pV3 Programmer's Guide

Rev. 2.05
Client Side & Concentrator Programming

Bob Haimes
Massachusetts Institute of Technology

December 12, 2001

Sections marked with this change-bar have had a major change from the last major release (Rev 1.35) and will require programming modifications.

Sections marked with this change-bar have changed from the last release (Rev 2.00) and may also require programming modifications.

License

Contents

1 Introduction **6**

2 pV3 in the Message Passing Environment **7**

2.1 Using PVM Message Passing with the Simulation 7

2.2 pV3 Startup 8

2.2.1 Server Startup 8

2.2.2 Environment Variables 8

2.2.3 Special Files 9

2.2.4 Notes on the Common Desktop Environment – CDE 11

3 Programming pV3 **12**

3.1 Programming Overview 12

3.1.1 Node Numbering 12

3.1.2 Cell Numbering 13

3.1.3 Connectivity 14

3.1.4 Blanking 14

3.1.5 Surfaces 15

3.1.6 Calling Sequences 15

3.1.7 Non-Time Accurate multi-client operation 17

3.1.8 Programming Notation 18

3.2 Programmer-called subroutines 19

3.2.1 pV_Init 19

3.2.2 pV_Stat 21

3.2.3 pV_Termin 21

3.2.4 pV_Update 22

3.2.5 pV_Console 22

3.3 Programmer-supplied subroutines 23

3.3.1 pVStruc 23

3.3.2 pVCell 24

3.3.3 pVSurface 26

3.3.4 pVGrid 27

3.3.5 pVScal 27

3.3.6 pVThres 27

3.3.7 pVVect . . . 28
3.3.8 pVEquiv . . . 28
3.3.9 pVBlank . . . 28
3.3.10 pVLocate . . . 29
3.3.11 pVConnect . . . 29
3.3.12 pVPolyhedra . . . 30
3.3.13 pVPGrid . . . 31
3.3.14 pVPScal . . . 32
3.3.15 pVPThres . . . 32
3.3.16 pVPVect . . . 32
3.3.17 pVZPrime . . . 33
3.3.18 pVXYPrime . . . 33
3.3.19 pVPZPrime . . . 34
3.3.20 pVPXYPrime . . . 34
3.3.21 pVSurf . . . 35
3.3.22 pVXYSurf . . . 35
3.3.23 pVPXYSurf . . . 35
3.3.24 pVSSurf . . . 36
3.3.25 pVVSurf . . . 36
3.3.26 pVPSSurf . . . 37
3.3.27 pVPVSurf . . . 37
3.3.28 pVString . . . 38
3.3.29 pVCatch . . . 38
3.4 C Programming . . . 39

4 Concentrator **40**
4.1 PV_MPISTART – FORTRAN . . . 41
4.2 pV_MPIStart – C . . . 42
4.3 pV_MPIStop . . . 42
4.4 Building Processes that can be Concentrators . . . 45

5 Portability **46**
5.1 FORTRAN Programming . . . 46
5.2 C Programming . . . 46

A Error Codes **47**

B Multi-client Connectivity Options **49**

B.1 pVConnect 49

B.2 pVSurface 50

1 Introduction

pV3 is the newest in a series of graphics and visualization tools to come out of the Department of Aeronautics and Astronautics at MIT. Like it's predecessors **Visual3**, **Visual2** and **Grafic**, **pV3** is a software package aimed at aiding in the analysis of a particular suite of problems. In this case it is the real time visualization of 3D large scale solutions of transient (unsteady) systems.

pV3 (which stands for parallel **Visual3**), is a completely new and different system, but builds heavily on the technology developed for **Visual3**. It has been designed specifically for co-processing visualization of data generated in a distributed compute arena. It is also designed to allow the solver to run as independently as possible. If the solution procedure takes hours to days, **pV3** can 'plug-into' the calculation, allow viewing of the data as it changes, then can 'unplug' with the worst side-effect being the temporary allocation of memory and a possible load imbalance.

pV3 provides the same kind of functionality as **Visual3** with the same suite of tools and probes. The data represented to the investigator (the 3D, 2D and 1D windows with cursor mapping) is the same. Also the same Graphical User Interface (GUI) is used.

pV3 programming is very **Visual3**-like. For the desired flexibility and the merging of the visualization with the solver, some programming is required. The coding is simple; like **Visual3**, all that is required of the programmer is the knowledge of the data. Learning the details of the underlying graphics, data extraction, and movement (for the visualization) is not needed. If the data is distributed in a cluster of machines, **pV3** deals with this, resulting in few complications to the user.

In most cases, the calls or routines provided are identical to the **Visual3** programming interface. For someone familiar with **Visual3**, programming of **pV3** requires little new knowledge.

Changes in the programming interface were required to support added functionality, and the distributed nature of the compute. Some changes were due to the separation of the display workstation (the server) from the volume of data (residing in the client or clients). **Visual3** programmers must pay particular attention to the routines pV_Init, pV_Update, pVStruc, pVSurface and pVBlank.

Because the **pV3** server does not contain the entire volume of data (but only the *extracts*), compatibility with **Visual3**'s advanced programming could not be perserved. See the **pV3** Advanced Programmers Guide for details.

2 pV3 in the Message Passing Environment

The software used for the movement of data across the network is **PVM** from Oak Ridge National Laboratory. **PVM** (parallel virtual machine) is public domain software that provides the mechanisms required to transform a (heterogeneous) network of machines to one parallel computer. **PVM** provides all the required 'hooks' including efficient data transfers, message passing, synchronization and the ability to target certain machines to specific tasks.

pV3 can also been used with **MPI**. In this model, **MPI** is the interface used for the solver (or simulation) and **PVM** is used to move the data for the visualization. **pV3** uses many of the job control features of **PVM** that are not in **MPI**, including the ability to start up the server and 'plug-into' a running calculation.

MPI can also be used for moving **pV3** data on Massively Parallel Processors (MPP) or clusters with 2 (or more) networks. In this case the **pV3** Concentrator collects visualization messages via **MPI** on the machine and can communicate to the **pV3** server/batch task via **PVM**. The Concentrator also concatinates messages to the server/batch to get better **PVM** network performance – less latency. See the section on the Concentrator for the programming details.

2.1 Using PVM Message Passing with the Simulation

pV3 requires **PVM** version 3.3.0 or higher. For co-processing in a cluster of workstations (multiple clients), where the simulation is also using **PVM**, certain rules must be followed. These rules insure that messages for the visualization and those for the compute task(s) do not interfere with each other.

- Open Send Buffer

  It is assumed by **pV3** that the default send buffer is free and available for use. This should not be a restriction because either pV_Init or pV_Update should be called at times when interclient communication is at a completed stage.

- Broadcasts

  Broadcasts should be avoided. You will end up sending messages to the **pV3** server(s). The server will report then ignore messages without the proper signature. In general, any client need not send the **pV3** server any messages (all the data communication necessary is handled internally by **pV3**).

  If a broadcast facility is required, use the multiple-cast send, and send only to known tasks.

- Receives

  Do not use a wild-card for the task id in the receive calls to **PVM**. You will get **pV3** requests. The message collection in the routine pV_Update only takes messages from any **pV3** servers, leaving other client message traffic alone.

Note: The last 2 rules can now be ignored. If PVM Rev 3.4.0 (or higher) is used, then a PVM context is opened for all **pV3** communication. This avoids the possibility of message confusion between the solvers and the **pV3** components.

2.2 pV3 Startup

The **PVM** daemon(s) and with co-processing, the solver, must be executing. Without a **pV3** server running, every time the solution is updated, a check is made for the number of members in the **PVM** group *pV3Server* (Note: this name can be changed for multiple jobs running under the same user ID – see the Section 2.2.2 for the environment variable 'pV3_Group'). If no servers are found, no action is taken. When a **pV3** server starts (usually from an interactive xterm session on the graphics workstation), it enrolls in the specified group. The next time the solution is updated, an initialization message is processed and the visualization session begins. Each subsequent time in the solver completes a time step, visualization state messages and *extract* requests are gathered, the appropriate data calculated, collected and sent to the active server(s).

When the user is finished with the visualization, the server sends a termination message and exits. The clients receive the message, and if no other servers are running, cleans up any memory allocations used for the visualization. Then the scheme reverts to looking for server initialization, if termination was not specified at **pV3** client initialization.

2.2.1 Server Startup

The interactive server, *pV3Server*, takes three arguments at the command line (all optional). The first is the *setup* file with the default of 'pV3.setup'. The second argument is the *color* file to be used at startup (the default is 'spec.col'). The third argument is the *extract start-up* file. There is no default. An argument of '−' is a place holder, allowing the default to be used.

examples: % pV3Server case.setup
% pV3Server pV3.setup bw.col
% pV3Server
% pV3Server case.setup − case.startup

2.2.2 Environment Variables

The **pV3** server uses six Unix environment variables. Some are the same as the ones used for **Visual3**. The variable 'Visual3_CP' defines the file path to be searched for color files, if they are not in the user's current directory. This allows all of the color files to be kept in one system directory.

The second variable, 'Visual_KB' is optional. This variable, if defined, must point to a file that contains alternate keyboard bindings for the special keys used by **pV3**. The file is ACSII. The first column is the key name (10 characters) and the second is the X-keysym value in decimal (use 'xev' to determine the appropriate values for the key strokes).

