SAND2016-10177

SANDIA REPORT

SAND2015-9562
Unlimited Release
Printed October 7, 2016

Percept User Manual

Brian Carnes and Steve Kennon

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

Sandia National Laboratories

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

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

SAND2015-9562
Unlimited Release
Printed October 7, 2016

Percept User Manual

Brian Carnes and Steve Kennon
PO Box 5800
Mail Stop 0828
Albuquerque, NM 87185

Abstract

This document is the main user guide for the Sierra/Percept capabilities including the `mesh_adapt` and `mesh_transfer` tools. Basic capabilities for uniform mesh refinement (UMR) and mesh transfers are discussed. Examples are used to provide illustration.

Future versions of this manual will include more advanced features such as geometry and mesh smoothing. Additionally, all the options for the `mesh_adapt` code will be described in detail. Capabilities for local adaptivity in the context of offline adaptivity will also be included.

Contents

Chapter 1

Introduction

Percept is a collection of tools to enable solution verification. The main emphasis is on spatial mesh modification, including capabilities for uniform and local mesh adaptation. Additionally, we provide separate tools for transfer of field data, verification of modal eigenvalue problems, and scripts for solution verification studies.

Chapter 2

Basic Uniform Mesh Refinement

Percept capabilities are provided through the `mesh_adapt` application code within Sierra. The most commonly used capability is uniform mesh refinement (UMR). For a given mesh called `input.g` in the ExodusII format, the simplest command to generate a uniformly refined mesh called `output.g` would be

```
mesh_adapt --refine=DEFAULT --input_mesh=input.g --output_mesh=output.g
```

This command runs the UMR in serial and uses the default options whereever possible. To see all the options possible just run `mesh_adapt` with the `--help` option (for more detail, use `--Help`).

```
mesh_adapt --help
mesh_adapt --Help
```

Boolean options are activated/deactivated using an integer value of 1/0, respectively. The format for all options is

```
--OPTION_NAME=OPTION_VALUE
```

2.1 Refine option

For uniform mesh refinement, the `--refine` option should always be set to `DEFAULT`.

2.2 Input mesh (Required)

The input mesh specified with `--input_mesh` is assumed to be decomposed into the number of processors specified. If not, an error will occur. Mesh decomposition can be specified from a single base mesh using the Ioss read options (see Section 2.5).

2.3 Output mesh (Required)

The output mesh is specified with `--output_mesh` option. When run in parallel, this mesh will be written in decomposed form, with no concatenation into a single file.

2.4 Number of refines

The number of mesh refinements is set using the `--number_refines` option. If this is not specified, a default value of one is used. Setting this value to zero is useful in some cases, for example to extract mesh quality data or to generate mesh based geometry.

2.5 Ioss read options

Percept provides some specific options that can be passed to the I/O subsystem (Ioss) library that affect how the meshes are input and output. The Ioss input options for reading a mesh can be set using the `--ioss_read_options` option, whose argument is a quoted string of comma-separated options. The list of acceptable options include: "large", "auto-decomp:yes", "auto-decomp:no". These can be combined to look like: "large,auto-decomp:yes". The default value is "auto-decomp:no".

The "large" option reads the input mesh using 64-bit integers as the type for integer fields, such as global IDs of nodes and elements, which is needed when generating meshes with IDs that exceed the 32-bit limit (there is no easy way to predict when this limit will be exceeded, but if the number of elements is approaching 10 million or more, you should consider using the “large” option). The "auto-decomp" option specifies the option to automatically decompose the input mesh in memory, without writing any files to disk.

2.6 Ioss write options

The corresponding Ioss output options can be set using the `--ioss_write_options` option. Acceptable values include: "large", "auto-join:yes", "auto-join:no". These can be combined to look like: "large,auto-join:yes". The default value is "auto-join:no".

The "large" option writes the output mesh using 64-bit integers as the type for integer fields, such as global IDs of nodes and elements (see above for when to use the “large” option). The "auto-join" option specifies the option to automatically combine the output mesh into a single file on disk. This feature is not well-tested and should not be relied upon.

2.7 Respect spacing

Percept has a capability to interpolate new nodes created during refinement using the spacing of the input mesh, which, for example, will preserve the grading towards a boundary. This option is enabled using `--respect_spacing=1` option, which is set on by default. The mesh spacing is computed using the Jacobians of elements in a patch around a node.

Limitations: this feature does not work on meshes with beam or shell elements. In case of meshes with beams and/or shells, the option should be disabled using `--respect_spacing=0`.

2.8 Pre-check memory usage

Before refinement begins, Percept can perform an estimate of the memory needed. This is activated using the `--precheck_memory_usage` option, which is set on by default. When insufficient memory is available, Percept will print an error message and abort.

2.9 Specify blocks to refine

Percept can perform uniform refinement on only a subset of the element blocks. This is specified using the `--blocks` option followed by a list of blocks with a very general syntax. There are several options, including:

1. from a file using “file:my_filename.my_ext” (e.g. “file:filelist.dat”) which will read input block names from the given file
2. a single input block name (e.g. block_3) to be refined
3. include blocks using [+]block_1,[+]block_2, etc ,block_n to include only these blocks (plus sign is optional)
4. exclude blocks using -block_3,-block_5 to exclude blocks from those included (all blocks or include-only blocks) (minus sign is mandatory)
5. include a range of blocks such as block_1..block_10 include the range of blocks numbered 1 to 10
6. any combination of [+] and - options and range (..) option can be specified, separated by commas
7. you can add the optional specification :Nx or :NX of the number of times to refine a particular block to any block name specification, e.g. –blocks=1..3:2x,5,6:3x would refine blocks 1,2 and 3 at most twice, block 5 every refinement pass, and block 6 at most three times.

8. finally, the prefix “block_” can be omitted with only the numbers used instead.

2.10 Use transition elements

When blocks are specified as in the previous section, Percept can refine adjacent elements using transition elements in order maintain a conforming mesh. This is specified using the option `--use_transition_elements=1`. When not specified, the resulting mesh may have hanging nodes when block refinement is used.

2.11 Smooth mesh after refinement

Percept can smooth the mesh after refinement using a mesh quality metric. To smooth all nodes you must enable both `--smooth_geometry=1` and `--smooth_surfaces=1`.

2.12 Handling of blocks, nodesets and sidesets

Percept maintains the structure of the original mesh as much as possible. For example, the refined mesh will almost always have the same number of element blocks. Each block will have the same topology, the same name and approximately eight times the number of elements (for a single refinement). The exception is with pyramid elements, which have an irregular refinement template even for UMR.

Sidesets are preserved in the refined mesh, with child sides created and attached to the corresponding child elements. The orientation of the sides should be consistent with the original mesh.

Nodesets are handled depending on the local membership of newly created nodes, while existing nodes retain their nodeset membership. For example, if all nodes of an edge or face are in a nodeset, the new child nodes on the edge or face are added to the nodeset.

2.13 Element enrichment

In some cases users would like to increase the polynomial order of the elements. Percept has an option for this called enrichment. For most cases it is enough to replace the `-refine=DEFAULT` option on the command line with `-enrich=DEFAULT`. For most element types Percept will add additional nodes and update connectivity so that the mesh is build out of the higher order (quadratic) elements. For example, if you pass in a mesh of 8-node hex elements, `mesh_adapt` will output a new mesh of 27-node hex elements. Similarly meshes of 4-node tet elements will be enriched to be 10-node tet elements.

