SAND REPORT

SAND2008-1933
Unlimited Release
Printed April 2008

Pamgen, a Library for Parallel Generation of Simple Finite Element Meshes

David M. Hensinger, Richard R. Drake, James G. Foucar, Thomas A. Gardiner

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2008-1933
Unlimited Release
Printed April 2008

Pamgen, a Library for Parallel Generation of Simple Finite Element Meshes

D. M. Hensinger, R. R. Drake, J. G. Foucar, T. A. Gardiner
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0378

Abstract

Generating finite-element meshes is a serious bottleneck for large parallel simulations. When mesh generation is limited to serial machines and element counts approach a billion, this bottleneck becomes a roadblock. PAMGEN is a parallel mesh generation library that allows on-the-fly scalable generation of hexahedral and quadrilateral finite element meshes for several simple geometries. It has been used to generate more that 1.1 billion elements on 17,576 processors.

PAMGEN generates an unstructured finite element mesh on each processor at the start of a simulation. The mesh is specified by commands passed to the library as a "C"-programming language string. The resulting mesh geometry, topology, and communication information can then be queried through an API. PAMGEN allows specification of boundary condition application regions using sidesets (element faces) and nodesets (collections of nodes). It supports several simple geometry types. It has multiple alternatives for mesh grading. It has several alternatives for the initial domain decompositon. PAMGEN makes it easy to change details of the finite element mesh and is very usesful for performance studies and scoping calculations.

Acknowledgment

The PAMGEN capabilities would not have been developed without the support and inspiration of the ALEGRA community of users and developers. Within the users community, Chris Garasi and his challenging analysis requirements were the spur for development of the most complex and rewarding capabilities. Tom Brunner severely stressed the code with his early testing on large platforms. Within the developers community, the build and benchmark utilities written by Richard Drake enabled continual improvement in functionality and robustness. Tom Gardiner provided the code that is the heart of the default load balance capability.

Contents

1 Introduction **11**

2 Capabilities **12**

2.1 Dimensions 12

2.2 Mesh Geometries 12

2.3 Boundary Conditions 13

2.4 Decomposition 13

2.5 Geometry Transformation 13

2.6 Element Density 14

3 Approach **15**

4 Usage **16**

5 Specifying a Mesh **17**

5.1 Dimensionality 17

5.2 Block IDs 18

5.3 Geometry and Topology 19

5.3.1 Rectilinear 19

5.3.2 Spherical 21

5.3.3 Cylindrical 23

5.3.4 Radial and Radial Trisection 25

5.3.5 Brick 32

5.4 Boundary Conditions (Nodesets and Sidesets) 34

5.5 User Defined Geometry Transformation 37
5.6 User Defined Element Density 40
5.7 Decomposition Strategy 42

6 Library Interface **46**

6.1 Creating a Mesh 46
6.1.1 Create_Pamgen_Mesh 46
6.1.2 getPamgenEchoStreamSize 47
6.1.3 getPamgenEchoStream 47
6.1.4 getPamgenErrorStreamSize 47
6.1.5 getPamgenErrorStream 47
6.1.6 getPamgenWarningStreamSize 48
6.1.7 getPamgenWarningStream 48
6.1.8 getPamgenInfoStreamSize 48
6.1.9 getPamgenInfoStream 48
6.2 Querying a Mesh 50
6.2.1 im_ex_get_init 50
6.2.2 im_ex_inquire 51
6.2.3 im_ex_get_coord 52
6.2.4 im_ex_get_coord_names 52
6.2.5 im_ex_get_map 53
6.2.6 im_ex_get_elem_num_map 53
6.2.7 im_ex_get_node_num_map 54
6.2.8 im_ex_get_elem_blk_ids 54

6.2.9 im_ex_get_elem_block 54
6.2.10 im_ex_get_elem_conn 55
6.2.11 im_ex_get_node_set_ids 56
6.2.12 im_ex_get_node_set_param 57
6.2.13 im_ex_get_node_set 57
6.2.14 im_ex_get_side_set_ids 58
6.2.15 im_ex_get_side_set_param 58
6.2.16 im_ex_get_side_set 59
6.2.17 im_ex_get_qa 59
6.2.18 im_ex_get_info 60
6.2.19 im_ne_get_init_global 61
6.2.20 im_ne_get_init_info 62
6.2.21 im_ne_get_eb_info_global 62
6.2.22 im_ne_get_ns_param_global 63
6.2.23 im_ne_get_ss_param_global 63
6.2.24 im_ne_get_loadbal_param 64
6.2.25 im_ne_get_elem_map 65
6.2.26 im_ne_get_node_map 66
6.2.27 im_ne_get_cmap_params 66
6.2.28 im_ne_get_node_cmap 67
6.2.29 im_ne_get_elem_cmap 68
6.3 Deleting a Mesh 69
6.3.1 Delete_Pamgen_Mesh 69

A Runtime Compiler Functionality **70**

A.1 The RTC language 70

A.1.1 Operators 70

A.1.2 Control flow 71

A.1.3 Line Structure 72

A.1.4 Variables 72

A.1.5 Math 73

A.1.6 Strings 74

A.1.7 Printf 74

A.1.8 Comments 75

A.1.9 Unsupported Features 75

B read_mesh_to_memory **76**

References **84**

Index **86**

List of Figures

1 The mesh geometry zoo in 3D. 12

2 Examples of geometry transformation. 13

3 Definition of a three-dimensional `RECTILINEAR` mesh. 20

4 Definition of a two-dimensional spherical mesh. 22

5 Definition of a three-dimensional spherical mesh. 22

6 Three dimensional `CYLINDRICAL` mesh. 24

7 Three dimensional `RADIAL` mesh. 27

8 A 360 degree `RADIAL` mesh. 28

9 A `RADIAL TRISECTION` mesh with three trisection blocks. . . . 29

10 A `RADIAL TRISECTION` mesh with three trisection blocks. . . . 30

11 A 360 degree `RADIAL TRISECTION` mesh. 31

12 Three dimensional `BRICK` mesh. 33

13 Block topology and labels. 35

14 A `RADIAL TRISECTION` mesh with sidesets. 36

15 3D Geometry Transformation Example 38

16 2D Geometry Transformation Example 39

17 2D mesh created with a `USER DEFINED ELEMENT DENSITY`. . . 41

18 `BISECTION` decomposition run on 7 processors. 43

19 `PROCESSOR LAYOUT` decomposition run on 8 processors. 44

20 `PROCESSOR LAYOUT` decomposition on 8 processors. 44

List of Tables

1 Keywords for `RECTILINEAR -- END`. 19

2 Keywords for `SPHERICAL -- END`. 21

3 Keywords for `CYLINDRICAL -- END`. 23

4 Keywords for `DECOMPOSITION STRATEGY -- END`. 45

1 Introduction

To overcome the challenge of producing multi-million finite element meshes for simulations using more than 1000 processors a library has been developed (PAMGEN) that for several simple geometries produces each processor's mesh as an early step of the analysis execution. The specification for these meshes is provided by a block of terse instructions that may be placed in an input file. These instructions are passed to the library as a "C"-programming language character array. PAMGEN is also referred to as an "in line" mesh generator because the meshing instructions may be included in one of the analysis input decks.

The simple input format allows analysts to change the resolution of a simulation by altering a few parameters. It also allows them to execute their simulations on different numbers of processors without requiring any pre-processing.

The mesh generation proceeds through steps of decomposition, local element creation, and communication information generation. The final product of the library is a data structure that can be queried using an API (Application Programming Interface) that is based on the NEMESIS and EXODUS APIs. Currently the library is limited to generating meshes of domains with cylindrical, tubular, and block shapes. Substantial control is allowed over the element density within these shapes. Boundary condition application regions can be specified on the surfaces and interior of the mesh .

Development of this capability revealed that the parallel mesh generation process can be reduced to answering a series of questions: What elements are on this processor? What nodes are on this processor? What is the connectivity of this element? What elements border this element? What processor does this element reside on?... Resolving these questions inductively, without resolution to communication, is essential for preserving scalability. Once a framework for posing and answering these questions for a particular geometry is established, expanding the capability to support additional geometries is straightforward.

2 Capabilities

The capabilities of PAMGEN are best understood by studying Section 5, which documents in detail the instructions available for specifying a mesh. This section provides a brief overview of the library's capabilities. PAMGEN will be distributed as part of the TRILINOS package of matrix and finite element tools.

2.1 Dimensions

PAMGEN can create both two and three dimensional meshes. It creates quadrilateral finite element meshes if two dimensions are specified, and it creates hexahedral finite element meshes if three dimensions are specified. For two dimensional quadrilateral meshes the Z component of nodal coordinates is not supplied.

2.2 Mesh Geometries

The PAMGEN library handles the mesh geometries shown below:

- Bricks
- Partial hollow cylinders
- Complete hollow cylinders
- Partial solid cylinders
- Complete solid cylinders

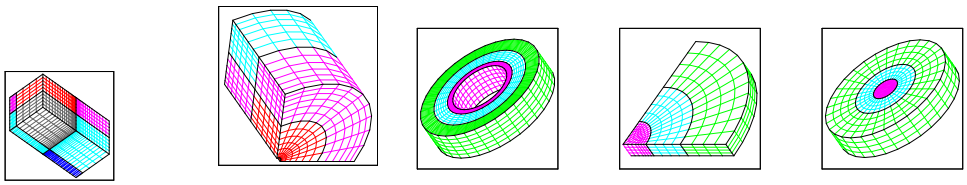


Figure 1: The mesh geometry zoo in 3D.

2.3 Boundary Conditions

Boundary conditions application regions in the form of node sets and side sets can be applied to the face, edge, or corner of any element block in the finite element mesh. They may also be applied to any face, edge, or corner of the entire mesh.

2.4 Decomposition

There are several mesh decomposition strategies available in PAMGEN:

- A default decomposition based on a constrained optimized solution that slices through the entire mesh in its three (or two in 2D) topological dimensions.
- A user defined slicing strategy that specifies the number of slices through the mesh in each topological direction.
- A sequential strategy that distributes elements beetween processors based on their element ids with the first n elements going to processor 0 ...
- A random strategy that assigns element to processors randomly.

2.5 Geometry Transformation

Any mesh can be modified by re-evaluating the nodal coordinate values using a user-supplied function. This function has the original nodal coordinates as input values. Its output values define the new nodal coordinates.

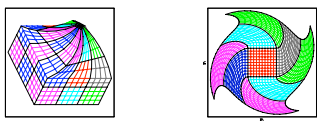


Figure 2: Examples of geometry transformation.

2.6 Element Density

Several of the geometry types allow specification of first and last element sizes within a block in a particular cartesian direction. In addition the user may control node distribution over any geometry by providing a user defined distribution function. This function is evaluated so that nodes are shifted towards areas where the function has its highest values.

3 Approach

PAMGEN operates on the premise that mesh generation is deterministic. This means that every execution of code compiled with compiler A and run on processor B under operating system C reading mesh instructions file D will produce an identical mesh. The mesh will have the same topology and the nodes will have the same coordinate locations. On a multi-processor machine of identical nodes an arbitrary number of processors executing identical instructions produce an identical mesh. Use of PAMGEN on heterogeneous machines may produce unexpected results.

With one significant exception, PAMGEN operates in the same way as these identical processors producing identical mesh. The exception is that each processor only allocates for a subset of the nodes and elements, and each processor performs topology and geometry calculations only for the entities and dependencies present on that processor. PAMGEN exploits the fact that each processor is capable of producing the entire mesh in order to allow each processor to produce its own mesh. The deterministic nature of the meshing process is essential to allow correspondence in topologies and geometries produced on adjacent processors.

The implementation of the mesh generation in PAMGEN is as much as possible implicit. Quantities are not allocated until they are ready for output, and they are not calculated until they can be stored, or until they are required by a dependent calculation. This approach can result in duplication of intermediate calculations, but it avoids the severe limitations on total problem size that occur if attempts are made to allocate any quantity with a size related to the total mesh size.

4 Usage

For the few simple geometries and element types it supports, the PAMGEN library is a substitute for pre-processed finite element mesh files. Successful usage of the library requires some modification of the analysis code.

The PAMGEN library must be linked into the analysis executable to allow access to the mesh creation, query and deletion functions.

The modules in the analysis code that read finite element mesh data from a file must be adapted to:

- Read in a "C"-progamming language string that specifices the geometry, topology, and boundary conditions of a mesh.
- Pass that string to the PAMGEN Create_Pamgen_Mesh(...) function along with the rank of the mesh requested and the total number of processors across which the mesh is spread.
- Handle message and possibly error strings available after calling Create_Pamgen_Mesh(...).
- Call the PAMGEN query functions to populate the analysis code's mesh and communication data structures.
- Call the PAMGEN Delete_Pamgen_Mesh() function release memory allocated within the library.

The source code for a stand-alone executable called "pamgen_lt_c" is distributed with the PAMGEN library. This example code is an excellent starting point for adapting an analysis code to use the PAMGEN library.