The third is 'pV3_TO' and should be used to change the internal Time-Out constant. If the variable is set, it must be an integer string which is the number of seconds to use for the Time-Out constant (the server's default is 60). This may be required if the time between solution updates is long. See the section in the **pV3** Server User's Reference Manual on *Time-Outs and Error Recovery.*

The fourth is 'pV3_Threading'. This is used to specify how the interactive Server and the post-processing Viewer handles the handshaking between their active threads. There are two methods; (1) 'Hard' where the thread sits in a hard loop (with a thread yeild) looking for a change of state,

or (2) 'Flag' where the threads use Semaphores for waiting until the state changes. The advantage of 'Hard' is interactivity, the advantage for 'Flag' is less processor time consumed. By default, this variable is set to 'Hard' for most situations with single processor workstations and 'Flag' for multi-processors.
Exception: The ALPHA and SUN default is 'Hard' reguardless of number of processors.

The fifth is 'pV3_Group'. This is usefull for differentiating multiple **PVM** jobs running under the same user ID. If this variable is set for the solver (client-side) before execution, it overrides the default client side group name *pV3Client*. The name used is the string assigned to this variable with *Client* appended. By setting this variable before server execution, it will set the server group to the variable's string with *Server* appended instead of using *pV3Server*. Only clients with the appropraite matching group name will be connected to this session.

The last is 'pV3_Warning'. If this variable is set (to anything except a NULL string) warning messages about ACK and maximum streamline segments are not reported.

2.2.3 Special Files

- Window Manager Resource File

  By default, all modern Xwindows Managers allow the closing or deleting of windows by either double-clicking on the menu pull-down or selecting 'Close' or 'Quit' from the pull-down. This will abort the execution of the Server or Viewer. To avoid this, an additional window manager menu description can be added to the default information for the Window Manager. This is accomplished by specifying a user-level resource. In the distribution the following files can be found in the "servers" subdirectory; 'user.4Dwmrc', 'user.mwmrc', and 'user.dtwmrc'. These files are for SGI's default window manager, the Motif window manager and CDE's window manager, respectively. The entire resoure file (for the appropriate window manager) must be copied from the system (usually something like '/usr/lib/X11/system.XXwmrc' or '/etc/dt/config/sys.dtwmrc') – if not already done. This file gets named '.XXwmrc' (where XX is 4D or m) and then the appropriate user resource file appended to the end. Note: for CDE, this file gets put in the '.dt' directory at the users top level and must be given the name 'dtwmrc'.

- .Xdefaults

  If the SGI window manager is used, '4DWm' must be told what to do with **pV3**s windows. These commands must be placed in the file '.Xdefaults'. See the file 'user.Xdefaults'. For DEC, IBM and SUN systems this information is found in the window manager's Setup File (Mwm or the appropriate file for the WM used).

  pV3 requires three X fonts. The file '.Xdefaults' in the users home directory is examined for the font names and are designated "Visual*large", "Visual*medium" and "Visual*small". The sample file 'user.Xdefaults' comes with the distribution and may be concatinated to the user's '.Xdefaults' file.

  The X fonts loaded on any system may be examined by the command 'xlsfonts'.

- 'Mwm' Setup File

  When a user begins a session using the Motif window manager, it reads the initialization file 'Mwm' in the user's home directory. This file contains information on how to treat various classes of windows and the window focus (as well as other things). A sample file 'user.Mwm' is in the distribution and may be used as 'Mwm' or its contents concatenated to the user's 'Mwm' file.

- 'twm' Setup File

  If the 'twm' window manager is used, when a user begins a session, twm reads an initialization file '.twmrc' in the user's home directory. This file defines certain key bindings and window attributes. It is necessary for correct **pV3** operation for the user to modify the standard '.twmrc' file as shown in the 'user.twmrc' file on the **pV3** distibution, or just use this file as '.twmrc'.

- Color Files

  A number of different color files are supplied on the distribution. For those who wish to define their own color files, the format of these ASCII files is as follows:

$$
\begin{array}{l}
nc \quad nb \\
\left.\begin{array}{l} r \quad g \quad b \\ \quad . \\ \quad . \\ \quad . \\ \quad . \end{array}\right\} nc \\
\left.\begin{array}{l} r \quad g \quad b \\ \quad . \\ \quad . \\ \quad . \end{array}\right\} nb
\end{array}
$$

  where nc is the number of colors, nb is the number of background colors (0–4), and r, g, b are red,green,blue intensity values (0.0–1.0). The four background colors are for window background; grid color; tuft/streamline/ribbon color; contour line color. The default values which are used if $nb = 0$ are black; white; white; white. If $nb \neq 0$ then the specified colors over-ride the defaults for the first nb colors.

  When **pV3** searches for named color files, it first looks in the current directory, and then follows the color file path specified by the environment variable 'Visual3_CP'.

- Setup File

  When the **pV3** server starts, it looks in the user's current directory for the setup file specified as the first argument on the command line. This is an ASCII file which contains a number of useful defaults that the user may want to set to different values than **pV3**'s initial defaults. It also contains a set of viewing positions and cutting plane positions. Normally this file is generated by **pV3** when the user wants to store certain favorite parameters and viewing positions so that they can be used again on another data set, which is particularly useful when the user wishes to directly compare two different data sets with the same computational grid

or geometry. However, an experienced user can also generate this file from scratch. NOTE: This file is NOT compatible with **Visual3**'s setup file.

- Start-up file

  The startup file is an ASCII list of static *extracts* that are desired at server initialization. This file can be generated by the **pV3** server at any time during the visualization session by hitting 'W' in the Key window. The result is a startup file that contains all of the saved *extracts* as listed in the Surface database and the Streamline database of the Dialbox window. This allows for easy restart.

  NOTE: The format for this type of file has changed for Rev 1.20 in order to support multi-disciplinary visualization. Older files are still valid at startup for single discipline cases.

- Lock File

  If the server is running on a multi-processor SGI workstation a file is used for the coordination of the 2 threads generated during execution. This file has the name 'pV3.lock' and is open in the current directory. It should be noted that running two invokations of the **pV3** server from the same directory will NOT work. Both will use the same file for the lock and semaphore arena!

2.2.4 Notes on the Common Desktop Environment – CDE

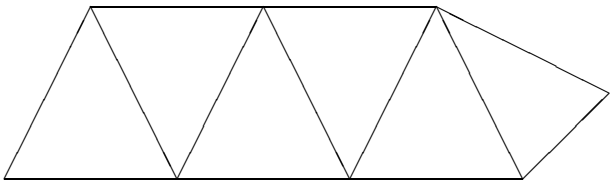
For the **pV3** suite to work properly with CDE, the Style Manager must be used to change the default methods CDE uses for cursor/window functioning. Under the Window section "Point in Window to Make Active" must be selected. Also, "Raise Window when Active", and "Allow Primary Windows on Top" must NOT be selected.

3 Programming pV3

3.1 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 **pV3**. The programmer gives **pV3** a list of unconnected simple cells, poly-tetrahedral strips, structured blocks and disjoint complex polyhedra. The simple 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 (or use the complex polyhedral option). Linear interpolation is used throughout **pV3**, so high order elements must be also be subdivided so that the linear interpolation assumption is valid.

Poly-tetrahedral strips are 'structured' collections of tetrahedra. The strip is started by a triangular face, one node is added to produce the first tetrahedron, another is added to produce the second cell (with the previous 3 nodes) and so forth. See Figure 1. Currently, no one is using this concept for calculating results but there is more than a factor of two savings in the storage required to represent a complete tetrahedral mesh.



tetra	#1	tetra	#2	tetra	#3	tetra	#n (last)
nodes:	1 2 3 4	nodes:	2 3 4 5	nodes:	3 4 5 6	nodes:	n n+1 n+2 n+3
face	nodes	face	nodes	face	nodes	face	nodes
1	1 2 3	1	-	1	-	1	-
2	-	2	-	2	-	2	n+1 n+2 n+3
3	3 4 1	3	4 5 2	3	5 6 3	3	n+2 n+3 n
4	4 1 2	4	5 2 3	4	6 3 4	4	n+3 n n+1

Figure 1: Poly-Tetrahedral Strip

Polyhedral cells are elements that are more complex than the simple disjoint primitives. This type of cell can be the result of agglomoration or the cut-cells found in Cartesian meshes. These cells are treated in an unusual manner. They are always (re)generated *on-the-fly*. The polyhedra can be comprised of of two types of nodes; (1) local to, and only seen by the element and (2) from the node space.

3.1.1 Node Numbering

The node numbering used within **pV3** is local. For multiple processor cases, this numbering need not have any reference outside the data on the client.

The node numbering used differentiates between the nodes in the non-block regions (formed by the disjoint cells and poly-tetrahedral strips) and the structured blocks. Figure 2 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 **pV3** node number $= base + i + (j-1) * NI_m + (k-1) * NI_m * NJ_m$. Note: all indices start at 1.

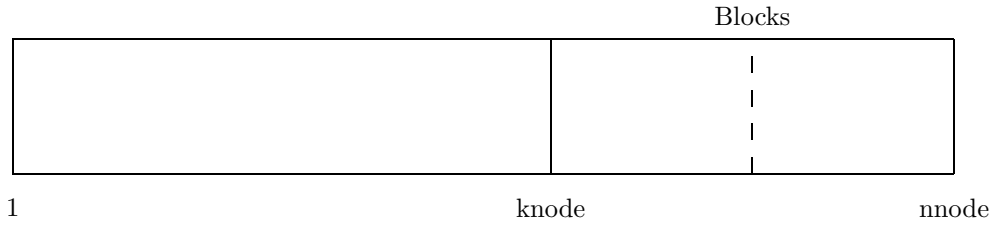


Figure 2: Node Space

3.1.2 Cell Numbering

The non-block cell types may contain nodes from the non-block and the structured block volumes. The cell numbering used within **pV3** orders the cells by type. Figure 3 shows a schematic of the cell space. The programmer explicitly defines all non-block cells by the call pVCell. 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 **pV3** cell number $= base + i + (j-1) * (NI_m - 1) + (k-1) * (NI_m - 1) * (NJ_m - 1)$. Note: i goes from 1 to $NI_m - 1$, j goes from 1 to $NJ_m - 1$, and k goes from 1 to $NK_m - 1$.

