Laser-Scan Ltd.

COORDLIB - Coordinate Manipulation Library

Reference Manual

Issue 1.1 - 12-Apr-1990

Copyright (C) 2002 Laser-Scan Ltd
Cambridge Science Park, Milton Road, Cambridge, England CB4 4FY
telephone: (0223) 420414

Document "COORDLIB Reference" Category "REFERENCE"
Document Issue 1.0 R J Hulme 19-Nov-1986
Document Issue 1.1 P Pan 12-Apr-1990

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 *GENERAL DESCRIPTION*

COORDLIB is Laser-Scan's coordinate manipulation library.

It provides several sets of routines for common operations on graphical elements in Laser-Scan software. Currently, facilities are provided for the following :-

- curve and linear interpolation
- line filtering

The library may be found in LSL$LIBRARY:COORDLIB.OLB, and parameter files available for general use in LSL$CMNCOORD:

The library sources are in LSL$SOURCE_ROOT:[LSLMAINT.COORDLIB...], and the documentation sources are in LSL$DOC_ROOT:[LSLMAINT.COORDLIB].

1.2 *Documentation Notation*

The following conventions are followed:

* all arguments are fully declared for each routine.

* the following input/output declarations are made:

  out - this variable will be written to by the routine.

  in - this variable is read by the routine - it is not written to.

  i/o - this variable may be both read by the routine, and written to.

* the following argument types are used:

  word - this is a Fortran INTEGER*2, a 16 bit variable.

  long - this is a Fortran INTEGER*4, a 32 bit variable. Note that this is the default integer size on the VAX, and is assumed unless there are good reasons for using a word.

  logical - this is a Fortran LOGICAL variable

  byte - this is a Fortran BYTE or LOGICAL*1 variable

  real - this is a Fortran REAL variable

  dreal - this is a Fortran REAL*8, or DOUBLE PRECISION variable

  char - this is a Fortran CHARACTER variable, but see the section on 'Fake strings' in the System chapter

  external - this is a routine declared as EXTERNAL in the calling routine

* all arguments are compulsory.

* parameter files are described using the same data-types as used for variables (long, word, real, etc)

* none of the common blocks are intended to be accessed by the user and are therefore not documented

1.3 *Naming Conventions*

* all common block names are listed in Appendix B.

* all routine names are also listed in Appendix B.

* the source code for every Fortran routine is contained in its own separate file, filenames being the same as the routine names.

1.4 *Coordinate Annotation Conventions*

All coordinate arrays are 2-dimensional REAL**4**, the first index being either 1 or 2 for X and Y respectively, and the second the point number. The parameter file LSL$CMNCOORD:XYIND.PAR contains parameters for these indices :-

```
long    X         - X coordinate index (= 1)

long    Y         - Y coordinate index (= 2)
```

This parameter file is available to users for their own programs.

CHAPTER 2

INTERPOLATION

2.1 *INTRODUCTION*

Three interpolation algorithms are available, namely

i) Akima cubic interpolation,

ii) McConalogue cubic interpolation, and

iii) linear interpolation

Akima is a bicubic spline method which preserves linearity if possible, while McConalogue is a circular arc pair method. Each of has been used by Laser-Scan since 1976, Akima having been based on an algorithm in CACM (433).

The linear interpolation algorithm was originally intended simply for adding extra points to lines, and is a much later addition.

Whichever algorithm is being used, it is a general routine which controls the interpolation; this is passed the appropriate routine to interpolate a single span of the line.

2.2 *INITIALISATION*

call INTRPL_INIT(toler, sf, draw)

in	- real	**toler**	a 3 element real array containing tolerances, in the coordinate units - **toler(1)** is the constant separation for curve and linear interpolation - **toler(2)** is the arc to chord distance for curve interpolation - **toler(3)** is the angular deviation for curve interpolation
in	- real	**sf**	a scale factor (which should be 1.0 if dealing with IFF units)
in	- byte	**draw**	TRUE if the interpolated line is being drawn on the screen

Prior to interpolating any lines, INTRPL_INIT should be called to perform the necessary initialisation. It should subsequently be called again if any of the tolerances change or if there is a scale change.

