A Full Field, 3-D Velocimeter for Microgravity Crystallization Experiments

Robert S. Brodkey and Keith M. Russ
Department of Chemical Engineering

NASA Lewis Research Center
Cleveland, Ohio 44135

Grant No. NAG 3-1039
Final Report

April 1991
A Full Field, 3-D Velocimeter for Microgravity Crystallization Experiments

Robert S. Brodkey and Keith M. Russ
Department of Chemical Engineering

NASA Lewis Research Center
Cleveland, Ohio 44135

Grant No. NAG 3-1039
Final Report
RF Project No. 767435/722039

April 1991
Contents

1. Overview ....................................................... 2
2. Basic experimental statement ............................... 2
3. View configuration ........................................... 2
4. Hardware configuration ........................................ 3
5. Computer configuration ....................................... 3
6. Image processing ................................................ 5
7. Image analysis ................................................... 5
8. Conclusions ...................................................... 7

Appendices
A. PART_ID.FOR .................................................. 21
B. DET_VEC.T.FOR ............................................... 31
C. Equipment Specifications ...................................... 47
Overview

In our proposal, we set forth to develop the programming and algorithms needed for implementing a full-field, 3D velocimeter for laminar flow systems, and to recommend appropriate hardware to fully implement this ultimate system. Over the course of the project, we realized that these two steps are not as distinctly separate as we once thought; the software solutions can be modified to take advantage of various hardware configurations.

This summary indicates one possible hardware solution, already provided in the informal report provided to NASA researchers in January, 1990. The current state of our programming efforts is also provided, as well as such code as is appropriate.

Basic experimental statement

The velocimeter should consist of two views of the flow system, in this case the interface area of a crystallization experiment. These views are directly digitized by the use of a video camera(s) and a computer digitizer board(s), and the velocities determined by a combination of hardware image processing and software image analysis.

View configuration

The simplest view configuration to visualize is a pair of video cameras mounted at right angles to each other (orthogonal views), with a common field of view in the flow system. This will require a synched pair of video cameras and digitizer boards, as well as synched rails for camera motion. These views were tested by NASA by simulating the flow in a crystallization ampoule (Plate 1(a-b)). Although it is not readily apparent in the image pair, some 50% or more of the viewed volume is blocked for 3D analysis by the heater element images (after simple processing). This blockage made 3D path matching an almost impossible task, especially when the lack of registration points in the images and the difference in the view magnification (obvious in Plate 1(a-b)) are considered.
It is recommended that the orthogonal viewing (90° angle) be replaced with a low to medium angle viewing, between stereo angle separation (approximately 10°) and about 30°. This will reduce blockage effects at the cost of depth accuracy. As the flow system is laminar, depth accuracy should be considered a minor point.

There has recently been developed a single camera 3D acquisition technique. Such a technique holds significant potential for replacing low angle stereo viewing, as well as reducing hardware costs without significantly degrading software computational requirements. We hope to investigate this option, pending further support.

**Hardware configuration**

This is perhaps the most flexible area, although dependent on (and instrumental in) the view configuration decided upon above. Two digitizer boards (to avoid image capture sequencing) and two video cameras would be required for stereo acquisition, each capable of 512x512 resolution (black and white). The EPIX 1MEG VIDEO Model 10 digitizer board appears to be an excellent choice for this work; it is capable of storing 4 images in its own memory, and has an on-board digital processor that can be programmed to perform the image processing we envision to be necessary. Such on-board processing should provide significant reductions to computation time.

The video cameras each need to be of around 512x512 non-interlaced resolution (either to the RS-170 resolution of 768x480 pixels, or perhaps the European CCIR standard of 768x580 pixels). Such units are relatively inexpensive ($1,500 apiece and up, without lenses). Of the two standards, the CCIR hardware appears to be more attractive with its higher resolution.

**Computer configuration**

The host computer for the imaging hardware would dictate the speed at which the images are analyzed for velocity, as well as any control actions one
may wish to take as a result of such data. For the computer, it seems necessary to obtain the absolute best performance at the most reasonable cost. Currently, this claim rests with Intel's 80486 processor, in whatever machine from whatever vendor. The majority of machines available now are based on Intel's 80386, which for similar clockspeeds operates up to 2 to 4 times more slowly than an 80486. Unfortunately, the 80486 is so recent that software availability is limited. The minimum requirements are an 80386 machine, at whatever clockspeeds are available (80386 computers can be found running at 16, 20, 25 and 33 MHz; costs start rising significantly with the 25 and 33 MHz models), using at the absolute minimum 2 MB of RAM and preferably 4 MB. Video adapters and hard drives are of secondary importance, since all of the grunt work will take place between the CPU and the RAM. It is therefore essential that the RAM be of sufficient speed to match the CPU.

The 80386 can run the MS-DOS or UNIX operating systems. UNIX is perhaps the preferential system, especially when paired with the C programming language. Since C and UNIX are still in standardization, however, it is perhaps better to stick to the older FORTRAN language in MS-DOS, which provides all the numerical routines you could ever wish for at the expense of data handling. Such a compiler needs to run under a MS-DOS extension to allocate and utilize large arrays 512x512 or larger), and should be specific for the 80386 instruction set (or, if an 80486 is used it should be specific to it). A particularly inviting compiler and related software that fits this description is marketed by MicroWay. Their NDP Fortran-386, as illustrated by their data sheet, provides excellent number-crunching facilities. When combined with the Phar-Lap DOS extender, this PC-based compiler can access arrays limited only by the on-board RAM (in this case, the recommended 2-4 MB); additionally, a virtual memory manage can be added to utilize the hard drive as virtual RAM, thereby economically increasing potential array size.

Finally, an Assembler linker would be needed to best utilize the EPIX boards. The EPIX boards include an on-board digital signal processor, programmable in
Assembler. The EPIX people recommend the product of Avocet, at around $350.

**Image processing**

Our current technique is to take a sequence of images (at least 3, but it could be up to 8), create a series of binary images from these, and add them together to create one image for analysis. From this one image particle velocities and paths can be calculated for each resultant frame. The combination of the images can be accomplished by

\[ S_{jk} = \sum_{l=1}^{n} 2^{l-1} I_{lk} \]

for \( n=3,4,5,6,7, \) or 8, and for every \((j,k)\)th point in the original images \((I)\). Three of the NASA slides (of which Plate I(a) is the first), digitized and added together by the above equation, is given in Plate II(a). This represents the final output of image processing.

**Image analysis**

Image analysis of the particle paths, clearly visible in Plate II(a-b), represents three operations: *particle identification*, whereby the particles are located by their centers, grey levels, and size; *particle tracking*, whereby the particles are tracked within the image; and *3D track matching*, which matches tracks within the two views to determine the 3D information. The particle identification algorithm, a modification of that by Chang et al. (1985)\(^1\), identifies the particles from top to bottom, left to right, row by row. The identified particles are screened to eliminate spurious anomalies left over from image processing (i.e. below a \( d_{min} \) or above a \( d_{max} \)). The particles found from this for Plate II(a) are given in Figure 1. The code is given in Appendix A, and a flowchart in Figure 2.

---

The particle tracking algorithm utilizes a rudimentary form of path and velocity coherence. The particles found from the previous step are sorted into four lists, one for each frame. Due to the nature of the particle ID algorithm, these particles are already approximately sorted in the lists from low to high y values. Starting from the beginning, each 1st frame particle is compared with candidate 2nd frame particles, and each 2nd frame with candidate 3rd frame particles, until one of the following conditions is met:

\[ \Delta y > \Delta y_{\text{max}} \quad (2a) \]

\[ \frac{|v_{1-2} - v_{2-3}|}{v_{1-2} + v_{2-3}} < \text{tol}_v \quad \text{and} \quad |\theta_{1-2} - \theta_{2-3}| < \text{tol}_\theta \quad (2b) \]

where 1, 2, and 3 refer to frames, and \( \theta \) is the angle as if the particle pair were in polar coordinates (of which the magnitude of the velocity is the other coordinate; see Figure 3). \( \Delta y_{\text{max}} \) is the maximum expected frame-to-frame particle movement in y, \( \text{tol}_v \) is the maximum change in average velocity between each pair of frames, and \( \text{tol}_\theta \) is the maximum angular motion of the particle. This represents the basic tracking algorithm; its implementation was modified by using a simulated image.

Some of the particle paths in Figure 1 are clearly visible; others are not. It is not easy to quantify the particle tracking algorithm in such a case. To help determine the accuracy of the tracking algorithm, a test image was created having 200 randomly determined velocity vectors (Figure 4).

It was found that the tracking algorithm (Eq. 2) ran exceedingly fast. To increase the tracker's accuracy, multiple passes (varying \( \Delta y_{\text{max}}, \text{tol}_v, \text{tol}_\theta \)) were implemented. In addition the lists were matched in reverse order, i.e. from bottom right to upper left, and compared to the forward direction; particle tracks that were found in both directions were kept, while the remaining were
discarded. The results of the tracking are given in Table I and Figure 5(a-b). The tracker only utilizes three frames' data; fourth frame data could be used to perhaps increase accuracy further.

Figure 6(a-b) represents the output of the tracker on the data from Figure 1. Not all particles represent a particle track, as some are the result of noise in the original image. There are approximately 94 actual tracks in view; of these, 81 (86.2%) were correctly matched, while 4 (4.3%) were mistracked. The code for the tracker is given in Appendix B, and a flowchart is given in Figure 7.

3-D matching has not, as yet, been implemented due to the problems mentioned in section 3. The identified particles and tracks for Plate II(b) are given in Figures 8 and 9(a-b), respectively.