Because the polyhedral cells are not stored within **pV3**'s structures, they are not explicity defined but do populate the cell space at the end.

Again, the numbering is local to the client for multiple processor applications.

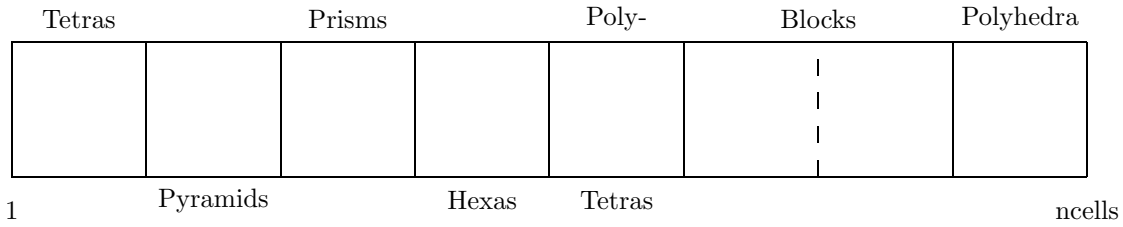


Figure 3: Cell Space

3.1.3 Connectivity

In order to calculate streamlines and particle paths from vector fields **pV3** requires information about which are cells are neighbors, i.e. share a common face. There are two options; (1) either the programmer gives **pV3** these connections by setting IOPT (of pV_Init) negative and supplying the routine pVConnect, or (2) **pV3** calculates this information by processing all cells with exposed faces. This process compiles a list of all of these faces and checks whether the face appears on another cell. If it does appear twice, then it is an interior face, and a streamline or particle can pass through from one cell to its neighbor. If it does not appear twice, then it is a surface face on the boundary of the computational domain, and a streamline will terminate when it hits the face.

Face matching between structured blocks is not possible using this automatic scheme. The node numbers that make up a face are different in both blocks even if they match in 3 space. The concept of 'node equivalency' allows the face matching to patch between regions. Node equivalency is simply a list of matching node numbers that get used only during this face matching procedure. This concept is generalized in **pV3** to allow equivalency to nodes anywhere in the local node space.

Anytime a streamline crosses a structured block, blanked or domain boundary and the integrator is able to continue (based on 'node equivalency' or other specified options), the accuracy is reduced for that segment.

Connectivity is supplied explicitly by the call-back pVPGrid for the complex polyhedral cells. Connections between non-polyhedral regions of the mesh and the polyhedra are accomplished via pVSurface.

3.1.4 Blanking

Blanking is an option (see the description of pVStruc) and only used with structured blocks to indicate that some region of the block is 'turned off'. This information is also used to give **pV3** an indication of how multi-block grids are connected in these areas. 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. A cell with an invalid node is considered not to exist and therefore cuts and iso-surfaces through that cell will not be plotted. Also streamlines will not pass through cells with invalid nodes.

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. The value of one is the indication of an interior point, the value of two marks the node as a wall. A negative value means that the physical space 'continues' in the block number that is the absolute value of this IBLANK entry.

For most meshing topologies **pV3** does not require knowing which nodes are wall (i.e., IBLANK = 2) because these tend to match the domain boundaries. There are special situations where a surface is in the midst of the block. In this case, streamlines and particle paths will terminate when hitting cell faces where all nodes that make up the face have IBLANKing that indicates a wall.

pV3 uses an algorithm for integrating streamlines and particle paths accross blanked boundaries by finding the closest node to the required position in the target block that has an IBLANK entry equal to the original block. If the negative blanked region is at the boundary of the block and all IBLANK entries of the exitting face are the same (and the case is not grid unsteady) the connectivity

information is updated with the connected cell if found. Future integrations in the current session that pass through this face will not require the IBLANK node searching.

In the case of C-meshes and other topologies where the blocks abut, it is advisable to use 'node equivalency', if appropriate. Streamlines and particle paths are always faster going through a volume that has had face matching.

When the visualization is grid or structure unsteady, a new IBLANK array is requested for each snap-shot in time (after the coordinate triads are retrieved). If the blanking has not changed, the data need not be updated, and pVBlank should just return.

3.1.5 Surfaces

In principle, all exposed surface facets could be grouped together to form one bounding surface for plotting purposes. However, in many applications it is more useful to split the bounding surface into a number of pieces, referred to here 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 far-field boundary, the symmetry plane, the fuselage and the wing. If the programmer specifies each of these as separate *domain surfaces* by grouping the appropriate faces, then **pV3** can offer the capability of plotting on just one or two of the surfaces (e.g. the fuselage and wing) and not on the others (far-field and symmetry plane).

Internal surfaces are those that get created when the computational domain is sub-divided and and placed in multiple computational clients or to connect non-polyhedral regions to polyhedra. These surfaces need only be defined to allow the passage of data (during integrations) from one region to another.

pV3 also suports the concept of a *bucket-of-faces* in much the same way as the exposed facets are handled. The difference is that the *bucket-of-faces* is not used for patching volumes together. In fact this option should be used when there is some feature of the mesh that should be displayed that is internal to the grid. This surface definition is completely general, in that there is no assumption that the triangles or quadrilaterals (that define the surface) are faces of the elements in the mesh.

3.1.6 Calling Sequences

pV3 supports steady-state visualization as well as three types of unsteady visualization. In a multi-client simulation, each client can have a different mode of unsteadyness. Each mode causes a different internal calling sequence. In general, the application must first call pV_Init to initialize the **pV3** client subsystem and then call pV_Update after every time the solution space has been updated.

- Steady-State

Call	Calls in Sequence
pV_Init	pVStruc
	pVCell (optional)
	pVSurface
	pVEquiv (optional)
	pVGrid
	pVBlank (optional)
	all others as needed
pV_Update	NOT required

pV_Init does not return until the visualization session is over.

- Data Unsteady

Call	Calls in Sequence
pV_Init	pVStruc
	pVCell (optional)
	pVSurface
	pVEquiv (optional)
	pVGrid
	pVBlank (optional)
pV_Update	pVScal
	pVVect (optional)
	all others as needed

- Grid Unsteady

Call	Calls in Sequence
pV_Init	pVStruc
	pVCell (optional)
	pVSurface
	pVEquiv (optional)
pV_Update	pVGrid
	pVBlank (optional)
	pVScal
	pVVect (optional)
	all others as needed

- Structure Unsteady

Call	Calls in Sequence
pV_Init	pVStruc – used to size for memory allocation
pV_Update	pVStruc
	pVCell (optional)
	pVSurface
	pVEquiv (optional)
	pVGrid
	pVBlank (optional)
	pVScal
	pVVect (optional)
	all others as needed

The following routines are only required for cases with complex polyhedral elements:

Call-Back	Comment
pVPolyherda	
pVPGrid	
pVPScal	required for local node scalars
pVPThres	required for local node threshold values
pVPVect	required for local node vectors
pVPZPrime	required for local programmed-cut nodes
pVPXYPrime	for mapping local programmed-cut nodes to the 2D window
pVPXYSurf	for mapping local surface nodes to the 2D window
pVPSSurf	for local node surface scalar values
pVPVSurf	for local node surface vector values

3.1.7 Non-Time Accurate multi-client operation

The default mode of operation for **pV3** is a lock-step synchronous mode where all clients call the routine pV_Update after the solution is sync'ed and the field varaibles have been updated. All clients should be at the same simulation time (the argument passed to pV_Update). This default method is considered *Time Accurate*.

There is another mode of operation where the processes may not be at the same simulation time. This may be used when a solution is using time-marching to get to a steady state. Under these circumstances, where 'local time-stepping' is used, it is not as important to update the internal boundary every iteration. If this method is being used in the solver, **pV3** can be set in a *Non-Time Accurate* mode. Therefore when pV_Update is called and there are no pending requests from the server, no action is taken and control is immediately returned to the calling routine. When **pV3** is in a *Non-Time Accurate* method of operation both particle tracking and streamlines are disabled.

The call to pV_Init informs **pV3** which method to use. If the case is either steady-state or there is only one client, the *Time Accurate* method is used reguardless of the information at initialization.

NOTE: do not use *Non-Time Accurate* functioning while debugging a solver. You will miss frames. In this case, set the simulation time (passed to pV_Update) to the iteration number and run in a *Time Accurate* mode.

3.1.8 Programming Notation

pV3 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. Also the following routine descriptions are from the FORTRAN programmer's point of view. All subroutines internal to **pV3** have names which begin with 'pV_' or 'XFtn' and the common blocks have names which begin with 'PVC': such names should be avoided in an application program.

In describing the arguments of the routines in the next sections, the following notation has been used. The variable name is first followed by 'i' or 'o', indicating input and output, respectively, showing whether or not the subroutine is to set the variable. The variable is then followed by an expression indicating the variable type (I=integer, R=real, C=character) and its dimensionality (e.g. R(2,MNODE) indicates a two-dimensional real array with the first dimension being 2 and the second MNODE, where MNODE is an integer variable).

For all routines described in this Programmer's Guide, the same bindings are used for both FORTRAN and C (except for pV_MPIStart – see the Concentrator section).