2.14 Example: basic UMR usage

We present a small example that includes all of the above options. The mesh is a simple 8-node hex mesh of a cube with non-uniform mesh spacing in each coordinate direction.

Below is a command line execution of `mesh_adapt` using many of the options above that can be executed on a workstation. In this case we are running in parallel using eight processors, with the base mesh automatically decomposed using the `auto-decomp` option. The option to re-combine the refined mesh is disabled. In addition, the `respect_spacing` is enabled by default.

```
  launch -n 8 mesh_adapt --refine=DEFAULT \
    --input_mesh=cube_BL.g --output_mesh=cube_R1.g \
    --ioss_read_options="large,auto-decomp:yes" \
    --ioss_write_options="large"
```

The "launch" command can be replaced by the "sierra" command when running on large HPC platforms. In this case additional options are needed to specify the queue time limit and account (WCID).

The screen output is included below. Key features include the memory usage (both initial, after reading the input mesh, and after performing the refinement), a table indicating element counts before and after refinement, and timing information.

```
INFO: ioss_read_options= large,auto-decomp:yes ioss_write_options= large
PerceptMesh:: opening cube_BL.g

Using decomposition method 'RIB' on 8 processors.

MEM: 130.6 M [hwm_tot] 16.64 M [hwm_max] initial memory after opening input mesh.
Refinement pass # 1 start...


                       Refinement Info

|                      | Original        | New             |
| Element Topology Type | Elements  Nodes | Elements  Nodes |
- --------------------- - --------  ----- - --------  ----- -
|          Hexahedron_8 |      150        |     1200        |
|       Quadrilateral_4 |       30        |      120        |
|                Totals |      180    952 |     1320   1573 |


P[0]  AdaptMain:: saving mesh...
Saving mesh cube_R1.g ... done
P[0]  AdaptMain:: mesh saved
MEM: 208.8 M [hwm_tot] 26.46 M [hwm_max] final memory after refining mesh.
P[0, 8]  max wall clock time = 0.0448601 (sec)
P[0, 8]  max cpu  clock time = 0.043994 (sec)
P[0, 8]  sum cpu  clock time = 0.349947 (sec)
```

In this example, the mesh is a cube created using Cubit, with variable spacings in each coordinate direction. The `respect_spacing` option enables refinement that correctly locates the new nodes along the same spacing as the original mesh, in all three dimensions.

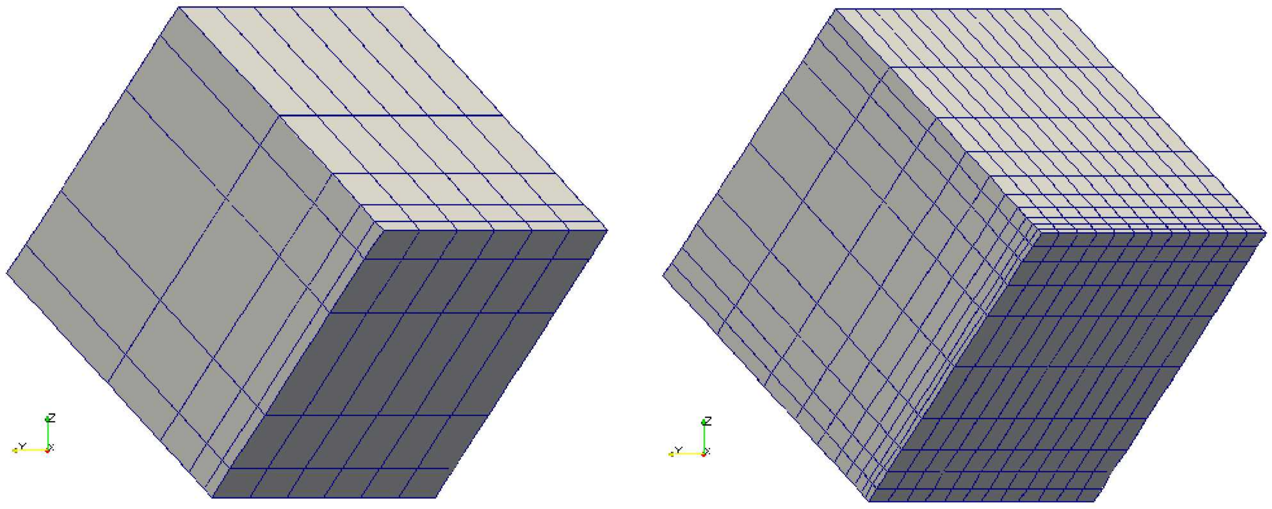


Figure 2.1: Example of basic UMR: cube with non-uniform edge spacings. Base mesh on left and first uniform refinement on right.

2.15 Example: block UMR usage

In this example we demonstrate how to refine a single element block within a tet mesh containing two element blocks. For the first

```
  launch -n 8 mesh_adapt --refine=DEFAULT \
    --input_mesh=two_blocks_tet4.g --output_mesh=two_blocks_tet4_R1.g \
    --ioss_read_options="auto-decomp:yes" --number_refines=1 \
    --blocks=1 --use_transition_elements=1 --respect_spacing=0
```

Further refinements are possible by incrementing the value of `--number_refines` and changing the name of the `--output_mesh` argument.

