FX Programmer's Guide

Rev. 0.90 (beta)

The **F**luid Feature E**X**traction Tool-kit

Bob Haimes

&

David Kenwright

Massachusetts Institute of Technology

July 10, 2000

License

Contents

1 Introduction

FX is the newest in a series of graphics and visualization tools to come out of the Department of Aeronautics and Astronautics at MIT. **FX** (which stands for **F**luid Feature **EX**traction) is designed to work with the results of Computational Fluid Dynamics in either steady-state of in a co-processing transient mode. The end result is the extraction of the feature so that it can be used directly with a visualization (such as **Visual3** or **pV3**) or applied to some "off-line" procedure such as mesh enrichment.

The **FX** tool-kit can be used directly with solvers and has been designed to function in parallel/distributed environments. This has required supporting a fairly complete set of grid discretizations as well as domain decomposition (partitioning).

The Application Programming Interface (API) is split into 2 basic sections:

- Support
  These are the utility and general routines that support the communication of the information that is used to determine the spatial, temporal and partitioning of the CFD data.

- Features
  These routines return the features as 3D structures and associated quantities, such as strength that may be displayed in visualization systems or used for other non-interactive ("off-line") applications.

2 Programming Overview

Before presenting the subroutine argument lists in detail it is helpful to discuss, in general terms, the data structures which the programmer supplies to **FX**. In some cases these data structures can be taken directly from either **Visual3** or **pV3**. See the appropriate Advanced Programmer's Guide.

The programmer gives **FX** a list of unconnected cells and structured blocks. The disjoint cells are of four types; tetrahedra, pyramids, prisms and hexahedra. This element generality covers almost all data structures being used in current computational algorithms. Any special cell type which is different must be split up into some combination of these primitives by the programmer. Linear interpolation is used throughout **FX**, so high order elements must be also be subdivided so that the linear interpolation assumption is valid.

The volume(s) are defined by face matching of the elements (based on equating node numbering). Any exposed face (not shared by 2 cells) must be treated as either a boundary (a *domain surface* in **Visual3/pV3** terminology) or covered with *halo* cells. Therefore for multi-structured block cases, the surfaces that are actually inside the volume must be treated so that **FX** can patch them together.

Note: Poly-tetrahedral strips are not supported.

2.1 Domain Decomposition

The **FX** tool-kit requires the calculation of spatial derivatives. This is performed in a *finite-element* manner. If the data is not completely resident within one computer, additional information is required so that the result is consistent. For all *internal* boundaries (created by the partitioning of the data) a halo of cells is required. This halo is constructed by including all the cells that touch a node on the boundary that exist in the neighboring partition. This produces additional cells and nodes in the partition. These are differentiated in the programming interface.

It should be noted for unstructured meshes that this will require more elements than those whose faces touch the boundary.

2.2 Node Numbering

The node numbering used within **FX** is local. For distributed memory cases information is required for the halo region(s). This is done by adding the nodes required to produce these cells at the end of the node space. It is the responsibility of the calling application to do any message passing and node number re-mapping so that the halo information is correct.

The node numbering used differentiates between the nodes in the non-block regions (formed by the disjoint cells), the structured blocks, and the halo regions. Figure 1 shows a schematic of the node space. **knode** is the number of nodes for the non-block grid. Each structured block (m) adds $NI_m * NJ_m * NK_m$ nodes to the node space (where NI, NJ and NK are the number of nodes in each direction). The node numbering within the block follows the memory storage, that is, (i,j,k) in FORTRAN and [k][j][i] in C. The **FX** node number $= base + i + (j-1) * NI_m + (k-1) * NI_m * NJ_m$. Where *base* is **knode** for the first block, and **knode** plus the number of nodes in the first block for the second, and etc.
Note: all indices start at 1.

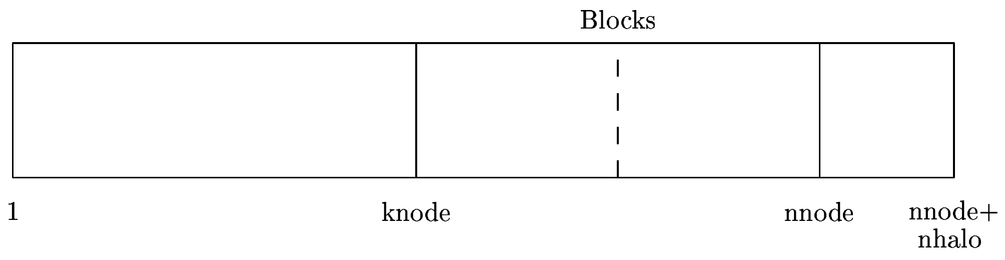


Figure 1: Node Space

nhalo is the number of nodes added to the domain for the halo elements. This is zero for a case with a single partition.

