER;
BEGIN
{writeln('calling try_to_put_in_vectors');}
  FOR index := 1 TO number_of_vectors DO {for all vectors }
   BEGIN
    IF point_not_stashed THEN  try_to_put_in_head(index);
    IF point_not_stashed THEN  try to_put_in_tail(index);
   END; {FOR}
END; {try_to_put_in_vectors}

PROCEDURE try_to_stash_a_point; {verify}
BEGIN
{writeln('calling try_to_stash_a_point');}
  IF number_of_vectors = 0
   THEN  new_point {verified}
   ELSE try_to_put_in_vectors; {verify}
END; {try_to_stash_a_point}
```

```
PROCEDURE stash_point; {verify}
BEGIN
  READLN(input_file, x, y);
  point_not_stashed := TRUE;
  try_to_stash_a_point; {verify}
  IF point_not_stashed
     THEN
       BEGIN
         point_not_stashed := FALSE;
         new_point; {verify}
       END; {IF}
END; {stash_point}

PROCEDURE print_out_data;
VAR
  index : INTEGER;
BEGIN
WRITELN('the number of vectors is ',number_of_vectors:1);
  FOR index := 1 to number_of_vectors DO
    WITH list_array[index] DO
      IF NOT tail_is_empty THEN
      writeln(output_file,
              head.x:1, ' ', head.y:1,' ',tail.x:1,' ',tail.y:1);
END; {print_out_data}

BEGIN  {main portion of code }
initialize;
WHILE NOT EOF(input_file) DO
   stash_point; {verify}
print_out_data;
CLOSE(output_file);
CLOSE(input_file);
END.
```

```
Program vec2order2(Input, Output);

CONST
  max_number_of_vectors = 5000;
  some_big_number = 1000000;

TYPE

  point_type = RECORD
  x, y : 0..511;
  END;

  vector_type = RECORD
  head       :point_type;
  tail       :point_type;
  END;

  input_type = RECORD
  head       :point_type;
  tail       :point_type;
  eligible :boolean;
  END;

VAR
  output_array         :ARRAY [1..max_number_of_vectors] OF
                                                          vector_ty
pe;
  input_array          :ARRAY [1..max_number_of_vectors] OF
                                                           input_ty
pe;
  input_file           :TEXT;
  output_file          :TEXT;
  x                    :INTEGER;
  y                    :INTEGER;
  index                :INTEGER;
  number_of_vectors    :INTEGER;

PROCEDURE reverse_input_vector(input_vector : INTEGER);
{reverses the head and tail of a vector in the input array}
VAR temp : INTEGER;
BEGIN
   WITH input_array[input_vector] DO
    BEGIN
      temp := head.x;
      head.x := tail.x;
      tail.x := temp;
      temp := head.y;
      head.y := tail.y;
```

```
      tail.y := temp;
    END; {with}
END; {reverse_input_vector}

PROCEDURE write_vectors;
{writes the vectors in the output array to the output_file}
VAR i : INTEGER;
BEGIN
  FOR i := 1  TO number_of_vectors DO
    WITH output_array[index] DO
      writeln(output_file, head.x, head.y, tail.x, tail.y);
  index := number_of_vectors;
END;

PROCEDURE initialize;
BEGIN
  REWRITE(output_file, 'order');
  RESET(input_file, 'vec');
  number_of_vectors := 1;
  WHILE NOT EOF(input_file) DO
    WITH input_array[number_of_vectors] DO
      BEGIN
        eligible := TRUE;
        number_of_vectors := number_of_vectors + 1;
        READLN(input_file, head.x, head.y, tail.x, tail.y);
      END;
  writeln('Read in ', number_of_vectors:1, ' vectors.');
  number_of_vectors := number_of_vectors - 1;
  {The number of vectors will never be negitive}
END; {initialize}

PROCEDURE move(from, fin :INTEGER);
{Move a vector from the input array to the output array}
BEGIN
  input_array[from].eligible := FALSE;
  WITH output_array[fin] DO
    BEGIN
      head.x := input_array[from].head.x;
      head.y := input_array[from].head.y;
      tail.x := input_array[from].tail.x;
      tail.y := input_array[from].tail.y;
    END;
END;

