MASSACHUSETTS INSTITUTE OF TECHNOLOGY
PROJECT MAC

Artificial Intelligence
Memo No. 206

July 1970

THE VISION LABORATORY: PART ONE

Thomas O. Binford

Some of the facilities for vision programming are discussed in the format of a user's manual.

Work reported herein was supported by the Warren McCulloch Laboratory, an M.I.T. research program sponsored by the Advanced Researc Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-70-A-0362-0002.

HOW TO MAKE A DLISP

The functions in DLISP are hand-coded in MIDAS using the macros and linking mechanism of Roland Silver (A.I. Memo 127A). They are contained in the file TOPO MQX (or latest version) and also in the file DLISP ENGL on the tape labelled VZA DIS. The procedure is:

```
make a file with DLISPF==1      ; suppresses TOPO functions
assemble with TS BMIDAS on ARCHIVE tape
                                ; requires large macro area
load the assembled file with the current relocatable LISP


$L UTn:TS_BMIDAS CR             ;TOB ARCHIVE tape
DLISP SUBRS←DLISP ENGL CR


:STINK_
JLISP$
MDLISP_SUBRS$L
MLISP;RLISP_107H$L$$
?$$                             ; ask errors
TD$$                            ; terminate
$Y DLISP_BIN CR                 ; dump
```

Why deal with stored data? Unless the whole image is stored, only certain routines can use the data (for example, those with fixed scan patterns).

1. Convenience: Setting up the vidisector takes me about 15 minutes. Setting up particular conditions, for example a particularly difficult edge, may take longer.
2. Reliability: The vidisector usually works, but it has been down for periods of three weeks and longer.
3. Repeatibility: A useful procedure has been to dump each scene, then process it. If the program encounters a bug, that data is valuable and allows the bug to be trapped. Repeatibility also allows isolating changes. Conceptual changes can also be tested more quickly with a well-known scene.

Loading Stored Data

Loading stored pictures is simple if the entire picture is to be loaded. A function LOAD is called:

```
(LOAD fname1 fname2 dev user array)
```

where the last argument specifies the array name for the data to be stored, and the other arguments specify a file name. The size of the array is determined by LOAD and is attached to the array as the SIZE property of the atom. Note that LOAD takes arguments very much like UREAD. The file LOAD > on the TOB ARCHIVE tape contains the necessary functions.

Where it is desired to work with only a portion of a stored picture, the user must struggle with the rather complicated set of arguments required by the binary read routines of Krakauer. His notes are repeated here for reference.

Making Stored Pictures

To make stored pictures:

set up the vidisector	
load LISP PICPAC	:LOAD_PICPAC_SYS_
visualize and frame the scene	
	$G (VIEW)
scan	(VSCAN_den)
dump.	(DUMP_fn1_fn2_dev_user_PICTURE)

After executing (VIEW), the user selects a grid with pots labelled 143-147 and finally types T_ to terminate.

The system requires about 44 blocks of core, depending on the size of the picture.

To make up a new version of PICPAC SYS, load the TOPO file and allocate as desired, load a file PICPAC X13 which generates a smaller version of the TOPOLOGIST, with TOPO inoperative and inessential.

```
:LOAD_TOPO_NSYS_
$G
ALLOC Y
CORE 36
.......
(UREAD PICPAC X13)Q
```

where PICPAC X13 is only REGION X13 with the TOPO core allocation removed and a smaller PICTURE array.

Preliminary PICPAC for LISP

L.J. Krakauer

8/3/67
Revised 10/20/67

Several functions have been added to LISP in order to allow the reading of vidisector images from tape, and the writing of such images onto tape.