2.3 Cell Numbering

The non-block cell types may contain nodes from the non-block and the structured block volumes but not from the halo nodes. The cell numbering used within **FX** orders the cells by type. Figure 2 shows a schematic of the cell space. The programmer explicitly defines all non-block cells by the call FXcell or provides the pointers by the call-back FXcellPtr. Again the cells within the blocks are defined by the block size. Each structured block (m) adds $(NI_m - 1) * (NJ_m - 1) * (NK_m - 1)$ cells to cell space. The cell numbering within the block follows the memory storage so that a **FX** cell number $= base + i + (j-1) * (NI_m - 1) + (k-1) * (NI_m - 1) * (NJ_m - 1)$. Where $base$ is **nTets+nPyra+ nPrism+nHexa** for the first block, and this value plus the number of cells in the first block for the second, and etc.
Note: i goes from 1 to $NI_m - 1$, j goes from 1 to $NJ_m - 1$, and k goes from 1 to $NK_m - 1$.

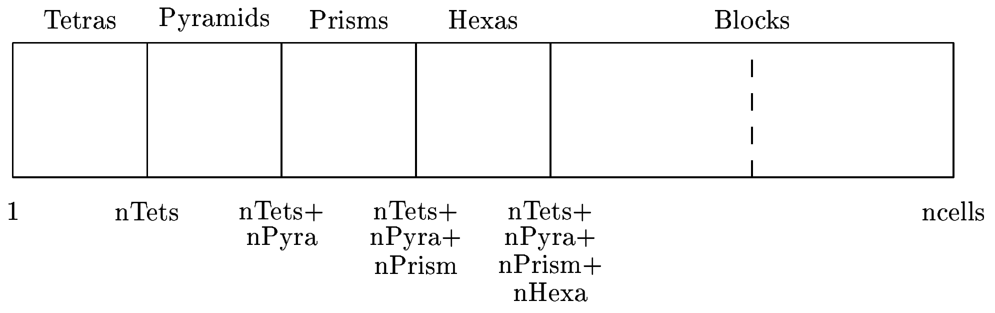


Figure 2: Cell Space

There are individual structures for each element type. This provides compatibility with both **Visual3** and **pV3** and minimizes the amount of memory required to fully describe complex gridding. The halo cells are handled in a different manner. Each cell is disjoint (either a tetrahedron, pyramid, prism or hexahedron) and is stored in the same structure. Node indices that make up the halo cells must contain at least one non-halo node and at least one halo node (index $>$ **nnode**). The exception to this is when blocks are patched or for C meshes where all node indices can be from the non-halos.

If the disjoint element storage is not consistant with that used by **Visual3** and **pV3**, and one wishes to minimize the total memory load, **FX** can construct these cells *on-the-fly*. This is a tradeoff of less speed for less memory utilization. If this option is used, the cell indexing need not reflect the collection of all like types (as seen in Figure 2), but can be random. And, the total number of cells is **nTets+nPyra+nPrism+nHexa**, but the number of individual cells of each type need to match these input values. Halo elements will also be constructed *on-the-fly* and will have negative indices.

Again, the numbering is local for multiple processor applications.

2.4 Blanking

Blanking is an option (see the description of FX_Init) and only used with structured blocks to indicate that some region of the block is 'turned off'. A part of a block is deactivated by flagging the appropriate nodes as invalid. This is done by an IBLANK array. An invalid node is never used.

When blanking is used, all the nodes (**nnode** − **knode**) in the structured block space are given a value; zero corresponds to an invalid mesh point, any non-zero value indicates an existing node point.

2.5 Surfaces

In principle, all exposed facets could be grouped together to form one bounding surface. However, in many applications it is more useful to split the bounding surface into a number of pieces, referred to in **Visual3** and **pV3** documentation as *domain surfaces.* For example, the outer bounding surface of a calculation of airflow past a half-aircraft (using symmetry to reduce the computation) would typically be split into four pieces, the inflow boundary, the outflow, the symmetry plane, the aircraft. The Residence Time functions of **FX** require information on which exposed facets to apply what boundary condition. These must be classified as either inflow, outflow, symmetry and no-flux (wall).

Internal surfaces are those that get created when the computational domain is sub-divided and placed in multiple machines. These artificial surfaces are handled by the halo elements so it appears to **FX** that they do not exist.

2.6 Programming Notation

FX was designed to be accessible from both FORTRAN and C. FORTRAN is more restrictive in argument passing and naming, therefore it has shaped the programming interface. The routine descriptions in this guide are from the C programmer's point of view. But because FORTRAN is supported with the same API all routine arguments are pass by reference. It is assumed that a routine's argument is not modified unless documented as such.

For IBM and HP ports, all **FX** entry points are the FORTRAN names in lower-case. On all other platforms except the CRAY and WindowsNT, external entries are lower-case with an underscore ('_') appended to the end. CRAY entry points are upper-case with no appended underscores. WindowsNT entry points must be declared as **__stdcall** and are upper-case with no appended underscores. See the file 'FX.h' in the distribution for a method to avoid these problems.