FUNCTION distance_from input_vectors_head_to_output_vectors_tail
          (i, o :INTEGER) :INTEGER;
VAR
          dx, dy :INTEGER;
BEGIN
  WITH output_array[o] DO
    BEGIN
      IF (input_array[i].eligible)
        THEN
          BEGIN
```

```
            dx := tail.x - input_array[i].head.x;
            dy := tail.y - input_array[i].head.y;
          distance_from_input_vectors_head_to_output_vectors_tail
              := dx * dx + dy * dy;
          END {IF..THEN}
        ELSE
              distance_from_input_vectors_head_to_output_vectors_t
ail
                                                      := -1;
      END; {IF}
END; {distance_from_input_vectors_head_to_output_vectors_tail}

FUNCTION distance_from_input_vectors_tail_to_output_vectors_tail
          (i, o :INTEGER) :INTEGER;
VAR
  dx, dy :INTEGER;
BEGIN
  WITH output_array[o] DO
    BEGIN
      IF (input_array[i].eligible)
       THEN
         BEGIN
            dx := tail.x - input_array[i].tail.x;
            dy := tail.y - input_array[i].tail.y;
            distance_from_input_vectors_tail_to_output_vectors_tai
l :=
               dx * dx + dy * dy;
         END {IF..THEN}
       ELSE distance_from_input_vectors_tail_to_output_vectors_tai
l := -1;
      END; {if}
END; {distance_from input_vectors_tail_to_output_vectors_tail}

FUNCTION next_closest_vector_head (base_output_vector :INTEGER) :I
NTEGER;
{look at an output vector and find the closest input vector, if th
e side
 of the input vector which is closest is the tail, flip it around}
VAR
  minimum_distance_so_far, d, d2, current_vector_candidate :INTEGE
R;
BEGIN
  minimum_distance_so_far := some_big_number;
  FOR current_vector candidate := 1 TO number_of_vectors DO
    BEGIN
      D := distance_from_input_vectors_head_to_output_vectors_tail
(
                            current_vector_candidate, base_output_
vector);
      D2
         := distance_from_input_vectors_tail_to_output_vectors_tai
l(
                            current_vector_candidate, base_output_
vector);
```

```
      IF (d < minimum_distance_so_far) AND ((d > 0) OR (d = 0) )
        THEN
          BEGIN
            minimum_distance_so_far := d;
            next_closest_vector head := current_vector_candidate;
          END {THEN}
        ELSE
          BEGIN
            IF (d2 < minimum_distance_so_far) AND ((d2 > 0) OR
                                   (d2.= 0) )
              THEN
                BEGIN
                  minimum_distance_so_far := d2;
                  next_closest_vector_head := current_vector candi
date;
                  reverse_input_vector(current_vector_candidate);
                END; {IF}
          END; {ELSE}
     END; {FOR}
END; {next_closest_vector_head}

PROCEDURE process_many_vectors;
VAR
  current_input_vector, next_output_vector :INTEGER;

BEGIN
  current_input_vector := 1;
  FOR next_output_vector := 1 TO number_of_vectors DO
    BEGIN
      move(current_input_vector, next_output_vector);
      current_input_vector :=
             next_closest_vector head( next_output_vector);
    END; {for}
END; {process_many_vectors}

PROCEDURE process_vectors;
BEGIN
  IF number_of_vectors = 1
    THEN move(1,1)
    ELSE process_many_vectors;
END; {process_vectors}