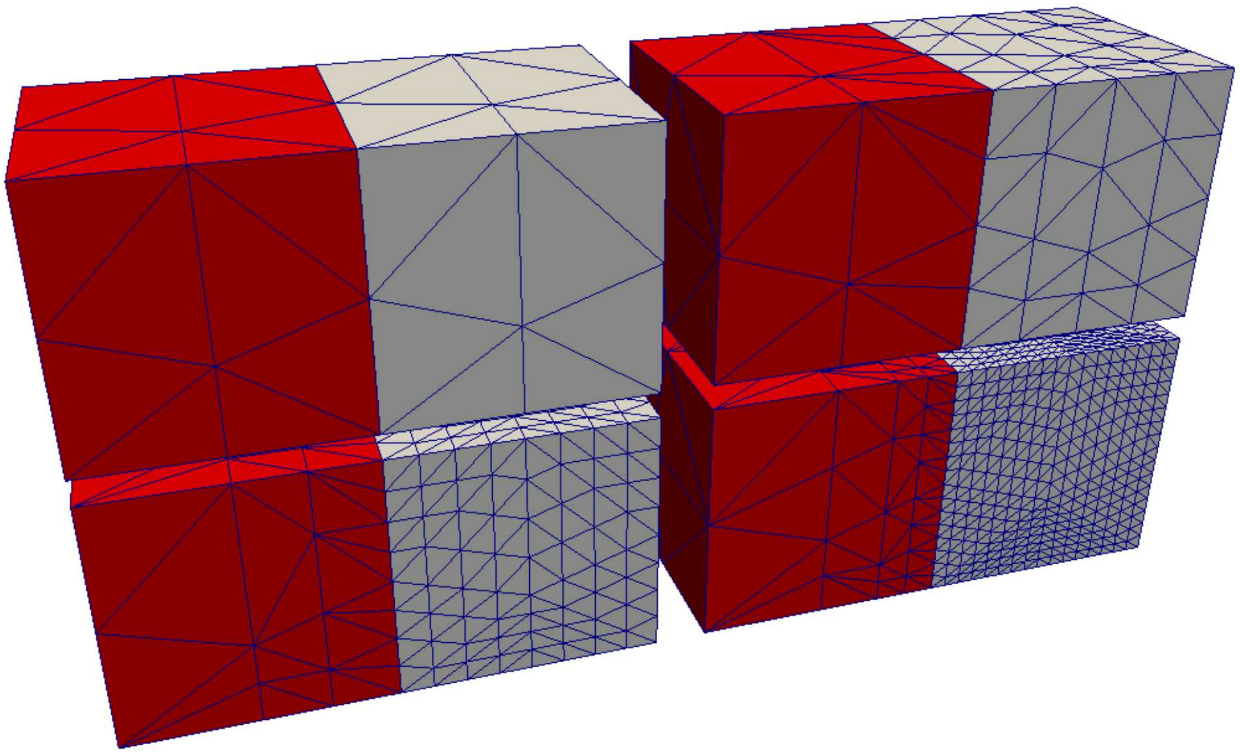


Figure 2.2: Example of block UMR: two tet blocks. Base mesh upper left with three refinements of the right block shown.

To illustrate the mesh quality more clearly, we show the refinement history of a surface that spans both blocks. Percept avoids poor quality elements by fully refining adjacent transition elements to the refined blocks on subsequent refinement passes as seen in Figure 2.3.

The method correctly preserves external sidesets and nodesets, even those spanning multiple blocks, not all specified. The exception is internal sidesets at the intersection of two blocks. For other types of elements such as hexes, the elements adjacent to the refined blocks will be converted to transition elements consisting of pyramids and tets.

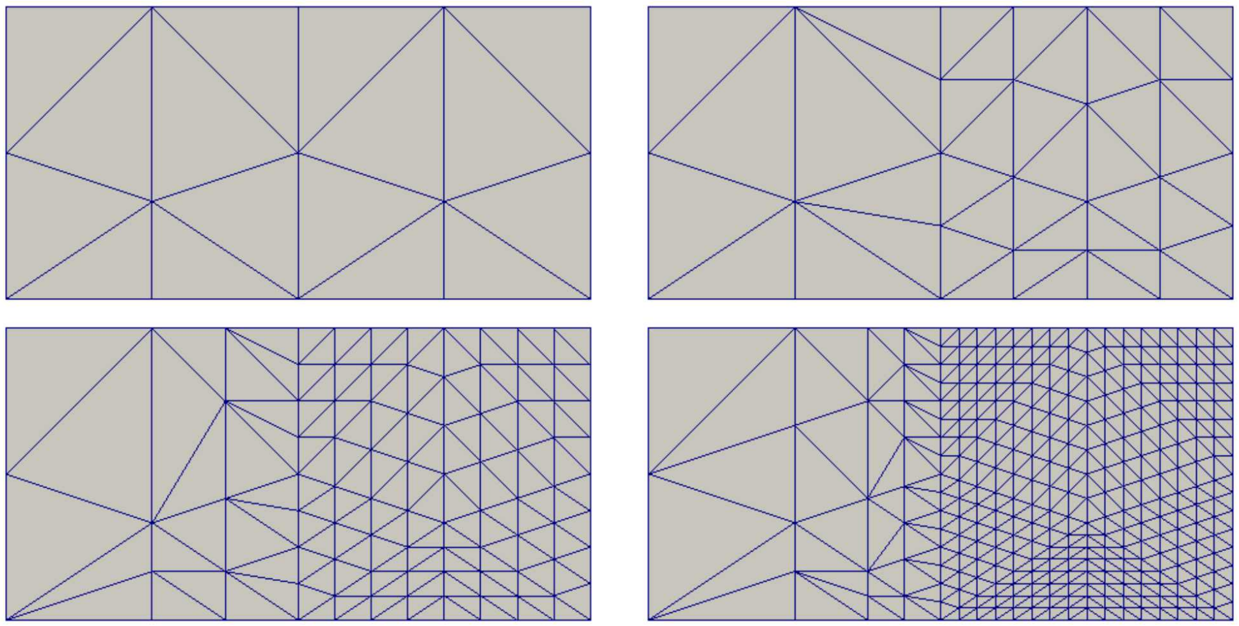


Figure 2.3: Example of block UMR: two tet blocks. Refinement history for surface spanning two blocks.

Chapter 3

Uniform Mesh Refinement with Geometry

When refining a mesh, it is important to resolve the underlying geometry as new nodes are created on geometric surfaces. Percept supports several ways to refine mesh to a geometry. These include support for CAD geometries as well as mesh-based geometries that are created solely from the input mesh file.

3.1 Refine to CAD: openNURBS

Percept currently supports CAD geometries using the openNURBS format. These geometry files use the .3dm file extension. In order to reference the geometry file, the `--input_geometry=` option must be specified.

Additionally, Percept requires the input mesh to be modified to contain additional data to provide associations between surfaces in the mesh and the corresponding geometric entities. This is currently done using a pre-processing step in Cubit and will be discussed in section 3.2.

An example command line execution is illustrated below using four processors.

```
launch -n 4 mesh_adapt --refine=DEFAULT \
  --input_mesh=speaker.g --output_mesh=speaker_R1.e \
  --input_geometry=speaker.3dm --respect_spacing=0 \
  --ioss_read_options="auto-decomp:yes"
```

For a second level of refinement, the output mesh can be re-used as the input mesh as follows:

```
launch -n 4 mesh_adapt --refine=DEFAULT \
  --input_mesh=speaker_R1.e --output_mesh=speaker_R2.e \
  --input_geometry=speaker.3dm --respect_spacing=0
```

The output of this example is shown in Figure 3.1 where the original geometry was meshed coarsely with 9974 linear tet elements.

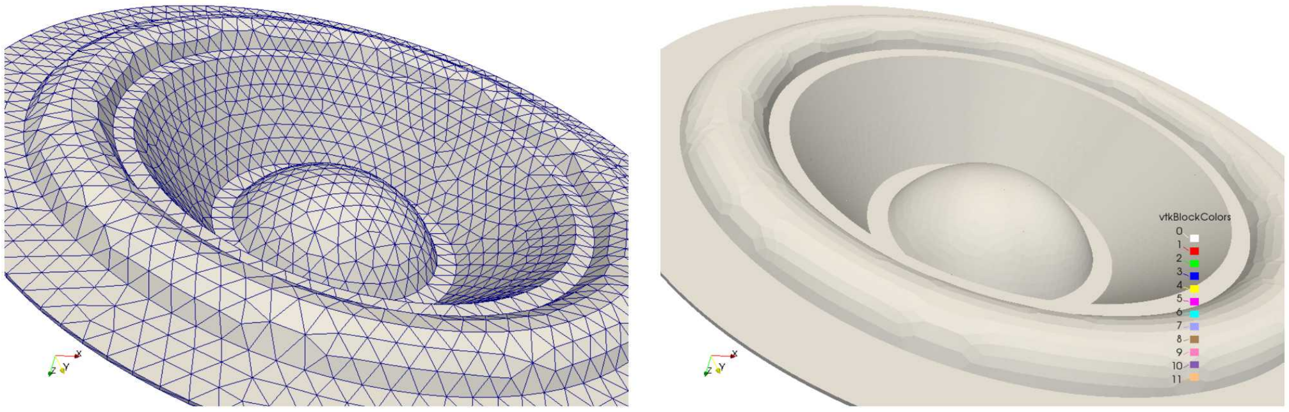


Figure 3.1: Example of UMR with geometry from openNURBS: base mesh on left and first uniform refinement on right.