Consistent with the **Visual3** and **pV3** naming conventions, the routines that are part of the **FX** tool-kit are prefixed with 'FX_', those that are supplied by the programmer start with 'FX' and do not have an underscore as the next character. There are a number of pairs (or triads) of these programmer-supplied call-backs. These exist in order in conserve memory, that is if the programmer already has the data in the proper form then the pointer to that data is passed to **FX**. Otherwise **FX** allocates the appropriate memory and it is the responsibility of the call-back to fill that structure. Only one of the pair (or triad) will be called during the **FX** session. The convention for the routines that return pointer(s) is the base routine name with the 'PTR' or 'P3D' suffix.

2.7 Calling Sequences

The **FX** tool-kit supports steady-state as well as three types of unsteadiness. In a multiple partition simulation, each sub-domain can have a different transient mode. Each mode causes a different internal calling sequence. In general, the application must first call FX_Init to initialize the **FX** system and then call FX_Update after every time the solution space has been updated. A schematic of a typical CFD solvers coupling with FX can be seen in Figure 3. The name FX_extract is an indication of any series of *underscore* routines documented in Sections 5 to 8.

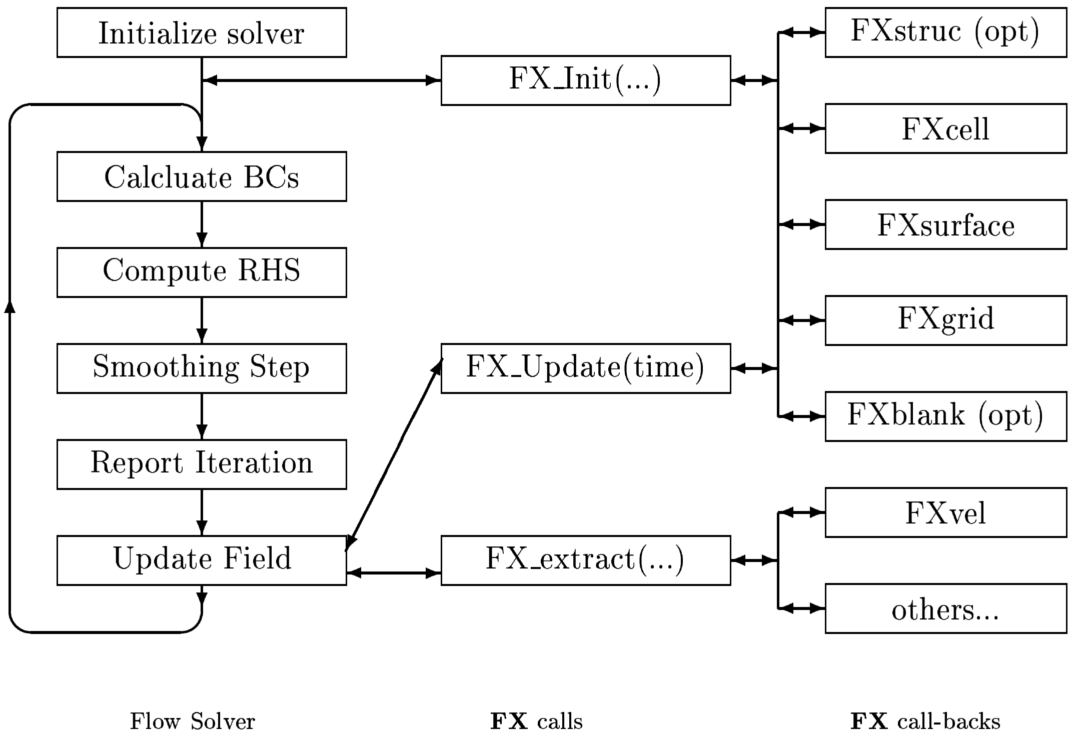


Figure 3: Co-processing Calling sequence

- Steady-State

Call	Calls in Sequence
FX_Init	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
	FXgrid, FXgridPtr or FXgridP3d
	FXBlank or FXblankPtr (optional)
FX_Update	NOT required
FX_extract	FXvel, FXvelPtr or FXvelP3d
	other call-backs as needed

- Data Unsteady

Call	Calls in Sequence
FX_Init	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
	FXgrid, FXgridPtr or FXgridP3d
	FXBlank or FXblankPtr (optional)
FX_Update	NONE
FX_extract	FXvel, FXvelPtr or FXvelP3d
	other call-backs as needed

- Grid Unsteady

Call	Calls in Sequence
FX_Init	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
FX_Update	FXgrid, FXgridPtr or FXgridP3d
	FXBlank or FXblankPtr (optional)
FX_extract	FXvel, FXvelPtr or FXvelP3d
	other call-backs as needed

- Structure Unsteady

Call	Calls in Sequence
FX_Init	NONE
FX_Update	FXstruc
	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
	FXgrid, FXgridPtr or FXgridP3d
	FXBlank or FXblankPtr (optional)
FX_extract	FXvel, FXvelPtr or FXvelP3d
	other call-backs as needed

3 Programmer-called subroutines

3.1 FX_Init

FX_INIT(GAMMA, IOPT, KNODE, NHALO, NTETS, NPYRA, NPRISM, NHEXA, NBLOCK, BLOCKS, NHCELL, NFACET, NBC, FLAGS)