{MAIN}
BEGIN
  initialize;
  process_vectors;
  write_vectors;
END. {main}
```

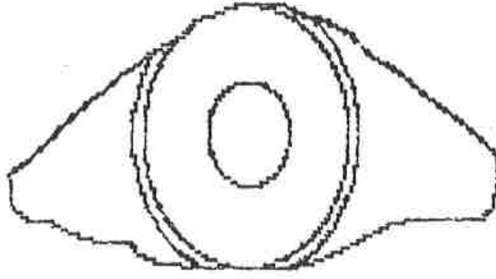


Fig. 1

Fig. 2

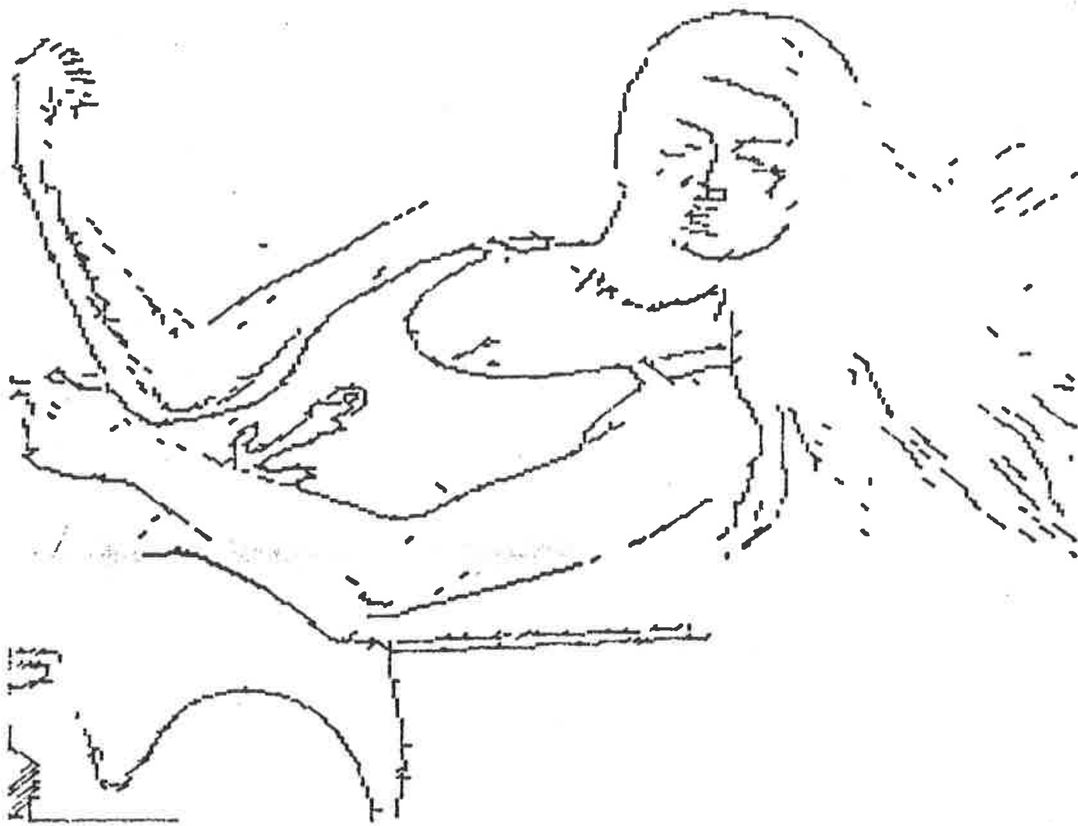