3.2 Generating openNURBS Geometry Files

Currently Percept requires the user to export the ExodusII mesh from Cubit in order to create the .3dm file containing the openNURBS geometry. This is done within Cubit using the command

```
refine parallel fileroot "meshname" over no_exec
```

where “meshname” is the file prefix of the output files. In this example, Cubit will produce two files, “meshname.3dm” and “meshname.in.e”. The second file is an ExodusII mesh that contains additional element blocks. The blocks with names beginning with “TBC_” contain beam elements that indicate any edge that is on a curve in the geometry. The blocks with names beginning with “TBST_” contain shell elements that represent any surface facet that is on a curved surface in the model. Figure 3.2 illustrates these blocks for the example in section 3.1.

It is possible to convert these element blocks into nodesets by processing the ExodusII output through mesh_adapt. An example of this using the `--convert_geometry_parts_OpenNURBS=` option would be

```
mesh_adapt --input_mesh=speaker.in.e --output_mesh=tmp.e \
  --input_geometry=speaker.3dm --refine=DEFAULT \
  --respect_spacing=0 --number_refines=1 \
  --convert_geometry_parts_OpenNURBS=speaker.g
```

An example of the nodesets resulting from beam elements is shown in Figure 3.3

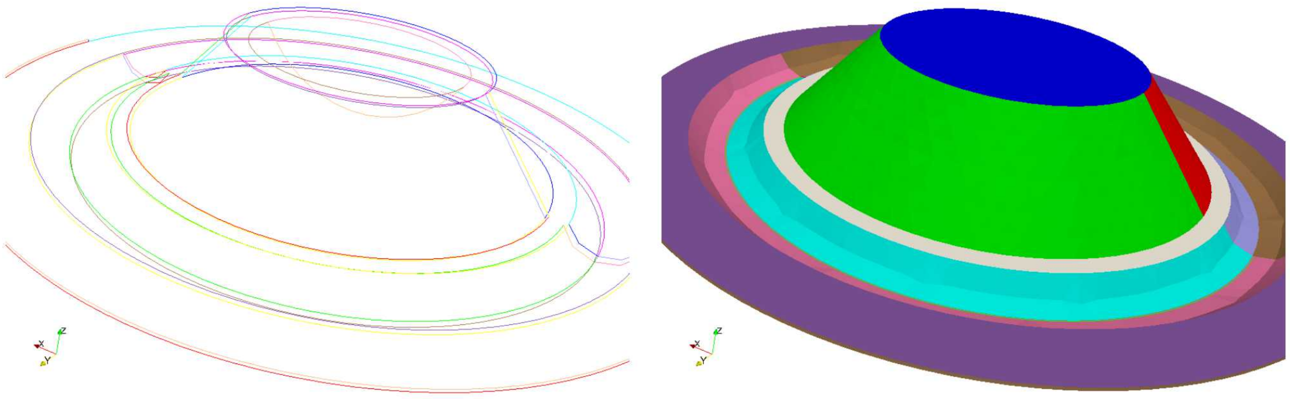


Figure 3.2: Example of additional element blocks created for openNURBS geometry: beam elements on left and shell elements on right.

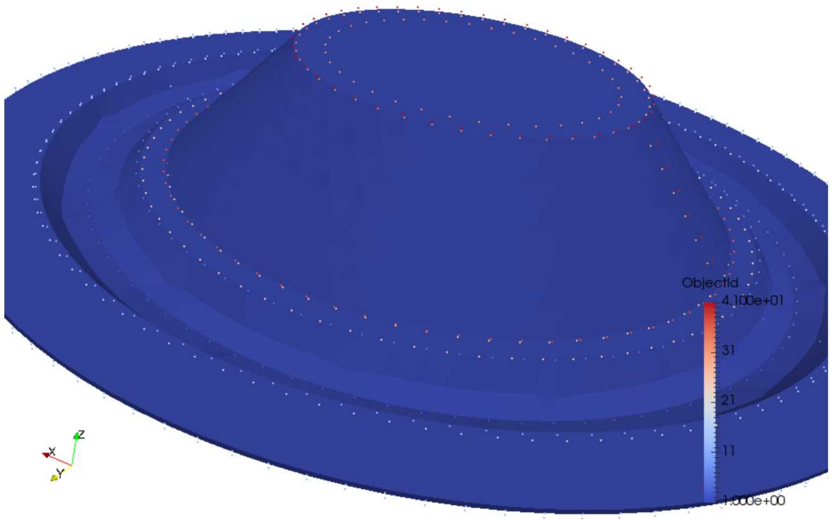


Figure 3.3: Example of additional nodesets created for openNURBS geometry from beam elements.

3.3 Refine to Mesh-Based Geometry

Percept can also generate a mesh-based geometry (MBG) based solely on an ExodusII mesh. In this case, the geometry is approximate, but can improve the accuracy of the new surface nodes on the refined meshes. The MBG is based on local smoothing of the surface curvature to generate a piecewise polynomial representation of the actual curved surface geometry.

This capability requires the mesh to contain sidesets. Any surfaces that are not contained within a sideset will be ignored by the code.

In order to activate MBG, the user must create an additional text input file with the .yaml extension. An example would be:

```
globalAngleCriterion: 135 #  degrees = 180 - 45
surface_sets:
  - set_1: [surface_1]
  - set_2: [surface_2,surface_3,surface_4]
QA:
  activate: yes
  file: qa
  num_divisions: 2
```

The first line indicates a feature angle used to detect sharp edges. This helps to avoid generating a smooth geometry where a sharp edge exists. The default value is 135 degrees - a value of 180 will result in all surface edges included, a value of 0 will result in no surface edges included.

The second part defines grouping of sidesets into surface sets. This enables curvature of edges (boundaries of sidesets) to be detected properly when sidesets have nonempty intersection. Finally, there are opportunities to visualize the edges detected by the algorithm using the QA section. This produces an ExodusII output file that can be inspected to see if the edge seams are correct.