5 Specifying a Mesh

The "C"-programming language string passed to PAMGEN is the complete definition of the mesh's geometry, topology, node sets, side sets, and parallel decomposition. It must begin with a `MESH` keyword, and it must end with an `END` keyword. The `MESH -- END` keyword pair must surround a `RECTILINEAR -- END`, `SPHERICAL -- END`, `BRICK -- END`, `RADIAL -- END`, `RADIAL TRISECTION -- END`, or `CYLINDRICAL -- END` keyword pair and may have additional `SET ASSIGN -- END`, `DECOMPOSITION STRATEGY -- END`, `USER DEFINED ELEMENT DENSITY -- END`, or `USER DEFINED GEOMETRY TRANSFORMATION -- END` keyword pairs.

```
MESH
 {RECTILINEAR | SPHERICAL | BRICK | RADIAL | RADIAL TRISECTION | CYLINDRICAL}
    [subkeyword-list]
   END
  [SET ASSIGN]
  [END]
  [DECOMPOSITION STRATEGY]
  [END]
  [USER DEFINED GEOMETRY TRANSFORMATION]
  [END]
  [USER DEFINED ELEMENT DENSITY]
  [END]
END
```

5.1 Dimensionality

The dimensionality of the mesh is not specified in the "C" programming language string. It is passed to the PAMGEN library at execution time through the PAMGEN API. In general a 2D mesh can be specified by removing 3D specific keywords and values (those referencing a third coordinate [for example Z or k]) from a 3D mesh description.

5.2 Block IDs

The finite elements created using PAMGEN are grouped into blocks. Each block has a positive non-zero id. These ids are automatically assigned by PAMGEN and are not under the control of the user. If there is a single block then its id is 1. When there are more than one block, the ids are assigned beginning with the block in the lowest topological position in i, j, k space. Subsequent blocks are incrementally numbered first in the i topological direction, next in the j topological direction, and finally in the k topological direction. In the case of `BRICK` and `RECTILINEAR` meshes i, j, and k correspond to the coordinate directions x, y, and z. In the case of `CYLINDRICAL`, `RADIAL`, and `RADIAL TRISECTION` meshes i, j, and k correspond to r, θ, and z.

5.3 Geometry and Topology

5.3.1 Rectilinear

```
RECTILINEAR
   [subkeyword-list]
END
```

The `RECTILINEAR -- END` block pair surrounds the description of the geometry of a rectilinear mesh. The extent of the domain is given by a pair of vectors (gmin and gmax). The number of blocks in each coordinate direction and the number of elements in each block are given by additional keywords. The total number of elements specified in this type of mesh is the product of the total number of blocks `BX` $\times$ `BY` $\times$ `BZ` and the total number of elements per block `NX` $\times$ `NY` $\times$ `NZ`. For a 2D mesh `NZ` and `BZ` must be omitted. The keywords associated with the `RECTILINEAR` keyword are given in Table 1.

Table 1: Keywords for `RECTILINEAR -- END`.

Sub-Keyword	Input	Description
`NX`	`int`	Number of cells in x-direction.
`NY`	`int`	Number of cells in y-direction.
`NZ`	`int`	Number of cells in z-direction.
`BX`	`int`	Number of blocks in the x-direction.
`BY`	`int`	Number of blocks in the y-direction.
`BZ`	`int`	Number of blocks in the z-direction.
`GMIN`	`vector`	Minimum domain coordinates (x,y,z).
`GMAX`	`vector`	Maximum domain coordinates (x,y,z).

An example of a mesh specification with the `RECTILINEAR` option is illustrated in Figure 3.

```
mesh
  rectilinear
    nx = 10
    ny = 10
    nz = 10
    bx = 4
    by = 7
    bz = 5
    gmin = 1.0 1.0 1.0
    gmax = 4.0 7.0 5.0
  end
  set assign
    nodeset,ihi,2
    nodeset,jhi,1
  end
end
```

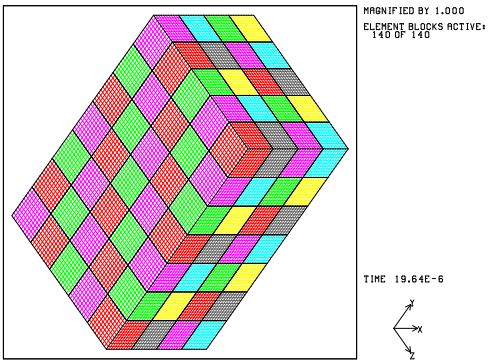


Figure 3: Definition of a three-dimensional `RECTILINEAR` mesh.

5.3.2 Spherical

```
SPHERICAL
   [subkeyword-list]
END
```

The `SPHERICAL -- END` block pair allows the description of a curvilinear spherical mesh centered at the origin and described by an inner and outer radius, and the extent of revolution in θ and ϕ directions. Angle θ is measured counter-clockwise about the z-axis from the x-axis, and angle ϕ is measured counter-clockwise from the y-axis about the x-axis. The number of elements and blocks in each curvilinear coordinate direction are specified by the keywords in Table 2. The parameters `PHI, NPHI, and BPHI` are only appropriate for 3D problems. When used in 2D simulations, `CYLINDRICAL` and `SPHERICAL` keywords produce identical meshes.

Table 2: Keywords for `SPHERICAL -- END`.

Sub-Keyword	Input	Description
`NR`	`int`	Number of cells in r-direction.
`NTHETA`	`int`	Number of cells in θ-direction.
`NPHI`	`int`	Number of cells in ϕ-direction.
`BR`	`int`	Number of blocks in the r-direction.
`BTHETA`	`int`	Number of blocks in θ-direction.
`BPHI`	`int`	Number of blocks in ϕ-direction.
`RI`	`real`	Inner radius
`RO`	`real`	Outer radius
`THETA`	`real`	Angular extent in θ (degrees,0.-180. in 3D, 0.-360. in 2D)
`PHI`	`real`	Angular extent in ϕ (0.-360)

An example of a mesh definition syntax with the `SPHERICAL` option is illustrated in Figure 4 for a 2D simulation. An example of 3D spherical mesh generation follows in Figure 5.

```
mesh
  spherical
    ri = 0.0
    ro = 0.5
    theta = 45
    ntheta = 10
    nr = 20
    br = 2
    btheta = 2
  end
end
```

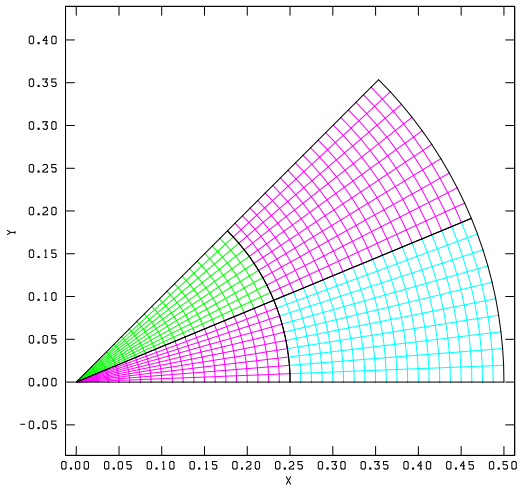


Figure 4: Definition of a two-dimensional spherical mesh.

```
mesh
  spherical
    ri = 0.5
    ro = 1.0
    theta = 180.0
    ntheta = 10
    nr = 10
    br = 2
    btheta = 2
    bphi = 2
    nphi = 10
    phi = 90
  end
end
```

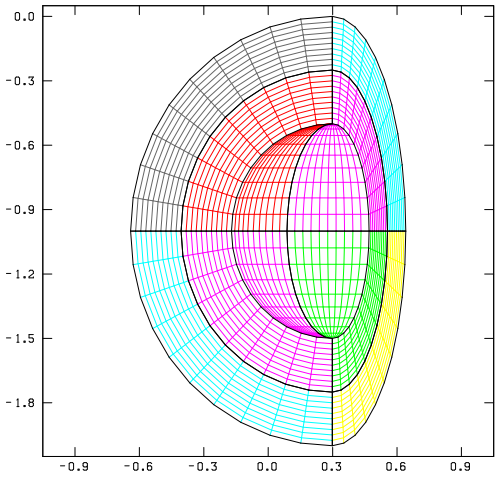


Figure 5: Definition of a three-dimensional spherical mesh.

5.3.3 Cylindrical

```
CYLINDRICAL
   [subkeyword-list]
END
```

The `CYLINDRICAL -- END` block pair allows the description of a curvilinear cylindrical mesh centered at the origin in x and y, and aligned along the z-axis. It is described by an inner and outer radius, the extent of revolution in angle θ about the z-axis, and its start and end in the z-direction. Angle θ is measured counter-clockwise about the z-axis from the x-axis. The number of elements and blocks in each indicial direction are specified by the keywords in Table 3. The parameters `ZMIN, ZMAX, NZ, and BZ` are only appropriate for 3D problems. When used in 2D solutions, `CYLINDRICAL` and `SPHERICAL` produce identical meshes.

Table 3: Keywords for `CYLINDRICAL -- END`.

Sub-Keyword	Input	Description
`NR`	`int`	Number of cells in r-direction.
`NTHETA`	`int`	Number of cells in θ-direction.
`NZ`	`int`	Number of cells in z-direction.
`BR`	`int`	Number of blocks in the r-direction.
`BTHETA`	`int`	Number of blocks in θ-direction.
`BZ`	`int`	Number of blocks in z-direction.
`RI`	`real`	Inner radius
`RO`	`real`	Outer radius
`THETA`	`real`	Angular extent in θ degrees, 0.-360.
`ZMIN`	`real`	Start of mesh in z-direction
`ZMAX`	`real`	End of mesh in z-direction

An example of a PAMGEN mesh definition with the `CYLINDRICAL` option is illustrated in Figure 6.

```
mesh
  cylindrical
    ri = 0.5
    ro = 1.0
    theta = 90.0
    ntheta = 10
    nr = 10
    br = 2
    zmin = 1.0
    zmax = 2.0
    nz = 10
    bz = 2
  end
end
```

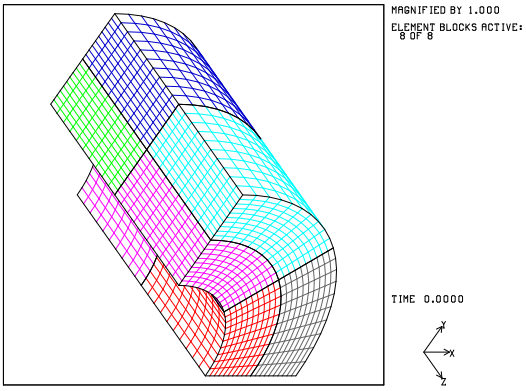


Figure 6: Three dimensional `CYLINDRICAL` mesh.

5.3.4 Radial and Radial Trisection

```
{RADIAL | RADIAL TRISECTION }
   [ENFORCE PERIODICITY]
   [TRISECTION BLOCKS, int]
   [TRANSITION RADIUS, int]
   [ZMIN real]
   NUMZ int
     ZBLOCK int real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
   {NUMR | NUMX} int INITIAL RADIUS real
     RBLOCK int real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
   {NUMA | NUMY} int
     ABLOCK int real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
END
```

The `RADIAL` and `RADIAL TRISECTION` block pairs allow the description of a curvilinear cylindrical mesh centered at the origin in x and y, and aligned along the z-axis. The `RADIAL TRISECTION -- END` block pair fills in the center of the cylindrical mesh with transition elements. These options are similar to the `CYLINDRICAL` but they have a different set of controls on element distribution. The successful creation of these meshes requires sequential specification of the information for the number of elements blocks and their sizes in each coordinate direction.

The `ENFORCE PERIODICITY` keyword applies only to `RADIAL` and `RADIAL TRISECTION` mesh descriptions that meet certain requirements:

- The meshes must have azimuthal angles of 90, 180, or 360 degrees.
- The meshes must have a single block of elements in the azimuthal direction.
- The meshes must have an even number of elements in each 90 degree segment of the mesh.
- For `RADIAL TRISECTION` meshes, one transition zone is required for each 90 degrees of azimuth.

This keyword causes the mesh generation to perform all node coordinate calculations in the first 45 degrees of the azimuthal domain. Coordinate

locations of nodes ouside of 45 degrees are formed by permuting the sign and oder of the components of periodically corresponding nodes' locations. This guarantees that there will be no differences between the absolute floating point values of periodically corresponding nodes' coordinates.

The `NUMZ | NUMR | NUMA` keywords are followed by an integer specifying the number of element blocks in that coordinate direction. The `NUMR` line includes an additional real parameter that specifies the inner radius of the cylindrical mesh. Specification of an inner radius of 0.0 will result in degenerate elements with co-located nodes along the `Z` axis. The innner radius specification is ignored for `RADIAL TRISECTION` meshes.