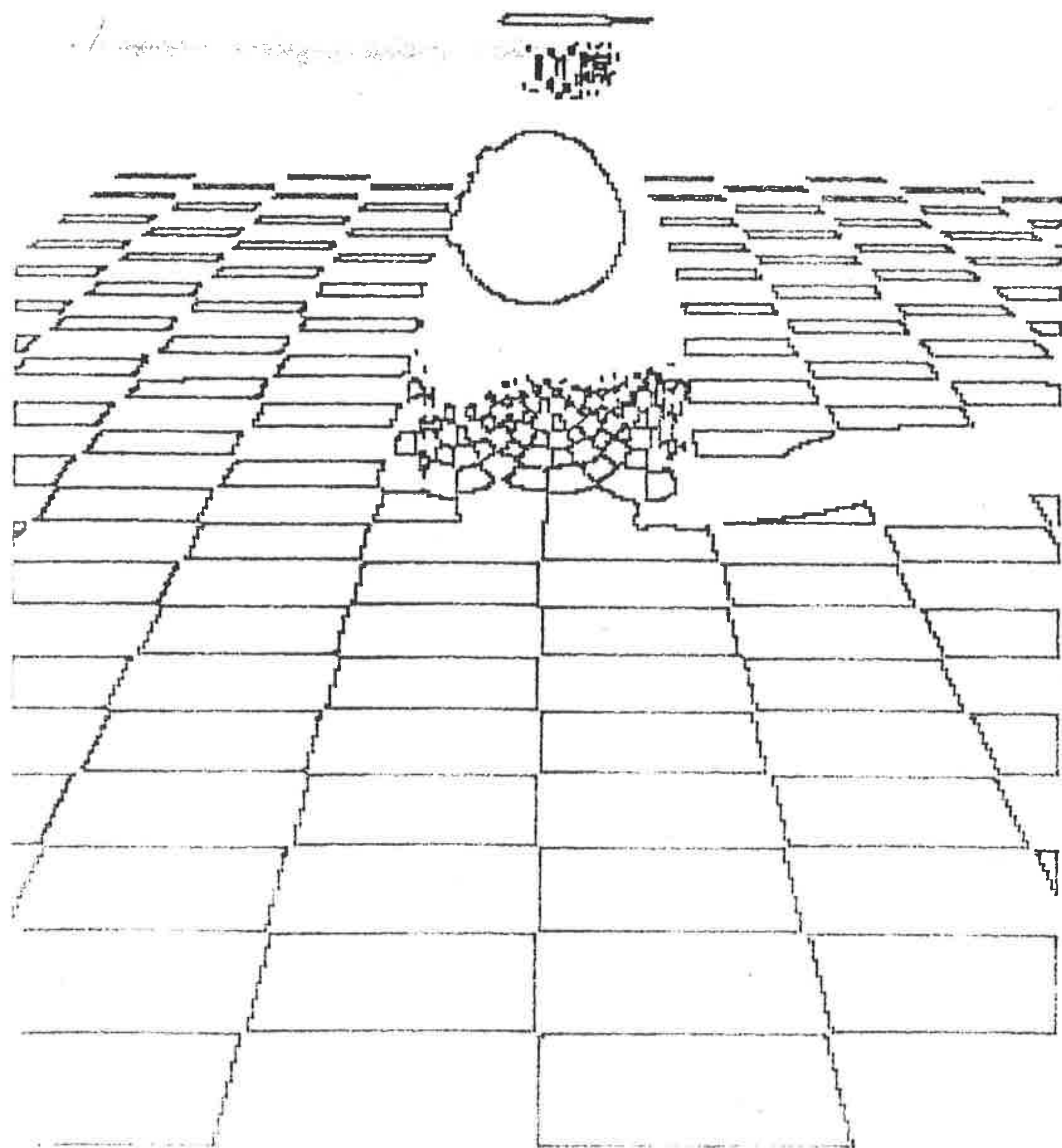


Fig. 3

Fig. 4



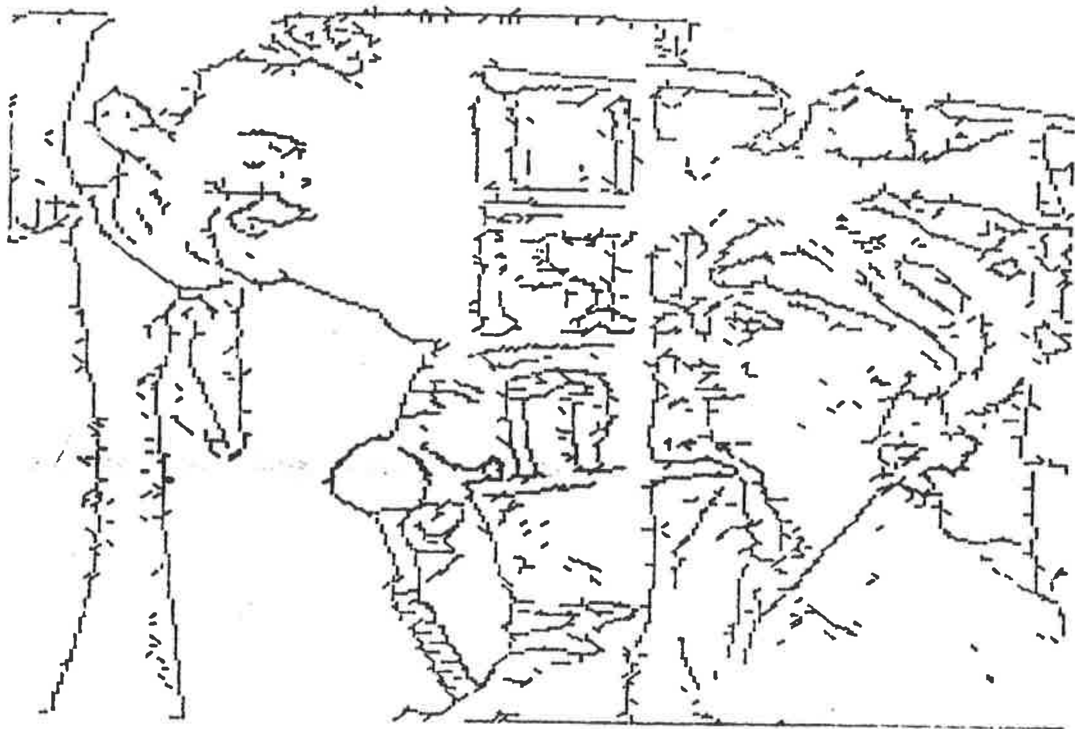
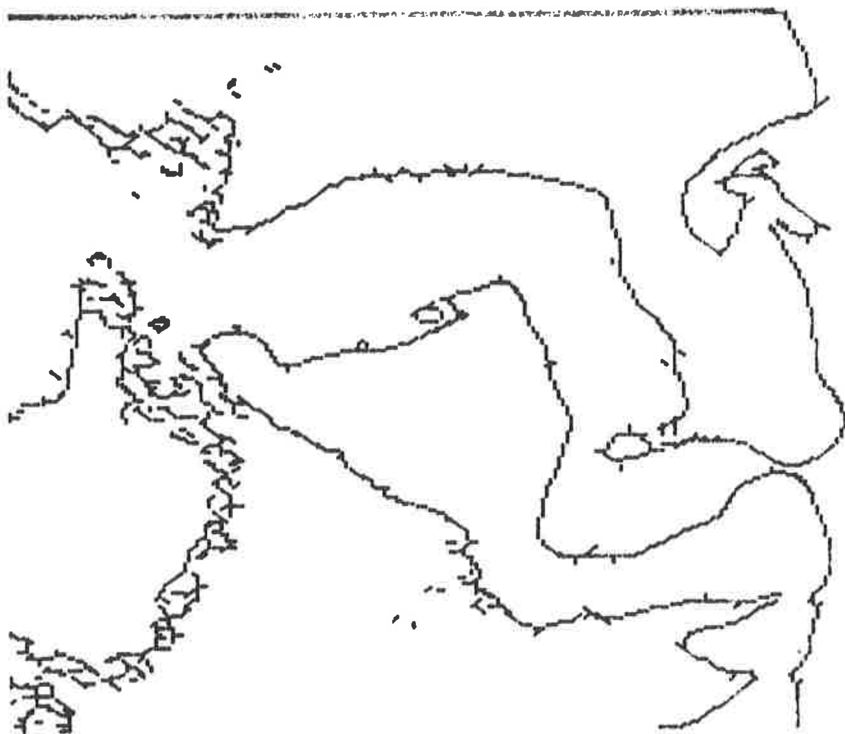


Fig. 5

Fig. 6



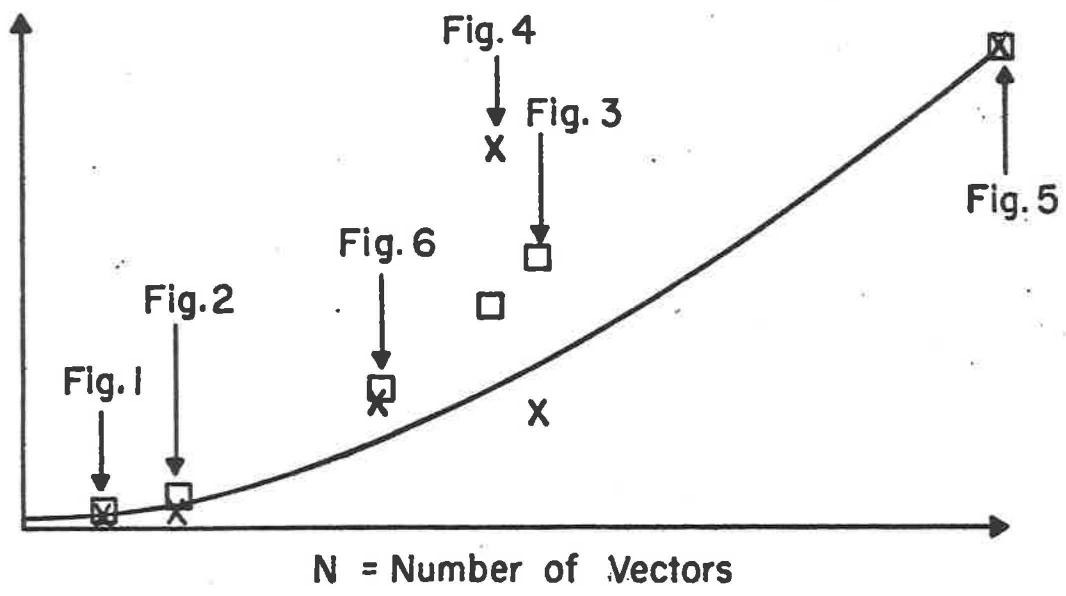



Fig. 7A