3.2 Programmer-called subroutines

3.2.1 pV_Init

PV_INIT(TITL, CID, CNAME, DNAME, IOPT, NPGCUT, TPGCUT, NKEYS, IKEYS, TKEYS, FKEYS, FLIMS, MIRROR, REPMAT, MAXBLK, ISTAT)

This subroutine initializes **pV3**. This process involves enrolling the task in the group 'pV3Client' which is required for connection to the graphics workstation. Calling this routine also defines the type of case, the sizes of various parameters and the types of functions defined. Returns immediately for all cases except steady-state ($IOPT = 0$).

TITL:i: C	Title (up to 80 characters used)
CID:i: I	This informs **pV3** of the unique client-id for this client within a multi-client application. The index must be in the range 1 to the total number of clients in this discipline.
CNAME:i: C*(*)	This string identifies the partition (client) with a name (up to 20 characters are used).
DNAME:i: C(*)	The name for the discipline. Required to be non-blank for multi-discipline cases (up to 20 characters used).
IOPT:i: I	Unsteady control parameter
	IOPT=-3 structure unsteady with connectivity supplied
	IOPT=-2 unsteady grid/data with connectivity supplied
	IOPT=-1 steady grid and unsteady data with connectivity supplied
	IOPT=0 steady grid and data
	IOPT=1 steady grid and unsteady data
	IOPT=2 unsteady grid and data
*NPGCUT:i: I	Number of programmer-defined cuts
*TPGCUT:i: C(NPGCUT)	Title for each cut (up to 32 characters used)
*NKEYS:i: I	Number of active keyboard keys
*IKEYS:i: I(NKEYS)	X-keypress return code for each key
*TKEYS:i: C(NKEYS)	Title for each key (up to 32 characters used)
*FKEYS:i: I(NKEYS)	Type of function controlled by each key:
	FKEYS()=1 Scalar
	FKEYS()=2 Vector
	FKEYS()=3 Surface scalar
	FKEYS()=4 Surface vector
	FKEYS()=5 Threshold

FLIMS:i: R(2,NKEYS) — Function limits/scales

FKEYS()=1,3,5 Min and max values of function

FKEYS()=2,4 Arrow/tuft scaling (only the first element is used)

MIRROR:i: I — Mirror/Replication flag:

MIRROR=-nrep Replicate the data *nrep* times using REPMAT

MIRROR=0 No mirroring or replication

MIRROR=1 Mirror about the plane X=0.0

MIRROR=2 Mirror about the plane Y=0.0

MIRROR=3 Mirror about the plane Z=0.0

REPMAT:i: R(16) — The Replication matrix (required for negative MATRIX values only).

MAXBLK:i: I — The maximum number of structured blocks used during the session.

ISTAT:i/o: I — On input this sets the startup/terminate state (the following are additive):

0 do not wait, do not terminate and *Time Accurate*

1 wait for the server to startup (the first time)

2 terminate with the server

***4** *Non-Time Accurate* mode

$ISTAT = 3$ is the only valid condition for steady-state cases ($IOPT = 0$).

On output any non-zero value is the indication of a startup error and the task is not included in the **pV3** client pool. See the Appendix for a list of the error codes.

Notes:

$*$) Multi-client cases: these parameters must match for all clients in a single discipline.

1) The X-keypress return codes for alphanumeric keys is identical to their usual ASCII integer codes.

2) If NKEYS is negative, the absolute value of NKEYS is used for the number of keys and streamline/ribbon/tube/bubble calculations are disabled. This frees up a large amount of memory for $IOPT = 0, 1, 2$ cases (the cell connection information is discarded).

3) It is not legal to mix steady-state ($IOPT = 0$) with unsteady clients.

4) The 4x4 matrix REPMAT is multiplied by the coordinates for the mesh repeatedly (up to $-MATRIX$ times) to fully represent the data in this client.

5) If a **PVM** error occured ($istat = -101$ upon return), the application can re-call pV_Init at some later time to again attempt to initialize **pV3**. A call to pV_Update has no effect if the pV_Init call returns this error.

3.2.2 pV_Stat

PV_STAT(ISTATE)

This subroutine allows the programmer to query the status of the **pV3** system.

ISTATE:o: I	the status:
	ISTATE < 0 **:** error code from **pV3** - positive errors are designated by $-(1000 + code)$
	ISTATE=0 client not initialized
	ISTATE=1 no server
	ISTATE=2 server active

3.2.3 pV_Termin

PV_TERMIN

This subroutine gracefully removes the client from the **pV3** system by leaving the group 'pV3Client' and deallocating associated memory.

No Arguments	pV_Init must be called again to use **pV3**

3.2.4 pV_Update

PV_UPDATE(TIME)

This subroutine must be called after the solver has updated the solution space. This is when the data is extracted and communication between the client(s) and the graphics workstation is done. The call to this routine is not needed if $IOPT = 0$.

TIME:i: R	The current simulation time. This sets the time in **pV3** for particle integration. Not used if the application is *Non-Time Accurate*.

This routine is where all interaction with the graphics workstation is performed. Therefore the overall response and the interactive latency depends on how often this routine is called. About one call per second is optimal. If the soultion is updated significantly faster then most of the compute cycles will be used for the visualization, moving the solution slowly forward in time. In this case it is advisable to call pV_Update only every N times the solver updates the solution.

The more difficult case is when the solution update rate is much slower than optimal. In this situation, there are two choices; (1) live with large lags between user requests and screen response or (2) setup another task between the solver and the **pV3** server. This software's responsibility is to communicate with the solver. It should be the task to make all **pV3** calls.

This secondary task can communicate with the solver using **PVM** or **MPI** (and therefore must be on the same machine to avoid large network transfers). Or, if the machine supports multi-threading, the task can be a second thread and perform double-buffering of the solution space, so no data need be transferred. These methods are a trade-off of memory usage for interactivity. Multi-thread examples can be seen in the distribution in the directory 'examples/mthread'.

3.2.5 pV_Console

PV_CONSOLE(STRING)

This subroutine allows the programmer to have a string printed in the text window of all currently running **pV3** server applications. The string will be prefaced by the client-id for multi-client cases.

STRING:i: C*80	character string to be output to the server

If a server is not running, this call does nothing.

3.3 Programmer-supplied subroutines

3.3.1 pVStruc

PVSTRUC(KNODE, KEQUIV, KCEL1, KCEL2, KCEL3, KCEL4, KNPTET KPTET, KNBLOCK, BLOCKS, KPHEDRA, KSURF, KNSURF, HINT)

This subroutine is required for all cases, but is only called multiple times for structure unsteady cases ($IOPT = -3$). This routine supplies the information for the structure of the discretization.

KNODE:o: I	Number of non-block nodes / static flag
KEQUIV:o: I	Number or node equivalency pairs
KCEL1:o: I	Number of tetrahedra
KCEL2:o: I	Number of pyramids
KCEL3:o: I	Number of prisms
KCEL4:o: I	Number of hexahedra
KNPTET:o: I	Number of poly-tetrahedral strips
KPTET:o: I	Number of cells in all poly-tetrahedra
KNBLOCK:o: I	Number of structured blocks NOTE: A negative value indicates that blanking will be supplied for the blocks.
BLOCKS:o: I(3,KNBLOCK)	Structured block definitions: **BLOCKS(1,m)** = NI **BLOCKS(2,m)** = NJ **BLOCKS(3,m)** = NK
KPHEDRA:o: I	Number of complex polyhedral cells
KSURF:o: I	Number of domain surface faces NOTE: A negative value indicates that faces not connected to cells should be allowed (not required for *bucket-of-faces*).
KNSURF:o: I	Number of domain surface groups
HINT:o: I	What to do with particle locations (used for $IOPT = -3$): **0 - nothing** - if the old cell number is still valid, use it to start **1 - nearest node** - use the nearest node to find a valid cell **2 - nearest surface node** - use the nearest surface node to find a valid cell **3 - use supplied info** - call pVLocate to get the new cell

Notes:

1) A domain surface group is a collection of faces, which do not have to form a single connected surface, but form instead a logical grouping referred to earlier as being a *domain surface*. **pV3**'s initialization phase ($IOPT = 0, 1, 2$) examines each exposed face of each primitive, and determines

whether or not it is shared with a neighboring cell. If not, it must be a surface face, but the user may choose to not declare it as such (see pVSurface). In this case, **pV3** takes all undeclared surface faces, splits them up into disjoint collections and calls the collection the 'Others' surface group. Therefore, the final number of domain surface groups can exceed KNSURF by one.
2) KNODE may be zero for purely structured block cases where the acutal number of nodes can be determined from the block sizes. It can also be zero for homogenous polyhedral cases were all nodes are *local* to the elements (see pVPolyhedra).

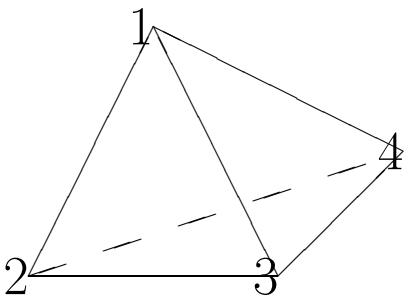
Notes for $IOPT = -3$ cases:
1) If KNODE is -1 that is a special flag to indicate that the structure has NOT changed for this iteration. With this flag set, no other paramaters should be modified, in that **pV3** reverts to the grid unsteady calling sequence.
2) Gets called every time in pV_Update even if no visualization is active.
3) Performance is enhanced by using HINT = 3. The other options exist in the case that a translation from the old structure does not exist. In these cases, pick the option that, in general, gives a cell number closest to the target. HINT = 2 is initially less compute intensive, but may require many cell *walks* to get to the actual location. Calls to pVConnect will be used to locate the the actual cell once the integration begins.

3.3.2 pVCell

PVCELL(CEL1, CEL2, CEL3, CEL4, NPTET, PTET)
This subroutine supplies **pV3** with the grid data structure. It is not required for a grid that contains only structured blocks and/or complex polyhedral elements.

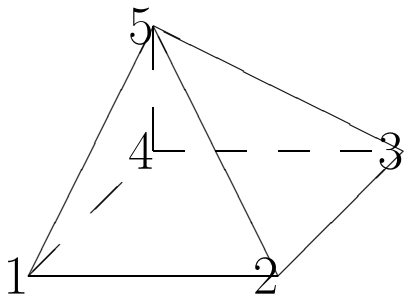
CEL1:o: I(4,KCEL1)	Node indices for tetrahedral cells
CEL2:o: I(5,KCEL2)	Node indices for pyramid cells
CEL3:o: I(6,KCEL3)	Node indices for prism cells
CEL4:o: I(8,KCEL4)	Node indices for hexahedral cells
NPTET:o: I(8,KNPTET)	Poly-Tetrahedra strip header: **NPTET(1,n)** = the pointer to the end of the strip n, i.e. it points to the last entry in PTET for the poly-tetrahedral strip **NPTET(2,n)** = the first node in the poly-tetrahedra **NPTET(3,n)** = the second node in the poly-tetrahedra **NPTET(4,n)** = the third node in the poly-tetrahedra **NPTET(5-8,n)** are used by **pV3**
PTET:o: I(KPTET)	The rest of each poly-tetrahedra, 1 node per cell

Notes:
1) If KCELn is zero, the corresponding CELn must NOT be filled. And the same holds true for NPTET and PTET.
2) The correct order for numbering nodes for the four disjoint cell types is shown in Fig. 4. The Poly-Tetrahedra numbering is shown in Fig. 1.



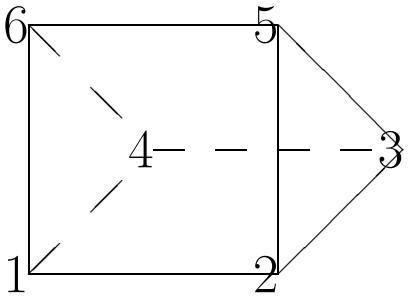
face	nodes
1	1 2 3
2	2 3 4
3	3 4 1
4	4 1 2

Tetrahedron



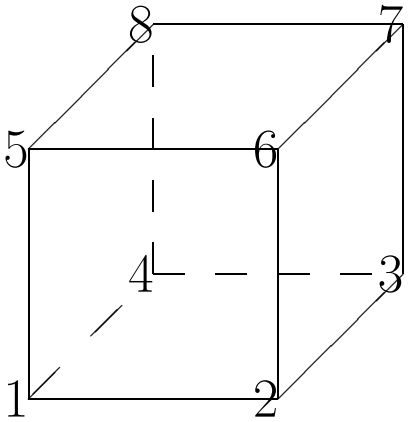
face	nodes
1	1 2 3 4
2	2 3 5
3	3 4 5
4	4 5 1
5	5 1 2

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

3.3.3 pVSurface

PVSURFACE(NSURF, SCON, SCEL, TSURF)

This subroutine supplies **pV3** with the surface data structure.

NSURF:o: I(3,KNSURF)

NSURF(1,n) is the pointer to the end of domain surface group n, i.e. it points to the last entry in both SCON and SCEL for that group.
NSURF(2,n) is the startup drawing/mapping state (the following are additive):

0 off

1 render

2 grid

4 grey flag

8 thresholded

16 contours

32 translucent

256 2D mapping is provided (see note 1)

NSURF(3,n) is the global surface number (needed for multi-client cases only). A non-positve number is the indication that the surface is an internal boundary caused by domain decomposition. The number must be the client-id (negated) or zero. Zero is a special flag (along with $SCON = -1$) to allow an integration to try all other clients. Any positive value (including zero) with a positive $SCON$ is an indication that this surface connects internally to this client at the cell indicated (usefull for periodic replication). The global surface number must be <= 3599 (except when indicating a periodic surface or a *bucket-of-faces*)! See Appendix B and note 5.

SCON:o: I(KSURF)

The cell number to connect. This is the cell number in the local cell space of the client-id (specified above) for the connecting cell. If the value is -1, then an attempt is made to pass the particle into that domain or if NSURF(3,n) is zero all domains (except the current). A -1 if NSURF(3,n) is greater than zero, or for single client cases indicates that a re-enter attempt should be tried in this volume of data. A value of zero signals that there is no connection.

SCEL:o: I(4,KSURF)

node numbers for surface faces. For quadrilateral faces SCEL must be ordered clockwise or counter-clockwise; for triangular faces, SCEL(4,n) must be set to zero.

TSURF:o: C*20(KNSURF)

titles for domain surfaces (optional)

Notes:

1) If the 2D mapping bit is set in NSURF(2,n), that is a flag to indicate that this surface has a 2D mapping. pVSurf, pVXYSurf, and optionally pVSSurf and pVVSurf will be provided and will respond to this surface.

2) The correct order for numbering faces for the four disjoint cell types is shown in Fig. 4. The face definitions for Poly-Tetrahedral cells is displayed in Fig. 1. 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 $j = 1$.

3) See Appendix B for a table of NSURF(3,n) and SCON options.

4) Groups may have no entries (in that they may be filled by facets constructed with *local* polyhedral nodes).

5) A *bucket-of-faces* is any collection of triangles or quadrilaterals that do NOT bound the domain. This can be used to display some feature that may be in the interior of the domain. No connectivity options are used. In fact the facets described need not be faces of cells.

3.3.4 pVGrid

PVGRID(XYZ)

This subroutine supplies **pV3** with the grid coordinates.

XYZ:o: R(3,NNODE)	(x, y, z)-coordinates of grid nodes, using left-handed coordinate system. If right-handed coordinates are desired reverse sign of the z values.

3.3.5 pVScal

PVSCAL(JKEY,S)

This subroutine supplies **pV3** with scalar function values (FKEY=1).

JKEY:i: I	Key index, relative to ordering specified in pV_Init. (i.e. first key is 1, second is 2, third is 3, etc.)
S:o: R(NNODE)	Scalar function values.

3.3.6 pVThres

PVTHRES(JKEY,XYZ,T)

This subroutine supplies **pV3** with threshold function values (FKEY=5).

JKEY:i: I	Key index
XYZ:i: R(3,NNODE)	(x, y, z)-coordinates of grid nodes
T:o: R(NNODE)	Threshold function values

Notes:

1) XYZ is passed by **pV3** to the user subroutine, in case it is needed to calculate T but, for storage reasons, the user's program has not kept a copy of XYZ. XYZ must not be changed by pVThres.

3.3.7 pVVect

PVVECT(JKEY,V)
This subroutine supplies **pV3** with vector function values (FKEY=2).

JKEY:i: I	Key index
V:o: R(3,NNODE)	Vector function values (Vx, Vy, Vz). If right-handed coordinates are desired reverse sign of the Vz values.

3.3.8 pVEquiv

PVEQUIV(LISTEQ)
This subroutine supplies **pV3** with node equivalency data. Required for $KEQUIV \neq 0$ cases.

LISTEQ:o: I(2,KEQUIV)	Node equivalency pairs.

Notes:
1) For multiple (more than 2) node matching, always have the lowest node number in each entry. e.g, for a node at an edge between 4 blocks that line up, 3 node equivalency pairs are required, each matching the lowest node number with the others.

3.3.9 pVBlank

PVBLANK(IBLANK, TBCON)
This subroutine supplies **pV3** with blanking data. Required for $KNBLOCK < 0$ cases.

IBLANK:o: I(NNODE-KNODE)	Blanking data: < 0 node is valid and connected to this local block number (negated) - fill TBCON for multi-client cases. $= 0$ off, invalid node > 0 on
TBCON:o: I(NNODE-KNODE)	Client-id for block number. Zero indicates this client (multi-client cases only).

3.3.10 pVLocate

PVLOCATE(PXYZ,KCOLD,KCNEW)

This subroutine supplies **pV3** with the cell locations for particles in an $IOPT = -3$ and $HINT = 3$ case. This will be called for each active particle and many of the StreamLine seed positions.

PXYZ:i: R(3)	The current (pre-integrated) position of the particle.
KCOLD:i: I	Cell number in the old structure that contained the point PXYZ.
KCNEW:o: I	The new cell number that contains PXYZ, used to continue (or start) the integration (this does not have to be the actual cell that contains the point, but should be close). A zero indicates that there is no new cell (the domain no longer exists at the location). If the location is now in another client, return a valid cell number (in this client) that will cause the integration to continue to the target client when pVConnect is called.

3.3.11 pVConnect

PVCONNECT(KCOUT,KFOUT,KCIN,IDTIN)

This subroutine supplies **pV3** with cell connectivity for cases where $IOPT < 0$. This is not called for the interior of structured blocks or poly-tetrahedral strips and is not required for complex polyhedral cells.

KCOUT:i: I	Exitting cell number for the integration
KFOUT:i: I	The face number of the cell
KCIN:o: I	The entering cell number inorder to continue the integration. A zero indicates that there is no entering cell (the entire domain has been exitted). A negative number is an indication that the integration should attempt to re-enter the domain. The number must be the index to SCEL of the exitting face for single client cases or when IDTIN is -1. This insures that a re-entry will not occur to that face.
IDTIN:o: I	Multi-client only. A -1 is the flag that the cell number is in this client. Zero flags an attempt to reenter any other client's domain (all values of KCIN except zero are ignored). Any other value specifies the client-id for the client that has the cell. Re-entries (negative KCINs) will be attempted in this client.

Notes:

1) See Appendix B for a table of IDTIN and KCIN options.

The following routines are required for cases with complex polyhedral elements.

3.3.12 pVPolyhedra

PVPOLYHEDRA(INDEX,NLNODE,NTETS,NFACET)

This subroutine returns the details of the complex polyhedron specified by INDEX. It is the responsibility of this routine and pVPGrid to provide **pV3** enough information about this cell so that the visualization tools can be applied.

INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
NLNODE:o: I	The total number of *local* vertices (i.e. the points not explicilty defined in **pV3**'s node space).
NTETS:o: I	The total number of tetrahedra that are required to fill the volume occupied by the polyhedron.
NFACET:o: I	Number of exposed triangular facets for the tetrahedral subdivision.

3.3.13 pVPGrid

PVPGRID(INDEX,XYZ,TETS,DSG,SCN)

This subroutine supplies **pV3** with scalar function values (FKEY=1) for the local nodes in the specified polyhedron.

INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
XYZ:o: R(3,NLNODE)	The coordinates for the local nodes.
TETS:o: I(8,NTETS)	Tetrahedra and connectivity definitions: **1-4** Nodes that support the tetrahedron. Any positive number refers to a node **pV3**'s node space. Any negative value indicates the index (bias 1) to a local node (absolute value). **5-8** The neighboring cell element for the face (see Fig. 4 for the tetrahedron face definitions). Any positive number refers to the index for another tetrahedon from within TETS. Any negative value indicates an index (absolute value) into DSG & SCN for the outside connection.
DSG:o: I(NFACET)	The domain surface group index for the exposed facet. This must be a value from zero to KNSURF. If zero, it flags this facet as internal to the volume and the neighboring cell can be found in SCN.
SCN:o: I(NFACET)	The connecting neighbor in **pV3's** cell space. All the connectivity rules apply as described for SCON in pVSurface and Appendix B.

3.3.14 pVPScal

PVPSCAL(JKEY,INDEX,S)
This subroutine supplies **pV3** with scalar function values (FKEY=1) for the local nodes in the specified polyhedron.

JKEY:i: I	Key index, relative to ordering specified in pV_Init. (i.e. first key is 1, second is 2, third is 3, etc.)
INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
S:o: R(NLNODE)	Scalar function values for the local nodes.

3.3.15 pVPThres

PVPTHRES(JKEY,INDEX,XYZ,T)
This subroutine supplies **pV3** with threshold function values (FKEY=5) for the local nodes in the specified polyhedron.

JKEY:i: I	Key index
INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
XYZ:i: R(3,NLNODE)	(x, y, z)-coordinates of local grid nodes
T:o: R(NPNODE)	Threshold function values for the local nodes.

3.3.16 pVPVect

PVPVECT(JKEY,INDEX,V)
This subroutine supplies **pV3** with vector function values (FKEY=2) for the local nodes in the specified polyhedron.

JKEY:i: I	Key index
INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
V:o: R(3,NLNODE)	Vector function values (Vx, Vy, Vz) for the local node. If right-handed coordinates are desired reverse sign of the Vz values.

The next three routines are needed for the programmer-defined cutting planes.

3.3.17 pVZPrime

PVZPRIME(IDCUT,XYZ,NNODE,ZP,ZPRIME,XPC,YPC,HALFW)

This subroutine is called when the programmer-defined cutting plane is initialized, to set up the 2D data.

IDCUT:i: L	Selected cut number (1 to NPGCUT). It will be positive to request ZPRIME, XPC, YPC and HALFW.
XYZ:i: R(3,NNODE)	(x, y, z)-coordinates of grid nodes (as set in pVGrid)
NNODE:i: I	total number of nodes
ZP:o: R(NNODE)	z' values
ZPRIME:o: R	starting z' value
XPC:o: R	desired x' value at center of 2D window
YPC:o: R	desired y' value at center of 2D window
HALFW:o: R	desired half-width for square 2D window

Notes:
1) The last four variables should be set only if IDCUT > 0.
2) For IOPT=+/-1, it is assumed that ZP does not change with time.
3) ZP must be returned with the same values for all clients in multi-client applications.

3.3.18 pVXYPrime

PVXYPRIME(ZPRIME,KN,XYZ,N,XYP)

This subroutine supplies **pV3** with the (x', y') values at selected dynamic surface nodes.

ZPRIME:i: R	current z' value
KN:i: I(N)	set of 3D node indices used to construct the surface
XYZ:i: R(3,NNODE)	(x, y, z)-coordinates of grid nodes (as set in pVGrid)
N:i: I	number of selected 3D nodes
XYP:o: R(2,N)	(x', y') values at 3D nodes

Notes:
1) There is no ordering of the nodes in KN, and in fact it may contain the same node more than once.
2) pVZPrime will be called before this routine to set the cut number.

3.3.19 pVPZPrime

PVPZPRIME(ZPRIME,INDEX,XYZ,ZP)

This subroutine supplies **pV3** with the z' values at selected dynamic surface *local* nodes for polyhedra.

ZPRIME:i: R	current z' value
INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
XYZ:i: R(3,NLNODE)	(x, y, z)-coordinates of the local grid nodes (as set in pVPGrid)
ZP:o: R(NLNODE)	z' values at the local nodes

Notes:

1) pVZPrime will be called prior to the invokation of this routine to specify the cut number.

3.3.20 pVPXYPrime

PVPXYPRIME(ZPRIME,INDEX,XYZ,XYP)

This subroutine supplies **pV3** with the (x', y') values at selected dynamic surface *local* nodes for polyhedra.

ZPRIME:i: R	current z' value
INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
XYZ:i: R(3,NLNODE)	(x, y, z)-coordinates of the local grid nodes (as set in pVPGrid)
XYP:o: R(2,NLNODE)	(x', y') values at local nodes

Notes:

1) pVZPrime will be called before this routine to set the cut number.

The next three routines are needed only for the mapping of domain surfaces.

3.3.21 pVSurf

PVSURF(ISURF,XPC,YPC,HALFW)

This subroutine is called once when a mapped 2D surface is required. This will only be called for those surfaces that have been described as mapped (see pVSurface, Note 1).

ISURF:i: I	the global surface number
XPC:o: R	desired x' value at center of 2D window
YPC:o: R	desired y' value at center of 2D window
HALFW:o: R	desired half-width for square 2D window

3.3.22 pVXYSurf

PVXYSURF(KN,XYZ,N,XYP)

This subroutine supplies **pV3** with the (x', y') values at selected mapped domain surface nodes.

KN:i: I(N)	set of surface nodes indices
XYZ:i: R(3,NNODE)	(x, y, z)-coordinates of grid nodes (as set in pVGrid)
N:i: I	number of selected surface nodes
XYP:o: R(2,N)	(x', y') values at surface nodes

Notes:
1) There is no ordering of the nodes in KN, and in fact it may contain the same node more than once.
2) pVSurf will be called before this routine to set the global surface number.

3.3.23 pVPXYSurf

PVPXYSURF(INDEX,KN,XYZ,N,XYP)

This subroutine supplies **pV3** with the (x', y') values at selected *local* mapped domain surface nodes for complex polyhedra.

INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
KN:i: I(N)	set of *local* surface nodes indices
XYZ:i: R(3,NLNODE)	(x, y, z)-coordinates of grid nodes (as set in pVPGrid)
N:i: I	number of selected *local* surface nodes
XYP:o: R(2,N)	(x', y') values at *local* surface nodes

Notes:
1) There is no ordering of the nodes in KN, and in fact it may contain the same node more than once.
2) pVSurf will be called before this routine to set the global surface number.

The next four routines are needed for mapped domain surfaces and for surface integrations using the special surface functions.

3.3.24 pVSSurf

PVSSURF(JKEY,KN,XYZ,N,S)
This subroutine supplies **pV3** with surface scalar values (FKEY=3) at selected mapped domain surface nodes.

JKEY:i: I	Key index
KN:i: I(N)	set of surface nodes indices
XYZ:i: R(3,NNODE)	(x, y, z)-coordinates of grid nodes (as set in pVGrid)
N:i: I	number of selected surface nodes
S:o: R(N)	Scalar function values at surface nodes

Notes:
1) There is no ordering of the nodes in KN, and in fact it may contain the same node more than once.
2) pVSurf will be called before this routine to set the global surface number.

3.3.25 pVVSurf

PVVSURF(JKEY,KN,XYZ,N,V)
This subroutine supplies **pV3** with surface vector values (FKEY=4) at selected mapped domain surface nodes.

JKEY:i: I	Key index
KN:i: I(N)	set of surface nodes indices
XYZ:i: R(3,NNODE)	(x, y, z)-coordinates of grid nodes (as set in pVGrid)
N:i: I	number of selected surface nodes
V:o: R(3,N)	Vector function values at surface nodes

Notes:
1) There is no ordering of the nodes in KN, and in fact it may contain the same node more than once.
2) pVSurf will be called before this routine to set the global surface number.

3.3.26 pVPSSurf

PVPSSURF(JKEY,INDEX,KN,XYZ,N,S)

This subroutine supplies **pV3** with surface scalar values (FKEY=3) for the *local* polyhedral nodes at selected mapped domain surface.

JKEY:i: I	Key index
INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
KN:i: I(N)	set of *local* surface nodes indices
XYZ:i: R(3,NLNODE)	(x, y, z)-coordinates of grid nodes (as set in pVPGrid)
N:i: I	number of selected *local* surface nodes
S:o: R(N)	Scalar function values at the *local* surface nodes

Notes:
1) There is no ordering of the nodes in KN, and in fact it may contain the same node more than once.
2) pVSurf will be called before this routine to set the global surface number.