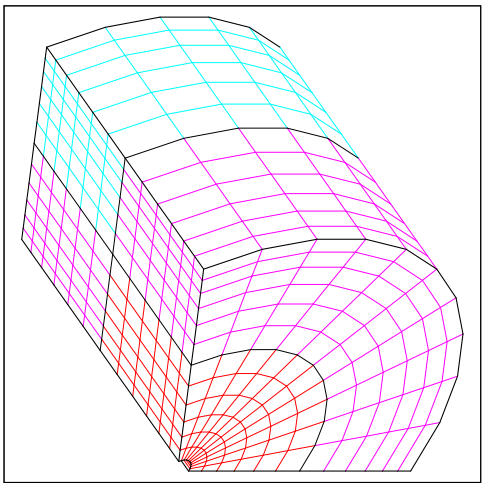
Immediately following the `NUMZ int | NUMR int INITIAL RADIUS real | NUMA int` lines, there must be a `ZBLOCK | RBLOCK | ABLOCK` line for each of the blocks in that direction. These lines specify the spatial extent of the particular block and the distribution of elements in the block in that direction. The `INTERVAL int` keyword specifies a fixed number of elements in the block. A `FIRST SIZE real LAST SIZE real` pair specifies the absolute size of the first and last elements in the block. When the `FIRST SIZE real LAST SIZE real` specification is used, the element sizing will be linear between the first and last elements. The sizes of the first and last elements may be adjusted slightly to provide linear sizing. The approximate number of elements that will be generated using `FIRST SIZE` and `LAST SIZE` sizing controls is given by truncating to an integer, the length of the mesh segment divided by the average of the `FIRST SIZE` and `LAST SIZE` values. There is a slight chance that this calculation will result in a different number of elements on different computer platforms. Omission of the `LAST real` keyword is equivalent to setting the element size to that given by the `FIRST SIZE real` keyword.

The `RADIAL TRISECTION -- END` block pair requires the additional input of `TRISECTION BLOCKS, int` and it accepts the optional `TRANSITION RADIUS, real` keyword value pair. The `TRISECTION BLOCKS` keyword specifies the number of transition zones that will be used in the central region of the mesh. This number must be supplied. One transition zone is recommended for each 90 degrees of azimuth. The `TRANSITION RADIUS` is the distance from the origin to the corners of the transition zones of mesh. Without this parameter the transition radius is chosen to be one half of the radial thickness of the first block. When `RADIAL TRISECTION` is selected, any `INITIAL RADIUS` supplied is ignored.

The `ZMIN real` keyword value pair is available to specify an offset for the entire mesh in the `Z` direction.

If the cumulative values of the sizes of the azimuthal blocks, given by the second argument of `ABLOCK int real` is equal to 360.0 degrees, then the mesh will form an single closed ring of elements.

An example of PAMGEN mesh definition with the `RADIAL` option is illustrated in Figure 7.



```
mesh
  radial
    numz 2
      zblock 1 10.0 first size 1 last size 2
      zblock 2 10.0 first size 2 last size 1
    numr 2 initial radius 1.
      rblock 1 10. first size 1. last size 2.
      rblock 2 10. first size 2. last size 1.
    numa 1
      ablock 1 120. interval 10
  end
end
```

Figure 7: Three dimensional `RADIAL` mesh.

A second example of PAMGEN mesh definition with the `RADIAL` option is

illustrated in Figure 8. In this case the sum of the azimuthal blocks is 360.0 and the mesh is a complete cylinder.

```
mesh
  radial
  numz 1
    zblock 1 10.0 interval 6
  numr 3
    rblock 1 2. interval 12
    rblock 2 5. interval 6
    rblock 3 5. interval 12
  numa 1
    ablock 1 360. interval 36
 end
end
```

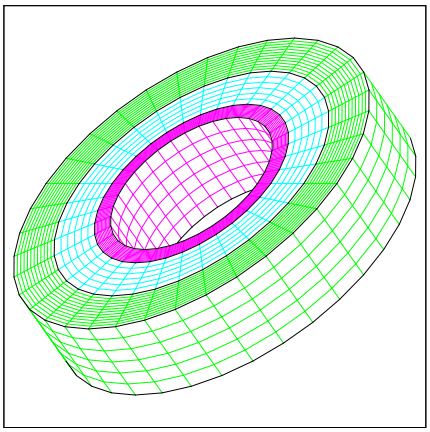
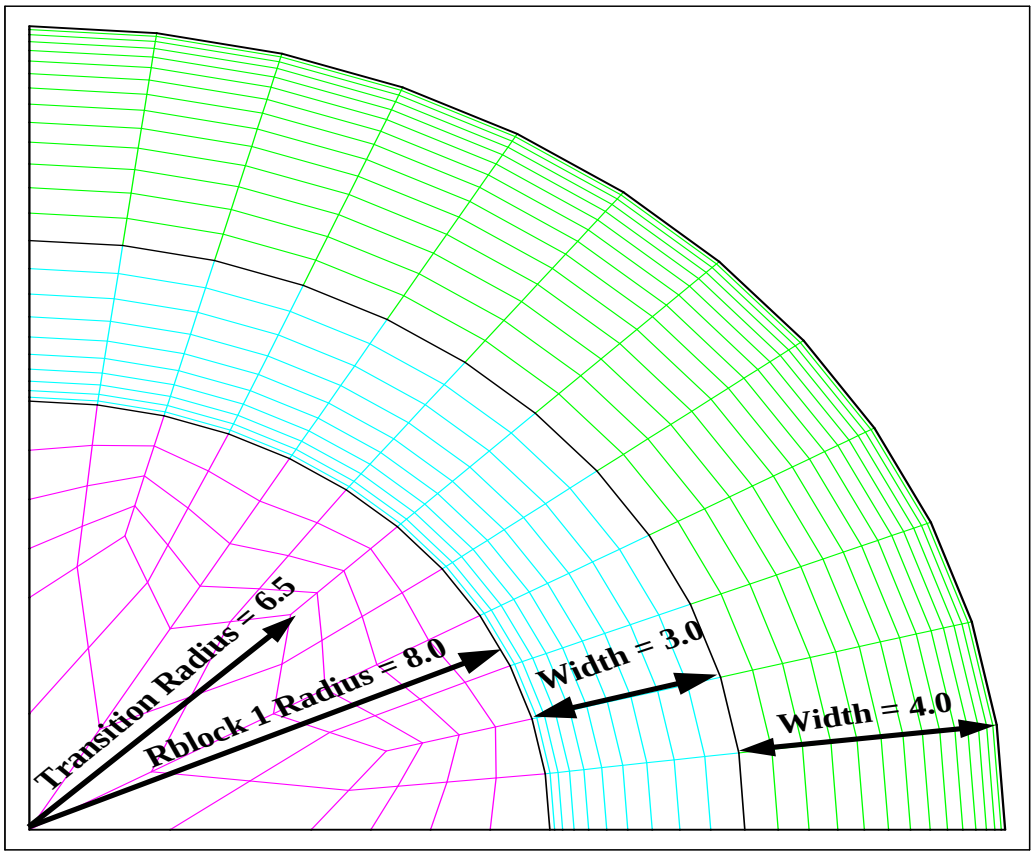


Figure 8: Three dimensional `RADIAL` mesh with azimuthal angle of 360 degrees.

An example of a PAMGEN mesh definition with the `RADIAL TRISECTION` option is illustrated in Figure 9. This figure is annotated to show the correspondence between input parameters and the resulting mesh. This mesh uses a `TRISECTION BLOCKS` setting of 3. In this case the azimuthal angle is 90. degrees and the `FIRST SIZE`, `LAST SIZE`, and `TRANSITION RADIUS` directives are used.

An second example of a PAMGEN mesh definition with the `RADIAL TRISECTION` option is illustrated in Figure 10.

A third example of a PAMGEN mesh definition with the `RADIAL TRISECTION` option is illustrated in Figure 11. In this case the azimuthal angle is 360. degrees and the mesh is a complete circular disk.




```
mesh
  radial trisection
    trisection blocks, 3
    transition radius, 6.5
    numz 1
      zblock 1 1. interval 1
    numr 3
      rblock 1 8.0 interval 4
      rblock 2 3.0 first size 0.05 last size 0.5
      rblock 3 4.0 first size 0.5 last size 0.05
    numa 1
      ablock 1 90. interval 12
  end
end
```

Figure 9: A `RADIAL TRISECTION` mesh with azimuthal angle of 90 degrees and three trisection blocks. `FIRST SIZE` and `LAST SIZE` commands are used to specify element density.

```
mesh
  radial trisection
    trisection blocks, 2
    zmin -0.00075
    numz 1
      zblock 1 1. interval 4
    numr 3
      rblock 1 2.0 interval 4
      rblock 2 3.0 interval 4
      rblock 3 4.0 interval 4
    numa 1
      ablock 1 90. interval 12
  end
end
```

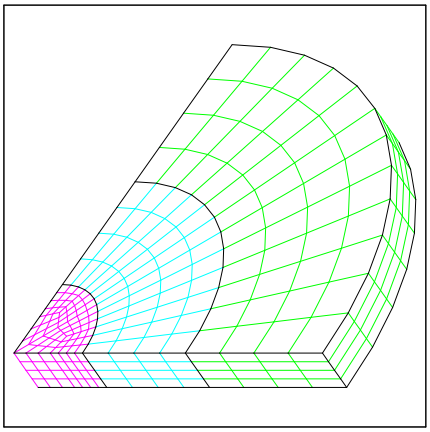


Figure 10: Three dimensional `RADIAL TRISECTION` mesh with azimuthal angle of 90 degrees and two trisection blocks.

```
mesh
  radial trisection
    trisection blocks, 4
    zmin -0.00075
    numz 1
      zblock 1 4. interval 4
    numr 3
      rblock 1 2.0 interval 4
      rblock 2 3.0 interval 4
      rblock 3 5.0 interval 4
    numa 1
      ablock 1 360. interval 32
  end
end
```

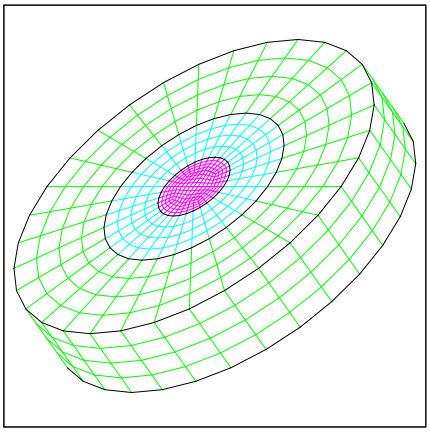


Figure 11: Three dimensional `RADIAL TRISECTION` mesh with azimuthal angle of 360 degrees and four trisection blocks.

5.3.5 Brick

```
BRICK
  NUMZ int (l)
    ZBLOCK 1 real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
    ZBLOCK 2 real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
    ...
    ...
    ZBLOCK l real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
  NUMX int (m)
    XBLOCK 1 real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
    XBLOCK 2 real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
    ...
    ...
    XBLOCK m real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
  NUMY int (n)
    YBLOCK 1 real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
    YBLOCK 2 real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
    ...
    ...
    YBLOCK n real {INTERVAL int | FIRST SIZE real [LAST SIZE real]}
END
```

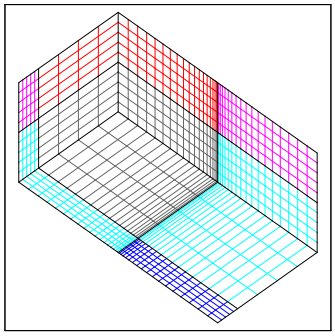
The `BRICK` mesh topology type is a more flexible version of the `RECTILINEAR` type in that it allows different numbers of elements in each element block in each coordinate direction. The creation of a `BRICK` mesh is analogous to the `RADIAL` mesh option and successful creation of these meshes requires sequential specification of the information for the number of elements blocks and their sizes in each coordinate direction.

The `NUMX | NUMY | NUMZ` keywords are followed by an integer specifying the number of element blocks in that coordinate direction. The

Immediately following the `NUMX int | NUMY int | NUMZ int` lines, there must be a `XBLOCK | YBLOCK | ZBLOCK` line for each of the blocks in that direction. The first integer on this line corresponds to the ordinal (beginning with 1) of the line. These lines specify the spatial extent of the particular block and the distribution of elements in the block in that direction. The `INTERVAL int` keyword specifies a fixed number of elements in the block. A `FIRST SIZE real LAST SIZE real` pair specifies the absolute size of the

first and last elements in the block. When the `FIRST SIZE real LAST SIZE real` specification is used, the element sizing will be linear between the first and last elements. The sizes of the first and last elements may be adjusted slightly to provide linear sizing. Omission of the `LAST real` keyword is equivalent to setting the element size to that given by the `FIRST SIZE real` keyword.

An example of a PAMGEN mesh definition with the `BRICK` option is illustrated in Figure 12.



```
mesh
  brick
    numz 2
      zblock 1 2. interval 5
      zblock 2 8. interval 4
    numx 2
      xblock 1 5.0 interval 5
      xblock 2 5.0 interval 5
    numy 2
      yblock 1 10. first size 1. last size .1
      yblock 2 10. first size .1 last size 1.
  end
end
```

Figure 12: Three dimensional `BRICK` mesh.

5.4 Boundary Conditions (Nodesets and Sidesets)

```
SET ASSIGN
   [{NODESET | SIDESET},{IHI | JHI | ... | V00 | V01 | ... | E00 |
   E01 | ... }, int]
   [{BLOCK SIDESET | BLOCK NODESET},{IHI | JHI | ... | V00 | V01 |
   ... | E00 | E01 | ... }, int, int]
   ...
END
```

The `SET ASSIGN -- END` keyword pair allows the specification of nodesets and sidesets on the exterior of meshes. These nodesets and sidesets can be used for specifying boundary conditions on the domain. The nodeset or sideset is applied to the topological face, edge, or vertex associated with the prescribed topological direction. The mesh domain topology and associated labels are shown in Figure 13.This specification applies to the entire domain and cannot be used to specifiy individual blocks.

