

MOOSE Reactor Module Meshing Enhancements to Support Reactor Analysis

Nuclear Science and Engineering 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 at OSTI.GOV (<http://www.osti.gov>), a service of the US Dept. of Energy's Office of Scientific and Technical Information.

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.

MOOSE Reactor Module Meshing Enhancements to Support Reactor Physics Analysis

prepared by
E. Shemon, Y. Miao, S. Kumar, K. Mo, Y.S. Jung, A. Oaks
Argonne National Laboratory

G. Giudicelli, L. Harbour, R. Stogner
Idaho National Laboratory

September 15, 2022

EXECUTIVE ABSTRACT

The U.S. Department of Energy Office of Nuclear Energy Advanced Modeling and Simulation (NEAMS) program develops an integrated suite of advanced reactor physics tools built upon the Multiphysics Object-Oriented Simulation Environment (MOOSE) framework. Each code generally requires an input finite element mesh on which the physics solution is calculated, reported, and transferred to other physics codes. The meshing process is often burdensome for the complex geometries present in reactors due to lack of easy-to-use, open-source meshing tools.

To address the bottleneck associated with meshing complex geometries found in nuclear reactors, NEAMS initiated the development of the MOOSE Reactor Module starting in FY21. The Reactor Module builds off the existing MOOSE Mesh System to include targeted meshing capabilities such as the ability to generate hexagonal pin cells, assemblies with ducts, rotating control drums, cores, peripheral zones around a core, as well as the automatic labeling (“reporting IDs”) of pin, assembly, and planar zones to simplify post-processing of results. As a Physics Module in MOOSE, the Reactor Module is open-source, available with any MOOSE installation, directly compatible with MOOSE-based tools, and can be invoked from MOOSE-based applications to generate meshes. Functionality from the Reactor Module has been applied to several advanced reactor concepts to demonstrate user workflow improvements and accuracy. The primary objective of the Reactor Module is to improve useability of MOOSE-based tools by streamlining mesh generation and output inspection processes.

During FY22, the functionality of the Reactor Module (and accompanying Mesh System) has been expanded based on user needs. First, the Reactor Geometry Mesh Builder capability developed primarily in FY21 has been refactored and merged to the public MOOSE repository. This capability wraps underlying Reactor Module mesh generators into a “Pin – Assembly – Core” workflow appropriate for conventional Cartesian and hexagonal geometries, and notably assigns material IDs during mesh generation stage and generates only the minimal number of blocks needed in order to reduce computational burden. Biasing and boundary layer options have been added to the base mesh generators as required by thermal hydraulics solvers. The reporting ID functionality has been expanded to differentiate ring-wise and azimuthal sectors within a pin for use with depletion algorithms, and VectorPostProcessor and Reporter objects are now available to integrate solution variables across zones based on ID combinations. Functionality to trim hexagonal meshes along the center or periphery has been developed so users may leverage symmetry and reflective boundary conditions to reduce the mesh size. A flexible and powerful tool to fill the space between two sidesets has been introduced to the framework and can be used for transition layers such as stitching two assemblies together with different numbers of pins, or for complex geometries which do not follow conventional Cartesian/hexagonal patterns. Finally, additional verification problems were performed with NEAMS physics tools in complement with existing NEAMS work.

ACKNOWLEDGEMENTS

This work was funded by the Department of Energy Nuclear Energy Advanced Modeling and Simulation Program (DOE-NEAMS) under the Multiphysics Applications Technical Area. Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Nuclear Energy, under contract DE-AC02-06CH11357. This manuscript has also been authored by Battelle Energy Alliance, LLC under Contract No. DE-AC07-05ID14517 with the US Department of Energy.

The Argonne authors gratefully acknowledge the feedback, collaboration, and code reviews provided by the Idaho MOOSE Framework team (specifically Idaho National Laboratory co-authors Guillaume Giudicelli, Roy Stogner, Logan Harbour as well as Cody Permann and Derek Gaston). The authors are grateful for the collaboration and input on priorities by staff working under the NEAMS Multiphysics Applications Technical Area on specific reactor concepts including Nicolas Stauff and Nicholas Wozniak, as well as staff in the Reactor Physics (Changho Lee, Javier Ortensi, Yaqi Wang, Vincent Laboure) and Thermal Fluids (April Novak) Technical Areas of NEAMS. Finally, the authors thanks Scott Richards for his early work in developing the Reactor Geometry Mesh Builder capability.

Table of Contents

| | |
|--|----------|
| EXECUTIVE ABSTRACT | I |
| ACKNOWLEDGEMENTS..... | II |
| TABLE OF CONTENTS | III |
| LIST OF FIGURES | V |
| LIST OF TABLES | VIII |
| 1 INTRODUCTION | 1 |
| 2 STAKEHOLDER ENGAGEMENT AND PRIORITIES | 2 |
| 3 MOOSE REACTOR MODULE DEVELOPMENT | 4 |
| 3.1 GRIFFIN-RELATED ITEMS | 4 |
| 3.1.1 <i>Migration of Griffin Mesh Generators to Reactor Module</i> | 4 |
| 3.1.2 <i>Automatic Compilation of Griffin with Reactor Module</i> | 4 |
| 3.2 HEXAGONAL MESHING ENHANCEMENTS | 4 |
| 3.2.1 <i>SimpleHexagonGenerator: Extension to Quad Meshes</i> | 4 |
| 3.2.2 <i>Biasing and Boundary Layers for PolygonConcentricCircleMeshGenerator</i> | 5 |
| 3.2.3 <i>TriPinHexAssemblyGenerator: Assembly with Corner Pins</i> | 6 |
| 3.3 CARTESIAN MESHING ENHANCEMENTS | 9 |
| 3.3.1 <i>Add rotation option to PCCMG in support of Cartesian patterns</i> | 9 |
| 3.4 AXIAL MESHING ENHANCEMENTS | 9 |
| 3.4.1 <i>Mapping FancyExtruder to AdvancedExtruderGenerator</i> | 9 |
| 3.4.2 <i>Axial Meshing Bias and Extra Element IDs</i> | 9 |
| 3.4.3 <i>Boundary ID Remapping</i> | 10 |
| 3.4.4 <i>Interface Boundaries</i> | 11 |
| 3.4.5 <i>On-The-Fly Inverted Element Fixing</i> | 12 |
| 3.5 CORE PERIPHERY MESHING ENHANCEMENTS | 12 |
| 3.5.1 <i>PeripheralRingMeshGenerator: Quadrilateral option with boundary layer and biasing</i> | 12 |
| 3.5.2 <i>PeripheralTriangleMeshGenerator Updates</i> | 14 |
| 3.6 DEVELOPMENT OF MULTI-PURPOSE TRANSITION LAYER MESHING TOOLS | 14 |
| 3.6.1 <i>FillBetweenPointVectorsTools</i> | 14 |
| 3.6.1.1 Fundamentals | 14 |
| 3.6.1.2 Single-Layer Transition Layer Meshing | 15 |
| 3.6.1.3 Multi-Layer Transition Layer Meshing | 15 |
| 3.6.1.3.1 Surrogate Node Interpolation Algorithm | 15 |
| 3.6.1.3.2 Weighted Surrogate Nodes | 16 |
| 3.6.1.3.3 Quadrilateral Element Transition Layer in a Special Case | 17 |
| 3.6.1.4 Applications of FillBetweenPointVectorsTools | 17 |
| 3.6.2 <i>FillBetweenPointVectorsGenerator</i> | 18 |
| 3.6.3 <i>FillBetweenSidesetsGenerator</i> | 21 |
| 3.6.4 <i>PatternedHexPeripheralModifier: Assembly Stitchability Tool</i> | 22 |
| 3.6.4.1 <i>Stitching Assemblies Using the Least Common Multiple Approach</i> | 22 |
| 3.6.4.2 <i>Modification of Peripheral Boundary to Allow Stitching</i> | 24 |
| 3.6.4.3 <i>Handling Reporting IDs</i> | 26 |
| 3.7 HEXAGON MESH TRIMMER | 27 |
| 3.7.1 <i>Peripheral Trimming</i> | 28 |
| 3.7.2 <i>Center Trimming</i> | 28 |
| 3.7.3 <i>Trimmability</i> | 29 |
| 3.7.4 <i>Handling Degenerate Quadrilateral Elements</i> | 30 |
| 3.8 RING AND SECTOR REPORTING IDS | 32 |
| 3.8.1 <i>PolygonConcentricCircleMeshGenerator</i> | 32 |

| | | |
|----------|--|-----------|
| 3.8.2 | <i>TriPinHexAssemblyGenerator</i> | 33 |
| 3.8.3 | <i>Utilization in Depletion ID Generation</i> | 34 |
| 3.9 | VECTOR POST PROCESSOR BASED ON REPORTING IDs | 34 |
| 3.10 | UPDATES TO REACTOR GEOMETRY MESH BUILDER (RGMB) CAPABILITIES | 35 |
| 3.10.1 | <i>Mesh Sub-Generator Objects in MOOSE</i> | 36 |
| 3.10.2 | <i>ReactorMeshParams</i> | 37 |
| 3.10.3 | <i>PinMeshGenerator</i> | 37 |
| 3.10.4 | <i>AssemblyMeshGenerator</i> | 39 |
| 3.10.5 | <i>CoreMeshGenerator</i> | 42 |
| 3.10.6 | <i>Periphery Mesh Generation</i> | 45 |
| 3.11 | USER SUPPORT..... | 47 |
| 4 | REACTOR GEOMETRY VERIFICATION EXAMPLES AND MESHES | 49 |
| 4.1 | HETEROGENEOUS LEAD-COOLED FAST REACTOR ASSEMBLY..... | 49 |
| 4.1.1 | <i>Mesh Generation</i> | 49 |
| 4.1.2 | <i>Computation and Results</i> | 51 |
| 4.2 | HETEROGENEOUS CARTESIAN LIGHT WATER REACTOR CORE | 53 |
| 4.3 | HEAT PIPE-COOLED MICROREACTOR (HPMR)..... | 55 |
| 4.3.1 | <i>2D Empire Model</i> | 55 |
| 4.3.2 | <i>3D HPMR Model</i> | 59 |
| 4.3.3 | <i>KRUSTY Heat-Pipe Microreactor Mesh</i> | 61 |
| 4.4 | 3D GAS-COOLED MICROREACTOR MODEL..... | 62 |
| 4.5 | MOLTEN-SALT REACTOR EXPERIMENT | 64 |
| 5 | SUMMARY AND FUTURE WORK | 66 |
| | REFERENCES | 67 |

LIST OF FIGURES

| | |
|--|----|
| Figure 3-1. SimpleHexagonGenerator triangle and (new) quadrilateral mesh options. | 5 |
| Figure 3-2 A schematic drawing showing the concepts of boundary layers..... | 6 |
| Figure 3-3 Polygon concentric circle mesh with boundary layers and biased main bodies. | 6 |
| Figure 3-4 A typical mesh generated by this TriPinHexAssemblyGenerator object with one large-circular-pin section, one small-circular-pin section, and one pin-free section. | 7 |
| Figure 3-5 Meshes generated by TriPinHexAssemblyGenerator. | 9 |
| Figure 3-6 Mesh biasing and element integer swapping options in AdvancedExtruderGenerator..... | 10 |
| Figure 3-7 An extruded mesh containing new boundary labels at various axial levels. | 11 |
| Figure 3-5 Definition of plane interface boundary ids..... | 12 |
| Figure 3-9 Core periphery meshing with quadrilateral elements..... | 13 |
| Figure 3-10 Peripheral ring mesh with a conformal inner boundary layer. | 13 |
| Figure 3-11 A schematic drawing showing the fundamental functionality of the FillBetweenPointVectorsTools | 15 |
| Figure 3-12 A schematic drawing showing the principle of single-layer transition layer meshing algorithm..... | 15 |
| Figure 3-13 A schematic drawing showing an example of surrogate node interpolation algorithm. | 16 |
| Figure 3-14 A schematic drawing showing an example of weighted surrogate node interpolation algorithm used for intermediate nodes generation when non-uniform distributed nodes are involved on the two original boundaries..... | 17 |
| Figure 3-15 Syntax of the FillBetweenPointVectorsTools | 17 |
| Figure 3-16 Some representative meshes generated by FillBetweenPointVectorsTools: (left) a transition layer mesh defined by two oppositely oriented arcs; (middle) a transition layer mesh defined by one arc and a complex curve; (right) a half-circle mesh..... | 18 |
| Figure 3-17 A schematic drawing showing different transition layer meshes generated between two arc boundaries: (left to right) very fine mesh, fine mesh, and coarse mesh; (top to bottom) uniformly distributed nodes, slightly biased nodes, and heavily biased nodes. | 19 |
| Figure 3-18 A schematic drawing showing different biasing option for sublayers: (left) non-bias; (middle) fixed biasing factor = 0.8; (right) automatic biased based on boundary nodes. 20 | 20 |
| Figure 3-19 A schematic drawing showing a transition layer meshed by quadrilateral elements..... | 20 |
| Figure 3-20. Syntax of FillBetweenPointVectorsGenerator. | 21 |
| Figure 3-21 Syntax and output from FillBetweenPointVectorsGenerator..... | 22 |
| Figure 3-22 A schematic drawing of an example assembly mesh with transition layer as its outermost mesh layer. | 25 |
| Figure 3-23 A schematic drawing showing a virtual core design with assemblies including 7, 19, 37 and 61 pins. | 25 |
| Figure 3-24 A close-up comparison between core meshes using (left) the new mesh generator and (right) the manual least common multiple approach..... | 26 |
| Figure 3-25 Different approaches to handle a reporting id: (Left) input mesh with reporting id (pin_id); (Middle) retained pin_id for transition layer; (Right) user provided pin_id value for transition layer..... | 27 |
| Figure 3-26 Syntax and output of PatternedHexPeripheralModifier | 27 |

| | |
|--|----|
| Figure 3-27 A schematic drawing showing different trimming schemes for a hexagonal mesh. | 28 |
| Figure 3-28 Example outputs of HexagonMeshTrimmer | 29 |
| Figure 3-29 Syntax and output for HexagonMeshTrimmer | 32 |
| Figure 3-30 Example of setting ring and sector reporting IDs inside PCCMG. | 33 |
| Figure 3-31 Rings and sectors colored by ID in hexagonal assembly | 33 |
| Figure 3-32 Rings and sectors colored by ID in hexagonal assembly | 33 |
| Figure 3-33 Setting sub-pin depletion IDs using DepletionIDGenerator | 34 |
| Figure 3-34 Integration of assembly and pin power using new VectorPostProcessor | 35 |
| Figure 3-35 Integration of assembly and pin power using new Reporter. | 35 |
| Figure 3-36 Creation of Cartesian pin using Reactor Geometry Mesh Builder's PinMeshGenerator | 39 |
| Figure 3-37 Creation of Cartesian assembly using Reactor Geometry Mesh Builder's AssemblyMeshGenerator depicting block IDs and Region IDs | 42 |
| Figure 3-38 Creation of Cartesian core using Reactor Geometry Mesh Builder's CoreMeshGenerator | 44 |
| Figure 3-38 Creation of 3D core with meshed peripheral region using Reactor Geometry Mesh Builder | 47 |
| Figure 3-40 Preliminary HTTR Mesh in collaboration with V. Laboure (INL) | 47 |
| Figure 3-41 (left) Target reactor assembly design showing center hole, ring of fuel, and circular sector at hexagon vertices, provided by M. Lindell (INL). (right) Initial mesh of fundamental features using the upcoming Delaunay triangulation mesh generator and the Reactor Module. | 48 |
| Figure 4-1 Reactor Geometry Mesh Builder input for LFR assembly | 50 |
| Figure 4-2. Top-down (a) and side (b) views of LFR heterogeneous assembly, created using RGMB objects in the Reactor Module. Each color represents a unique material region in the assembly. | 51 |
| Figure 4-3. Axially integrated pin-by-pin group 0 normalized scalar flux distribution for the LFR assembly example. | 53 |
| Figure 4-4. RGMB input for C5G6 Cartesian core example. | 55 |
| Figure 4-5 Region ID map of full-core C5G7 mesh (left) and zoomed in mesh discretization of MOX assembly (right). | 55 |
| Figure 4-6. Coarse meshes of Empire Core: (A) mesh generated using Argonne's Mesh Tools system; and (B) mesh generated using MOOSE meshgenerators | 56 |
| Figure 4-7. Fine meshes of Empire Core: (A) mesh generated using Argonne's Mesh Tools system and CUBIT; and (B) mesh generated using MOOSE meshgenerators | 56 |
| Figure 4-8. Construction of coarse Empire mesh | 57 |
| Figure 4-9. Construction of fine Empire mesh | 58 |
| Figure 4-10. Construction of 3D HPMR mesh | 60 |
| Figure 4-11. KRUSTY meshes: (a) half-core Cubit mesh; (b) cross-section of Cubit mesh near the fuel zone (core center); (c) full core mesh MOOSE mesh (a quarter of core was removed to show the internal structures); and (d) cross-section of MOOSE mesh near the fuel zone (core center). | 62 |
| Figure 4-12. KRUSTY meshes generated using MOOSE: (a) full core, (b) quarter core, and (c) 1/16 core meshes | 62 |
| Figure 4-. Construction of 3D GCMR assembly mesh | 63 |

Figure 4-14. MOOSE-generated MSRE mesh: (a) 3D full core (about a quarter of core was removed to show the internal structures); and (b) cross-section near the fuel zone (core center) and view of control rod zone..... 64

LIST OF TABLES

| | |
|---|----|
| Table 3-1. Manual selection of assembly discretization to ensure stitchability..... | 23 |
| Table 3-2. Global Parameters used in Reactor Geometry Mesh Builder | 37 |
| Table 3-3. Parameters used in Reactor Geometry Mesh Builder’s PinMeshGenerator..... | 38 |
| Table 3-4. Parameters used in Reactor Geometry Mesh Builder’s PinMeshGenerator..... | 40 |
| Table 3-5. Extra Element IDs Assigned in Reactor Geometry Mesh Builder’s AssemblyMeshGenerator | 41 |
| Table 3-6. Parameters in Reactor Geometry Mesh Builder’s CoreMeshGenerator..... | 42 |
| Table 3-7. Extra Element IDs Assigned in Reactor Geometry Mesh Builder’s CoreMeshGenerator | 43 |
| Table 3-8. Common parameters in CoreMeshGenerator for core periphery meshing..... | 45 |
| Table 3-9. Extra parameters in CoreMeshGenerator for core periphery meshing depending on type of mesh desired..... | 45 |
| Table 4-1. Griffin-computed k-effective results for LFR Assembly Problem: MOOSE mesh vs. Argonne Mesh Tools. | 52 |
| Table 4-2. Griffin-computed k-effective results for 2-D EMPIRE Problem: MOOSE mesh vs. CUBIT / Argonne Mesh Tools..... | 59 |

1 Introduction

The U.S. Department of Energy Office of Nuclear Energy Advanced Modeling and Simulation (NEAMS) program (Stanek, 2019) develops an integrated suite of advanced reactor physics tools built upon the Multiphysics Object-Oriented Simulation Environment (MOOSE) framework (Permann, et al., 2020).. Each code generally requires an input finite element mesh on which the physics solution is calculated, reported, and transferred to other physics codes. The meshing process is often burdensome for the complex geometries present in reactors due to lack of easy-to-use, open-source meshing tools.

To address the bottleneck associated with meshing complex geometries found in nuclear reactors, NEAMS initiated the development of the MOOSE Reactor Module starting in FY21 (Shemon, et al., 2021). The Reactor Module builds off the existing MOOSE Mesh System to include targeted meshing capabilities such as the ability to generate hexagonal pin cells, assemblies with ducts, rotating control drums, cores, peripheral zones around a core, as well as the automatic labeling (“reporting IDs”) of pin, assembly, and planar zones to simplify post-processing of results. As a Physics Module in MOOSE, the Reactor Module is open-source, available with any MOOSE installation, directly compatible with MOOSE-based tools, and can be invoked from MOOSE-based applications to generate meshes. Functionality from the Reactor Module has been applied to several advanced reactor concepts to demonstrate user workflow improvements and accuracy. The primary objective of the Reactor Module is to improve useability of MOOSE-based tools by streamlining mesh generation and output inspection processes.

During FY22, the functionality of the Reactor Module (and accompanying Mesh System) has been expanded based on user needs. First, the Reactor Geometry Mesh Builder capability developed primarily in FY21 has been refactored and merged to the public MOOSE repository. This capability wraps underlying Reactor Module mesh generators into a “Pin – Assembly – Core” workflow appropriate for conventional Cartesian and hexagonal geometries, and notably assigns material IDs during mesh generation stage and generates only the minimal number of blocks needed in order to reduce computational burden. Biasing and boundary layer options have been added to the base mesh generators as required by thermal hydraulics solvers. The reporting ID functionality has been expanded to differentiate ring-wise and azimuthal sectors within a pin for use with depletion algorithms, and VectorPostProcessor and Reporter objects are now available to integrate solution variables across zones based on ID combinations. Functionality to trim hexagonal meshes along the center or periphery has been developed so users may leverage symmetry and reflective boundary conditions to reduce the mesh size. A flexible and powerful tool to fill the space between two sidesets has been introduced to the framework and can be used for transition layers such as stitching two assemblies together with different numbers of pins, or for complex geometries which do not follow conventional Cartesian/hexagonal patterns. Finally, additional verification problems were performed with NEAMS physics tools in complement with existing NEAMS work. Software enhancements and verification problems related to the MOOSE Reactor Module are described in this technical report.

2 Stakeholder Engagement and Priorities

In the first year of Reactor Module development, priorities were gathered from the MOOSE framework team, Griffin (Lee C. , et al., 2021) developers, and Bison (Williamson, et al., 2021) developers. Ideas were also drawn from previous experience developing and using Argonne’s Mesh Tools (Smith & Shemon, 2015) which is a reactor-oriented meshing toolkit specifically for use with the PROTEUS finite element transport code (Shemon, Smith, & Lee, 2016). Initial work focused on addressing immediate meshing needs of the reactor physics community and resulted in a series of mesh generator capabilities for hexagonal geometry (pins, assemblies, cores, core periphery) as well as reporting ID functionality.

Stakeholder engagement expanded in FY22 to include developer/users of Thermal Fluids Technical Area codes, NEAMS multiphysics users (both inside and outside the program) working on specific reactor concepts, and research staff from the Nuclear Regulatory Commission. Oral presentations on new capabilities were given to NEAMS Senior Leadership, the Reactor Physics Technical Area, audience at the NEAMS Annual Review Meeting, the mid-year NEAMS Multiphysics Applications non-LWR focus meeting, and the Nuclear Regulatory Commission research group during FY22. Additional briefings were held with the NEAMS Workbench (Lefebvre, et al., 2019) development team as they incorporate meshing workflow into PyGriffin. The team coordinated and provided feedback on the general Delaunay triangulator capability led by INL’s MOOSE framework team in FY23. Meetings with the Illinois Rocstar Small Business Innovation Research (SBIR) awardee were held regularly to understand the meshing capabilities they intended to deliver. Finally, users were supported throughout the year with input creation, debugging, and implementation of new requested features.

The following work was prioritized based on stakeholder input:

- **Griffin-Related Items**
 - Move generic Griffin mesh generators to Reactor Module
 - Set up Griffin to utilize Reactor Module automatically
- **Hexagonal Meshing**
 - Add option to SimpleHexagonGenerator to generate 2 quads instead of 6 triangles
 - Add mesh biasing and boundary layer capabilities needed for thermal hydraulic calculations
 - Add option to mesh a tri-pin assembly with pins in each corner, needed for specialized geometries like the HTTR
 - Assembly trimming capability through center and peripheral pins, used for reducing mesh size due to symmetry and creation of unit cell geometries
- **Cartesian Meshing**
 - Add rotation option to PolygonConcentricCircleMeshGenerator to streamline Cartesian mesh patterning
- **Axial Meshing**
 - Add axial biasing options

- Permit definition of sidesets at each axial level as well as interfaces between axial levels
- On the fly inverted element fixing
- **Core periphery meshing**
 - Finalize merge request from FY21 which leverages the open-source poly2tri library (Hasse, 2021) to create triangulated core peripheries
 - Add quadrilateral meshing option to address element quality issues in triangulated case with heterogeneous assemblies
 - Add mesh biasing and boundary layer capability option in quad mesh option
- **Tool to Mesh Between Vectors of Points**
 - Develop versatile tool for meshing between two vectors of points (this arose as a fundamental capability needed for meshing more complex geometries as well as the transition layer capability for assemblies)
 - Enhanced assembly stitching capabilities to allow stitching of assemblies with differing edge discretization
- **Reporting IDs**
 - Finalize merge requests from FY21 which includes pin, assembly, planar reporting ID functionality as well as vector postprocessors
 - Add support for ring-wise and azimuthal sector IDs within a pin, needed for depletion zones
 - Add VectorPostProcessor to integrate solution variables across zones based on ID combinations
- **Reactor Geometry Mesh Builder (RGMB)**
 - Finalize pending merge request from FY21 which required a major refactor in order to maximize material assignment flexibility while minimizing number of blocks generated
 - Update with core periphery meshing capability
- **Reactor Verification Problems**
 - Exercise tools for a variety of reactor concepts and use cases to assess whether they are functioning properly and whether additional features are needed (e.g. lead cooled fast reactor, C5G7 for testing Cartesian geometry, Empire microreactor (DeHart, Ortensi, & Labouré, 2020) with rotating control drums, , gas-cooled microreactor concept, KRUSTY heat pipe cooled microreactor experiment, Molten Salt Reactor Experiment, upgrade Griffin verification problems to use Reactor Module tools)
- **User Support**
 - Direct support to users who reached out to the team for meshing help, ranging from helping them use existing tools to creating new features (in above list) to address their needs

3 MOOSE Reactor Module Development

This chapter describes the main software capabilities added to the MOOSE reactor module. Many of the capabilities mentioned here are expansions on capabilities previously described in detail in last year's report (Shemon, et al., 2021) and recent conference papers (Shemon, et al., 2022; Kumar, et al., 2022).

3.1 Griffin-Related Items

3.1.1 Migration of Griffin Mesh Generators to Reactor Module

A few custom mesh generators that were initially created for the Griffin reactor physics code were migrated to the MOOSE Reactor Module, since the functionality of these mesh generators were easily generalized to the typical MOOSE user instead of tying them solely to Griffin-based applications. The following mesh generators were migrated from Griffin to the open-source Reactor Module:

- `CoarseMeshExtraElementIDGenerator`: Assigns coarse element IDs to elements on a fine mesh, where the coarse element IDs are taken from a separate coarse mesh
- `ExtraElementIDCopyGenerator`: Copies an existing extra element ID to another extra element ID
- `SubdomainExtraElementIDGenerator`: Assigns extra elements IDs for elements on a mesh based on the subdomain ID of the mesh

3.1.2 Automatic Compilation of Griffin with Reactor Module

MOOSE-based applications are compiled with certain flags to allow for direct compilation and linking with MOOSE modules. Given the tight integration of mesh generators in the Reactor Module to reactor physics-based applications, the build system for the Griffin reactor physics code was updated to compile the Reactor module and link it to the Griffin executable by default. All mesh generators defined in the Reactor module are usable out of the box with the latest Griffin executables (post-December 2021), and unit tests have been put in place in Griffin to make sure the executable is able to call the Reactor module mesh generators. Additionally, examples of how the Reactor module can be used to define reactor mesh geometries for Empire (DeHart, Ortensi, & Labouré, 2020), C5G7 (Lewis, 2001), and Advanced Burner Test Reactor (Shemon, Grudzinski, Lee, Thomas, & Yu, 2015) have been built and placed in the directory `tests/moose_modules/reactor` of the Griffin source code.

3.2 Hexagonal Meshing Enhancements

3.2.1 `SimpleHexagonGenerator`: Extension to Quad Meshes

`SimpleHexagonGenerator` functionality was extended to include a two quadrilateral element option. This mesh generator is most typically used for generating homogeneous assemblies in nodal diffusion calculations. The two types of meshes possible with `SimpleHexagonGenerator` are depicted in Figure 3-1.