Conclusions

It appears that imaging will provide a viable solution to the laminar tracking problem. Certainly the algorithms given here are simple, which in turn should speed processing. Accuracy is good, but processing times are unknown as we haven't the hardware or software to properly test the code, as a result of not being able to fully implement the second year of the proposed study. On a heavily loaded VAXstation 3100 the particle identification can take 15-30 seconds, and the tracking completed in under a second. It seems reasonable to assume that 4 image pairs can be thus be acquired and analyzed in under 1 minute.

The remaining stumbling blocks in the project appear to be the choice of tracer particle and two processing problems: 3-D matching and settling velocity estimation (the extent of this problem is dependent on tracer particle choice, obviously).
Plate I: (a) Left flow image, first frame (t=0), of particles in laminar flow in the crystallization furnace. (b) Right flow image, corresponding to (a).
Plate II: (a) Summation of binary images for left view flow, using Eq. 1. (b) Equivalent image for right view flow.
Figure 2: Flowchart for PART_ID.FOR.
Figure 3: 3 point simplistic tracker.
Table I: Results of the tracker on the simulated image, Figure 4.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$\Delta y_{\text{max}}$</th>
<th>TOL$_v$</th>
<th>TOL$_\theta$</th>
<th>% Not Found</th>
<th>% Wrong</th>
<th>% Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>1.0</td>
<td>1.0</td>
<td>81.00</td>
<td>1.00</td>
<td>18.00</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0.8</td>
<td>1.0</td>
<td>12.00</td>
<td>4.50</td>
<td>83.50</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>0.5</td>
<td>1.0</td>
<td>5.00</td>
<td>5.50</td>
<td>89.50</td>
</tr>
</tbody>
</table>
Figure 5: (a) Particle tracks identified for the simulated image, Figure 4, including particle locations (• first frame; * second frame; + third frame). (b) Particle tracks identified for the simulated image, Figure 4, vectors only.
Figure 6: (a) Particle tracks identified for the real image, Figure 1, including particle locations (• first frame; + second frame; + third frame). (b) Particle tracks identified for the real image, Figure 1, vectors only.
Figure 7: Flowchart for DET_VEC.T.FOR.
Figure 9: (a) Particle tracks identified for the real image, Figure 8, including particle locations (● first frame; ● second frame; + third frame). (b) Particle tracks identified for the real image, Figure 8, vectors only.
This program takes an ASCII image (formatted output from program XF11_TO_ASCII.FOR) and identifies and outputs particle locations by grey. It is meant to work with a 4-binary summed image, using 1, 2, 4, and 8 as the binary image multipliers. Output is directed to FOR008.DAT

The structure /AREA_TYPE/ is used to hold the particle information. Particle centers are determined by presuming them to be reasonably shaped, ie average between min & max extensions are used. NUM_PIX is used to count actual particle size. COLOR is actually grey level, ie 1, 2, 3, or 4.

```
STRUCTURE /AREA_TYPE/
    INTEGER MINHOR, MAXHOR
    INTEGER MINVER, MAXVER
    INTEGER NUM_PIX
    INTEGER COLOR
END STRUCTURE
RECORD /AREA_TYPE/ AREA(2000)
STRUCTURE /SORTED_AREA_TYPE/
    REAL XC, YC
    INTEGER NUM_PIX
    INTEGER NEXT, AVAIL, LINK, ACT
END STRUCTURE
RECORD /SORTED_AREA_TYPE/ SA(4, 1000)
```

IMAGE holds the 512x512 image array. Because of the size of this array, this program works significantly better on the VAX 8550 than on the MicroVAX. IMTE is temporary storage; this always contains the current line and the line directly above it. Makes identification a lot easier.

```
INTEGER IMAGE(512, 512), X, Y, JUNK(54), IMTE(2, 512, 5)
CHARACTER*20 FILENAME
INTEGER AREA_COUNT, JUMP, COUNT(4), XEXTENT, YEXTENT
INTEGER I, J, I2, J2, I3, J3, PSEUDO_GREY
INTEGER ORIG(8), ORIG2(8), ORIG3(8), ORIG4(8)
INTEGER PSEUDO_AREA, MIN_PIX, MAX_PIX, COL, NUM
```

Set some arbitrary tolerances on particle sizes, where MIN_PIX represents the minimum number of pixels to define a particle, and MAX_PIX defines the maximum number of pixels.

```
MIN_PIX = 4
MAX_PIX = 150
```
TOL=0.5
TYPE*, 'Enter ASCII image Filename'
READ(5,1) FILENAME
1 FORMAT(A20)
OPEN ( UNIT=3, STATUS='OLD', READONLY,
1     FILE = FILENAME, DEFAULTFILE=' .I' )
READ(3,*) LINES,PIXELS
DO 10 I=1,LINES
   TYPE*, I
   READ(3,5) (IMAGE(I,J), J=1,PIXELS)
   READ(3,5)
   READ(3,5)
5 FORMAT(18I4)
10 CONTINUE
AREA_COUNT=0

I1 and I2 are switching parameters for IMTE, allowing us
to only refill the row needed to be refilled.
I1=1

Main iteration loop. Since we check the line above the
current line, start at line 2.
DO 100 I=2,LINES
   IF (I1.EQ.2) GOTO 220
   I1=2
   I2=1
   GOTO 230
220   I1=1
   I2=2
230 CONTINUE

Row iteration loop. Since we check to left and right of
current particle position, loop so that these positions are
occupied.
DO 200 J=2,(PIXELS-1)

Set IMTE storage. IMTE at each point contains the original
grey level (IMTE(any,any,5)), and particle id for each grey
found in that pixel and identified (IMTE(any,any,1--4)).
IMTE(I2,J,5)=IMAGE(I,J)
IMTE(I2,J,1)=0
IMTE(I2,J,2)=0
IMTE(I2,J,3)=0
IMTE(I2,J,4)=0

Check if background.
IF (IMAGE(I,J).EQ.0) GOTO 200

DECOM_GREY takes the grey level and returns, in ORIG, a 1D matrix representing the original image flags.

CALL DECOM_GREY(IMAGE(I,J),ORIG)

PSUEDO_GREY is a buildup of identified greys, which once it equals IMAGE(I,J), a full match has been found. This is needed since we could match in more than one direction correctly. Continue matching until PSEUDO_GREY is satisfied or all directions checked.

PSUEDO_GREY=0

Check left, same row...

IF (IMTE(I2,J-1,5).NE.0) GOTO 140

Check left, row above...

110 IF (IMTE(I1,J-1,5).NE.0) GOTO 150

Check above, row above...

120 IF (IMTE(I1,J,5).NE.0) GOTO 160

Check right, row above...

130 IF (IMTE(I1,J+1,5).NE.0) GOTO 170

GOTO 400

The following set JUMP parameters from the above checks, so that the next position can be checked if PSEUDO_GREY is unfulfilled.

140 I3=I2
    J3=J-1
    JUMP=1
    GOTO 180

150 I3=I1
    J3=J-1
    JUMP=2
    GOTO 180

160 I3=I1
    J3=J
    JUMP=3
    GOTO 180

170 I3=I1
    J3=J+1
    JUMP=4
CONTINUE

Check, quick and dirty, if a complete match is made from checked pixel to the current one. If so, ignore all previous matches and substitute, on a one-for-one basis, the matches in IMTE(I3,J3,x).

IF (IMTE(I3,J3,5).EQ.IMAGE(I,J)) GOTO 190

No quick and dirty. Decompose the grey in the appropriate pixel...

CALL DECOM_GREY(IMTE(I3,J3,5),ORIG2)

Now check the grey, matrix element by matrix element, against ORIG (ie the pixel greys to be added to some particle, somewhere).

DO 300 ICNT=1,4
  IF (IMTE(I2,J,ICNT).NE.0) GOTO 300
  IF ((ORIG(ICNT).EQ.0).OR.(ORIG2(ICNT).EQ.0)) GOTO 300
A match has been made; update PSEUDO_GREY, and set IMTE(I2,J,ICNT) equal to the new grey, IMTE(I3,J3,ICNT).

PSEUDO_GREY=PSEUDO_GREY+2**(ICNT-1)
IMTE(I2,J,ICNT)=IMTE(I3,J3,ICNT)
300 CONTINUE

Quick; has PSEUDO_GREY been completed?

IF (PSEUDO_GREY.EQ.IMAGE(I,J)) GOTO 320
Otherwise, jump back and check rest of directions.

IF (JUMP.EQ.1) GOTO 110
IF (JUMP.EQ.2) GOTO 120
IF (JUMP.EQ.3) GOTO 130

OK, PSEUDO_GREY has not been completely filled, ie a new particle area must be created. Find out colors/areas that need to be created.

CALL DECOM_GREY(PSEUDO_GREY,ORIG2)
DO 330 ICNT=1,4
  IF (ORIG(ICNT).EQ.0) GOTO 330
  IF (ORIG(ICNT).EQ.ORIG2(ICNT)) GOTO 340
One last check; two to right and above. A perfect sphere, when digitized, will have this sort of structure. If the grey being created already exists at (I1,J+2), then
check at \(I_2, J+1\) for the same grey - ie a continuous path. Otherwise, ignore and create a new particle.

CALL DECOM_GREY(IMTE(I1,J+2,5),ORIG3) IF (ORIG(ICNT).EQ.ORIG3(ICNT)) GOTO 301

CALL DECOM_GREY(IMTE(12,J+1,5),ORIG4) IF (ORIG4(ICNT).NE.ORIG3(ICNT)) GOTO 302

IMTE(I2,J,ICNT)=IMTE(I1,J+2,ICNT)

IMTE(I2,J,ICNT)=AREA_COUNT GOTO 300

CALL DECOM_GREY(IMTE(I1,J+2,5),ORIG3) IF (ORIG(ICNT).EQ.ORIG3(ICNT)) GOTO 301

IMTE(I2,J,ICNT)=IMTE(I1,J+2,ICNT)

CALL UPDATE_AREA(I,J,IMTE(I2,J,ICNT),AREA) GOTO 200

Update IMTE storage.