For an example, we apply the MBG approach to uniform refinement of the example in the previous two sections. The identified mesh-based geometry of this example is shown in Figure 3.4 where the original geometry was meshed coarsely with 9974 linear tet elements. An example command line execution is illustrated below using four processors. The first step produces a new mesh which contains additional parts and fields to store both the geometry representation as well as the association with sidesets.

```
  launch -n 4 mesh_adapt --respect_spacing=0 \
    --refine=DEFAULT --input_mesh=speaker.g \
    --output_mesh=speaker.withgeom.g --number_refines=0 \
    --fit_3d_file=speaker.yaml \
    --ioss_read_options="auto-decomp:yes"
```

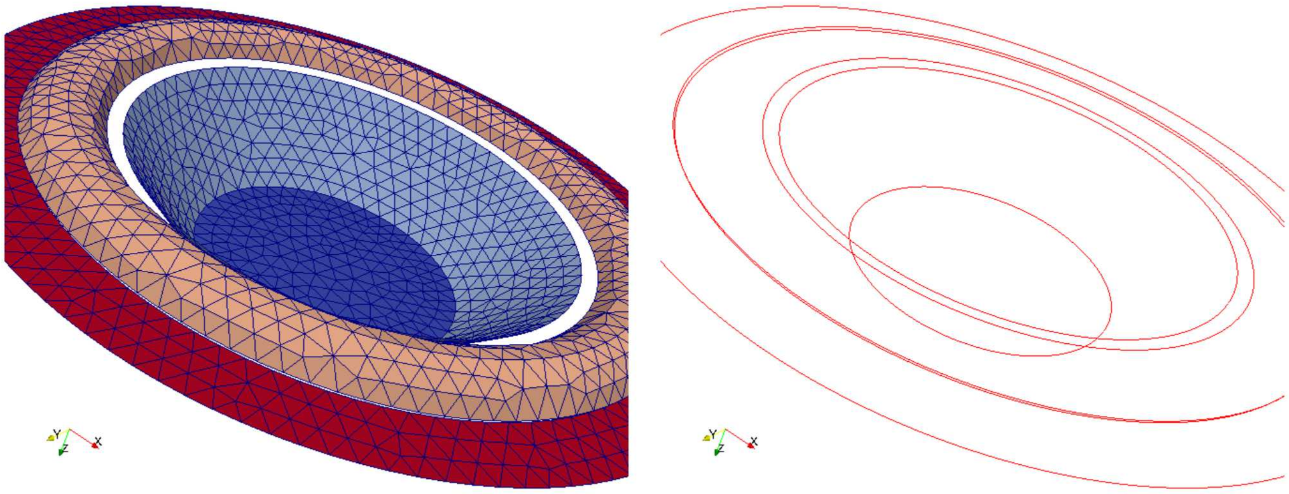


Figure 3.4: Example of UMR with geometry from mesh-based geometry: base mesh on left with sidesets highlighted and identified edge seams on right.

```
launch -n 4 mesh_adapt --respect_spacing=0 \
  --refine=DEFAULT --input_mesh=speaker.g \
  --output_mesh=speaker_R1.g --number_refines=1 \
  --input_geometry=speaker.withgeom.g --smooth_geometry=0 \
  --ioss_read_options="auto-decomp:yes"
```

The input .yaml file is referenced using the `--fit_3d_file=` option and the MBG ExodusII file is referenced using `--input_geometry=` option. Please note that the input mesh is the same for both commands.

For a second level of refinement, the output refined mesh can be re-used as the input mesh as follows:

```
launch -n 4 mesh_adapt --respect_spacing=0 \
  --refine=DEFAULT --input_mesh=speaker_R1.g \
  --output_mesh=speaker_R2.g --number_refines=1 \
  --input_geometry=speaker.withgeom.g --smooth_geometry=0
```

In Figure 3.5 we plot two levels of uniformly refined meshes using mesh-based geometry where it is clear that the geometry of the refined mesh conforms better to the true geometry than the faceted geometry of the coarse mesh. Finally in Figure 3.6 we plot some areas on the second refined mesh where no sidesets were present, in order to indicate that no refinement to any geometry was applied there.

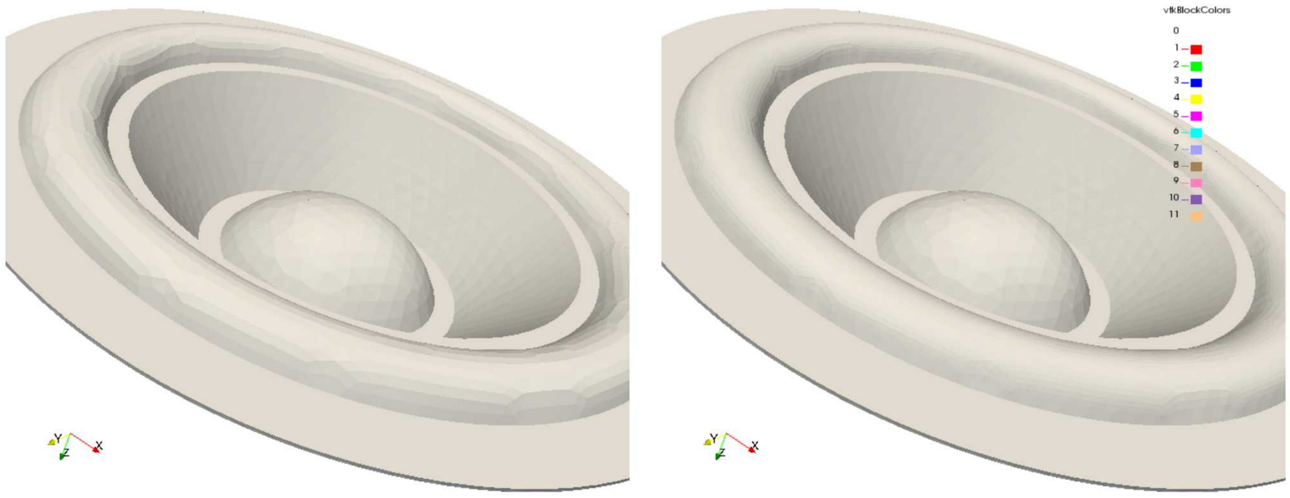


Figure 3.5: Example of UMR with geometry from mesh-based geometry: two levels of uniformly refined meshes.

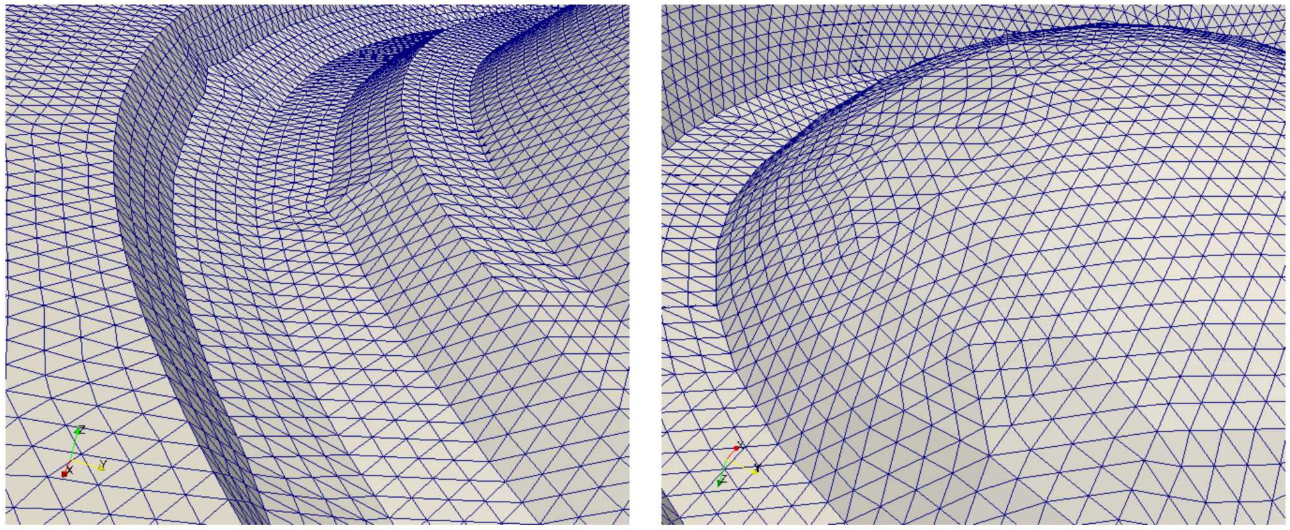


Figure 3.6: Example of UMR with geometry from mesh-based geometry: regions where no sidesets were present.

3.4 Example of refinement to CAD with smoothing

We conclude this chapter with a demonstration of geometry combined with smoothing. Below we are refining a half torus that has been meshed very coarsely. The CAD geometry is available which enables the new nodes to lie on the correct geometry. However, with smoothing we can move both the new nodes and the original nodes to improve the mesh quality.

```
mesh_adapt --refine=DEFAULT --respect_spacing=0 \
  --input_mesh=parallel_refine.in.e --output_mesh=out.e \
  --number_refines=1 --input_geometry=parallel_refine.3dm \
  --smooth_geometry=1 --smooth_surfaces=1
```

In Figure 3.7 we present three refined meshes based on no geometry, geometry and geometry with smoothing. The worst quality element without smoothing (and with geometry) had a value of 0.399 for scaled Jacobian. After smoothing this improved to 0.444.

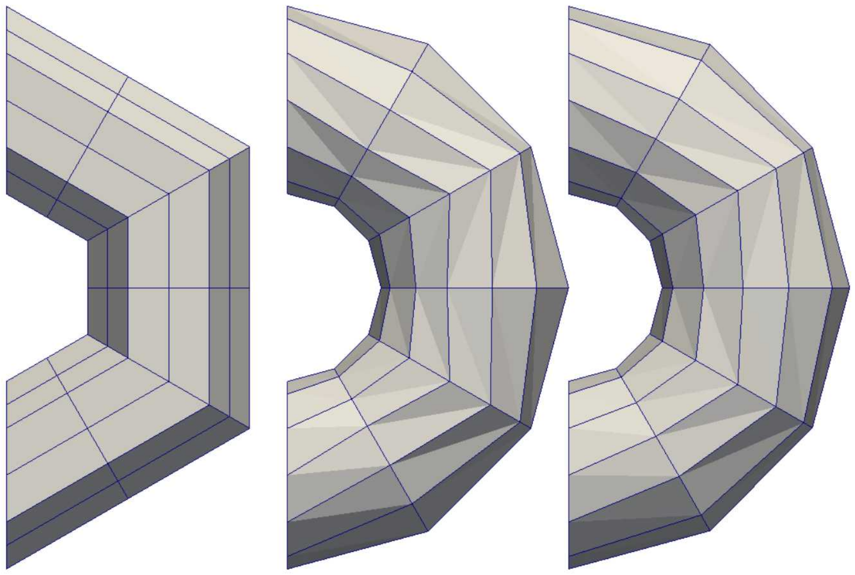


Figure 3.7: Example of UMR with CAD geometry and smoothing. Left mesh is refined without geometry or smoothing. Middle mesh is refined with geometry but no smoothing. Right mesh uses both geometry and smoothing.

This page intentionally left blank.

Chapter 4

Offline Adaptive Mesh Refinement

4.1 Overview of offline adaptive refinement

Percept provides a local adaptive mesh refinement interface that enables adapted meshes to be generated for analysis codes. We refer to this capability as "offline" adaptivity since the mesh adaptation process occurs outside the analysis code execution. This enables workflows where the analyst can run analysis, then adapt the mesh, followed by another analysis run (also referred to as "run-adapt-run").

The offline adaptivity workflow requires more interaction between the meshing and the analysis than is common for analysis using a single mesh. For example, suppose the analysis uses a base mesh called "mesh.g"; we will need some notation to refer to the subsequent adapted meshes. We assume that the mesh file name has a base name (here it would be "mesh") and a file extension (here it would be "g") which are connected with a dot "." character.

In offline adaptivity the base name of the mesh is required to end in zero. The adapted meshes generated will then end in increasing integers beginning with one. This is similar to the case with uniform refinement, where we need to number the meshes to reflect their position in the mesh refinement hierarchy. For example, we might have a sequence of meshes called "mesh0.g", "mesh1.g", etc. where increasing index in the base name means finer meshes.

The reason for this requirement is that Percept generates an additional family of meshes to contain the entire mesh hierarchy of elements from the base mesh to the finest refinement level (called “full mesh” here). These meshes contain the string "_ft" in the name to distinguish them from the meshes containing only the active elements (called “analysis mesh” here). Table 4.1 illustrates the naming conventions for an initial base mesh called “mesh0.g”:

The adaptive mesh refinement is based on an input Exodus mesh database that must contain an

refine level	analysis mesh	full mesh
0	mesh0.g	N/A
1	mesh1.g	mesh1_ft.g
2	mesh2.g	mesh2_ft.g

Table 4.1: Example of mesh naming conventions for offline adaptivity for two levels of refinement.

element scalar field of error indicator values. This field can be computed by an analysis code using an application-specific algorithm to estimate local discretization error. Alternatively, another analysis code such as Encore can be used to compute the error indicator using only the solution fields output from the analysis code, using for example a gradient or stress patch recovery algorithm.

Elements are marked for refinement/unrefinement based on the error indicator field using separate refinement/unrefinement treshold values. This works as long as the user can provide appropriate threshold values. An alternative approach that we suggest, is using a parameter on the growth of the number of elements to let Percept determine the refinement threshold automatically. For example, if the user set the growth parameter to be 1.2, then the refinement threshold would be computed iteratively to result in a new mesh with approximately 1.2 times the original number of elements or a 20 percent increase. Marking can also be limited to within a geometry region. Currently Percept supports rectangular boxes, cylinders, and spheres as the geometric regions.

In order to use offline adaptivity, we suggest using an additional input file (typically called "adapt.yaml"). An example file is shown below

```
error_indicator_field: error_indicator
marker:
  refine_fraction: 0.0
  unrefine_fraction: 0.0
  type: fraction
max_number_elements_fraction: 1.5
max_refinement_level: 5
do_rebalance: yes
```

The first line provides the name of the element error indicator field. The next section specifies options associated with the marking of the elements. The refine fraction is a number θ_r between zero and one that determines the refinement threshold to be $(1-\theta_r)$ times the max value of the error indicator field. Similarly, the unrefine fraction is a number θ_u between zero and one that determines the unrefinement threshold to be θ_u times the max value of the error indicator field.

In this example, the refine/unrefine fractions are both set to zero. This results in an unrefine threshold of zero and a refine threshold equal to the max value of the error indicator. When the error indicator field is nonzero (typical case), this prevents any elements to be marked for refinement or unrefinement. The next option in the example (max_number_elements_fraction) is a target multiplier insures that some elements are actually marked for refinement. This is done by iteratively adjusting θ_r until the estimated number of total new elements will hit the target multiplier times the number of original elements.

The maximum amount of refinements that can occur for an element in the original mesh is specified using the max_refinement_level parameter. The do_rebalance parameter is very important for insuring good parallel performance on the resulting adaptive meshes. When specified, this option enables re-partitioning of the mesh based on the new adapted mesh.

In order to limit the adaptivity to a geometric region, the bounding_region option must be

Option	Default Value
error_indicator_field	error_indicator
marker: refine_fraction	0.2
marker: unrefine_fraction	0.1
marker: type	fraction
max_number_elements_fraction	1.0
max_refinement_level	3
do_rebalance	no
max_marker_iterations	100

Table 4.2: Default values for offline adaptivity.

present. Examples are shown below (only one can be used during an adaptive refinement step):

```
bounding_region:
  type: sphere
  radius: 0.5
  center: [1.0,0.5,0.0]
bounding_region:
  type: cylinder
  radius: 0.75
  start: [1.0,0.0,1.5]
  end:   [2.0,0.0,1.5]
bounding_region:
  type: box
  start: [0.0,0.0,0.0]
  end:   [1.0,0.5,0.75]
```

4.2 Example: offline adapt for an unstructured triangle mesh

In this example we demonstrate offline adaptive refinement for a simple two-dimensional unstructured triangle mesh. The coarse mesh contains 14 elements and covers the domain $[0,2] \times [0,1]$. The error indicator for each element is interpolated to the centroid using the analytic function $e(x,y) \equiv \exp^{(y-x^2/4-x/2)^2}$. This function acts as a surrogate for an analysis code producing an error indicator for a steady state calculation.

The contents of the adapt.yaml file are shown below. Here we will run with two processors and enable rebalance. At each adaptive step we expect to get roughly twice the number of elements in the new mesh.

```
error_indicator_field: error_indicator
marker:
  refine_fraction: 0.0
  unrefine_fraction: 0.0
  type: fraction
max_number_elements_fraction: 2.0
max_refinement_level: 5
do_rebalance: yes
```

Sample screen output from the refinement of mesh2 into mesh3 is shown below using the run command

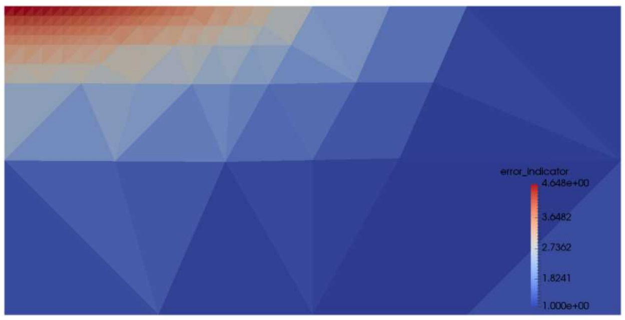
```
launch -n 2 mesh_adapt adapt mesh3.e out3.e
```

Here the number of elements increased from 112 to 208 or about a 1.86 multiplier. The rebalance did not improve the imbalance on this mesh which was about 43%. The table called “Marker binary search convergence history” displays the progress to determine the refinement threshold needed to achieve the target max_number_elements_fraction parameter.

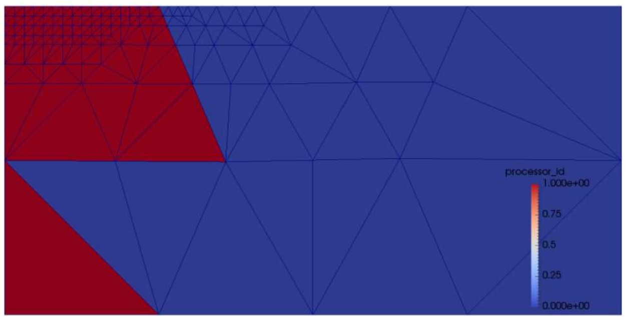
```
PerceptMesh:: opening mesh3_ft.e
PerceptMesh:: opening out3.e
Error indicator read from file.
Refinement initialized.
Marker binary search convergence history:
                     iter          current error                 min error
                       -1                      0                         0
                        0                    0.5                       0.5
                        1                   0.75                      0.75
                        2                  0.875                      0.75
                        3                 0.8125                      0.75
                        4                0.78125                   0.78125
                        5               0.796875                   0.78125
                        6              0.7890625                 0.7890625
                        7             0.79296875                 0.7890625

Marking complete.  Beginning refinement.
Refinement complete.
Refined mesh has 126 nodes and 287 elements.
Refinement complete.
  imbalance before= 1.43269 imbalance after= 1.43269
Saving mesh mesh4_ft.e ... done
Saving mesh mesh4.e ... done
Timings: max wall clock time = 0.297303 (sec)
Timings: max cpu  clock time = 0.151977 (sec)
Timings: sum cpu  clock time = 0.298955 (sec)
```

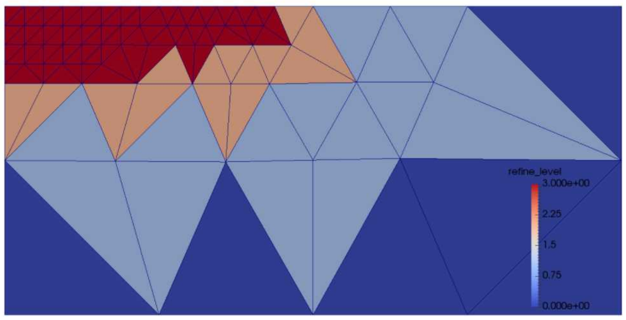
In Figure 4.1 we provide some plots of the adapted meshes. Figure 4.1(a) shows the distribution of the error indicator using the final adapted mesh. In Figure 4.1(b) we plot the processor decomposition on the final mesh, which has shifted to include more of the refined elements on proc 1 with most of the coarser elements on proc 0. The overall load imbalance on this mesh is only about 10%. Figure 4.1(c) illustrates the local refinement level, which indicates how many times an



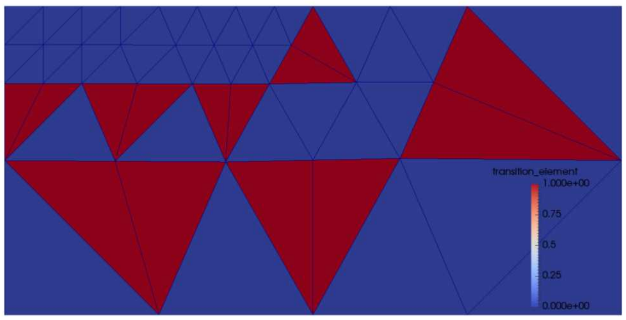
(a) error indicator on mesh4



(b) processor decomposition on mesh4



(c) refinement level on mesh3



(d) transition elements on mesh2

Figure 4.1: Example of offline adaptivity on a simple triangular mesh.

element has been refined from the original mesh. Here we are looking at the third refinement, so this value can be zero to three.

Finally in Figure 4.1(d) we highlight the transition elements, which are elements that do not refine using the standard pattern for triangles, which splits them into four child triangles. The triangles in red were all split into two by splitting only a single edge of the parent element.

This page intentionally left blank.

Chapter 5

Mesh Transfer Tool

Percept mesh transfer capabilities are provided through the `mesh_transfer` application code. Support is provided for transfer between two meshes, referred to as the source and destination meshes. Both meshes can have the same spatial dimension (2D or 3D), or the source mesh can be 2D axisymmetric with a 3D destination mesh. In the latter case, support for components of axisymmetric vector fields is provided.

To see all the options possible just run

```
mesh_transfer --help
```

The following sections describe the various options. Each option requires an equals (=) sign followed by the argument.

5.1 Source file (Required)

The `--src-file` option specifies the ExodusII file that contains the fields to be transferred. This file is assumed to be already decomposed when running in parallel.