A detailed explanation of the tolerances follows.

2.2.1 *Curve Interpolation Tolerances*

In the formula below, The three coefficients (a,b,c) control the spacing of interpolated points. The approximate separation of points (d) is given by:

d = a + 2*SQRT(2br) + cr (r is radius of curvature)

which means (if other coefficients were zero) that

a gives a constant separation of a IFF units
b gives a constant 'arc to chord' distance of b IFF units
c gives a constant angular deviation of c radians

2.2.2 *Linear Interpolation Tolerance*

The constant separation represents the minimum distance required between data points. If this is 0.0, one point will be inserted in the middle of the line segment regardless of segment length.

2.3 *INTERPOLATING A WHOLE LINE*

call INTRPL(n, xy, first, last, looped, extra, polate)

in	- long	**n**	the number of points in the line
in	- real	**xy**	the line to be interpolated, in the form of a 2***n** array
in	- byte	**first**	TRUE if the line includes the first point
in	- byte	**last**	TRUE if the line includes the last point
in	- byte	**looped**	TRUE if the line is looped
in	- byte	**extra**	TRUE if extra points have to be extrapolated for each section of the line (this is the case for Akima but not for the other methods)
in	- long	**polate**	the actual interpolation routine to use

INTRPL interpolates points between given master points in array **xy**, calling routine **polate** to do the interpolation. The actual subroutines used should be declared as EXTERNAL in the calling routine (the example in Appendix A illustrates how to do this). The interpolation routines currently available are AKIMA, MCONAL, and LINEAR, but users could of course supply their own; the specification for **polate** is given in the next section.

INTRPL, AKIMA, MCONAL and LINEAR all call the user-supplied subroutine ADDPTS to output interpolated points. The specification of ADDPTS is given in the next section; note that it may be called using part of the original line string passed to INTRPL.

The argument **extra** controls whether or not additional points are extrapolated for each section of the line as it is interpolated. Care should be taken to ensure that it is compatible with **polate** (i.e. TRUE for AKIMA).

Note that a call with **n** = 0 is valid, and can be used with **last** = TRUE to finish off a feature without adding more points.

After interpolation, closed loops start at the original third point, the first two appearing at the end. This is of no consequence if drawing on the screen.

Note that interpolation will not be carried out on two-point lines or three-point loops; this is an anomaly if linear interpolation is being performed.

2.4 *Routines Supplied by Users*

These routines are called by the library routines and should be linked with the library and other modules of the user's program.

2.4.1 *Output Of Points*

Both the filtering and interpolation routines require a subroutine named ADDPTS to be supplied by the user. This routine is called to output points, and its specification is as follows :-

```
SUBROUTINE ADDPTS(NPTS,XY,DRAW)

INTEGER4        NPTS            ! number of points
REAL            XY(2,NPTS)      ! line string
LOGICAL1        DRAW            ! draw line on screen ?
```

It is referenced by

```
CALL ADDPTS(NPTS,XY,DRAW)
```

and can allow the points either to be buffered up for subsequent output or passed to a graphics library for display on a screen.

Thus it is possible to start with a complete line and simply add the filtered points to an output buffer as in IFILTER. Alternatively, as in LITES2, the original line could be read in and dealt with section by section with the filtered or interpolated being points copied either to the output item using the normal flushing mechanism, or to the graphics output buffer.

2.4.2 *Interpolation Of A Line Segment*

The dummy subroutine POLATE is supplied by the user as an actual argument to INTRPL. As mentioned above, the library already contains three routines for this purpose, namely AKIMA, MCONAL and LINEAR. It will therefore not normally be necessary to supply any separate versions. The specification is :-

```
SUBROUTINE POLATE(XY)

REAL XY(2,6)            ! XY holds the master points
```

POLATE is passed the start of an array of six points. It interpolates between the third and fourth, putting in the last point but not the first. It is referenced by

```
CALL POLATE(XY)
```

and should call ADDPTS to output interpolated points.