DO 310 ICNT=1,4 IMTE(I2,J,ICNT)=IMTE(I3,J3,ICNT) GOTO 200

Complete set of grey level matches, ie no new particles found for a pixel. Update appropriate areas.

DO 350 ICNT=1,4 IF (ORIG(ICNT).EQ.0) GOTO 350 CALL UPDATE_AREA(I,J,IMTE(I2,J,ICNT),AREA) GOTO 200

WRITE(6,500)I,AREA_COUNT FORMAT(' After ',I4,' lines, ',I4,' areas have been ',1 'found') CONTINUE

Image has now been reduced to identified areas. Check against some preliminary idea of what is being identified, ie against MIN_PIX and MAX_PIX (set at beginning of program), and for a roughly spherical shape...


PSEUDO_AREA represents area of square bounded by AREA's
C min's and max's. This is comparable to NUM_PIX, knowing
C roughly what the expected shape is.
C TOL is used to set the value of the shape parameter;
C A perfect circle, for example, would be a TOL of
C 0.785 ((pi*d^2/4)/d^2, or (pi/4)). Since this is a
C finite world, a less restrictive TOL is required
C (I generally use 0.5, which would allow a 3x3 particle,
C identified as a '+', to pass (ie 5/9)).
C
PSEUDO_AREA=(AREA(I).MAXHOR-AREA(I).MINHOR)
PSEUDO_AREA=PSEUDO_AREA*(AREA(I).MAXVER-AREA(I).MINVER)
IF ((TOL*PSEUDO_AREA).LT.(AREA(I).NUM_PIX)) GOTO 911
XEXTENT=(AREA(I).MAXHOR-AREA(I).MINHOR+1)
YEXTENT=(AREA(I).MAXVER-AREA(I).MINVER+1)
IF ((MAX(XEXTENT,YEXTENT)/MIN(XEXTENT,YEXTENT)).GT.6) 1 GOTO 911
1 GOTO 911

C OK, passes all tests... ready for output to FOR008.DAT
C
COUNT(AREA(I).COLOR)=COUNT(AREA(I).COLOR)+1
XC=FLOAT(AREA(I).MAXHOR+AREA(I).MINHOR)/2
IF (AREA(I).COLOR.NE.4) GOTO 202
202 YC=FLOAT(AREA(I).MAXVER+AREA(I).MINVER)/2
X=AREA(I).COLOR
Y=COUNT(AREA(I).COLOR)
SA(X,Y).XC=XC
SA(X,Y).YC=YC
SA(X,Y).NUM_PIX=AREA(I).NUM_PIX
IF (X.EQ.1) GOTO 921
921 SA(X,Y).NEXT=COUNT((AREA(I).COLOR)-1)
SA(X,Y).AVAIL=1
SA(X,Y).LINK=0
SA(X,Y).ACT=I
MIH=AREA(I).MINHOR
MXH=AREA(I).MAXHOR
MIV=AREA(I).MINVER
MXV=AREA(I).MAXVER
COL=AREA(I).COLOR
NUM=AREA(I).NUM_PIX
WRITE(8,945) I,COL,NUM,XC,YC,MIH,MXH,MIV,MXV
945 FORMAT(′ ′,3I6,2F10.1,4I6)
GOTO 910
911 XC=FLOAT(AREA(I).MAXHOR+AREA(I).MINHOR)/2
YC=FLOAT(AREA(I).MAXVER+AREA(I).MINVER)/2
MIH=AREA(I).MINHOR
MXH=AREA(I).MAXHOR
MIV=AREA(I).MINVER
MXV=AREA(I).MAXVER
COL=AREA(I).COLOR
NUM=AREA(I).NUM_PIX
WRITE(9,945) I,COL,NUM,XC,YC,MIH,MXH,MIV,MXV
920   FORMAT( ' ',2I6,2I8)
930   FORMAT( ' ',2F8.1)
940   FORMAT( ' ',2I6)
950   FORMAT( ' ',2I6,/)  
910   CONTINUE
     WRITE(8,945) 0,0,0,0.0,0.0,0,0,0,0
     WRITE(8,920)0,0
     WRITE(8,961)NUMA
     WRITE(9,920)0,0,0
     WRITE(9,961)NUMB
961   FORMAT( ' ',I6)
     STOP
     END
SUBROUTINE DECOM_GREY(GREY_LEVEL, ORIG)

C Subroutine to decompose the pixel value into its constituent grey level components, namely 1, 2, 4 and 8. These are returned as flags in the ORIG matrix.

C

INTEGER GREY_LEVEL, ORIG(8), GTEMP
GTEMP = GREY_LEVEL
DO 10 I = 1, 4
   IF (GTEMP .LT. (2**(4-I))) GOTO 20
   GTEMP = GTEMP - (2**(4-I))
   ORIG(5-I) = 1
   GOTO 10
20   ORIG(5-I) = 0
10   CONTINUE
END
SUBROUTINE START_NEW_AREA(I, J, AREA_COUNT, GREY, AREA)

C Subroutine to start a new area, and initialize its values.

C

STRUCTURE /AREA_TYPE/
    INTEGER MINHOR, MAXHOR
    INTEGER MINVER, MAXVER
    INTEGER NUM_PIX
    INTEGER COLOR
END STRUCTURE

RECORD /AREA_TYPE/ AREA(7000)

INTEGER AREA_COUNT, GREY
AREA_COUNT = AREA_COUNT + 1
AREA(AREA_COUNT).MINHOR = J
AREA(AREA_COUNT).MAXHOR = J
AREA(AREA_COUNT).MINVER = I
AREA(AREA_COUNT).MAXVER = I
AREA(AREA_COUNT).NUM_PIX = 1
AREA(AREA_COUNT).COLOR = GREY

END
SUBROUTINE UPDATE_AREA(I, J, AREA_NUM, AREA)

C Subroutine to update an area and its parameters.

C

STRUCTURE /AREA_TYPE/
    INTEGER MINHOR, MAXHOR
    INTEGER MINVER, MAXVER
    INTEGER NUM_PIX
    INTEGER COLOR
END STRUCTURE

RECORD /AREA_TYPE/ AREA(7000)

INTEGER AREA_NUM, IVAL
IVAL = AREA_NUM
AREA(IVAL).NUM_PIX = AREA(IVAL).NUM_PIX + 1
END
Appendix B
This program takes particle location data (formatted output from PART_ID.FOR) and compiles likely vector matches. It reads sequential matching parameters from a file VECTCONTROL.DAT, and will attempt a 3 point match directly (1->2->3). Output is directed to FOR090.DAT (vectors) and FOR099.DAT (summary).

The structure /SORTED_AREA_TYPE/ is used to hold the particle information found from PART_ID.FOR. The array SA(4,1000) is of /SORTED_AREA_TYPE/. The first element in the array (1-4) indicates the 'grey' of the particle, while the second is that particles relative location within its own sorted list. Because of the nature of the particle identification, the particles are already roughly sorted from top to bottom by Yc, where Xc and Yc are located particle centers.

NEXT is an integer pointer to the element in the next grey that would be the particle immediately above the current Yc of the current color. AVAIL is a flag to indicate the particle has or has not been matched. LINK is an integer pointer at the next particle in a vector, ie to the next color. Because the algorithm is checking 1 --> 2 --> 3, if AVAIL is false (ie 0) for SA(1,any), then SA(1,any).LINK has some non-zero value, and SA(1,SA(2,any).LINK).LINK also has some non-zero value (since a three-point match is required).

STRUCTURE /SORTED_AREA_TYPE/
  REAL Xc,Yc
  INTEGER NUM_PIX
  INTEGER NEXT,AVAIL,LINK,ACT
  INTEGER ICNVC
END STRUCTURE
RECORD /SORTED_AREA_TYPE/  SA(4,1000),SA_TWO(4,1000),
                              SA_CORRECT(4,1000)
  CHARACTER*20 FILENAME
  CHARACTER*30 FILEVC
  INTEGER I,J,I2,J2,I3,J3,COUNT(4),MAXDIS,DIRECT,ICNVC
  REAL TOLV,TORL,BEST_TOLR,BEST_TOLV,BEST_TEST,BEST_XPTP,WF
  INTEGER BEST_MAXDIS,SUPERPASS

To prevent data corruption, data file outputs from PARTICLE_ID were renumbered to whatever seemed appropriate at the time. FOR008, FOR031, FOR041, and FOR051 were all commonly used; this program is not limited to these values.

WRITE(6,671)
33

671  FORMAT(' Enter 10 number for particle data')
WRITE(6,672)
672  FORMAT(8 for NASA, 51 for TURB, 31 for SIMULL',
1    ' and 41 for SIMULR')
READ(5,*) IO2

C At one point, the various iterations were being sent
C to separate IO values. These IO values were being
C recorded in FOR001.DAT, so later consolidation could be
C done. This statement is almost useless without
C the separation of IO values, but required for operation
C of the consolidation programs (ie, VECTORT.FOR).
C Writing IO2 to FOR001.DAT lets VECTORT.FOR know what
C data file is being consolidated (changing output file names
C correspondingly).

WRITE(1, 1010)IO2

C Read in particle data.

C Note secondary arrays, SA_TWO and SA_CORRECT. SA_TWO is
C reserved for backwards (in the list) matching, and SA_CORRECT
C is reserved for matches between SA and SA_TWO

DO I=1,4
  DO J=1,1000
    SA(I,J) .LINK=0
    SA_TWO(I,J) .LINK=0
    SA_CORRECT(I,J) .LINK=0
    SA(I,J) .ACT=0
    SA_TWO(I,J) .ACT=0
    SA_CORRECT(I,J) .ACT=0
  END DO
END DO

C Read in data point (format given in PARTICLE_ID).