The most commonly used sub-domains are the exterior faces of the domain. They can be prescribed using IHI, JHI, KHI, ILO, JLO, or KLO. For RECTILINEAR meshes I,J, and K correspond to the coordinate directions x, y, and z. For SPHERICAL meshes I,J, and K correspond to the coordinate directions r, θ, and ϕ. For CYLINDRICAL meshes I,J, and K correspond to the coordinate directions r, θ, and z. The KHI, KLO options do not exist in two dimension simulations.

Specifying a nodeset on the ILO face of a `RADIAL TRISECTION` mesh refers to the edge aligned with the Z axis (see Figure 14). Sidesets may not be specified on the ILO face of `RADIAL TRISECTION` meshes.

BLOCK NODESETS and BLOCK SIDESETS are only available to the BRICK, RADIAL, and RADIAL TRISECTION geometry mesh types.

The `[BLOCK SIDESET | BLOCK NODESET, IHI | JHI | KHI | ILO | JLO | KLO , int, int]` command allows specification of nodesets and sidesets on the topological faces of blocks that may be within the finite element mesh. The first integer specifies the id that the sideset or nodeset will have, the second integer specifies the block on which the sideset or nodeset is applied.

An example of `BLOCK SIDESET` applied to a mesh definition with the

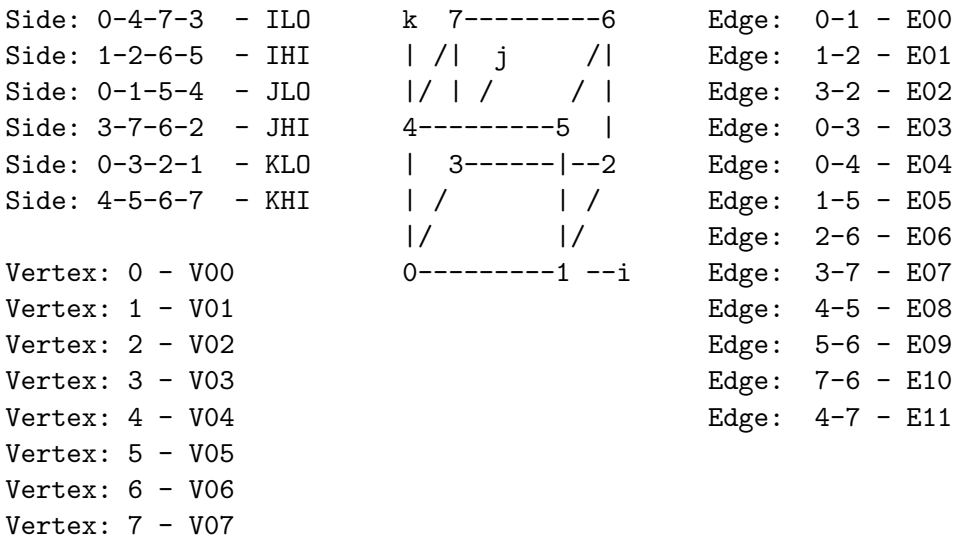


Figure 13: Block topology and labels.

`RADIAL TRISECTION` option is illustrated in Figure 14. In this example a sidet is applied to the `IHI` face of block 2. The sideset applied to block 2 will have sideset id 45. The faces called out in these sidesets will have outward normals facing in the `IHI` direction.

```
mesh
  radial trisection
    trisection blocks, 2
    zmin -0.00075
    numz 1
      zblock 1 1. interval 4
    numr 3
      rblock 1 2.0 interval 4
      rblock 2 3.0 interval 4
      rblock 3 4.0 interval 4
    numa 1
      ablock 1 90. interval 12
  end
  set assign
    nodeset, ilo, 100
    block sideset, ihi, 45, 2
  end
end
```

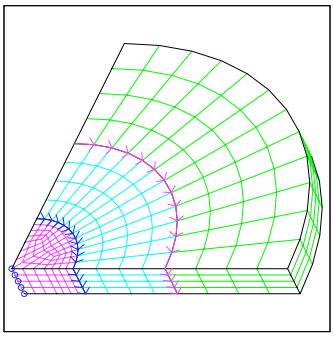


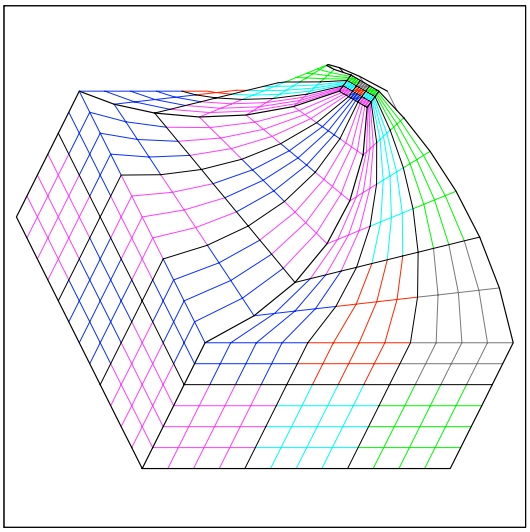
Figure 14: Three dimensional `RADIAL TRISECTION` mesh with azimuthal angle of 90 degrees and two trisection blocks having sidesets specified on the radially outward directed faces of blocks 1 and 2. A nodeset is specified on the ILO face of this mesh and marks the edge corresponding to the z axis (blue circles).

5.5 User Defined Geometry Transformation

```
USER DEFINED GEOMETRY TRANSFORMATION
  "
    user supplied 'C' language instructions;
  "
END
```

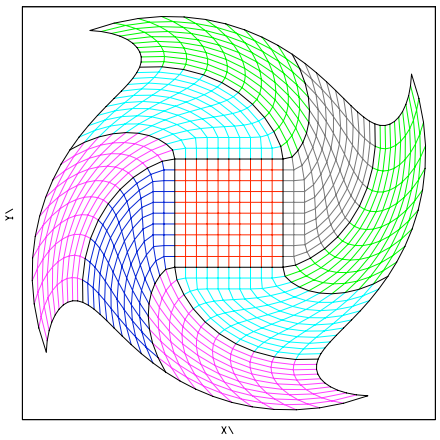
The `USER DEFINED GEOMETRY TRANSFORMATION -- END` keyword pair provides a powerful way to modify the coordinats of any node of a mesh. The keyword-end pair must surround a double quote surrounded block of 'C' code. This code will be called with coordinates of every node in the mesh. It may modify the the coordinats by setting the output variables `outxcoord`, `outycoord`, and `outzcoord`. The unmodified values of the node's coordinates are available in the input variables `inxcoord`, `inycoord`, and `inzcoord`. The coordinates will remain unchanged if the output variables are not modified. A presentation of the capabilities and limitations of runtime compiled 'C' functions is included in Appendix A.

Examples of meshes produced using this capability feature of PAMGEN are shown below in Figure 15 and Figure 16. In the first example the nodes with positive Z coordinate values are rotated about the Z axis an angle proportional to their distance from the Z=0 plane. In the second example nodes a distance of 0.5 from the origin are rotated about the origin in proportion to their distance from the origin.



```
mesh
  rectilinear
    nx = 4
    ny = 4
    nz = 4
    bx =  3
    by =  3
    bz =  3
    gmin = -1.0 -1.0 -1.0
    gmax =  1.0  1.0  1.0
  end
  user defined geometry transformation
     "
     double r = sqrt(inxcoord*inxcoord+inycoord*inycoord);
     double theta = atan2(inycoord,inxcoord);
     if(inzcoord > 0.0)
      {
        theta = theta + (3.14159 / 4.0)*(inzcoord/1.0);
        r = r*(1.0-inzcoord/1.1);
        outxcoord = r*cos(theta);
        outycoord = r*sin(theta);
      }
     "
  end
end
```

Figure 15: 3D mesh illustrating the ability to modify nodal coordinates using `USER DEFINED GEOMETRY TRANSFORMATION`.



```
mesh
  rectilinear
    nx = 10
    ny = 10
    bx =  3
    by =  3
    gmin = -1.0 -1.0
    gmax =  1.0  1.0
  end
  user defined geometry transformation
    "
    double r = sqrt(inxcoord*inxcoord+inycoord*inycoord);
    double theta = atan2(inycoord,inxcoord);
    if(r > 0.5)
     {
       theta = theta + (3.14159 / 4.0)*((r-0.5)/0.5);
       outxcoord = r*cos(theta);
       outycoord = r*sin(theta);
     }
    "
  end
end
```

Figure 16: 2D mesh illustrating the ability to modify nodal coordinates using `USER DEFINED GEOMETRY TRANSFORMATION`.

5.6 User Defined Element Density

```
USER DEFINED ELEMENT DENSITY, {I|J|K}
  "
    user supplied 'C' language instructions;
  "
END
```

The `USER DEFINED ELEMENT DENSITY -- END` keyword pair provides a flexible way to bias `RECTILINEAR`, `SPHERICAL`, and `CYLINDRICAL` meshes. The keyword-end pair must surround a double quote surrounded block of 'C' code that evaluates on the input variable `coord` and sets the return value `field`. The return value `field` must be set to a positive value across the range of the mesh in the selected topological direction. A presentation of the capabilities and limitations of runtime compiled 'C' functions is included in Appendix A.

The mesh biasing adjusts the nodal coordinates such that the density of the elements in a region of the mesh in the selected coordinate direction is proportional to the value of `field` relative to the integral of `field` across the mesh domain. This is implemented by numerically solving the equation given below. In this equation x_i is the coordinate of node i, n is the total number of nodes in the coordinate direction, and $f(u)$ is the user supplied function.

$$\frac{\int_0^{x_i} f(u)du}{\int_0^{x_n} f(u)du} = \frac{i}{n} \tag{5.1}$$

When these functions are applied to a two dimensional `RECTILINEAR` mesh spanning from $(0.0, 0.0)$ to $(1.0, 1.0)$ and having two blocks and 10 elements in both the 'I', and 'J' directions, the resulting mesh is graded as shown in Figure 17. The grading is a continuous exponential function in the 'I' direction and is a discontinuous function in the 'J' direction. In the 'J' direction the domain stretching from 0.0 to 0.5 has twice the element density as the range from 0.5 to 1.0.

Diagnostic information for the user provided functions is included in the `runid.out` file. This information includes the total integrated value of the function, the minimum and maximum value of the function, and a plot of the function's values across the range of evaluation.

```
user defined element density, i
  "
    field = 1.*exp(-5.*(coord));
  "
end
user defined element density, j
  "
    field = 1;
    if(coord < 0.5)  { field = 2;}
    if(coord >= 0.5) { field = 1;}
  "
end
```

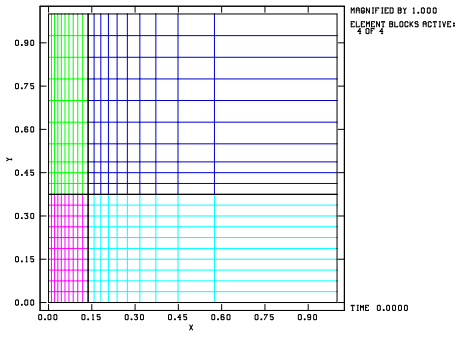


Figure 17: 2D mesh created with a `USER DEFINED ELEMENT DENSITY`.

5.7 Decomposition Strategy

```
DECOMPOSITION STRATEGY
   {BISECTION}
   {PROCESSOR LAYOUT}
     {NUMPROCS, I, int (1)}
     {NUMPROCS, J, int (1)}
     {NUMPROCS, K, int (1)}
   {END}
   {SEQUENTIAL}
   {RANDOM}
END
```

An optional `DECOMPOSITION STRATEGY -- END` block pair surrounds the description of the decomposition method used for parallel simulations. The default strategy is `BISECTION`. The keywords associated with the `DECOMPOSITION STRATEGY` keyword are given in Table 4.

The `BISECTION` decomposition strategy is the default for parallel calculations. This is because it is robust in providing decompositions and the resulting regions have satisfactory surface area to volume ratios. This strategy attempts to automatically determine the number of slices to make through the entire mesh domain to provide an equal number of elements to each processor. This strategy will be most successful when the number of processors, and the number of elements in each direction are a power of 2 or a product of several prime numbers.

The `PROCESSOR LAYOUT` decomposition strategy offers the user improved control of the distribution of elements to each processor in a parallel simulation. The strategy divides the mesh into the number of segments specified by the keyword-value pair for each coordinate direction. The number of processors must equal the product of the values given for each of the `NUMPROCS` directions. The default value for a coordinate direction is one.

When using a `RADIAL TRISECTION` mesh the number of processors in the `I` (radial) direction is fixed at 1, and the total number of processors must be equal to the product of the values given for the `J` and `K` `NUMPROCS` directions. For this mesh type the decomposition assigns elements from the inner transition blocks to the processor that owns the adjacent elements in the outer cylinderical blocks. An example of the `PROCESSOR LAYOUT` decomposition

option applied to a RADIAL TRISECTION mesh is shown in Figure 20.