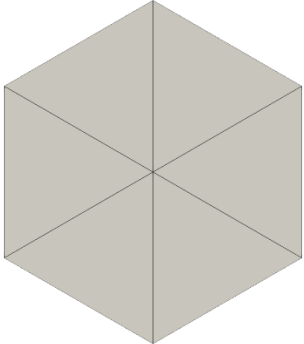
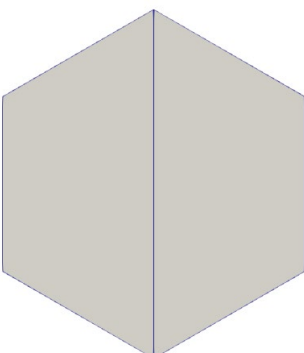
| | |
|--|--|
| <pre>[Mesh] [simplehex] type = SimpleHexagonGenerator hexagon_size = 0.146 hexagon_size_style = 'apothem' # optional block_id = 40 block_name = 'FuelAssembly' external_boundary_id = 9999 external_boundary_name = 'FuelBdry' [] []</pre> |  |
| <pre>[Mesh] [simplehex] type = SimpleHexagonGenerator hexagon_size = 0.146 hexagon_size_style = 'apothem' element_type=QUAD [] []</pre> |  |

Figure 3-1. SimpleHexagonGenerator triangle and (new) quadrilateral mesh options.

3.2.2 Biasing and Boundary Layers for PolygonConcentricCircleMeshGenerator

PolygonConcentricCircleMeshGenerator (PCCMG) was developed in FY21 with a uniform (unbiased) radial meshing option. Users of thermal hydraulic codes have requested boundary layer meshing capability and main body biased meshing. Boundary layer meshing is widely used in thermal hydraulics models to capture the detailed phenomena near domain interfaces (solid-fluid interface). In this region, dense and biased meshing is needed. For the main body, further from domain interfaces, coarse meshing usually works although sometimes a biased main body mesh is also needed.

To be specific, boundary layers and the main body are depicted in an example subdomain in Figure 3-2.

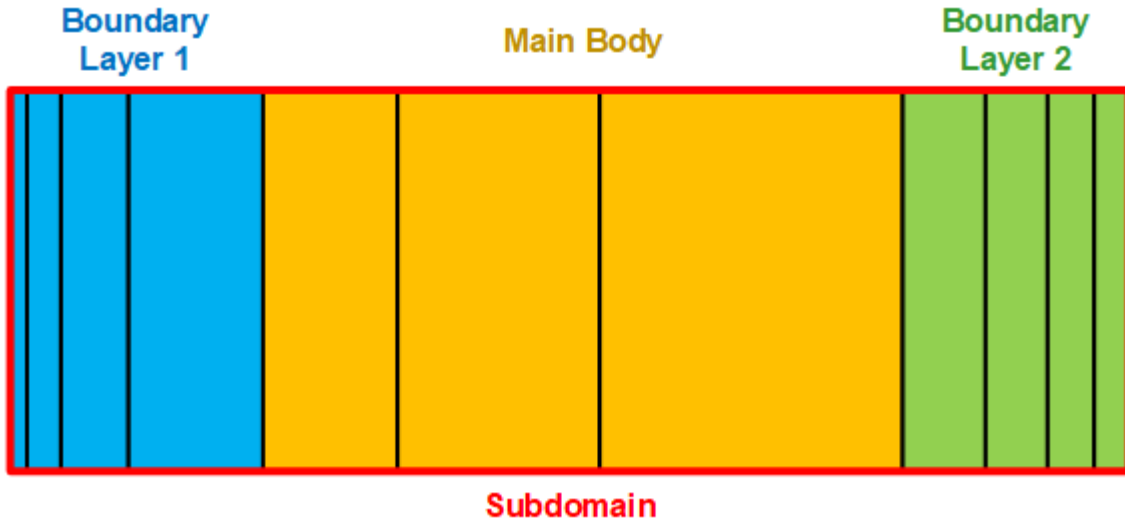


Figure 3-2 A schematic drawing showing the concepts of boundary layers

In the radial direction, both boundary layer and main body biased meshing features have been added to PolygonConcentricCircleMeshGenerator (see Figure 3-19 for an example).

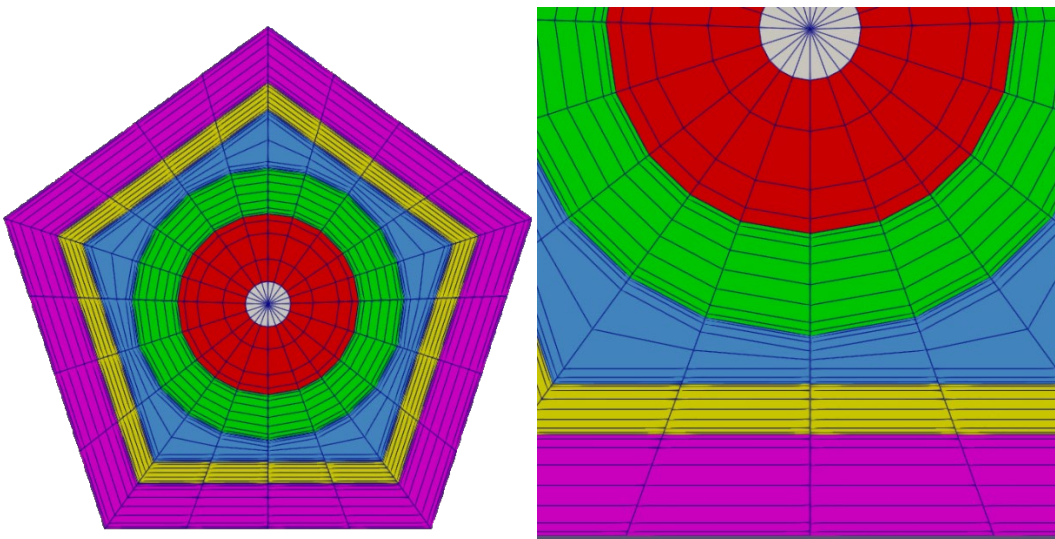


Figure 3-3 Polygon concentric circle mesh with boundary layers and biased main bodies.

Since PCCMG also generates Cartesian meshes, Cartesian geometry also benefits from the boundary layer and biasing enhancement.

3.2.3 TriPinHexAssemblyGenerator: Assembly with Corner Pins

A new `TriPinHexAssemblyGenerator` object generates a 2D hexagonal assembly mesh consisting of three diamond sections. Each of these sections may contain one pin defined as a series of concentric circles. An example of such an assembly mesh is illustrated in Figure 3-4. This type of geometry is known to occur in the High Temperature Test Reactor (HTTR) (Shiozawa, Fujikawa, Iyoku, Kunitomi, & Tachibana, 2004).

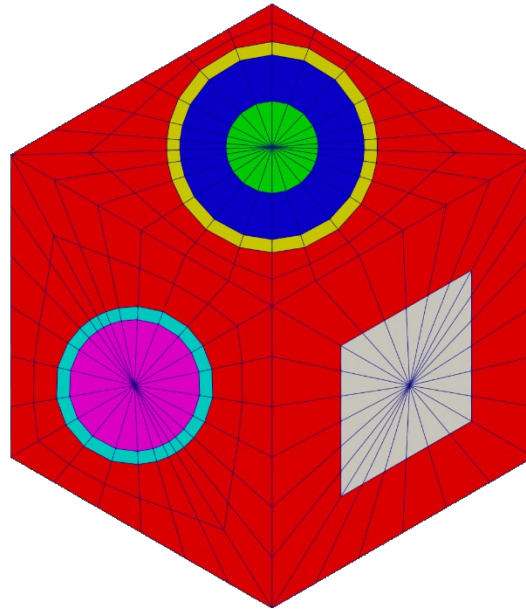


Figure 3-4 A typical mesh generated by this `TriPinHexAssemblyGenerator` object with one large-circular-pin section, one small-circular-pin section, and one pin-free section.

The size of the assembly is defined by `"hexagon_size"`. Users can input either radius (which is the same as side length for a hexagon) or apothem of the hexagon by setting `"hexagon_size_style"`. On each side of the hexagon, the azimuthal meshing density is controlled by `"num_sectors_per_side"`. The nodes on each side are uniformly distributed.

The hexagon is naturally divided azimuthally into three diamond sections. The first diamond section has an optional pin at 12 o'clock (90 degrees) from the center of the hexagon; the second diamond section has an optional pin at 8 o'clock (210 degrees); and the third diamond section has an optional pin at 4 o'clock (330 degrees). `"ring_radii"` is a 2D vector parameter used to define concentric ring regions within the diamond sections from one through three. `"ring_intervals"` defines the number of radial meshing subintervals for each of the concentric rings. Optionally, `"ring_block_ids"` and `"ring_block_names"` can be used to assign block ids/names to these rings. For all the four aforementioned ring-related parameters, if only one vector is provided instead of three, the same ring parameters will be adopted for all three sections, providing a concise way to define 3 identical pins. The default center of a pin is halfway between the hexagon center and a vertex point (half the hexagon's radius). The center of the pin may be offset radially from the center of the diamond by a distance defined by `"ring_offset"`. A positive `"ring_offset"` means the center of the concentric rings is radially offset towards to the assembly boundary. A negative `"ring_offset"` means the center of the concentric rings is radially offset closer to the assembly center. Users can set `"preserve_volumes"` as true to correct the polygonization effect and preserve ring volume.

The nodes on each interface between the two diamond sections are also uniformly distributed with the same number of nodes determined by "num_sectors_per_side". Therefore, the azimuthal intervals of each diamond section are non-uniformly distributed and are determined based on three factors: "num_sectors_per_side", "ring_offset" and the constraints due to the uniformly distributed nodes on external sides as well as interfaces between diamond sections.

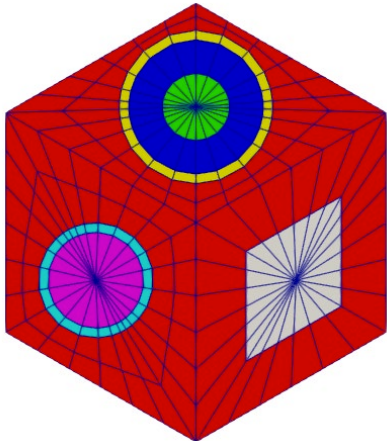
Each diamond section also contains a "background" region, which is the region outside the concentric rings (or the full diamond, if no rings are present). The background radial intervals and block id of each diamond can be defined by "background_intervals" and "background_block_ids"/"background_block_names". In most cases, "background_block_ids" and "background_block_names" have a length of one if provided. However, if there exists a least one ring-free section, lengths of "background_block_ids" and "background_block_names" need to be two to accommodate the additional triangular element region required when no pin exists at the center of the diamond.

The TriPinHexAssemblyGenerator generates a complete set of MeshMetaData needed for future stitching with other assemblies. Therefore, meshes generated by this object can be directly used in "inputs" of PatternedHexMeshGenerator to form a core mesh.

As mentioned, by default (i.e., "assembly_orientation" is set as pin_up), the first section is at 12 o'clock. The assembly can be rotated by 180 degrees by setting "assembly_orientation" as pin_down.

Optionally, users can also assign an element extra integer for each diamond sections. The name of the element extra integer is defined by "pin_id_name", while the assigned values of the three sections are defined by a three-element vector parameter, "pin_id_values". If "assembly_orientation" is set as pin_up, the first element of "pin_id_values" is assigned to the top section; the second element is assigned to the lower-left section; and the third element is assigned to the lower-right section. On the other hand, If "assembly_orientation" is set as pin_down, the first element of "pin_id_values" is assigned to the bottom section; the second element is assigned to the upper-right section; and the third element is assigned to the upper-left section.

More detailed syntax and output examples can be found in the following table.

| | |
|--|---|
| <pre>[Mesh] [assm_up] type = TriPinHexAssemblyGenerator ring_radii = '7 8;5 6; ' ring_intervals = '2 1;1 1; ' ring_block_ids = '200 400 600;700 800; ' background_block_ids = '30 40' num_sectors_per_side = 6 background_intervals = 2 hexagon_size = \${fparse 40.0/sqrt(3.0)} ring_offset = 0.6 external_boundary_id = 200 external_boundary_name = 'surface' [] []</pre> |  |
|--|---|

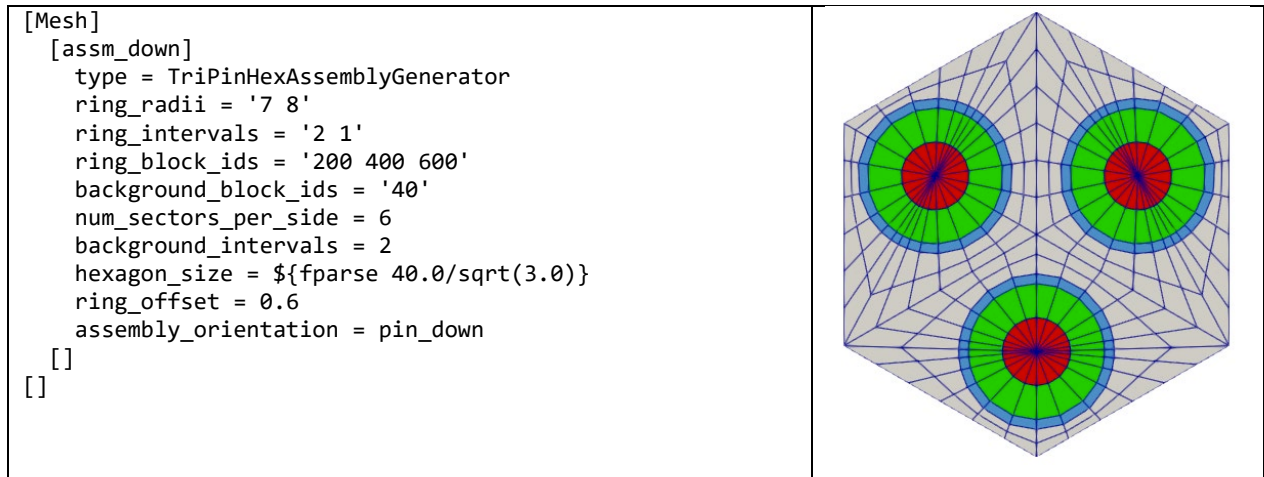


Figure 3-5 Meshes generated by TriPinHexAssemblyGenerator.

3.3 Cartesian Meshing Enhancements

3.3.1 Add rotation option to PCCMG in support of Cartesian patterns

By default, the polygon mesh generated by PCCMG is oriented vertex-up (aligned with the y -direction). In some applications including Cartesian grids, it is preferred that a flat side instead of a vertex should face up. Thus, the PCCMG has been updated to provide both orientation options. Users can use the new Boolean input parameter, `flat_side_up`, to control the polygon orientation.

3.4 Axial Meshing Enhancements

3.4.1 Mapping FancyExtruder to AdvancedExtruderGenerator

Based on MOOSE framework reviewer feedback, the mesh generator known as FancyExtruder has been renamed to AdvancedExtruderGenerator to clarify its purpose. A detailed documentation page was created for the renamed mesh generator, which did not previously exist. More importantly, a series of new features have been added, as detailed in the following sections. This merge request is still pending as of publication of this report, but when the change goes live, FancyExtruder will be supported post-merge temporarily with deprecation warnings.

3.4.2 Axial Meshing Bias and Extra Element IDs

In each layer of extrusion, the axial meshing density can now be biased using a specific growth factor (see Figure 3-6). Additionally, the original subdomain swap feature has been expanded to extra element integer IDs so that reporting IDs on the 2D input mesh can be maintained or changed during extrusion on each axial level.

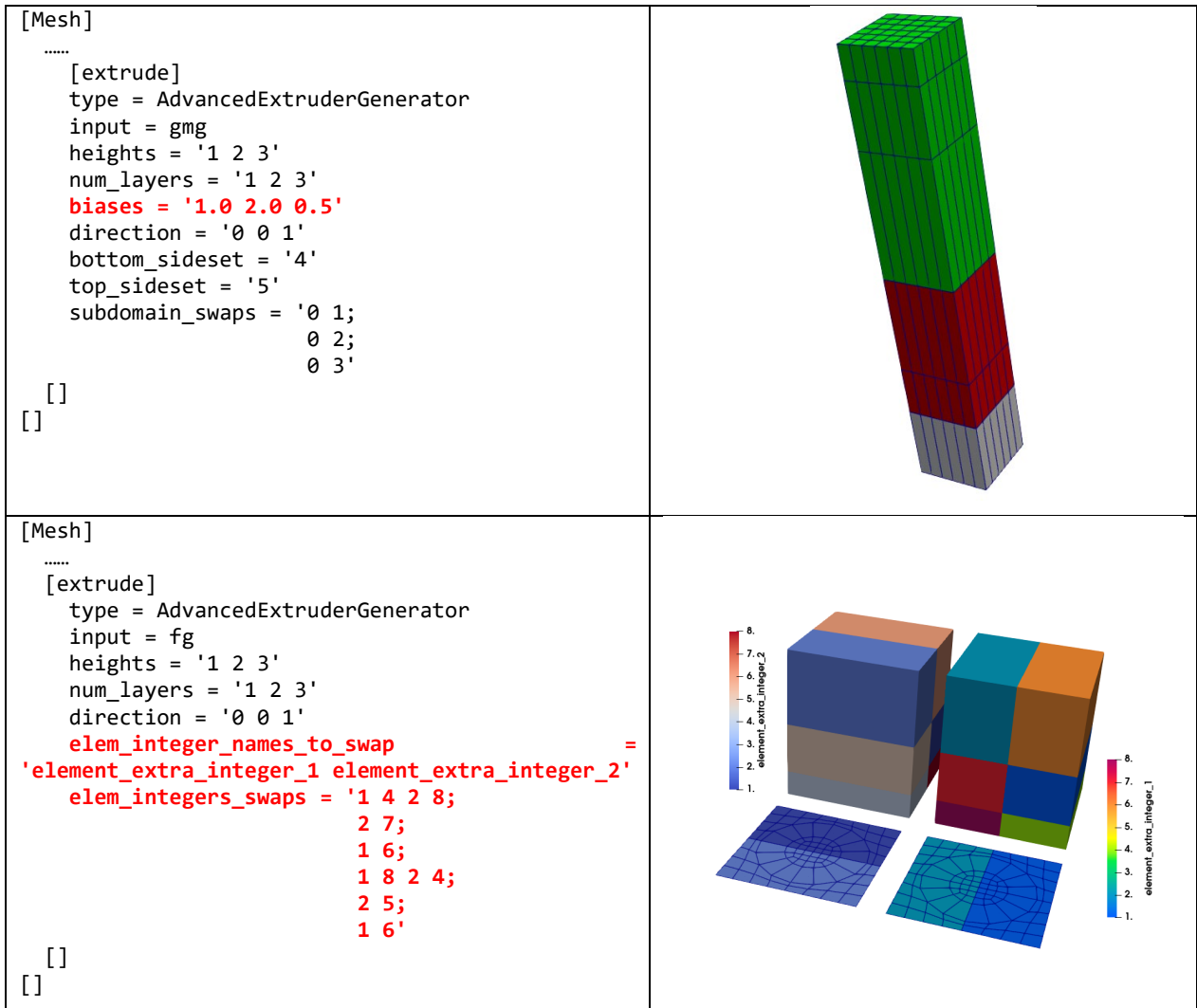


Figure 3-6 Mesh biasing and element integer swapping options in AdvancedExtruderGenerator.

3.4.3 Boundary ID Remapping

Boundary ID remapping has been implemented to work similarly to the previously existing subdomain ID remapping feature. During extrusion, the lower-dimension boundaries are also converted into higher-dimension boundaries. A double indexed array input parameter, "boundary_swaps", can be used to remap the boundary ids. Here, the boundary ids to be remapped must exist in the input mesh, otherwise, dedicated boundary defining mesh generators, such as SideSetsBetweenSubdomainsGenerator and SideSetsAroundSubdomainGenerator, need to be used to define new boundary ids along different axial heights. An example of using boundary ID remapping is provided in Figure 3-7.

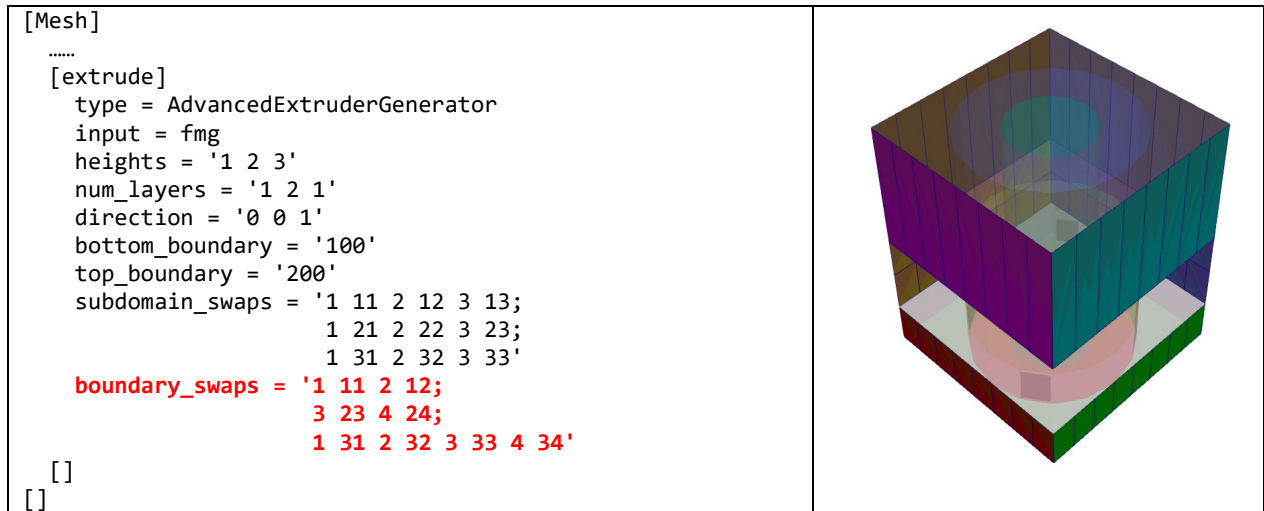


Figure 3-7 An extruded mesh containing new boundary labels at various axial levels.

Specification of sideset names within this mesh generator is not yet available, but users can presently leverage `RenameBoundaryGenerator` to assign sideset names.

3.4.4 Interface Boundaries

The other categories of the boundaries that can be defined are the interfaces between subdomains in different elevations, as well as the top/bottom surfaces of the subdomains. As each elevation interface (or top/bottom surface) is simply a duplicate of the input mesh, these interface (or top/bottom surface) boundaries correspond to the subdomains of the input mesh, which are referred to as `source_blocks`. These sidesets can be defined on either side of the elevation interface. Thus, both upward and downward boundaries can be defined. Here upward means the normal vector of the sideset has the "same-ish" direction as the "direction" vector; downward means the normal vector of the sideset has the "opposite-ish" direction as the "direction" vector. A potential use case of this is modeling thermo-mechanical contact at the fuel and heat pipe top/bottom interfaces in a heat-pipe cooled microreactor. There are likely to be additional use cases in thermal hydraulic applications.

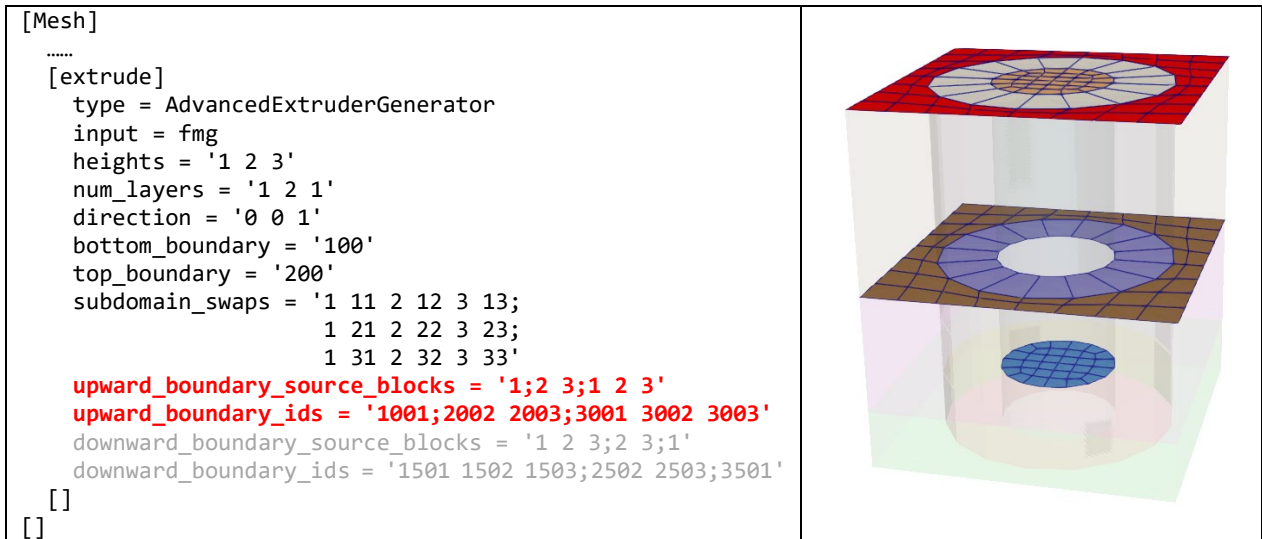


Figure 3-8 Definition of plane interface boundary ids.

3.4.5 On-The-Fly Inverted Element Fixing

The node numbering of an element must follow a certain convention, otherwise the element is regarded as inverted and cannot be used for simulation due to negative Jacobian. Such a faulty 3D element can be caused by extruding a 2D element along a direction vector that is not compatible with the 2D element's node numbering. To avert an unpleasant user experience with attempting to use a mesh with inverted elements, an on-the-fly element sanity checker has been added to automatically check and fix elements during the extrusion.

3.5 Core Periphery Meshing Enhancements

The core periphery terminology in this section refers to the irregularly shaped zone surrounding a patterned cluster of assemblies but contained within a typically circular outer boundary.

3.5.1 PeripheralRingMeshGenerator: Quadrilateral option with boundary layer and biasing

In FY21, the PeripheralTriangleMeshGenerator (PTMG) was developed to triangulate core periphery regions. This mesh generator utilizes the poly2tri library and works well for cores consisting of homogenized assemblies. Element quality issues were observed when using this algorithm with finely meshed assemblies (such as pin-heterogeneous assemblies), which motivated an alternative option using quadrilateral meshes.

PeripheralRingMeshGenerator (PRMG) was therefore created to mesh quadrilateral elements in the core periphery as shown in teal in Figure 3-9. The nodes on the core periphery are extended in straight lines normal to the outer cylinder to form element edges, resulting in higher quality elements for finely meshed cores. The number of layers between the core and periphery edge can be specified via the input.

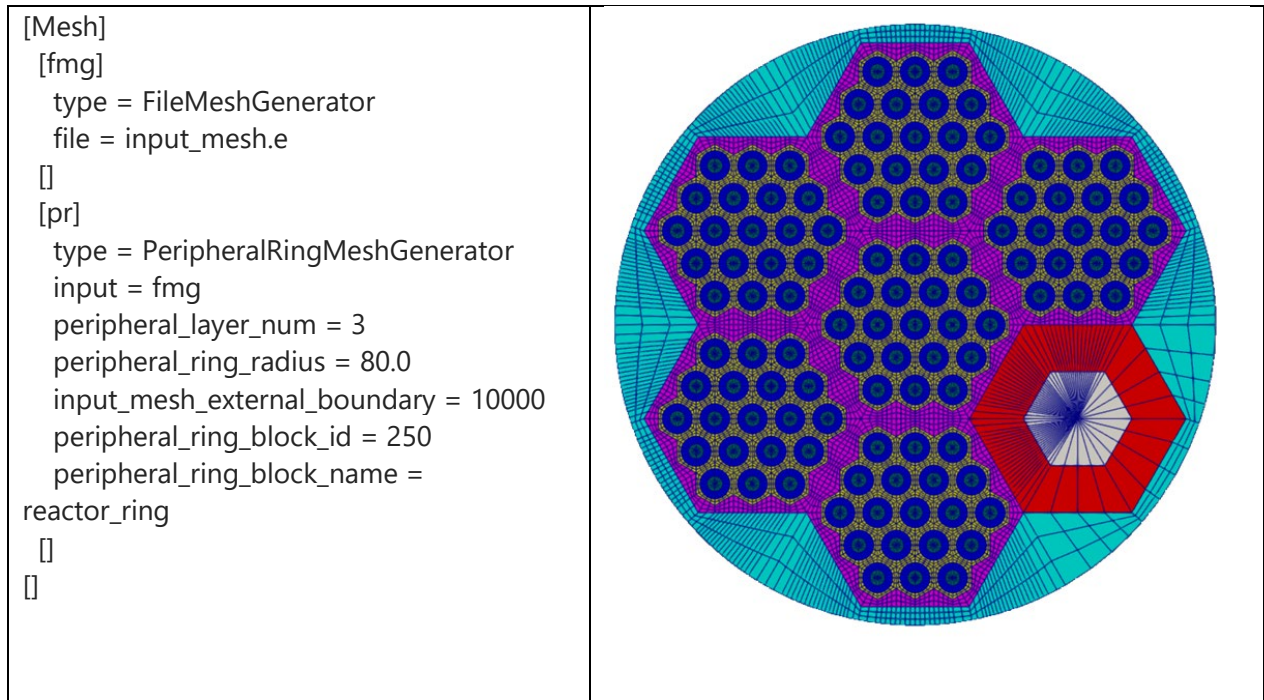


Figure 3-9 Core periphery meshing with quadrilateral elements.

