2014 MeshKit Release

Mathematics and Computer Science Division

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free via DOE's SciTech Connect (<http://www.osti.gov/scitech/>)

Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: orders@ntis.gov

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: reports@osti.gov

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

2014 MeshKit Release

Rajeev Jain and Vijay Mahadevan
Mathematics and Computer Science Division, Argonne National Laboratory

September 30, 2014

SUMMARY

MeshKit v1.2 was released on July 25th 2014 as part of SIGMA¹ v1.0 (Scalable Interfaces for Geometry and Mesh-Based Applications). SIGMA consists of CGM, MOAB, Lasso and MeshKit developed and managed at Argonne National Laboratory. This release of MeshKit focuses on documentation and adding support for developing RGG (Reactor Geometry (and mesh) Generator GUI along with fixes and new enhancements to the RGG tools AssyGen and CoreGen. Several important and user required code enhancements and fixes are a part of this release. A new scheme for distributing the component assemblies forming the reactor core in parallel CoreGen has improved the overall computational time taken by CoreGen to assemble the reactor core from it's component assemblies. Verdict mesh quality library has been integrated and is being used by quad mesher and post mesh boundary layer algorithm. Development of interfaces to NetGen and GRUMMP tri mesher are under active development. Various reactor types such as ABTR, PWR, MONJU etc. were modeled using MeshKit. These models applied to individual and coupled physics solvers such as Nek5000, PROTEUS, Diablo and SHARP.

¹<http://www.sigma.mcs.anl.gov>

TABLE OF CONTENTS

Summary	i
Table of Contents	ii
List of Figures	iv
List of Tables	v
1 Introduction	6
1.1 Algorithms and Tools	7
1.2 Reactor Geometry (and mesh) Generator: RGG	9
2 Code Development	11
2.1 Build System	12
2.2 RGG Enhancement and Fixes	12
2.2.1 Introduction of Common Input File	12
2.2.2 New Parallel CoreGen Scheme	13
2.2.3 Material and Neumann Set Shifting in CoreGen	13
2.2.4 Fixes in SIGMA Components	14
2.2.5 Other MeshKit Fixes	15
2.3 Make watertight	16
2.4 Sweep Mesh Generation	16
2.5 Verdict Integration	18
2.6 Tri Mesh Generator	19
2.7 Quad Mesh Cleanup	19
3 Meshing Activities	21
3.1 ABTR	21
3.1.1 ABTR Single Assembly Model	21
3.1.2 ABTR 7-Assembly Model	23
Key Assumptions and Requirements	23
7-Assembly Core with Four Different Assemblies	24
7-assembly Core with All Fuel Assemblies	26
3.1.3 ABTR Full Core	27
3.2 Meshes Created For CoreGen Scalability Study	28
3.2.1 Fine ABTR Core Model	28
3.2.2 1/6 th VHTR Core	28
3.2.3 MONJU Reactor	30
3.3 VERA Benchmark Problems	32
4 Documentation	33
5 RGG GUI 1.0 Release and Collaboration with Kitware	33
6 Conclusions	38
Acknowledgments	39

References	39
Appendix A. Autotools Based Configure/Build/Installation of MeshKit	40
Appendix B. CMake-Based Builds: RGG Nuclear GUI Repository (SBIR: Kitware)	42
Appendix C. Input File for a Fuel Assembly with 24 Axial Regions	44
Appendix D. Input File for a Fuel-only Mini-core with Restraint Rings	46

LIST OF FIGURES

Figure 1 Algorithms available in MeshKit	7
Figure 2 Algorithms available in MeshKit (Contd.)	8
Figure 3 RGG Workflow	9
Figure 4. (a) CoreGen MeshOp di-graph. (b) Internal nodes created during setup-phase of CoreGen, shown in box. (c) 2-processor digraph for CoreGen MeshOp.	11
Figure 5. Common.inp file for AssyGen. These are core-level parameters, the same for all assemblies.	13
Figure 6. CoreGen input file demonstrating the use of “same_as” for 7-assembly ABTR.	14
Figure 7. Input and output geometry to make watertight algorithm.	16
Figure 8. Flowchart of surface mesh generation based on edge patch imprinting.	16
Figure. 9. A real part from Caterpillar: (a) a geometric model; (b) source surface meshes; (c) partitioned source and target surfaces by imprinting edge patches; (d) target surface meshes by morphing.	17
Figure. 10. Example of generating surface meshes and matching the source and target surfaces by imprinting edge patches: (a) geometric model; (b) partitioned source and target surfaces; (c) source surface meshes; (d) 3D-view surface meshes; (e) target surface meshes by morphing.	17
Figure 11. Surface mesh generation for crankshaft by imprinting: (a) geometric model with imprinted edge patches (denoted by blue curves); (b) target surface meshes by morphing.	18
Figure 12. Graph-based tetrahedral mesh generation.	19
Figure 13. Tetrahedral mesh created by NGTetMesher MeshOp.	19
Figure 14. Input and output to quad cleanup operation (a) original mesh. (b) final mesh after quad-cleanup.	20
Figure 15. (a) Detailed (nonhomogenized ABTR fuel assembly); (b) Three homogenized fuel assemblies showing the Above Core Load Pad (ACLP), Top Load Pad (TLP), and outlet plenum regions.	21
Figure 16. Section of metal fuel pin showing dimensions in cold condition.	22
Figure 17. (a) Control assembly absorber region containing two ducts; (b) Inlet of actual control assembly with homogenized control pins (magenta), two ducts (red), and half of inter-assembly sodium gap (green).	22
Figure 18. Minicore assembly: (a) configuration; (b) numbering scheme of assemblies (red).	24
Figure 19. Minicore mesh: (a) elements and blocks;	25
(b) internal assembly blocks used in PROTEUS; (c) power distribution.	25
Figure 20. Composition map for fuel-only minicore.	26
Figure 21. Assembly structural components: (a) restraint rings; (b) load pads on fuel assemblies;	27

(c) gap between assemblies and restraint rings ACLP (0.0235 cm, gray) and TLP (1.2025 cm, green).	27
Figure 22. ABTR full-core configuration: (a) with lines of constant logical I,J assembly regions; (b) with homogenized assemblies, outer covering, and restraint rings at TLP and ACLP locations	27
Figure 23. One-sixth of a VHTR core model generated by using CoreGen (left); closeup of assembly mesh in this model (right).	29
Figure 24. Log scale results from Table 2 vs. number of processors for 1/6 VHTR reactor core.	30
Figure 25. Full-core MONU reactor; closeup area in red rectangular region is highlighted from left to right.	30
Figure 26. Log scale results from Table 3 vs number of processors plot for MONJU reactor core.	31
Figure 27. Simple pincell model. (left: coarse, right: refined).....	32
Figure 28. 17x17 PWR assembly model.....	32
Figure 29. Pin being edited using RGG's pin cell editor.	34
Figure 30. Simple duct being designed that consists of two materials. One material forms the walls of the subassembly, while water is assigned as the inside material.....	35
Figure 31. Example reactor assembly in RGG	35
Figure 32. A simple hexagonal reactor core modeled in RGG.	36
Figure 33. An example of a nuclear reactor core mesh generated using RGG.	37
Figure 34. 199 Assembly ABTR created using RGG.	37
Figure 35. A PWR reactor core with five different types of assemblies modeled using RGG.	38

LIST OF TABLES

Table 1. Properties of input and output mesh to the quad cleanup algorithm.....	20
Table 2. CPU time in minutes and maximum memory in gigabytes used for 1/6 VHTR core.....	29
Table 3. CPU time in mins and maximum memory in GB used for MONJU core	31

1 Introduction

The MeshKit library provides a robust parallel infrastructure for developers and a scheme for algorithm users for various mesh generation functionality. It is implemented in C++ and provides a traditional C++-based API for interactions with other codes. A Python interface is also provided for interactive access to the library. MeshKit relies on geometry and mesh libraries developed as a part of the SIGMA (Scalable Interfaces for Geometry and Mesh-Based Applications) [1] libraries developed at Argonne National Laboratory; SIGMA v1.0 was released on July 25, 2014, and MeshKit v1.2 was a part of SIGMA v1.0 release. The SIGMA component CGM (Common Geometry Module) [2] provides functions for constructing, modifying, and querying geometric models in solid model-based and other formats. CGM can evaluate geometry from two underlying geometry engines: ACIS [3], with preliminary support for Open Cascade-based [4] version. Finite-element mesh and mesh-related data are stored in the SIGMA component MOAB (Mesh-Oriented database) [5]. MOAB provides query, construction, and modification of finite-element meshes, plus polygons and polyhedra. Various options are available for writing and visualizing the final meshes produced by meshing algorithms. MOAB uses an HDF5-based file format, which can be visualized by using a ParaView plugin that is implemented by the MOAB library. The Visit visualization tool can also be configured and built with MOAB to provide a similar import capability. The overall goal of MeshKit is to develop a complete package for generating reactor core models, while at the same time being general enough to serve as a platform for geometry and mesh generation research and tool development. External users from United States (ORNL, GE, Westinghouse), Scotland (University of Glasgow), Germany (SIMSCALE, University of Stuttgart) and China (Xiangtan University) are now using MeshKit.