Examples of BISECTION and PROCESSOR LAYOUT decomposition options applied to a mesh definition with the RADIAL option are shown below in Figures 18 and 19. The total number of elements in this problem was 204, 17 in the radial or I direction and 12 in the azimuthal or J direction.

For the BISECTION decomposition the recursive cuts made on the 17x12x1 array of elements results in three processors with 3x12x1 elements, four processors with 2x12x1 elements and a single processor 6x8x1 elements.

For the PROCESSOR LAYOUT decomposition the 17 elements in the I or radial direction are divided by 4 to set the size of segments produced in that direction at 4. The first segment's size is increased by one to handle the remainder of dividing 14 by 4. This decomposition would equally distribute elements to each processor if the I direction had a number of elements evenly divisible by 4.

```
mesh
  radial
    numz 1
      zblock 1 10.0 interval 1
    numr 4 initial radius 1.
      rblock 1 2. interval 3
      rblock 2 4. interval 3
      rblock 3 6. interval 5
      rblock 4 8. interval 6
    numa 1
      ablock 1 60. interval 12
  end
  decomposition strategy
    bisection
  end
end
```

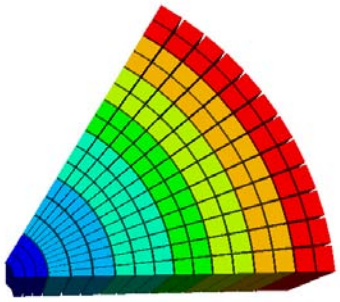


Figure 18: Three dimensional RADIAL mesh with azimuthal angle of 60 degrees run on 7 processors using BISECTION decomposition.

```
mesh
  radial
    numz 1
      zblock 1 10.0 interval 1
    numr 4 initial radius 1.
      rblock 1 2. interval 3
      rblock 2 4. interval 3
      rblock 3 6. interval 5
      rblock 4 8. interval 6
    numa 1
      ablock 1 60. interval 12
  end
  decomposition strategy
    numprocs i, 4
    numprocs j, 2
  end
end
```

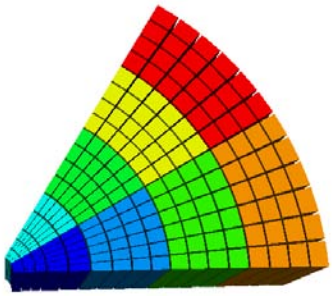


Figure 19: Three dimensional `RADIAL` mesh with azimuthal angle of 60 degrees run on 8 processors using `PROCESSOR LAYOUT` decomposition.

```
mesh
  radial trisection
    trisection blocks, 4
    numz 1
      zblock 1 4.0 interval 1
    numr 3
       rblock 1 2. interval 4
       rblock 2 3. interval 4
       rblock 3 5. interval 4
    numa 1
      ablock 1 360. interval 32
  end
  decomposition strategy
    numprocs j, 8
  end
end
```

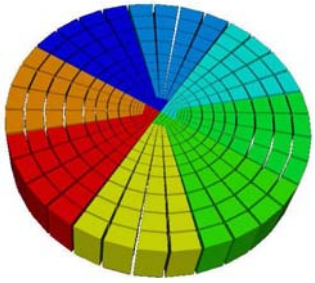


Figure 20: Three dimensional `RADIAL TRISECTION` mesh run on 8 processors using `PROCESSOR LAYOUT` decomposition.

Table 4: Keywords for `DECOMPOSITION STRATEGY -- END`.

Sub-Keyword	Input	Description
`BISECTION`		Recursively bisect domain making slices calculated to assign nearly equal numbers of elements to each processor. This option is the default.

`PROCESSOR LAYOUT`
`NUMPROCS {I|J|K}`, int (1)
`END`
	`int` (1)	Invokes a decomposition strategy that slices up the mesh in accordance with the request of the user. The integer value value is number of segments into which the mesh should be divided in the given direction. The product of the number of segments requested in each direction must equal the number of processors.

`SEQUENTIAL`		Invokes a decomposition strategy that distributes the elements between processors in sequential order. If there are k elements and n processors an average of k/n elements will go to each processor. *This decomposition strategy is not for large simulations and is intended mainly for testing and verification purposes.*
`RANDOM`		Invokes a decomposition strategy that randomly distributes the elements between processors. It results in tremendous communications overhead. *This decomposition strategy is not for production simulations and is intended mainly for testing and verification purposes.*

6 Library Interface

6.1 Creating a Mesh

Mesh creation proceeds through a single function call. Additional functions are available to access messages generated during the mesh creation.

6.1.1 Create_Pamgen_Mesh

```
int Create_Pamgen_Mesh( char * file_char_array,
                        int dimension,
                        int rank,
                        int num_procs);
```

This function creates a representation of the mesh for the processor of the specified rank out of the total num_procs. It returns an enumerated value. A return value of **ERROR_FREE_CREATION** signifies success. A return value of **ERROR_CREATING_IMD** significes an error in the specification of the mesh geometry, topology, or boundary conditions. A return value of **ERROR_CREATING_MS** significes an error in allocating and populating the arrays that store the mesh geometry and topology. A return value of **ERROR_PARSING_DEFINITION** significes an error occurred while parsing the string passed in file_char_array. The details of the syntax error are recoverable by subsequent calls.

char *file_char_array This input variable points to a null terminated string that holds a terse description of the desired mesh. This form of this description is given in a later section.

int dimension This input variable indicates the dimension of the desired mesh. Acceptable values are 2 (quadrilaterals created in x,y plane) and 3 (hexahedral elements created in 3 space).

int rank This input variable may range from 0 to one less than num_procs. It specifies for which processor the mesh is being generated.

int num_procs This input variable must be greater than 0. It specifes the total number of processors across which the mesh is decomposed.

6.1.2 getPamgenEchoStreamSize

```
int getPamgenEchoStreamSize(void);
```

This function returns the size of the string (not counting termination character) that contains an echo of the **char * file_char_array** string previously passed to Create_Pamgen_Mesh. If a parsing error occurred, this string will be annotated with a summary of the error. Use of this function and subsequent access and display of this string is highly recommended if the value pointed to by **int * parse_error_count** is non-zero on return.

6.1.3 getPamgenEchoStream

```
char * getPamgenEchoStream(char * echo_stream_pointer);
```

This function takes a character pointer and returns that same pointer after it has been filled.

char *echo_stream_pointer This input variable must point to allocated memory big enough to hold the the results of "getEchoStreamSize(void)" plus a termination character.

6.1.4 getPamgenErrorStreamSize

```
int getPamgenErrorStreamSize(void);
```

This function returns the size of an error string associated with a return value of **ERROR_CREATING_MS** from Create_Pamgen_Mesh.

6.1.5 getPamgenErrorStream

```
char * getPamgenErrorStream(char * error_stream_pointer);
```

This function takes a character pointer and returns that same pointer after it has been filled.

char *error_stream_pointer This input variable must point to allocated memory big enough to hold the the results of getErrorStreamSize plus a termination character.

6.1.6 getPamgenWarningStreamSize

```
int getPamgenWarningStreamSize(void);
```

This function returns the size of a string containing warnings generated within the Create_Pamgen_Mesh function.

6.1.7 getPamgenWarningStream

```
char * getPamgenWarningStream(char * warning_stream_pointer);
```

This function takes a character pointer and returns that same pointer after it has been filled.

char *warning_stream_pointer This input variable must point to allocated memory big enough to hold the the results of getWarningStreamSize plus a termination character.

6.1.8 getPamgenInfoStreamSize

```
int getPamgenInfoStreamSize(void);
```

This function returns the size of a string containing information messages generated within the Create_Pamgen_Mesh function. These messages include information such as the total number of elements in the mesh, the total number of nodes in the mesh, and the mesh distribution based on decomposition.

6.1.9 getPamgenInfoStream

```
char * getPamgenInfoStream(char * info_stream_pointer);
```

This function takes a character pointer and returns that same pointer after it has been filled.

char *info_stream_pointer This input variable must point to allocated memory big enough to hold the the results of getInfoStreamSize plus a termination character.

6.2 Querying a Mesh

All of the mesh query and access functions are based on the `EXODUS II` [2] and `NEMESIS` [1] APIs. These APIs were written to standardize a platform independent interface for writing and reading binary mesh specification files. `NEMESIS` is a parallel extension of the serial `EXODUS II` API. The PAMGEN function names are formed by prefixing the `EXODUS II` or `NEMESIS` function name with **im_**. The remainder of the function signature and functionality remains unchanged. It may help to note that the original `EXODUS II` functions begin with **ex_**, and the original `NEMESIS` functions begin with **ne_**.

Querying a mesh database to build up a complete representation in accessible memory is a straightforward process. Typically a query function is used to ascertain the number of items in an array, then memory is allocated and passed in through an access function which fills the memory with the requested information. The query and access functions will be presented in the same order they appear in the example function "read_mesh_to_memory()" shown in Appendix B.

6.2.1 im_ex_get_init

```
int im_ex_get_init( int exoid,
                    char *title,
                    int  *num_dim,
                    int  *num_nodes,
                    int  *num_elem,
                    int  *num_elem_blk,
                    int  *num_node_sets,
                    int  *num_side_sets);
```

This function is based on the `EXODUS II` API and is concerned only with serial information. The number of nodes, elements, element blocks is limited to those entities local to the rank processor for which the mesh was created. It returns a non-zero value if an error occurs.

int exoid An unused input variable.

char *title A title string containing "PAMGEN Inline Mesh".

char *num_dim On return points to the number of coordinates per node (2 or 3).

char *num_nodes On return points to the number of nodes.

char *num_elem On return points to the number of elements.

char *num_elem_blk On return points to the number of element blocks.

char *num_node_sets On return points to the number of node sets.

char *num_side_sets On return points to the number of side sets.

6.2.2 im_ex_inquire

```
int im_ex_inquire( int exoid,
                   int query_value,
                   int *int_value,
                   float *float_value,
                   char * char_array_value);
```

This function is based on the `EXODUS II` API and is concerned only with serial information. This is a general purpose query function that takes an enumerated query value and changes the value pointed to by int_value, float_value, or char_array_value depending on the data requested by the query_value. It returns a non-zero value in case of error. Im_ex_inquire supports the following query values:

IM_EX_INQ_NS_NODE_LEN The length of the concatenated node-set list is returned in int_value.

IM_EX_INQ_NS_DF_LEN The length of the concatenated nodeset distribution list is returned in int_value.

IM_EX_INQ_SS_ELEM_LEN The length of the concatenated sidesets element list is returned in int_value.

IM_EX_INQ_SS_NODE_LEN The aggregate length of the sideset nodes is returned in int_value.

IM_EX_INQ_SS_DF_LEN The length of the concatenated side sets distribution factor is returned in int_alue.

IM_EX_INQ_API_VERS The API version is returned in float_value.

IM_EX_INQ_EB_PROP The number of element block properties is returned in int_value.

IM_EX_INQ_NS_PROP The number of node sets properties is returned in int_value.

IM_EX_INQ_SS_PROP The number of side set properties is returned in int_value.

IM_EX_INQ_QA The number of QA records is returned in int_value.

IM_EX_INQ_INFO The number of INFO records is returned in int_value.

6.2.3 im_ex_get_coord

```
int im_ex_get_coord( int exoid,
                     double* x_coors,
                     double* y_coors,
                     double* z_coors);
```

This function is based on the `EXODUS II` API and is concerned only with serial information. This function takes pointers to memory allocated to a length equal to the number of nodes local to the processor and fills in the nodal coordinate values. A non-zero return value indicates an error.

int exoid Unused input variable.

double * x_coors Returned X coordinates of the nodes.

double * y_coors Returned Y coordinates of the nodes.

double * z_coors Returned Z coordinates of the nodes (if num_dim = 3).

6.2.4 im_ex_get_coord_names

```
int im_ex_get_coord_names( int exoid,
                           char ** coord_names);
```

This function is based on the `EXODUS II` API and is concerned only with serial information. It returns the names of the coordinates. A non-zero return indicates an error.

int exoid Unused input variable.

char ** coord_names Returned vector pointing to num_dim coord names. **coord_names** can be declared and allocated as shown below.

```
char* coord_names[3];
for(int i = 0; i < num_dim; i++)
   coord_names[i] = (char*)calloc((MAX_STR_LENGTH+1),sizeof(char));
```

6.2.5 im_ex_get_map

```
int im_ex_get_map( int exoid,
                   int * element_map);
```

This function is based on the `EXODUS II` API and is concerned only with serial information. It loads the global element numbers into the provided storage. A non-zero return value indicates an error.

int exoid Unused input variable.

int * element_map On return this array holds the global element ids of the elements local to this processor. For a serial problem this runs sequentially from 1 to num_elem. Memory sized to num_elem must be allocated prior to making this call.

6.2.6 im_ex_get_elem_num_map

```
int im_ex_get_elem_num_map( int exoid,
                            int * element_num_map);
```

For PAMGEN this function is identical to "im_ex_get_map".

int exoid Unused input variable.

int * element_num_map On return this array holds the global element ids of the elements local to this processor. For a serial problem these values run sequentially from 1 to num_elem. Memory sized to num_elem must be allocated prior to making this call.