5.2 Destination mesh (Required)

The destination mesh used in the transfer is specified with `--dst-mesh` option. This ExodusII file is assumed to be already decomposed when running in parallel.

5.3 Target file (Required)

The new ExodusII file containing the results of the transfer is specified with the `--target` option. When run in parallel, this mesh will be written in decomposed form, with no concatenation into a single file.

5.4 Source field

The source field is specified using the `--src-field` option. Currently only one field can be transferred at a time. These can be scalar or vector fields.

5.5 Source vectors for 2D axisymmetric to 3D transfers

In the 2D axisymmetric to 3D case, the source field can be a component of a vector field. We assume that the *y*-axis in the source mesh is the same as the *z*-axis in the destination mesh, and that the *x*-axis is the same in both.

The normal component (out of plane) scalar is specified using the `--src-rznvec` option. This field is transformed to a new vector field according to

$$u = -\alpha \sin(\theta), \quad v = \alpha \cos(\theta), \quad w = 0$$

where α is the value of the scalar field and θ is the angular location source/destination mesh.

The two in-plane components are specified as a vector using the `--src-rzpvec` option, and is transformed according to

$$u = \beta_1 \cos(\theta), \quad v = \beta_1 \sin(\theta), \quad w = \beta_2$$

where the components of the source vector are β_1, β_2.

5.6 Destination entity

By default, the entity type of the destination field is chosen to match the type of the source field (node or element). But the user can specify this type using the `--dst-entity` option and it can be different from the source field.

5.7 Destination name

By default, the name of the destination field is chosen to match the name of the source field. But the user can also specify this name using the `--dst-name` option.

5.8 Rotations and translations

The target file can be rotated and translated after the transfer occurs. This can be helpful if the target result is needed in a different position or orientation than the source model. The options for rotation angles are `--xrot`, `--yrot`, and `--zrot`; the translation components are specified using `--xtrans`, `--ytrans`, and `--ztrans`.

5.9 Repeated transfers and existing fields

The code is written to support repeated transfers. If the destination field already exists on the destination mesh, the transferred field values are summed in to the existing values and written to the target file. Values of all other fields in the destination mesh are preserved in the target file.

5.10 Example of mesh transfer usage

We present a small example that includes all of the above options. Here we have a 2D source mesh containing a scalar field “jetheta” and a 2D vector field “je”. These are components of a 3D axisymmetric vector field which we will call “jn”. There is also an element scalar field called “scalar”.

The vector field is assembled using two repeated calls to `mesh_transfer`. The first call will transfer the in-plane axisymmetric vector field “je” to a new 3D vector field called “jn” on an intermediate results mesh called out1.exo. We next transfer a scalar field called “scalar” using a second transfer call to a second intermediate file called out2.exo, preserving the intermediate values of the field “jn”.

Finally, the third call will transfer the scalar field “jetheta” as the out-of-plane component of an axisymmetric vector field to a final target mesh called out.exo. Because the destination mesh is the second intermediate mesh (out2.exo), the new transferred values of “jn” will be added in with the original values from the first transfer and the values of the field “scalar”. Options are added to the final transfer to apply a rotation and translation of the final target result.

```
mpirun -n 8 mesh_transfer \
  --src-file=q4_2d.e \
  --dst-mesh=h8_3d.g \
  --src-rzpvec=je \
  --dst-entity=node \
  --dst-name=jn \
  --target=out1.exo
  mpirun -n 8 mesh_transfer \
    --src-file=q4_2d.e \
    --dst-mesh=out1.exo \
    --src-field=scalar \
    --dst-entity=node \
    --dst-name=scalar \
    --target=out2.exo

  mpirun -n 8 mesh_transfer \
    --src-file=q4_2d.e \
    --dst-mesh=out2.exo \
    --src-rznvec=jetheta \
    --dst-entity=node \
    --dst-name=jn \
    --target=out.exo \
    --xrot=30 --zrot=45 \
    --xtrans=-1 --ytrans=-2 --ztrans=4
```

Sample screen output from the first transfer:

```
PerceptMesh:: opening q4_2d.e
PerceptMesh:: opening h8_3d.g
MeshTransfer: initializing transfer
MeshTransfer: performing transfer at time = 1
MeshTransfer: performing transfer at time = 2
MeshTransfer: performing transfer at time = 3
...
MeshTransfer: performing transfer at time = 8
```

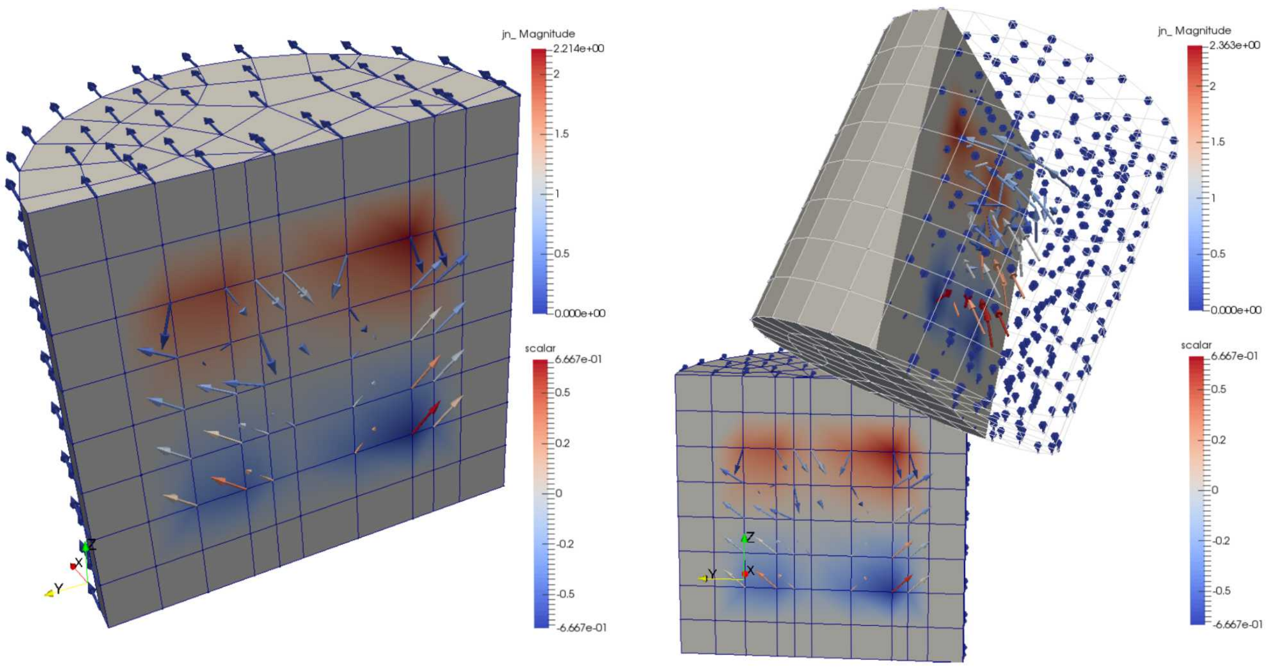


Figure 5.1: Example of mesh transfer at final time step: (left) cutaway of second intermediate mesh "out2.exo" and (right) cutaway of final target mesh "out.exo" shown along with "out2.exo" result.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)

v1.38

Sandia National Laboratories