CHAPTER 3

FILTERING

3.1 *FILTERING ALGORITHMS*

Two filtering algorithms are available, namely

i) a least squares "BUNCH" filter, and

ii) the Douglas-Peuker filter

The former was originally developed by Laser-Scan for the LASERAID software, while the latter is based on an implementation of a line reduction algorithm by D. H. Douglas of the University of Ottawa. The algorithm is attributed to himself and T. K. Peucker.

3.2 *BUNCH FILTER*

The BUNCH filter uses tolerances related to chords on the incoming point strings. The filter performs a least squares fit through all the existing data points, and when a point lies more than the ***lateral*** threshold distance from the trend line it is considered to be a provisional master point. A new fit is then conducted forwards from the last master point, until again the threshold deviation is exceeded. The last provisional point is taken to be the next master point and the intervening points are rejected. The process is repeated until the end of the line is reached. If the lateral threshold distance is large, it will rarely be exceeded and many points will be thrown away.

The number of points which are kept as master points or are thrown away may be additionally controlled by the minimum and maximum separation.

* the minimum separation is the shortest distance allowed between successive master points ***along*** the line. If this is set to a large value, more points will be thrown away giving increasingly angular linework.
* the maximum separation is the distance travelled ***along*** the line before forcing out a master point. A large value will result in very sparse points along straight and nearly straight lines. A maximum separation of 0.0 is equivalent to one of infinity, and means that no points will be forced out on distance criteria.

The maximum separation must be greater than or equal to the minimum separation which must be greater than or equal to the lateral threshold distance.

As the BUNCH filter uses a least squares algorithm, all new points will be placed to the **outside** of the original curve of the line being filtered.

3.2.1 *Initialisation*

call BUNCH_INIT(toler, res, sf)

in	- real	**toler**	a 3 element real array containing tolerances, in the coordinate units - **toler(1)** is the minimum point separation - **toler(2)** is the maximum point separation - **toler(3)** is the lateral tolerance
in	- real	**res**	the resolution of the coordinate units
in	- real	**sf**	a scale factor (which should be 1.0 if dealing with IFF units)

Prior to filtering any lines, BUNCH_INIT should be called to perform the necessary initialisation. It should also be called if any of the tolerances change.

3.2.2 *Filtering A Whole Line*

call BUNCH_LINE(numpts, line)

in	- long	**numpts**	the number of points in the line
in	- real	**line**	the line to be filtered, in the form of a 2***numpts** array

This routine applies the bunch filter to the whole line, calling BUNCH_POINT to deal with each point in turn. This in turn calls ADDPTS to output the filtered line (see next section and the chapter dealing with interpolation).

Note that if the line is a closed loop, the first and last points will probably not coincide after filtering. In such cases, the first point should then be dragged onto the last by the user's calling routine.

3.2.3 *Filtering A Line Point By Point*

call BUNCH_POINT(x, y, forced)

in	- real	**x**	X coordinate
in	- real	**y**	Y coordinate

in - long **forced** a flag with one of the following values, indicating the action to take :-
0 - examine the point to see if a new master point has to be created. If not, the point is used to calculate a new straight line approximation to the line segment being followed.
1 - tidy up, probably producing a new master point, taking account of the given point.
2 - tidy up, probably producing a new master point, but ignoring the given point.

Some users might prefer the flexibility of dealing with points individually, for instance when it is not known that the end of the line has been reached until after the last point has been dealt with. This would be the case if the input line was being dealt with section by section.

BUNCH_POINT calls the user-supplied subroutine ADDPTS to output filtered points as and when necessary. The specification of ADDPTS is given in the chapter on interpolation. Note that display on the screen is not relevant to filtering, so the last argument to ADDPTS is always false.