Before describing the functions, however, a word or two must be said about the image conventions of PICPAC. Images are considered to be rectangular subportions of the unit square, and hence image coordinates are floating point numbers between 0 and 1. This convention facilitates the mapping of this "image space" onto various I/O devices, such as both vidisectors, the display, the plotter, etc. Since fixed point coordinates have often been used in the past, however, all functions needing floating point arguments will perform the conversion from fixed point if fixed point arguments are supplied. The fixed point values are assumed to be new vidisector coordinates, so that the conversion amounts to floating the coordinates and dividing by 4096.0.

The currently available functions are:

(PICARRAY arr gc xdim ydim): This function declares an image array. Its use is exactly the same as the function ARRAY: the arguments are, respectively, the array name, gc=NIL, the array x dimension, and the y dimension. Since an image array will contain numbers, and not pointers to S-expressions, the second argument, gc, should always be NIL. The array so declared looks like a normal LISP array; that is, (arr n m) will evaluate to the x=n, y=m entry in the array.

(UREAD name1 name2 unit): The regular UREAD is used to open a file for reading (but do <u>not</u> type ⌊Q.)

(READPIC arr llx lly del) or (READPIC arr llx lly delx dely): This function performs the read from the file previously specified in a UREAD into the array arr. The arguments are respectively the lower-left x and y coordinates, and the x and y deltas respectively (the y delta will be assumed the same as the y delta if the last argument is omitted). The number of points read is determined by the array's dimensions. Thus the coordinates of the upper-right point of the image area read in are given by:

urx = llx + delx*xdim

ury = lly + dely*ydim

These arguments are normally to be floating point, but if fixed point numbers are given, they will be assumed to be new vidisector coordinates and will be converted, as previously noted.

The value of READPIC will be arr, the name of the array, if the read is successful. In order to be successful, however, the area of the image requested must be a subpart of the area recorded on the tape. The area on the tape will not in general be the entire unit square, however. If a portion of the area requested is not on the tape, READPIC will print an error comment and return the value NIL.

Note that if the delta given is not an integer multiple of the delta on the tape, no error comment is printed, but rather READPIC tries to do the best it can, returning for each point requested the value of the closest lattice point actually recorded on the tape.

(UWRITE unit): The regular UWRITE is used.

(WRITEPIC arr): The entire array arr is written out on the unit previously opened for writing.

(UFILE name1 name2): The same UFILE is used as for ordinary ASCII files.

(DESCR a): usually (DESCR (QUOTE ARR)) or (DESCR):
The argument a is evaluated; it should evaluate to either the name

of an array or to NIL. This function (its name stands for "des-cribe") evaluates to a list of 10. numbers describing the array, which are, in order:

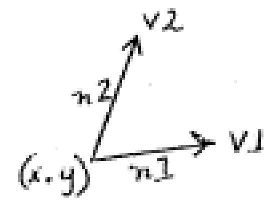
(xdim ydim llx lly delx dely -335577777776 vd light data),

where the last three numbers give information about the vidisector used, the lighting, and the mode of the data. Numbers 3 through 6 are in floating point, and number 7 is a byte pointer used internally, which can be ignored. (DESCR) evaluates to a similar list which describes the image on tape which was last read from by a READPIC, whether successfully or otherwise. (DESCR NIL)=NIL. A useful trick is to execute a (SETQ ARR (QUOTE ARR)) for all arrays. (DESCR ARR) may then be typed instead of (DESCR (QUOTE ARR)).

(DESCRX a): This function is the same as DESCR, except that all floating point numbers are fixed, after being converted to new vidisector coordinates by multiplying by 4096.0. Images on tape will generally have integral deltas.

SCAN FUNCTION IN LISP

A scan function which has wide utility is available in DLISP. The function evaluates functional arguments at the locations in two dimensions given by the parallelogram specified by a point and (n1 . n2) steps along two vectors (v1 . v2).