3.3.27 pVPVSurf

PVPVSURF(JKEY,INDEX,KN,XYZ,N,V)

This subroutine supplies **pV3** with surface vector values (FKEY=4) for the *local* polyhedral nodes at selected mapped domain surface.

JKEY:i: I	Key index
INDEX:i: I	The index to the polyhedral element (in the range one to KPHEDRA).
KN:i: I(N)	set of *local* surface nodes indices
XYZ:i: R(3,NLNODE)	(x, y, z)-coordinates of grid nodes (as set in pVPGrid)
N:i: I	number of selected *local* surface nodes
V:o: R(3,N)	Vector function values at the *local* surface nodes

Notes:
1) There is no ordering of the nodes in KN, and in fact it may contain the same node more than once.
2) pVSurf will be called before this routine to set the global surface number.

The following routines are used to communicate within the pV3 system.

3.3.28 pVString

PVSTRING(STRING)

This subroutine allows the programmer to provide a label for all plots, in addition to the title supplied to pV_Init. This is particularly useful for labelling plots with the time in unsteady applications.

STRING:o: C*80	character string label

3.3.29 pVCatch

PVCATCH(STRING)

This routine allows the the client to get a text string from the the running **pV3** server application. This is usefull for steering.

STRING:i: C*80	character string sent from the server

3.4 C Programming

Using **pV3** with C requires that the programmer do some things that may not be intuitive. This is because FORTRAN is supported with the same bindings.

The following rules must be followed:

- argument passing

  FORTRAN expects all arguments to be passed by reference, not value. Character variables also have their length appended to the stack (end of the call) in the order that character arguments appear in the call. These lengths are passed by value. See the CRAY & WIN32 note below.

- character strings

  FORTRAN character strings have a specified length (hence passing the length). If the string is not fully used, it is padded with blanks. The null byte that terminates a C string gets interpretted by FORTRAN as a zero character. To avoid passing the null either overpad the string with blanks, specify the length as the position before the null, or remove the null. For programmer supplied routines with character arguments the strings must be padded with blanks to the specified length.

  CRAY & WIN32 note: CRAY vector machines pack the character string length into the higher order bits of the address and do NOT place the length as an extra item in the argument list. CRAY T3Ds and WIN32 architectures put the length of the string right after the pointer to the characters. See 'pV3.h', 'osdepend.h' and 'cprism.c' in the examples directory.

- arrays

  FORTRAN array indexing, by default, starts at 1. Also in FORTRAN, the left-most index produces addresses that are consecutive in memory. Therefore, when filling multiple dimension arrays, reverse the order of the indices as documented and start at index zero.

  Note: key, node and cell numbering MUST start at 1. This gives an offset of 1 between the index and the number.

- defalut types

  The translation between FORTRAN INTEGERs and REALs and their C counter-parts depends on the FORTRAN default size. In most all cases *float* may be used for REALs (the T3E is the exception where *double* must be used). And, currently all supported ports use *int* for INTEGER.

4 Concentrator

The message-passing model of **pV3** with **PVM**, requires that the graphics workstation be accessible off the MPP with **PVM**'s 'group' functionality intact (this is a problem for both the CRAY T3D and the Intel Paragon). Also, for most installations of IBM SP2s the Ethernet interface may be the only avenue to the graphics workstation. This may require that the solver run off the high-speed interconnect (if **PVM** is also used for message-passing). Running off the switch poses significant performance problems for the solver.

In order to provide an efficient software environment for MPP platforms (which also includes SGI Arrays and DEC TurboLaser clusters) that enables continued improvement in all applications to high-end scaleability, a new module has been added to the **pV3** suite of software. This 'Concentrator' executes on the front-end of the MPP (or a node that has the proper off-machine high-speed network interface). All **pV3** messages on the machine use the high-speed interconnect via the **MPI** protocol. Communication off the MPP is performed by the Concentrator using **PVM** to the graphics workstation in a point to point manner. Off-MPP communication is improved by the concatenation of smaller messages at the Concentrator to avoid the accumulated latency encountered by many packets.

The **pV3** client-side API does not change in order to support the addition of this new software component. A different network module is required at solver link in order to use **MPI** (as well as **PVM** for the Concentrator). At execution time the Concentrator starts as one of the solver processes (this is because **MPI** currently provides no job control mechanisms). When the server starts at the graphics workstation, the Concentrator acts like a normal **pV3** client, but passes through the requests to the actual clients and transmits back the requested data. When **pV3** reports any client data (such as from the point-probe) the actual client-ID is displayed (not the ID from the Concentrator) for proper feedback to the investigator.

The use of a Concentrator also simplifies the **pV3** startup procedure. The **PVM** daemon need only be started on the processor that will run the Concentrator. If the **PVM** daemon is not running at startup, the Concentrator waits until the daemon exists.

The design allows for multiple Concentrators with a possible mixture of normal (**PVM**) clients. Great flexibility is obtained for large and complex solver systems.

NOTE: There is a different binding used for the Concentrator start, pV_MPIStart, between FORTRAN and C. This is because **MPI** does not specify a method for translating the FORTRAN (integer) handles and the C pointers and structures used. In this case the **MPI** communicator is the only object that is passed.

4.1 PV_MPISTART – FORTRAN

PV_MPISTART(COMMIN, RANK, NCL, BSIZE, COMMOUT, IERROR)
This function starts up the Concentrator and returns a new **MPI** communicator to be used by the normal clients – the Concentrator is not a part of this communicator. This should be called sometime after MPI_INIT and before and solver/client communication.

COMMIN:i: I	**MPI** input communicator handle. The communicator used to cleave off the Concentrator. Most tasks will specify MPI_COMM_WORLD.
RANK:i: I	The rank in COMMIN for the Concentrator. The process with this RANK does not return from the call and becomes the Concentrator.
NCL:i: I	The total number of **pV3** clients to be used. If this is zero, the number used is the size of COMMIN − 1. This is required for applications that are master/slave or are using some processes for other tasks.
BSIZE:i: I	The buffer size for attached buffers. The Concentrator and **MPI** clients use buffered sends. If a buffer has already be attached, specify the buffer size. If not specify zero and this call will attach a buffer big enough to hold at least one message. Look at 'NetMPI.c' to see the minimum buffer size.
COMMOUT:o: I	The resultant **MPI** communicator handle. This should be used for all solver/client communication.
IERROR:o: I	Function status: **0** - OK **-1** - BSIZE too small **-2** - RANK not in COMMINs range **-3** - Allocation Error

Implementors note. Because the **MPI** Standard does not specify anything about mixed language coding, and the Concentrator code is written in C, there needs to be a translation for the communicator. The **mpich** method was used in 'NetMPI.c'. If this is not appropriate for the version of **MPI** being used, then the FORTRAN binding for PV_MPISTART must be changed to reflect the translation.

4.2 pV_MPIStart – C

ier=pV_MPIStart(MPI_Comm in, int rank, int ncl, int bsize, MPI_Comm *out)

This function starts up the Concentrator and returns a new **MPI** communicator to be used by the normal clients – the Concentrator is not a part of this communicator. This should be called sometime after MPI_Init and before and solver/client communication.

in	**MPI** input communicator. The communicator used to cleave off the Concentrator. Most tasks will specify MPI_COMM_WORLD.
rank	The rank in 'in' for the Concentrator. The process with this 'rank' does not return from the call and becomes the Concentrator.
ncl	The total number of **pV3** clients to be used. If this is zero, the number used is the size of 'in' − 1. This is required for applications that are master/slave or are using some processes for other tasks.
bsize	The buffer size for attached buffers. The Concentrator and **MPI** clients use buffered sends. If a buffer has already be attached, specify the buffer size. If not, specify zero and this call will attach a buffer big enough to hold at least one message. 'NetMPI.c' defines the minimum buffer size.
out	The resultant **MPI** communicator. This should be used for all solver/client communication.
ier	Function status: **0** - OK **-1** - bsize too small **-2** - rank not in 'in's range **-3** - Allocation Error

4.3 pV_MPIStop

PV_MPISTOP()

This routine should be called before MPI_Finalize(). Its purpose is to inform the Concentrator that the solver/client processes are shutting down.

No Arguments

A Simple Programming Example – C

```
#include <stdio.h>
#include "mpi.h"
/* NOTE: osdepend.h must be after mpi.h */
#include "osdepend.h"

main(argc, argv)
int argc;
char *argv[];
{
  int      total, rank;
  MPI_Comm mycomm;

  MPI_Init(&argc, &argv);
  pV_MPIStart(MPI_COMM_WORLD, 0, 0, 0, &mycomm);
  MPI_Comm_size(mycomm, &total);
  MPI_Comm_rank(mycomm, &rank);
  :
  :
  pV_MPISTOP();
  MPI_Finalize();
}
```

A Simple Programming Example – FORTRAN

```
      include 'mpif.h'
      integer mycomm, total, rank, ierror

      call MPI_Init(ierror);
      call pV_MPIStart(MPI_COMM_WORLD, 0, 0, 0, mycomm, ierror);
      call MPI_Comm_size(mycomm, total, ierror);
      call MPI_Comm_rank(mycomm, rank, ierror);
      :
      :
      call pV_MPIStop();
      call MPI_Finalize();
      stop
      end
```

A More Complex Programming Example – FORTRAN

This example is from a master/slave initialization that does buffered sends. The Concentrator is rank 0 of MPI_COMM_WORLD.

```
      parameter (LENBUF = 500000)
      include 'mpif.h'
      integer mycomm, total, rank, ierr, nclients
      common /mpicomm/ mycomm
      dimension send_buffer(LENBUF)
c
      call MPI_Init(ierr)
      call MPI_Buffer_attach(send_buffer, 4*LENBUF, ierr)
      call MPI_Comm_size(MPI_COMM_WORLD, nclients, ierr)
c
c     remove the master and the concentrator
c
      nclients = nclients - 2
      call pV_MPIStart(MPI_COMM_WORLD, 0, nclients, 4*LENBUF,
     &                 mycomm, ierr)
c
      call MPI_Comm_size(mycomm, total, ierr)
      call MPI_Comm_rank(mycomm, rank, ierr)
      :
      :
```

4.4 Building Processes that can be Concentrators

MPI does not specify how processes are started, and some implementations of **MPI** only allow a single executable to be run on the processors. These facts forced the design so that any process can run as the Concentrator. It is best to first get your application running using **MPI** for message passing coupled with **pV3** in the standard manner. Once this is accomplished, if you are using MPI_COMM_WORLD as the communicator, your code must be changed. After calling MPI_Init, duplicate the communicator via a call to MPI_Comm_dup. Use only this communicator in your code as you were using MPI_COMM_WORLD.