The MeshKit library uses a graph-based process for specifying the overall meshing approach, with graph nodes representing meshing and other operation, and with graph edges as dependencies between those operations. The graph-based approach supports the traditional BREP-driven (BREP stands for boundary representation) meshing process, which usually proceeds by meshing BREP entities in increasing topological dimension, first with vertices, then edges, and so on. However, a graph-based process also enables meshing tasks not possible with a strict BREP-based approach. First, not every meshing process needs or has a geometric model representing the entire domain to be meshed. The best example is the Reactor Geometry (and mesh) Generator (RGG) tool, in which individual assembly types have geometric models but are then copied/moved into a lattice of assembly models forming a reactor core. Second, a meshing procedure may not involve only a once-through meshing of each BREP entity; again, RGG is a good example, in which the first part of the process involves meshing BREP models but the last step involves copy/moving mesh subsets into a larger core lattice. Third, the procedure-driven approach to meshing represented by most CAD-based meshing tools fails to capture the parallelism and dependency structure that can be found in most meshing problems (including BREP-based ones); representing and exploiting this richer structure provide more flexibility while still being applicable to BREP-based problems.

Design details and a description of all the algorithms available in MeshKit were given in our FY13 milestone report [6]. During the past year several important algorithms and applications have been developed using MeshKit. The infrastructure provided by MeshKit is robust and has been applied in various serial and parallel meshing related activities. This report is structured as follows. In the remainder of this section we list of all the algorithms currently available and also describes the RGG tool. Section 2 deals with code development over the past fiscal year. Mesh generation activities including scalability studies of parallel-enabled RGG are given in Section 3. Section 4 provides the details of MeshKit documentation work. Collaboration with Kitware for RGG GUI development is highlighted in Section 5. The report is concludes in Section 6 with an outline of future work.

1.1 Algorithms and Tools

The SIGMA v1.0 libraries provide robust infrastructure for developing geometry and mesh algorithms and applications. Indeed, in a short span of time, several algorithms and applications in MeshKit have been developed and published. MeshKit's algorithms that are described in separate papers published elsewhere has been listed here.

Traditional Native Algorithms	Mesh Handling/Modification	Support Algorithms
<ul style="list-style-type: none"> • UNSTRUCTURED MESHING: <ul style="list-style-type: none"> • <i>2D Quad Mesher: Jaal QuadMesher</i> • <i>OneToOneSweep</i> • STRUCTURED MESHING <ul style="list-style-type: none"> • <i>Embedded Boundary Mesher: EBMesh</i> • <i>Block Mesher or SCDMesh MeshOp</i> 	<ul style="list-style-type: none"> • Post Mesh Boundary Layer Generation: PostBL • Quad Mesh Cleanup • MergeMesh 	<ul style="list-style-type: none"> • Interval Matching Algorithm: IntervalAssignment • Mesh-Based Geometry Modifications: MBGeomOp, MBSplitOp, MBVolOp
(a)	(b)	(c)

Figure 1 Algorithms available in MeshKit

Algorithms available in MeshKit can be categorized in several ways, such as the dimension of entities, type of mesh elements, or application. Here, we categorize the algorithms based on MeshKit usage. Traditional meshing algorithms are 2D and 3D algorithms for developing tri/tet and quad/hex meshes. Figure 1(a) lists some of the primitive algorithms developed in MeshKit, for structured mesh generation, an embedded boundary mesher [9] and block mesher were developed. For unstructured mesh generation, a quad mesher [7] and one-to-one sweep [8] were developed. Other algorithms required for traditional meshing were integrated as external algorithms (Figure 2(b)) by leveraging

existing state-of-art existing meshing packages. Several traditional algorithms use helper algorithms listed in Figure 2(f). Papers have been published in peer-reviewed journal that include the details results and implementation details of quad mesher, one-to-one sweep and embedded boundary meshing. The mesh modifications algorithms listed in Figure 1(b) were also developed from scratch in MeshKit; implementation details along with results for post mesh boundary layer and interval matching algorithms have been published last year [10] [11], respectively. Supports algorithms that are used by several other algorithms are listed in Figure 1(c). Interval matching algorithms works closely with all other algorithms in MeshKit to assign the sizes on geometric entities when meshing a complicated geometry. Mesh-based geometry modification operations listed in Figure 1(c) were developed for building hexahedral meshes and associated geometry of ice sheets models from latitude and longitude data obtained from satellites.

Integrated Tools	External Algorithms	Helper Algorithms
<ul style="list-style-type: none"> •RGG: •<i>Nuclear Reactor Modeling: Assembly Geometry Creation Tool: AssyGen</i> •<i>Nuclear Reactor Modeling: Core Mesh and Geometry Creation Tool: CoreGen</i> •Sealing Facet Based Geometry: MakeWaterTight 	<ul style="list-style-type: none"> •Triangle •CAMAL/CUBIT •Tri Mesher •Quad Mesher •Tet Mesher •NetGen •Tri Mesher •Tet Mesher •Mesquite •Gmsh 	<ul style="list-style-type: none"> •TFIMapping •QSlimMesher •Vertex Mesher •Edge Mesher •Extrude Mesh •Copy Mesh/Geom Operation •Set Operations: Copy/Expand/Extrude
(d)	(e)	(f)

Figure 2 Algorithms available in MeshKit (Contd.)

Tools that are part of MeshKit are listed in Figure 2(d). The RGG tools AssyGen and CoreGen are described in Section 1.2 and developments in make watertight algorithms used for sealing facet-based geometries are reported in Section 2.3. Tools developed using MeshKit library may be a part of MeshKit or exist as an external library; all these algorithms and tools mentioned in Figure 1 and Figure 2 adhere to the graph-based design that MeshKit uses. MeshKit takes advantage of other open and closed source meshing libraries by providing a consistent interface. Interface to Gmsh and the NetGen tri mesher library are currently under development and in the testing phase. Several traditional algorithms use helper algorithms listed in Figure 2. Helper algorithms such as Vertex and Edge mesher are used by higher-dimensional algorithms and are often automatically created by the graph-based design of MeshKit. The RGG tool uses copy/move/merge/extrude algorithms and set operations for copy/expand/extrude tag specified during the copy/move/extrude process respectively.

1.2 Reactor Geometry (and mesh) Generator: RGG

A nuclear reactor core consists of uranium fuel rods, instrumentation rods, control rods, ducts, coolant, grid spacers, load pads, restraint ring systems, and other supporting structure. Although these core models are quasi-2.5-dimensional, modeling of interassembly gaps forming the whole core model from the component assemblies can get cumbersome and difficult to manage by using scripts or traditional meshing packages. Creating a fully detailed reactor core model with a few million elements can take a few days. Moreover, constructing these models on typical desktop computers can require large runtimes, and for large models such efforts may not succeed at all because of memory constraints. Analysts require a parallel modeling tool that can generate large core models in a few hours and can simplify the overall process of generating reactor core models.

RGG methodology involves three steps; the overall workflow is highlighted in Figure 3. First, AssyGen reads an input file describing a reactor assembly lattice and generates an ACIS or OCC-based geometry file, along with a template script for generating a mesh for the assembly using the Cubit meshing toolkit [12]. For the Core Mesh shown in Figure 1, AssyGen was run twice to generate the two types of assemblies shown. AssyGen supports several features, such as axial numbering material and boundary conditions, sectioning, rotation, and information about location of pins. These have been described in a paper presented at the International Meshing Roundtable [13].

The second step is meshing, where the user may choose to perform meshing using the Cubit mesh script generated by AssyGen or using meshing algorithms in MeshKit. The RGG GUI app developed by Kitware Inc. is available for download from the website: <http://www.computationalmodelbuilder.org/download/>

Figure 3 RGG Workflow

A single reactor assembly may comprise several thousand geometric regions if one considers the materials and axially varying properties of the assembly. One possible approach for parallelizing AssyGen involves having each processor create a region based on the location of the pin it obtains from the master processor. The problem with this approach is that the subtraction of regions from the outermost duct is a serial process. Because of the unavailability of parallel subtraction of geometry from components, this feature has not been developed. Instead, we are developing in MeshKit a parallel meshing algorithm that partitions geometry created by AssyGen and meshes these chunks in different processors to create the assembly mesh.

Once all the assembly and core input files are created, the third step of the workflow occurs: the CoreGen program is invoked. This tool reads an input file describing the reactor core arrangement and generates the reactor core mesh or geometry from its component assemblies. CoreGen uses the CopyMesh, ExtrudeMesh, CopyGeom, and MergeMesh algorithms in MeshKit. Figure 1 shows the two assembly meshes and an interstice mesh file that form a 19-assembly reactor core. A makefile is generated by CoreGen to automate the whole process from geometry creation. This makefile can be invoked in parallel by using the “-j” option of the “make” command to run the geometry creation and mesh generation of individual assemblies in different CPU cores. Thus, while the creation of individual assembly geometry and the meshing of individual assemblies are not parallel, the generation of all assemblies needed for a given core is parallelized, by using the basic parallel execution facilities of the standard make process. Since complicated reactor cores often use more than ten different assembly types, and since hundreds of assembly meshes along with interstices meshes such as grid spacers and restraint rings form a complete reactor core mesh, the models can involve several billion mesh elements. Generation of such models in serial is a time-consuming process, and parallelization can therefore be a tremendous boon.

The parallel CoreGen algorithm comprises seven steps:

1. On each processor, read CoreGen input file, parse, and determine assembly copies assigned to this processor based on a round-robin distribution.
2. Locally read assembly meshes for assemblies determined in step 1.
3. Perform assembly copy/move operations assigned to this processor.
4. Perform local merge of on-processor mesh.
5. Perform parallel merge of mesh between processors.
6. Parallelize the expand/extrude/copy set (metadata) handling.
7. Save output mesh.