40 READ(IO2,*) I,COL,XC,YC
  IF (IO2.EQ.31) COL=COL-1
  IF (I.EQ.0) GOTO 30
  COUNT(COL)=COUNT(COL)+1
  X=COL
  Y=COUNT(COL)
  SA(X,Y) .XC=XC
  SA(X,Y) .YC=YC
  SA(X,Y) .NUM_PIX=NUM
  SA_TWO(X,Y) .XC=SA(X,Y) .XC
  SA_TWO(X,Y) .YC=SA(X,Y) .YC
  SA_TWO(X,Y) .NUM_PIX=SA(X,Y) .NUM_PIX
  IF (X.EQ.4) GOTO 20
  SA(X,Y) .NEXT=COUNT(COL+1)
Vector determination. Tolerances are set by the file VECTCONTROL, which has one line per iteration.

IO=90
WRITE(1,1010) IO
ITER=0
TYPE*, 'Do you want preset VECTCONTROL.DAT info? (1/0)'
READ(5,*) IVC
FILEVC='VECTCONTROL.DAT'
IF (IVC.NE.1) THEN
   TYPE*, 'Please enter control file data name',
   1  ' (usually VECTCONTROL.DAT or FOR003.DAT)'
   READ(5,5050) FILEVC
END IF
5050 FORMAT(A30)

Read in VECTCONTROL file, which contains the parameters needed for particle vector matching. Each line in vectcontrol is executed sequentially on the particle list. Order of parameters is TolV, TolR, and MAXDIS. The file must end with the line, "0.0 0.0 0".

Sample VECTCONTROL.DAT file:

0.5 0.5 10
0.5 0.5 20
0.5 1.0 30
0.5 1.5 30
0.0 0.0 0

ICNTVC=0

Superpass is an integer counter; basically, VECTCONTROL.DAT is opened twice; in the regular mode, the forward/backwards error correction is in place. In the superpass mode, the error correction is disabled. Superpass is implemented after the error correction, under the premise that at that point, the number of tracks are so few that overlap (and hence error and the need for error correction) is diminished.
SUPERPASS=0
SUPERPASS=SUPERPASS+1
OPEN(UNIT=2, FILE=FILEVC, STATUS='OLD')
TYPE*, '' , 'Iteration ', ICNTVC+1
CONTINUE
ICNTVC=ICNTVC+1
READ(2,*), TOLV, TOLR, MAXDIS
FORMAT(I4)
IF (TOLV.EQ.0.0) THEN
  IF (SUPERPASS.EQ.1) THEN
    CLOSE(UNIT=2)
    GOTO 734
  ELSE
    GOTO 778
  END IF
END IF
ITER=ITER+1

C Main routine for matching. Every iteration reduces the
C eligible particles for matching, so VECTCONTROL can use
C less and less restrictive parameters.
C DIRECT=1 for forward match (SA), DIRECT=-1 for backward
C match (SA_TWO).
C
DIRECT=1
CALL DO_VECT(COUNT, SA, TOLV, TOLR, MAXDIS, 1 DIRECT, ICNTVC)

C Record keeping; for simulated images, accurate
determination of correct and incorrect matching can
be done. For non-simulated images, you can only gather
matched/unmatched percentages.

C All percentages are based on the number of grey=1 particles
C initially present.

C TALLYVECT output is sent to the appropriate file.
C CALL TALLYVECT(SA, COUNT(1), ICNTVC, 1, TOLV, TOLR, MAXDIS)

C Backwards matching (SA_TWO, DIRECT=-1).
C
DIRECT=-1
CALL DO_VECT(COUNT, SA_TWO, TOLV, TOLR, MAXDIS, 1 DIRECT, ICNTVC)

C Tally up number found
C CALL TALLYVECT(SA_TWO, COUNT(1), ICNTVC, 2, TOLV, TOLR, MAXDIS)
Time to start filling SA_CORRECT. This is done by comparing vector matches made in SA and SA_TWO; if they agree, they are considered correct and the entry made in SA_CORRECT. If they do not agree, the particles are made available for further matching.

IF (SUPERPASS.EQ.1) THEN
  DO I=1,COUNT(1)

Check that the match was made in the most recent iteration (otherwise already accounted for)...

IF (SA(1,I).ICNTVC.EQ.ICNTVC) THEN

Check that Ith particle is matched...

IF (((SA(1,I).LINK.NE.0).AND.(SA_TWO(1,I).LINK.NE.0))
  THEN

Check if Ith particles are equivalent... if so, update SA_CORRECT.

IF ((SA(1,I).LINK.EQ.SA_TWO(1,I).LINK).AND.
  (SA(2,SA(1,I).LINK).LINK).LINK.EQ.
  SA_TWO(2,SA_TWO(1,I).LINK).LINK)) THEN
  SA_CORRECT(1,I).LINK=SA(1,I).LINK
  SA_CORRECT(1,I).ACT=SA(1,I).ACT
  SA_CORRECT(1,I).ICNTVC=SA(1,I).ICNTVC
  SA_CORRECT(2,SA(1,I).LINK).LINK=SA(2,SA(1,I)
  .LINK).LINK
  SA_CORRECT(2,SA(1,I).LINK).ACT=SA(2,SA(1,I)
  .LINK).ACT
  SA_CORRECT(3,SA(2,SA(1,I).LINK).LINK).ACT=
  SA(3,SA(2,SA(1,I).LINK).LINK).ACT
  SA_CORRECT(3,SA(2,SA(1,I).LINK).LINK).LINK=
  SA(3,SA(2,SA(1,I).LINK).LINK).LINK
END IF
END IF
END IF

Update SA array for bad matches...

IF (((SA(1,I).LINK.NE.0).AND.(SA(1,I).ICNTVC.EQ.ICNTVC)
  .AND.(SA_CORRECT(1,I).LINK.EQ.0)) THEN
  SA(3,SA(2,SA(1,I).LINK).LINK).AVAIL=1
  SA(2,SA(1,I).LINK).LINK=0
  SA(2,SA(1,I).LINK).AVAIL=1
  SA(1,I).LINK=0
  SA(1,I).ICNTVC=0
  SA(1,I).AVAIL=1
C Update SA_TWO array for bad matches...

   SA_TWO(3,SA_TWO(2,SA_TWO(1,I).LINK).LINK).AVAIL=1
   SA_TWO(2,SA_TWO(1,I).LINK).LINK=0
   SA_TWO(2,SA_TWO(1,I).LINK).AVAIL=1
   SA_TWO(1,I).LINK=0
   SA_TWO(1,I).ICNTVC=0
   SA_TWO(1,I).AVAIL=1
END IF
END DO
ELSE
   DO I=1,COUNT(1)
      IF (SA(1,I).ICNTVC.EQ.ICNTVC) THEN
         C Check that Ith particle is matched.
         IF (SA_TWO(1,I).LINK.NE.0) THEN
            SA_CORRECT(1,I).LINK=SA(1,I).LINK
            SA_CORRECT(1,I).ACT=SA(1,I).ACT
            SA_CORRECT(1,I).ICNTVC=SA(1,I).ICNTVC
            SA_CORRECT(2,SA(1,I).LINK).LINK=SA(2,SA(1,I).LINK)
            SA_CORRECT(2,SA(1,I).LINK).ACT=SA(2,SA(1,I).LINK)
            SA_CORRECT(3,SA(2,SA(1,I).LINK).ACT=SA(3,SA(2,SA(1,I).LINK)
            SA_CORRECT(3,SA(2,SA(1,I).LINK).LINK=SA(3,SA(2,SA(1,I).LINK)
         ELSE IF (SA_TWO(1,I).LINK.NE.0) THEN
            SA_CORRECT(1,I).LINK=SA_TWO(1,I).LINK
            SA_CORRECT(1,I).ACT=SA_TWO(1,I).ACT
            SA_CORRECT(1,I).ICNTVC=SA_TWO(1,I).ICNTVC
            SA_CORRECT(2,SA_TWO(1,I).LINK).LINK=SA_TWO(2,SA_TWO(1,I).LINK)
            SA_CORRECT(2,SA_TWO(1,I).LINK).ACT=SA_TWO(2,SA_TWO(1,I).LINK)
            SA_CORRECT(3,SA_TWO(2,SA_TWO(1,I).LINK).LINK=SA_TWO(3,SA_TWO(2,SA_TWO(1,I).LINK)
         END IF
      END IF
      IF ((SA(1,I).LINK.EQ.0).AND.(SA(1,I).ICNTVC.EQ.ICNTVC).AND.(SA_TWO(1,I).LINK.NE.0)) THEN
         SA(3,SA(2,SA(1,I).LINK).LINK).AVAIL=SA_TWO(3,
I
1   SA_TWO(2,SA_TWO(1,I).LINK).LINK).AVAIL
SA(2,SA(1,I).LINK).LINK=SA_TWO(2,SA_TWO(1,I).LINK)
1
   .LINK
   SA(2,SA(1,I).LINK).AVAIL=SA_TWO(2,SA_TWO(1,I)
1
   .LINK).AVAIL
   SA(1,I).LINK=SA_TWO(1,I).LINK
SA(1,I).ICNTVC=SA_TWO(1,I).ICNTVC
SA(1,I).AVAIL=SA_TWO(1,I).AVAIL
END IF