A typical call is:

```
(SCAN '(fun1 fun2 rowfun) (x . y)(n1 . n2)(v1 . v2))
```

where v1 and v2 are dotted pairs, vectors defining the directions of the steps. Typically, (v1 . v2) = ((0 . den) . (den . 0)). The functional arguments fun2 and rowfun are optional if present, rowfun is called at the beginning of each row as an initialization function, then at each point,

```
(fun2 (fun1 x y))
```

is evaluated. The function SCANA assumes that the second argument is an array, and stores into it.

```
(SCANA '(fun1 array) (x . y)(n1 . n2)(v1 . v2))
```

A complication in the use of these functions rests on the LISP convention with the order of elements in an array. LISP stores elements backwards from the usual convention of the faster moving index as the first index. The SCAN routines were designed to work with these arrays, and thus have reversed x and y coordinates for real world devices like the vidisector.

THE GEOMETER

The analysis performed by the GEOMETER has been described in another note. We go into programming detail here. A package of modules is called in a dozen subroutine calls, by a very brief routine called EXECUTE. The flow of control is outlined below, but can be followed directly in EXECUTE. The primary data is the list REGIONS and various properties of each region, primarily inclusion and BOUND.

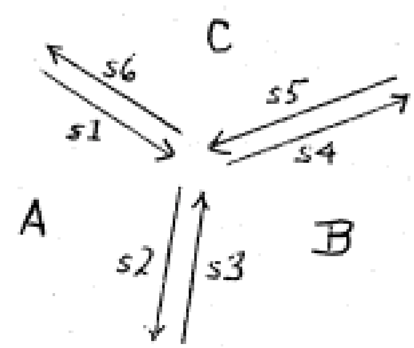
```
REGIONS
(R12 H13 R14 R15 R16 R17 R20 R21 H22 H23 H24 H25 R26 R27
 H30 H31 R32 R33 R34 H35 R36 R37 R40 R41 R42 H43 R44 R45 R46
 R47 R50 R51 R52 R53 R54 R55 F56 R57 R60 R61 R62)
```

The BOUND property is a list of sublists consisting of a code for the neighbor, followed by a list of points.

```
R14 BOUND
((NIL (174 . 20))
 (15 (174 . 22) (174 . 24) (174 . 26) (174 . 30) (174 . 32)
     (174 . 34) (174 . 36) (174 . 40) (174 . 42) (174 . 44)
     (174 . 46) (174 . 50) . . . .
```

We choose big regions on the basis of perimeter, then determine the list SEGMENTS of boundaries of big regions. The properties of interest are the CORNERS of a segment, the SEGMENTS property of a segment (list of sublists of points), and the REGIONS property of a segment. The S property of a region is also of interest; it is a list of dotted pairs of nhbr and segment, cyclic around the boundary. The subsegmentation into straight lines is done at this time; its results are the CORNERS property of a segment.

We go from segments to vertices by the syntactic analysis on neighborhoods. By pairing segments across a common boundary and by cycling around a vertex using alternately successor and pairing operations:



S2 is the successor of S1

S3 pairs with S2

S4 is the successor of S3

S5 pairs with S4

We come to vertices involving three or more regions. The properties of interest are:

the CYCLE of a vertex, a ccw list of sublists of paired segments

VERTEX1 and VERTEX2 of a segment, names of vertices.

VERTICES

(V30 V27 V26 V25 V24 V23 V22 V21 V20 V17 V16 V15 V14 V13)

```
(PRINTL (GET 'V30 'CYCLE))
(((15 . S22) (14 . S13))
 ((12 . S26) (15 . S23))
 ((14 . S31) (12 . S32)))
```

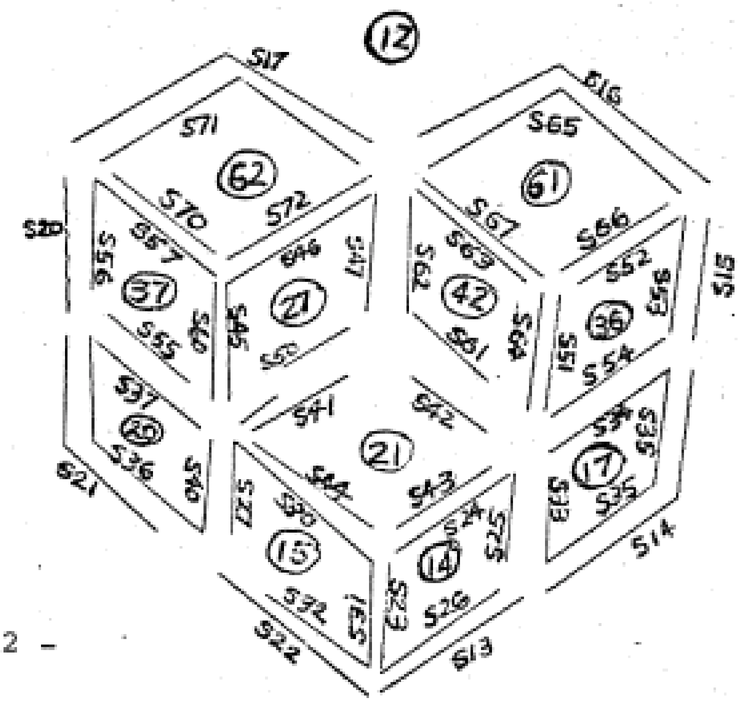
We obtain the location of vertices by intersection, and make a better approximation to the straight lines between three-region vertices. Then we prepare the format for output; that form is a list of vertices with their positions and connectivity.

CONNECT property of a vertex, ccw list of connected vertices

POSITION property of a vertex, dotted pair floating point (x . y)

The PROPOSER works with that format and possibly adds some new connections.

The figure which follows will be a useful model for the examples of the new few pages.



EXEC1 defines:

BREGIONS, a list of big regions
NBREGIONS, a corresponding list of region codes
TRL2, an assoc translation list from codes to regions

EDGES defines:

SEGMENTS, a list of boundaries of big regions
S property of a region, list of dotted pairs, nhbr and segment cyclic around boundary

It calls:

S: which strings together sublists of BOUND with a constant big region nhbr
SEGMENTS: which subsegments into straight lines. Returns end points and any intervening corners.

An example of the S property:

```
R14 S
(12 . S26)
(15 . S23)
(21 . S24)
(17 . S25)
(12 . S26)
```

S2 makes only a small format change.

VA defines:

PAIR property of a segment

VA calls:

FINDS
PAIRS

The action of VA is to pair segments on opposite sides of a common boundary by neighborhood and parallel-opposite. This pairing rejects much noise which fails to affect both elements of a pair. The variables PARALLEL (radians, currently set at about 30°) and PDTOL, a loose tolerance on perpendicular distance, control these conditions. The overlap condition is that one end of one of the lines must be interior to the other in projection.

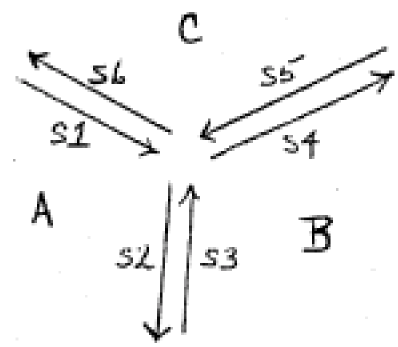
V1 defines:

VERTICES, a list of three-region vertices
CYCLE property of a vertex, a list of sublists of length 2, each element of which is a dotted pair, nhbr and segment
VERTEX1 and VERTEX2 property of a segment

V1 calls:

CYCLE

The syntactic operation extracts three-or-more region vertices. CYCLE cycles around the vertex by alternating successor and pair operations. Successor of S1 is S2; the pair of S2 is S3; the successor of S3 is S4; etc.




```
V30 CYCLE
((15 . S22) (14 . S13))
((12 . S26) (15 . S23))
((14 . S31) (12 . S32))
```

V2 defines:

POSITION of a vertex, dotted pair of floating point numbers

V2 calls:
VTEX

V2 determines a best intersection of lines at a vertex by determining a position with minimum mean square perpendicular distance from the lines. Each line appears paired. Iterates a second and third time, weighting square distances inversely by square extrapolation errors, thus giving most credit to the most accurate estimates.

VTEST calls:

VTEX

VTEST occasionally merges two adjacent vertices. Useful to suppress spurious vertices caused by low resolution.

LV

begins with the line between the three-region vertices at either end of a segment. If all interior vertices are sufficiently close to the line, the segment is treated as a straight line. If not, cluster by proximity of end points, and fit a vertex to each cluster.

PAIRV
CONNECT 3 - these two functions define:

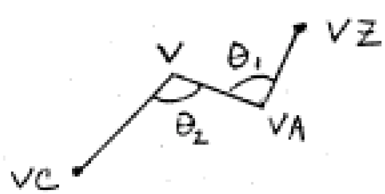
CONNECT property of a vertex

These routines complete the format of the output: a list of vertices with their positions and connectivities.

PROPOSE calls:

PREDECESSOR
SUCCESSOR
COLINEAR, CONVEX, and CONCAVE

PROPOSE uses the simple format: a list of vertices, their positions, and their connectivity. For each vertex, it defines predecessor and successor properties to simplify traveling around the net. It looks only at concave vertices. It tries to close parallelograms. Given a concave vertex, it examines each connecting vertex. The angle θ_1 must be less than the angle θ_2. If so, the routine looks by multi-entry for a vertex near the point predicted by translating V among VA,VZ. An earlier version used broken lines, to extend them, and to connect two vertices with an edge parallel to one in the region, provided it did not cross any other edge.



OUTPUT

writes out the data in the final format and with the data a little program which reads the data in and formats it. The function is called with the name of the output file:

```
(OUTPUT fname1 fname2).
```

Two flags are of interest:

SHOW if non-nil, causes display of various steps in processing.

PLOT if non-nil, causes plots of the steps displayed.

Some useful functions for looking at data are:

SHOWEDGES - paired straight lines

SHOWENDS - straight lines

SHOW - line drawing from final format

SHOWBOUND - unprocessed boundary

To help examine property lists:

(PLIST list property) FEXPR prints the indicated property in a useful form, for each element in the list

(PROPS atom) FEXPR prints the indicators only on the property list of the atom.

SEGMENTS

The purpose of SEGMENTS is to take a portion of a boundary and break it up into a sequence of straight lines. There are two entries, which differ only in whether the list returned contains the endpoints. The routines are in MIDAS, loaded into LISP using the macros and linking mechanism of Roland Silver (A.I. Memo 127A). They live in the TOPO MQX and in TVJ ＞.

The method is simple: given an ordered list of points, we take the line between end points, and subdivide the list or not depending on the maximum perpendicular distance of points from the line. If the maximum perpendicular distance is greater than approximately four times the transverse point scatter dx (dx is approximately half a raster unit in a typical case), the list is subdivided and the procedure is applied recursively to the sublists. The limit of perpendicular distance is set from LISP by:

```
(SEGI limit)          ; initialization
```

where limit is floating point.

If there results more than one division, the conditions for finding a corner are not always well met (no line should be near parallel to the line between end points). Therefore, we repeat the process twice more on the lines obtained, but consider alternate vertices. Some corners are shifted and others disappear.

The routine returns a list of corner points (which are actually points from the list). It would be better to return lists of fitted points. The routine is called with a list of dotted pairs, not necessarily floating point.

The perpendicular distance test is quite fast. The perpendicular distance corresponds to the y-coordinate in a coordinate system rotated along the line connecting end points as the x-axis.

The parameters used by the program are:

SEGLIM: the limit to perpendicular distances considered colinear. Set by (SEGI limit), it is a floating point machine number, not a LISP atom.

SUBRs defined:

SEGI: initialize the parameter SEGLIM

SEGMENTS: returns a list of corners with end points in the form of dotted pairs of floating point LISP numbers.

SEGMENT: returns list of corners without end points.

We make a short description of the operation and of the principal MIDAS entries. A user who wished to incorporate the routines would need to change the input form and the output form.

SEGPUSH: convert from input format to internal format; push points on a point pdl (which overlaps TOPO and thus smashes TOPO). The internal format is alternate x and y in a block.

SEGB2: segment portion of line between two pointers on the point pdl and recurse. Arguments are a list of corners in A, and begin and end pointers in C and R4.

SEGR: repeat segmentation on each sublist between alternate corners.

The program, roughly speaking, is:

```
    PUSHJ P, SEGPUSH          ;set up internal format
    PUSHJ P, SEGB2            ;initial segmentation
    PUSHJ P, SEGR             ;repeat segmentation
    PUSHJ P, SEGR
    JRST SEGPO                ;output list of dotted pairs.
```

MULTI-ENTRY CODING

The multi-entry coding routines are a module available in LISP and MIDAS. The routines provide the mechanism for two-dimensional proximity, i.e. find all points near a point p. Topology requires that there be N+1 overlapping cells in N dimensions to guarantee proximity. For simplicity, we have four instead of three overlapping cells. The distance for proximity defines the dimension of the array; if delta is this distance, the array must correspond to cells of twice this size. The array has dimensions

N1/(2*DELTA), N2/(2*DELTA), 4

but is a half word array. In the MIDAS version, hash coding is used if the image array exceeds the storage area in size. In each half word, a list of the entries is kept.

Functions to initialize, to store, and to retrieve associations comprise the package. In LISP:

(MATCHA nx ny (cons sx sy)) initializes by calculating the scale and allocating an array of the calculated size. Here, nx and ny are upper limits of coordinates which are assumed to run from (0 . 0) to (nx . ny); sx and sy are cell sizes along the two dimensions.

(MULTISTORE p ptr) stores the pointer ptr at the position of the dotted pair p.

(MULTIFIND p ptr) returns a list of elements different from ptr and with no repetitions, elements near the dotted pair position p.

The functions occupy one page of EXPR code in the file labelled A 262 on the tape TOB ARCHIVE.

In the MIDAS version, the array location and array size are stored in variables:

```
MULTIA:          ;array location
MULTIL:          ;array length
A:               ;xlow float pt
B:               ;ylow float pt
C:               ;xhigh float pt
R4:              ;yhigh float pt
```

To initialize, either:

```
MULTIM:          ;x dimension fixed pt
MULTIN:          ;y dimension fixed pt
PUSHJ P,MULTII   ;calculates scale and scaling function
```

or the user can specify the scale factors and allow the program to calculate the dimensions of the image array:

```
MULTIQ:          ;scale factor
MULTIQ+1         ;scale factor y
PUSHJ P, MULTIZ  ;initialize, calc dimensions and choose scale
                  function
```

To store in the multi-entry array:

A: pointer

B: x float pt

C: y float pt

PUSHJ P, MULTIS

which returns:

A: original pointer

B-R5: four lists of associations, exactly, in each of the four registers are two lists: after cons list, before cons list.

To find associations:

A: pointer

B: x float pt

C: y float pt

PUSHJ P, MULTIV

which returns a list in A, without repetitions and without the original pointer.

To store vectors, interpolating points between end points, sufficient to guarantee proximity:

A: pointer

B: x1 float pt

C: y1 float pt

R4: x2 float pt

R5: y2 float pt

MULTVF: funarg evaluated at end points and interpolated points

PUSHJ P, MULTIV

At each interpolated point and end point, the program calls the functional argument in MULTVF (which might be MULTIS, but which woul preferably be a function which calls MULTIS, then processes the associations which occur). A useful way of using MULTIV is with a hash-coded table of pairs to avoid repetition. The CONSes can be done from a pdl or free storage area.

STRAIGHT LINE FITTING

We usually represent a line in the symmetric form:

x*sint - y*cost + z = 0.

The special cases

y = ax + b

x = cy + d

are simply:

x*tant - y + z/cost = 0

x - y*cott + z/sint = 0

The solution to the special cases is straightforward in terms of the method of projection:

y = ax + b

$$\sum_i y_i = a\sum x_i + bN$$
$$\sum_i y_i x_i = a\sum x_i x_i + b\sum x_i$$

This is a system of two equations in two unknowns:

$$a = \left(\frac{1}{N}\sum_i y \sum_i x_i - \sum y_i x_i\right) \Big/ \left(\frac{1}{N}\sum x_i \sum x_i - \sum x_i x_i\right)$$

$$b = \left(\sum x_i y_i \sum x_i - \sum x_i x_i \sum y_i\right) \Big/ \left(\sum x_i \sum x_i - N\sum x_i x_i\right)$$

whose solution corresponds to the line with the same first two moments as the sample. The equations are exactly those of the least squares solution. The solution for the form

x = cy + d

can be obtained by interchanging x and y in the two equations. The case of a symmetric interval in x is often very useful. Then, the sum on x vanishes, $\sum_i x_i = 0$ and:

$$a = \sum y_i x_i / \sum x_i x_i$$

$$b = \sum y_i / N$$

The general linear form has a nonlinear normalization condition and the solution is:

$$\tan 2\alpha = \frac{\left[-\sum_i x_i y_i + \frac{1}{N}\sum_i x_i \sum_i y_i\right]}{\frac{1}{2}\left[\sum_i x_i x_i - \sum_i y_i y_i - \frac{1}{N}\left(\sum_i x_i \sum_i x_i - \sum_i y_i \sum_i y_i\right)\right]}$$

One useful quantity for description of a line is its angle; slopes are not continuous through the full range of π directions. Although the general linear solution involves transcendental functions which involve a certain amount of computation, the alternative is to take the special case solution about the axis which lies nearer the line of the data. This amounts to choosing the larger of the denominators

$$\left(\frac{1}{N}\sum_i x_i \sum_i x_i - \sum_i x_i x_i\right) \quad \text{OR} \quad \left(\frac{1}{N}\sum_i y_i \sum_i y_i - \sum_i y_i y_i\right)$$

The straight line fits should be adequate even though the procedure is not rotationally invariant, but tan θ is a poor approx to the angle θ at angles near 45°. The simplest solution is to calculate the angle θ corresponding to a few terms in the atan θ series expansion.

Straightforward error propagation shows the mean squared error in slope to be:

$$\langle da^* da \rangle = \langle dy^* dy \rangle / \left(\frac{1}{N} \sum_i x_i \sum_i x_i - \sum_i x_i x_i \right)$$

For the general case, we can use the same result after a rotation of coordinates with x axis along the direction of the line. A usual test for linearity is the mean squared error:

$$M = \sum (ax + by + c)^2$$

which can be computed in terms of the sums already calculated:

$$M = a^2 \sum x^2 + 2ab \sum xy + b^2 \sum y^2 + 2ac \sum x + 2bc \sum y + c^2 \sum 1$$

A description of the line segment by two of the sample points is deficient but useful. Instead, we describe a line segment by end points, projected on the line. These are equivalent to the best fit line, and are an alternative representation.

Functions in both LISP and MIDAS are available to compute straight line fits. In LISP, the procedure is to evaluate:

```
(STLINE L)
```

where L is a list of dotted pairs. The value is a list of three parameters in one of the two special case forms.