A simple graph constructed by the user for running CoreGen is shown in Figure 4(a). In the setup traversal phase, CoreGen adds two nodes, “CopyMesh” and “MergeMesh” (shown in the box), to the graph. In the execute phase, based on a user specified input file, the assembly meshes are copy/moved and then merged. After populating the material and

boundary conditions for the core model, CoreGen saves the desired core model. Note that in serial CoreGen, CopyMesh does the copy/move of all the assemblies. Consider the graph with two processors in Figure 4(c). All the operations—copy/move, merge, boundary/material conditions, setup, and mesh saving—are parallelized, thereby making CoreGen MeshOp truly parallel and enabling creation of large reactor models. Indeed, parallel-enabled CoreGen has been used to create large meshes such as MONJU, VHTR, HTGR, EBRII, and PWR for different reactor simulation codes. Speedup results for VHTR and MONJU reactors using the parallel version of CoreGen can be found in Section 3.4.

Figure 4. (a) CoreGen MeshOp di-graph. (b) Internal nodes created during setup-phase of CoreGen, shown in box. (c) 2-processor digraph for CoreGen MeshOp.

2 Code Development

This section focuses on the code development in MeshKit over the past year. We moved the repository from subversion version control system to git version control system on Bitbucket, which is a free web-hosting service for projects that use git. Git is a relatively new distributed revision control and source code management system that has already become the most widely adopted version control system for software development. Total time required and the quality and efficiency of code development have improved after moving to git. We have spent considerable effort cleaning up the code (removed all warnings), the library is tested with various configurations of dependencies and compiler nightly on our buildbot system [14]. The error messages and warnings are reported using this system. This system is very helpful in ensuring the robustness and workability of the code. We have also added several new builds and compilers, including the clang compiler (used on Mac OSX), to the suite of builds that are run nightly.

Discussed in this section are efforts to simplify the overall build system, RGG enhancements and fixes, make watertight algorithm, sweep mesh generation, verdict integration, trimesh generation and quad mesh cleanup.

2.1 Build System

MeshKit has three mandatory dependencies—CGM, MOAB, and Lasso (each having its own dependencies, such as HDF5, NetCDF, libCCMIO, Zoltan, CUBIT, and OpenCascade)—and several options dependencies, including Triangle, CAMAL, NetGen, IPOPT, MPI, and Mesquite. Because of these dependencies, the packaging and build system for applications using MeshKit can get cumbersome. We have two efforts under way to make these steps easy for the end user: CMake and Autotools.

MeshKit v1.2 supports serial CMake-based builds. The CMake-based build system also allows the user to build an RGG GUI App developed by Kitware Inc. and all the SIGMA tools: CGM, MOAB, Lasso, and MeshKit. It provides options to use system installed or perform a fresh download and install for all dependencies. This effort is in collaboration with Kitware, the makers of the CMake build system. We also have a script tailored to download and compile SIGMA v1.0 from scratch.

All SIGMA tools from their inception have used the Autotools build system, and it continues to be supported. We are also working on scripting the Autotools build system to make it easier for the user to build and install all dependencies with one command. See Appendix A and B for details on building MeshKit using Autotools and CMake, respectively.

Our build systems provides very efficient infrastructure for development. When a collaborator or developer creates algorithms or tools in MeshKit, they submit a “pull request” that is peer-reviewed, tested and fine tuned before adding the code to the master branch. Development branches and pull requests can be tested using our buildbot system (<http://gnep.mcs.anl.gov:8010/waterfall?category=meshkit>) for a variety of configurations of MeshKit.

2.2 RGG Enhancement and Fixes

Various enhancements and bug fixes were applied to the RGG code. The following subsections cover the major developments incorporated in this release.

2.2.1 Introduction of Common Input File

Nuclear reactor cores usually consist of multiple assemblies and each assembly has several properties that remain same for all the assemblies. Our experience has shown that most of the AssyGen input file (discussed in Section 1.2) has several common variables such as mesh type, geometry engine, geometry type, merge tolerance, edge interval, radial mesh size, height of ducts and axial mesh size. This common.inp file is read by default along with each assembly input (see Figure 5).

MeshType Tet	! Specifies the mesh type
GeomEngine ACIS	! Specifies geometry engine
GeometryType Rectangular	! Shape of assembly geometry
TetMeshSize 2.0	! Size of tetrahedral mesh
CreateSideset No	! Turn on/off boundary creation
MergeTolerance 1e-6	! Merge tol for nodes b/w assemblies
Info on	! Generate CSV with pin locations
END	! Marks the end of this file

Figure 5. Common.inp file for AssyGen. These are core-level parameters, the same for all assemblies.

2.2.2 New Parallel CoreGen Scheme

During a parallel CoreGen run there is no master processor; each processor analyzes the overall workload and decides the mesh file(s) that it must load. During this task distribution phase three different cases (A, B, and C) may occur, where nA is the total number of different assembly mesh files, np is the number of processors, and nT is the total number of assemblies in the reactor core.

- A. $np < nA$
- B. $nA < np < nT$
- C. $np > nT$

In case A, each processor loads one or more assembly mesh files and solely performs the copy/move operation associated with that assembly for the entire core. In case B, some mesh files are loaded into multiple processors, and the copy/move task is distributed among them. Two schemes have been developed for selecting mesh files for copy/move task distribution. Scheme 1 is based on the frequency or the number of occurrence of that mesh file in the core; the assembly type that appears the most number of times in the core is assigned to multiple processors in a round-robin fashion. Scheme 2 is based on recursive analysis of load on each processor after each mesh file load has been decided. Scheme 2 leads to better load balancing of meshes among processors and faster overall completion time when compared with scheme 1. Another scheme based on the size of the mesh file and a combination of scheme 1 and 2 is under development. In case C, there are more processors than available work units, so only nT processors can take part in this parallel algorithm.

Results for the MONJU reactor are detailed in Section 3.4.3. In this case the round-robin distribution scheme leads to highly unbalanced load among processors, causing the problem to not fit in the available CPU memory. Therefore, the reactor uses scheme 2 for copy/move task distribution.

2.2.3 Material and Neumann Set Shifting in CoreGen

In several reactors such as the ABTR the only difference between different assemblies is the name or ids of material and boundary conditions. Creating different mesh files for all such assemblies is time consuming and wasteful of resources. Consider the ABTR core in

Section 3.3, in which all the 199 assemblies that form the core are different, each assembly having its own material and boundary condition. We therefore create only distinct assemblies, and we use the “same_as” CoreGen keyword to create the core model by duplicating the nodes and elements and shifting or reassigning the material and Neumann sets for the new assembly. Consider the 7-assembly input file shown in Figure 6, in which only two mesh files are created: `assm_mesh.exo` and `assm_mesh22.exo`; other assemblies, namely `1123000.exo` `1124000.exo`, are specified by using the “same_as” keyword. After specifying this keyword, the name of the assembly mesh file, material set start id, and Neumann set start id are specified.

```
! #####
! Homogenized 7-assembly ABTR Core Model, Demo of same_as keyword for
! shifting material and neumann set id's
! #####

Geometry Volume          ! 'Geometry' can be volume or surface
Symmetry 1               ! symmetry in the model
GeometryType HexFlat     ! Various hex types or rectangular
Assemblies 4 14.598      ! assembly mesh files and their pitch
assm_mesh.exo           1121
assm_mesh22.exo         1122
1123000.exo             1123   same_as   assm_mesh.exo   1123000   1123000
1124000.exo             1124   same_as   assm_mesh22.exo 1124000   1124000
Lattice 2
    1121 1124 &
    1122 1123 1121 &
    1122 1124
NeumannSet Top 1
NeumannSet Bot 2 ! Values 'top', 'bot' and 'side' are acceptable
OutputFileName sc.h5m
END
```

Figure 6. CoreGen input file demonstrating the use of “same_as” for 7-assembly ABTR.

2.2.4 Fixes in SIGMA Components

MeshKit is a part of SIGMA tools and relies on other SIGMA components: CGM, MOAB, and Lasso. Changes in these components may result in changes in the MeshKit interface, and similarly fixes or issues in MeshKit often cause a fix in one of these components. Various fixes and additions to SIGMA components that directly affect MeshKit include the following:

1. Addition of CMake support for CGM, MOAB, and Lasso.
2. Mesquite-related build fixes in CGM that were causing a failure in MeshKit build.
3. Fixes to OpenCascade subtraction; pin subtraction from duct of a rectangular PWR assembly that took 64 minutes dropped to 29 minutes after this fix.
4. Release of CGM v14.0 to use CUBIT v14.0; the previous release of MeshKit v1.0 used CUBIT v12.2.

5. Fixes to MOAB's CUBIT file reader that were causing failure in CoreGen.
6. Build system fixes to Zoltan mesh partitioner that was causing linking issues in MeshKit.
7. Fixes to parallel merge mesh routine, including a new parameter to turn on/off skin entities creation in MOAB and a test for parallel merge mesh in MOAB.
8. Fixes to NetCDF writer (2D sidesets weren't getting converted as expected).