C C Update SA_TWO array for bad matches...
C
IF ((SA_TWO(1,I).LINK.EQ.0).AND.(SA_TWO(1,I).ICNTVC
1
   .EQ.ICNTVC).AND.(SA(1,I).LINK.NE.0)) THEN
   SA_TWO(3,SA_TWO(2,SA_TWO(1,I).LINK).LINK).AVAIL=
1
   SA(3,SA(2,SA(1,I).LINK).LINK).AVAIL
   SA_TWO(2,SA_TWO(1,I).LINK).LINK=SA(2,SA(1,I).LINK)
1
   .LINK
   SA_TWO(2,SA_TWO(1,I).LINK).AVAIL=SA(2,SA(1,I).LINK)
1
   .AVAIL
   SA_TWO(1,I).LINK=SA(1,I).LINK
SA_TWO(1,I).ICNTVC=SA(1,I).ICNTVC
SA_TWO(1,I).AVAIL=SA(1,I).AVAIL
END IF
END IF

C C Determine %'s based on cumulative tracking/combination.
C
CALL TALLYVECT(SA_CORRECT,COUNT(1),ICNTVC,3,
1
   TOLV,TOLR,MAXDIS)
C
C Final piece of information: determine %'s based on most recent
C iteration/combination, non-cumul. Algorithm similar to
C TALLYVECT.
C
INOT_FOUND=0
IWRONG=0
ITREE_PT=0
ICNTNEW=0
DO I=1,COUNT(1)
   IF ((SA(1,I).ICNTVC.EQ.ICNTVC).OR.(SA(1,I).ICNTVC.EQ.0))
1
   THEN
      ICNTNEW=ICNTNEW+1
   IF ((SA_CORRECT(1,I).LINK).EQ.0) THEN
      INOT_FOUND=INOT_FOUND+1
   ELSE
      IA=SA_CORRECT(1,I).ACT
      IB=SA_CORRECT(2,SA_CORRECT(1,I).LINK).ACT
      IC=SA_CORRECT(3,SA_CORRECT(2,SA_CORRECT(1,I).LINK)
1
      .LINK).ACT
IX = SA_CORRECT(1, I) . LINK
IY = SA_CORRECT(2, IX) . LINK
WRITE(IO, 120) I, IA, IB, IC, SA(1, I). XC, SA(1, I). YC,
1 SA(2, IX) . XC, SA(2, IX) . YC, SA(3, IY) . XC, SA(3, IY) . YC
IF ((IA . EQ. IB) . AND. (IB . EQ. IC)) THEN
ITHREE_PT = ITHREE_PT + 1
ELSE
IWRONG = IWRONG + 1
ENDIF
END IF
END DO
IF (ICNTNEW . NE. 0) THEN
XPNF = FLOAT(INOT_FOUND) * 100.0 / FLOAT(ICNTNEW)
XPTP = FLOAT(ITHREE_PT) * 100.0 / FLOAT(ICNTNEW)
XPWR = FLOAT(IWRONG) * 100.0 / FLOAT(ICNTNEW)
WRITE(99, 561) ITER, MAXDIS, TOLV, TOLR, XPNF, XPTP, XPWR, ICNTNEW
WRITE(6, 561) ITER, MAXDIS, TOLV, TOLR, XPNF, XPTP, XPWR, ICNTNEW
END IF
GOTO 777
561 FORMAT(' Iteration ', I3, ', MAXDIS = ', I4, ', TOLV = ', F5.2,
1 TOLR = ', F5.2, ', Not Found = ', F6.2, ',
2 Three = ', F6.2, ', Wrong = ', F6.2, 
3 (Combined track) = ', I4, ' remaining tracks')

C Out of loop. Write final values to output and end.
C
778 WRITE(1, 1010) 0
C Final record-keeping is to output all non-matched particles to
C an output file (FOR020.DAT is used here).
C
IO_NF = 20
DO I = 1, COUNT(1)
IF (SA(1, I). AVAIL . EQ. 1) THEN
WRITE(IO_NF, 320) I, 3, SA(1, I). XC, SA(1, I). YC, SA(1, I). ACT
ELSE
WRITE(IO, 120) I, SA(1, I). ACT, SA(2, SA(1, I). LINK). ACT,
1 SA(3, SA(2, SA(1, I). LINK). LINK). ACT, SA(1, I). XC,
2 SA(1, I). YC, SA(2, SA(1, I). LINK). XC, SA(2, SA(1, I)
4 SA(3, SA(2, SA(1, I). LINK). LINK). YC
END IF
END DO
WRITE(IO, 120) 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
120 FORMAT(' ', 4I8, 6F10.2)
DO I = 1, COUNT(2)
IF (SA(2, I). AVAIL . EQ. 1) WRITE(IO_NF, 320) I, 2, SA(2, I). XC,
1 SA(2, I). YC, SA(2, I). ACT
END DO
DO I = 1, COUNT(3)
IF (SA(3,I).AVAIL.EQ.1) WRITE(IO_NF,320) I,3,SA(3,I).XC,
1     SA(3,I).YC,SA(3,I).ACT
END DO
WRITE(IO_NF,320) 0,0,0,0,0
320 FORMAT(' ',2I8,2F10.2,I8)
STOP
END
SUBROUTINE DO_VECT (COUNT, SA, TOLV, TOLR, MAXDIS,
  DIRECT, ICNTVC)

Subroutine that does the grunt work of particle matching,
using the parameters passed from the main program.

STRUCTURE /SORTED_AREA_TYPE/
  REAL XC, YC
  INTEGER NUM_PIX
  INTEGER NEXT, AVAIL, LINK, ACT
  INTEGER ICNTVC
END STRUCTURE
RECORD /SORTED_AREA_TYPE/ SA(4, 1000)
INTEGER COUNT(4), DJ, DK, DL, DIRECT, DI, ICNTVC, ISWITCH

Set tolerances and limits (passed variables)
TOLR is maximal radial change in vector (in radians)
TOLV is maximal magnitude change in vector (as fraction of
average vector magnitude)
MAXDIS is maximum displacement frame-to-frame of particles

Begin main loop, on grey=4. Start at 1 and go to end.
Do not match grey=2 or above if grey=1 particle is not
available.

ISWITCH = 0
IF (DIRECT.EQ.1) THEN
  ID1 = 1
  ID2 = COUNT(1)
  DI = 1
ELSE
  ID1 = COUNT(1)
  ID2 = 1
  DI = -1
END IF
DO 110 I = ID1, ID2, DI
  IF (SA(1, I).AVAIL.EQ.0) GOTO 110
C
C Switching parameters. A double-switching search is used;
initial search is in the above direction (ie start at
SA(1, I).NEXT and decrement back to 1), and switches to the
below direction (start at SA(1, I).NEXT+1 and increment up
to max) if no match is found.
C
250 IF (DIRECT.NE.1) GOTO 50
  DJ = -1
  I3 = 1
  I2 = SA(1, I).NEXT+1
  IF (I2.GT.COUNT(2)) I2 = COUNT(2)
GOTO 60
50  DJ=1
   J3=COUNT(2)
   J2=SA(1,I).NEXT
   IF (J2.LT.1) J2=1

C Main grey=2 loop.

60  DO 20 J=I2, I3, DJ

C Check availability. Also check MAXDIS (quickest elimination).
   IF (SA(2,J).AVAIL.EQ.0) GOTO 20
   DY1=(SA(1,I).YC-SA(2,J).YC)
   IF (ABS(DY1).GT.MAXDIS) GOTO 10
   DX1=(SA(1,I).XC-SA(2,J).XC)
   IF (ABS(DX1).GT.MAXDIS) GOTO 20
   DV1=SQRT(DX1*DX1+DY1*DY1)
   IF (DV1.EQ.0) GOTO 20

C DV1 rep. 1->2 vector
   IF (DJ.EQ.1) GOTO 80

C Switching parameters for the grey=3 loop, following the same procedure as for grey=2. It is likely that the grey=3 particle is in the same direction as grey=2, and this direction is checked first. ISWITCH is used to reverse the grey=3 track locally.

280  DK=-1
     J3=1
     J2=SA(2,J).NEXT+1
     IF (J2.GT.COUNT(3)) J2=COUNT(3)
     GOTO 70

80  DK=1
     J3=COUNT(3)
     J2=SA(2,J).NEXT
     IF (J2.LT.1) J2=1

C Main grey=3 loop.

70  DO 30 K=J2, J3, DK

C Check availability. Check MAXDIS for quick elimination.
   IF (SA(3,K).AVAIL.EQ.0) GOTO 30
   DY2=(SA(2,J).YC-SA(3,K).YC)
   IF (ABS(DY2).GT.MAXDIS) GOTO 31
   DX2=(SA(2,J).XC-SA(3,K).XC)
   IF (ABS(DX2).GT.MAXDIS) GOTO 30
   DV2=SQRT(DX2*DX2+DY2*DY2)
DV2 rep. 2->3 vector

IF (DV2.EQ.0) GOTO 30

Check TolV...

IF (ABS((DV2-DV1)/((DV2+DV1)/2)).GT.TOLV) GOTO 30

Check TolR (note 'heavy' calculations).

IF (DY1.GT.0) THEN
   DR1=ACOS(DX1/DV1)
ELSE
   DR1=2*3.14159-ACOS(DX1/DV1)
ENDIF
IF (DY2.GT.0) THEN
   DR2=ACOS(DX2/DV2)
ELSE
   DR2=2*3.14159-ACOS(DX2/DV2)
ENDIF
IF (ABS(DR2-DR1).GT.TOLR) GOTO 30

At this point a three-point match exists, 1 --> 2 --> 3.
Set the appropriate linkages and availability flags, and
check whether a fourth-point match is required. 4 point
matching tends to significantly reduce matching, and is
not used.