3.3 **DOUGLAS-PEUCKER FILTER**

The Douglas-Peucker filter works by joining the first and last points of the line being filtered with a straight line. The longest perpendicular to this straight line which cuts the line being filtered is then found. The two parts of the original line are then treated separately and the same process applied to each of them. This is repeated for successively smaller sections of the original line until any further perpendiculars which might be derived would be shorter than a specified lateral tolerance. The new line consists of the succession of points where the perpendiculars cut the original line.

The lateral tolerance is the shortest distance by which the new line will deviate laterally from the original line. It must be greater than 0.005, otherwise that constant will be used instead.

3.3.1 *Initialisation*

call DP_INIT(toler, sf)

in - real **toler** the lateral tolerance, in the coordinate units
in - real **sf** a scale factor (which should be 1.0 if dealing with IFF units)

Prior to filtering any lines, DP_INIT should be called to perform the necessary initialisation. It should also be called if the tolerance changes.

3.3.2 *Filtering A Whole Line*

call DP_LINE(numpts, line)

in	- long	**numpts**	the number of points in the line
in	- real	**line**	the line to be filtered, in the form of a 2***numpts** array

This routine transfers points into the format required for filtering, and on completion, calls the user-supplied subroutine ADDPTS to output the filtered points. Note that the number of points which can be processed is limited, and if a line has more than the maximum, the extra points will be ignored. Note also that in this case ADDPTS is passed points from the original line which was passed to DP_LINE.

The specification of ADDPTS is given in the chapter on interpolation. Note that display on the screen is not relevant to filtering, so the last argument to ADDPTS is always false.

3.3.3 *Line String Size*

The parameter file LSL$CMNCOORD:MAXPTS.PAR contains a parameter used in the declaration of point arrays in the Douglas-Peucker filtering routines, as follows :-

long **MAXPTS** - the maximum number of points in a line string, currently 10000

This parameter file is available to users for their own programs.

CHAPTER 4

QUADTREE PROCESSING

4.1 *INTRODUCTION*

The routine QUADSCAN uses a quadtree based technique (outlined below) to select from a set of N (two dimensional) points each pair of points which are within a given small distance (relative to the maximum overall coordinate range), in a time which is proportional to N, rather than to N squared (as would be the case if every point was checked for proximity with every other). Each pair of points selected is passed on to a user provided routine for processing.

The speed advantage over simple order N squared processing is approx:

N	:	50	100	1000	10000	100000
advantage	:	0.5	1.0	50	500	5000

QUADSCAN works by dividing the original point set bounding rectangular box into four quarters, each overlapping by the given point separation tolerance. Each quarter box is then divided in exactly the same way. This process of recursive subdivision continues untill a box contains less than a given number of points. The box is then processed (i.e. every point is checked for proximity with every other and pairs within the sepcified tolerance are passed through to the user routine for processing) and the scan moves on to the next box.

```
      status = QUADSCAN(      user_routine,
     &                        max_pts,pt,n_pts,xmin,xmax,ymin,ymax,
     &                        linked_list,max_next,next_pt,touch_tol,
     &                        n_lines,line_ends,line_fc,point_strings,
     &                        check_fcs,check,max_fcs,self_check,
     &                        max_work,workspace,show_progress)
```

Notes

12	out - long	function return		
1	in - long	user_routine	user routine.	
	in - long	max_pts	'pt' dimension.	
13	i/o - real	pt	all points in image to be processed.	
11	i/o - long	n_pts	number of points in 'pt'.	
	in - real	xmin,xmax	'pt' x-coordinate range.	
	in - real	ymin,ymax	'pt' y-coordinate range.	
2,10	in - logical	linked_list	is 'pt' linked list or contiguous?	
	in - long	max_next	'nxt_pt' dimension.	
2	i/o - long	next_pt	'pt' linked list pointers.	
7	in - real	touch_tol	points/vector touch tolerance.	
	in - long	n_lines	number of lines in 'line'.	
11	i/o - long	line_ends	line start/end point 'pt' indices.	
3	in - short	line_fc	feature code for each line.	
	in - logical	point_strings	'line' holds point_strings or vectors?	
5	in - logical	check_fcs	use feature code check?	
4	in - short logical	check	which fc pairs to process?	
	in - long	max_fcs	'check' dimension.	
6	in - logical	self_check	check points/vectors from same feature?	
9	ws - long	max_work	'workspace' dimension.	
8	in - long	workspace	internal workspace for QUADSCAN	
14	in - logical	show_progress	show running % completion?	