2.2.5 Other MeshKit Fixes

Major additions to the build system are discussed in Section 2, and documentation-related changes are discussed in Section 4. Several other important additions and fixes to MeshKit are listed below:

1. DAGMC library dependencies in MOAB and examples for make watertight.
2. Bug fix reported by Kitware in rectangular assembly creation during CoreGen step of RGG.
3. Compiler-related fixes for newer version of OSX and Linux.
4. Fixes to `get_entities_by_dimension` and `get_entities_by_handle`. These functions will now return `ModelEnts` of dimension 4 if requested. If the dimension passed in is -1, the function will return all `ModelEnts` in the MeshKit instance.
5. Correction to the `ModelEnt` constructor for `iGeom EntSets`.
6. RGG AssyGen feature to "CreateFiles" and "CreateMatFiles."
7. Fixes to quadmesher, removed dependency on BOOST library.
8. Fixes to PostBL along with addition of methods to create mixed element meshes.
9. RGG enhancements for creating superblocks, automated multiple file creation, automatic mesh size specification.
10. AssyGen script added support for scheme hole surfaces for automatic meshing of concentric pins in an assembly.
11. Fixes to parallel HDF5 reader and writer used by CoreGen.
12. Some bugs and changes in external packages: Zoltan, NetCDF and OpenCascade were reported to their respective developers lists.

2.3 Make watertight

Make watertight algorithms have been implemented in the graph-based scheme of MeshKit. Also, the robustness of the algorithm [15] has been improved for large and complex geometries. The sealing process developed in this algorithm has been successfully implemented in the DAGMCNP workflow, resulting in no lost particles for several ITER models. Figure 7 shows the unsealed original ITER blanket lite model on the left and the completely sealed model on the right.

Figure 7. Input and output geometry to make watertight algorithm.

2.4 Sweep Mesh Generation

One of the most robust and widely used algorithms for all-hexahedral meshes is the sweeping algorithm [8]. For multisweeping, however, the most difficult problems are the surface matching and interval assignment for edges on the source and target surfaces. A new method to generate surface meshes by imprinting edge patches between the source and target surfaces is proposed. The edge patch imprinting is based on a cage-based morphing of edge patches on the different sweeping layers where deformed and undeformed cages are extracted by propagating edge patches on the linking surfaces. As a result of the imprinting, the source or target surfaces will be partitioned with the imprinted edge patches. After partitioning, every new source surface should be matched to a new specific target surface where surface mesh projection from one-to-one sweeping based on harmonic mapping can be applied. In addition, 3D edge patches are projected onto 2D computational domains where every sweeping level is planar in order to increase the robustness of imprinting.

Figure 8. Flowchart of surface mesh generation based on edge patch imprinting.

Figure 8 shows a flowchart of our proposed algorithm. It starts with volumes without any surface mesh. The parametric space f_i , kg , and vertex types on the linking surfaces should be provided. This does not imply that all the boundary edges have to be meshed; no restriction is made on the number of points on the boundary edges. Prior to imprinting, the sweeping layer number for each source and target surface should be identified in the sweeping direction. Next, edge patch imprinting between the source and target surfaces is performed, including edge patch extraction, edge patch propagation, cage-based morphing, and intersection processing. The surfaces then are split to make a one-to-one matching between the source and target surfaces. Finally, quad meshes on the source surfaces are mapped onto the target surface by morphing.

Shown in Figures 9, 10, and 11, respectively, are three examples of source and target surface meshes that have been generated by using the edge patch imprinting algorithm and morphing algorithm. Users can manually match edge patches. Otherwise, the imprinting algorithm will use the cage-based morphing to propagate edge patches, partition the source and target surfaces, and match edges between the source and target surfaces. Note that volumes are not decomposed and that only surfaces are partitioned in order to match the source and target surfaces. The reason rests with the inherent characteristics of the sweeping algorithm: every quad element on the source surfaces has its corresponding quad element on the target surfaces. The great disadvantage for decomposing volumes is that interior nodes cannot be moved from one subvolume to another subvolume if poor volume mesh quality is produced.

Figure. 9. A real part from Caterpillar: (a) a geometric model; (b) source surface meshes; (c) partitioned source and target surfaces by imprinting edge patches; (d) target surface meshes by morphing.

Figure. 10. Example of generating surface meshes and matching the source and target surfaces by imprinting edge patches: (a) geometric model; (b) partitioned source and target surfaces; (c) source surface meshes; (d) 3D-view surface meshes; (e) target surface meshes by morphing.

Figure 11. Surface mesh generation for crankshaft by imprinting: (a) geometric model with imprinted edge patches (denoted by blue curves); (b) target surface meshes by morphing.

2.5 Verdict Integration

In finite-element analysis, the shape of individual elements can significantly affect the accuracy of a simulation. Several metrics for each cell type (shape) exist; which one is most closely related to the accuracy of a simulation is application-specific. VERDICT, a separate module providing these quality criteria, has been developed by Sandia/Kitware to help standardize these quality metrics. MeshKit integrates the Kitware Inc. version of VERDICT, which has a BSD licensed version for VTK that is based on Sandia's version of VERDICT. Quality measures for triangular, quadrilateral, tetrahedral, and hexahedral cells are available. The filter will compute the value of one quality metric per cell and allows the user to choose this metric on a per-cell-type basis. For example, one might compute the radius ratio for triangular cells and the Frobenius aspect for tetrahedral cells.

MeshKit tools can now compute the summary information for the entire mesh including the following:

- Average value of some quality metric per cell type (e.g., the average over all tetrahedral cells)
- Variance of some quality metric per cell type
- Minimum and maximum quality metric values per cell type
- Number of cells of each type.

VERDICT documentation from Kitware contains the mathematical definition of each quality metric as well as implementation details. It also has documentation on implementation and definition of all the quality metrics provided by VERDICT [16].

2.6 Tri Mesh Generator

Simple pyramid geometry is meshed with tetrahedral elements by using an external meshing library NetGen, which has been integrated with MeshKit for tet mesh generation. The graph setup by the user is shown in Figure 12(a). During the setup phase of the graph, a tri-mesher MeshOp is created. This MeshOp further creates EdgeMesher and VertexMesher graph nodes. Figure 12(b) shows the nodes created during setup traversal phase in a box. Note that different tri-meshers can be selected with NGTetMesher. Detailed documentation for this example is available in MeshKit doxygen page [17]. Final mesh output from the algorithm is shown in Figure 13.

Figure 12. Graph-based tetrahedral mesh generation.

Figure 13. Tetrahedral mesh created by NGTetMesher MeshOp.

In order for a truly open-source solution to meshing problems, MeshKit needs a robust tri-mesher for open-cascade geometries. This tetrahedral mesh example uses the CAMAL Tri Mesher, which is closed source. We are working on getting AssysGen geometry to mesh with our native or the NetGen tri mesher.

2.7 Quad Mesh Cleanup

QuadMesher (presented in our previous report [6] and International Meshing Roundtable paper [7]) internally cleans up the quad mesh generated in the algorithm. This

quad cleanup algorithm is available as an independent algorithm that reads a quad mesh and produces a mesh with better node-valence, shape quality, and so forth. The options available include the following:

- Removal of interior doublets
- Removal of boundary singlets
- Removal of diamonds
- Vertex degree reduction
- Laplace smoothing
- Advancing front edge swapping
- Shape optimization
- Shifting of irregular nodes inside domain
- Automation of everything

Figure 14(a) shows a mesh before quad cleanup, and Figure 14(b) shows the final mesh after using the automatic option of quad cleanup. The model has a total of 7k nodes. Quad cleanup removes all the singlets and 4 diamonds in the model; 845 edges are swapped. The final model has better valence and fewer irregular nodes. This example can be found in the Meshkit repository. Table 1 lists some of the properties of input and output mesh.

Table 1. Properties of input and output mesh to the quad cleanup algorithm.

Property	Input Mesh	Output Mesh
Nodes with degree > 8	24	1
Nodes with degree = 4	2946	4292
Irregular nodes	3708	3276

(a) Input

(b) Output

Figure 14. Input and output to quad cleanup operation (a) original mesh. (b) final mesh after quad-cleanup.

The dimensions of a single fuel pin are shown in Figure 16. Each pin for a particular axial height consists of four geometric volumes.

The control assembly has an additional duct and sodium layer compared with fuel, shield, and reflector assemblies. The inner and outer ducts of the control rod assembly are shown in Figure 17(a). Figure 17(b) shows the coarse homogenized control assembly mesh. The AssyGen input file used for creating the fuel assembly can be found in Appendix C. Since all assemblies that form the ABTR core (Section 3.1.2 and Section 3.1.3) have varying properties in the axial direction, a common axial configuration must be determined in order to have coincident nodes along the height of all eight assemblies that form this 199-assembly core model. This configuration leads to a conformal mesh that is fit for simulations.

Figure 16. Section of metal fuel pin showing dimensions in cold condition.

New keywords “NumSuperBlocks” and “SuperBlocks” were introduced to combine material blocks for different physics. “NumSuperBlocks” specifies the number of superblocks, and “SuperBlocks” specifies the blocks to be merged to form the new superblock. For example, neutronics models each fuel pin with tens of materials along the height, whereas thermohydraulics models consider all the fuel pins to be one material. These keywords help in the creation of one-mesh files that can be used by all physics simulations. Material and boundary conditions that are not required by a particular physics are ignored. We note that superblocks are not required when modeling meshes for individual physics separately. The clock time to create the homogenized geometry and mesh for this fuel assembly is 2 minutes. The mesh has 2.5k hex elements.