Now follow these instructions:

- Modify your source

  The call to MPI_Comm_dup should be replaced with pV_MPIStart as documented above. You need to figure out which process (rank) you want as the Concentrator. This requires some knowledge of how your version of **MPI** starts tasks and which processor is the best candidate (has the fastest off-machine network interface).

  Insert a call to pV_MPIStop before the call to MPI_Finalize.

- Compile the **pV3** network interface for **MPI**

  The routine 'NetMPI.c' must be compiled for the architecture you intend to use. This file needs to be compiled with the same options as used in the compilation of 'Network.c' (both of these sources can be found in the 'clients' directory of the distribution). Look at 'Makefile' in the appropriate 'clients' subdirectory. Another "-Ipath" argument should be added so that the include for 'mpi.h' can be found.

  Do not rebuild the library.

- Build your application

  Use the same invokation of the loader as you did for the standard **pV3** build. The change you make is to include 'NetMPI.o' on the build line before the libraries are searched. This will use 'NetMPI.o' instead of 'Network.o' to resolve all **pV3** message passing routines.

- Running the application

  Specify one more processor via the **MPI** run command when starting. After the call to pV_MPIStart, the size of the returned communicator will be the correct number for the application.

  If you have a FORTRAN application and get a communicator error message from **MPI** please refer to the *implementors note* in the description of PV_MPISTART.

5 Portability

5.1 FORTRAN Programming

For the most part FORTRAN source that runs on one implementation of **pV3** will work on others.

5.2 C Programming

Unfortunately **pV3** is not source code compatible for C across all machines. This has to do with supporting FORTRAN.

The C programmer that wishes to have their **pV3** application running on many platforms should be aware that:

- main program

  On IBM and HP workstations as well as CRAYs, the normal C conventions apply. For DEC, SGI and WIN32 ports the name of the main program must be 'MAIN__', and on all others the name is 'MAIN_'.

- routine names

  For IBM and HP ports, all **pV3** entry points are the FORTRAN names in lower-case. On all other platforms except CRAY and WIN32, external entries are lower-case with an underscore ('_') appended to the end. CRAY and WIN32 entry points are upper-case with no appended underscores. WIN32 routine names must also be prefixed with "__stdcall" so that the correct Microsoft name handling is done.

  Note: this is NOT true for pV_MPIStart. This routine is only accessible from C and has the actual name 'pV_MPIStart'.

- integers

  pV3 uses the default FORTRAN INTEGER for all integer arguments. This always corresponds to an *int*.

- floats

  pV3 uses the default FORTRAN REAL for all floating point arguments. This almost always corresponds to a *float*. The exception is the CRAY T3E where the REAL corresponds to a *double*.

See the file 'pV3.h' or 'osdepend.h' in the examples subdirectory of the distribution for a method to avoid these problems.

A Error Codes

Any error generated at the call to pV_Init invalidates the client for inclusion in the visualization.

If any of these errors are generated during a Structure Unsteady visualization, that volume is invalid for the current time step. The user will see no data coming from this client. The structure is checked again at the next call to pV_Update.

The following codes report a more detailed message to standard output:

-1	Maximum Number of Surface Faces Exceeded! The size for the temporary face structure was too small and could not be increased by re-allocation.
-2	Degenerate face! A degenerate face was found. If the cell is really degenerate (has the same node numbers in multiple entries) use the appropriate cell type!
-3	Face Hit by 3 (or more) Cells/Surfaces! The is part of more than 2 objects. Any face must be touched by either 2 cells (an internal face) or a cell and a surface face. If you are trying to put a *domain surface* where **pV3** thinks the face is internal, change the node numbering on one of the cells to indicate a new face. This will probably require that additional nodes be added to the node space.
-4	Surface Face with no Connecting Cell! A face has been specified that is not part of a cell in this volume.
-5	Maximum Number of Surface Faces Exceeded while constructing the surface 'Others'! This can occur if the number of cells is small compared to the number of surface faces. To increase the storage, give KSURF a larger value so that the face matching procedure can complete.
-6	Memory Allocation Error!
-7	Structure Check error. A more detailed list is given on standard output.
-8	Edge Table Overflow! The size for the temporary edge structure was too small and could not be increased by re-allocation.
-9	Maximum Number of Edge Lines Exceeded!

The following codes generate no additional information:

-102	The client library was not built properly! You should not see this error.
-101	Some PVM error was encountered during initialization! This happens when either the PVM daemon is not running, or the PVM group subsystem cannot be initialized.
-100	The pV3 client system is already initialized!
1	Control Parameter Out of Range! IOPT is greater than 2 or less than -3.
2	NKEYS is zero!
3	NKEYS is greater that the maximum (Obsolete)!
4	FKEYS(i) out of range! An entry in FKEYS is less than zero or greater than five. Also, 2 state-vectors (zeros) or a large state-vector rank will cause this error.
5	FLIMS(1,i) equals FLIMS(2,i) $->$ FKEYS(i) = 3 or FLIMS(1,i) = 0.0 $->$ FKEYS(i) = 4!
6	Memory Allocation Error! **pV3** has requested a block of memory and has been refused. This is usually do to the problem's size.
7	Degenerate Block! One of the block sizes is less than two.
8	KPTET is less than zero!
9	NPTET(1,MAX) <> KPTET! The last entry in NPTET does not match the value KPTET.
10	KNODE <> 0 but no non-structured block cells!
11	NSURF(1,KNSURF) > KSURF! The last entry in NSURF is larger than the size given at initialization (or from the call to pVStruc).
12	Structure Unsteady - no change flag set without any prior definition!
13	Structure Unsteady - a reallocation error has occured.
14	Structure Unsteady - current number of blocks (KNBLOCK) is greater than specified at pV_Init.
15	NSURF(3,n) > 3599 or periodicity specified with MIRROR >= 0.

B Multi-client Connectivity Options

There are a limitted number of segments allowed for streamlines. When that limit is reached (currently 4 times the number of clients) a warning is displayed and that streamline will terminate.

If the interface between clients is ragged (such as found with tetrahedral meshes), a command to enter the domain (cell number = -1) may fail.

An attempt to (re)enter (cell number = -1) a client is **MUCH** more compute intensive than specifying the cell number (or even a cell that is close to the target)!

B.1 pVConnect

When the programmer is responsible for the connectivity (any negative IOPT in pV_Init) the routine pVConnect must be supplied. It is called during particle integrations to move data from one cell to the next. There are many options based on where the the target cell resides.

- IDTIN = -1

  The integration continues in the current client.

KCIN	Comment
negative	try to re-enter in this client
	value must be index to SCEL of exitting face (negated)
zero	stop - hit boundary
positive	continue to this cell number

- IDTIN = 0

  Try to enter into all other clients. KCIN is ignored unless zero.

- IDTIN = positive number

  Continue the integration into client with this client-id.

KCIN	Comment
-1	try to enter in the client specified by IDTIN
zero	stop
positive	continue to this cell in IDTIN

B.2 pVSurface

The programmer can also specified the multi-client topology by the data returned from the routine pVSurface. In this case *internal surfaces* must be constructed to patch regions together. The surfaces define the connectivity with the following options:

- NSURF(3,n) = negative number

  The absolute value of the number is the client-id to continue the integration.

SCON(i)	Comment
-1	try to enter in the client specified by -NSURF(3,n)
zero	stop - hit boundary
positive	continue to this cell in -NSURF(3,n)

- NSURF(3,n) = 0

  Special internal interface.

SCON(i)	Comment
-2	try to enter all clients (including locally)
-1	try to enter all other clients
zero	stop
positive	continue in this cell locally (usefull in closing C and O grids as well as connecting non-polyhedral regions with polyhedra)

- NSURF(3,n) = positive number

  The integration continues in the current client.

SCON(i)	Comment
-1	try to re-enter in this client
zero	stop
positive	connect to this cell locally

- NSURF(3,n) = positive number > 10000

  global surface number = NSURF(3,n) − 10000

  The integration continues in the current client through a periodic boundary by applying the replication matrix once.

SCON(i)	Comment
-1	try to re-enter in this client
zero	stop
positive	connect to this cell locally

- NSURF(3,n) = positive number > 20000
  global surface number = NSURF(3,n) − 20000

  The integration continues in the current client through a periodic boundary by applying the replication matrix *nrep* (or −1) times.

SCON(i)	Comment
-1	try to re-enter in this client
zero	stop
positive	connect to this cell locally

- NSURF(3,n) = positive number > 30000
  global surface number = NSURF(3,n) − 30000

  This grouping defines a *bucket-of-faces*. No connectivity is assumed – SCON is not used. The fecets need not match the faces of a cell, they can be defined via any node in the node space.