This subroutine initializes the **FX** tool-kit. Calling this routine defines the type of case and the sizes of various parameters having to do with the volume discretization. This calling sequence also defines how and which call-backs are invoked so that **FX** can get the required data. This routine must be the first **FX** tool-kit call.

float *GAMMA	Ratio of specific heats
int *IOPT	Unsteady control parameter
	IOPT=0 steady grid and data
	IOPT=1 steady grid and unsteady data
	IOPT=2 unsteady grid and data
	IOPT=3 structure unsteady
int *KNODE	Number of non-block nodes
int *NHALO	Number of halo nodes
int *NTETS	Number of tetrahedra
int *NPYRA	Number of pyramids
int *NPRISM	Number of prisms
int *NHEXA	Number of hexahedra
int *NBLOCK	Number of structured blocks
int BLOCKS[][3]	Structured block definitions:
	BLOCKS[m][0] = NI
	BLOCKS[m][1] = NJ
	BLOCKS[m][2] = NK
int *NHCELL	Number of halo elements
int *NFACET	Number of domain surface facets
int *NBC	Number of domain surface groups (boundary conditions)

int *FLAGS	An error code on return (0 is success). This is the call mask on input:

bit 0 – **1/0** = 0 - call FXcell for the disjoint cell data,
1 - call FXcellPtr for the disjoint information.

bit 1 – **2/0** = 0 - call FXsurface for the Boundary Condition data, 2 - call FXsurfacePtr for the Boundary Conditions.

bit 2 – **4/0** = 0 - call FXgrid for coordinates,
4 - call FXgridPtr or FXgridP3d for using pointers.

bit 3 – **8/0** = 0 - call FXvel for the flow vector field,
8 - call FXvelPtr or FXvelP3d for using pointers.

bit 4 – **16/0** = 0 - no Blanking, 16 - Blanking.

bit 5 – **32/0** = 0 - FXblank is called for Blanking,
32 - FXBlankPtr is called.

bit 6 – **64/0** = 0 - use FXgridPtr and FXvelPtr,
64 - use Plot3D style calls FXgridP3d and FXvelP3d.

bit 7 – **128/0** = 0 - use FXcell or FXcellPtr,
128 - build elements *on-the-fly* with FXcellGet.

Notes:
1) For structure unsteady cases ($IOPT = 3$), the parameters that describe the sizes of the node and cell space should be a good guess at the sizes used during the simulation. For structured block cases, NBLOCK must be the maximum number of blocks for the run. The current sizes are set by a call to FXstruc from within FX_Update.
2) If FXcellGet is used (bit 7 on) then the total number of elements used is the sum of NTETS, NPYRA, NPRISM and NHEXA. FXcellGet is also used to define the halo elements. The index order need not match the cell space (Figure 2) order, but can be random in mixed element cases.

3.2 FX_Update

FX_UPDATE(TIME)
This subroutine must be called after the solver has updated the solution space. This is when all communication between any partitions is complete including the messages required to transmit the halo data. The call to this routine is not needed if $IOPT = 0$.

float *TIME	The current simulation time.

3.3 FX_Close

FX_CLOSE( )
This subroutine closes the **FX** tool-kit. This deallocates all memory. No **FX** routines can be used until FX_Init is called again.

No Arguments

3.4 FX_Free

FX_FREE(PTR)
This function is equivalent to the C routine 'free'. It deallocates a block of memory. NOTE: Use this utility routine to free up blocks of that have been allocated by **FX** and returned when they are no longer needed. These pointers are labeled as freeable in the routine definition.

void **PTR	The address of the memory block.

4 Call-backs

4.1 FXcell

FXCELL(TETS, PYRA, PRISM, HEXA, HCELL)
This subroutine supplies **FX** with the grid data structure. It is not required for a grid that contains only structured blocks and no halo cells.

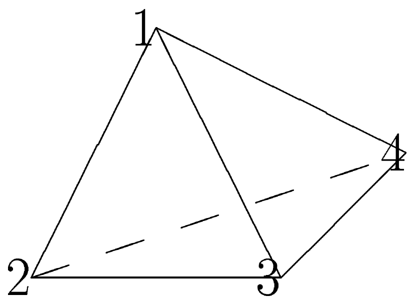
int TETS[NTETS][4]	Node indices for tetrahedral cells (filled)
int PYRA[NPYRA][5]	Node indices for pyramid cells (filled)
int PRISM[NPRISM][6]	Node indices for prism cells (filled)
int HEXA[NHEXA][8]	Node indices for hexahedral cells (filled)
int HCELL[NHCELL][9]	Halo cell descriptions (filled)
	HCELL[m][0-7] = Node indices for the cell
	HCELL[m][8] = 1 - tetrahedron, 2 - pyramid, 3 - prism, 4 - hexahedron

The correct order for numbering nodes for the four disjoint cell types is shown in Fig. 4.