Figure 17. (a) Control assembly absorber region containing two ducts; (b) Inlet of actual control assembly with homogenized control pins (magenta), two ducts (red), and half of inter-assembly sodium gap (green).

3.1.2 ABTR 7-Assembly Model

This section highlights the two minicore models for testing standalone and coupled physics simulations. The assumptions and requirements listed below are applicable to core models detailed in this Section, Section 3.1.3, and Section 3.2.1.

Key Assumptions and Requirements

Several meshing restrictions and assumptions were made for modeling the ABTR assemblies, restraint-ring, and full core mesh. Moreover, physics codes: PROTEUS/Diablo and Nek5000 impose certain features and required characteristics on the mesh:

1. Geometry models are created using dimensions when the reactor is cold.
2. The model is homogenized; pins and other instrumentation inside the assemblies are not modeled.
3. All 199 assemblies are modeled with different material and boundary conditions in order to specifically identify and prescribe inlet/outlet boundary conditions to a particular assembly. Each assembly is numbered.
4. The axial and radial mesh size is coarse to keep the element count low.
5. The nosepiece region at the bottom of each assembly is not modeled.
6. Additional axial materials are created for modeling fuel regions in PROTEUS.
7. Nek5000's spectral element solver requires hex27 elements.
8. Nek5000 requires a plenum region that connects all the coolant flowing through the individual assemblies at the top of the reactor. The axial height of outlet plenum is 30 cm.
9. Diablo and Nek5000 model fuel elements as a uniform "mush" across all the axial fuel regions.
10. PROTEUS needs only 3 boundary conditions: top, bottom, and side of the entire core.
11. Boundaries modeled for Nek5000 and Diablo are as follows:
 - a. Inner/outer walls of the restraint rings
 - b. Wall of gap between assemblies and restraint rings
 - c. Inlet for whole core and each assembly
 - d. Outlet for whole core and each assembly

- e. Wall of TLP for each assembly
- f. Wall of ACLP for each assembly

7-Assembly Core with Four Different Assemblies

This minicore consists of 7 assemblies surrounded by a restraint ring. The core contains 3 inner core fuel assemblies, 2 reflector assemblies, 1 shield assembly, and 1 central control assembly (Figures 18(a) and (b)). This case was intended to test four of the assembly types (fuel, reflector, shield, control) present in the ABTR geometry.

The small problem size allowed for easier debugging of the mesh via both visualization and manual inspection of input/output. During the specification of this problem, conventions were agreed upon for the mesh block ordering, which was instrumental to streamlining the input generation for the full-core case. All the assemblies used in this model are same as those described for the 199-assembly core (Section 3.3).

Each of the four types of assemblies has a material and boundary condition names prefixed with IJJ (Figure 18(b)). To overcome the problem of manually creating three separate AssyGen files for fuel assemblies, the “CreateMatFiles” keyword was introduced. This keyword creates AssyGen files with name “IJ”.inp and sets the start material and boundary condition numbers based on “IJ.” This is important for the 199-assembly core model, where tens of files of each kind are present. One AssyGen run on the base file that describes all required “IJ”.inp files via CreateMatFiles keyword generates all the input files corresponding to that particular assembly. This enables a numbering scheme that is manageable and helps prescribe temperature, flow rate, and so forth for a particular assembly easily. Figure 19 shows the final mesh along with power distribution plots.

Figure 18. Minicore assembly: (a) configuration; (b) numbering scheme of assemblies (red).

(a)

(b)

(c)

Figure 19. Minicore mesh: (a) elements and blocks; (b) internal assembly blocks used in PROTEUS; (c) power distribution.

7-assembly Core with All Fuel Assemblies

A second minicore consisting of 7 fuel assemblies surrounded by a restraint ring was created to provide a more realistic model, while keeping the problem size small (Figure 20). The CoreGen input file for creating the mesh for the all-fuel minicore can be found in Appendix D.

Figure 21 highlights the ACLP and TLP regions. The gap between assemblies and restraint ring is kept the same as the full-core restraint ring model (these gaps are same for the full core model shown in Section 3.3). In Figure 21(c), three of the load pads on fuel assemblies are highlighted. These load pads are present in all assemblies outside the structural steel covering. The sodium flow region between the load pads of individual assemblies is divided in half and modeled with each assembly separately.

Figure 20. Composition map for fuel-only minicore.

Figure 21. Assembly structural components: (a) restraint rings; (b) load pads on fuel assemblies; (c) gap between assemblies and restraint rings ACLP (0.0235 cm, gray) and TLP (1.2025 cm, green).

3.1.3 ABTR Full Core

The 199-assembly full core with restraint ring consists of 60 fuel assemblies (24 inner, 30 outer, 6 test), 10 control assemblies, 48 shield assemblies, 78 reflector assemblies, and 3 material test assemblies (which are modeled like reflector assemblies). Figure 22(a) also shows the detailed configuration with I-J numbering and the number of occurrences of each of the assembly types. Horizontal lines represent J increasing from top to bottom, and slanted vertical lines following the core arrangement represent I from left to right. All 199 assemblies must be modeled independently with different materials in order to enable specification of varying densities, inlet/outlet boundary conditions, and so forth for a particular assembly in the core model. The core is modeled with four different assemblies: fuel, reflector, shield, and control.

Figure 22. ABTR full-core configuration: (a) with lines of constant logical I, J assembly regions; (b) with homogenized assemblies, outer covering, and restraint rings at TLP and ACLP locations

The final homogenized core model is created from assembly meshes (Section 3.1) and the outer covering mesh. The mesh shown in Figure 22(b) consists of 800,000 hex elements. The structural mechanics and thermal-hydraulics mesh consists of 1,500 material blocks, whereas the neutronics mesh consists of 7,200 material blocks. The neutronics mesh requires more

material blocks along the height of the fuel pins. It takes 30 minutes for CoreGen to create this model on a Linux workstation using 32 processors.

RGG tools can handle changes in assembly/core dimensions and process only the parts that are modified. Also, various arrangements of fuel and other assemblies can be simulated very easily. For example, different outer rings with varying TLP and ACLP gaps can be created by modifying the parameters that generates the restraint rings. The currently available geometry tools are capable of generating a nonhomogenized assembly and core model, which taxes readily available computing resources; therefore, simpler, homogenized models were used in this ABTR demonstration.

3.2 Meshes Created For CoreGen Scalability Study

Creating detailed geometry and meshes for reactors using commercial tools for expert users require a large workstation (~1 TB RAM), and experienced users take months of time just to create meshes for such reactor problems (~1 billion elements). In order to resolve the detailed flow in the reactor, a mesh size of 5-10 billion cells might be required. The CoreGen tool in MeshKit has been tested to work in parallel and create a few million element meshes. The ability to quickly create detailed meshes enables studies of rod vibrations caused by cross flow and turbulence vibration flows. Also, mesh convergence studies can be done for testing and validating the physics involved. MeshKit uses parallel HDF5 reader/writer for handling meshes; this feature is obtained from MOAB library that works closely with the HDF5 group. This section further highlights some large mesh models and scalability results for CoreGen.

3.2.1 Fine ABTR Core Model

CoreGen is run to create the final homogenized core model from assembly meshes and this outer covering mesh. Two models were created for this ABTR core model:

1. Coarse model: The final mesh is shown in Figure 22(b), and details are given in Section 3.1.3. CoreGen takes 5 seconds to assemble the core using 128 processors. The maximum memory used by a processor is 110 MB. The serial runtime for CoreGen is 45 seconds with 280 MB of memory used.
2. Fine model: Each assembly in this model consists of 5–10M hexahedral elements. The final mesh size of the core model created is **1.2 billion hexahedral elements**, the **size of this file on disk is 137GB**. CoreGen on 200 processors (each processor loads one mesh file: 199 assembly + 1 interstices mesh file) takes 30 mins to create this file of which parallel save takes up 27 mins and parallel merge takes 1.7 mins. The maximum memory used by a processor is 1.96GB.

3.2.2 1/6th VHTR Core

The 1/6 Very High Temperature Reactor (VHTR) core model shown in Figure 23 consists of 19.6M hexahedral elements and 20.5M mesh vertices. The size of this mesh file on disk is 2.2 GB.

Figure 23. One-sixth of a VHTR core model generated by using CoreGen (left); closeup of assembly mesh in this model (right).

AssyGen took 10.5 minutes to generate the geometry, journal files, and run CUBIT in order to mesh individual assemblies. The serial CoreGen program took 48 minutes to generate the core model. When using 56 processors for running CoreGen, this model can be generated in less than 3 minutes (2.1 minutes with 12 processors; 12 assemblies form this 58-assembly model) from scratch. The AssyGen+CUBIT are run on 1, 4, 8, and 12 processors by using the `-j` option of the makefile; the maximum number of processors is limited to 12 for this step.

Table 2 lists the AssyGen+CUBIT, copy/move, merge, parallel save, total time, and the maximum memory used for various steps of the CoreGen stage, when using different number of processors. Figure 24 shows these timing results and maximum memory used vs. number of processors. Superlinear speedups are observed in almost all cases, because of the job fitting in available memory. Mesh joining (or merge) is observed to be actually slow as the number of processors increases from one to four; this result is probably due to the communication overhead required in the parallel algorithm. At larger numbers of processors, however, the joining time is reduced far below the serial time. As expected, the total time, time taken to save, and maximum memory used by a processor decrease as the number of processors increases.