Notes:

1. The user routine must have the form:

```
INTEGER FUNCTION userprocess(ilin,jlin,ipt,jpt)

in - long      ilin            line index for point IPT
in - long      jlin            line index for point JPT
in - long      ipt             'pt' index for some point
in - long      jpt             'pt' index for some other point
```

   It must return SS$_NORMAL for success and some other error condition code on failure.

   This routine may also communicate with routines outside QUADSCAN via common blocks.

2. It is often required to insert points into a point string during processing. This calls for the use of a linked list - i.e. the sequence of points along the line is no longer pt(i)-pt(i+1), but pt(i)-pt(nxt_pt(i)). If this "one way linked list" structure is to be used then max_next must = max_pts. If it is not used, then max_next may = 1.

3. Dimension line_fc(2,n_lines); By giving each line a "feature code", it is possible to use the 'check' array as a mechanism for selecting only a restricted set of line-pairs for co-processing. If 'check_fcs' is false, then these feature codes are not used, and max_fcs may = 1.

4. Dimension check(max_fcs,max_fcs); Determines which fc pairs to process when 'check_fcs' is true. If a point/vector pair is within tolerance then they will only be passed to the user routine for processing if 'check_fcs' is false or if 'check_fcs' is true and they are from lines with feature codes line_fc(i) and line_fc(j) where both check(i,j) and check(j,i) are true.

5. If no feature code pair check is required, then no feature codes will be used, so max_fcs may = 1, and 'line_fc' and 'check' need not be set.

6. This mechanism for deciding whether points/vectors within the same feature should ever be passed to the user routine is entirely independant of feature code checking.

7. 'Touch_tol' is the distance within which points/vectors will be passed to the user routine for processing. The larger this tolerance, the longer QUADSCAN will take. As a rough guide, this tolerance should not be greater than about 1/50th of the greatest coordinate interval, unless the number of points in the image is very low - in which case you should not be using QUADSCAN in any case.

8. Dimension workspace(12,max_work); During processing, QUADSCAN must keep a record of the start and end 'pt' indices, and 'line' index, of every section of 'line' contained by the quarter box at every level of subdivision at any time during processing. 'max_work' is the max_imum total number of such line sections. The exact value for 'max_work' required depends on the density and distribution of the point data. As a general rule, 'max_work' should be = n_lines, but it is safer if 'max_work' is larger than this.

9. Vector closed loops for contiguous arrays must have start 'pt' = end 'pt' (i.e. line_ends(1,iline) = line_ends(2,iline)), but vector closed loops for linked lists must have last point index = index of 'pt' before start/end 'pt', and the linked list must be circular (i.e. line_ends(1,iline) = ipt; line_ends(2,iline) = jpt; nxt_pt(jpt) = ipt).

10. If the linked-list data structure is used then the user routine may create new points to be inserted between existing points, so the number of points 'n_pts' in 'pt' may be increased. Also, while n_pts may at no stage be decreased, points may be 'deleted' by the user routine by bridging them out of the linked list, so that the line's end point 'pt' indices in 'line_ends' will also be changed if any line end points are deleted.

11. Dimension line_ends(2,n_lines); At any stage during processing, QUADSCAN will assume that lines for which line_ends(1,iline) = 0 have been deleted (by the user routine) and will subsequently be ignored.