The MIDAS version (available in TVJ ⟩) fits the general form of the linear equation. The internal representation is a block of alternate x and y floating point positions. It expects:

A: pointer to the first point

B: pointer to the last point

```
PUSHJ P, LFIT
```

The results are in a block of about 20 words to BLT into a header block for the line. Other entries are:

```
A:  x float pt
B:  y float pt
PUSHJ P, LFITP
```

which adds a point to the line sums.

```
PUSHJ P, LFIT 3
```

which takes its data from the LFIT data block and fits the parameters of a general straight line.

```
A:  x float pt
B:  y float pt
PUSHJ P, LFITPR
```

which projects the point (x,y) on the line in the LFIT data block.

For the purpose of testing colinearity, a function for the special case line fit also exists, but has not been debugged. The time required per fit is around .5 msec, sufficiently fast to allow rather free testing of colinearity hypotheses for extension and redundancy of lines.

LINE VERIFICATION

LV

Given two points Q_1 and Q_2 in the field of view, the program will tell whether an edge extends from $Q_1 \pm \vec{i}$ to $Q_2 \pm \vec{i}$ ($\vec{i}$ normal to Q_1Q_2).

The central part of the program is in MIDAS to be used within LISP. A few EXPR execute top level functions.

Instructions for use:

1. Assemble the MIDAS program MLV 1 (or MLV 2) which is on tape AHD.

   MIDAS↑H

   device:user; MLV BIN←MLV 1 CR

2. Link the assembled version with LISP and TOPO as follows:

   STINK↑H

   JLISP$

   Mdevice:user;TOPO BIN$L

   Mdevice:user;MLV BIN$L

   Mdevice:user;RLISP 107H$L$$

   ?$$

   TD$$

   $Y LV BIN CR

3. Read in file LV 1 (or LV 2) of tape AHD

   (UREAD LV 1 device user)

4. Setq Q1 and Q2 to coordinates of extremal points of possible edge, i.e.:

```
(SETQ Q1 (CONS X-coord. Y-coord.))
(SETQ Q2 (CONS X-coord. Y-coord.))
```

Coordinates should be fixed point quantities between 0 and 1777 octal.

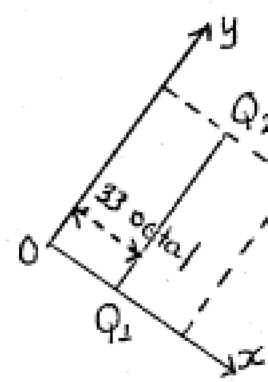
5. Execute

```
(LV)
```

If no edge is found it returns NIL, otherwise a description of the edge as follows:

```
(STEP DARK RIGHT (20 . 25))
```

- -edge of the step type
- -the dotted pair represents the x-coordinates of the lower and upper point of the actual edge with respect to axes Ox and Oy as shown in the figure
- -the darker face is right of Q_1Q_2.



Another description:

```
(ROOF UP (22 . 42))
```

-roof looking upward.

etc...

6. a) $|\vec{i}|$ is set to 10. If you wish to change it execute:

(SETQ INCERT new value)

floating point number

(LVCST)

(TABALF)

b) The shortest and longest edges for which the verification process is secure are 50. and 400. respectively (2000 octal being the side of the whole field). To change this execute:

(SETQ MAXLE new value of maximal length)

(SETQ MINLE new value of minimal length)

(LVCST)

(TABALF)

Values should be floating point numbers.

Note: if you make MINLE less than 50., you should use 5 bands instead of 10 (for clarification see paragraph 7 below). Then MINLE can be lowered down to 25.

c) If you want to use the program with canned data, read in the canned data after reading file LV 1, as follows:

(LOAD Edge File No. device user LID)

array where cross-sections are stored

Presumably you would before have dumped LID:

(DUMP Edge File No. device user LID)

7. The procedure is set with 10 bands (see On Boundary Detection, A.I. Memo No. 183, p. 45). If you wish to use five bands execute:

```
(SETQ NBD 5)

(SETQ THBIN 4)

(LVCST)

(SETPAR 'TV)
```