Table 2. CPU time in minutes and maximum memory in gigabytes used for 1/6 VHTR core

# Procs	AssyGen+CUBIT	Copy/Move	Merge	Save	Total	Memory
1	10.5	17.6	10.4	0.7	48.2	4.9
4	3.4	11.0	11.7	0.01	26.2	2.6
8	2.8	11.1	5.6	0.01	19.5	2.5
16	2.1	0.4	4.7	0.01	7.3	1.7
32	2.1	0.03	0.56	0.01	2.7	0.48
56	2.1	0.0005	0.31	0.005	2.43	0.33

2

Figure 24. Log scale results from Table 2 vs. number of processors for 1/6 VHTR reactor core.

3.2.3 MONJU Reactor

Figure 25 shows a full-core MONJU reactor, which comprises 8 assembly types and consists of 715 assemblies. AssyGen and meshing take 5.5 minutes (8 processors using make -j8 option) to mesh the 8 assemblies. CoreGen on 712 processors takes only 90 seconds. The total wall-clock time required to generate this 101M hexahedral element model from scratch is 8 minutes. The maximum memory used by a processor is only 196 MB. This model cannot be run in serial because the problem does not fit in memory. The size of the mesh file on disk is 14 GB.

Figure 25. Full-core MONJU reactor; closeup area in red rectangular region is highlighted from left to right.

Using a round-robin scheme to establish the copy/move work for this reactor causes the load on some processors to be larger than the available memory. Therefore, we use a recursive load-balanced scheme, wherein after deciding every assembly load, the resulting assembly copy/move work due to this load is analyzed and then the decision on the next assembly load is made. Table 3 lists the AssyGen+CUBIT, copy/move, merge, parallel save, total time, and maximum memory used with different numbers of processors.

Table 3. CPU time in mins and maximum memory in GB used for MONJU core

#Procs	AssyGen+CUBIT	Copy/Move	Merge	Save	Total	Memory
16	5.5	35	163	3.8	207.64	4.2
64	5.5	0.03	61	9.01	76.7	1.04
128	5.5	0.03	293	15.5	315	0.532
256	5.5	0.03	291	24	321.5	0.244
320	5.5	0.002	0.45	16.5	22.8	0.233
512	5.5	0.002	0.49	3.85	9.99	0.233
712	5.5	0.002	0.18	2.3	7.89	0.196

Figure 26. Log scale results from Table 3 vs number of processors plot for MONJU reactor core.

Figure 26 shows the log scale results from Table 2 vs the number of processors used. One can see the sudden jump in merge/save and total time for 128 and 256 processors. We determined that a few processors performing local merge spend a majority of the time. Local merge creates an adaptive kd-tree and then finds and merges the nodes on the skin for all elements local to a particular processor. This jump in time for merging is possibly due to the large number of searches for determining the nodes to be merged locally. We plan to fix this problem in future versions of the tool by using a different tree and/or fixing the existing kd-tree implementation.

Compared with serial, the total time required to generate this model drops to 22 minutes for 320 processors, and for 512 and 712 processors this time further drops to 10 minutes and 8 minutes, respectively. The overall maximum memory required by a processor decreases as the number of processors increases.

3.3 VERA Benchmark Problems

RGG tools were used to create 2D meshes for CASL-VERA benchmark problems and test the PROTEUS-SN code developed at Argonne National Laboratory. The meshes shown in this section were developed by the Oak Ridge team using RGG GUI, MeshKit, AssyGen, and Cubit. AssyGen was used for modeling both the single pin and 17x17 assembly model shown in Figure 28. AssyGen keywords were used to specify radial mesh size and edge mesh interval. Using regular paving to mesh assemblies with several concentric pins fails to generate a valid mesh for the geometry. Instead, the scheme “hole” must be used for better meshing stability and generation of a good-quality mesh.

Figure 27. Simple pincell model. (left: coarse, right: refined)

Figure 28. 17x17 PWR assembly model

4 Documentation

MeshKit contains detailed documentation for a new user or developer to get started. The SIGMA v1.0 release focused on releasing all the associated libraries at the same time, including MeshKit v1.2 with documentation and a common webpage. Documentation and guides for MeshKit can be obtained from four major sources :

1. **SIGMA Website:** This is a one-stop place containing introduction, related links, and all the latest news and developments about CGM, MOAB, Lasso, and MeshKit. It also contains links to other sources of documentation mentioned in this section. Publications and download and build instructions for all the SIGMA components are available here.
2. **Doxygen:** MeshKit user and developers guide along with detailed documentation of all of MeshKit C++ classes are available from the MeshKit doxygen page. This is updated nightly, enabling the documentation to go hand in hand with code development. Doxygen pages also detail examples and tests for new users to start using MeshKit.
3. **Bitbucket:** All SIGMA code is hosted on Bitbucket, which includes build instructions along with information on branches and other developer-related information.
4. **GitHub:** Kitware has hosted the code for RGG GUI on GitHub. This code is publicly available and can be used to build MeshKit and the RGG GUI app discussed in Section 5. Information about branches and other code-related documentation are available on this site.
5. **RGG GUI Manual:** The instructions manual on using the RGG GUI application for creating reactor assembly and core models is available from Kitware GUI App. This can be generated by specifying the option to “build documentation” during the build process of the RGG GUI app.

5 RGG GUI 1.0 Release and Collaboration with Kitware

In collaboration with Kitware Inc., the SIGMA team worked on developing a graphical user interface for MeshKit and RGG. Kitware Inc. is also the developers of the CMake build system and their staff is contributing to streaming and setting up CMake build systems for all SIGMA libraries. This year we have hosted training sessions at Oak Ridge and Argonne National Laboratory for the RGG GUI app. The feedback from users and experts along with our collaboration has made RGG a powerful application.

AssyGen, from Meshkit, takes in text files that describe a collection of pins and ducts and generates solid models of these assemblies. The assemblies are then passed to CUBIT in order to generate hexahedral meshes of these assemblies. CoreGen, from Meshkit, takes these meshes, along with a core layout text file, and produces an acceptable mesh of the reactor core. Any modifications to the assemblies or core text files will require the appropriate part of

the workflow to be re-executed. Although this workflow is powerful and can generate a large range of reactor core models and meshes, it has been used primarily by expert meshing computational scientists, not by non-expert/non-meshing computational scientists, nuclear engineers, or technicians. The RGG application was designed to address this specific issue while enhancing productivity of the meshing expert. Without the ability to visualize and graphically interact with key sections of the workflow, all users will produce errors in a file-based workflow, which may not be caught until the simulation itself is run, resulting in wasted time and resources. The overall MeshKit RGG workflow is given in Figure 1.

Steps involved in RGG GUI app are as follows:

1. **Designing Pins:** The first stage involves designing the individual pin cells that will be used in a reactor's subassemblies. These pins represent fuel, control rods, and various instrumentation. The pin cell editor presents a pin cell as a collection of segments. Each of these segments is shaped as either a cylinder or a truncated cone. By default, the editor constrains neighboring segments to have the same mating radii. If the end user changes one segment's radius, all constrained radii are automatically updated. In addition, the pin editor allows the end user to define contiguous layers throughout the pins. Each layer is assigned its own material. See Figure 29.

Figure 29. Pin being edited using RGG's pin cell editor.

2. **Designing Ducts:** The second stage involves designing the ducts of the sub-assemblies that will be used in laying out the core. Each assembly consists of a set of ducts that will contain the pins. The overall height of the ducts and pins must be the same across all of the core assemblies. To enforce this requirement, the user can set the overall core length, which is then enforced through the workflow. Similar to the pin editor, the GUI provides users with the ability to specify the duct structure of each sub-assembly.

Figure 30. Simple duct being designed that consists of two materials. One material forms the walls of the subassembly, while water is assigned as the inside material.

3. **Designing Assemblies:** In the third stage, the pins and ducts are available for placement within a subassembly lattice, shown in Figure 30. The subassembly editor provides both a 3D view of the assembly and an editable 2D lattice view that supports drag and drop interaction. The offset between neighboring pins in the assembly is referred to as the *pitch* and depends on the size of the inner layer of the duct. Since the majority of assembly designs place the pins so that they are equidistant within the duct, RGG provides a mechanism to autocenter the pins. An example reactor assembly is shown in Figure 31.

Figure 31. Example reactor assembly in RGG

4. **Core Layout:** In the fourth stage, the complete core is defined by placing the different subassemblies into the core's lattice. Similar to the sub-assembly editor, the core editor view provides a 3D view of the core, as well as the editable 2D lattice views. Figure 32 depicts various examples of core layouts.

Figure 32. A simple hexagonal reactor core modeled in RGG.

5. **Generating Mesh:** The final step involves calling AssyGen, CoreGen, and Cubit in order to produce the required solid models and meshes. The process is initiated within the RGG application. RGG does this through another Kitware toolkit, the open-source Remote Meshing Utilities (ReMUs). ReMUs can execute the various tools while allowing RGG to remain interactive. ReMUs relays the status of these external tools back, while providing the ability to abort the sequence while it is running. Once these processes are complete, RGG enables the display of the core mesh, using MOAB, as well as visualization of the various components of the mesh, based on volumes, materials, and boundary condition element sets.

Figures 33, 34, and 35 show various examples of a reactor core mesh, a 199 assembly ABTR, and a PWR reactor core with five different types of assemblies, respectively, each generated by using RGG.

Figure 33. An example of a nuclear reactor core mesh generated using RGG.

Figure 34. 199 Assembly ABTR created using RGG.