12. If successful, the QUADSCAN function result returns SS$_NORMAL, otherwise it returns the condition code returned by the last call to the user routine if a problem was encountered there, or else a COORDLIB condition code to the effect that some other problem was encountered within the QUADSCAN routine itself.

13. Dimension(2,max_pts); 2D coords of all points in all lines in image to be processed.

14. Percent progress will not be displayed unless process is interactive.

```
        PROGRAM EXAMPLE
C
        IMPLICIT NONE
C
C Parameters
        INTEGER*4       MAX_PTS
        INTEGER*4       MAX_NEXT
        INTEGER*4       MAX_LINES
        INTEGER*4       MAX_FCS
        INTEGER*4       MAX_WORK
        PARAMETER(      MAX_PTS = 9, MAX_NEXT = 1)
        PARAMETER(      MAX_FCS = 2, MAX_LINES = 10)
        PARAMETER(      MAX_WORK = MAX_LINES)
C
C Workspace
        INTEGER*4       N_PTS
        INTEGER*4       N_LINES
        INTEGER*4       NEXT_PT(MAX_NEXT)
        INTEGER*4       LINE_ENDS(2,MAX_LINES)
        INTEGER*4       WORKSPACE(12,MAX_WORK)
C
        INTEGER*2       LINE_FC(MAX_LINES)
C
        LOGICAL*4       POINT_STRINGS
        LOGICAL*4       LINKED_LIST
        LOGICAL*4       CHECK_FCS
        LOGICAL*1       CHECK(MAX_FCS,MAX_FCS)
        LOGICAL*4       SELF_CHECK
        LOGICAL*4       SHOW_PROGRESS
C
        REAL*4          PT(2,MAX_PTS)
        REAL*4          TOUCH_TOL
        REAL*4          XMIN,XMAX,YMIN,YMAX
C
C Functions
        LOGICAL*4       QUADSCAN
C
C Common blocks
        INCLUDE 'LSL$CMNLSL:STATUS.CMN'
        COMMON/EXAMPLE/ PT
C
C External routines
        LOGICAL*4       CHECK_VECTORS
        EXTERNAL        CHECK_VECTORS
C
C Data - coordinates for 9 points, and coordinate range limits
        DATA PT/-10, 10,   10, 10,   10,-10,  -10,-10,  -10, 10,
     &          -20,  0,   20,  0,
     &            0,-20,    0, 20/
        DATA XMIN,XMAX,YMIN,YMAX/-20, 20,-20, 20/
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C Initialise LSLLIB
C
```

```
        CALL LSL_INIT(.TRUE.)
C
C Do points in PT represent strings of isolated points or of vectors?
C
        POINT_STRINGS = .FALSE.         ! points represent vector ends
C
C Are points in PT sequenced by PT index, or by linked-list?
C
        LINKED_LIST = .FALSE.           ! contig.array, not linked_list
C
C Set up line-end PT indices for each line
C
        N_LINES = 3                     ! there are just 3 lines
        N_PTS = 9                       ! there are 9 points altogether
        LINE_ENDS(1,1) = 1              ! line 1 starts at point 1
        LINE_ENDS(2,1) = 5              ! line 1 ends at point 5
        LINE_ENDS(1,2) = 6              ! line 2 starts at point 6
        LINE_ENDS(2,2) = 7              ! line 2 ends at point 7
        LINE_ENDS(1,3) = 8              ! line 3 starts at point 8
        LINE_ENDS(2,3) = 9              ! line 3 ends at point 9
C
C Use feature-code check?
C
        CHECK_FCS = .TRUE.
C
C Set up a "feature code" for each line
C
        LINE_FC(1) = 1
        LINE_FC(2) = 2
        LINE_FC(3) = 2
C
C Set up CHECK array so that only lines with different FCs are processed
C
        CHECK(1,1) = .FALSE.            ! don't check type 1s with same
        CHECK(1,2) = .TRUE.             ! do check type 1s with type 2s
        CHECK(2,1) = .TRUE.             ! do check type 2s with type 1s
        CHECK(2,2) = .FALSE.            ! don't check type 2s with same
C
C Decide on a suitable "touch tolerance"
C
        TOUCH_TOL = 0                   ! check is for crossing vectors
C
C Don't check any vector against another vector in the same line (they
C won't be anyway here, because the FC check would preclude this).
C
        SELF_CHECK = .FALSE.
C
C Don't bother to show % completion
C
        SHOW_PROGRESS = .FALSE.
C
C Call QUADSCAN, passing it the routine CHECK_VECTORS, to find every
C intersection between vectors from features with different FCs
C
        LSL_STATUS = QUADSCAN(  CHECK_VECTORS,
     &                          MAX_PTS,PT,N_PTS,XMIN,XMAX,YMIN,YMAX,
```

```
     &                          LINKED_LIST,MAX_NEXT,NEXT_PT,TOUCH_TOL,
     &                          N_LINES,LINE_ENDS,LINE_FC,POINT_STRINGS,
     &                          CHECK_FCS,CHECK,MAX_FCS,SELF_CHECK,
     &                          MAX_WORK,WORKSPACE,SHOW_PROGRESS)
C
        CALL LSL_EXIT
        END
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
        LOGICAL FUNCTION CHECK_VECTORS(ILIN,JLIN,IPT,JPT)
C
        IMPLICIT NONE
C
C Common block data
        INTEGER*4       MAX_PTS
        PARAMETER(      MAX_PTS = 9)
        REAL*4          PT(2,MAX_PTS)
        COMMON/EXAMPLE/ PT
C
C Parameters
        INTEGER*4       SS$_NORMAL
        EXTERNAL        SS$_NORMAL
        REAL*4          SMALLREAL       ! smallest accurate real
        PARAMETER(      SMALLREAL = 1E-5)
C
C Data in
        INTEGER*4       ILIN,JLIN       ! LINE indices
        INTEGER*4       IPT,JPT         ! PT indices
C
C Workspace
        REAL*4          P1X,P1Y,P2X,P2Y ! point coords at ends of vec P
        REAL*4          Q1X,Q1Y,Q2X,Q2Y ! point coords at ends of vec Q
        REAL*4          DXP,DYP         ! vector P12
        REAL*4          DXQ,DYQ         ! vector Q12
        REAL*4          PF,QF           ! dist ratio of int along P & Q
        REAL*4          X,Y             ! intersection point coords
        REAL*4          D1P             ! dist of P1 from Q12
        REAL*4          D1Q             ! dist of Q1 from P12
        REAL*4          DET             ! a matrix determinant
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
C       If vectors intersect, intersection point is found and printed.
C
C       Function result always returns SS$_NORMAL here.
C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
        CHECK_VECTORS = %LOC(SS$_NORMAL)
C
        P1X = PT(1,IPT)                 ! point at start of vec P
        P1Y = PT(2,IPT)
        P2X = PT(1,IPT+1)
        P2Y = PT(2,IPT+1)
        Q1X = PT(1,JPT)
```

```
        Q1Y = PT(2,JPT)
        Q2X = PT(1,JPT+1)
        Q2Y = PT(2,JPT+1)
C
C Check for edge-box ontersection as most wont intersect at all
C
        IF (MAX(P1X,P2X).LT.MIN(Q1X,Q2X)-SMALLREAL) GOTO 10
        IF (MIN(P1X,P2X).GT.MAX(Q1X,Q2X)+SMALLREAL) GOTO 10
        IF (MAX(P1Y,P2Y).LT.MIN(Q1Y,Q2Y)-SMALLREAL) GOTO 10
        IF (MIN(P1Y,P2Y).GT.MAX(Q1Y,Q2Y)+SMALLREAL) GOTO 10
C
        DXQ = Q2X-Q1X
        DYQ = Q2Y-Q1Y
        DXP = P2X-P1X
        DYP = P2Y-P1Y
C
        DET = DXQ*DYP-DYQ*DXP
        IF (ABS(DET).LT.SMALLREAL) GOTO 10      ! vectors parallel?
C
        D1P = (P1Y-Q1Y)*DXQ-(P1X-Q1X)*DYQ
        D1Q = (Q1Y-P1Y)*DXP-(Q1X-P1X)*DYP
C
        PF = -D1P/DET
        QF =  D1Q/DET
C
        IF (PF.LT.-SMALLREAL.OR.PF.GT.1+SMALLREAL) GOTO 10
        IF (QF.LT.-SMALLREAL.OR.QF.GT.1+SMALLREAL) GOTO 10
C
        X = P1X+PF*DXP
        Y = P1Y+PF*DYP
C
        WRITE(*,'('' Intersection between lines '',I2,'' and '',I2,
     &  '' at ('',F6.2,'','',F6.2,'')'')') ILIN,JLIN,X,Y
C
10      RETURN
        END
```