Boundary layer and main body biasing features have also been added PeripheralRingMeshGenerator. The boundary layer adjacent to the input mesh's external boundary (i.e., inner side of the peripheral ring) needs to maintain consistent width all the way around the core, so a dedicated algorithm has been implemented to achieve such a functionality (see Figure 3-10 for an example).

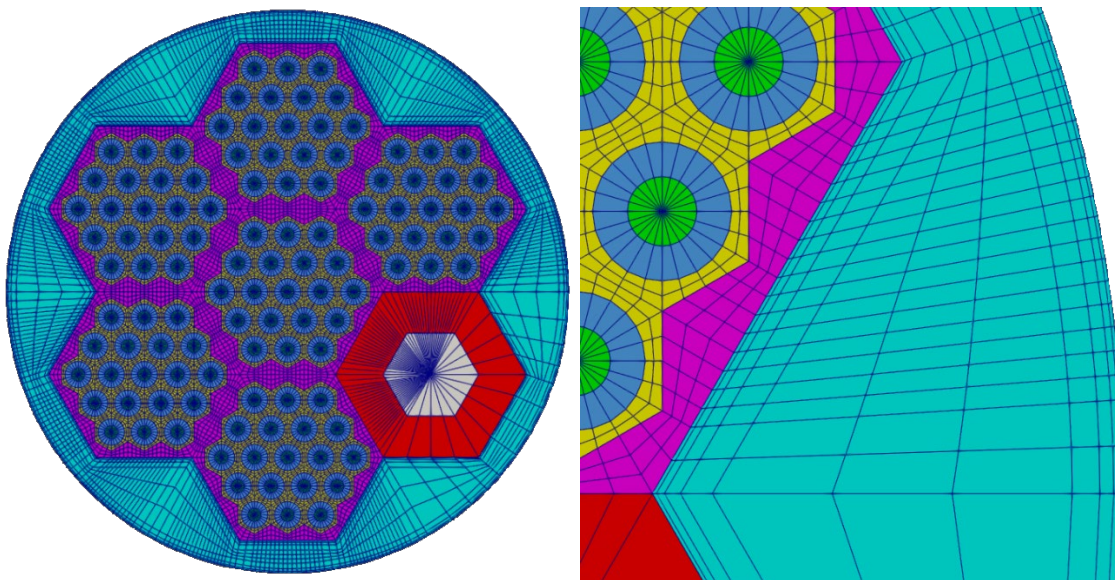


Figure 3-10 Peripheral ring mesh with a conformal inner boundary layer.

3.5.2 *PeripheralTriangleMeshGenerator Updates*

PeripheralTriangleMeshGenerator (PTMG) was initially developed in FY21 and fully merged during FY22. This mesh generator utilizes the poly2tri library to triangulate the core periphery region. As part of the integration of this mesh generator and PeripheralRingMeshGenerator into the Reactor Geometry Mesh Builder (RGMB) workflow, PTMG was extended to accept additional subdomain/boundary name parameters to provide consistent functionality with PRMG and provide a consistent interface to use either mesh generator in RGMB.

The element quality issues associated with this mesh generator are planned to be improved upon once the TriangleMeshGenerator (developed by the MOOSE team in FY22 which contains a generally Delaunay triangulator capability) is fully merged in MOOSE.

3.6 *Development of Multi-Purpose Transition Layer Meshing Tools*

In the process of developing a targeted capability to mesh transition layers between assemblies, a more general capability was first developed and modularized. This capability is called FillBetweenPointVectorsTools and described first. The tool turned out to have very useful applications beyond what was originally envisioned. The application of this tool to assist with assembly stitching is then described.

3.6.1 *FillBetweenPointVectorsTools*

FillBetweenPointVectorsTools contains tools that can be used to generate a triangular element transition layer mesh to connect two given curves (i.e., vectors of points) in the XY plane. It was originally developed for PeripheralModifyGenerator of the Reactor module. As these tools may also be useful for other applications, they are made available in this namespace.

3.6.1.1 *Fundamentals*

This tool set was designed to create a mesh for a transition layer. A transition layer accommodates the shape and node placement of two pre-existing boundaries and fills the gap between them with elements. The most important input data needed to generate a transition layer is the node positions of the two boundaries. The generated mesh conforms to these two boundaries and connects the end nodes of each boundary using a straight line, as indicated in Figure 3-11.

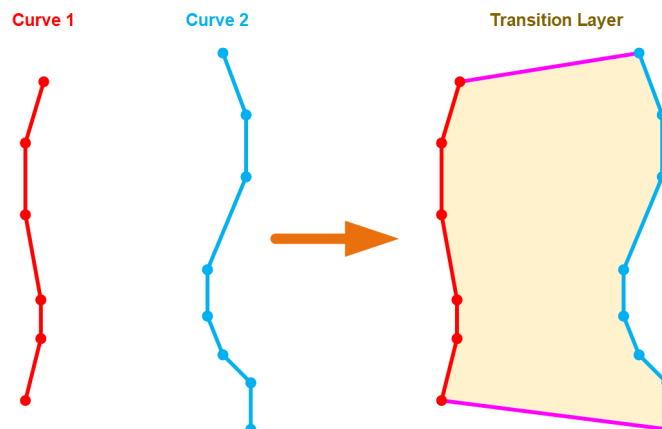


Figure 3-11 A schematic drawing showing the fundamental functionality of the FillBetweenPointVectorsTools

3.6.1.2 Single-Layer Transition Layer Meshing

The most straightforward solution is to create a single layer of triangular elements as the transition layer. A triangular element is created by selecting and connecting three vertices from the two sets of boundary nodes. One node is selected from one of the two pre-existing boundaries and two nodes are selected from the other boundary. The selection of the nodes should minimize the length of sides connecting the two boundaries. This algorithm is illustrated in Figure 3-12.

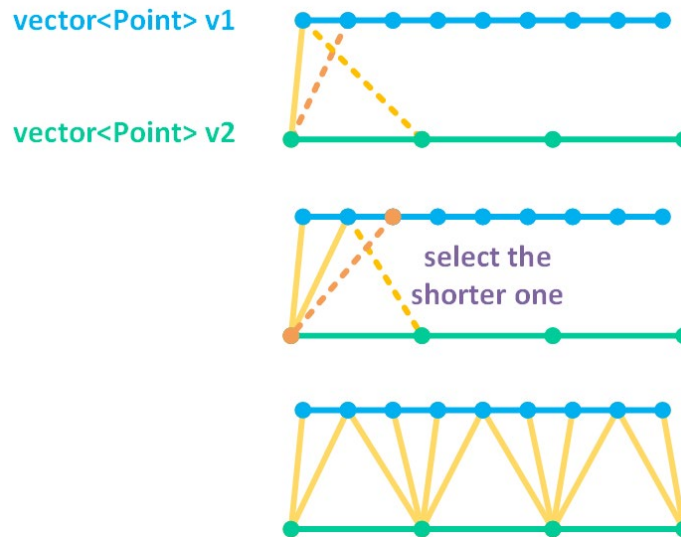


Figure 3-12 A schematic drawing showing the principle of single-layer transition layer meshing algorithm.

3.6.1.3 Multi-Layer Transition Layer Meshing

In many cases, more than one layer of triangular elements is desired to improve mesh quality. The generation of a transition layer containing multiple sublayers can be done by repeating the single-layer transition layer meshing steps once the nodes of the intermediate sublayers are generated. Thus, the key procedure here is to create those intermediate nodes based on the two given vectors of nodes on the input boundaries. Here, the algorithm to generate the nodes for each sublayer is described from the simplest case to the most generalized scenario.

3.6.1.3.1 Surrogate Node Interpolation Algorithm

Surrogate node interpolation algorithm is the most fundamental method used in this tool set for intermediate node generation. For simplicity, assume a case where all the nodes on each boundary are uniformly distributed. (Namely, the distance between neighboring nodes within a boundary is equal.) Assume that the two boundaries have M nodes (Side 1) and N nodes (Side 2), respectively, and that there are K sublayers of elements in between. From Side 1 to Side 2, using arithmetic progression, the k th layer of intermediate nodes have $S = [M + k(N - M) / K]$ nodes. To get the positions of these nodes, surrogate nodes are first calculated on the two input boundaries using interpolation leveraging MOOSE's LinearInterpolation utility.

Here, take Side 1 as an example. As mentioned above, Side 1 has M nodes, the coordinates of which are (x_0, y_0, z_0) , (x_1, y_1, z_1) , ..., $(x_{M-1}, y_{M-1}, z_{M-1})$. To get interpolated coordinates of the nodes on Side 1, the coordinate parameters $\{x_i\}$ and $\{y_i\}$ will be the dependent variables of interpolation (i.e., Y in the LinearInterpolation of MOOSE), while the X was set as $\{0, 1/(M-1), 2/(M-1), \dots, (M-2)/(M-1), 1\}$ (equal intervals). Note that $\{z_i\}$ does not need interpolation as we are working in the XY plane. For an intermediate layer with S , SS surrogate nodes are created on Side 1 using the aforementioned interpolation data and the following X values $\{0, 1/(S-1), 2/(S-1), \dots, (S-2)/(S-1), 1\}$. Meanwhile, another S surrogate nodes are created on Side 2 using a similar approach. Finally, the positions of the S intermediate nodes can be calculated by further interpolating the surrogate nodes created on the two boundaries. An example of applying surrogate node interpolation algorithm to a boundary with 9 uniformly distributed nodes and a boundary with 4 uniformly distributed nodes to generate an intermediate node layer with six nodes is illustrated in Figure 3-13.

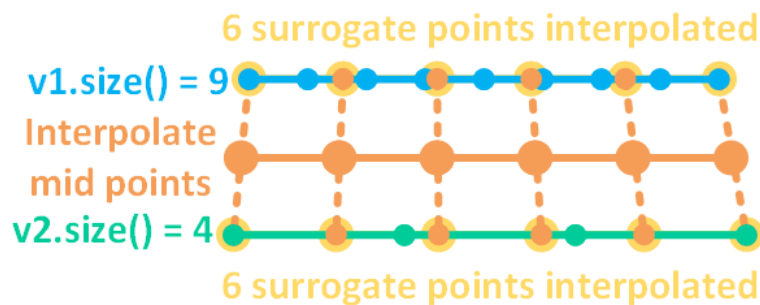


Figure 3-13 A schematic drawing showing an example of surrogate node interpolation algorithm.

Blue and green nodes belong to the original boundaries; yellow nodes are surrogate nodes generated by linear interpolation on the two original boundaries; and orange nodes are the produced intermediate layer nodes calculated by interpolating the surrogate nodes on the two boundaries.

3.6.1.3.2 Weighted Surrogate Nodes

A more general scenario is that the nodes on the two original boundaries are not uniformly distributed. In that case, weights need to be used during the linear interpolation for surrogate node generation. Again, given a boundary (Side 1) with M nodes, $\{^1p_0, ^1p_1, \dots, ^1p_{M-2}, ^1p_{M-1}\}$, the distance between the neighboring nodes are $\{^1l_1, ^1l_2, \dots, ^1l_{M-2}, ^1l_{M-1}\}$. The total length of Side 1 is L , which is the summation of $\{^1l_i\}$. This boundary can be mapped to a boundary with uniformly distributed nodes. For the new boundary, each segment has a weight $^1w_i = (M-1)l_i/L$. Surrogated nodes can then be generated on the new boundary using the same approach as mentioned in the previous subsection. After that, using the weights calculated before, the surrogate nodes are derived to weighted surrogate nodes. After repeating these steps on Side 2, the intermediate nodes can be generated. These procedures are visualized in Figure 3-14.

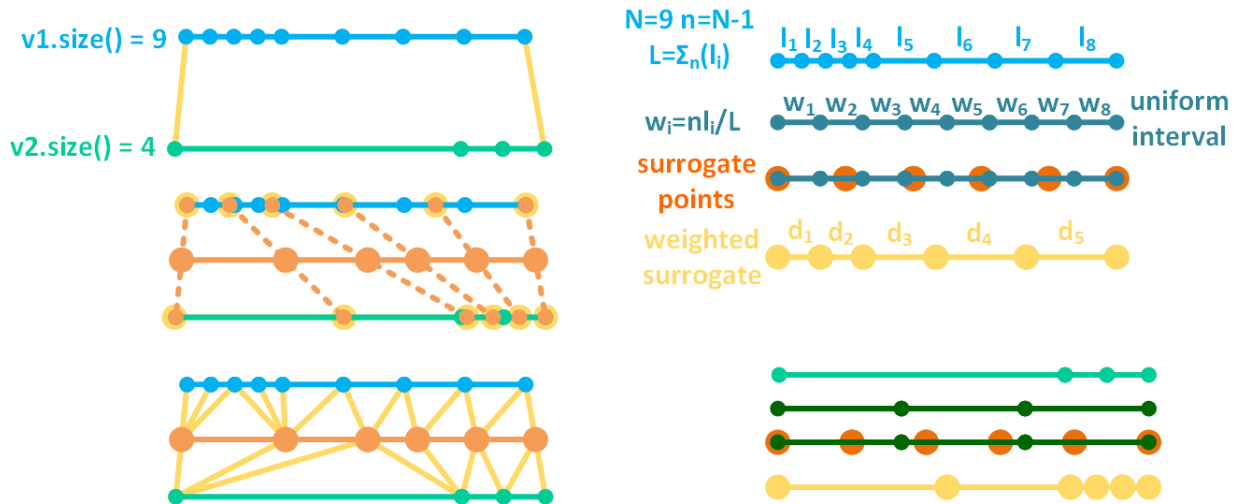


Figure 3-14 A schematic drawing showing an example of weighted surrogate node interpolation algorithm used for intermediate nodes generation when non-uniform distributed nodes are involved on the two original boundaries.

3.6.1.3.3 Quadrilateral Element Transition Layer in a Special Case

FillBetweenPointVectorsTools is generally designed for meshing with triangular elements because of their flexibility in accommodating complex node distribution. However, if Side 1 and Side 2 boundaries have the same number of nodes, then the transition layer can be meshed using quadrilateral elements straightforwardly. FillBetweenPointVectorsTools is equipped with this special quadrilateral meshing capability.

3.6.1.4 Applications of FillBetweenPointVectorsTools

FillBetweenPointVectorsTools is a utility for developers and users cannot directly access it. We will later describe a mesh generator which uses this utility.

In FillBetweenPointVectorsTools, the transition layer generation functionality is provided as a method shown as follows:

```
void fillBetweenPointVectorsGenerator(ReplicatedMesh & mesh,
    const std::vector<Point> boundary_points_vec_1,
    const std::vector<Point> boundary_points_vec_2,
    const unsigned int num_layers,
    const subdomain_id_type transition_layer_id,
    const boundary_id_type input_boundary_1_id,
    const boundary_id_type input_boundary_2_id,
    const boundary_id_type begin_side_boundary_id,
    const boundary_id_type end_side_boundary_id,
    const std::string type,
    const std::string name,
    const bool quad_elem = false,
    const Real bias_parameter = 1.0,
    const Real sigma = 3.0);
```

Figure 3-15 Syntax of the FillBetweenPointVectorsTools

Here, `mesh` is a reference `ReplicatedMesh` to contain the generated transition layer mesh; `boundary_points_vec_1` and `boundary_points_vec_2` are vectors of nodes for Side 1 and Side 2 boundaries; `num_layers` is the number of element sublayers; `transition_layer_id` is the subdomain ID of the generated transition layer elements; `input_boundary_1_id` and `input_boundary_2_id` are the IDs of the boundaries of the generated transition layer mesh corresponding to the input Sides 1 and 2, respectively; `begin_side_boundary_id` and `end_side_boundary_id` are the IDs of the other two boundaries of the generated transition layer mesh that connect the starting and ending points of the two input Sides; and `type` and `name` are the class type and object name of the mesh generator calling this method for error message generation purpose.

If `boundary_points_vec_1` and `boundary_points_vec_2` have the same size, `quad_elem` can be set as true so that quadrilateral elements instead of triangular elements are used to construct the transition layer mesh. In addition, `bias_parameter` can be used to control the meshing biasing of the element sublayers. By default, a non-biased sublayer meshing (i.e., equally spaced) is selected by setting `bias_parameter` as 1.0. Any positive `bias_parameter` is used as the manually set biasing factor, while a zero or negative `bias_parameter` activates automatic biasing, where the local node density values on the two input boundaries are used to determine the local biasing factor. If automatic biasing is selected, `sigma` is used as the Gaussian parameter to perform Gaussian blurring to smoothen the local node density to enhance robustness of the algorithm.

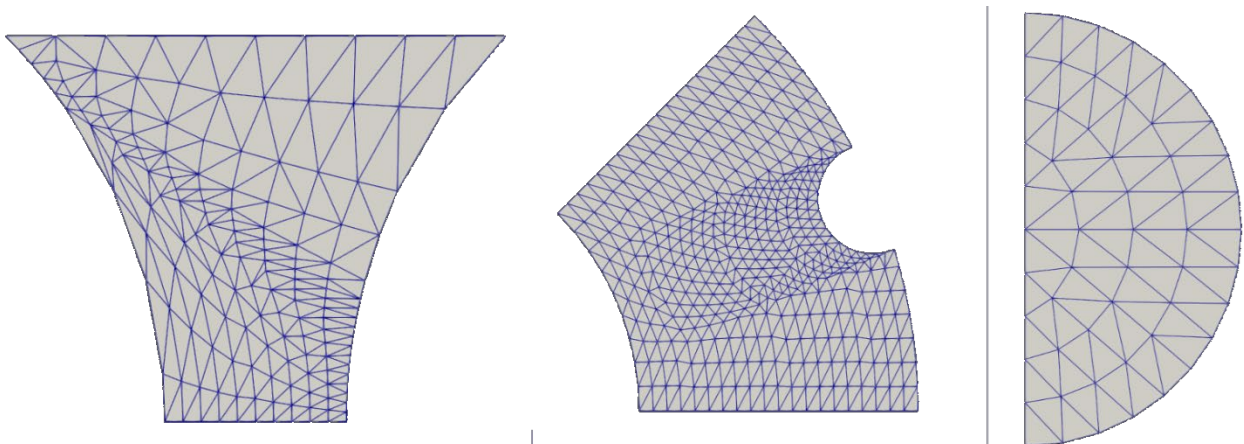


Figure 3-16 Some representative meshes generated by `FillBetweenPointVectorsTools`: (left) a transition layer mesh defined by two oppositely oriented arcs; (middle) a transition layer mesh defined by one arc and a complex curve; (right) a half-circle mesh.

One application of this tool is to generate a mesh with two curves and two straight lines as its external boundaries. As shown in Figure 3-16, a series of simple and complex shapes can be meshed.

3.6.2 *FillBetweenPointVectorsGenerator*

The `FillBetweenPointVectorsGenerator` class uses the fundamental functionalities of `FillBetweenPointVectorsTools` and is accessible to users. Therefore, this class provides a testing tool for `FillBetweenPointVectorsTools`, as well as a generalized platform for users to create meshes

using the tool set. Users are required to provide the three major inputs needed to use `FillBetweenPointVectorsTools`:

- `"positions_vector_1"` and `"positions_vector_2"`: the vectors of points on the two boundaries (i.e., Side 1 and Side 2).
- `"num_layers"`: number of element sublayers.

Aside from these fundamental input parameters, users can also assign block and the external boundary IDs through `"block_id"` and `"input_boundary_{1,2}_id"`, `"{begin,end}_side_boundary_id"`.

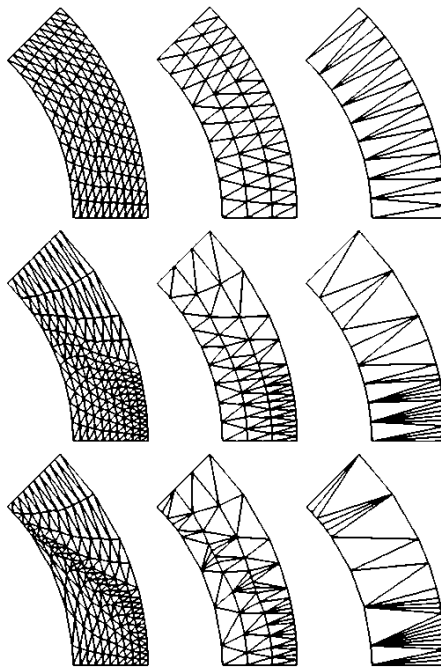


Figure 3-17 A schematic drawing showing different transition layer meshes generated between two arc boundaries: (left to right) very fine mesh, fine mesh, and coarse mesh; (top to bottom) uniformly distributed nodes, slightly biased nodes, and heavily biased nodes.

In general, `FillBetweenPointVectorsGenerator` handles many different scenarios. As shown in Figure 3-17, non-uniformly distributed boundary nodes (i.e., biased) may be input. The mesh generator does have some limitations. For example, the two input curves cannot intersect each other; and the interpolated nodes should not lead to flipped elements or overlapped elements. Due to the complexity of geometry, the mesh generator may not produce an error message in all the problematic cases. Users should cautiously examine the generated mesh by setting `"show_info"` as true and by running a simple diffusion problem.

The spacings of element sublayers can be biased by setting `"bias_parameter"`. Any positive `"bias_parameter"` is directly used as the fixed mesh biasing factor with the default value 1.0 for non-bias. By setting `"bias_parameter"` as 0.0, automatic biasing will be used, where the local node density values on the two input boundaries are used to determine the local biasing factor (see Figure

3-18 as an example). In that case, Gaussian blurring is used to smoothen the local node density to enhance stability of the algorithm, which can be tuned through "gaussian_sigma".

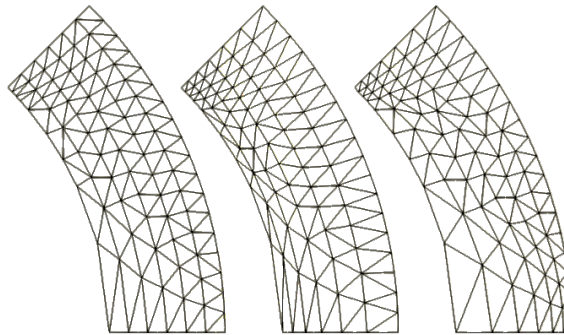


Figure 3-18 A schematic drawing showing different biasing option for sublayers: (left) non-bias; (middle) fixed biasing factor = 0.8; (right) automatic biased based on boundary nodes.

In some special cases, when "positions_vector_1" and "positions_vector_2" have the same length, users can set "use_quad_elements" as true to construct the transition layer mesh using quadrilateral elements (see Figure 3-19 as an example).

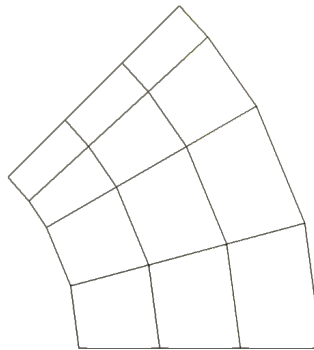


Figure 3-19 A schematic drawing showing a transition layer meshed by quadrilateral elements.

More detailed syntax and output examples can be found in the following table.

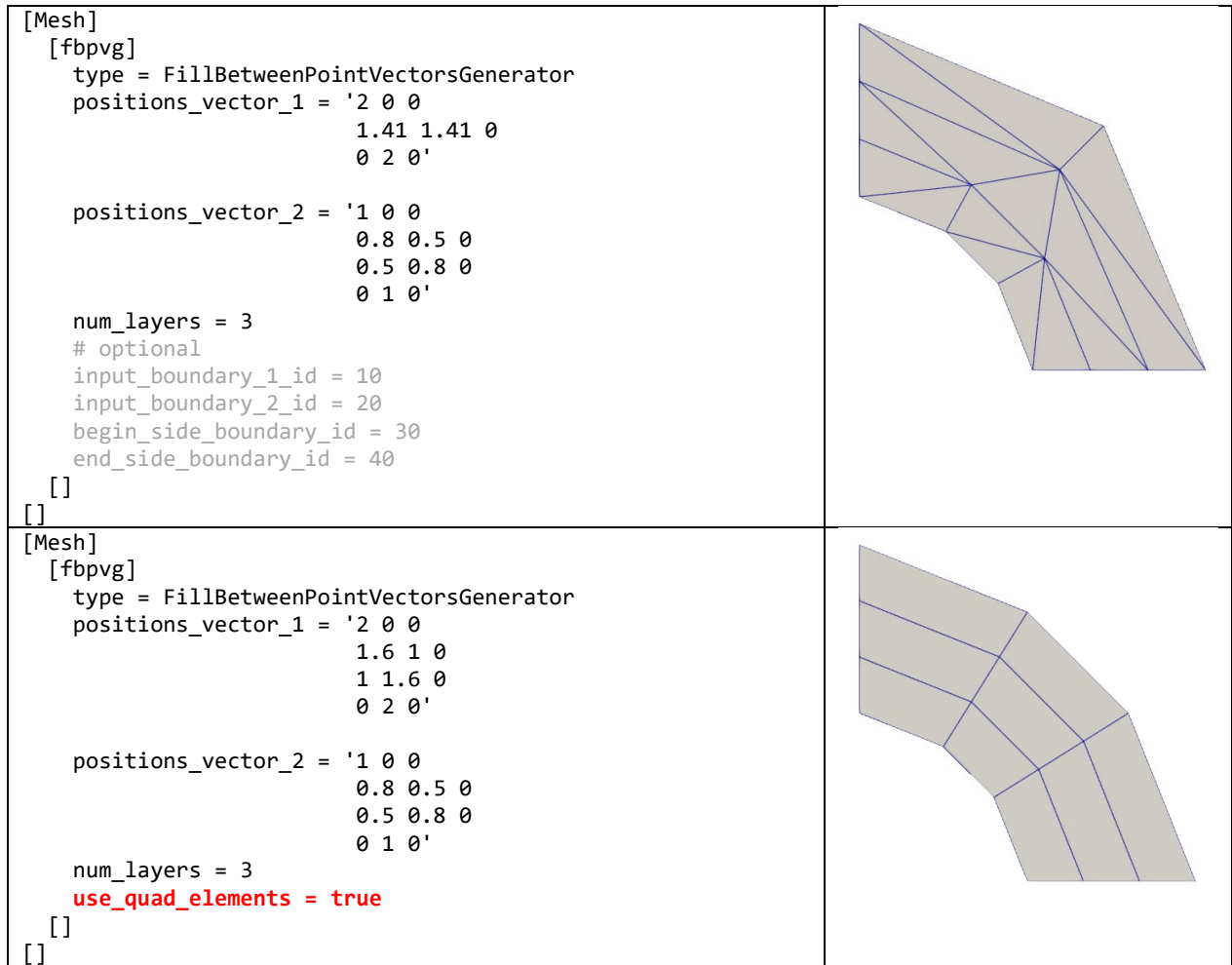


Figure 3-20. Syntax of FillBetweenPointVectorsGenerator.

3.6.3 FillBetweenSidesetsGenerator

The FillBetweenSidesetsGenerator offers similar functionality to FillBetweenPointVectorsGenerator by leveraging the FillBetweenPointVectorsTools utility. Instead of *manually* inputting the two boundaries "positions_vector_1" and "positions_vector_2", The FillBetweenSidesetsGenerator directly takes boundary information ("boundary_1" and "boundary_2") of two *input meshes*, "input_mesh_1" and "input_mesh_2". The input meshes can be translated using "mesh_1_shift" and "mesh_2_shift". The generated transition layer mesh can be output as a standalone mesh or a stitched mesh with the input meshes, depending on "keep_inputs". If "keep_inputs" is set as true, the original boundaries of the input meshes defined by "boundary_1" and "boundary_2" are deleted after stitching the input meshes with the generated transition layer mesh.

All the other meshing options are the same as FillBetweenPointVectorsGenerator. The syntax and output mesh can be found

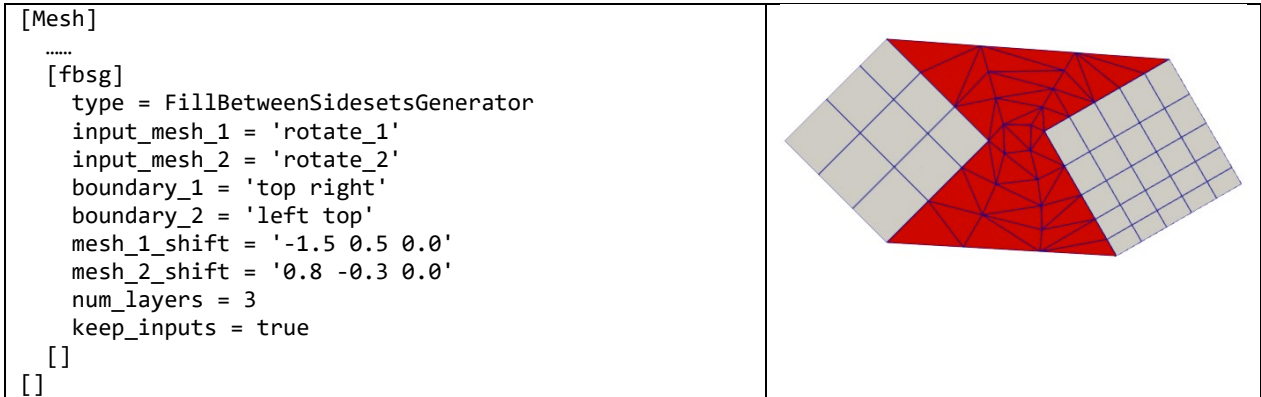


Figure 3-21 Syntax and output from FillBetweenPointVectorsGenerator.

3.6.4 PatternedHexPeripheralModifier: Assembly Stitchability Tool

The PatternedHexPeripheralModifier class utilizes FillBetweenPointVectorsTools to replace the outmost layer of quad elements of a 2D hexagonal assembly mesh generated by PatternedHexMeshGenerator (or its derived class HexIDPatternedMeshGenerator) with a transition layer consisting of triangular elements so that the assembly mesh can have nodes on designated positions on the external boundary. This boundary modification facilitates the stitching of hexagonal assemblies which have different node numbers on their outer periphery due to differing numbers of interior pins and/or different azimuthal discretization.

3.6.4.1 Stitching Assemblies Using the Least Common Multiple Approach

When PatternedHexMeshGenerator is used to generate a hexagonal assembly mesh, the number of nodes on each hexagon side follows a pre-determined formula based on the pin cell meshes which comprise the assembly. The pin cell hexagonal meshes used by PatternedHexMeshGenerator have a uniform even number of sectors per side (e.g., $2M$) given by "num_sectors_per_side" in PolygonConcentricCircleMeshGenerator. When PatternedHexMeshGenerator creates a hexagonal bundle with N ($N > 1$) pins on each side of the outermost ring, there are $2M \cdot (2N - 1)$ sectors and $2M \cdot (2N - 1) + 1$ nodes on each side of the hexagonal assembly mesh. If all the assemblies within a reactor core contain identical numbers of pins, it is straightforward to make assembly meshes stitchable with each other by using the same M number for the azimuthal discretization of each pin cell.

However, if a reactor core includes assemblies with different numbers of pins, M must be wisely selected based on the least common multiple of $4N_i - 2$ (i is the assembly index) of all the assemblies involved. This approach may be practical in cases where two assembly types with different pin numbers are involved, as shown in the following table:

Table 3-1. Manual selection of assembly discretization to ensure stitchability.

| Assm. #1 Number of pins per side N_1 | Assm. #1 Number of azimuthal intervals per sector M_1 | Assm. #2 Number of pins per side N_2 | Assm. #2 Number of azimuthal intervals per sector M_2 | Required Nodes on Assembly Side |
|--|---|--|---|--|
| 2 | 5 | 3 | 3 | 30 |
| 2 | 7 | 4 | 3 | 42 |
| 2 | 3 | 5 | 1 | 18 |
| 2 | 11 | 6 | 3 | 66 |
| 2 | 13 | 7 | 3 | 78 |
| 3 | 7 | 4 | 5 | 70 |
| 3 | 9 | 5 | 5 | 90 |
| 3 | 11 | 6 | 5 | 110 |
| 3 | 13 | 7 | 5 | 130 |
| 4 | 9 | 5 | 7 | 126 |
| 4 | 11 | 6 | 7 | 154 |
| 4 | 13 | 7 | 7 | 182 |
| 5 | 11 | 6 | 9 | 198 |
| 5 | 13 | 7 | 9 | 234 |
| 6 | 13 | 7 | 11 | 286 |

However, when multiple different assemblies with unique numbers of pins are involved in a reactor core, the "num_sectors_per_side" (i.e., $2M$) may be impractically large, as indicated in the following table:

| Assm. #1 N_1 | Assm. #1 M_1 | Assm. #2 N_2 | Assm. #2 M_2 | Assm. #3 N_3 | Assm. #3 M_3 | Required Nodes on Assembly Side |
|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|------------------------------------|
| 2 | 35 | 3 | 21 | 4 | 15 | 210 |
| 2 | 15 | 3 | 9 | 5 | 5 | 90 |
| 2 | 55 | 3 | 33 | 6 | 15 | 330 |
| 2 | 21 | 4 | 9 | 5 | 7 | 126 |
| 2 | 77 | 4 | 33 | 6 | 21 | 462 |
| 2 | 33 | 5 | 11 | 6 | 9 | 198 |
| 3 | 63 | 4 | 45 | 5 | 35 | 630 |
| 3 | 77 | 4 | 55 | 6 | 35 | 770 |
| 3 | 99 | 5 | 55 | 6 | 45 | 990 |
| 4 | 99 | 5 | 77 | 6 | 63 | 1386 |

3.6.4.2 Modification of Peripheral Boundary to Allow Stitching

The `PatternedHexPeripheralModifier` class modifies assembly meshes so that assemblies with different number of pins can be stitched together without increasing the mesh fidelity to an impractically fine fidelity (as shown in the previous section). This mesh generator only works with the "input" mesh created by `PatternedHexMeshGenerator`. Users must specify the external boundary of the input assembly mesh through "input_mesh_external_boundary". Given this input, the mesh generator identifies and deletes the outmost layer of elements and uses the newly formed external boundary as one of the two vectors of boundary nodes needed by `FillBetweenPointVectorsTools` after symmetry reduction. In addition, uniformly distributed nodes are placed along the original external boundary of the mesh and defined as the second vector of boundary nodes needed by `FillBetweenPointVectorsTools`. The number of new boundary nodes is specified using "new_num_sector". Thus, the outmost layer of the assembly mesh can be replaced with a triangular element transition layer mesh that can be easily stitched with another transition layer mesh. An example of the assembly mesh modified by this mesh generator is shown in the following figure.

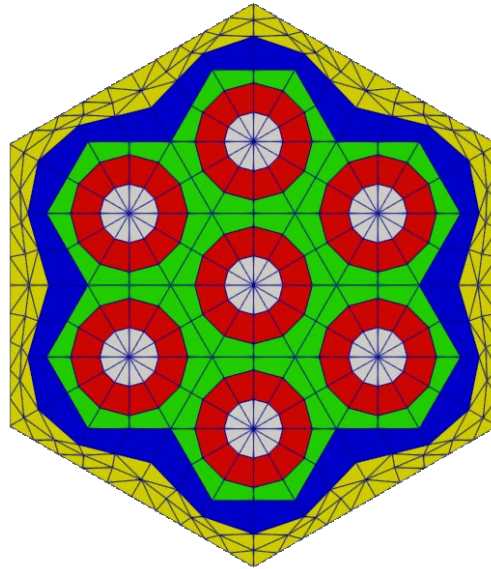


Figure 3-22 A schematic drawing of an example assembly mesh with transition layer as its outmost mesh layer.

This mesh generator forces the number of nodes on a hexagonal mesh to match a user-specified input, which allows assemblies with different number of pins or azimuthal discretizations (and consequently different numbers of boundary nodes) to be stitched together without increasing the mesh density to an unreasonable level.

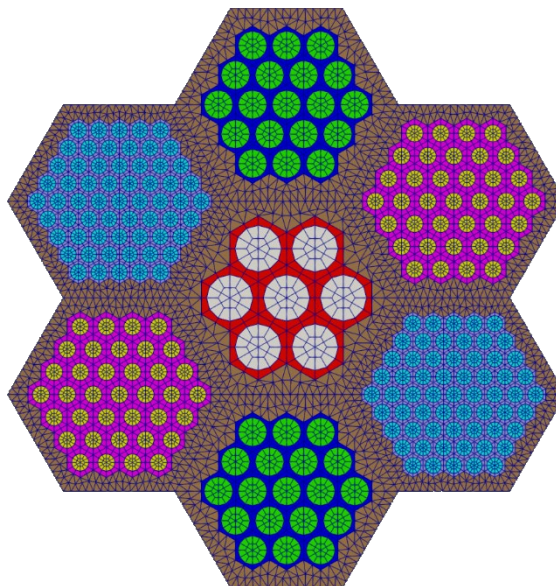


Figure 3-23 A schematic drawing showing a virtual core design with assemblies including 7, 19, 37 and 61 pins.

Figure 3-23 illustrates a core comprising four types of assemblies. This mesh generator's functionality was leveraged to force a common mesh density on each hexagonal assembly side (16 nodes on each assembly side) so that the assemblies can be easily stitched. In the absence of this mesh generator, the least common multiple approach would require 631 nodes on each assembly side as shown in Figure 3-24. The mesh density would be increased dramatically just to ensure stitchability, showing the prominent advantage of using this mesh generator instead of the least common multiple approach.

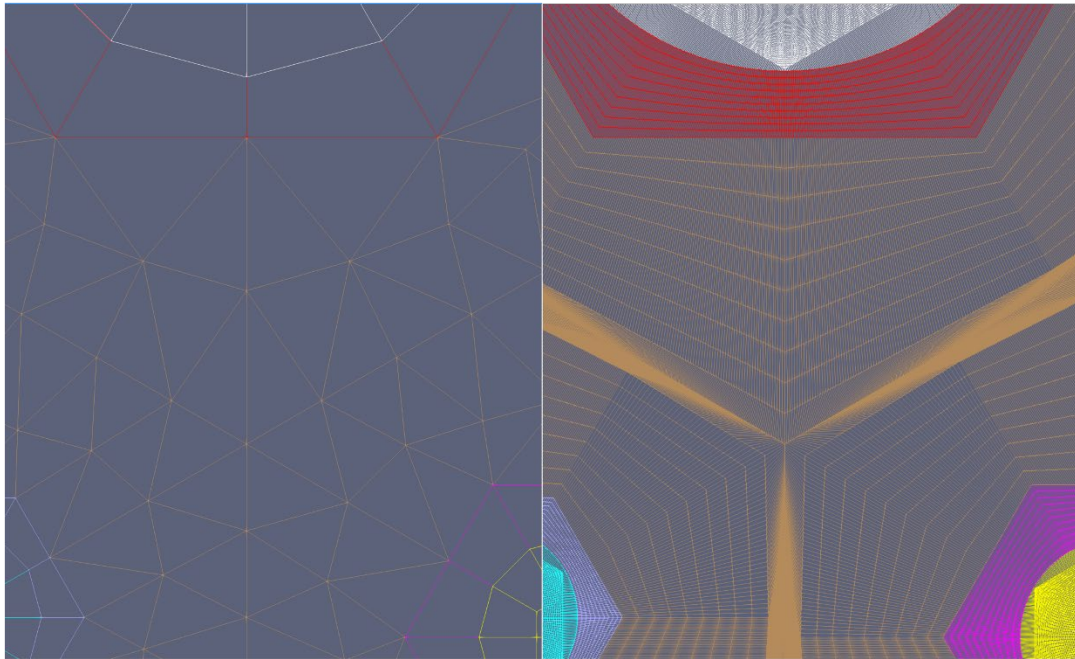


Figure 3-24 A close-up comparison between core meshes using (left) the new mesh generator and (right) the manual least common multiple approach.

3.6.4.3 Handling Reporting IDs

If the input mesh contains extra element integers (reporting IDs), the `PatternedHexPeripheralModifier` provides options to retain or reassign these reporting IDs (see Figure 3-25). By default, all the extra element integers existing on the input mesh are retained. Due to the nature of the transition layer which creates a new set of elements, the original boundaries between different reporting ID values have to be slightly shifted after modification. When `PatternedHexPeripheralModifier` assigns reporting ID values to a new element in the transition layer, it utilizes the reporting ID values of the original element that is nearest to the new element (based on centroid positions) to retain the setting of the input mesh. Alternatively, users can specify the names of reporting IDs to be reassigned through "extra_id_names_to_modify". The customized reporting ID values can then be set by providing the parameter "new_extra_id_values_to_assign".

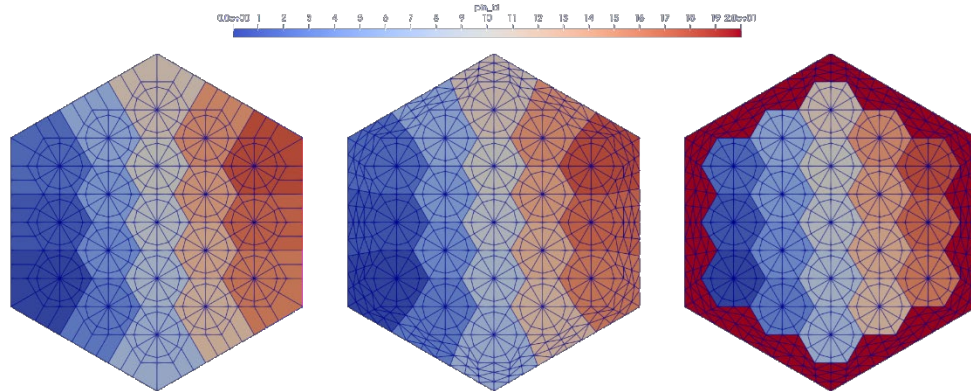


Figure 3-25 Different approaches to handle a reporting id: (Left) input mesh with reporting id (pin_id); (Middle) retained pin_id for transition layer; (Right) user provided pin_id value for transition layer.

More detailed syntax and output examples can be found in the following table.

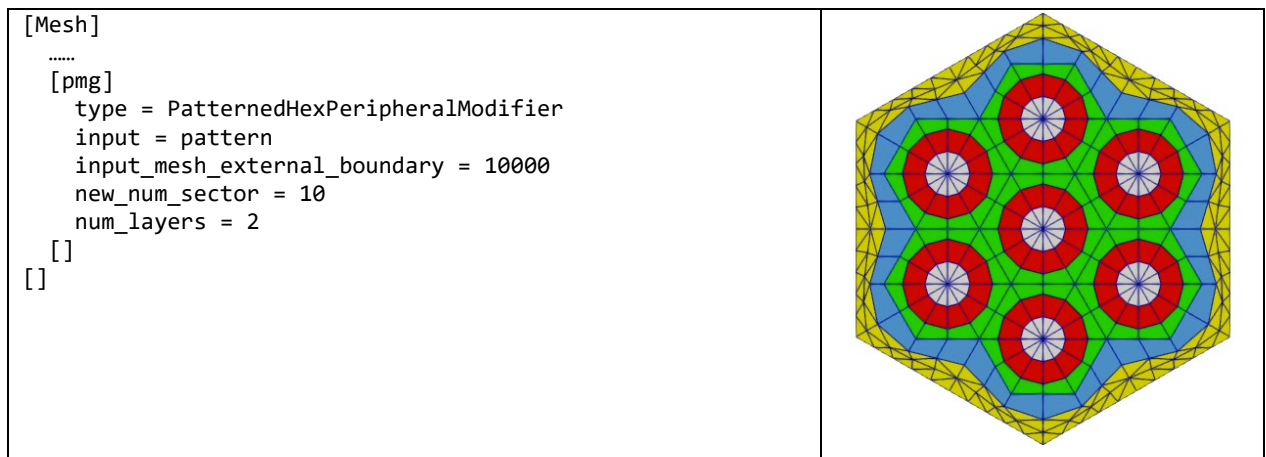


Figure 3-26 Syntax and output of PatternedHexPeripheralModifier

3.7 Hexagon Mesh Trimmer

The HexagonMeshTrimmer object takes the hexagonal mesh generated by PatternedHexMeshGenerator as "input" and trims off part of the mesh. The input mesh can also be PatternedHexMeshGenerator's output processed by PeripheralRingMeshGenerator or PatternedHexPeripheralModifier.

Two types of trimming can be performed by HexagonMeshTrimmer: Peripheral Trimming and Through-the-Center Trimming, which will be introduced separately as follows.

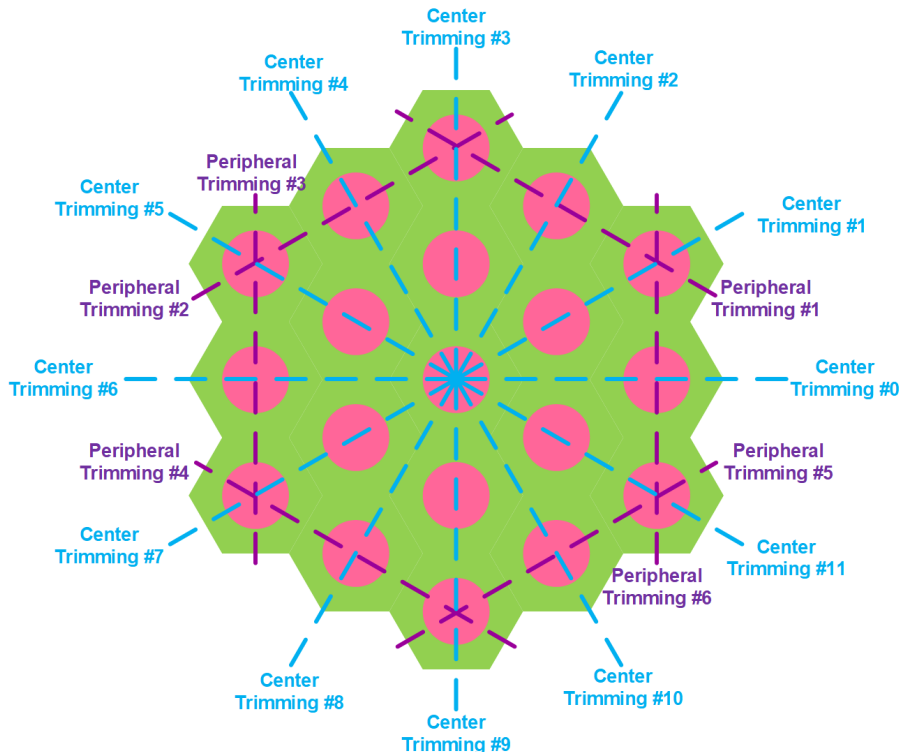


Figure 3-27 A schematic drawing showing different trimming schemes for a hexagonal mesh.

3.7.1 Peripheral Trimming

Peripheral trimming trims off peripheral region(s) of one or multiple sides of the input hexagonal assembly mesh. To be specific, for each side, half of the unit pin meshes are trimmed off, as shown by the purple lines in Figure 1. Each side of the hexagonal assembly is assigned an index as illustrated in Figure 1. Users can use "trim_peripheral_region" to set which sides need to be trimmed off (1) and which need to be retained (0). The setting "trim_peripheral_region" = "1 1 1 1 1 1" trims off all the sides (full peripheral trimming) to create a hexagonal assembly mesh with half-pins on each of the external 6 boundaries. Partial peripheral trimming may be employed for practical applications to create peripheral assemblies in a core whose normal interior assembly units contain half-pins on the boundary. The mesh metadata generated by PatternedHexMeshGenerator are retained, with pattern_pitch_meta updated to take trimming into consideration. The output of this object can be assembled into a patterned lattice using PatternedHexMeshGenerator.

3.7.2 Center Trimming

Center trimming removes azimuthal sectors from the input hexagonal assembly or core mesh. The mesh may be trimmed along lines of symmetry in the input mesh. Only certain hexagonal meshes are eligible to be trimmed by this object due to imposed symmetries (see Trimmability). Valid hexagonal input meshes may be trimmed at twelve possible center trimming lines, indexed from 0 to 11 as the blue lines shown in Figure 1. Each unit azimuthal sector is 30°. A practical application of center trimming is to reduce the domain size (and simulation scale) by leveraging symmetry through reflected boundary conditions. Therefore the largest possible output mesh after center

trimming contains six consecutive azimuthal sectors (i.e., half of the input mesh), while the smallest possible output mesh has only one azimuthal sector (i.e., one twelfth of the input mesh). This mesh trimmer object *retains* any sectors which are included between the trimming line defined by "center_trim_starting_index" to the trimming line defined by "center_trim_ending_index" swept out in a counterclockwise direction. Other sectors are discarded.

3.7.3 Trimmability

In general, HexagonMeshTrimmer trims meshes generated by PatternedHexMeshGenerator. An assembly mesh consisting of patterned pin meshes has both peripheral and center trimmability; whereas a core mesh consisting of patterned assembly meshes only has center trimmability. Two mesh metadata entries (peripheral_trimmability and center_trimmability) are created by PatternedHexMeshGenerator to tell HexagonMeshTrimmer which trimming options are valid. In the absence of these two meta data, HexagonMeshTrimmer will throw an incompatible error message.

In addition, PeripheralRingMeshGenerator and PatternedHexPeripheralModifier, which apply quadrilateral and triangle (respectively) peripheral meshes, retain these two mesh meta data from the input mesh so that valid meshes generated by PeripheralRingMeshGenerator may also be trimmed.

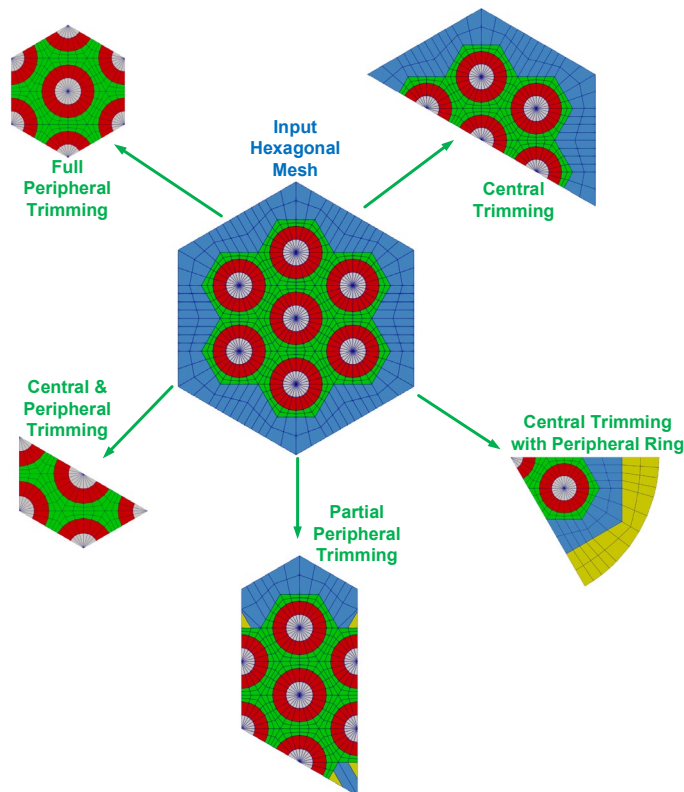


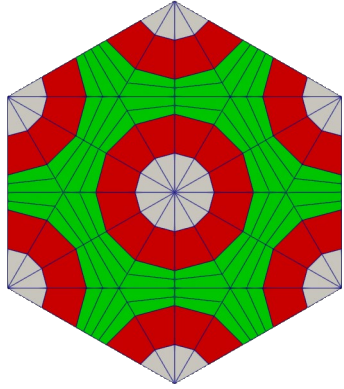
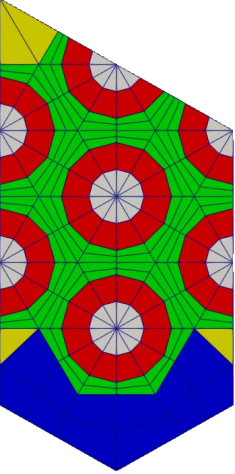
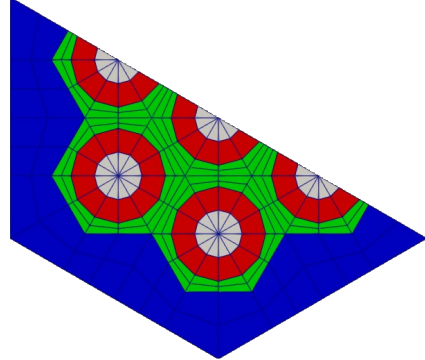
Figure 3-28 Example outputs of HexagonMeshTrimmer

3.7.4 Handling Degenerate Quadrilateral Elements

When trimming a mesh, some elements may be located across the trimming line and thus need to be processed to ensure a smooth trimming boundary. `PlaneDeletionGenerator` is capable of trimming meshes but leaves a zigzag trimming boundary in the presence of elements which lay across the trimming line.

To avoid this zig-zag boundary, `HexagonMeshTrimmer` adopts a post-trimming processing algorithm to smooth the trimming boundary. The algorithm moves the nodes of the across-trimming-line elements in the normal direction of the trimming line onto the trimming line. During this procedure, some elements may become zero volume and will be removed. More importantly, after node moving, some quadrilateral elements may have three co-linear vertices on the trimming line, which make the element degenerate. To fix this issue, these degenerate quadrilateral elements are converted into triangular elements. As triangular elements and quadrilateral elements cannot share a single subdomain id/name, new subdomains are created for any affected quadrilateral element subdomains. The subdomain ids of the new subdomains are decided by shifting the original subdomain ids by `"tri_elem_subdomain_shift"` (default shift value is the maximum subdomain id of the mesh), while the subdomain names of the new subdomains are created by appending `"tri_elem_subdomain_name_suffix"` after the original subdomain names.

Syntax and output example can be found in the following table.

| | |
|---|---|
| <pre>[Mesh] [trim_0] type = HexagonMeshTrimmer input = pattern trim_peripheral_region = '1 1 1 1 1 1' peripheral_trimming_section_boundary peripheral_section [] []</pre> | <p style="text-align: center;">=</p>  |
| <pre>[Mesh] [trim_0] type = HexagonMeshTrimmer input = pattern trim_peripheral_region = '1 0 1 0 0 1' peripheral_trimming_section_boundary peripheral_section [] []</pre> | <p style="text-align: center;">=</p>  |
| <pre>[Mesh] [trim] type = HexagonMeshTrimmer input = pattern center_trim_starting_index = 5 center_trim_ending_index = 11 center_trimming_section_boundary = center_section [] []</pre> |  |

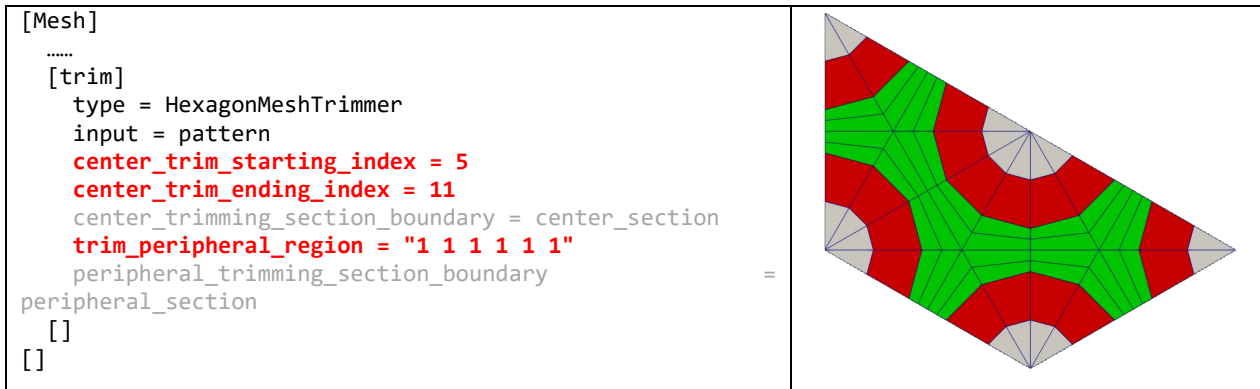


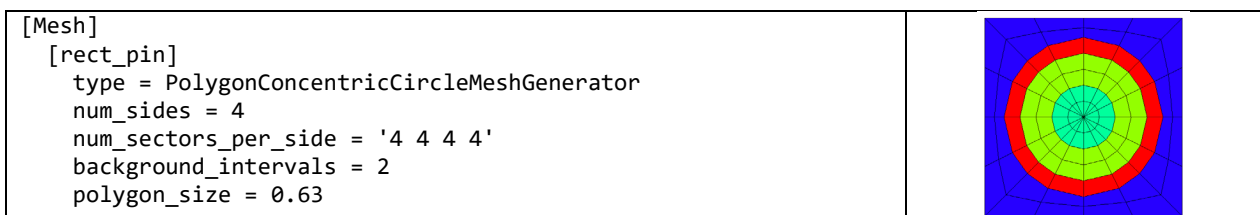
Figure 3-29 Syntax and output for HexagonMeshTrimmer.

3.8 Ring and Sector Reporting IDs

Reporting ID capabilities were implemented in the mesh generators to provide a way to identify the geometric components in the reactor mesh without using coordinate information. In order to expand the reporting ID capabilities to the sub-pin level, reporting IDs for rings and sectors within the Cartesian and hexagonal pins were introduced in the base pin mesh generator in the Reactor module. Users can utilize the ring and sector IDs to identify and manipulate sub-pin level information in the mesh generation and the post-processing stages of the simulation. The new functionality to assign ring and sector ID was added to *PolygonalConcentricMeshGenerator* and *TriPinHexAssemblyGenerator*. The detailed descriptions for these two pin mesh generators and the utilization of ring and sector IDs for the depletion ID setting are given in the following sub-sections.

3.8.1 PolygonConcentricCircleMeshGenerator

The ring and sector IDs are optionally assigned in the pin mesh generation by defining the extra integer names in *ring_id_name* and *sector_id_name* parameters, respectively. User can choose whether the ring IDs are assigned for each annular block defined in *ring_radii* or for each actual annular meshes by defining the parameter *ring_id_assign_type* to *block-wise* or *ring-wise*. Note that the ring IDs are not defined for the background region of pin cell. Unique sector IDs are assigned to each sector formed by the pin cell center point and boundary surfaces. Thus, the number of sector IDs are equal to the number of boundary sides. Once set up in the pin cell mesh generation stages, IDs are automatically propagated to subsequent mesh generation stage and made available in the final mesh output, as shown in Figure X. More detailed syntax and output examples can be found in the following table.



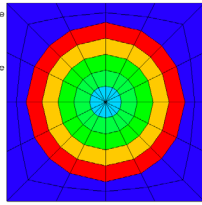
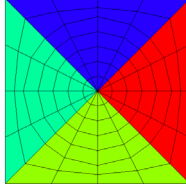
| | |
|---|---|
| <pre> polygon_size_style = 'apothem' ring_radii = '0.2 0.4 0.5' ring_intervals = '2 2 1' flat_side_up = true ring_id_name = 'ring_id' ring_id_assign_type = 'block_wise' # block_wise or ring_wise sector_id_name = 'sector_id' [] [] </pre> |  |
| <pre> [Mesh] [rect_pin] type = PolygonConcentricCircleMeshGenerator ... sector_id_name = 'sector_id' [] [] </pre> |  |

Figure 3-30 Example of setting ring and sector reporting IDs inside PCCMG.

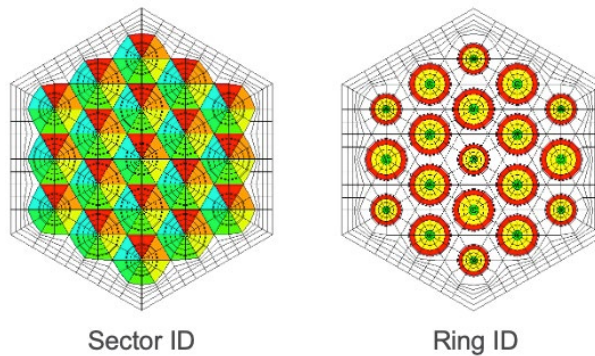


Figure 3-31 Rings and sectors colored by ID in hexagonal assembly

3.8.2 TriPinHexAssemblyGenerator

Ring and sector IDs described in the above sub-section were also implemented to TriPinHexAssemblyGenerator. If the settings for sector and ring IDs are defined in the input block, they are universally applied to the three pin cells in hexagonal assembly generated here. More detailed syntax and output examples can be found in the following figure.

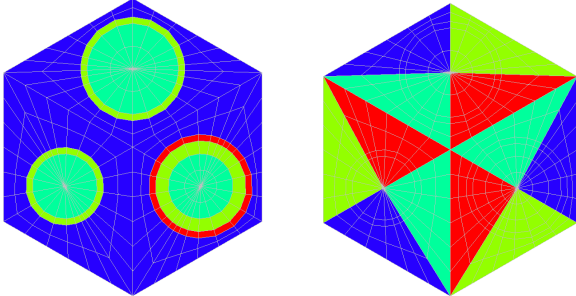
| | |
|--|--|
| <pre> [Mesh] [assm_up] type = TriPinHexAssemblyGenerator ... pin_id_values = '0 1 2' sector_id_name = 'sector_id' ring_id_name = 'ring_id' [] [] </pre> |  |
|--|--|

Figure 3-32 Rings and sectors colored by ID in hexagonal assembly

3.8.3 Utilization in Depletion ID Generation

Sub-pin level detailed depletion zones can be easily set up by utilizing the ring and sector IDs. DepletionIDGenerator assigns depletion IDs on individual elements by finding unique combinations of user-specified IDs. Users can easily control the level of detail for which depletion IDs are assigned through the selection of reporting IDs used in the depletion ID generation. Thus, by including the ring and sector IDs, the unique depletion IDs are set on each sector and ring of individual pins. Thus, the sub-pin level detailed zones can be readily generated as shown in the below table.

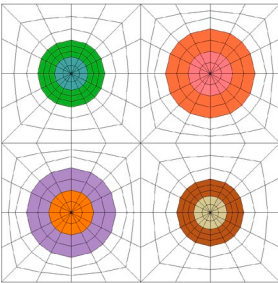
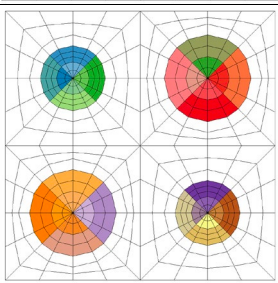
| | |
|---|--|
| <pre>[Mesh] [depletion_id] type = DepletionIDGenerator input = 'core' id_name = 'pin_id ring_id' material_id_name = 'material_id' exclude_id_name = 'material_id ring_id' exclude_id_value = '8 9; 0' [] []</pre> |  |
| <pre>[Mesh] [depletion_id] type = DepletionIDGenerator input = 'core' id_name = 'pin_id ring_id sector_id' material_id_name = 'material_id' exclude_id_name = 'material_id ring_id' exclude_id_value = '8 9; 0' [] []</pre> |  |

Figure 3-33 Setting sub-pin depletion IDs using DepletionIDGenerator

3.9 Vector Post Processor Based on Reporting IDs

MOOSE VectorPostprocessors can be used to query data on elements with extra IDs, simplifying output processing significantly since collections of elements forming pins and assemblies are now identifiable without providing information on their physical location. ExtraIDIntegralVectorPostProcessor integrates solution variables over zones identified by combinations of reporting IDs. For reactor applications, component-wise values such as pin-by-pin power distribution can be easily yielded by specifying integration over pin and assembly reporting IDs. The detailed syntax and output examples can be found in the following table.

| <pre>[VectorPostprocessors] [integral] type = ExtraIDIntegralVectorPostprocessor variable = 'power' id_name = 'assembly_id pin_id' [] []</pre> | <table border="1"> <thead> <tr> <th>assembly_id</th> <th>pin_id</th> <th>Power</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1.567145053</td> </tr> <tr> <td>0</td> <td>1</td> <td>1.506959147</td> </tr> <tr> <td>...</td> <td>...</td> <td>...</td> </tr> <tr> <td>0</td> <td>14</td> <td>1.078768104</td> </tr> <tr> <td>0</td> <td>15</td> <td>0.945523933</td> </tr> <tr> <td>1</td> <td>0</td> <td>0.999022329</td> </tr> <tr> <td>1</td> <td>1</td> <td>0.742321547</td> </tr> <tr> <td>...</td> <td>...</td> <td>...</td> </tr> <tr> <td>1</td> <td>14</td> <td>0.355067787</td> </tr> <tr> <td>1</td> <td>15</td> <td>0.119894792</td> </tr> <tr> <td>...</td> <td>...</td> <td>...</td> </tr> </tbody> </table> | assembly_id | pin_id | Power | 0 | 0 | 1.567145053 | 0 | 1 | 1.506959147 | ... | ... | ... | 0 | 14 | 1.078768104 | 0 | 15 | 0.945523933 | 1 | 0 | 0.999022329 | 1 | 1 | 0.742321547 | ... | ... | ... | 1 | 14 | 0.355067787 | 1 | 15 | 0.119894792 | ... | ... | ... |
|--|--|-------------|--------|-------|---|---|-------------|---|---|-------------|-----|-----|-----|---|----|-------------|---|----|-------------|---|---|-------------|---|---|-------------|-----|-----|-----|---|----|-------------|---|----|-------------|-----|-----|-----|
| assembly_id | pin_id | Power | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 1.567145053 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 1.506959147 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | ... | ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 14 | 1.078768104 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 15 | 0.945523933 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 0 | 0.999022329 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 1 | 0.742321547 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | ... | ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 14 | 0.355067787 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 | 15 | 0.119894792 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ... | ... | ... | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Figure 3-34 Integration of assembly and pin power using new VectorPostProcessor.

ExtraIDIntegralVectorPostProcessor exports the post-processed results in CSV file format. Its derivative ExtraIDIntegralVectorReporter, based on the MOOSE reporting system, can output in JSON file format, which can be easily parsed using script languages such as Python and Perl.

| | |
|---|---|
| <pre>[Reporters] [extra_id_integral] type = ExtraIDIntegralReporter variable = 'power' id_name = 'assembly_id pin_id' [] []</pre> | <pre>"extra_id_integral": { "extra_id_data": { "extra_id_data": { "id_name": ["assembly_id", "pin_id"], "map_id_to_value": [[...], [...]], "num_id_name": 2, "num_values_per_integral": 1 }, "integrals": { "num_variables": 1, "value1_integral": [1.5671450533836675, 1.5069591468755417, ...], "variable_names": ["power"] } } }</pre> |
|---|---|

Figure 3-35 Integration of assembly and pin power using new Reporter.

3.10 Updates to Reactor Geometry Mesh Builder (RGMB) Capabilities

The Reactor Geometry Mesh Builder (RGMB) is a set of mesh generators that was introduced to the Reactor module to simplify the process of generating conventional Cartesian and hexagonal reactor cores by calling underlying mesh generators for pin, assembly, and core definitions while simplifying the user options needed to define the mesh specifications and lattice structures (Shemon, et al., 2021). The ability to call mesh generators inside of mesh generators is known as “mesh sub-generators” and was developed by MOOSE framework team at INL (Lindsay, et al.,

2021) in direct support of this project. This capability is integral to RGMB's simplicity and is detailed in Section 3.10.1.

In addition to simplifying inputs for the user, RGMB mesh generators also provide two additional benefits to the reactor analyst. First, RGMB automates the assignment of pin-level, assembly-level, and plane-level (if mesh is extruded) reporting IDs by calling and setting the relevant parameters needed to define these extra element integers under the hood.

Second, the mapping between mesh elements and region IDs can be made directly from the RGMB mesh generators, which can pre-identify regions in the mesh that share similar material properties and simplify input in the downstream physics problem. These region IDs (referred to as material IDs in the FY21 MOOSE meshing report) are set as extra element integers, which can later be referenced directly by the physics problem to assign shared material properties to common regions in the mesh without having to rely on mesh-specific block IDs that may not be easily determined prior to mesh generation. While RGMB mesh generators were introduced in FY21, the review process and eventual merging of these mesh generators into the MOOSE reactor module codebase saw numerous changes to the RGMB functionality. Thus, the mesh generators comprising of the RGMB system are summarized here once again with latest specifications and capabilities. Additionally, Sections 4.1-4.5 illustrate how the RGMB mesh generators can be used for definition of two candidate reactor mesh geometries.

3.10.1 Mesh Sub-Generator Objects in MOOSE

Before discussing the Reactor Geometry Mesh Builder, we first review recent improvements in the framework which make the Reactor Geometry Mesh Builder workflow possible.

The MeshGenerator class hierarchy enables MOOSE users to create meshes for a number of basic domain shapes, controlling domain and mesh parameters via input file. These meshes can then be modified and combined via multiple generator objects specified in other input file subsections. This allows users to create arbitrarily large trees of MOOSE mesh generator dependencies at runtime so they can construct meshes for complex simulation domains.

The flexibility of this interface, however, has also been a source of difficulty: to construct a mesh in this fashion, parameters for every sub-mesh must be specified independently, correctly, and sometimes redundantly, in each user input file. Even the number of subgenerators can only be varied by adding or deleting input file sections for each.

For common complex combinations of mesh generators, MOOSE now alternatively encourages a complex MeshGenerator object to create its own "sub-generator" objects via C++ code, specifying sub-generator parameters at runtime, based on higher-level-generator parameters, without the need for additional user input. Application-specific MeshGenerator subclasses thereby take the micromanagement of mesh subcomponent construction out of user hands, and construct detailed meshes of known or parametrically defined domains based on only a simplified set of input parameters. The recent MeshGenerator::addMeshSubgenerator() APIs now in MOOSE are the preferred (documented, regression-tested, supported) mechanism for doing this in higher-level generator codes, controlled by either providing a MOOSE InputParameters object or by allowing MOOSE to build up parameters on the fly from a variadic argument list.

The RGMB mesh generator objects routinely use sub-generators objects under-the-hood in order to pass information from the original mesh generator object through a series of additional objects

“hidden” from the user. This allows the use to see a very simple and concise input structure while performing complex operations behind the scenes.

3.10.2 ReactorMeshParams

ReactorMeshParams is the mesh generator that defines all static parameters that are unchanged throughout the RGMB mesh generation process. While it cannot be used to generate a mesh on its own, it is responsible for storing any information about the mesh that needs to be accessed by multiple subsequent RGMB mesh generators. Table 3-2 summarizes the parameters that can be set through ReactorMeshParams:

Table 3-2. Global Parameters used in Reactor Geometry Mesh Builder

| Parameter Name | Description of Parameter |
|----------------------|---|
| dim | The dimension of the mesh to be generated (2-D or 3-D) |
| geom | The geometry type of the reactor mesh (Square or Hex) |
| assembly_pitch | Center-to-center distance of adjacent assemblies |
| bottom_boundary_id | Boundary ID given to bottom boundary of mesh (Required for extruded meshes) |
| top_boundary_id | Boundary ID given to top boundary of mesh (Required for extruded meshes) |
| radial_boundary_id | Boundary ID given to radial boundary of CoreMeshGenerator mesh |
| axial_regions | Length of each axial region |
| axial_mesh_intervals | Number of elements in the z-direction for each axial region. |

It should be noted that the option for procedural subdomain assignment is no longer available in RGMB and instead the users must provide an explicit mapping of region IDs to mesh elements throughout the RGMB mesh generation process. This process will be explained in the following subsections.

3.10.3 PinMeshGenerator

PinMeshGenerator is the mesh generator in charge of defining a single pin or all unique pins that belong to a larger assembly lattice. These pins can have three substructures – the innermost radial pin regions, the single bridging background region (required for all pins), and the duct regions, and this mesh generator calls PolygonConcentricCircleMeshGenerator to define the 2-D pin and FancyExtruderGenerator if extruding the pin to three dimensions. (AdvancedExtruderGenerator will be supported once the merge request discussed in Section 3.4.1 is finalized.) The following parameters are exposed to the user to define the pin geometry:

Table 3-3. Parameters used in Reactor Geometry Mesh Builder’s PinMeshGenerator

| Parameter Name | Description of Parameter |
|----------------------|--|
| reactor_params | The name of the ReactorMeshParams mesh generator used to define reactor mesh properties |
| pin_type | Integer ID given to pin |
| pitch | Center-to-center pitch of pin |
| num_sectors | Number of azimuthal sectors in each quadrant |
| ring_radii | Radii of major concentric circles of pin. If unspecified, no pin rings are created |
| duct_halfpitch | Apothems of duct regions. If unspecified, no duct regions are created |
| mesh_intervals | Number of mesh intervals for each pin region, starting at the center. The length of this vector should be (length of ring_radii + 1 + length of duct_halfpitch). The additional interval defines the number of mesh elements in the background region |
| block_names | 2-D vector of size $A * R$ specifying the block names for each radial and axial zone, where A is the number of axial regions (defined in ReactorMeshParams) and R is the number of radial regions (defined in mesh_intervals). If this parameter is unspecified, one block name is assigned to all quad elements and another block name is assigned to all tri elements in the mesh. |
| region_ids | 2-D vector of size $A * R$ specifying the region IDs for each radial and axial zone, where A is the number of axial regions (defined in ReactorMeshParams) and R is the number of radial regions (defined in mesh_intervals) |
| extrude | Boolean to specify whether pin mesh should be extruded to 3-D. Extrusion must occur as the last step of mesh generation in RGMB so if this parameter is true, this pin cannot be used in further mesh building in the Reactor workflow |
| quad_center_elements | Boolean to set whether center elements of this mesh are quad (true) or tri (false) |

The definition of the parameters `region_ids` and `block_names` sets the region IDs and block names of the pin mesh. `region_ids` provides a map of radial and axial zones that share material properties. Downstream physics calculations can then use the “region_id” extra element integer map created by RGMB to set material properties, instead of relying on a mapping between the block ID of the input mesh and the material ID of the physics problem, thus saving an additional step for the user. In this workflow, users no longer have to monitor information related to the block ID and block name of the mesh. RGMB defaults to generating a single block ID for all quad elements in the pin mesh and another block ID for the tri elements in the pin mesh. The quad elements will also have block name `RGMB_PIN<pin_type_id>`, where `<pin_type_id>` is the pin ID provided by the `pin_type` parameter, while all tri elements will have the block name `RGMB_PIN<pin_type_id>_TRI`. However, for users who require explicit definition of block names, the `block_names` parameter can be used to define the block name of each radial and axial region of the mesh.

After the pin mesh is created, `PinMeshGenerator` sets a number of extra element IDs based on the "region_id" map and "pin_type". Finally, if the pin mesh is extruded to three dimensions, each axial layer is tagged with the extra integer name "plane_id" to distinguish mesh elements that belong to the same axial plane.

With respect to the exterior boundary information, `PinMeshGenerator` automatically generates boundary sidesets and nodesets for the pin. The radial pin boundary is assigned the ID equal to $(20000 + \langle \text{pin_type_id} \rangle)$ and is named "outer_pin_<pin_type_id>", where <pin_type_id> is the pin ID provided by the `pin_type` parameter. For example, a pin with a pin type ID of 1 will have a boundary ID of 20001 and boundary name of "outer_pin_1". If the pin is extruded to three dimensions the top-most boundary ID must be assigned using `ReactorMeshParams/top_boundary_id` and will have the name "top", while the bottom-most boundary must be assigned using `ReactorMeshParams/bottom_boundary_id` and will have the name "bottom".

The following example shows how to define a circular pin within a Cartesian enclosure using `PinMeshGenerator`. The left image shows the resulting mesh block layout, where by default a single block is assigned to the triangular elements and another block is assigned to the quadrilateral elements. On the other hand, all region-wise heterogeneities are controlled by the "region_id" extra element integer layout, which is shown in the right image. These region IDs are set for each radial region in `Mesh/pin1/region_ids`.

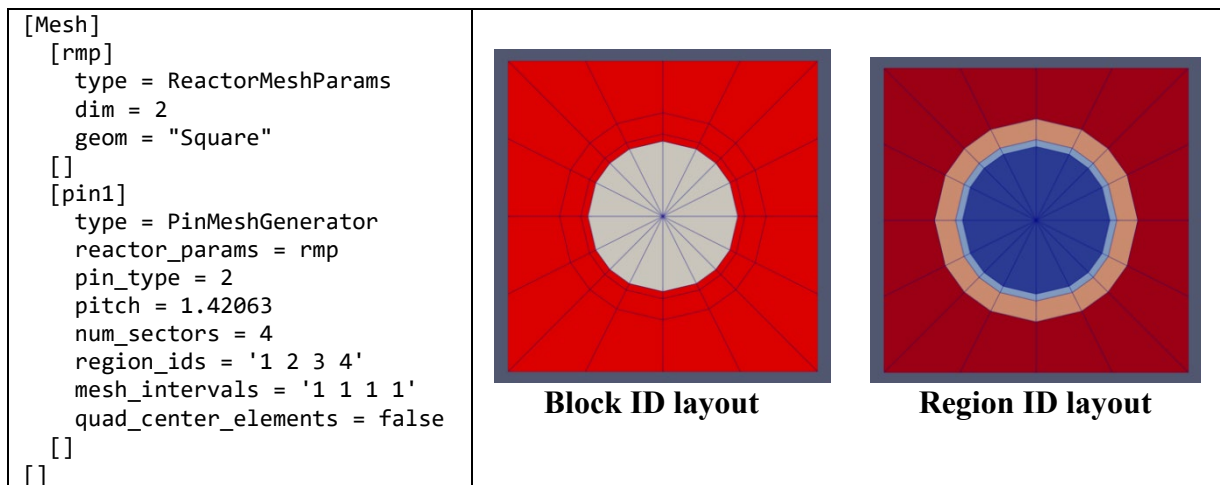


Figure 3-36 Creation of Cartesian pin using Reactor Geometry Mesh Builder's `PinMeshGenerator`.

3.10.4 `AssemblyMeshGenerator`

`AssemblyMeshGenerator` defines a single assembly-like structure or all unique assemblies that belong to a larger core lattice. The assembly-like structures must consist of a full pattern of equal sized pins from `PinMeshGenerator`. Pins inside the hexagonal assembly will be placed inside of a bounding hexagon consisting of a background region and, optionally, one or more duct regions.

This object automates the use and functionality of the `CartesianIDPatternedMeshGenerator` for Cartesian reactor geometry, and `HexIDPatternedMeshGenerator` for hexagonal reactor geometry. If extruding to three dimensions, `AssemblyMeshGenerator` also calls `FancyExtruderGenerator`.

Like PinMeshGenerator, AssemblyMeshGenerator automates block ID and region ID assignment for the assembly background and duct regions (duct regions only available for hexagonal assemblies) and boundary ID and name assignment. The following parameters are exposed to the user to define the assembly geometry using AssemblyMeshGenerator:

Table 3-4. Parameters used in Reactor Geometry Mesh Builder’s PinMeshGenerator

| Parameter Name | Description of Parameter |
|-----------------------|---|
| inputs | The names of the PinMeshGenerator objects that will be used to form the assembly lattice. |
| assembly_type | Integer ID given to assembly |
| pattern | A double indexed array starting with the upper-left corner, where the index represents the layout of input pins in the assembly lattice |
| duct_halfpitch | Distance(s) from center of assembly to duct inner boundary(ies). |
| background_intervals | Radial intervals in the assembly background region. |
| duct_intervals | Number of meshing intervals in each enclosing duct. |
| background_region_id | The region ID for the assembly background region, used for setting “region_id” extra element integers. If the assembly is extruded to three-dimensions, then a region ID must be provided for each axial level, starting from the bottom layer to the top layer. |
| duct_region_ids | The region ID for the assembly duct regions, used for setting “region_id” extra element integers. If the assembly is extruded to three-dimensions, then a duct region ID must be provided for each axial level, starting from the bottom layer to the top layer. Dimension of 2-D vector is $A * D$, where A is the number of axial layers and D is the number of radial duct regions in each axial layer. |
| background_block_name | The block names for the assembly background region. If the assembly is extruded to three-dimensions, then a region ID must be provided for each axial level, starting from the bottom layer to the top layer. If this parameter is unspecified, one block name is assigned to all quad elements in the background region and another block name is assigned to all tri elements in the background region. |
| duct_block_names | 2-D vector of size $A * D$, where A is the number of axial layers and D is the number of radial duct regions in each axial layer, used for setting the block names of each duct region. If this parameter is unspecified, one block name is assigned to all quad elements in the background region and another block name is assigned to all tri elements in the background region. |
| extrude | Boolean to specify whether assembly mesh should be extruded to 3-D. Extrusion must occur as the last step of mesh generation in RGMB so if this parameter is true, this assembly cannot be used in further mesh building in the Reactor workflow |

Similar to the conventions followed in `PinMeshGenerator`, the parameters `background_region_id` and `duct_region_ids` must be set by the user to identify regions within the assembly around the lattice of fuel pins. This functionality is intended for identification of regions within the mesh that will have the same physics properties such as material assignments, and will be used to set the “`region_id`” extra element integer of the resultant mesh.

Likewise, block IDs are generated automatically by `AssemblyMeshGenerator`, and for users who require element identification by block name, the optional parameters `background_block_name` and `duct_block_names` can be defined to set block names for the assembly background and duct regions respectively. By default, each quad element within the assembly (pin elements inclusive) will have the block name “`RGMB_ASSEMBLY<assembly_type_id>`”, where `<assembly_type_id>` is the assembly ID provided by the user through the parameter `assembly_type`. All tri elements within the mesh will have the block name `RGMB_ASSEMBLY<assembly_type_id>`.

`AssemblyMeshGenerator` will tag all elements with the following extra element integer map names, followed by a description of these maps:

Table 3-5. Extra Element IDs Assigned in Reactor Geometry Mesh Builder’s `AssemblyMeshGenerator`

| Extra element ID name | Extra element integer map description |
|----------------------------|---|
| <code>region_id</code> | Groups all elements within the assembly with equivalent material properties |
| <code>assembly_type</code> | All elements within the assembly will have the same value <code><assembly_type_id></code> , based on the value of the <code>assembly_type</code> parameter. |
| <code>plane_id</code> | Integer map to distinguish each axial layer, if assembly mesh is extruded to three dimensions |
| <code>pin_id</code> | ID given to each pin in the assembly lattice, based on the conventions used by <code>CartesianIDPatternedMeshGenerator</code> or <code>HexIDPatternedMeshGenerator</code> . Pin IDs are set using the “ <code>cell</code> ” assignment type |

Finally, `AssemblyMeshGenerator` automatically generates boundary sidesets and nodesets for the assembly, where the radial assembly boundary is assigned the ID equal to $(2000 + \text{<assembly_type_id>})$ and is named “`outer_assembly_<assembly_type_id>`”, where `<assembly_type_id>` is the assembly ID provided by the `assembly_type` parameter. For example, an assembly with an assembly type ID of 1 will have a boundary ID of 2001 and boundary name of “`outer_assembly_1`”. If the assembly is extruded to three dimensions the top-most boundary ID must be assigned using `ReactorMeshParams/top_boundary_id` and will have the name “`top`”, while the bottom-most boundary must be assigned using `ReactorMeshParams/bottom_boundary_id` and will have the name “`bottom`”.

The following example shows how to define a Cartesian assembly lattice using `AssemblyMeshGenerator`. The top image shows the resulting mesh block layout, where by default a single block is assigned to the triangular elements and another block is assigned to the quadrilateral elements. Region-wise heterogeneities are controlled by the “`region_id`” extra element integer layout, which is shown in the bottom image.

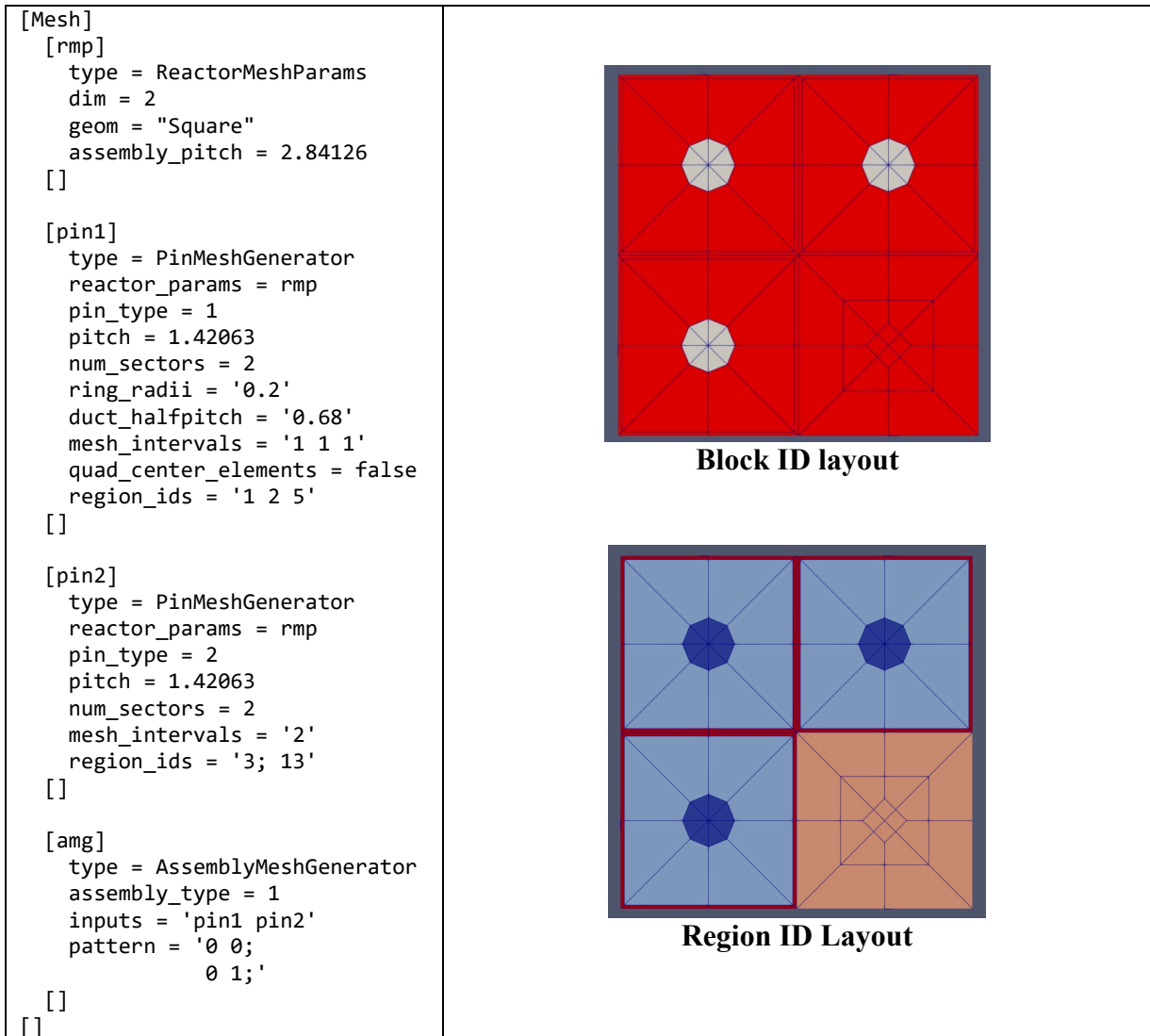


Figure 3-37 Creation of Cartesian assembly using Reactor Geometry Mesh Builder’s AssemblyMeshGenerator depicting block IDs and Region IDs

3.10.5 CoreMeshGenerator

CoreMeshGenerator is the mesh generator in charge of defining a core-like structure consisting of assemblies with equivalent pitch sizes in a lattice configuration. This core lattice layout is permitted to have empty or “dummy” locations.

This object automates the use and functionality of the CartesianIDPatternedMeshGenerator for Cartesian reactor geometry, and HexIDPatternedMeshGenerator for hexagonal reactor geometry. If extruding to three dimensions, CoreMeshGenerator also calls FancyExtruderGenerator. The following parameters are exposed to the user to define the core geometry using CoreMeshGenerator:

Table 3-6. Parameters in Reactor Geometry Mesh Builder’s CoreMeshGenerator

| Parameter Name | Description of Parameter |
|---------------------|--|
| inputs | The names of the AssemblyMeshGenerator objects that will be used to form the core lattice. |
| pattern | A double indexed array starting with the upper-left corner, where the index represents the layout of input assemblies in the core lattice |
| dummy_assembly_name | Name given to the dummy assembly which can be used in the inputs and pattern parameters to place dummy assemblies in the core lattice. |
| extrude | Boolean to specify whether core mesh should be extruded to 3-D. Extrusion must occur as the last step of mesh generation in RGMB so if this parameter is true, this assembly cannot be used in further mesh building in the Reactor workflow |

CoreMeshGenerator will tag all elements with the following extra element integer map names, followed by a description of these maps:

Table 3-7. Extra Element IDs Assigned in Reactor Geometry Mesh Builder's CoreMeshGenerator

| Extra element ID name | Extra element integer map description |
|-----------------------|--|
| plane_id | Integer map to distinguish each axial layer, if core mesh is extruded to three dimensions |
| assembly_id | ID given to each assembly in the assembly lattice, based on the conventions used by CartesianIDPatternedMeshGenerator or HexIDPatternedMeshGenerator. Assembly IDs are set using the "cell" assignment type and any "dummy" assembly locations (identified via the dummy_assembly_name parameter) will be excluded from the assembly ID numbering. |

CoreMeshGenerator automatically generates boundary sidesets and nodesets for the assembly, where the radial core boundary is assigned the ID specified in ReactorMeshParams/radial_boundary_id and is named "outer_core". If the assembly is extruded to three dimensions the top-most boundary ID must be assigned using ReactorMeshParams/top_boundary_id and will have the name "top", while the bottom-most boundary must be assigned using ReactorMeshParams/bottom_boundary_id and will have the name "bottom".

The following example shows how to define a Cartesian core lattice using CoreMeshGenerator. The top image shows the resulting mesh block layout, where a single block is assigned to elements since they are all quadrilateral type elements. All region-wise heterogeneities are controlled by the "region_id" extra element integer layout, which is shown in the bottom image.

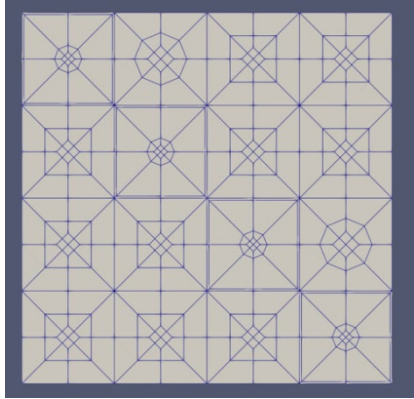
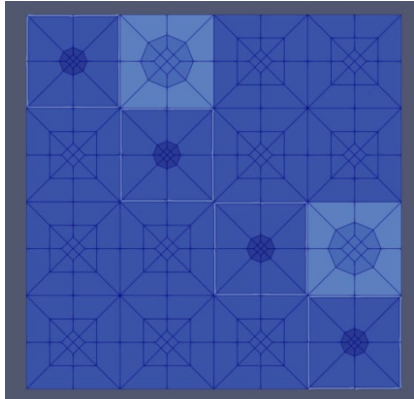
| | |
|--|---|
| <pre>[Mesh] [rmp] type = ReactorMeshParams dim = 2 geom = "Square" assembly_pitch = 2.84126 top_boundary_id = 201 bottom_boundary_id = 202 radial_boundary_id = 200 [] # Pin Definitions [pin1] type = PinMeshGenerator reactor_params = rmp pin_type = 1 pitch = 1.42063 num_sectors = 2 ring_radii = '0.2' duct_halfpitch = '0.68' mesh_intervals = '1 1 1' region_ids = '1 2 5' quad_center_elements = true [] [pin2] type = PinMeshGenerator reactor_params = rmp pin_type = 2 pitch = 1.42063 num_sectors = 2 mesh_intervals = '2' region_ids = '2' quad_center_elements = true [] [pin3] type = PinMeshGenerator reactor_params = rmp pin_type = 3 pitch = 1.42063 num_sectors = 2 ring_radii = '0.3818' mesh_intervals = '1 1' region_ids = '3 4' quad_center_elements = true [] # Assembly Definitions [amg1] type = AssemblyMeshGenerator assembly_type = 1 inputs = 'pin2' pattern = '0 0; 0 0' [] [amg2] type = AssemblyMeshGenerator assembly_type = 2 inputs = 'pin3 pin1 pin2' pattern = '0 1; 1 2' [] [cmg] type = CoreMeshGenerator inputs = 'amg2 amg1' pattern = '1 0; 0 1' [] []</pre> | <div style="text-align: center;">  <p>Block ID layout</p> </div> <div style="text-align: center;">  <p>Region ID Layout</p> </div> |
|--|---|

Figure 3-38 Creation of Cartesian core using Reactor Geometry Mesh Builder's CoreMeshGenerator

3.10.6 Periphery Mesh Generation

CoreMeshGenerator also now integrates in periphery mesh generation capabilities using either the PeripheralTriangleMeshGenerator (PTMG) or PeripheralRingMeshGenerator (PRMG). Both periphery mesh generators create a circular reactor boundary surrounding the core region and meshes the area between the reactor boundary and the core region that was created as part of the primary CMG functionality. PTMG meshes the periphery region with triangles (TRI3 elements) while PRMG meshes the periphery region with quads (QUAD4 elements).

CoreMeshGenerator adds in the periphery region after the core region has been created, and before extrusion into 3D if that step is being performed as part of the mesh builder workflow. The following parameters are exposed to the user to define the periphery region using CoreMeshGenerator.

Table 3-8. Common parameters in CoreMeshGenerator for core periphery meshing

| Parameter Name | Description of Parameter |
|----------------------|--|
| mesh_periphery | Boolean to determine if the periphery meshing should be performed |
| periphery_generator | Selector for which periphery mesh generator to use (PTMG or PRMG) |
| outer_boundary_id | The boundary id to set on the generated outer boundary |
| periphery_region_id | The extra element region_id to be assigned to the periphery region |
| periphery_block_name | The block name to be assigned to the periphery region |
| outer_circle_radius | The radius of the periphery to be meshed |

Since PTMG and PRMG generate the periphery region in different ways, additional parameters are specific to the periphery generator being used:

Table 3-9. Extra parameters in CoreMeshGenerator for core periphery meshing depending on type of mesh desired

| Parameter Name | Description of Parameter |
|-------------------------------------|--|
| outer_circle_num_segments (PTMG) | Number of segments to subdivide the outer circle boundary |
| extra_circle_radii (PTMG) | Vector of radii for extra Steiner point circles |
| extra_circle_num_segments (PTMG) | Vector of number of segments to subdivide the extra_circle_radii for extra Steiner point circles |
| periphery_num_layers (PRMG) | Number of layers to subdivide the peripheral region |

The following example shows how to create a triangulated periphery region using PTMG around the core region using CoreMeshGenerator. The mesh quality in the triangulated periphery is poor due to lack of Steiner points and could be improved by adding Steiner points or switching the quadrilateral option.

```
[Mesh]
[rmp]
  type = ReactorMeshParams
  dim = 3
  geom = "Hex"
  assembly_pitch = 7.10315
  axial_regions='10.0 10.0'
  axial_mesh_intervals='1 1'
  top_boundary_id = 201
  bottom_boundary_id = 202
  radial_boundary_id = 200
[]

### pin definitions omitted for brevity

# Assembly definitions
[amg1]
  type = AssemblyMeshGenerator
  assembly_type = 1
  background_intervals = 1
  inputs = 'pin2'
  pattern = '0 0;
            0 0 0;
            0 0'
  background_region_id='41 141'
  background_block_name='A1_R41 A1_R141'

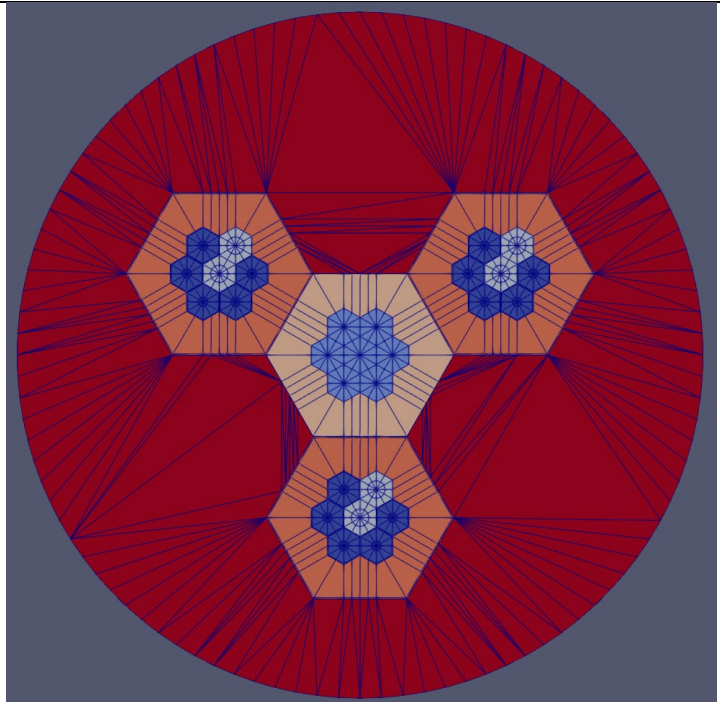
[]
[amg2]
  type = AssemblyMeshGenerator
  assembly_type = 2
  background_intervals = 1
  inputs = 'pin1 pin3'
  pattern = '0 0;
            1 0'
  background_region_id='51 151'
  background_block_name='A2_R51 A2_R151'
  duct_region_ids='52; 152'
  duct_block_names='A2_R52; A2_R152'
  duct_halfpitch='3.5'
  duct_intervals='1'
[]

# Core definition
[cmg]
  type = CoreMeshGenerator
  inputs = 'amg1 amg2 empty'
  dummy_assembly_name = empty
  pattern = '2 1;
            1 0 2;
            2 1'

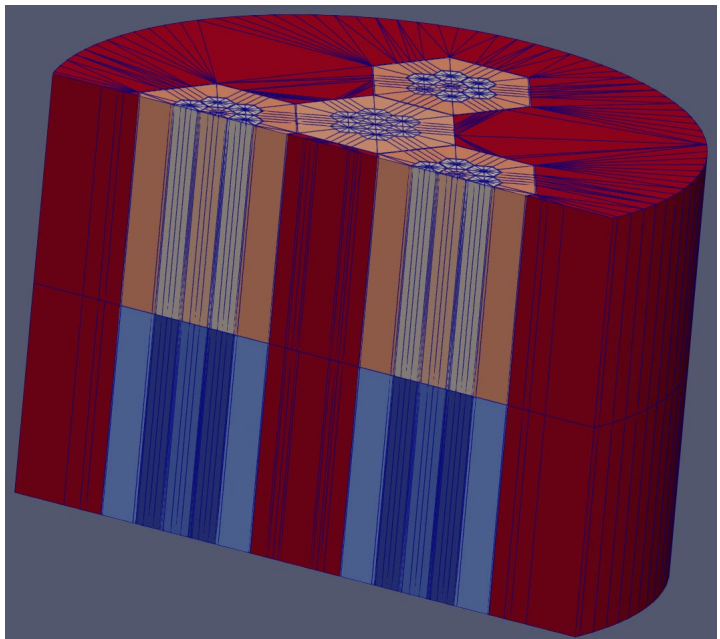
# periphery mesh occurs before extrusion
extrude = true

# periphery meshing
mesh_periphery=true
periphery_region_id='200'
periphery_block_name = 'PERIPHERY'

# PTMG periphery
periphery_generator=PTMG
outer_circle_radius=15
outer_circle_num_segments=100
[]
[]
```



Region ID Layout (2D)



Region ID Layout (After extrusion to 3D)

Figure 3-39 Creation of 3D core with meshed peripheral region using Reactor Geometry Mesh Builder

3.11 User Support

The team assisted users at various laboratories with mesh generation. Two more unusual examples are depicted below. The first example, shown in Figure 3-40, drove the development of new capability (TriPinHexAssemblyGenerator described in Section 3.2.3) to create the individual 3-pin assemblies needed in a Griffin HTTR model developed by V. Laboure (INL).

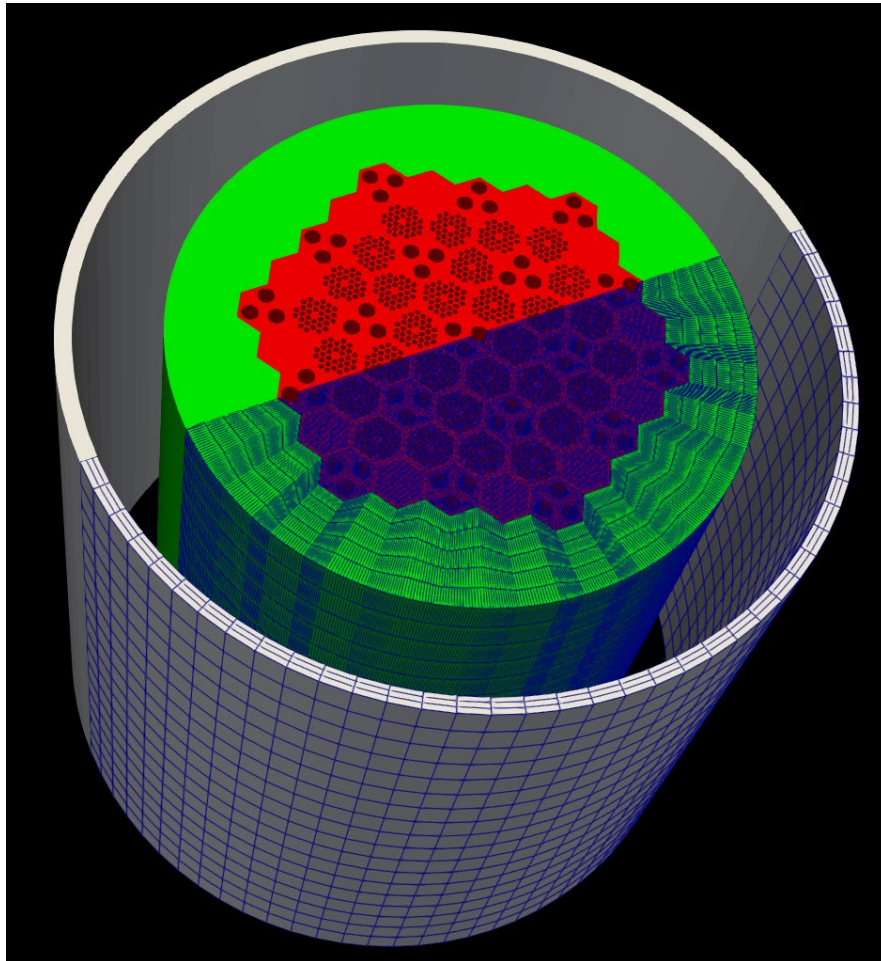


Figure 3-40 Preliminary HTTR Mesh in collaboration with V. Laboure (INL)

The second example Figure 3-41 leveraged combinations of Reactor Module tools produced by this team as well as the new Delaunay triangulation mesh generator (official name is pending merge request) developed by the INL MOOSE Framework team. This second example was constructed as a request from Griffin laboratory users to mesh fuel pins placed in a ring among a homogenized, triangulated background region. The corner of the assemblies consist of 1/3 pin sectors as shown in the left reference figure. Some of the basic geometry features were constructed to demonstrate

feasibility using the Reactor Module and the Delaunay triangulation mesh generator as shown in the right figure. This input was provided to the user.

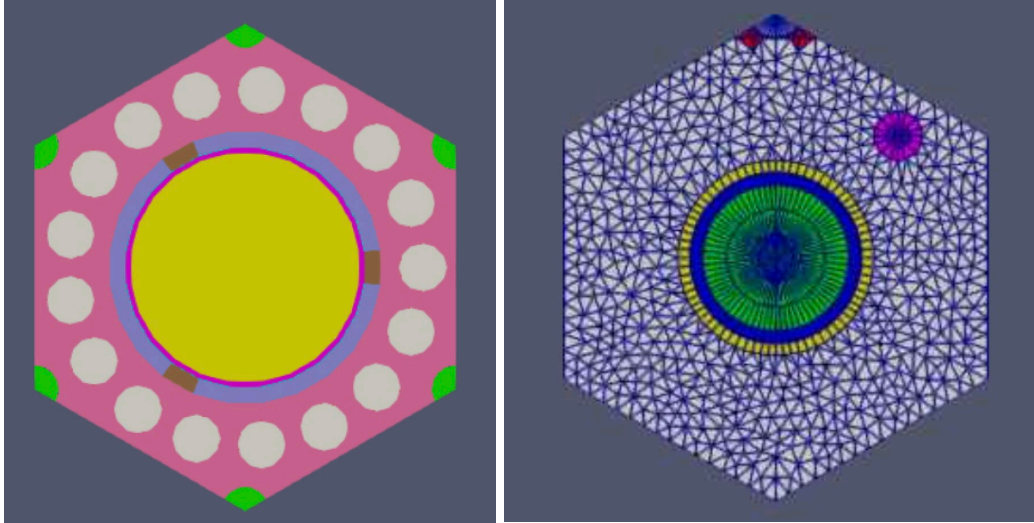


Figure 3-41 (left) Target reactor assembly design showing center hole, ring of fuel, and circular sector at hexagon vertices, provided by M. Lindell (INL). (right) Initial mesh of fundamental features using the upcoming Delaunay triangulation mesh generator and the Reactor Module.

4 Reactor Geometry Verification Examples and Meshes

To verify the mesh generators developed in this work were functioning properly, several physics benchmark cases were set up using the new MOOSE mesh tools. Some results were compared to pre-existing results that used externally generated meshes from either CUBIT or Argonne's Mesh Tools system.

4.1 Heterogeneous Lead-Cooled Fast Reactor Assembly

Griffin was employed to test the RGMB functionality described in Section 3.10. As described in Section 3.1.2, Griffin is compiled by linking directly to the MOOSE Reactor module, so all Reactor module mesh generators are accessible through the Griffin executable. To demonstrate the benefits of using RGMB mesh generators, a 3-D lead-cooled fast reactor (LFR) assembly example is studied. The geometry and design specifications of the assembly are based on inner zone fuel assembly of a 950 MWth liquid lead-cooled, fast neutron spectrum core designed by Westinghouse Electric Company (WEC) (Grasso, Levinsky, Franceschini, & Ferroni, 2019). The assembly consists of 7 rings of hexagonal pins discretized into 10 axial zones based on material heterogeneity. MOX fuel composed of depleted uranium and enriched plutonium is used in the inner fuel zones of the fuel pins in the assembly.

4.1.1 Mesh Generation

Figure 4-1 illustrates the top-down and side view of the assembly geometry generated using RGMB objects. Each hexagonal pin region is discretized radially into 4 rings representing (1) central helium hole surrounded by annular rings of (2) fuel, (3) helium gap, and (4) cladding, plus a background region of coolant. Two duct regions surround the entire assembly, representing the solid duct and the inter-assembly coolant gap. The 2D mesh is axially extruded into 3D with 10 regions of varying material composition. The general input structure for defining this mesh is provided below:

```
[Mesh]
[rmp]
  type = ReactorMeshParams
  dim = 3
  geom = "Hex"
  assembly_pitch = 16.4165
  axial_regions = '10.07 30.79 6.56 85.85 1.52 106.07 1.51 12.13 5.05 93.87'
  axial_mesh_intervals = '1 3 1 9 1 20 1 2 1 9'
  top_boundary_id = 201
  bottom_boundary_id = 202
  radial_boundary_id = 200
[]
[pin1]
  type = PinMeshGenerator
  reactor_params = rmp
  pin_type = 1
  pitch = 1.3425
  num_sectors = 2
  mesh_intervals = '1 3 1 1 1'
  ring_radii = '0.2020 0.4319 0.4495 0.5404'
  region_ids='1 1 1 1 1;
              2 2 2 2 2;
              3 3 3 3 3;
              4 4 4 5 6;
              8 8 8 9 10;
              20 19 20 13 21;
              24 24 24 25 26;
              28 28 28 29 30;
              32 32 32 32 32;
```

```

33 33 33 33 33'
quad_center_elements = false
[]
# Define all other pins with unique region IDs radially / axially
# This step is omitted for brevity
[assembly]
type = AssemblyMeshGenerator
inputs = 'pin1 pin2 pin3 pin4 pin5 pin6 pin7'
pattern = ' 0 0 0 0 0 0 0;
           0 1 1 1 1 1 1 0;
           0 1 2 2 2 2 2 1 0;
           0 1 2 3 3 3 3 2 1 0;
           0 1 2 3 4 4 4 3 2 1 0;
           0 1 2 3 4 5 5 4 3 2 1 0;
           0 1 2 3 4 5 6 5 4 3 2 1 0;
           0 1 2 3 4 5 5 4 3 2 1 0;
           0 1 2 3 4 4 4 3 2 1 0;
           0 1 2 3 3 3 3 2 1 0;
           0 1 2 2 2 2 2 1 0;
           0 1 1 1 1 1 1 0;
           0 0 0 0 0 0 0'
extrude = true
assembly_type = 1
background_region_id = '1 2 3 6 10 21 26 30 32 33'
background_intervals = '1'
duct_halfpitch = '7.6712 8.0245'
duct_intervals = '1 1'
duct_region_ids = '1 1; 2 2; 3 3; 7 6; 11 10;
                  22 23; 27 26; 31 30; 32 32; 33 33'
[]
[]

```

Figure 4-1 Reactor Geometry Mesh Builder input for LFR assembly

In the first step, the ring radii, pin pitch, and radial and azimuthal discretizations of the pin are specified. In addition, the region IDs corresponding to material zones in the pin are specified through the `region_ids` parameter. This generates a map of all radial and axial region IDs within the pin as an extra element integer map, which the Griffin code uses downstream to set material properties directly instead of relying on a separate mapping input for blocks to materials. In the second step, each of the pins defined in the first step are placed in a hexagonal lattice, and the `extrude=true` option is used to inform the mesh generator to extrude the geometry into 3-D. At this stage, the dimensions and region IDs of the assembly duct and background regions are also provided.

Upon completion, `AssemblyMeshGenerator` generates the 3-D assembly mesh and automatically produce the extra element integers related to pin IDs (unique id given to each pin region in the assembly), plane IDs (unique id given to each axial region in the assembly), and region IDs (unique id given to each material region, specified through the user input). These reporting IDs can be queried by MOOSE's `ExtraIDIntegralVectorPostprocessor` to compute integral quantities of interest such as scalar flux and fission source based on pin location, axial location, and/or material region.

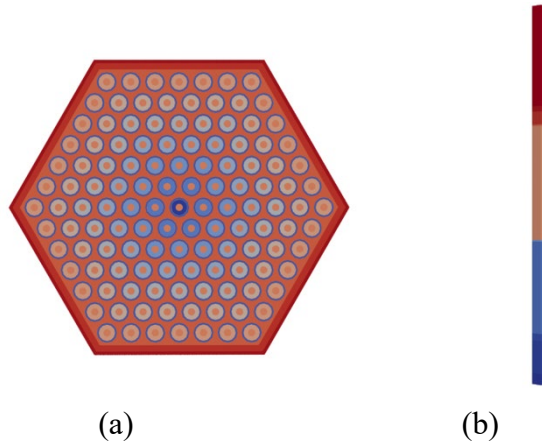


Figure 4-2. Top-down (a) and side (b) views of LFR heterogeneous assembly, created using RGMB objects in the Reactor Module. Each color represents a unique material region in the assembly.

4.1.2 Computation and Results

The neutronic behavior for the LFR assembly was simulated with Griffin, and the “region_id” extra element integer map generated by the RGMB mesh generators is used directly to specify material regions in Griffin. The multigroup cross sections for each material region were generated prior to the Griffin simulation with MC²-3 (Lee, Jung, & Yang, 2018). For this verification test, three avenues for mesh generation are explored: (1) using the RGMB mesh generators described above, (2) using the base Reactor Module mesh generators that are called by the RGMB mesh generators, and (3) using Argonne Mesh Tools (external software used for generating reactor-based meshes) to generate the mesh. All neutronics parameters are kept the same across Griffin runs: 9 energy groups, P1A3 Gauss-Chebyshev cubature, and a solver scheme that utilizes the discontinuous finite element method (DFEM) with discrete ordinates (S_N). Vacuum boundary conditions are applied to the top and bottom of the assembly; reflective boundary conditions are applied to the radial boundary. Coarse mesh finite differences (CMFD) is used to accelerate the transport problem convergence, and the input coarse mesh for this acceleration scheme – also defined with MOOSE mesh tools - homogenizes each pin region into a single hexagonal region composed of 12 triangles.

K-effective is used as a metric for comparing neutronics results between the three methods for mesh generation described above, and these results are summarized in Table 4-1. The MOOSE-based mesh generators produce identical eigenvalues, while there is a 20pcm difference in the Griffin simulation that uses Argonne’s Mesh Tools for mesh generation, which is due to the minor variation in the discretization schemes employed by MOOSE and Argonne’s Mesh Tools in the assembly background region. This is described in more detail in the FY21 report (Shemon, et al., 2021). While the two MOOSE-based mesh generation approaches produce identical results, the RGMB-based procedure offers several advantages.

First, the parameters within `PinMeshGenerator` and `AssemblyMeshGenerator` have been optimized to simplify input generation to focus solely on parameters of interest to the reactor analyst while masking any extraneous parameter definitions in base Reactor module mesh generators that would otherwise need to be specified. Secondly, material ID assignments can be made directly onto the resultant RGMB mesh, which Griffin can natively read to greatly simplify

the procedure of mapping cross section materials to elements within the input mesh. Base mesh generators defined in the Reactor module do not have this capability, and instead material assignments must be made based on block ID assignments of the mesh, which can be a very cumbersome process for problems with a large degree of radial and axial material heterogeneity. Finally, RGMB mesh generators are seamlessly integrated with reporting ID generation (pin, assembly, plane, and region IDs), and these reporting IDs streamline the definition of tally regions commonly used by reactor analysts.

Table 4-1. Griffin-computed k-effective results for LFR Assembly Problem: MOOSE mesh vs. Argonne Mesh Tools.

| Mesh Generation Tool | Griffin Solver Scheme | K-Effective |
|---|--------------------------------|-------------|
| MOOSE RGMB Mesh Generators | DFEM-SN with CMFD Acceleration | 1.17142 |
| MOOSE Base Reactor Module Mesh Generators | DFEM-SN with CMFD Acceleration | 1.17142 |
| Argonne Mesh Tools | DFEM-SN with CMFD Acceleration | 1.17122 |

To illustrate how automatically-generated reporting IDs from RGMB mesh generators can be used for reactor analysis workflows, the `ExtraIDIntegralVectorPostprocessor` is used in conjunction with the RGMB reporting IDs to inspect pin-by-pin tallies of interest. For this specific problem, scalar flux was specified as the tally variable, and both pin ID and axial plane ID were set as the reporting IDs that tell Griffin where to tally. Based on these specifications, an output CSV file is produced by Griffin, which tabulates the scalar flux for each unique combination of pin ID and axial plane ID. The data from this file can be used with a custom processing script to aggregate the scalar flux over like reporting IDs to generate pin-wide and axial plane-wide flux distributions over the entire core. For reference, the axially integrated pin scalar flux map produced from the processing script is illustrated in Figure 4-3.

This LFR verification example illustrates how the Reactor Module can streamline the Griffin workflow to execute a 3-D reactor analysis from input mesh construction, simulation execution, and output data generation for post-processing neutronics assembly or core distributions.

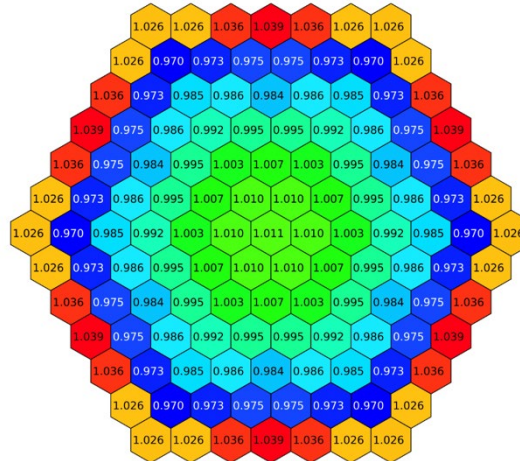


Figure 4-3. Axially integrated pin-by-pin group 0 normalized scalar flux distribution for the LFR assembly example.

4.2 Heterogeneous Cartesian Light Water Reactor Core

As an example of how a Cartesian-based core example can be meshed with the RGMB tools, the 2-D C5 problem was also investigated. In a similar fashion to Section 4.1, RGMB mesh generators are used to define the pin, assembly, and core configurations, and Griffin is used to simulate the neutronics problem. In this example, the DFEM-SN solver with CMFD acceleration is employed with 11 energy groups. The input file to define this mesh is provided in Figure 4-4, and the region ID map of the resultant mesh is shown in Figure 4-5.

```
[Mesh]
# ReactorMeshParams and PinMeshGenerator
# definitions
[rmp]
  type = ReactorMeshParams
  dim = 2
  geom = "Square"
  assembly_pitch = 21.42
  radial_boundary_id = 200
[]
[pin1_mox4.3]
  type = PinMeshGenerator
  pin_type = 1
  region_ids='2 2 6'
  pitch = 1.26
  reactor_params = rmp
  num_sectors = 2
  ring_radii = '0.18 0.54'
  mesh_intervals = '1 2 2'
  quad_center_elements = false
[]
[pin2_mox7.0]
  type = PinMeshGenerator
  pin_type = 2
  region_ids='3 3 6'
  reactor_params = rmp
  pitch = 1.26
  num_sectors = 2
  ring_radii = '0.18 0.54'
  mesh_intervals = '1 2 2'
  quad_center_elements = false
[]
[pin3_mox8.7]
  type = PinMeshGenerator
  pin_type = 3
  region_ids='4 4 6'
  # Full specifications omitted for brevity
[]
[pin4_fission_chamber]
  type = PinMeshGenerator
  pin_type = 4
  region_ids='6 6 6'
  # Full specifications omitted for brevity
[]
[pin5_guide_tube]
  type = PinMeshGenerator
  pin_type = 5
  region_ids='6 6 6'
  # Full specifications omitted for brevity
[]
[pin6_uo2]
  type = PinMeshGenerator
  pin_type = 6
  region_ids='1 1 6'
  # Full specifications omitted for brevity
[]
[pin7_mod]
  type = PinMeshGenerator
  reactor_params = rmp
  pin_type = 7
  pitch = 1.26
  num_sectors = 2
  duct_halfpitch = '0.1575'
  mesh_intervals = '1 3'
  region_ids='6 6'
  quad_center_elements = false
[]

# AssemblyMeshGenerator and CoreMeshGenerator
definitions
[mox_assy]
  type = AssemblyMeshGenerator
  assembly_type = 1
  inputs = 'pin1_mox4.3 pin2_mox7.0 pin3_mox8.7
           pin4_fission_chamber pin5_guide_tube'
  pattern = '0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0;
            0 1 1 1 1 4 1 1 4 1 1 4 1 1 1 1 1 0;
            0 1 1 4 1 2 2 2 2 2 2 2 2 1 4 1 1 0;
            0 1 1 1 2 2 2 2 2 2 2 2 2 1 1 1 0;
            0 1 4 2 2 4 2 2 4 2 2 4 2 2 4 1 0;
            0 1 1 2 2 2 2 2 2 2 2 2 2 2 1 1 0;
            0 1 1 2 2 2 2 2 2 2 2 2 2 2 1 1 0;
            0 1 4 2 2 4 2 2 3 2 2 4 2 2 4 1 0;
            0 1 1 2 2 2 2 2 2 2 2 2 2 2 1 1 0;
            0 1 1 2 2 2 2 2 2 2 2 2 2 2 1 1 0;
            0 1 4 2 2 4 2 2 4 2 2 4 2 2 4 1 0;
            0 1 1 1 2 2 2 2 2 2 2 2 2 2 1 1 0;
            0 1 1 4 1 2 2 2 2 2 2 2 2 1 4 1 1 0;
            0 1 1 1 1 4 1 1 4 1 1 4 1 1 1 1 0;
            0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0'
[]
[uo2_assy]
  type = AssemblyMeshGenerator
  assembly_type = 2
  inputs = 'pin6_uo2 pin4_fission_chamber
           pin5_guide_tube'
  pattern = '0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 2 0 0 2 0 0 2 0 0 0 0 0;
            0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 2 0 0 2 0 0 2 0 0 2 0 0 2 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 2 0 0 2 0 0 1 0 0 2 0 0 2 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 2 0 0 2 0 0 2 0 0 2 0 0 2 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 2 0 0 0 0 0 0 0 0 0 2 0 0 0;
            0 0 0 0 0 2 0 0 2 0 0 2 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0'
[]
[ref1_assy]
  type = AssemblyMeshGenerator
  assembly_type = 3
  inputs = 'pin7_mod'
  pattern = '0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0'
[]
[c5g7_2d]
  type = CoreMeshGenerator
  inputs = 'uo2_assy mox_assy ref1_assy'
  pattern = '0 1 2;
            1 0 2;
            2 2 2'
[]
```

Figure 4-4. RGMB input for C5G6 Cartesian core example.

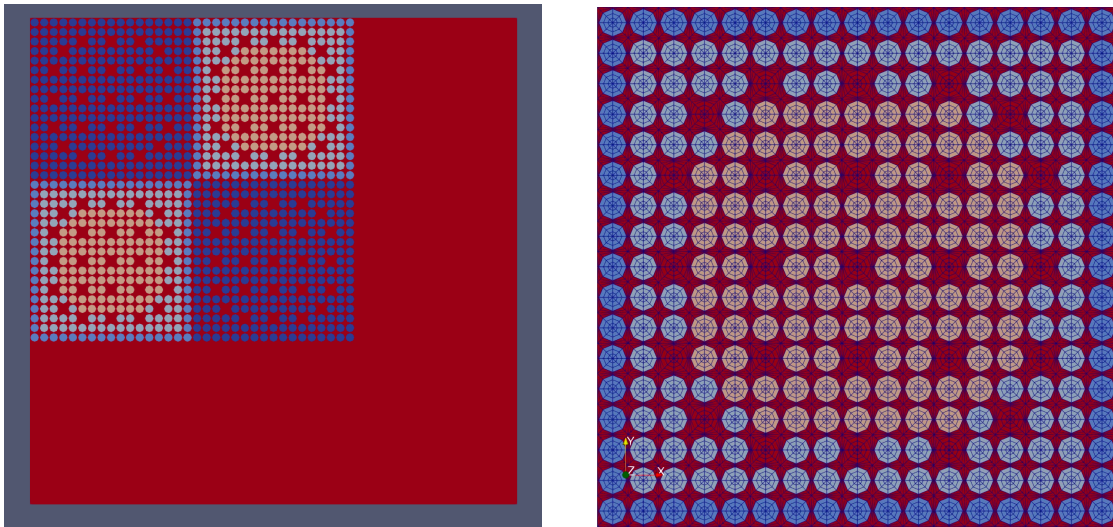


Figure 4-5 Region ID map of full-core C5G7 mesh (left) and zoomed in mesh discretization of MOX assembly (right).

Results from the Griffin problem are compared to the case where base mesh generators called by RGMB – namely PolygonConcentricCircleMeshGenerator and PatternedMeshGenerator – are defined explicitly to define the input mesh in place of the RGMB mesh generators. In both cases, the computed eigenvalue in Griffin is 1.07226, showing that RGMB mesh generators are capable of reproducing the same results as their constituent mesh generators while providing users the options to automatically define pin-wise and assembly-wise extra element IDs and directly map mesh elements to the material IDs in Griffin.

4.3 Heat Pipe-Cooled Microreactor (HPMR)

As a subset of Small Modular Reactors (SMRs) (Peakman, Hodgson, & Merk, 2018), nuclear microreactors have unique advantages over other typical types of large-scale nuclear reactors. The main advantages include factory fabrication, high transportability, and fast on-site installation (US DOE Office of Nuclear Energy, 2022), stemmed from their small size and simple design compared to commercial light water reactors (LWRs). A typical microreactor generates thermal power in the range of 1-20 MWth that can be directly used as heat, or further converted to electric power (US DOE Office of Nuclear Energy, 2022) or other secondary energy forms such as hydrogen (Dasari, Trelue, & Arafat, 2020). One of the advanced cooling methods that have been applied to microreactor designs is heat pipes featured of highly efficient heat transfer. To demonstrate and verify the capabilities of MOOSE meshgenerator system, two heat-pipe cooled microreactor (HPMR) examples are shown in the below sections.

4.3.1 2D Empire Model

Empire (Matthews, et al., 2021) is a 2MW (thermal) heat-pipe microreactor with up to 18 unit assemblies, originally developed at Los Alamos National Laboratory (LANL). Empire was selected

to be one of the examples in Griffin. Two different meshes in Griffin were generated by using Argonne’s Mesh Tools (AMT) system assisted with CUBIT (CUBIT, 2021): (1) Coarse mesh in Figure 4-6(a) & (2) Fine mesh in Figure 4-7(a).

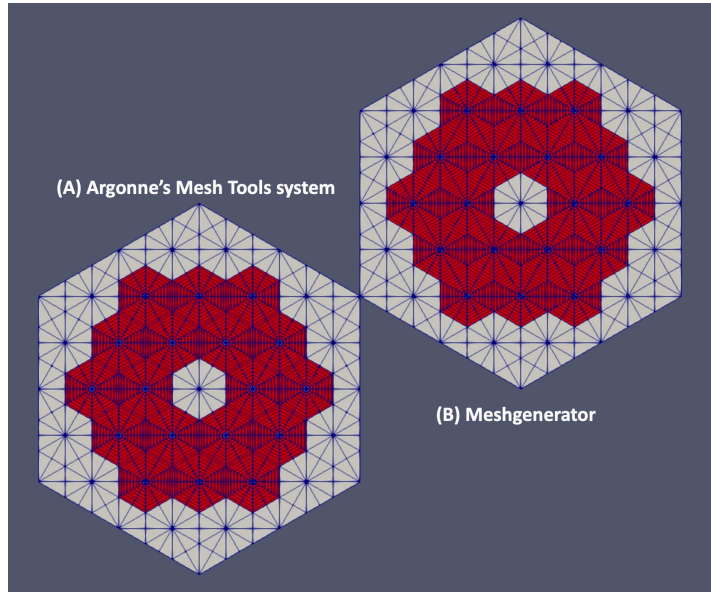


Figure 4-6. Coarse meshes of Empire Core: (A) mesh generated using Argonne’s Mesh Tools system; and (B) mesh generated using MOOSE meshgenerators

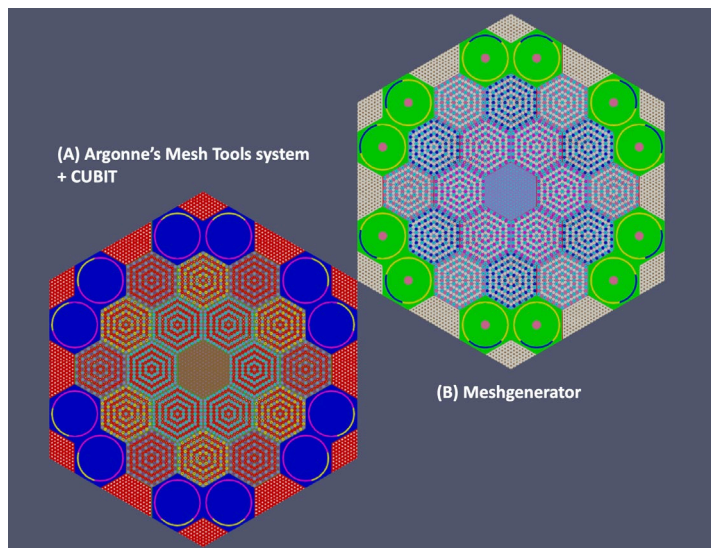


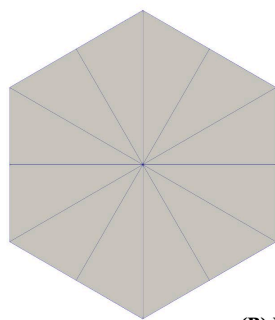
Figure 4-7. Fine meshes of Empire Core: (A) mesh generated using Argonne’s Mesh Tools system and CUBIT; and (B) mesh generated using MOOSE meshgenerators

The recent developed capabilities in MOOSE enabled these meshes to be generated directly in MOOSE without other tools.

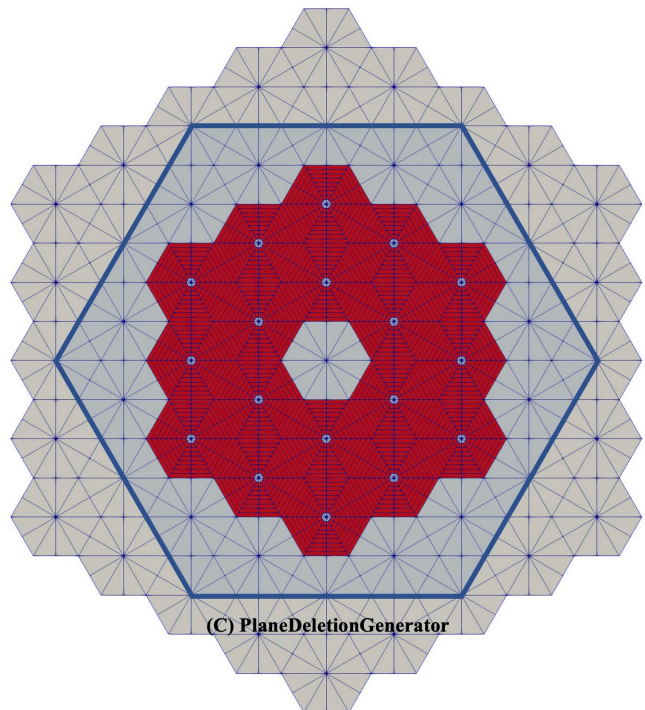
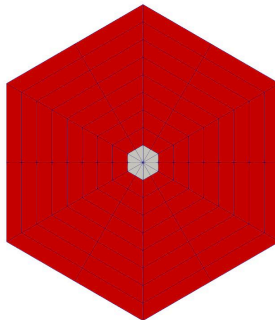
The coarse Empire mesh was generated using three objects in meshgenerator systems:

1. PolygonConcentricCircleMeshGenerator: generate two unit hexagonal meshes (Figure 4-8(A)).
2. PatternedHexMeshGenerator: stitch the unit hexagonal meshes with a given pattern to form the reactor core (Figure 4-8(B)).
3. PlaneDeletionGenerator: trim the mesh into hexagonal shape with flat boundaries, by using the PlaneDeletionGenerator to remove the meshes outside the blue hexagon region (Figure 4-8(C)).

(A) PolygonConcentricCircleMeshGenerator



(B) PatternedHexMeshGenerator



(C) PlaneDeletionGenerator

Figure 4-8. Construction of coarse Empire mesh

Construction of the fine Empire mesh is more complicated. The procedure to construct the fine Empire mesh is show below and demonstrated in Figure 4-9:

1. Generation of unit hexagonal mesh (fuel, heat pipe and moderator pin cells) using PolygonConcentricCircleMeshGenerator (Figure 4-9(A)).
2. Construction of the unit assembly mesh by stitching the unit hexagonal meshes with a given pattern using PatternedHexMeshGenerator (Figure 4-9(B)).
3. Generation of control drum mesh using HexagonConcentricCircleAdaptiveBoundaryMeshGenerator and AzimuthalBlockSplitGenerator (Figure 4-9(C)).

4. Generation of air hole mesh using PolygonConcentricCircleMeshGenerator (Figure 4-9(D)).
5. Construction of Empire core mesh by stitching the assembly, control drum and air hole meshes with a given pattern, using PatternedHexMeshGenerator (Figure 4-9(E)).
6. Trimming the Empire core mesh into hexagonal shape with flat boundaries, by using PlaneDeletionGenerator to remove the meshes outside the blue hexagon frame (Figure 4-9(F)).

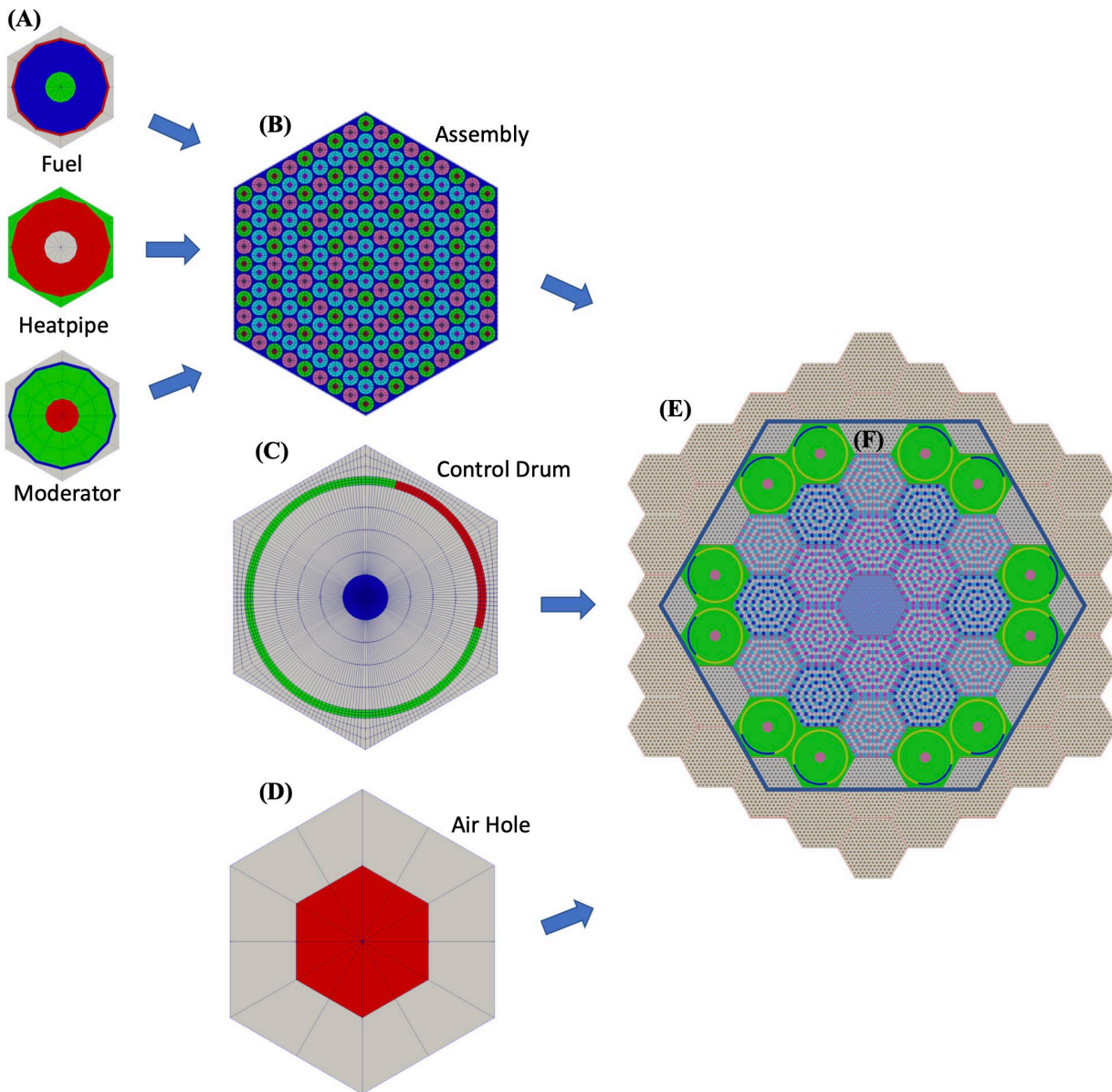


Figure 4-9. Construction of fine Empire mesh

Figure 4-6(b) & Figure 4-7(b) show the coarse and fine meshes generated using MOOSE meshgenerators. There is no visible difference in Empire coarse meshes generated by MOOSE and

AMT. The only difference in Empire fine meshes by MOOSE and AMT with CUBIT is the control drum mesh (Figure 4-9(c)). For the MOOSE mesh, the center of the control drum mesh was meshed by triangular elements using “HexagonConcentricCircleAdaptiveBoundaryMeshGenerator,” while CUBIT used the “pave” algorithm to mesh the control drum by quadrilateral elements.

Griffin was used to compare the eigenvalue results between a CUBIT/AMT-based input mesh and a mesh generated entirely with the Reactor module. The Griffin simulation used 11 energy groups and a P3A4 Gauss-Chebyshev cubature, and vacuum boundary conditions were applied around the entire 2-D mesh. In addition, the DFEM-SN solver with CMFD acceleration was used with the fine mesh and coarse mesh generation procedure described above. The eigenvalue results show close agreement of 20pcm between the two methods for mesh generation.

Table 4-2. Griffin-computed k-effective results for 2-D EMPIRE Problem: MOOSE mesh vs. CUBIT / Argonne Mesh Tools.

| Mesh Generation Tool | Griffin Solver Scheme | K-Effective |
|---|-----------------------------------|--------------------|
| MOOSE Reactor Module Mesh Generators | DFEM-SN with CMFD Acceleration | 1.20068 |
| Argonne Mesh Tools / CUBIT | DFEM-SN with CMFD Acceleration | 1.20088 |

4.3.2 3D HPMR Model

As the EMPIRE examples in Griffin is only for 2D neutronic simulations, a 3D HPMR core design is employed to demonstrate and verify capabilities of MOOSE mesh generator system for 3D reactor geometry generation. This 3D HPMR design is similar to the example reported in FY21 (Shemon, et al., 2021) but with an additional outer shield zone. A recent developed object, “PeripheralRingMeshGenerator,” was utilized to generate the mesh in the outer shield zone.