SA(3,K).AVAIL=0
SA(2,J).LINK=K
SA(2,J).AVAIL=0
SA(1,I).LINK=J
SA(1,I).AVAIL=0
SA(1,I).ICNTVC=ICNTVC
ISWITCH=0
GOTO 11

30 CONTINUE
31 IF (ISWITCH.EQ.1) THEN
   ISWITCH=0
   GOTO 20
ELSE
   ISWITCH=1
   IF (DJ.EQ.1) THEN
      GOTO 280
   ELSE
      GOTO 80
   END IF
END IF
20 CONTINUE
10 IF (DIRECT.EQ.DJ) GOTO 11
IF (DJ.EQ.1) GOTO 250
GOTO 50
11 CONTINUE
110 CONTINUE
RETURN
END
SUBROUTINE TALLYVECT(SA, IEND, ICNTVC, NUMB, TV, TR, MD)

C Subroutine to output some simple stats on matched particles

STRUCTURE /SORTED_AREA_TYPE/
   REAL XC, YC
   INTEGER NUM_PIX
   INTEGER NEXT, AVAIL, LINK, ACT
   INTEGER ICNTVC
END STRUCTURE

RECORD /SORTED_AREA_TYPE/ SA(4, 1000)

INTEGER IEND, ICNTVC, NUMB, MD, IO
REAL TV, TR

INOT_FOUND = 0
IWRONG = 0
ITHREE_PT = 0
IFOUR_PT = 0
DO I = 1, IEND
   IF ((SA(1, I) .LINK) .EQ. 0) THEN
      INOT_FOUND = INOT_FOUND + 1
   ELSE
      IA = SA(1, I) .ACT
      IB = SA(2, SA(1, I) .LINK) .ACT
      IC = SA(3, SA(2, SA(1, I) .LINK) .LINK) .ACT
      IX = SA(1, I) .LINK
      IY = SA(2, IX) .LINK
      IF ((IA .EQ. IB) .AND. (IB .EQ. IC)) THEN
         ITHREE_PT = ITHREE_PT + 1
      ELSE
         IWRONG = IWRONG + 1
      END IF
   END IF
END DO

XPNF = FLOAT(INOT_FOUND) * 100.0 / FLOAT(IEND)
XPTP = FLOAT(ITHREE_PT) * 100.0 / FLOAT(IEND)
XPWR = FLOAT(IWRONG) * 100.0 / FLOAT(IEND)

IF (NUMB .EQ. 1) THEN
   WRITE(99, 571) ITER, MD, TV, TR, XPNF, XPTP, XPWR, IEND
   WRITE(6, 571) ITER, MD, TV, TR, XPNF, XPTP, XPWR, IEND
END IF

IF (NUMB .EQ. 2) THEN
   WRITE(99, 572) ITER, MD, TV, TR, XPNF, XPTP, XPWR, IEND
   WRITE(6, 572) ITER, MD, TV, TR, XPNF, XPTP, XPWR, IEND
END IF

IF (NUMB .EQ. 3) THEN
   WRITE(99, 573) ITER, MD, TV, TR, XPNF, XPTP, XPWR, IEND
   WRITE(6, 573) ITER, MD, TV, TR, XPNF, XPTP, XPWR, IEND
END IF

571 FORMAT(' Iteration ', I3, ' MAXDIS - ', I4, ' TOLV - ', F5.2,
   1 ' TOLR - ', F5.2, '/', ' Not Found - ', F6.2, '/')
RETURN
END
Appendix C
The VIDEK MEGAPLUS™ Camera

**Feature**
- 1320 H x 1035 V pixel format
- Square pixels
- 100% fill (No space between pixels)
- Digital output

**Benefit**
- 4 times greater resolution than other "Hi-Res" solid-state cameras
- Accurate and simplified dimensional measurements in any direction
- 5 to 10 times improvement in sub-pixel measurement accuracy
- Maximizes signal-to-noise ratio
4MEG VIDEO Model 10
Flexible Image Processor and Application Accelerator
For The PC/AT

- Nonstandard Sensor Interface (up to 19 MHz)
- 10 MIPS Programmable Accelerator
- Up to 4 Megabytes of Reconfigurable Image Memory

Flexible Video Acquisition/Display

4MEG VIDEO Model 10 offers unparalleled flexibility in video acquisition and display. Analog video signals can be digitized at up to 19 million samples per second. Alternatively, the board can accept direct 8-bit digital data. The pixel clock can be derived from an external source or generated internally. 4MEG VIDEO Model 10 can genlock to composite video, composite sync, or horizontal and vertical drive signals from a variety of video formats. Likewise, a variety of output formats can be generated.

Programmable video resolution is provided by a unique Horizontal Control Memory (HCM) that allows pixel-by-pixel control over image memory transfers during acquisition and display. Integer zoom and subsampling are supported. Sampling resolution can vary across a given line of video. The vertical resolution is also programmable.

EPIX
310 Anthony Trail, Northbrook, IL 60062
(312) 498-4002
**4MEG VIDEO™ Model 10**

Flexible Image Processor and Application Accelerator For The PC/AT

**Fast On-Board Processor**

4MEG VIDEO Model 10 facilitates acceleration of image processing functions with a 10 MIPS Texas Instruments TMS320C25 Digital Signal Processor. In addition to performing math-intensive image processing functions at high speed, the TMS320C25 can function much like a general-purpose microprocessor. This facilitates the implementation of custom image processing and inspection algorithms.

The TMS320C25 is programmed in assembly language. The 8K word on-board program memory is ample for most image processing functions. Programs may be downloaded rapidly via the PC/AT bus.

**Reconfigurable Image Memory**

Pixel data is stored in image memory as sequential 8-bit values. Using the HCM, this memory can be organized as one large image, or many smaller images. Portions of the memory can also be used to store intermediate results, menus, or overlays. A bit-plane write protect feature allows text and overlays to be written over image data.

Image memory is accessed by the PC/AT or the TMS320C25 in 64K byte blocks. A Memory Offset Register allows this window to be located on any 16 Kbyte boundary.

The memory has data paths for video acquisition, video display, and processing. One of these paths is utilized at any given time. During image acquisition the display is driven with live video. Image processing can be performed during video blanking intervals to maintain an uninterrupted display.

**4MEG VIDEO Model 10 Software Support**

4MEG VIDEO Model 10 is supported by driver software that allows easy application development by OEMs and VARs. A driver subroutine library (4MDRIVER) simplifies control of the board from programs written in C or other high-level languages. The library is easy to use; familiarity with structures, pointers, etc. is not required for fundamental operations.

An optional, interactive, menu driven program (4MIP) allows immediate access to 4MEG VIDEO Model 10 functionality. 4MDRIVER functions and image processing routines can be invoked via a mouse interface. 4MIP also allows the user to save sequences of operations in command files that can be re-executed on demand.

To develop loadable routines for the TMS320C25, standard macro-assembler development tools from Texas Instruments or Avocet Systems, Inc. can be employed. 4MDRIVER and 4MIP facilitate downloading of user-developed programs.

**Specifications:**

**4MEG VIDEO Model 10 Hardware**

- **Video Acquisition/Display**
  - Up to 19 MHz sampling/display rate
  - Up to 1984 pixels per line
  - RS-170, RS-330, and CCIR input/output
  - Analog or digital inputs
  - Variable timing for nonstandard formats
  - Genlock to external timing sources
  - Generates master video timing
  - Software programmable timing/resolution
  - External input/output for event synchronization
  - Pseudocolor display
  - Nondestructive cursor
  - PC or AT bus compatible

- **On-Board Processor**
  - TMS320C25 digital signal processor
  - 10 million instructions per second
  - 16-bit fixed point arithmetic w/ 32-bit accumulation
  - 8K word program memory
  - Direct image memory access

- **Image Memory**
  - 1 megabyte or 4 megabyte image memory options
  - 8-bits per pixel
  - Selectable bit plane write protect
  - Configurable as one or many images
  - Programmable image size

- **Options**
  - VIDEK MEGAPLUS camera interface card
  - 16 input video multiplexor card
  - Reticon camera interface (third party)

**4MEG VIDEO Model 10 Software**

- **4MDRIVER**
  - Simple-to-use functions/subroutines
  - Image capture and display
  - Adjustable video formats
    - resolution
    - interlaced/noninterlaced
    - Serration/equalization
  - Selectable timing source
  - Split screen digitize/display
  - Integer zoom (1-30X)
  - Pan, Scroll
  - TMS320C25 convolution, image sequence average and difference

- **4MIP Only**
  - PC-based image processing
    - Arithmetic/logic functions
    - Convolution
    - Temporal average
    - Inter-image operations
    - Contrast enhancement
  - Real-Time motion analysis
  - Histogram display
  - Command file training/replay
  - Disk, File I/O
  - Image Printing

©1989 EPIX INC. EPIX and 4MEG VIDEO are trademarks of EPIX, INC. PC/AT is a trademark of IBM CORP.
GATEWAY 2000
33MHZ 486 CACHE

DESKTOP
64K SRAM CACHE (25NS)
8 MB DRAM (70 NS SIMMS)
EXPANDABLE TO 64MB
1.2 MB 5 1/4" DRIVE (EPSON)
1.44 MB 3.5" DRIVE (EPSON)
200 MB IDE WESTERN DIGITAL (15MS) 10 M/BITS DTR
W/64K MULTI SEGMENTED CACHE BUFFER
DIAMOND SPEEDSTAR 16 BIT VGA BOARD W/1MB (1024 X 768)
14" GATEWAY 2000 NON-INTERLACED 1024 CRYSTAL SCAN MONITOR (1024 x 768)
1 PARALLEL PORT/2 SERIAL PORTS
GATEWAY 2000 101 KEY KEYBOARD
MICROSOFT WINDOWS 3.0/MICROSOFT MOUSE
DOS 4.01 OR 3.3
WEITEK SOCKET/CLOCK/CALENDAR
PHOENIX BIOS

PRICE: $3995.00

HARD DRIVE OPTIONS:
40 MB IDE WESTERN DIGITAL (17MS) 10 M/BITS DTR
W/32K READ-LOOK-AHEAD CACHE BUFFER SUBTRACT: $ 525.00
80 MB IDE WESTERN DIGITAL (17MS) 10 M/BITS DTR SUBTRACT: $ 400.00
110 MB ESDI MICROSCIENCE (18MS) 10 M/BITS DTR SUBTRACT: $ 200.00
120 MB IDE SEAGATE (18MS) 10 M/BITS DTR SUBTRACT: $ 300.00
150 MB ESDI SEAGATE (17MS) 10 M/BITS DTR ADD: $ 250.00
300 MB ESDI SEAGATE (17MS) 10 M/BITS DTR ADD: $ 750.00
650 MB ESDI SEAGATE (17MS) 10 M/BITS DTR ADD: $1250.00

MONITOR OPTIONS:
12" SAMSUNG MONOCHROME SUBTRACT: $ 425.00
14" SAMSUNG VGA MONOCHROME SUBTRACT: $ 250.00
14" NEC 3D ADD: $ 225.00
14" SONY 1304 ADD: $ 250.00
16" NANAIO 9070U ADD: $ 800.00
20" MITSUBISHI 6935 ADD: $1900.00

OTHER OPTIONS:
VERTICAL CASE ADD: $ 150.00
120 MB TAPE BACK UP ADD: $ 295.00
2400 BAUD ATI INTERNAL MODEM ADD: $ 100.00
UPGRADE TO 16 MB ADD: $ 600.00
KX-P1124 PANASONIC PRINTER (INCLUDES CABLE) ADD: $ 295.00
KX-P1624 PANASONIC PRINTER (INCLUDES CABLE) ADD: $ 425.00

MASTER CARD, VISA, AMERICAN EXPRESS, DISCOVER (NO SURCHARGE)
C.O.D. CASHIER'S CHECK ACCEPTABLE (CONTINENTAL USA)
SHIPPING: (2 DAY DELIVERY) $95.00 IN THE CONTINENTAL USA
APO SHIPPING: $125.00
FULL LINE DISTRIBUTOR - PLEASE CALL FOR OTHER CONFIGURATIONS
NDP Fortran-386™ Optimized FORTRAN 77 with VAX/VMS, UNIX BSD 4.2, and MIL STD-1753 Extensions for 80386-based Systems

NDP Fortran-386 is a globally optimizing compiler that has been developed at MicroWay for the Intel 80386. It generates native 80386 code that runs in protected mode under UNIX 386 System V Release 3, SCO XENIX Release 2.3, and Phar Lap extended DOS. Separate releases are available for each operating system. The execution speed of code generated by NDP Fortran-386 is exceptionally fast. Recompiling existing 16-bit 80286 compiler code with the 32-bit NDP Fortran-386 can increase the speed of execution by 200-500%. When the MicroWay mW1167 or the Weitek 3167 numeric coprocessor is added, performance of the NDP Fortran-386 equals that of a VAX 8600, or 60 times the speed of an IBM PC.

NDP Fortran-386 makes it possible to port mainframe FORTRAN applications to your 80386 machine that use as much memory as your system will hold: the upper limit on segment size in the linear address mode is 4 gigabytes. The compiler generates code which optionally utilizes the Phar Lap Virtual Memory Manager, making it possible for a two megabyte 80386 system to run programs that are as large as the free memory on your hard disk! The compiler also supports any of five possible coprocessors for which your 80386 system is socketed, including the MicroWay mW1167, Weitek 3167, Intel 80387 and 80287, and Cyrix 83D87. The mW1167 provides two to four times the throughput of an 80387 as measured by popular benchmarks.

NDP Fortran-386 is a full implementation of FORTRAN 77 and includes the extensions needed to write new applications or port existing ones. These include the popular FORTRAN 66 extensions to FORTRAN 77, plus features added by the UNIX f77 portable FORTRAN compiler (including UNIX BSD 4.2 features), D.O.D., and DOS FORTRAN compilers. NDP Fortran-386 compiles most FORTRAN 66 and 77 applications without modification.

NDP Fortran-386 v.2.0 Features:

- 98% compatible with VAX/VMS extensions, including "NAMELIST."
- Incorporates several new optimizations, including loop unrolling, repeat common subexpression elimination, register caching, and peephole optimization.
- Includes a command line switch to allow the compiler to run in virtual memory.
- NDP-386 Virtual Memory option allows executable programs to run in virtual memory using the Phar Lap Virtual Memory Manager.
- Generates programs, procedures, and arrays limited only by the amount of memory in the system, up to 4 gigabytes.
- Exceptional runtime speed due to global optimizations, sophisticated register utilization to store 32-bit entities, use of inline 32-bit arithmetic instead of library calls, and the effective doubling of the system data bus.
- Simplifies the porting of existing applications by fully implementing FORTRAN 77 (full language) as specified by ANSI X3.9-1978, the D.O.D. supplement to FORTRAN 77 (MIL STD-1753), and the documented and undocumented extensions to the Berkeley 4.2 BSD UNIX f77 compiler for VAX/VMS.
- Generates inline code for coprocessors which makes excellent use of all numeric registers.
- Supports full 80387 and mW1167/3167 numeric instruction sets, including 80387 inline transcendentals.
- Allows customized coprocessor exception handling procedures to be designed and implemented by the user. (examples are provided)
- Includes a library of graphics and keyboard routines with enhanced features that supports the CGA, MDA, EGA VGA, and Hercules adapters. (DOS version only)
- Includes mouse support.
- Provides a trace facility to aid in debugging.
- Memory mapped devices and physical memory can be mapped into the program's linear address space.
- NDP Fortran-386 can call or be called from NDP C-386 or NDP Pascal-386 programs. Assembly language routines can be interfaced with compiled output.
- The f77 compiler driver makes it possible to use the same switches for compiling, assembling, and linking when working with DOS, UNIX V, or XENIX 2.3.
- Fast I/O feature in DOS version makes it possible to specify the size and number of runtime buffers, resulting in an I/O speed up that ranges up to 15 times faster.
- Can be used with the newest generation of Phar Lap tools to produce embedded and ROMable code.
- NDP Plot is an optional Calcomp-compatible package including high-level plotting and 3-D graphics routines.
- NDP to Halo '88 Interface is an optional graphics interface to Media Cybernetics Halo '88™.
- NDP Hoops is an optional advanced object-oriented graphics library.
Optimization Features:
The compiler converts the ASCII FORTRAN text one procedure at a time into a memory-based operator tree. During global optimization, this tree is traversed from 5 to 50 times depending upon the options selected and the structure of the code. The primary goal of the global optimization is to store variables in registers as opposed to memory. Eliminating stores and loads to memory, on average, results in code that runs a factor of 3 faster while taking only 1/3 the space of code which stores variables in memory. The global analysis takes into account variable lifetime, activity, size, and the benefits of using faster running 16-bit addressing modes over the slower running 32-bit modes where possible. The optimizer produces code which takes maximum advantage of the registers available in the numeric coprocessors that the compiler supports. The generation of very high quality inline numeric code is one of the outstanding features of the compiler.

The process of traversing the tree includes the application of code transformations to the tree. These transformations include numeric strength reduction, dead code elimination, removal of loop invariant code from loops, hoisting of common code out of blocks, constant propagation, elimination of stack frame setup on procedure entry where possible, conversion of small procedures into inline code where possible, and a number of processor-related peephole optimizations. Loop optimizations which make the code larger but faster can also be optionally performed. These optimizations rearrange loops so that array base values are computed outside of loops, and then stored in registers where they are used indirectly for addressing, and incremented, when necessary. The optimizer also performs global common subexpression elimination, caches array elements in registers, and unrolls short "hot" loops into inline code.

VAX/VMS FORTRAN Extensions:
- Symbolic Names may be 31 characters long and contain the $ character.
- Nested INCLUDEs are allowed up to 10 levels.
- IMPLICIT UNDEFINED (A-Z) turns off IMPLICIT typing.
- All MIL STD-1753 Binary and bit functions are supported including IOR, IAND, IEOR, NOT, ISHFT, ISHTC, ISHTL, ISHFR, BTSTE, IBSET, IBCLR, and MVBITS. Bessel, Gamma, and error functions are also included.
- Z and Q field descriptors allow octal and hexadecimal editing of I/O list items.
- Hollerith, hexadecimal, binary, and octal constants are supported.
- One trip DO LOOPs compatible with FORTRAN 66 can be optionally turned on.
- Free formatted input using commas is supported.
- The $ can be used to eliminate the carriage return that normally follows a read or write.
- Conditional compiles are signaled in column 1 by x, X, d, or D. Continuation lines in free format are indicated by &.
- The mixing of numeric and character data types in COMMON and EQUIVALENCE statements is allowed.
- INTERNAL files are expanded from SEQUENTIAL only to include DIRECT.
- Types include REAL*4, REAL*8, INTEGER*1, INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*2, LOGICAL*4, COMPLEX*8, and COMPLEX*16. Default of LOGICAL and INTEGER is 4 but can be changed to 2.