Figure 35. A PWR reactor core with five different types of assemblies modeled using RGG.

6 Conclusions

MeshKit provides a robust parallel infrastructure for geometry/meshing researchers, tool developers, and users needing to generate geometry/mesh. MeshKit has most of the traditional meshing algorithms required for meshing complex geometries. Moreover, we have devoted significant effort to the build system and documentation in MeshKit v1.2.

The RGG tools consist of AssyGen and CoreGen, both of which use text-based input files. These input files are based on a set of keywords that help define the geometry and meshing parameters for creating a nuclear reactor core. The RGG GUI enables users to build the reactor model without writing text-based input files. The GUI is intuitive and easy to understand; it comes with a users manual; and it works with Linux, Mac OSX and Windows operating system. The RGG GUI developed by Kitware as a part of an SBIR can display the geometry and meshes created by AssyGen and CoreGen, respectively.

MeshKit is ideally suited for development of new meshing tools. Simulation of complex systems such as nuclear reactors requires detailed models that properly capture the geometric shape and have correct specification of material and boundary conditions. Different physics such as neutron transport, fluid flow, thermal expansion, and heat transfer must be studied in order to fully understand the performance and safety aspects of nuclear reactors. The parallel RGG tools enable the creation of such large and complicated reactor models for

different physics simulations. Several enhancements and fixes to the AssyGen/CoreGen tools in RGG, including introduction of shifting material and Neumann set ids, have been incorporated in this release. With the introduction of the new distribution scheme for CoreGen, we can create meshes such as the finer ABTR core mesh (file on disk is 137 GB and consists of more than a 1 billion hexahedral elements). Such meshes are impossible to construct by using serial meshing processes on a standalone workstation. We plan to create fully detailed models and coupling results for these models. Work is in progress for developing geometry partitioning-based meshing algorithms to mesh the assemblies in parallel. Also, new schemes are being formulated for a better load-balancing during copy/move task distribution; this scheme will combine existing schemes with more weightage given to meshes with larger element count. Parallel AssyGen development would involve creation of individual pins and components in parallel, but we lose the speedup with serial subtraction of pins; this development therefore is subject to development of parallel subtraction of geometries in CGM.

Current work involves adapting the RGG GUI app to run in parallel and developing an open source tri mesher to have a completely open RGG GUI app, which currently depends on Cubit (closed source). We are also involved in linking of the Gmsh meshing package with MeshKit. Also planned are development of automatic scheme selection and support for higher-order elements in postmesh boundary layer tools.

Acknowledgments

We thank the SIGMA group at Argonne, who maintain the libraries required by MeshKit and Kitware Inc. for collaborating on the development of RGG GUI application. This material was based on work supported in part by the U.S. Department of Energy, Office of Nuclear Energy, Nuclear Energy Advanced Modeling and Simulation (NEAMS) Program and by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program and by the U.S. Department of Energy's Scientific Discovery through Advanced Computing program, under Contract DE-AC02-06CH11357.

References

1. SIGMA (Scalable Interfaces for Geometry and Mesh-Based Applications) website: <http://sigma.mcs.anl.gov/>
2. Tautges, T. J. (2005) CGM: a geometry interface for mesh generation, analysis and other applications. *Eng. Comput.* 17:486–490.
3. Spatial website (2010) <http://www.spatial.com/>
4. Open CASCADE technology website (2000–2010) <http://www.opencascade.org>
5. Tautges, T. J., Meyers, R., Merkley, K., Stimpson, C., and Ernst, C. (2004). MOAB: A mesh-oriented database, SAND2004-1592. Sandia National Laboratories, Albuquerque, NM.
6. Jain, R., and Tautges, T. J. (2013). MeshKit. *Report of US DOE, Reactor Campaign*. ANL/MCS-TM-336.
7. Verma, C. S., and Tautges, T. (2012). Jaal: Engineering a high quality all-quadrilateral mesh generator. In *Proceedings of the 20th International Meshing Roundtable* (pp. 511–530).

8. Cai, S., and Tautges, T. (2014). Robust one-to-one sweeping with harmonic ST mapping and cages: Post-mesh boundary layer generation tool. In *Proceedings of the 22nd International Meshing Roundtable* (pp. 118).
9. Kim, H., and Tautges, T. J. (2010). EBMes: An embedded boundary layer meshing tool. In *Proceedings of the 19th International Meshing Roundtable* (pp. 227–242).
10. Jain, Rajeev, and Tautges, T. J. (2014). PostBL: Post-mesh boundary layer generation tool. In *Proceedings of the 22nd International Meshing Roundtable* (pp. 445–464).
11. Mitchell, S. (2014). Simple and fast interval assignment using nonlinear and piecewise linear. In *Proceedings of the 22nd International Meshing Roundtable* (pp. 203–221).
12. Sjaardema, G. D., Tautges, T. J., Wilson, T. J., Owen, S. J., Blacker, T. D., Bohnhoff, W. J., Edwards, T. L., Hipp, J. R., Lober, R. R., and Mitchell, S. A. (1994). CUBIT mesh generation environment, users manual, vol 1. Sandia National Laboratories, Albuquerque.
13. Jain, Rajeev, and Tautges, T. J. (2012). RGG: Reactor Geometry (and mesh) Generator. *International Congress on the Advances in Nuclear Power Plants*, Chicago.
14. Buildbot: <http://gnep.mcs.anl.gov:8010/>
15. Smith, B., Wilson, P., and Tautges, T. J. (2010). Sealing faceted surfaces to achieve watertight CAD models. In *Proceedings of the 19th International Meshing Roundtable* (pp. 177–194).
16. VERDICT documentation: <http://www.vtk.org/Wiki/images/6/6b/VerdictManual-revA.pdf>
17. MeshKit doxygen page: <http://ftp.mcs.anl.gov/pub/fathom/meshkit-docs/index.html>
18. GRUMMP: <http://tetra.mech.ubc.ca/GRUMMP/>

Appendix A. Autotools Based Configure/Build/Installation of MeshKit

MeshKit is maintained as open source software under an LGPL license and is therefore distributed in source code form. The library uses several required and optional libraries that must be built and installed prior to MeshKit installation. MeshKit is currently supported on Linux and Linux-like operating systems (including MacOS).

Prerequisites

MeshKit requires the following libraries to be installed before configuration:

- CGM: a library for representation, query and modification of geometric models; see [2] for details on obtaining and building CGM.
- MOAB: a library for representing structured and unstructured mesh; see [5] for details on obtaining and building MOAB.
- Autotools: a set of Linux utilities for configuring software packages. Autotools can be found in most Linux package managers, and usually consists of the Autoconf and Automake packages.

In addition, if a parallel version of MeshKit is desired, the Message Passing Interface (MPI) must be available on the user's computer; binary versions of MPI can be found in most Linux package managers.

Download, Configure, Build, Install

MeshKit source code is maintained in a world-readable svn repository, located at <https://www.bitbucket.org/fathomteam/moab/moab.git>. By default, MeshKit uses a GNU Autotools-based configuration process. The following steps should be used to configure, build, and install MeshKit:

- Unpack the source tarball into a directory referred to below as <MK_DIR> and change directory into that location.
- Execute ‘autoreconf –fi’. This executes a series of tools in the autotools suite, storing some generated files in the ‘config’ subdirectory.
- Execute ‘./configure’ with appropriate options. Two configure options are required, specifying the locations of CGM (--with_igeom=<location>) and MOAB (--with-imesh=<location>). Other useful configure options are the installation location (--prefix=<location>) and specifying debug or optimized builds (--enable-debug, --enable-optimized, respectively). For a complete list of options, execute the command ‘./configure –help’. After a successful configuration, a set of Makefile’s are generated in the proper subdirectories.
- To complete the build of MeshKit, execute ‘make’.
- To install MeshKit, execute ‘make install’. If the install location was not specified on the configure line, one can specify a location in this step by using the command ‘make prefix=<location> install’.

For those wishing to use the Python interface, MeshKit and its dependencies should be configured to build shared libraries, using the ‘--enable-shared’ configure option where appropriate.

Once the MeshKit library has been built, it is ready for inclusion into user-developed applications (any MeshKit-packaged programs; e.g., those that constitute RGG, will be installed in the ‘bin’ directory). To aid in building user-developed applications, MeshKit also writes a file ‘meshkit.make’, which can be included directly into application makefiles. This file defines the following make variables useful for building MeshKit-based applications:

- **MESHKIT_INCLUDES, MESHKIT_CPPFLAGS:** compiler options pointing to all directories containing include files available to applications, including those for CGM and MOAB; also, CPP definitions controlling which optional external meshing tools have been configured into MeshKit.
- **MESHKIT_LIBS_LINK:** linker options necessary to satisfy all functions included in MeshKit.

The ‘examples’ subdirectory in the MeshKit source installation contains an example makefile showing how these make variables can be used to compile and link MeshKit-based applications.

Appendix B. CMake-Based Builds: RGG Nuclear GUI Repository (SBIR: Kitware)

An open-source public release of RGG version 1.0 was done on August 10, 2014. The code is maintained in a world readable repository, located at <https://github.com/Kitware/RGG.git>. The source code includes a CMake SuperBuild for building MeshKit as well as RGG’s other open-source dependencies, CGM, MOAB, OpenCASCADE, ReMUs, VTK, and QT. The Cubit meshing tool is available through Sandia National Labs. CMake-based build systems for RGG GUI and MeshKit have been developed and are part of the RGG GUI source code.