4.2 FXcellPtr

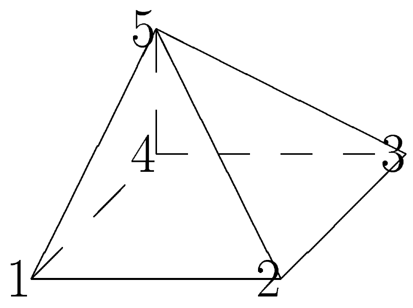
FXCELLPTR(PTETS, PPYRA, PPRISM, PHEXA, HCELL)
This subroutine supplies **FX** with the pointers to grid data structure. It is not required for a grid that contains only structured blocks and no halo cells.

int **PTETS	Pointer to node indices for tetrahedral cells (returned)
int **PPYRA	Pointer to node indices for pyramid cells (returned)
int **PPRISM	Pointer to node indices for prism cells (returned)
int **PHEXA	Pointer to node indices for hexahedral cells (returned)
int HCELL[NHCELL][9]	Halo cell descriptions (filled)
	HCELL[m][0-7] = Node indices for the cell
	HCELL[m][8] = 1 - tetrahedron, 2 - pyramid, 3 - prism, 4 - hexahedron



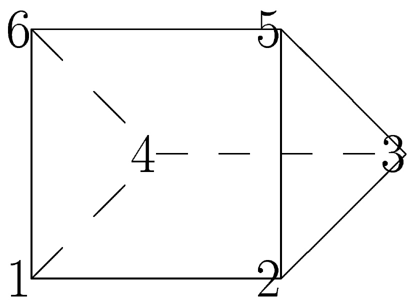
face	nodes
1	1 2 3
2	2 3 4
3	3 4 1
4	4 1 2

Tetrahedron



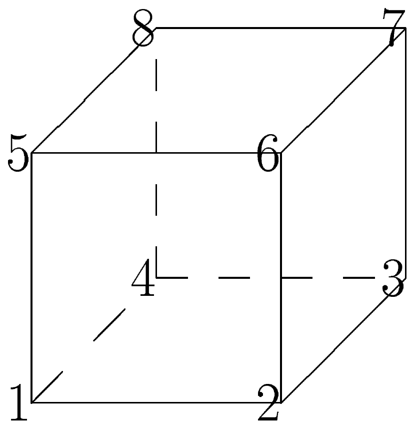
face	nodes
1	1 2 3 4
2	2 3 5
3	3 4 5
4	4 5 1
5	5 1 2

Pyramid



face	nodes
1	1 2 3 4
2	2 5 6 1
3	3 4 6 5
4	4 6 1
5	5 2 3

Prism



face	nodes
1	1 2 3 4
2	2 3 7 6
3	3 4 8 7
4	4 8 5 1
5	5 6 7 8
6	6 5 1 2

Hexahedron

Figure 4: Disjoint cell types and node/face numbering

4.3 FXcellGet

FXCELLGET(INDEX, TYPE, CELL)

This subroutine supplies **FX** with the disjoint cell data for *on-the-fly* construction of elements. It is required if bit 7 is on in the FLAGS arguments of FX_Init.

int *INDEX	Cell index, 1 to total number of cells and -1 to -NHCELL to indicate halo elements.
int *TYPE	1 - tetrahedron, 2 - pyramid, 3 - prism, 4 - hexahedron (returned)
int CELL[8]	Node indices for the requested cell (filled)

4.4 FXgrid

FXGRID(XYZ, HXYZ)

This subroutine supplies **FX** with the grid coordinates for all of the nodes.

float XYZ[NNODE][3]	(x, y, z)-coordinates of grid nodes (filled)
float HXYZ[NHALO][3]	(x, y, z)-coordinates of halo grid nodes (filled)

4.5 FXgridPtr

FXGRIDPTR(PXYZ, HXYZ)

This subroutine supplies **FX** with the pointer to the grid coordinates for all of the nodes.

float **PXYZ	the pointer to the structure containing (x, y, z)-coordinates of grid nodes (returned)
float HXYZ[NHALO][3]	(x, y, z)-coordinates of halo grid nodes (filled)

4.6 FXgridP3d

FXGRIDP3D(PX, PY, PZ, HXYZ)

This subroutine supplies **FX** with the pointers to the grid coordinates for all of the nodes using the PLOT3D strorage scheme.

float **PX	the pointer to the memory block containing the x-coordinates of grid nodes (returned)
float **PY	the pointer to the memory block containing the y-coordinates of grid nodes (returned)
float **PZ	the pointer to the memory block containing the z-coordinates of grid nodes (returned)
float HXYZ[NHALO][3]	(x, y, z)-coordinates of halo grid nodes (filled)

4.7 FXsurface

FXSURFACE(NSURF, CELL, FACET)

This subroutine supplies **FX** with the surface data structure. This specifies that these are exposed facets and indicates the type of boundary condition to apply.

int NSURF[NBC][2]	NSURF[m][0] is the pointer to the end of domain boundary group n, i.e. it contains the index to the last entry in CELL and FACET for that group. NSURF[m][1] is the boundary type: **1** inflow **2** outflow **3** wall **4** wall (slip) **5** symmetry **6** nothing – extrapolate
int CELL[NFACET]	the index for the cell containing the face defined by FACET. (filled)
int FACET[NFACET][4]	node numbers for surface faces. For quadrilateral faces FACET must be ordered clockwise or counter-clockwise; for triangular faces, FACET[m][3] must be set to zero. (filled)

Note:

The correct order for numbering faces for the four disjoint cell types is shown in Fig. 4. For structured blocks; face #1 is for exposed cells with cell index $k = 1$, face #2 is for $i = NI_m - 1$, face #3 is for cells with $j = NJ_m - 1$, face #4 is for $i = 1$, face #5 is associated with $k = NK_m - 1$, and face #6 is for $j = 1$.

4.8 FXsurfacePtr

FXSURFACEPTR(NSURF, PCELL, PFACET)

This subroutine supplies **FX** with the surface data pointer. This specifies that these are exposed facets and indicates the type of boundary condition to apply.

int NSURF[NBC][2]	NSURF[m][0] is the pointer to the end of domain boundary group n, i.e. it contains the index to the last entry in CELL and FACET for that group. NSURF[m][1] is the boundary type.
int **PCELL	pointer to the structure containing cell indices for the facets (returned)
int **PFACET	pointer to the structure containing node numbers for surface faces (returned)

Note:

The pointer returned from within **Visual3** using V3_GetStruc (OPT = 323) can not be used for PCELL because that data has the face index encoded with the cell number.

4.9 FXblank

FXBLANK(IBLANK)

This subroutine supplies **FX** with blanking data. Required for bit 4 on and bit 5 off in FLAGS (of FX_Init).

int IBLANK[NNODE-KNODE]	Blanking data (filled):
	= 0 off, invalid node
	≠ 0 on

4.10 FXblankPtr

FXBLANKPTR(PIBLANK)

This subroutine supplies **FX** with a pointer to the blanking data. Required for bit 4 on and bit 5 on in FLAGS (of FX_Init).

int **PIBLANK	pointer to blanking data (returned)

4.11 FXvel

FXVEL(V, HV)

This subroutine supplies **FX** with the velocity field.

float V[NNODE][3]	Velocity function values (Vx, Vy, Vz) (filled)
float HV[NHALO][3]	Halo velocity function values (Vx, Vy, Vz) (filled)

4.12 FXvelPtr

FXVELPTR(PV, HV)

This subroutine supplies **FX** with the pointer to the velocity field.

float **PV	Pointer to the Velocity structure (returned)
float HV[NHALO][3]	Halo velocity function values (Vx, Vy, Vz) (filled)

4.13 FXvelP3d

FXVELP3D(PVX, PVY, PVZ, HV)

This subroutine supplies **FX** with the pointers to the velocity field using the PLOT3D stroage scheme.

float **PVX	Pointer to the memory block that contains Vx (returned)
float **PVY	Pointer to the memory block that contains Vy (returned)
float **PVZ	Pointer to the memory block that contains Vz (returned)
float HV[NHALO][3]	Halo velocity function values (Vx, Vy, Vz) (filled)

4.14 FXscal

FXSCAL(TYPE, S, HS)
This subroutine supplies **FX** with the specified scalar field.

int TYPE	Scalar field indicator
float S[NNODE]	Scalar functional values based on TYPE (filled):
	TYPE = 1 - density
	TYPE = 2 - pressure
	TYPE = 3 - Mach number
	TYPE = 4 - Total viscosity (laminar and turbulent)
	TYPE = 5 - Enthalpy
float HS[NHALO]	Halo scalar functional values based on TYPE

4.15 FXstruc

FXSTRUC(KNODE, NHALO, NTETS, NPYRA, NPRISM, NHEXA, NBLOCK, BLOCKS, NHCELL, NFACET, NBC)
This subroutine is required for structure unsteady cases ($IOPT = 3$) only. This routine supplies the sizes of the current state of the problem.

int *KNODE	Number of non-block nodes / static flag
int *NHALO	Number of halo nodes
int *NTETS	Number of tetrahedra
int *NPYRA	Number of pyramids
int *NPRISM	Number of prisms
int *NHEXA	Number of hexahedra
int *NBLOCK	Number of structured blocks
int BLOCKS[][3]	Structured block definitions
int *NHCELL	Number of halo elements
int *NFACET	Number of domain surface facets
int *NBC	Number of domain surface groups (boundary conditions)

Notes:
1) If KNODE is -1 that is a special flag to indicate that the structure has NOT changed for this iteration. With this flag set, no other parameters should be modified, and **FX** reverts to the grid unsteady calling sequence.
2) If NHALO is non-zero at initialization, it must remain non-zero.

5 Shock Routines

5.1 FX_ShockFind

FX_SHOCKFIND(PTEST)
This subroutine returns the result of the shock test function.

float **PTEST	Pointer to a block of floats (freeable), in the form TEST[NNODE] (returned) If a NULL (zero) is returned, then some error occured. These values produce a scalar field for the shock test function. Any value greater than 1.0 is an indication that the node is in a shock region.