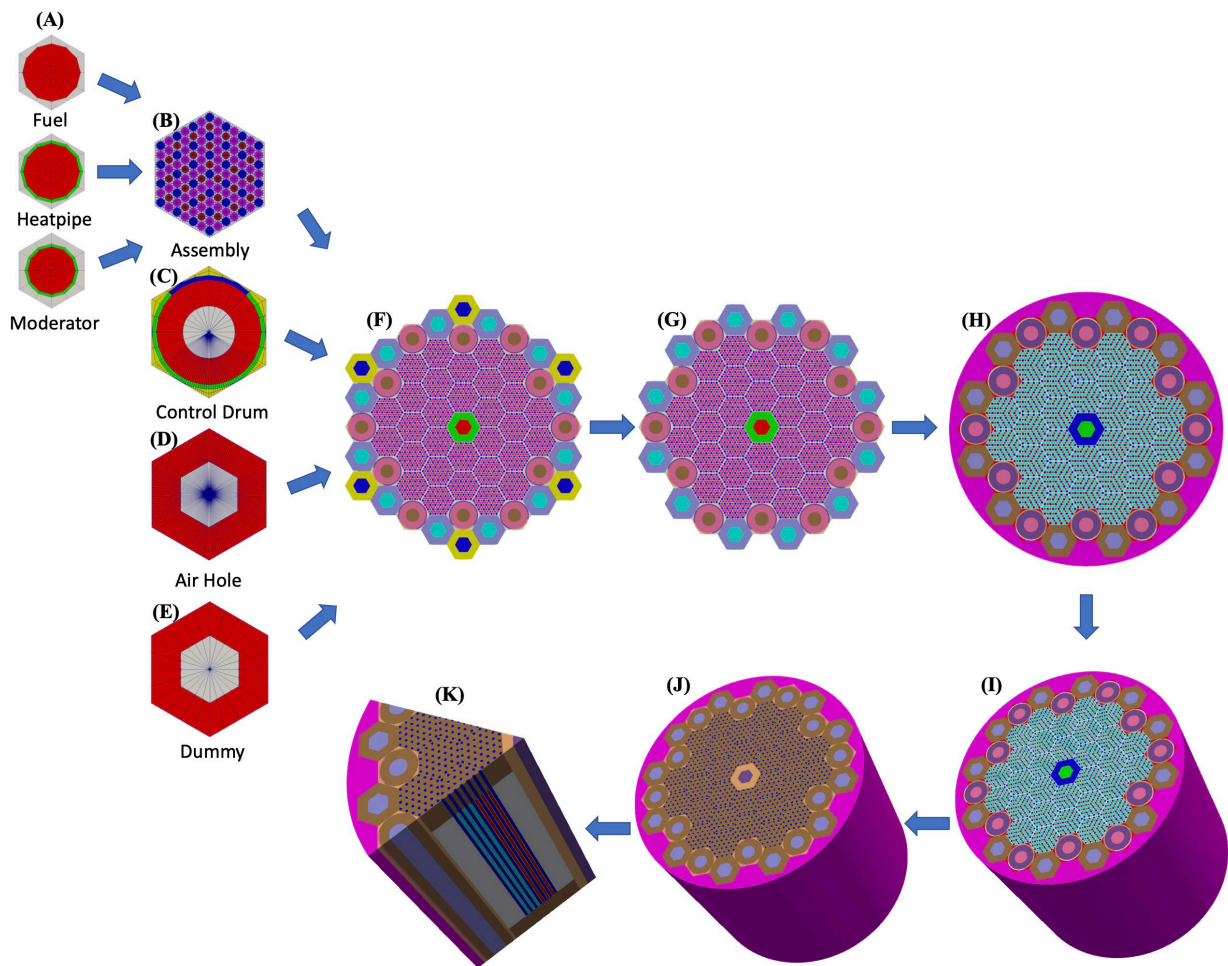


Figure 4-10. Construction of 3D HPMR mesh

Construction of the 3D HPMR mesh is similar to that of the Empire fine mesh but with an extrusion to 3D structure (Figure 4-10):

1. Generation of unit hexagonal mesh (fuel, heat pipe and moderator pin cells) using PolygonConcentricCircleMeshGenerator (Figure 4-10(A)).
2. Construction of the unit assembly mesh by stitching the unit hexagonal meshes with a given pattern using PatternedHexMeshGenerator (Figure 4-10 (B)).
3. Generation of control drum mesh using HexagonConcentricCircleAdaptiveBoundaryMeshGenerator and AzimuthalBlockSplitGenerator (Figure 4-10 I).
4. Generation of air hole mesh using HexagonConcentricCircleAdaptiveBoundaryMeshGenerator (Figure 4-10 (D)).
5. Generation of dummy mesh using PolygonConcentricCircleMeshGenerator (Figure 4-10 I).

6. Construction of 2D HPMR core mesh by stitching the assembly, control drum, air hole and dummy meshes with a given pattern, using `PatternedHexMeshGenerator` (Figure 4-10 (F)).
7. Removal of the dummy meshes using `BlockDeletionGenerator` (Figure 4-10 (G)).
8. Addition of the outer shield zone using `PeripheralRingMeshGenerator` (Figure 4-10 (H))
9. Extrusion of the 2D HPMR core mesh to 3D mesh using `FancyExtruderGenerator` (Figure 4-10 (I))
10. Definition of upper and lower reflector regions using `ParsedSubdomainMeshGenerator` (Figure 4-10 (J)); herein, the 3D HPMR full core mesh is completed.
11. Slice of 1/6 core mesh from the full core mesh using `PlaneDeletionGenerator` (Figure 4-10 (K)); the 1/6 core mesh was used in Multiphysics simulation (Stauff, et al., 2021), which saved significant amount computation resource compared to the simulation using the full core mesh.

Griffin standalone neutronic calculations were performed based on the 1/6 full core mesh and showed reasonable agreements with the Monte Carlo Serpent calculations: depends on the SN transport solver and numbers of the polar and azimuthal angles with SN, the difference in keff ranges from ~10 to ~500 pcm (Stauff, Personal Communication, 2022).

4.3.3 KRUSTY Heat-Pipe Microreactor Mesh

MOOSE-based mesh generation using the newly available tools was also performed for KRUSTY, a prototypic nuclear-powered test of a fission space reactor (Poston, Gibson, Godfroy, & McClure, 2020). The reactor was fueled with U8Mo (the actual weight fraction of Mo was 7.65% to produce 1 kW electric power). The heat generated in the core was carried to the Stirling converters using heat pipes for electric power generation.

Generation of a full core mesh was performed in preparation for multiphysics (neutronics, thermo-mechanics, and heat pipe analysis) simulation of KRUSTY using the MultiApp System of MOOSE (Gaston, et al., 2015). Due to the geometrical complexity of KRUSTY, the core model was simplified and initially meshed with Cubit by Los Alamos National Laboratory (Wilkerson, B., et. al., 2021) for multiphysics simulation. Due to high demand for computation resource, a half symmetric core was meshed as shown in Figure 4-11 (a-b). The LANL model was rebuilt using the MOOSE meshgenerators as shown in Figure 4-11 (c-d). Unlike the examples of HPMR and GCMR, the process to build the KRUSTY core is not hierarchical. Construction of the KRUSTY mesh relies on the `FillBetweenPointVectorsGenerator` and `FillBetweenSidesetsGenerator` objects (described in Section VII.A) to generate transition layers to connect vectors/boundaries to form irregular geometries like the fuel zones with heat pipes. The completed mesh is 1/16 angular symmetric and could be divided into half, quarter, 1/8, & 1/16 meshes using `PlaneDeletionGenerator`. Figure 4-12 shows the examples of full core, quarter and 1/16 meshes, all of which have been utilized in multiphysics computation. Leveraging symmetry to reduce the mesh size significantly decreases computational resource requirements for performing the physics simulation, which is particularly useful in the code testing stages and the steady-state simulation.

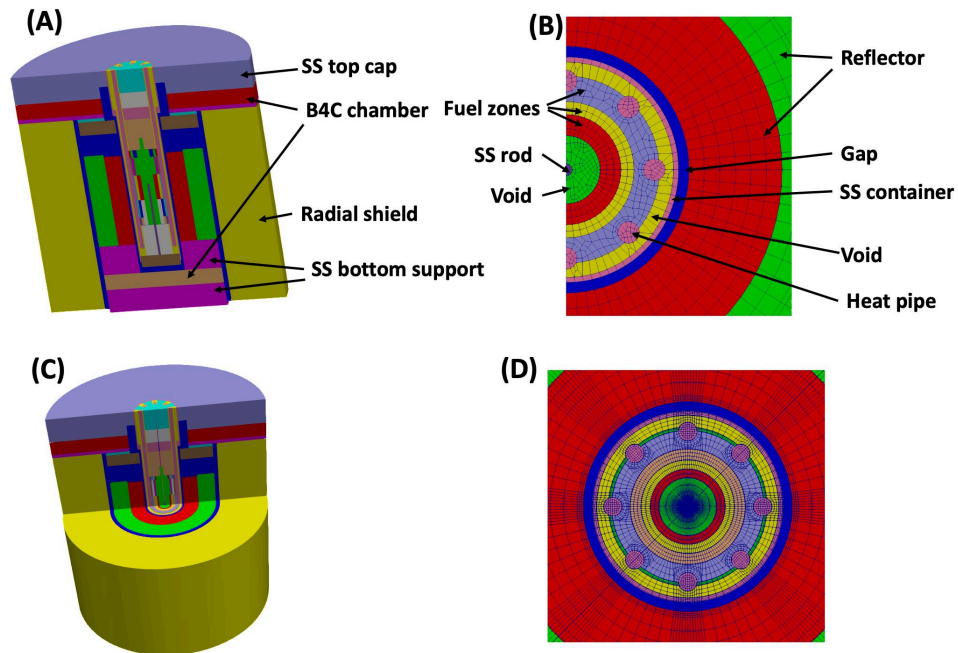


Figure 4-11. KRUSTY meshes: (a) half-core Cubit mesh; (b) cross-section of Cubit mesh near the fuel zone (core center); (c) full core mesh MOOSE mesh (a quarter of core was removed to show the internal structures); and (d) cross-section of MOOSE mesh near the fuel zone (core center).

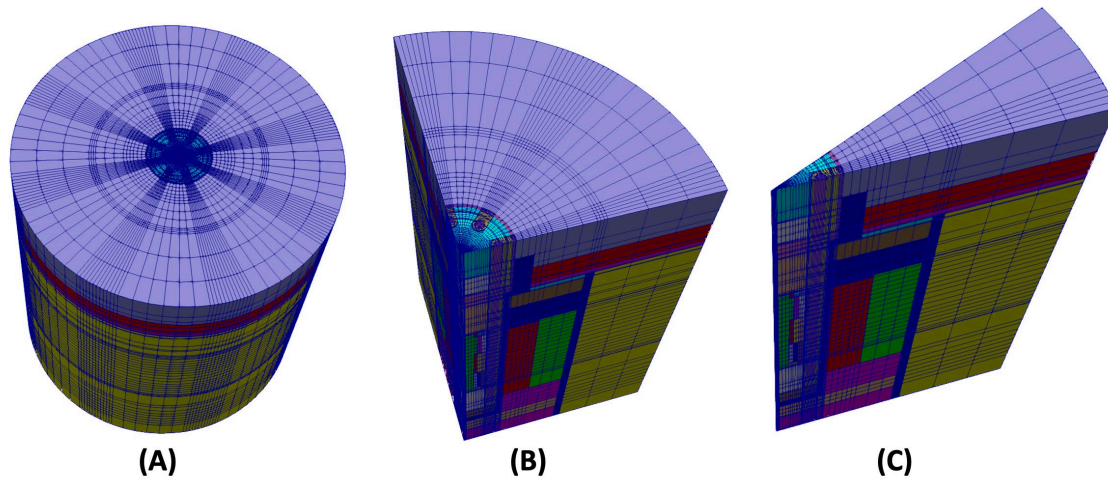


Figure 4-12. KRUSTY meshes generated using MOOSE: (a) full core, (b) quarter core, and (c) 1/16 core meshes

4.4 3D Gas-Cooled Microreactor Model

In addition to heat pipe-cooled designs, gas-cooled designs have also been adopted by microreactor developers. HolosGen LLC is developing a gas-cooled microreactor (GCMR) that

can provide safe, mobile and low-cost nuclear energy (htt2). A GCMR assembly model was developed at ANL (Stauff, et al., 2021) to evaluate the performance of NEAMS tools (including BISON, Griffin and Sockeye). The 3D GCMR assembly model was employed to test and verify the performance of MOOSE meshgenerators.

The procedure to construct the 3D GCMR assembly mesh is show below and demonstrated in Figure 4:

1. Generation of unit hexagonal mesh (yttrium hydride (YH), fuel, poison, coolant hole, and control rod pin cells) using PolygonConcentricCircleMeshGenerator (Figure 4(A)); note that two different fuel and coolant hole meshes are generated: the meshes on the assembly boundary use triangular elements in the center to facilitate the trimming (see step 3 below), while the meshes not on the assembly boundary use quadrilateral elements.
2. Construction of the unit assembly mesh by stitching the unit hexagonal meshes with a given pattern using PatternedHexMeshGenerator (Figure 4(B)).
3. Trimming the assembly mesh into hexagonal shape with flat boundaries, by using PlaneDeletionGenerator to remove the meshes outside the white hexagon frame (Figure 4(C)).
4. Extrusion of the 2D assembly mesh to 3D mesh using FancyExtruderGenerator; and finally, defining upper and lower reflector regions using ParsedSubdomainMeshGenerator (Figure 4(D)).

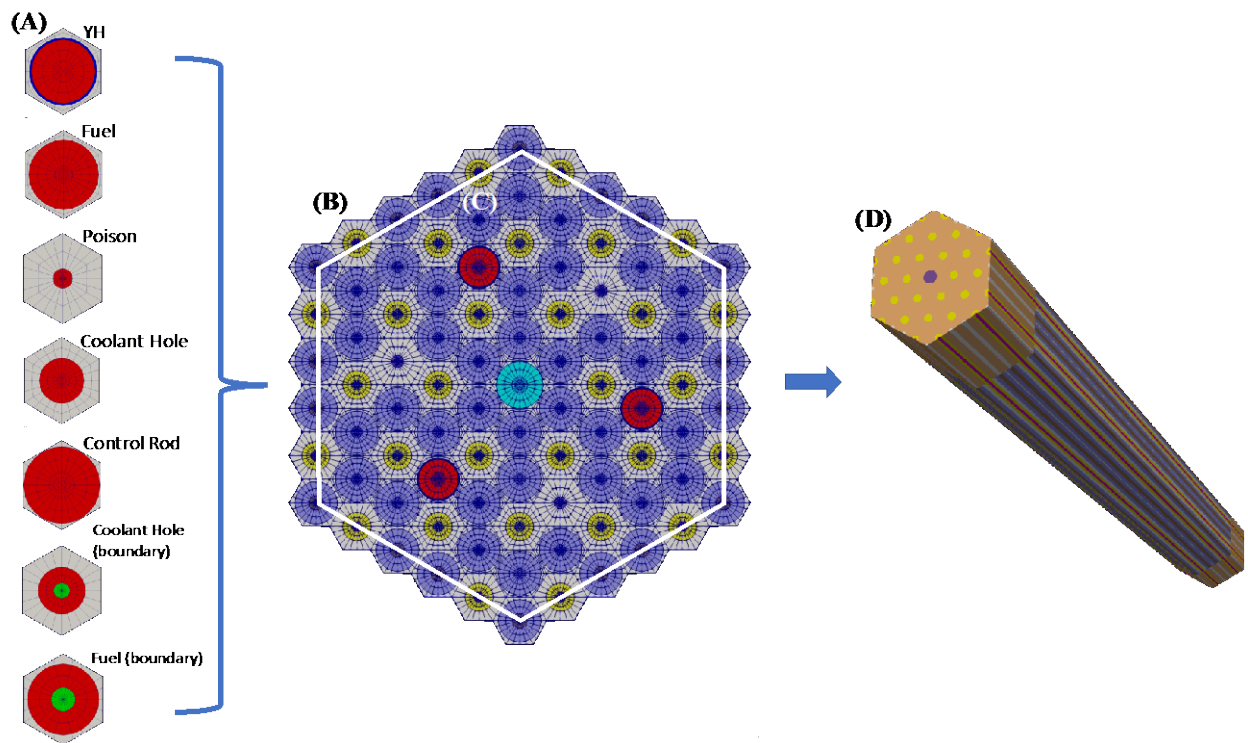


Figure 4-13. Construction of 3D GCMR assembly mesh

Griffin standalone neutronic calculations were performed based on the 3D GCMR assembly mesh and showed reasonable agreements with the Monte Carlo Serpent calculations: depends on the S_N transport solver and numbers of the polar and azimuthal angles with S_N , the difference in k_{eff} ranges from ~ 40 to ~ 500 pcm (Stauff, Personal Communication, 2022).

4.5 Molten-Salt Reactor Experiment

The Molten-Salt Reactor Experiment (MSRE) is an 8MWth molten salt reactor developed at Oak Ridge National Laboratory (ORNL) and operated in 1960s (Haubenreich & Engel, 1970). The experiments conducted at the MSRE provides invaluable benchmark experimental data for modelling and simulation of molten salt reactors. Fuel depletion computation using MSRE data was conducted at Argonne based on MSRE lattice (Fei, Shahbazi, Fang, & Shaver, 2022) with plans to employ a full core model soon. Figure 4-14 shows the MSRE mesh generated by MOOSE mesh generators for fuel depletion computation. More details of the model can be found in (Fratoni, Shen, Ilas, & Powers, 2020). Except for the control rods, the full core mesh was built hierarchically from the MSRE unit lattice containing graphite and fuel. The control rod regions were built with the FillBetweenSidesetsGenerator object and stitched to other MSRE lattices to form the 2D mesh as shown in Figure 4-14(b). The 3D core mesh in Figure 4-14 (a) was completed by extruding the 2D mesh using FancyExtruderGenerator (now named AdvancedExtruderGenerator). This example shows the type of complex geometry now capable through MOOSE.

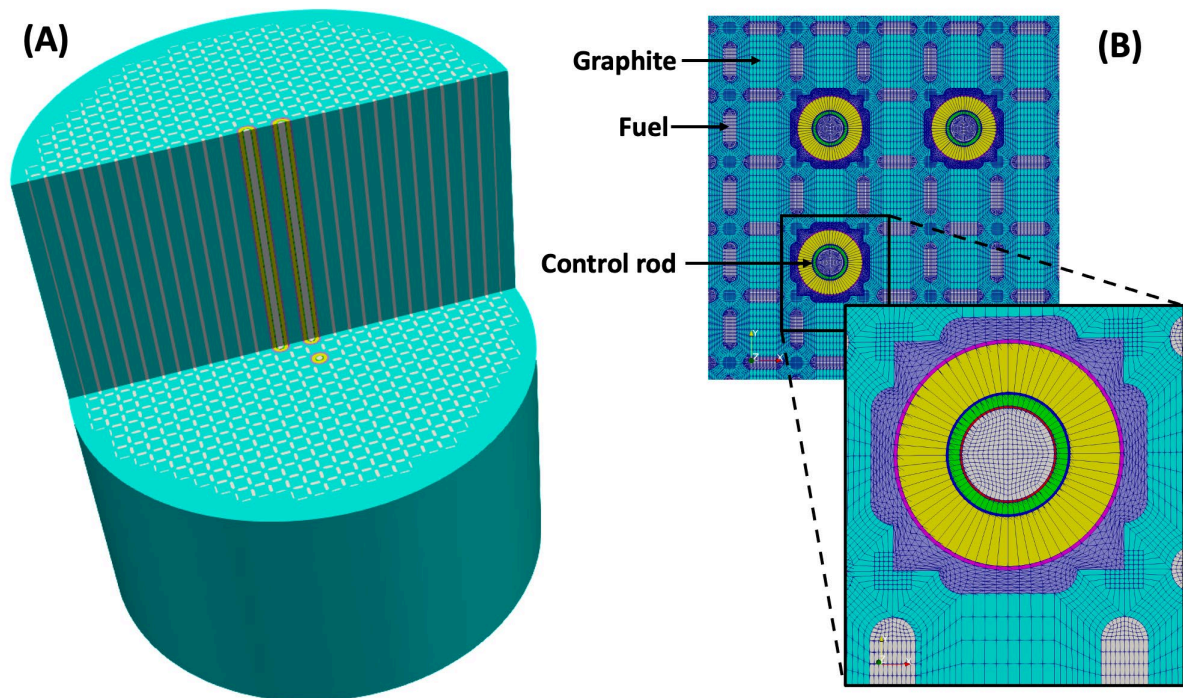


Figure 4-14. MOOSE-generated MSRE mesh: (a) 3D full core (about a quarter of core was removed to show the internal structures); and (b) cross-section near the fuel zone (core center) and view of control rod zone

5 Summary and Future Work

Numerous capabilities have been added to the Reactor Module during FY22. Reactor Module capabilities continue to be tested using NEAMS tools for analysis of various reactor concepts. The aim of this module is to streamline the mesh generation process for NEAMS users analyzing reactor geometries. The capabilities implemented this year were primarily driven by stakeholder request and prioritized by broadest impact. Additional capabilities are planned to make the tools easier to use and more flexible to accommodate additional reactor types and geometries. Several areas of future work are proposed to continue enhancing MOOSE's native meshing capability for NEAMS users. Some priorities are listed here, although this list is not exhaustive.

- Development of a comprehensive tutorial and training
- Incorporate depletion ID assignment in RGMB
- Concisely define lattice similar assemblies while using different block or region names to permit different material assignments
- Add functionality to RGMB (Cartesian ducts, ability to create a control drum object, enhanced sideset control, homogenized assemblies)
- Refactor PeripheralTriangleMeshGenerator to use the upcoming Delaunay triangulation mesh generator to improve quality of triangulated triangles and/or extend outer boundary of to additional shapes besides circle, set of points (e.g., from external program), and/or boundary of existing mesh generator mesh (if core periphery is part of another mesh)
- Enhanced sideset support (includes items like supporting internal “paired” sidesets needed for TH codes, permit sidesets to be constructed with combinatorial operations, construction of nodesets along axial duct corners / edges, develop robust numbering/naming scheme for sideset generation to avoid conflicts between mesh generators)
- Upgrade PatternedMeshGenerator capabilities to streamline sideset definition for Cartesian geometries (started in FY22)
- Evaluation of Rocstar Illinois meshing capabilities developed under SBIR for inclusion in MOOSE
- Simplify meshing for MultiApps system (long term goal). For example, for a Griffin-BISON Multiphysics simulation, a coarse and gap-free mesh can be generated for Griffin, and a finer mesh with more component details can be generated for BISON using a single mesh generator block.
- Continued physics verification of new capabilities

REFERENCES

- (n.d.). Retrieved from <http://www.holosgen.com>
- CUBIT. (2021). Retrieved from The CUBIT Geometry & Meshing Toolkit: <https://cubit.sandia.gov/>
- Dasari, V. R., Trellue, H. R., & Arafat, Y. (2020). *Microreactors: A Technology Option for Accelerated Innovation*. LA-UR-20-22435: Los Alamos National Laboratory.
- DeHart, M., Ortensi, J., & Labouré, V. (2020). *NEAMS Reactor Physics Assessment Problem*. Lemont, IL: INL/LTD-20-59184, Idaho National Laboratory.
- Fei, T., Shahbazi, S., Fang, J., & Shaver, D. (2022). *Validation of NEAMS Tools Using MSRE Data*. ANL/NSE-22/48: Argonne National Laboratory.
- Fratoni, M., Shen, D., Ilas, G., & Powers, J. (2020). *Molten Salt Reactor Experiment Benchmark Evaluation*. DOE-UCB-8542: University of California-Berkeley and Oak Ridge National Laboratory.
- Gaston, D., Permann, C., Peterson, J., Slaughter, A., Andrs, D., Wang, Y., . . . Martineau, R. (2015). Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84, 45-54.
- Grasso, G., Levinsky, A., Franceschini, F., & Ferroni, P. (2019). A MOX-fuel core configuration for the Westinghouse Lead Fast Reactor. *ICAPP - International Congress on Advances in Nuclear Power Plants*. France.
- Hasse, J. N. (2021). *poly2tri*. Retrieved from GitHub: <https://github.com/jhasse/poly2tri>
- Haubenreich, P. N., & Engel, J. R. (1970). Experience with the molten-salt reactor experiment. *Nuclear Applications and Technology*, 8(2), 118-136.
- Kumar, S., Mo, K., Shemon, E., Miao, Y., Wozniak, N., & Jung, Y. S. (2022). Physics Demonstration and Verification of MOOSE Framework Reactor Module Meshing Capabilities. *International Conference on Physics of Reactors*, (pp. 221-231). Pittsburgh, PA.
- Lee, C., Jung, Y. S., & Yang, W. S. (2018). *MC2-3: Multigroup Cross Section Generation Code for Fast Reactor Analysis*. ANL/NE-11/41: Argonne National Laboratory.
- Lee, C., Jung, Y., Park, H., Shemon, E., Ortensi, J., Wang, Y., . . . Prince, Z. (2021). *Griffin Software Development Plan*. INL/EXT-21-63185 & ANL/NSE-21/23, Idaho National Laboratory and Argonne National Lab Technical Report.
- Lefebvre, R., Langley, B., Miller, P., Delchini, M., Baird, M., & Lefebvre, J. (2019). *NEAMS Workbench Status and Capabilities*. ORNL/TM-2019/1314, Oak Ridge National Laboratory.
- Lewis, E. e. (2001). *Benchmark for Deterministic 2-D/3-D MOX Fuel Assembly Transport Calculations*. NEA/NSC/DOC (2003): Nuclear Energy Agency.
- Lindsay, A., Kong, F., Guidicelli, G., Carlsen, R., Stogner, R., & Slaughter, R. (2021). *NEAMS-Multiphysics Technical Assistance in FY-21*. Idaho Falls, ID: INL/EXT-21-64493, Idaho National Laboratory.

- Matthews, C., Laboure, V., DeHart, M., Hansel, J., Andrs, D., Wang, Y., . . . Martineau, R. C. (2021). Coupled Multiphysics Simulations of Heat Pipe Microreactors Using DireWolf. *Nuclear Technology*, 207(7), 1142-1162.
- Peakman, A., Hodgson, Z., & Merk, B. (2018). Advanced micro-reactor concepts. *Progress in Nuclear Energy*, 107, 61-70.
- Permann, C., Gaston, D., Andrs, D., Stogner, R., Carlsen, R., Kong, F., . . . Martineau, R. (2020). MOOSE: Enabling massively parallel multiphysics simulation. *SoftwareX*, 11(2352-7110), 100430. doi:<https://doi.org/10.1016/j.softx.2020.100430>
- Poston, D. I., Gibson, M. A., Godfroy, T., & McClure, P. R. (2020). KRUSTY Reactor Design. *Nuclear Technology*, V206, S13-S30.
- Shemon, E., Grudzinski, J., Lee, C., Thomas, J., & Yu, Y. (2015). *Specification of the Advanced Burner Test Reactor Multi-Physics Coupling Demonstration Problem*. ANL/NE-15/43, Argonne National Laboratory.
- Shemon, E., Jung, Y. S., Kumar, S., Miao, Y., Mo, K., Oaks, A., & Richards, S. (2021). *MOOSE Framework Meshing Enhancements to Support Reactor Physics Analysis*. ANL/NSE-21/43: Argonne National Laboratory.
- Shemon, E., Mo, K., Miao, Y., Jung, Y. S., Richards, S., Oaks, A., & Kumar, S. (2022). MOOSE Framework Enhancements for Meshing Reactor Geometries. *International Conference on Physics of Reactors*, (pp. 210-220). Pittsburgh, PA.
- Shemon, E., Smith, M., & Lee, C. (2016). *PROTEUS-SN User Manual*. Argonne, IL: ANL/NE-14/6 (Rev 3.0), Argonne National Laboratory. doi:10.2172/1240157
- Shiozawa, S., Fujikawa, S., Iyoku, T., Kunitomi, K., & Tachibana, Y. (2004). Overview of HTTR Design Features. *Nuclear Engineering and Design*, 233(1-3), 11-21.
- Smith, M., & Shemon, E. (2015). *User Manual for the PROTEUS Mesh Tools*. Argonne, IL: ANL/NE-15/17 Rev. 1.0, Argonne National Laboratory. doi:10.2172/1212714
- Stanek, C. (2019). *Overview of DOE-NE NEAMS Program*. Los Alamos, NM: LA-UR-19-22247, Los Alamos National Laboratory. doi:<https://doi.org/10.2172/1501761>
- Stauff, N. (2022). Personal Communication.
- Stauff, N., Mo, K., Cao, Y., Thomas, J., Miao, Y., Lee, C., . . . Feng, B. (2021). Preliminary Applications of NEAMS Codes for Multiphysics Modeling of a Heat Pipe Microreactor. *ANS Transactions*. 124, pp. 21-24. American Nuclear Society.
- US DOE Office of Nuclear Energy. (2022). *What is a Nuclear Microreactor?* Retrieved from <https://www.energy.gov/ne/articles/what-nuclear-microreactor>
- Wilkerson, B., et. al. (2021). *Thermo-Mechanical Neutronic Considerations of the KRUSTY Criticality Experiment*. LA-UR-21-30496: Los Alamos National Laboratory.
- Williamson, R., Hales, J., Novascone, S., Pastore, G., Gamble, K., Spencer, B., . . . Chen, H. (2021). BISON: A Flexible Code for Advanced Simulation of the Performance of Multiple Nuclear Fuel Forms. *Nuclear Technology*, 207(7), 954-980. doi:doi.org/10.1080/00295450.2020.1836940



Nuclear Science and Engineering Division

Argonne National Laboratory
9700 South Cass Avenue, Bldg. 208
Argonne, IL 60439

www.anl.gov



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