6.2.7 im_ex_get_node_num_map

```
int im_ex_get_node_num_map( int exoid,
                            int * node_num_map);
```

This function is based on the EXODUS II API and is concerned only with serial information. It loads the global node numbers into the provided storage. A non-zero return indicates an error.

int exoid Unused input variable.

int * node_num_map On return this array holds the global node ids of the nodes local to this processor. For a serial problem this runs from 1 to num_nodes. Memory sized to num_nodes must be allocated prior to making this call.

6.2.8 im_ex_get_elem_blk_ids

```
int im_ex_get_elem_blk_ids( int exoid,
                            int * elem_blk_ids);
```

This function is based on the EXODUS II API and is concerned only with serial information. It loads the element block ids into the provided storage. A non-zero return value indicates an error.

int exoid Unused input variable.

int * elem_blk_ids On return this array holds the ids of the element blocks on this processor. For PAMGEN meshes the element block ids across the entire problem will run sequentially from 1 to num_elem_blk. On any particular processor of a parallel mesh any set of the global element blocks may be present. Storage sized to num_elem_blk must be allocated prior to making this call.

6.2.9 im_ex_get_elem_block

```
int im_ex_get_elem_block( int exoid,
                          int elem_blk_id,
```

```
                                char * elem_type,
                                int * num_elem_this_blk,
                                int * num_nodes_per_elem,
                                int * num_attr);
```

This function is based on the EXODUS II API and is concerned only with serial information. It provides information about the requested element block. Review of the conventions documented in the EXODUS II manual is the most effective way to understand the mesh storage and retrieval. Under the EXODUS II convention, a finite element mesh is composed of one or more blocks of elements. Each block contains elements of the same type. Within a block, all elements have the same number of nodes and the same connectivity convention. The number of nodes is stored explicitly, and the connectivity convention is called out by a string. This string corresponds to a table of conventional element connectivities in the EXODUS II manual. A non-zero return value indicates an error.

int exoid Unused input variable.

int elem_blk_id Input variable specifying the id of the block for which information is requested. This id must be one of the values in the elem_blk_ids array.

char * elem_type On return this variable holds one of the standard EXODUS II element types as a string. The element_type pointer must be allocated as length MAX_STR_LENGTH+1. For PAMGEN the stored value will be QUAD in 2D or HEX in 3D.

int * num_elem_this_blk On return this variable holds the number of elements in this block.

int * num_nodes_per_elem On return this variable holds the number of nodes per element for the elements in this block.

int * num_attr On return this variable holds the number of attributes for this block for PAMGEN this is always 0.

6.2.10 im_ex_get_elem_conn

```
int im_ex_get_elem_conn( int exoid,
```

```
                                int elem_blk_id,
                                int * connect);
```

This function is based on the EXODUS II API and is concerned only with serial information. It fills the storage pointed to by connect with the connectivity of the elements in the block referred to by elem_blk_id. A non-zero return value indicates an error.

int exoid Unused input variable.

int elem_blk_id Input variable specifying the id of the block for which information is requested. This id must be one of the values in the elem_blk_ids array.

int * connect On return the storage pointed to by **connect** holds the connectivity for the requested block. **Connect** must point to storage sized to num_elem_this_block*num_nodes_per_elem. **Connect** holds (in the order specified by the EXODUS II convention) the nodes of each element in the specified block. The first element's connectivity begins at an offset of 0 and the nth element's connectivity begins at an offset of n*num_nodes_per_element. The EXODUS II standard specifies that the indices of the connectivity are numbered from 1 so that in order to retrieve the coordinates of an element's nodes the indices given in **connect** must be decrimented by 1.

6.2.11 im_ex_get_node_set_ids

```
int im_ex_get_node_set_ids( int exoid,
                            int * node_set_ids);
```

This function is based on the EXODUS II API and is concerned only with serial information. It loads the node set ids into the provided storage. Node set ids are specified in the string passed to the Create_Pamgen_Mesh function. By convention they must be non-zero and positive. A non-zero return value indicates an error.

int exoid Unused input variable.

int * node_set_ids On return this array holds the ids of the node sets on this processor. Storage sized to num_node_sets must be allocated prior to making this call.

6.2.12 im_ex_get_node_set_param

```
int im_ex_get_node_set_param( int exoid,
                              int node_set_id,
                              int * num_nodes_in_node_set,
                              int * num_df_in_node_set);
```

This function is based on the EXODUS II API and is concerned only with serial information. It function provides sizing information for node set data. A non-zero return value indicates an error.

int exoid Unused input variable.

int node_set_id Input variable specifying the id of the node set for which information is requested. This id must be one of the values in the node_set_ids array.

int * num_nodes_in_node_set On return this variable is set to the number of nodes in the specified node set.

int * num_df_in_node_set On return this variable is set to the number of df in the specified node set. Distribution factors are scalar values linked to the members of node sets or side sets. PAMGEN does not produce any distribution factors on node sets or side sets. For PAMGEN this will be 0.

6.2.13 im_ex_get_node_set

```
int im_ex_get_node_set( int exoid,
                        int node_set_id,
                        int * node_set_node_list);
```

This function is based on the EXODUS II API and is concerned only with serial information. On return the provided storage is populated with the local nodes of the node set.

int exoid Unused input variable.

int node_set_id Input variable specifying the id of the node set for which information is requested. This id must be one of the values in the node_set_ids array.

int * node_set_node_list On return this array holds the local ids of the nodes in the node set. The ids are numbered from 1. Storage must be sized for num_nodes_in_nodeset.

6.2.14 im_ex_get_side_set_ids

```
int im_ex_get_side_set_ids( int exoid,
                            int * side_set_ids);
```

This function is based on the EXODUS II API and is concerned only with serial information. It loads the side set ids into the provided storage. Side set ids are specified in the string passed to the Create_Pamgen_Mesh function. By convention they must be non-zero and positive. A non-zero return value indicates an error.

int exoid Unused input variable.

int * side_set_ids On return this array holds the ids of the side sets on this processor. Storage sized to num_side_sets must be allocated prior to making this call.

6.2.15 im_ex_get_side_set_param

```
int im_ex_get_side_set_param( int exoid,
                              int side_set_id,
                              int * num_sides_in_side_set,
                              int * num_df_in_side_set);
```

This function is based on the EXODUS II API and is concerned only with serial information. It provides sizing information for side set data. A non-zero return value indicates an error.

int exoid Unused input variable.

int side_set_id Input variable specifying the id of the side set for which information is requested. This id must be one of the values in the side_set_ids array.

int * num_sides_in_side_set On return this variable is set to the number of sides in the specified side set.

int * num_df_in_side_set On return this variable is set to the number of df in the specified side set. Distribution factors are scalar values linked to the members of node sets or side sets. PAMGEN does not produce any distribution factors on node sets or side sets. For PAMGEN this will be 0.

6.2.16 im_ex_get_side_set

```
int im_ex_get_side_set( int exoid,
                        int side_set_id,
                        int * side_set_element_list,
                        int * side_set_side_list);
```

This function is based on the `EXODUS II` API and is concerned only with serial information. It populates storage that specifies the sides of elements that are in a particular side set. The `EXODUS II` convention specifies a side by listing an element id and the side of that element that is in the sideset. This information is provided in two corresponding arrays of equal length. The sides specify a set of nodes on the element by reference to element topology tables in the `EXODUS II` manual. A non-zero return value indicates an error.

int exoid Unused input variable.

int side_set_id Input variable specifying the id of the side set for which information is requested. This id must be one of the values in the side_set_ids array.

int * side_set_element_list On return this array holds the local ids of the elements having sides in the side set. Storage must be sized for num_sides_in_sideset.

int * side_set_side_list On return this array holds the sides that are in the side set. Storage must be sized for num_sides_in_sideset.

6.2.17 im_ex_get_qa

```
int im_ex_get_qa( int exoid,
```

```
                    char * qa_record[][4]);
```

This function is based on the EXODUS II API and is concerned only with serial information. It populates previously allocated qa_record storage with Quality Assurance (QA) information. By convention the four components of each record are:

1. The analysis code name
2. The analysis code QA descriptor
3. The analysis time
4. The analysis date

PAMGEN will return "PAMGEN", "Parallel Mesh Generator" and then two copies of the date and time. A non-zero return value indicates an error.

int exoid Unused input variable.

char * qa_record[][**4**] Previously allocated storage that is filled with QA records. The memory may be allocated as shown below.

```
char * qa_record[10][4];
for(int i = 0; i < 10; i++)
   for(int j=0; j<4; j++) qa_record[i][j] = (char*)malloc(MAX_STR_LENGTH+1);
```

6.2.18 im_ex_get_info

```
int im_ex_get_info( int exoid,
                    char ** info_record);
```

This function is based on the EXODUS II API and is concerned only with serial information. It populates previously allocated info_record storage. A non-zero return value indicates an error.

int exoid Unused input variable.

char ** info_record Previously allocated storage into which info records are copied. At present for PAMGEN num_info_records is zero. Memory should would be allocated as shown below.

```
char ** info_record;
info_record = (char**)malloc(num_info_records*sizeof(char*));
for(int i = 0; i < num_info_records; i++)
   info_record[i] = (char*)malloc(MAX_STR_LENGTH+1);
```

6.2.19 im_ne_get_init_global

This function is adapted from the NEMESIS API. It is typically the first function called when gathering information about the parallel nature of the mesh. It retrieves the mesh sizing information for the complete mesh. A non-zero return value indicates an error.

```
int im_ne_get_init_global( int   neid,
                           int  *num_nodes_global,
                           int  *num_elems_global,
                           int  *num_elem_blks_global,
                           int  *num_node_sets_global,
                           int  *num_side_sets_global );
```

int neid Unused input variable.

int * num_nodes_global On return this variable is set to the total number of nodes in the mesh.

int * num_elems_global On return this variable is set to the total number of elements in the mesh.

int * num_elem_blks_global On return this variable is set to the total number of element blocks in the mesh.

int * num_node_sets_global On return this variable is set to the total number of node sets in the mesh.

int * num_side_sets_global On return this variable is set to the total number of side sets in the mesh.

6.2.20 im_ne_get_init_info

This function is adapted from the NEMESIS API. It retrieves information about the decomposition of the mesh. A non-zero return value indicates an error.

```
int im_ne_get_init_info( int   neid,
                         int  *num_proc,
                         int  *num_proc_in_file,
                         char *file_type);
```

int neid Unused input variable.

int *num_proc On return this variable is filled with the total number of processors over which the mesh is spread.

int *num_proc_in_file On return this variable is filled with the number of processors for which mesh is available via query. This value will always be 1 when using PAMGEN.

char *file_type Unused variable should be sized as shown below.

```
char file_type [2];
```

6.2.21 im_ne_get_eb_info_global

```
int im_ne_get_eb_info_global( int neid,
                              int *el_blk_ids_global,
                              int *el_blk_cnts_global);
```

This function is adapted from the NEMESIS API. It retrieves the element block ids from the entire mesh as well as the sizes of each of these element blocks. A non-zero return value indicates an error.

int neid Unused input variable.

int *el_blk_ids_global On return this array holds the element block ids for the entire mesh. It must be sized to hold num_elem_blks_global integers.

int *el_blk_cnts_global On return this array holds the number of elements in each element block of the entire mesh. It must be sized to hold num_elem_blks_global integers.

6.2.22 im_ne_get_ns_param_global

```
int im_ne_get_ns_param_global(int  neid,
                              int *ns_ids_global,
                              int *ns_n_cnt_global,
                              int *ns_df_cnt_global);
```

This function is adapted from the NEMESIS API. It retrieves node set information for the entire mesh. A non-zero return value indicates an error.

int neid Unused input variable.

int *ns_ids_global On return this array holds the node set ids for the entire mesh. It must be sized to hold num_node_sets_global integers.

int *ns_n_cnt_global On return this array holds the number of nodes in each node set for the entire mesh. It must be sized to hold num_node_sets_global integers.

int *ns_df_cnt_global On return this array holds the number of node set distribution factors for the entire mesh. It must be sized to hold num_node_sets_global integers. For PAMGEN the number of distribution factors for each node set will be 0.

6.2.23 im_ne_get_ss_param_global

```
int im_ne_get_ss_param_global(int  neid,
                              int *ss_ids_global,
                              int *ss_s_cnt_global,
                              int *ss_df_cnt_global);
```

This function is adapted from the NEMESIS API. It retrieves side set information for the entire mesh. A non-zero return value indicates an error.

int neid Unused input variable.

int *ss_ids_global On return this array holds the side set ids for the entire mesh. It must be sized to hold num_side_sets_global integers.

int *ss_s_cnt_global On return this array holds the number of sides in each side set for the entire mesh. It must be sized to hold num_side_sets_global integers.

int *ss_df_cnt_global On return this array holds the number of side set distribution factors for the entire mesh. It must be sized to hold num_side_sets_global integers. For PAMGEN the number of distribution factors for each side set will be 0.