This produces the following output:

```
Intersection between lines  1 and  3 at (  0.00, 10.00)
Intersection between lines  1 and  2 at ( 10.00,  0.00)
Intersection between lines  1 and  3 at (  0.00,-10.00)
Intersection between lines  1 and  2 at (-10.00,  0.00)
 ELAPSED:    0 00:00:00.50  CPU: 0:00:00.10  BUFIO: 0  DIRIO: 0  FAULTS: 39
```

This is a trivial example. QUADSCAN should not be used unless the number of points in the dataset is at least 100.

CHAPTER 5

BEZIER INTERPOLATION

5.1 *INTRODUCTION*

Bezier is a polynomial interpolation algorithm - it has been used widely in interactive graphics to obtain approximate solutions to curve fitting problems. Instead of using the data points directly to specify a polynomial, a set of guide points is first generated from data points.

It resides in this seperate chapter because it utilitises a different mechanism from other interpolation routines in COORDLIB.

The routine BEZIER INTERPOLATION is the library routine that should be called by an applications program.

call BEZIER_INTERPOLATION(ncoords_in, xy_in, ncoords_out, xy_out, looped, do_average, bezier_tolerances, do_change_tol, error)

in	- long	**ncoords_in**	the number of points in the input line
in	- real	**xy_in**	the line to be interpolated, in the form of a 2*__n__ array
in	- long	**ncoords_out**	the number of points in the output (interpolated) line
in	- real	**xy_out**	the line after interpolation, in the form of a 2*__n__ array
in	- byte	**looped**	TRUE if the line is looped (ie. first point equals the last point)
in	- byte	**do_average**	TRUE if the AVERAGE sub-qualifier is to be used)
in	- real	**bezier_tolerances**	the lateral and corner tolerance values
in	- byte	**do_change_tol**	TRUE if the tolerance for the sub-division of long segments is to be incremented automatically
out	- long	**error**	set to 0 if the operation is success

BEZIER_INTERPOLATION calls the following library routines:

1. **SPCPTS** - removes consecutive coincident points, and inserts additional points in long line segments. If DO_CHANGE_TOL is set to TRUE and the intermediate buffer is full, the program will automatically increment the segment tolerance for sub-dividing long segments by a factor of one until

the total number of intermediate points is less than that allowed in the buffer(i.e. 15000 points).

2. **AVPTS** - generates guide points to be used to specify the bezier polynomial

3. **THIN** - removes surplus guide points. As many points as possible are removed so that the distance between points is never greater than the lateral tolerance specified in BEZIER_TOLERANCES(1)

4. **BEZIER** - interpolates using Bezier polynomial interpolation formula using generated guide points. The routine is passed BEZIER_TOLERANCES.

BEZIER_TOLERANCES contains the two tolerance values used by the interpolation algotithm.

1. **BEZIER_TOLERANCES(1)** is the lateral deviation tolerance. It is the distance from a straight line joining consecutive points to a true curve.

2. **BEZIER_TOLERANCES(2)** is the corner hug tolerance. It is the distance of the curve from the guide points.