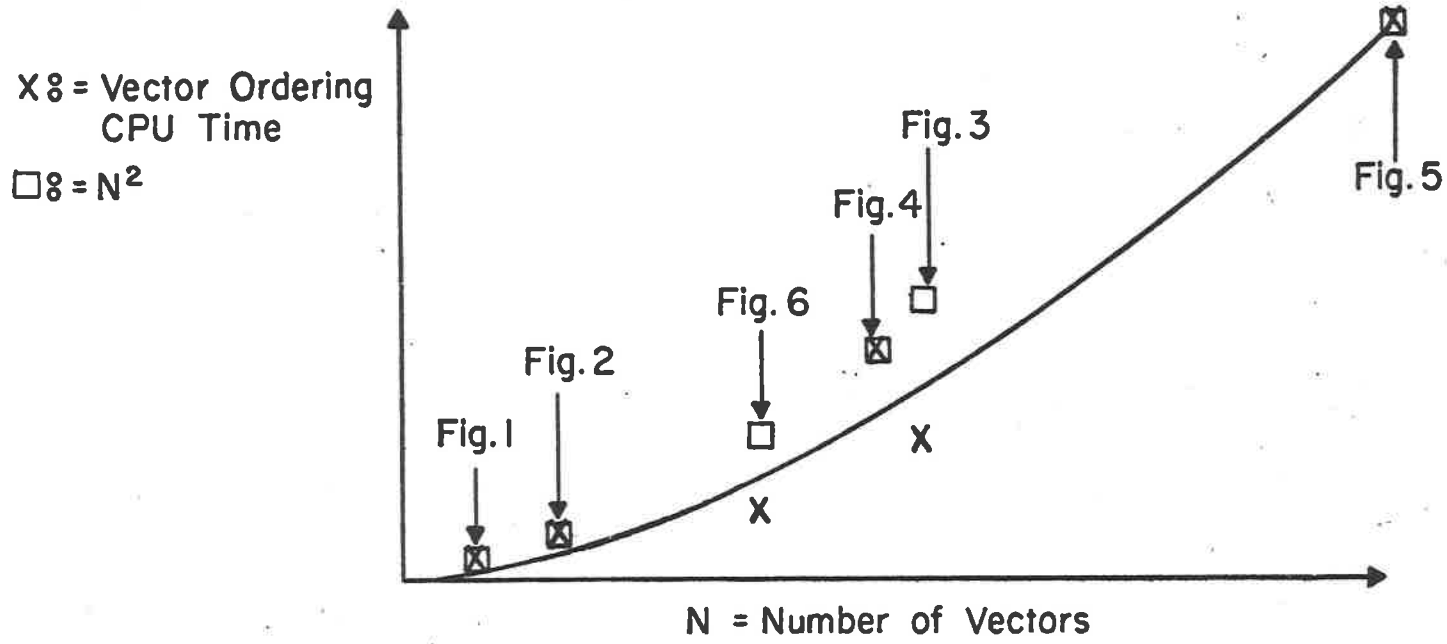



Fig. 7B

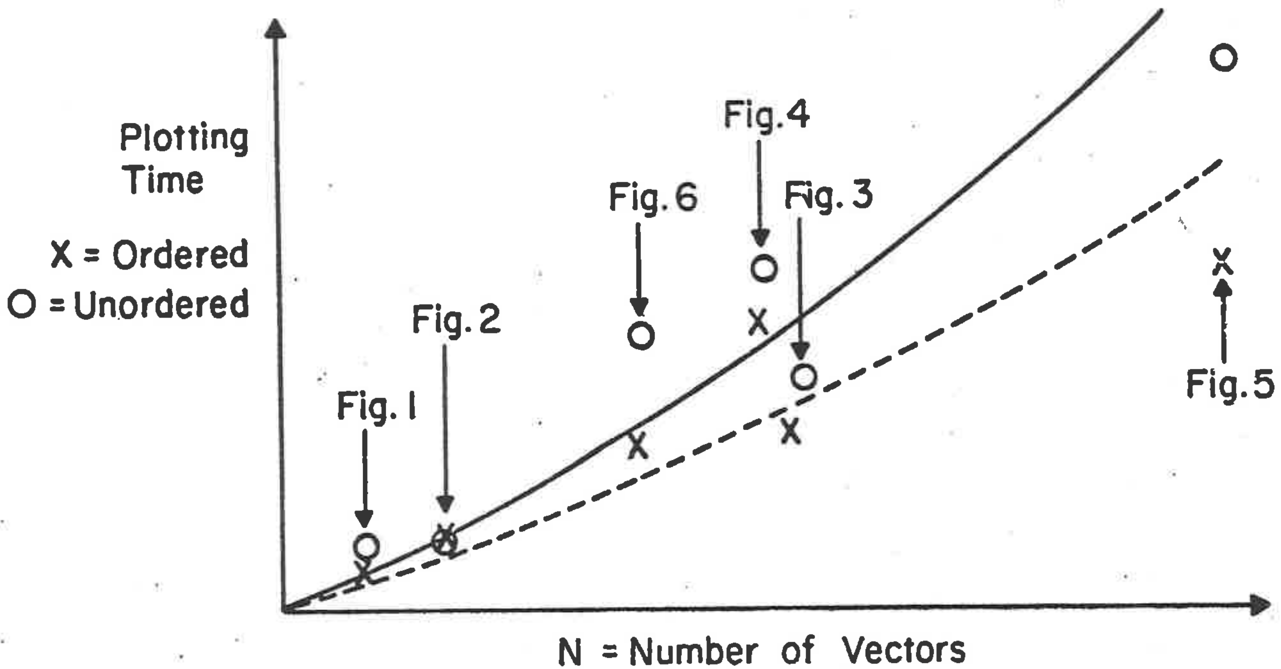



Fig. 7C