6.2.24 im_ne_get_loadbal_param

```
int im_ne_get_loadbal_param( int  neid,
                             int *num_internal_nodes,
                             int num_border_nodes,
                             int *num_external_nodes,
                             int *num_internal_elems,
                             int *num_border_elems,
                             int *num_node_cmaps,
                             int *num_elem_cmaps,
                             int  proc);
```

This function is adapted from the NEMESIS API. On return its arguments are filled with sizing information for processor communication data. This information is the first step in gathering all the information required to construct communication protocols between adjacent regions of decomposed mesh. A non-zero return value indicates an error.

int neid Unused input variable.

int * num_internal_nodes On return this variable is filled with the number of nodes that are local to the mesh on this processor.

int * num_border_nodes On return this variable is filled with the number of nodes that are common to the mesh on this processor and to adjacent processors.

int * num_external_nodes On return this variable is filled with the number

of nodes that are not local to this processor but are common to elements that share nodes with this processor.

int * num_internal_elems On return this variable is filled with the number of elements that are local to this processor.

int * num_border_elems On return this variable is filled with the number of elements that are not local to this processor but do share nodes with elements local to this processor.

int * num_node_cmaps On return this variable is filled with the number of node communication maps.

int * num_elem_cmaps On return this variable is filled with the number of element communication maps.

int proc Unused input variable.

6.2.25 im_ne_get_elem_map

```
int im_ne_get_elem_map( int   neid,
                        int  *elem_mapi,
                        int  *elem_mapb,
                        int   proc);
```

This function is adapted from the NEMESIS API. On return the the arguments of this function are populated with the internal and boundary element maps.

int neid Unused input variable.

int * elem_mapi On return this variable is filled with the internal element ids. Storage sized to num_internal_elems must be supplied.

int * elem_mapb On return this variable is filled with the border element ids. Storage sized to num_border_elems must be supplied.

int proc Unused input variable.

6.2.26 im_ne_get_node_map

```
int im_ne_get_node_map( int   neid,
                        int  *node_mapi,
                        int  *node_mapb,
                        int  *node_mape,
                        int   proc);
```

This function is adapted from the NEMESIS API. On return the the arguments of this function are populated with the internal and boundary node maps.

int neid Unused input variable.

int * node_mapi On return this variable is filled with the internal node ids. Storage sized to num_internal_nodes must be supplied.

int * node_mapb On return this variable is filled with the border node ids. Storage sized to num_border_nodes must be supplied.

int * node_mape On return this variable is filled with the external node ids. Storage sized to num_external_nodes must be supplied.

int proc Unused input variable.

6.2.27 im_ne_get_cmap_params

```
int im_ne_get_cmap_params( int neid,
                           int *node_cmap_ids,
                           int *node_cmap_node_cnts,
                           int *elem_cmap_ids,
                           int *elem_cmap_elem_cnts,
                           int  processor);
```

This function is adapted from the NEMESIS API. On return the storage passed in its arguments is filled with the communication map ids and counts. A non-zero return value indicates an error.

int neid Unused input variable.

int * node_cmap_ids On return this storage is filled with the ids for each node communication map. Storage sized to num_node_cmap must be provided.For PAMGEN these values will run from 0 to num_node_cmaps-1.

int * node_cmap_cnts On return this storage is filled with the number of nodes in each node communication map. Storage sized to num_node_cmap must be provided.

int * elem_cmap_ids On return this storage is filled with the ids for each element communication map. Storage sized to num_elem_cmap must be provided. For PAMGEN these values will run from 0 to num_elem_cmaps-1.

int * elem_cmap_elem_cnts On return this storage is filled with the number of elements in each element communication map. Storage sized to num_elem_cmap must be provided.

int proc Unused input variable.

6.2.28 im_ne_get_node_cmap

```
int im_ne_get_node_cmap( int  neid,
                         int  map_id,
                         int *node_ids,
                         int *proc_ids,
                         int  processor);
```

This function is adapted from the NEMESIS API. On return the storage passed in its arguments is filled with the node communication maps. A nonzero return value indicates an error.

int neid Unused input variable.

int map_ids The id of the node communication map that is being queried. This must be one of the entries in node_cmap_ids.

int * node_ids On return this storage is filled with the ids of nodes in the map. Storage must be sized to node_cmap_cnts[map_id].

int * proc_ids The processor id onto which the associated node in node_ids maps. Storage must be sized to node_cmap_cnts[map_id].

int proc Unused input variable.

6.2.29 im_ne_get_elem_cmap

```
int im_ne_get_elem_cmap( int  neid,
                         int  map_id,
                         int *elem_ids,
                         int *side_ids,
                         int *proc_ids,
                         int  processor);
```

This function is adapted from the `NEMESIS` API. On return the storage passed in its arguments is filled with the element communication maps. A non-zero return value indicates an error.

int neid Unused input variable.

int map_ids The id of the element communication map that is being queried.This must be one of the entries in elem_cmap_ids.

int * elem_ids On return this storage is filled with the ids of elements in the map. Storage must be sized to elem_cmap_cnts[map_id].

int * side_ids On return this storage is filled with the ids sides of elements in the map. Storage must be sized to elem_cmap_cnts[map_id].

int * proc_ids The processor id onto which the associated element in elem_ids maps. Storage must be sized to elem_cmap_cnts[map_id].

int proc Unused input variable.

6.3 Deleting a Mesh

6.3.1 Delete_Pamgen_Mesh

```
int Delete_Pamgen_Mesh(void);
```

This function clears and deletes the memory used to store mesh data created with the "Create_Pamgen_Mesh(...)" function. After this function is called "Create_Pamgen_Mesh(...)" may be called again to create a different mesh.

A Runtime Compiler Functionality

The runtime compiler allows inclusion of double quoted (“ “) 'C' language style functions within unformatted input files. The functions are evaluated during program setup or execution to calculate independent solution variables.

This provides the user with an endlessly flexible method for describing boundary conditions, initial conditions, source terms, material properties, or any other independent variable.

The specific variable names expected within runtime compiled functions depends on the host code and the context of the function use. In general it should be remembered that the runtime functions return quantities by modifying variables that are passed in by reference.

A.1 The RTC language

The RTC language can be thought of as a small subset of the C language with a couple minor modifications.

A.1.1 Operators

The RTC language has the following operators that work exactly as they do in C and have the same precedence as they do in C:

- + Addition
- − Subtration
- − Negation
- ∗ Multiplication
- / Division
- == Equality
- > Greater than

- < Less than
- >= Greater than or equal to
- <= Less than or equal to
- = Assignment
- || Logical or
- && Logical and
- ! = Inequality
- % Modulo
- ! Logical not

The following operators do not occur in the C language, but were added to the RTC language for convenience:

- `^ Exponentiation`

A.1.2 Control flow

The RTC language has the following control flow statements:

- for(expr ; expr ; expr) { ... }
- while(expr) { ... }
- if (expr) {...}
- else if (expr) {...}
- else {...}

These control flow statements work exactly as they do in C except that the code blocks following a control flow statement **MUST** be enclosed within braces even if the block only consists of one line.

A.1.3 Line Structure

The line structure in the RTC language is the same as that of C. Expressions end with a semicolon unless they are inside a control flow statement.

A.1.4 Variables

Declaring scalar variables in RTC is done exactly as it is done in C except that only the following types are supported:

- int
- float
- double
- char

For scalars, variables can be declared and assigned all at once. Both of the following approaches will work:

```
int myVar = 9;

OR

int myVar;
myVar = 9;
```

Arrays work a little differently in RTC than they do in C. There are no *new* or *malloc* operators, instead the user may declare dynamically sized arrays in the same manner as statically sized arrays. Also, in C all the values of an array may be initialized at once by putting the values within braces. This is not supported in the RTC language. Users will have to loop through the array and assign the values one by one. For example:

```
LEGAL:
   int ia[x*y];  //Note: in C this would not be legal for non-const x,y
```

```
   int ia2[3];

NOT LEGAL:
   int ia[3] = {1, 2, 3};
```

Indexing arrays can be done using the index operator: array[expr] = ...;

Bounds checking is done at run time. If the bounds of an array are exceeded, it will dump an error to stdout.

A.1.5 Math

The following math.h functions are available in RTC:

- asin(arg) : returns the arc sine of arg
- acos(arg) : returns the arc cosine of arg
- atan(arg) : returns the arc tangent of arg
- atan2(y, x): returns the arc tangent of y/x
- sin(arg) : returns the sine of arg
- cos(arg) : returns the cosine of arg
- tan(arg) : returns the tangent of arg
- sqrt(arg) : returns the square root of arg
- exp(arg) : returns the natural logarithm base e raised to the arg power
- sinh(arg) : returns the hyperbolic sine of arg
- cosh(arg) : returns the hyperbolic cosine of arg
- tanh(arg) : returns the hyperbolic tangent of arg
- log(arg) : returns the natural logarithm for arg
- log10(arg) : returns the base 10 logarithm for arg

- rand(arg) : returns a system-generated random integer between 0 and RAND_MAX using seed arg
- fabs(arg) : returns the absolute value of arg
- pow(b, e) : returns b to the e power (Note: the Exponentiation operator is available)
- j0(arg) : Bessel function of order zero
- j1(arg) : Bessel function of order one
- i0(arg) : Modified Bessel function of order zero
- i1(arg) : Modified Bessel function of order one
- erf(arg) : Error function
- erfc(arg) : Complementary error function (1.0 - erf(x))
- gamma(arg) : returns $\Gamma(arg)$

A.1.6 Strings

The user may pass quoted strings as arguments to functions. Note: it may be necessary to escape-out the double quotes so that they do not confuse the input-file parser. See printf section below for an example.

A.1.7 Printf

The RTC printf method is called just like its C counterpart. The first argument is a quoted character string. This string will contain the % symbol which will tell RTC to output the corresponding argument. The only difference between RTC's printf and C's printf is that in RTC's version, a type character after the % is unnecessary. For example, inside an RTC method the following is appropriate:

```
printf(\"One:% Two:% Three:% \", 5-4, 2.0e0, 'c');\n\
```

Which would generate this output: One:1 Two:2 Three:c

A.1.8 Comments

The traditional C-comment mechanism may be used inside RTC functions. Use /* to begin a comment and */ to end the comment.

A.1.9 Unsupported Features

Implementing the entire C-language was well beyond the intent of RTC. Features that were too difficult or did not add enough value have been left out. The following is a list of common C features that are unsupported in RTC:

- There are no ++ or −− operators. Use $i = i + 1$ instead of $++i$
- Structs
- Pointers
- Instant array initialization: int array[5] = 1,2,3,4,5;
- Case statements
- Casting
- Labels and gotos
- Function definition/declaration
- stdio
- Keywords: break, continue, const, enum, register, return, sizeof, typedef, union, volatile, static.