5.2 FX_ShockSurface

FX_SHOCKSURFACE(TEST, NSPTS, PSXYZ, NSTRIS, PSTRIS, PSCELL)
This subroutine takes the shock test function, generates and returns the surface(s) at the value 1.0. The surface(s) can be constructed from the triangle indices (bias 1) into the shock nodes pointed to by PSXYZ.

float TEST[NNODE]	This must be the data returned by FX_ShockFind.
int *NSPTS	The number of points that support the shock surface (returned)
float **PSXYZ	Pointer to the block of memory (freeable) that contains the coordinates (returned) The memory block is of the form SXYZ[NSPTS][3].
int *NSTRIS	The number of triangles that make up the surface (returned)
int **PSTRIS	Pointer to the block of memory (freeable) that contains the triangle indices (returned) The memory block is of the form STRIS[NSTRIS][3].
int **PSCELL	Pointer to the block of memory (freeable) that contains the cell indices for the triangle (returned) The memory block is of the form SCELL[NSTRIS].

5.3 FX_ShockVolumes

FX_SHOCKVOLUMES(TEST, NREGION, PVOLS, PECELL, PCELLS, PENODE, PNODES)

This subroutine takes the shock test function and partitions space (based on cells) into regions that cross 1.0 or are above 1.0 for the function. The physical volume as well as the list of cells and nodes contained with each region are returned. Note: a cell can only be in one region so there are some circumstances where the results of FX_ShockSurface will display more regions than returned here.

float TEST[NNODE]	This must be the data returned by FX_ShockFind.
int *NREGION	The number of distinct regions or volumes (returned) Zero is the indication of an error.
float **PVOL	Pointer to the block of memory (freeable) that contains the actual volume for the region (returned) – can be NULL (zero) under an error The memory block is of the form VOL[NREGION].
int **PECELL	Pointer to the block of memory (freeable) that contains the last index (bias 1) for the list of cells for the region (returned) The memory block is of the form ECELL[NREGION].
int **PCELLS	Pointer to the block of memory (freeable) that contains the complete list of cells for all regions (returned) The block looks like CELLS[ECELL[NREGION-1]].
int **PENODE	Pointer to the block of memory (freeable) that contains the last index (bias 1) for the list of nodes for the specific region (returned) The memory block is of the form ENODE[NREGION].
int **PNODES	Pointer to the block of memory (freeable) that contains the complete list of nodes for all regions (returned) The block looks like NODES[ENODE[NREGION-1]].

6 Vortex Cores

6.1 FX_VortexCore

FX_VORTEXCORE(TYPE, NVCSEG, PVCSEG, PVCXYZ, PVCSTREN)
This routine returns the vortices found in the domain. They are processed as a number of segments each with a particular length.

int *TYPE	The method used to extract the core: **TYPE = 0** - vorticity vector **TYPE = 1** - eigenmodes of the Velocity Gradient Tensor
int *NVCSEG	The number of vortex core segments (returned)
int **PVCSEG	Pointer to the block of memory (freeable) that contains the core end point indices (returned) The memory block is of the form VCSEG[NVCSEG].
float **PVCXYZ	Pointer to the block of memory (freeable) that contains the vortex core points for all segments (returned) The memory block is of the form VCXYZ[VCSEG[NVCSEG-1]][3].
float **PVCSTREN	Pointer to the block of memory (freeable) that contains the vortex core strength (returned) The memory block is of the form VCSTREN[VCSEG[NVCSEG-1]].

7 Separation & Attachment Lines

7.1 FX_SepnLine

FX_SEPNLINE(SLNP, PSLXYZ, ALNP, PALXYZ)
This routine returns the separation and attachment lines found on all facets for the domain bounds. They are processed per BC with indices pointing to the end of each suite of segments.

int SLNP[NBC]	Index (bias 1) that points to the last point in SLXYZ for that group of facets. The total number of line segments is 1/2 the number of points. (filled)
float **PSLXYZ	Pointer to the block of memory (freeable) that contains the separation points – each pair creating a disjoint line segment (returned) The memory block is of the form SLXYZ[][3]. If a NULL (zero) is returned, then some error occured.
int ALNP[NBC]	Index (bias 1) that points to the last point in ALXYZ for that group of facets. The total number of line segments is 1/2 the number of points. (filled)
float **PALXYZ	Pointer to the block of memory (freeable) that contains the attachment points – each pair creating a disjoint line segment (returned) The memory block is of the form ALXYZ[][3]. If a NULL (zero) is returned, then some error occured.

8 Residence Time Routines

8.1 FX_RTParams

FX_RTPARAMS(RTTYPE, SM2, SM4, KAPPA)
This routine must be called before any other residence time functions. It is best to put this call after FX_Init when computing residence time. Once called, Residence Time is integrated (for the domain) during the call to FX_Update. If already on, this call terminates the integration.

int *RTTYPE	0 to 3 for inviscid incompressible, viscous compressible, constant viscosity and density and inviscid compressible, respectively. Returned with status – O is OK.
float *SM2	second-difference smoothing coefficient (σ_2).
float *SM4	fourth-difference smoothing coefficient (σ_4).
float *KAPPA	$\kappa = \frac{\mu}{\rho}$, required for RTYPE = 2 only.

8.2 FX_RTTimeStep

FX_RTTIMESTEP(MAXDT)
This routine can be called to get the current maximum delta-time that may be used to insure stability. The residence time equation has less of a time step constraint than either the Euler of Navier-Stokes equations, so this is not required for co-processing with explicit solvers. This call may be required when using residence time integration with steady-state solutions.

float *MAXDT	The maximum delta-time that is acceptable.

8.3 FX_RTGet

FX_RTGET(RT)
This subroutine returns the result of the shock test function.

float RT[NNODE]	The residence time for each node in the domain.

8.4 FX_RTSurface

FX_RTSURFACE(RT, RTV, NRTPT, PRTXYZ, NRTTRI, PRTTRI, PRTCELL)
This subroutine takes the residence time values, generates and returns the surface(s) at the value RTV. The surface(s) can be constructed from the triangle indices (bias 1) into the residence time nodes pointed to by PRTXYZ.

float RT[NNODE]	This must be the data returned by FX_RTGet.
float *RTV	This is the residence time value used to generate the surface.
int *NRTPT	The number of points that support the residence time surface (returned)
float **PRTXYZ	Pointer to the block of memory (freeable) that contains the coordinates (returned) The memory block is of the form RTXYZ[NRTPT][3].
int *NRTTRI	The number of triangles that make up the surface (returned)
int **PRTTRI	Pointer to the block of memory (freeable) that contains the triangle indices (returned) The memory block is of the form RTTRI[NRTTRI][3].
int **PRTCELL	Pointer to the block of memory (freeable) that contains the cell indices for the triangle (returned) The memory block is of the form RTCELL[NRTTRI].

8.5 FXmodifyRT

FXMODIFYRT(RT, DRT)
This optional call-back is invoked from FX_Update and exposes both the internal array of residence time values and the deltas to be applied. This is called just before the values are updated. FXmodifyRT allows the modification of either RT or DRT directly. This is required for special boundary conditions, such as moving interfaces, periodics boundaries or other treatments not supported.

float RT[NNODE]	Node based residence time values.
float DRT[NNODE]	Node based updates of the residence time values.

9 Boundary Layer/Wake Routine

9.1 FX_BLSurface

FX_BLSURFACE(NBLPTS, PBLXYZ, PBLD, NBLTRIS, PBLTRIS, PBLCELL)
This subroutine returns the boundary layer and wake surfaces found with the domain. The surface(s) can be reconstructed from the triangle indices (bias 1) into the BL nodes pointed to by PBLXYZ.

int *NBLPTS	The number of points that support the boundary layers (returned)
float **PBLXYZ	Pointer to the block of memory (freeable) that contains the coordinates (returned) The memory block is of the form BLXYZ[NBLPTS][3].
float **PBLD	Pointer to the block of memory (freeable) that contains the thickness – a negative value is the indication of a wake (returned) The memory block is of the form BLD[NBLPTS].
int *NBLTRIS	The number of triangles that make up the boundary layer(s) (returned)
int **PBLTRIS	Pointer to the block of memory (freeable) that contains the triangle indices (returned) The memory block is of the form BLTRIS[NBLTRIS][3].
int **PBLCELL	Pointer to the block of memory (freeable) that contains the cell indices for the triangle (returned) The memory block is of the form BLCELL[NBLTRIS].

A Non-dimensionalization

When using **FX** it is important that the supplied vector and scalar fields are provided in a consistant and non-dimensionalized manner. This non-dimensionalization is based on a number of key quantities. These values should be used to divide the *raw* quantities before supplying them to **FX**.

The key values are:

- ρ_{ref} – reference density
- u_{ref} – reference speed
- ℓ_{ref} – reference length
- μ_{ref} – reference viscosity

The following quantaties may also be important:

- Reynolds Number $= \frac{\rho_{ref} \cdot u_{ref} \cdot \ell_{ref}}{\mu_{ref}}$
- Specific Heat Ratio $= \gamma = C_p / C_v$
- $C_p = \frac{\gamma}{\gamma - 1} R$

The overbar quantaties are the dimensional (or inconsistant non-dimensional) values:

- Distance (each component of the mesh coordinates) – $X = \frac{\overline{X}}{\ell_{ref}}$
- Time – $t = \frac{\overline{t}}{\ell_{ref}/u_{ref}}$
- Velocity ($\vec{v}$) components – $u = \frac{\overline{u}}{u_{ref}}$
- Density – $\rho = \frac{\overline{\rho}}{\rho_{ref}}$
- Pressure – $p = \frac{\overline{p}}{\rho_{ref} \cdot u_{ref}^2} = (\gamma - 1)\rho[E - \frac{1}{2}|\vec{v} \cdot \vec{v}|]$
- Enthalpy – $H = \frac{\overline{H}}{u_{ref}^2} = E + \frac{p}{\rho}$
- Viscosity – $\mu = \frac{\overline{\mu}}{\mu_{ref}}$

Note: R is the gas constant, E is internal energy (usually a component of the CFD state vector – non-dimensionalized like Enthalpy). The above assumes that γ is a constant.