Other Key Words:
- ACCEPT statement
- NAMELIST
- ASSOCIATEVAR
- NOSPACE
- VARIABLE
- OPTIONS
- BYTE
- STATEMENT
- CARRIAGECONTROL
- ORGANIZATION
- DATE and IDATE
- PARAMETER
- DEFAULTFILE
- Q Edit descriptor
- DEFINE FILE
- READONLY
- DELET E statement
- RECORDSIZE
- DISPOSE
- RECORDTYPE
- DO...WHILE, extended range
- REWRITE statements
- ENCODE and DECODE
- SECNDS
- EXTENDSIZE
- STRUCTURE declaration
- FIND statement
- TIME
- %LOC, %REF, %VAL
- USEROPEN
- MAXREC
- VOLATILE and VIRTUAL
- NAME
- SYSTEM

UNIX/C-Like Features:
- Command line processing using getarg, iargc, and getenv are supported.
- Strings may be declared with quotes or apostrophes, and internal string members may be defined using backslash editing identical to C.
- Upper and lower case are supported. For compatibility with C, the compiler converts upper to lower except in strings; a compile time option shuts off this conversion.
- AUTOMATIC and STATIC variables in procedures make recursive procedures possible.
- Programs written with NDP Fortran-386 can call or be called from NDP C or Pascal-386. Assembly language routines can also be interfaced with the compiled output.

System Requirements:
- Any 80386-based system; or any PC, XT, AT, or compatible with an Intel Inboard/386 or MicroWay Number Smasher-386.
- A numeric coprocessor is not required to compile. However, an Intel 80287 or 80387, Cyrix 83D87, MicroWay mW1167, or Weitek 3167 coprocessor is necessary to execute programs containing floating point routines.
- Double-sided high density floppy drive.
- Hard disk drive with a minimum of two free megabytes.
- Two megabytes of extended memory (four megabytes recommended).
- DOS version 3.2 or later as extended by Phar Lap Development Tools (version 2.0 or later), UNIX 386 System V Release 3, or SCO XENIX Release 2.3.
NDP C-386™:  Globally Optimizing, Native Code C Compiler for the Intel, Cyrix, and Weitek Coprocessors

NDP C-386 is a globally optimizing compiler developed at MicroWay for the Intel 80386. It generates native 80386 code that runs in protected mode under UNIX 386 System V Release 3, SCO XENIX Release 2.3, and Phar Lap extended DOS. NDP C-386 makes it possible to port mainframe C applications to your 80386 that use as much memory as your system will hold: the upper limit on segment size in the linear address mode used is 4 gigabytes. The compiler also supports the following coprocessors: Intel 80287 and 80387, Cyrix 83D87, MicroWay/Weitek mW1167, and Weitek 3167, which have two to four times the throughput of an 80387.

The compiler generates code which optionally takes advantage of the Phar Lap Virtual Memory Manager. The latter makes it possible for a two megabyte 80386 system to run programs as large as the free memory on your hard disk!

The compiler is a full implementation of PCC (the Bell Labs Portable C Compiler, whose syntax is a superset of Kernighan and Ritchie C). It includes all standard PCC extensions as well as the December, 1988, draft of ANSI C and many Microsoft C v.5.0 functions. Among these new extensions are a set of graphics and BASIC-like screen handling functions, in addition to hooks to the operating system. These features make NDP C-386 compatible enough to compile most existing 16-bit applications, regardless of the source environment, provided they conform to standard techniques for portability between computers.

NDP C-386 v.2.0 Features:

- Passes 98% of the Plum Hall Validity Suite for ANSI System V UNIX C.
- Incorporates several new optimizations, including loop unrolling, repeat common subexpression elimination, register caching, and peephole optimization.
- Includes a command line switch to allow the compiler to run in virtual memory.
- NDP-386 Virtual Memory option allows executable programs to run in virtual memory using the Phar Lap Virtual Memory Manager.
- Generates programs, procedures, and arrays limited only by the amount of memory in the system, up to 4 gigabytes.
- Exceptional runtime speed due to global optimizations, sophisticated register utilization to store 32-bit entities, use of inline 32-bit arithmetic instead of library calls, and the effective doubling of the system data bus.
- Ports existing applications by fully implementing AT&T's PCC and its K & R subset.
- Generates inline code for coprocessors which makes excellent use of all numeric registers.
- Supports full 80387 and mW1167/3167 numeric instruction sets, including 80387 inline transcendental.
- Supports customized coprocessor exception handling procedures to be designed and implemented by the user (examples are provided).
- Allows function and variable names of 31 characters which may include the $ character.
- Includes the following types: 32-bit pointer and enum types along with 8 byte double, 4 byte floats, 4 byte longs, 4 byte int, 2 byte short int, 1 byte char.
- An extended error function gets (and optionally prints) DOS errors, mapping them separately through errno.
- Incorporates a library of graphics and keyboard routines with enhanced features that supports CGA, MDA, EGA, VGA, and Hercules adapters. (DOS version only)
- Includes mouse support.
- Provides a trace facility to aid in debugging.
- Memory mapped devices and physical memory can be mapped into the program’s linear address space.
- NDP C-386 can call or be called from NDP Fortran-386 or NDP Pascal-386 programs. Assembly language routines can be interfaced with compiled output.
- The CC compiler driver makes it possible to use the same switches for compiling, assembling, and linking when working with DOS, UNIX V, or XENIX 2.3.
- Command line processing includes the name of the current process.
- Fast I/O feature in DOS version makes it possible to specify the size and number of runtime buffers, resulting in an I/O speed up that ranges up to 15 times faster.
- Can be used with the newest generation of Phar Lap tools to produce embedded and ROMable code.
- NDP Windows is an optional library for creating menus and storing, moving, or saving windows. It runs on MDA, CGA, EGA, and Hercules adapters.
- NDP to Halo ‘88 Interface is an optional graphics interface to Media Cybernetics Halo ‘88’.
- NDP Hoops is an optional advanced object-oriented graphics library.
- MicroWay also has a port of the AT&T C++ preprocessor v.1.2 that runs in protected mode with the NDP C-386 compiler.
**Optimization Features:**
NDP C-386 converts the ASCII C text, one procedure at a time, into a memory-based operator tree. During global optimization, this tree is traversed from 5 to 50 times depending upon the options selected and the structure of the code. The primary goal of the global optimization is to store variables in registers as opposed to memory. Eliminating stores and loads to memory, on average, results in code that runs a factor of 3 faster while taking only 1/3 the space of code which stores variables in memory. The global analysis takes into account variable lifetime, activity, size, and the benefits of using faster running 16-bit addressing modes over the slower running 32-bit modes where possible. The optimizer produces code which takes maximum advantage of the registers available in the numeric coprocessors that the compiler supports. The generation of very high quality inline numeric code is one of the outstanding features of the NDP compilers.

The process of traversing the tree includes the application of code transformations to the tree. These transformations include numeric strength reductions, dead code elimination, removal of loop invariant code from loops, hoisting of common code out of blocks, constant propagation, elimination of stack frame setup on procedure entry where possible, conversion of small procedures into inline code where possible, and a number of processor related peephole optimizations. Loop optimizations which make the code larger but faster can also be optionally performed. These optimizations rearrange loops so that array base values are computed outside of loops and then stored in registers where they are used indirectly for addressing and incremented when necessary. The optimizer also performs global common subexpression elimination, caches array elements in registers, and unrolls short "hot" loops into inline code.

**Numeric Coprocessor Support:**
The NDP C-386 compiler provides different code generation for the following numeric coprocessors: Intel 80287, 80387, and 80387SX; Cyrix 83D87; and Weitek 1167 and 3167. NDP C-386 generates code to use the Weitek 3167 multiply and accumulate instruction.

NDP C-386 supports IEEE-754 floating point arithmetic. The compiler provides a complete set of functions which allows the programmer to read and change any value in the numeric coprocessor control register.

The NDP C-386 compiler contains a general purpose numeric exception handler. Under DOS, users can change the response characteristics of the default handler, or write their own customized handlers. The user manual includes two examples of user-written handlers: one traps division by zero and substitutes a very large number for infinity on the response characteristics of the default handler, or write and applications which take advantage of the underlying hardware. The NDP C-386 inline assembler is unusual in that it makes it possible to write assembly language in C! For example, to increment the EAX register in assembly language, you could write INC EAX. In NDP C-386, you simply declare EAX to be a register aliased variable of type unsigned, and use the conventional C statement, EAX++. The compiler translates this C code into its corresponding assembly language, INC EAX.

Register aliased variables come in very handy for reading and writing ports inline as well as setting up and using software interrupts inline. The following example puts the current directory path into the string "string":

```c
char string[64];
reg$eax unsigned eax;
reg$edx unsigned edx;
reg$esi char *esi;
eax = 0x4700;
edx = 0;
esi = string;
asm (eax, edx, esi, "int 21h");
printf ("Current directory is "%\n",string);
```

The code that results is inline, as opposed to the MS-DOS INT386 technique. The latter, which is also supported for compatibility, requires two data structures to be set up and a 50 line procedure (INT86) to be called.

**Graphics Support:**
The DOS version of the NDP C-386 compiler comes with a library of over 100 functions to draw pixels, lines, ellipses and text, move images and graphics cursors, read and write ports, and execute interrupts. Special routines are included to provide compatibility with the Microsoft C graphics library. The NDP C-386 graphics library works with the MDA, CGA, EGA, VGA, Super VGA, and Hercules graphics adapters. It includes routines to automatically detect the hardware configuration and determine the best graphics mode. Complete documentation is provided, including simple, clear examples.

**System Requirements:**
- Any 80386-based system; or any PC, XT, AT, or compatible with an Intel Inboard/386 or MicroWay Number Smasher-386.
- A numeric coprocessor is not required to compile. However, an Intel 80287 or 80387, Cyrix 83D87, MicroWay mW1167, or Weitek 3167 coprocessor is necessary to execute programs containing floating point routines.
- Double-sided high density floppy drive.
- Hard disk drive with a minimum of two free megabytes.
- Two megabytes of extended memory (four megabytes recommended).
- DOS version 3.2 or later as extended by Phar Lap Development Tools (version 2.0 or later), UNIX 386 System V Release 3, or SCO XENIX Release 2.3.