B read_mesh_to_memory

This appendix lists an example of the "C" source code that an application linked to PAMGEN would use to read a finite element mesh description from PAMGEN.

```
/*****************************************************************************/
void read_mesh_to_memory()
/*****************************************************************************/
{
  int im_exoid = 0;
  int idum = 0;
  float fdum;

  mss.bptr[0] = mss.buffer[0];
  mss.bptr[1] = mss.buffer[1];
  mss.bptr[2] = mss.buffer[2];

  for(int i = 0; i < 100; i++)
    for(int j=0; j<4; j++) mss.qaRecord[i][j] = (char*)malloc(MAX_STR_LENGTH+1) ;


  char * cdum = NULL;
  int error = 0;
  int id = 0;
  error += im_ex_get_init (  id,
                             mss.title,
                             &mss.num_dim,
                             &(mss.num_nodes),
                             &mss.num_elem,
                             &mss.num_elem_blk,
                             &mss.num_node_sets,
                             &mss.num_side_sets);


  error += im_ex_inquire(id, IM_EX_INQ_NS_NODE_LEN, (int*)&mss.num_node_set_nodes,
                         &fdum, cdum);
  error += im_ex_inquire(id, IM_EX_INQ_NS_DF_LEN,   (int*)&mss.num_node_set_dfs,
                         &fdum, cdum);
  error += im_ex_inquire(id, IM_EX_INQ_SS_ELEM_LEN, (int*)&mss.num_side_set_elements,
                         &fdum, cdum);
  error += im_ex_inquire(id, IM_EX_INQ_SS_NODE_LEN, (int*)&mss.num_side_set_nodes,
```

```
                        &fdum, cdum);
error += im_ex_inquire(id, IM_EX_INQ_SS_DF_LEN,   (int*)&mss.num_side_set_dfs,
                        &fdum, cdum);

// // get version number

error += im_ex_inquire(id, IM_EX_INQ_API_VERS, &idum, &fdum, cdum);

mss.version_number = (double) fdum;

mss.version = (int) mss.version_number;

// // get genesis-II parameters

error += im_ex_inquire(id, IM_EX_INQ_EB_PROP,
                        (int*)&mss.num_block_properties,
                        &fdum, cdum);

error += im_ex_inquire(id, IM_EX_INQ_NS_PROP,
                        (int*)&mss.num_node_set_properties,
                        &fdum, cdum);

error += im_ex_inquire(id, IM_EX_INQ_SS_PROP,
                        (int*)&mss.num_side_set_properties,
                        &fdum, cdum);

mss.coord = (double *)malloc(mss.num_nodes*mss.num_dim*sizeof(double));

error += im_ex_get_coord(id,
                          mss.coord,
                          mss.coord+mss.num_nodes,
                          mss.coord+2*mss.num_nodes);


error += im_ex_get_coord_names (id, mss.bptr);

if (mss.num_elem){
  mss.element_order_map = (int *)malloc(mss.num_elem * sizeof(int));
  error += im_ex_get_map(id, mss.element_order_map);

  if (mss.num_elem){
    mss.global_element_numbers = (int *)malloc(mss.num_elem*sizeof(int));
    error += im_ex_get_elem_num_map(id, mss.global_element_numbers);
```

```
}

if (mss.num_nodes){
  mss.global_node_numbers = (int *)malloc(mss.num_nodes * sizeof(int));
  error += im_ex_get_node_num_map(id, mss.global_node_numbers);
}


//block info

mss.block_id           = (int *)malloc(mss.num_elem_blk*sizeof(int));
mss.nodes_per_element  = (int *)malloc(mss.num_elem_blk*sizeof(int));
mss.element_attributes = (int *)malloc(mss.num_elem_blk*sizeof(int));
mss.elements           = (int *)malloc(mss.num_elem_blk*sizeof(int));
mss.element_types      = (char **)malloc(mss.num_elem_blk*sizeof(char *));
mss.elmt_node_linkage  = (int **)malloc(mss.num_elem_blk*sizeof(int*));

error += im_ex_get_elem_blk_ids(id, mss.block_id);

for(int i = 0; i < mss.num_elem_blk; i ++){
  mss.element_types[i] = (char *)malloc((MAX_STR_LENGTH + 1)*sizeof(char));
  error += im_ex_get_elem_block(id,
                                mss.block_id[i],
                                mss.element_types[i],
                                (int*)&(mss.elements[i]),
                                (int*)&(mss.nodes_per_element[i]),
                                (int*)&(mss.element_attributes[i]));
}

//connectivity
for(int b = 0; b < mss.num_elem_blk; b++){
  mss.elmt_node_linkage[b] = (int*)malloc(mss.nodes_per_element[b]*
                                          mss.elements[b]*sizeof(int));
  error += im_ex_get_elem_conn(id,mss.block_id[b],mss.elmt_node_linkage[b]);
}


if(mss.num_node_sets){
  mss.node_set_id           = (int *) malloc(mss.num_node_sets*sizeof(int));
  mss.num_nodes_in_node_set = (int *) malloc(mss.num_node_sets*sizeof(int));
  mss.node_set_nodes        = (int **)malloc(mss.num_node_sets*sizeof(int*));
  mss.num_df_in_node_set    = (int *) malloc(mss.num_node_sets*sizeof(int*));
```

```
  error += im_ex_get_node_set_ids(id, mss.node_set_id);


  for(int i = 0; i < mss.num_node_sets; i ++){
    error += im_ex_get_node_set_param(id, mss.node_set_id[i],
                                      (int*)&mss.num_nodes_in_node_set[i],
                                      (int*)&mss.num_df_in_node_set[i]);

          mss.node_set_nodes[i] = NULL;

    if(mss.num_nodes_in_node_set[i]) {
      mss.node_set_nodes[i] =
       (int *)malloc(mss.num_nodes_in_node_set[i]*sizeof(int));
      error += im_ex_get_node_set(id,
                                  mss.node_set_id[i],
                                  mss.node_set_nodes[i]);
    }
  }
}

//side sets
if(mss.num_side_sets){
  mss.side_set_id = (int*)malloc(mss.num_side_sets*sizeof(int));
  mss.num_elements_in_side_set = (int*)malloc(mss.num_side_sets*sizeof(int));
  mss.num_df_in_side_set = (int*)malloc(mss.num_side_sets*sizeof(int));
  mss.side_set_elements = (int**)malloc(mss.num_side_sets*sizeof(int *));
  mss.side_set_faces = (int **)malloc(mss.num_side_sets*sizeof(int*));

  error += im_ex_get_side_set_ids(id, mss.side_set_id);
  for(int i = 0; i < mss.num_side_sets; i ++){

    error += im_ex_get_side_set_param(id, mss.side_set_id[i],
                                      (int*)&mss.num_elements_in_side_set[i],
                                      (int*)&mss.num_df_in_side_set[i]);

    int ne = mss.num_elements_in_side_set[i];
    mss.side_set_elements[i] = (int*)malloc(ne*sizeof(int));
    mss.side_set_faces[i] = (int*)malloc(ne*sizeof(int));
    if(ne){
      error += im_ex_get_side_set(id, mss.side_set_id[i],
                                  mss.side_set_elements[i],
                                  mss.side_set_faces[i]);
```

```
    }
  }
}

error += im_ex_inquire(id, IM_EX_INQ_QA, (int*)&mss.num_qa_records, &fdum, cdum);

if(mss.num_qa_records)error +=  im_ex_get_qa(id,mss.qaRecord);


error += im_ex_inquire(id, IM_EX_INQ_INFO, (int*)&mss.num_info_records, &fdum, cdum);
if(mss.num_info_records) {
  mss.info_records = (char **)malloc(mss.num_info_records*sizeof(char *));
  for(int i = 0; i < mss.num_info_records; i ++){
    mss.info_records[i] = (char *)malloc(MAX_STR_LENGTH+1);
  }
  error += im_ex_get_info(id, mss.info_records);
}


//nemesis data
// global info
if ( im_ne_get_init_global(id, &mss.num_nodes_global, &mss.num_elems_global,
                           &mss.num_elm_blks_global, &mss.num_node_sets_global,
                           &mss.num_side_sets_global) < 0 )
  ++error;



if ( im_ne_get_init_info(id,
                         &mss.num_total_proc,
                         &mss.num_proc_in_file, mss.type) < 0 )
  ++error;

mss.elem_blk_ids_global = (int*)malloc(mss.num_elm_blks_global*sizeof(int));
mss.elem_blk_cnts_global = (int*)malloc(mss.num_elm_blks_global*sizeof(int));

if ( im_ne_get_eb_info_global(id,
                              mss.elem_blk_ids_global,
                              mss.elem_blk_cnts_global) < 0 )
  ++error;

mss.ns_ids_global = (int *)malloc(mss.num_node_sets_global*sizeof(int));
mss.ns_cnts_global = (int *)malloc(mss.num_node_sets_global*sizeof(int));
```

```
mss.ns_df_cnts_global = (int *)malloc(mss.num_node_sets_global*sizeof(int));
mss.ss_ids_global = (int *)malloc(mss.num_side_sets_global*sizeof(int));
mss.ss_cnts_global = (int *)malloc(mss.num_side_sets_global*sizeof(int));
mss.ss_df_cnts_global = (int *)malloc(mss.num_side_sets_global*sizeof(int));


if ( mss.num_node_sets_global > 0 ) {
  if ( im_ne_get_ns_param_global(id,mss.ns_ids_global,mss.ns_cnts_global,
                                 mss.ns_df_cnts_global) < 0 )++error;
}

if ( mss.num_side_sets_global > 0 ) {
  if ( im_ne_get_ss_param_global(id,mss.ss_ids_global,mss.ss_cnts_global,
                                 mss.ss_df_cnts_global) < 0 )  ++error;
}

//parallel info
if ( im_ne_get_loadbal_param( id,
                              &mss.num_internal_nodes,
                              &mss.num_border_nodes,
                              &mss.num_external_nodes,
                              &mss.num_internal_elems,
                              &mss.num_border_elems,
                              &mss.num_node_comm_maps,
                              &mss.num_elem_comm_maps,
                              0/*unused*/ ) < 0 )++error;

mss.internal_elements = (int *)malloc(mss.num_internal_elems*sizeof(int));
mss.border_elements   = (int *)malloc(mss.num_border_elems*sizeof(int));
mss.internal_nodes    = (int *)malloc(mss.num_internal_nodes*sizeof(int));
mss.border_nodes      = (int *)malloc(mss.num_border_nodes*sizeof(int));
mss.external_nodes    = (int *)malloc(mss.num_external_nodes*sizeof(int));

if ( im_ne_get_elem_map( id,
                         mss.internal_elements,
                         mss.border_elements,
                         0/* not used proc_id*/ ) < 0 )++error;

if ( im_ne_get_node_map( id,
                         mss.internal_nodes,
                         mss.border_nodes,
                         mss.external_nodes,
                         0/* not used proc_id*/ ) < 0 )++error;
```

```
if(mss.num_node_comm_maps > 0){

  mss.node_cmap_node_cnts = (int*) malloc(mss.num_node_comm_maps*sizeof(int));
  mss.node_cmap_ids       = (int*) malloc(mss.num_node_comm_maps*sizeof(int));
  mss.comm_node_ids       = (int**)malloc(mss.num_node_comm_maps*sizeof(int*));
  mss.comm_node_proc_ids  = (int**)malloc(mss.num_node_comm_maps*sizeof(int*));

  mss.elem_cmap_elem_cnts = (int*) malloc(mss.num_elem_comm_maps*sizeof(int));
  mss.elem_cmap_ids       = (int*) malloc(mss.num_elem_comm_maps*sizeof(int));
  mss.comm_elem_ids       = (int**)malloc(mss.num_elem_comm_maps*sizeof(int*));
  mss.comm_side_ids       = (int**)malloc(mss.num_elem_comm_maps*sizeof(int*));
  mss.comm_elem_proc_ids  = (int**)malloc(mss.num_elem_comm_maps*sizeof(int*));

  if ( im_ne_get_cmap_params( id,
                              mss.node_cmap_ids,
                              (int*)mss.node_cmap_node_cnts,
                              mss.elem_cmap_ids,
                              (int*)mss.elem_cmap_elem_cnts,
                              0/*not used proc_id*/ ) < 0 )++error;

  for(int j = 0; j < mss.num_node_comm_maps; j++) {
    mss.comm_node_ids[j]       = (int *)malloc(mss.node_cmap_node_cnts[j]*sizeof(int));
    mss.comm_node_proc_ids[j]  = (int *)malloc(mss.node_cmap_node_cnts[j]*sizeof(int));
    if ( im_ne_get_node_cmap( id,
                              mss.node_cmap_ids[j],
                              mss.comm_node_ids[j],
                              mss.comm_node_proc_ids[j],
                              0/*not used proc_id*/ ) < 0 )++error;

  }



  for(int j = 0; j < mss.num_elem_comm_maps; j++) {
    mss.comm_elem_ids[j]       = (int *)malloc(mss.elem_cmap_elem_cnts[j]*sizeof(int));
    mss.comm_side_ids[j]       = (int *)malloc(mss.elem_cmap_elem_cnts[j]*sizeof(int));
    mss.comm_elem_proc_ids[j]  = (int *)malloc(mss.elem_cmap_elem_cnts[j]*sizeof(int));
    if ( im_ne_get_elem_cmap( id,
                              mss.elem_cmap_ids[j],
                              mss.comm_elem_ids[j],
                              mss.comm_side_ids[j],
```

```
                                    mss.comm_elem_proc_ids[j],
                                    0 /*not used proc_id*/ ) < 0 )++error;


      }//loop over num_elem_co
    }
  }
}
```

References

[1] G. L. Hennigan, M. St. John, and J. N. Shadid. NEMESIS I: A set of functions for describing unstructured finite-element data on parallel computers. Technical report, Sandia National Laboratories, Albuquerque, NM, May 1998.

[2] L. A. Schoof and V. R. Yarberry. EXODUS II: A Finite Element Data Model. Technical report SAND92-2137, Sandia National Laboratories, Albuquerque, NM, November 1995.

Index

Block IDs, 18
Boundary Conditions, 34
Brick, 32

Cylindrical, 23

Decomposition Strategy, 42
Dimensionality, 17

Geometry and Topology, 19

Nodesets, 34

Radial, 25
Radial Trisection, 25
Rectilinear, 19

Set Assign, 34
Sidesets, 34
Spherical, 21

Trisection, 25

User Defined Element Density, 40
User Defined Geometry Transformation, 37

DISTRIBUTION:

1 MS 0321
J. Peery, 1400

1 MS 0384
A. C. Ratzl, 1500

1 MS 0826
D. K. Gartling, 1500

1 MS 1318
D. E. Womble, 1410

1 MS 0321
J. L. Mitchiner, 1410

1 MS 0380
H. S. Morgan, 1540

1 MS 0380
G. D. Sjaardema, 1541

1 MS 0382
M. W. Glass, 1541

1 MS 0836
R. E. Hogan, 1514

1 MS 0370
J. H. Strickland, 1434

1 MS 1322
J. B. Aidun, 1435

1 MS 1318
S. S. Collis, 1416

1 MS 1416
M. A. Heroux, 1416

1 MS 1318
K. D. Devine, 1416

1 MS 0378
R. M. Summers, 1431

1 MS 0378
S. Carroll, 1431

1 MS 0378
D. M. Hensinger, 1431

1 MS 0378
A. C. Robinson, 1431

1 MS 0378
R. R. Drake, 1431

1 MS 0378
O. E. Strack, 1431

1 MS 0378
D. A. Labreche, 1431

1 MS 0378
C. B. Luchini, 1431

1 MS 0378
S. J. Mosso, 1431

1 MS 0378
S. V. Petney, 1431

1 MS 0378
M. K. Wong, 1431

1 MS 0975
J. M. Foucar, 5522

1 MS 1318
T. A. Gardiner, 9326

1 MS 0899
Technical Library, 9536 (electronic copy)


Sandia National Laboratories