For all CMake-based builds, the latest version of CMake must be installed. The latest version of CMake is available here: <http://www.cmake.org/download/>.

RGG GUI Superbuild

The RGG GUI can be downloaded and built to work on a Linux, Mac OSX, or Windows operating system. The build system for RGG GUI can be separate from MeshKit. In future versions of the tool, binaries to the GUI app for all types of OSX will be available.

Prerequisites

- Git for downloading the source code from the web.
- Latest version of CMake.
- OpenGL. This isn’t installed as a part of RGG GUI installation and must be available in the system on which installation is performed.

Steps for building the RGG GUI

- Get the repository git clone <https://github.com/Kitware/RGG.git>
- cd RGG/superbuild, mkdir build, cd build
- Invoke cmake or ccmake on RGG directory: ccmake ..
- Choose the appropriate options.
 - Turn BUILD_DOCUMENTATION ON to generate the RGG GUI users guide.

- There are options to use libraries already available in the system, e.g., use of existing Qt library.
- Building with MOAB support is optional. If the GUI is built without MOAB support, loading of the final mesh file generated by CoreGen doesn't work.
- Building with MeshKit and CUBIT is also part of the options; they are turned off by default.
- Run `ccmake ..`, configure, generate and exit.
- Run `'make'`.
- For linking the libraries LD or DYLD, `_LIBRARY_PATH=<your directory>/superbuild/build/install/lib`.
- RGGNuclear app is installed in the folder `<your directory>/superbuild/build/nuclearRGG/src/nuclearRGG-build/Application/`

Why is this last point here - it is not a step unless you mean Install the RGG...

MeshKit Superbuild

The source code in GitHub repository contains a separate build system for MeshKit, and associated SIGMA tools can be downloaded and built to work on a Linux, Mac OSX, or Windows operating system. The build system for RGG GUI can be separate from MeshKit. In future versions of the tool binaries to the GUI app for all types of OSX would be available

Prerequisites

- Git for downloading the source code from the web.
- Latest version of CMake.

Steps for building MeshKit using CMake are:

- Get the repository git clone <https://github.com/Kitware/RGG.git>
- `cd RGG/meshkit, mkdir build, cd build`
- Invoke `cmake` or `ccmake` on meshkit directory: `ccmake ..`
- Choose the appropriate options.
 - Turn `BUILD_WITH_CUBIT ON` to build CGM with CUBIT option; the script requires the CUBIT installation directory to proceed.

- MOAB dependencies HDF5 and NetCDF are currently supported in serial mode only. Options to use system-installed version are also available.
- Support for OCE-based CGM is also available and is turned on by default.
- For linking the libraries set LD or DYLD, `_LIBRARY_PATH=<your directory>/meshkit/build/install/lib`.
- All done, launch the executable “assygen” and “coregen” from the folder `<your directory>/meshkit/build/install/bin`.

Appendix C. Input File for a Fuel Assembly with 24 Axial Regions

```
#####
! ABTR Minimal Fuel (cm).
#####
Geometry Volume
GeometryType Hexagonal
!
Materials 36 MTLP MTLP &
MACLP MACLP &
Oplenum Oplenum &
StHT9 StHT9 &
Ustructure Ustructure &
GPNA_BC GPNA_BC &
GPNA01 GPNA01 &
GPNA02 GPNA02 &
GPNA03 GPNA03 &
GPNA04 GPNA04 &
GPNA05 GPNA05 &
GPNA06 GPNA06 &
GPNA07 GPNA07 &
GPNA08 GPNA08 &
GPNA09 GPNA09 &
GPNA10 GPNA10 &
GPNA11 GPNA11 &
GPNA_BACLP GPNA_BACLP &
GPNA_ACLP GPNA_ACLP &
GPNA_BTLP GPNA_BTLP &
GPNA_TLP GPNA_TLP &
GPNA_ATLP GPNA_ATLP &
NAHT9 NAHT9 &
Active01 Active01 &
Active02 Active02 &
Active03 Active03 &
Active04 Active04 &
Active05 Active05 &
Active06 Active06 &
Active07 Active07 &
Active08 Active08 &
Active09 Active09 &
Active10 Active10 &
GP_FuelBond GP_FuelBond &
GP_Bond GP_Bond &
GP_GasBond GP_GasBond
Duct 4 0 0 38 80.1 13.598 14.198 14.463 14.598 NAHT9 StHT9
GPNA_BC GPNA_BC
```

Duct	4	0	0	80.1	98	13.598	14.198	14.463	14.598	NAHT9	StHT9
	GPNA_BC		GPNA_BC								
Duct	4	0	0	98	106	13.598	14.198	14.463	14.598	Active01	
	StHT9	GPNA01	GPNA01								
Duct	4	0	0	106	114	13.598	14.198	14.463	14.598	Active02	
	StHT9	GPNA02	GPNA02								
Duct	4	0	0	114	122	13.598	14.198	14.463	14.598	Active03	
	StHT9	GPNA03	GPNA03								
Duct	4	0	0	122	130	13.598	14.198	14.463	14.598	Active04	
	StHT9	GPNA04	GPNA04								
Duct	4	0	0	130	138	13.598	14.198	14.463	14.598	Active05	
	StHT9	GPNA05	GPNA05								
Duct	4	0	0	138	146	13.598	14.198	14.463	14.598	Active06	
	StHT9	GPNA06	GPNA06								
Duct	4	0	0	146	154	13.598	14.198	14.463	14.598	Active07	
	StHT9	GPNA07	GPNA07								
Duct	4	0	0	154	162	13.598	14.198	14.463	14.598	Active08	
	StHT9	GPNA08	GPNA08								
Duct	4	0	0	162	165.2	13.598	14.198	14.463	14.598	Active09	
	StHT9	GPNA09	GPNA09								
Duct	4	0	0	165.2	170	13.598	14.198	14.463	14.598	Active09	
	StHT9	GPNA09	GPNA09								
Duct	4	0	0	170	178	13.598	14.198	14.463	14.598	Active10	
	StHT9	GPNA10	GPNA10								
Duct	4	0	0	178	182	13.598	14.198	14.463	14.598	GP_FuelBond	
	StHT9	GPNA11	GPNA11								
Duct	4	0	0	182	182.89	13.598	14.198	14.463	14.598	GP_FuelBond	
	StHT9	GPNA11	GPNA11								
Duct	4	0	0	182.89	183.12	13.598	14.198	14.463	14.598	GP_Bond	
	StHT9	GPNA_BACLP	GPNA_BACLP								
Duct	4	0	0	183.12	193.28	13.598	14.198	14.463	14.598	GP_Bond	
	StHT9	MACLP	GPNA_ACLP								
Duct	4	0	0	193.28	202.654		13.598	14.198	14.463	14.598	GP_Bond
	StHT9	GPNA_BTLP	GPNA_BTLP								
Duct	4	0	0	202.654	267		13.598	14.198	14.463	14.598	GP_GasBond
	StHT9	GPNA_BTLP	GPNA_BTLP								
Duct	4	0	0	267	298		13.598	14.198	14.463	14.598	GP_GasBond
	StHT9	GPNA_BTLP	GPNA_BTLP								
Duct	4	0	0	298	312.72	13.598	14.198	14.463	14.598	Ustructure	
	StHT9	GPNA_BTLP	GPNA_BTLP								
Duct	4	0	0	312.72	322.88	13.598	14.198	14.463	14.598	Ustructure	
	StHT9	MTLP	GPNA_TLP								
Duct	4	0	0	322.88	328		13.598	14.198	14.463	14.598	Ustructure
	StHT9	GPNA_ATLP	GPNA_ATLP								
Duct	4	0	0	328	358	13.598	14.198	14.463	14.598	Oplenum	
	Oplenum		Oplenum		Oplenum						
Assembly 1											
XX											

```

1 &
1 &
1 &
1 &
1 &
1 &
1 &
1 &
1 &
1
EdgeInterval 3
CreateSideset No
CreateMatFiles 6
List_MaterialSet_StartId 6 &
1622000      &
1818000      &
1821000      &
1916000      &
2118000      &
2219000
List_NeumannSet_StartId 6 &
1622000      &
1818000      &
1821000      &
1916000      &
2118000      &
2219000
END

```

Appendix D. Input File for a Fuel-only Mini-core with Restraint Rings

```

!#####
###
! Simple 7 assembly All Fuel ABTR Mini-Core With 3 Rings
!#####
###
!
Geometry Volume      ! 'Geometry' card defines if the meshes are volume or surface
Symmetry 1           ! 'Symmetry' card defines the desired symmetry in the model
GeometryType HexFlat ! 'GeometryType' card can take values Hexagonal Rectangular
Assemblies 7 14.598  ! 'Assemblies' is #assembly mesh files and their pitch
1916000.exo 1916      ! Meshfile name followed by alias
1622000.exo 1622
2118000.exo 2118
2219000.exo 2219
1818000.exo 1818
1821000.exo 1821
1620000.exo 1620
Lattice 2
    1916 1622      &
    2118 2219 1818  &
    1620 1821
NeumannSet Top 1
NeumannSet Bot 2
Background or.exo    ! Background mesh file
OutputFileName 7a_all_fuel.h5m
END

```

Mathematics and Computer Science Division

Argonne National Laboratory

9700 South Cass Avenue, Bldg. 240

Argonne, IL 60439

www.anl.gov

U.S. DEPARTMENT OF
ENERGY

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC