ANL/NSE-21/43

MOOSE Framework 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.

ANL/NSE-21/43

MOOSE Framework Meshing Enhancements to Support Reactor Physics Analysis

---

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

September 15, 2021

EXECUTIVE ABSTRACT

MOOSE-based physics codes require an input finite element mesh on which the physics solution is calculated, reported, and transferred to other physics codes. The use of difficult-to-use, external licensed software is often required to generate high quality meshes for reactor geometries. High-fidelity geometry modeling also requires elaborate tracking of groups of elements for material property assignment and output reporting which can be considerably complex for the user to identify and maintain. Under the U.S. Department of Energy Office of Nuclear Energy Advanced Modeling and Simulation (NEAMS) program, several meshing-related enhancements have been developed for the MOOSE framework to address user challenges in creating finite element meshes for advanced reactor geometries.

MOOSE mesh generators have been developed to mesh hexagonal geometries (pins, ducted assemblies, and cores) commonly found in liquid-metal cooled fast reactor concepts. The mesh generator used for hexagonal pin cells is generic for regular polygons and therefore may also be used for Cartesian pin cells. Hexagonal pin cells can be stitched into ducted assemblies, and assemblies can be stitched together into a core. The user may specify region ids, region names, and other preferences on the mesh. This control is useful for later material mapping in the MOOSE-based physics codes input. A capability was also developed for meshing rotating control drums including determination of material volume fractions in each mesh element as a function of time. Control drum meshes may be stitched to other hexagonal assemblies to create a core configuration.

Additional mesh generators were developed that wrap around the hexagonal meshing capabilities and utilize “extra element integer” ID values on each element. In regular Cartesian or hexagonal assemblies or cores, the bookkeeping of element groups for both material assignment and output reporting can now be automated through assignment of pin, assembly, core, axial and depletion id values stored as extra element integers. The extra element tags on the mesh greatly speed the reactor analyst’s efforts to map materials to meshes, track depletion zones, and parse output such as axial pin power distributions.

At the highest level, pin, assembly, and core mesh generators (with this reactor terminology) have also been developed to easily generate regular Cartesian and hexagonal cores, including axial extrusion. These reactor geometry builders call upon the previously mentioned capabilities to produce analysis-ready 3D meshes including material assignments.

Open source mesh triangulation capabilities were also investigated for integration into the MOOSE framework to address the need for meshing the core periphery region which extends from the irregular outer assembly border to a cylindrical boundary. Options are limited due to licensing constraints, and the recommendation is pursue building a native MOOSE Delaunay triangulator routine with full functionality.

Finally, a series of verification problems were performed with NEAMS physics tools. All developed capabilities will be available in the new open-source “Reactor” module of the MOOSE framework, which is accessible to any MOOSE-based NEAMS physics tool.

ACKNOWLEDGEMENTS

This work was funded by the Department of Energy Nuclear Energy Advanced Modeling and Simulation Program (DOE-NEAMS).

The authors gratefully acknowledge the guidance provided by MOOSE (Derek Gaston, Cody Permann), Griffin (Changho Lee, Javier Ortensi, Yaqi Wang), and Bison (Stephen Novascone, Benjamin Spencer) development teams, as well as staff working under the NEAMS Application Drivers Technical Area to identify priorities for nuclear reactor analysis meshing improvements. We thank Nicholas Wozniak of Argonne National Laboratory for graciously assisting with a MOOSE Tensor Mechanics verification problem detailed in this report.

The authors also gratefully acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

Table of Contents

REPORTS NOT IN DIGITAL FORMAT ARE AVAILABLE TO DOE AND DOE CONTRACTORS FROM THE OFFICE OF SCIENTIFIC AND TECHNICAL INFORMATION (OSTI): 2

EXECUTIVE ABSTRACT I

ACKNOWLEDGEMENTS II

TABLE OF CONTENTS III

LIST OF FIGURES V

LIST OF TABLES VIII

1 INTRODUCTION 1

2 TASK IDENTIFICATION 2

3 HEXAGONAL GEOMETRY MESHING CAPABILITY 4

3.1 Reactor Analysis Motivation 4
3.2 Homogeneous Hexagonal Pin Cells and Assemblies 5
3.2.1 SimpleHexagonGenerator 5
3.2.2 PolygonConcentricCircleMeshGenerator 5
3.3 Heterogeneous Hexagonal Pin Cells 8
3.3.1 PolygonConcentricCircleMeshGenerator 8
3.3.2 HexagonConcentricCircleAdaptiveBoundaryMeshGenerator 14
3.3.3 IDs and Names of Blocks and Boundaries 15
3.4 Hexagonal Assemblies and Cores 16
3.4.1 PatternedHexMeshGenerator 16
3.5 Current Limitations 20

4 CONTROL DRUM GEOMETRY MESHING CAPABILITY 22

4.1 Reactor Analysis Motivation 22
4.2 Steady State Control Drum Position 22
4.2.1 AzimuthalBlockIDMeshGenerator 22
4.3 Time-Dependent Control Drum Rotation 25
4.3.1 Use of PatternedHexMeshGenerator MeshMetaData 25
4.3.2 MultiControlDrumFunction Rotation of Control Drums 26
4.4 Current Limitations 28

5 REPORTING ID CAPABILITY 30

5.1 Reactor Analysis Motivation 30
5.2 Assigning Pin, Assembly and Plane Reporting IDs 30
5.3 Assigning Depletion IDs 35

6 CORE PERIPHERY MESHING CAPABILITY 37

6.1 Reactor Analysis Motivation 37
6.2 Triangle Licensing Compatibility Issues 37
6.3 Core Periphery Triangulation Mesher 38
6.4 Other Uses of the Triangulation Mesher 46
6.5 Limitations and Recommendations 46

7 REACTOR GEOMETRY MESH BUILDER (RGMB) CAPABILITY 47

7.1 Reactor Analysis Motivation 47
7.2 Building a Repeated Reactor Geometry 47

7.2.1 GlobalMeshParams 47
7.2.2 PinMeshGenerator 48
7.2.3 AssemblyMeshGenerator 49
7.2.4 CoreMeshGenerator 50
7.2.5 Automatic Sideset Generation 50
7.3 CARTESIAN RGMB EXAMPLE 51
7.4 HEXAGONAL RGMB EXAMPLE 54
7.5 LIMITATIONS AND RECOMMENDATIONS 57

8 MISCELLANEOUS ENHANCEMENTS 58

8.1 UPDATING OF FANCYEXTRUDERGENERATOR 58
8.2 ELEMENT CENTROID LOCATER (2D 1ST ORDER ELEMENTS) 58

9 PHYSICS CODE VERIFICATION OF DEVELOPED CAPABILITIES 60

9.1 GRIFFIN VERIFICATION: HOMOGENIZED FAST REACTOR EXAMPLE 60
9.1.1 ABTR Mesh Generation in New MOOSE Mesh Tools 61
9.1.2 Griffin Neutronics Parameters for ABTR Problem and Comparison of Neutronics Results between New MOOSE Mesh Tools and Argonne's Mesh Tools 66
9.2 GRIFFIN VERIFICATION: HETEROGENEOUS FAST REACTOR ASSEMBLY EXAMPLE 67
9.2.1 LFR Mesh Generation in New MOOSE Mesh Tools 68
9.2.2 Comparison of Griffin Neutronics Results between New MOOSE Mesh Tools and Argonne's Mesh Tools for LFR problem 71
9.2.3 Extension of New MOOSE Meshing Capabilities to LFR Problem with Coarse Mesh Diffusion Acceleration 72
9.3 MOOSE TENSOR MECHANICS VERIFICATION: DUCTED HEXAGONAL ASSEMBLY EXAMPLE 74
9.3.1 Ducted Assembly Mesh Generation with MOOSE 76
9.3.2 Comparison of Results using MOOSE vs. Cubit meshes 79
9.4 MULTIPHYSICS VERIFICATION: MICROREACTOR EXAMPLE 81
9.4.1 Microreactor Core Mesh Generation with MOOSE Mesh Generators 81
9.4.2 BISON Heat-Conduction Simulation with MOOSE mesh 84
9.4.3 Griffin-BISON-Sockeye Multiphysics Simulations 86
9.5 SUMMARY OF PHYSICS TESTS 87

10 SUMMARY 88

REFERENCES 90

LIST OF FIGURES

Figure 3-1. Example LMFR geometries: heterogeneous pin cell (top left), heterogeneous ducted assembly (top right), extruded assembly (bottom left), homogeneous assemblies in core. 4
Figure 3-2. Homogenized pin cell/assembly with basic discretization (SimpleHexagonGenerator) 5
Figure 3-3. Homogenized pin cell/assembly with all tri mesh (PolygonConcentricCircleMeshGenerator)........ 6
Figure 3-4. Homogenized pin cell/assembly with all quad mesh (PolygonConcentricCircleMeshGenerator)........ 7
Figure 3-5. Homogenized pin cell/assembly showing extra block required due to >1 background intervals, mixed tri-quad mesh (PolygonConcentricCircleMeshGenerator)........ 7
Figure 3-6. Homogenized pin cell/assembly showing extra block required due to >1 background intervals, all quad mesh (PolygonConcentricCircleMeshGenerator)........ 7
Figure 3-7. Different node algorithms for Tri and Quad central elements 8
Figure 3-8. A schematic drawing showing the different regions generated by PolygonConcentricCircleMeshGenerator. 9
Figure 3-9. Heterogeneous hexagonal cell with pin and duct regions 12
Figure 3-10. Heterogeneous hexagonal cell with pin and duct regions, showing extra block required due to >1 subinterval in center zone 13
Figure 3-11. Heterogeneous hexagonal cell with pin and duct regions, showing varied azimuthal discretization per face and preservation of ring volumes 14
Figure 3-12. A mesh generated by `HexagonConcentricCircleAdaptiveBoundaryMeshGenerator`. Note that the side 0 of the hexagonal mesh adaptively matches side 3 of the input hexagonal mesh. 15
Figure 3-13. A schematic drawing showing the difference between "none" and "hexagon" *pattern_boundary* options. 16
Figure 3-14. A patterned hexagonal mesh based on unit mesh generated by `PolygonConcentricCircleMeshGenerator` with "hexagon" boundary 17
Figure 3-15. A patterned hexagonal mesh based on unit mesh generated by `PolygonConcentricCircleMeshGenerator` with "none" boundary 18
Figure 3-16. A patterned hexagonal mesh based on unit mesh generated by `PatternedHexMeshGenerator` with hexagonal boundary 19
Figure 3-17. A patterned hexagonal mesh with unit assembly meshes containing 7 and 19 pins. 21
Figure 4-1. A typical control drum structure and important geometrical parameters. 22
Figure 4-2. A schematic drawing showing the functionalities of this `AzimuthalBlockIDMeshGenerator` object. 23
Figure 4-3. Polygon meshes modified by `AzimuthalBlockIDMeshGenerator` without and with external boundary nodes moved. 24
Figure 4-4. A schematic drawing the indexing rule of `control_drum_id` in the `PatternedHexMeshGenerator` object. 26
Figure 4-5. An example of control drums simulated by `MultiControlDrumFunction` object. 28
Figure 5-1. Reporting ID Generation for Cartesian Geometry 31

Figure 5-2. Reporting ID Generation for Hexagonal Geometry 32
Figure 5-3. Illustration of Reporting ID Generation for Pins and Assemblies in Cartesian Lattice.. 32
Figure 5-4. Illustration of Reporting ID Generation for Pins and Assemblies in Hexagonal Lattice.. 33
Figure 5-5. Illustration of auto-numbering using exclude_id option 34
Figure 5-6. Illustration of Reporting ID Generation for Plane... 35
Figure 5-7. Depletion ID Generation using Reporting IDs.. 36
Figure 6-1: Reactor core geometry with meshed core periphery region (Cubit). 37
Figure 6-2: Example TMG mesh, simple GMG core. 39
Figure 6-3: Example TMG mesh, simple cartesian core... 40
Figure 6-4: Example TMG mesh, detailed hex assembly. 41
Figure 6-5: Example TMG mesh, detailed hex assembly with Steiner points......................... 42
Figure 6-6: Example TMG mesh, detailed hex core with Steiner points, and extruded into 3D. 43
Figure 6-7: Example TMG mesh, multi-assembly core... 44
Figure 6-8: Example TMG mesh, full core with (left) only outer boundary nodes, and (right) ring of Steiner points added. 45
Figure 7-1: 3D Cartesian core input and mesh built using MOOSE's new reactor geometry mesh builder. 53
Figure 7-2. 3D Hexagonal core input and mesh built using MOOSE's new reactor geometry mesh builder. 57
Figure 9-1. Top-down (left) and side (right) views of ABTR homogeneous model, generated by Argonne's Mesh Tools 60
Figure 9-2. ABTR 2-D Assembly Definition Using SimpleHexagonGenerator 61
Figure 9-3. ABTR 2-D Core Lattice Definition and Dummy Deletion 62
Figure 9-4. ABTR 3-D Extrusion Process 64
Figure 9-5. ABTR Sideset Renaming Process 65
Figure 9-6. MOOSE Mesh block when using MOOSE mesh tools (left) vs. Argonne's Mesh Tools (right) 66
Figure 9-7. Top-down view of entire assembly (left), zoomed top-down view of central pins (middle), and side view (right) of the LFR heterogeneous assembly model, generated by Argonne's Mesh Tools 68
Figure 9-8. LFR 2-D Pin Definition Using PolygonConcentricCircleMeshGenerator 69
Figure 9-9. LFR Assembly 2-D Pin Lattice Definition 70
Figure 9-10. LFR Assembly 3-D Extrusion Process... 71
Figure 9-11. LFR Assembly Sideset Renaming Process 71
Figure 9-12. Zoomed top-down view of background region between outermost pins and duct for LFR heterogeneous assembly model, generated by Argonne's Mesh Tools (left) and new MOOSE mesh tools (right) 72
Figure 9-13. Zoomed top-down view of fine-mesh LFR heterogeneous assembly (left) and coarse-mesh LFR heterogeneous assembly (right), generated by new MOOSE mesh tools (right) 73
Figure 9-14. Coarse Mesh LFR 2-D Pin Definition Using PolygonConcentricCircleMeshGenerator 73

Figure 9-15. Geometry for hexagonal ducted assembly: (left) 2D cross section at load pad, and (right) vertical cross section showing location of load pad and active core...................... 75
Figure 9-16. Schematic showing the thermal gradient developed axially along the core region of the duct, showing the temperature difference for different corners of the duct.................. 75
Figure 9-17. Maximum corner temperatures at the top of the core region 76
Figure 9-18 Ducted hexagonal assembly mesh generation.. 77
Figure 9-19 Ducted hexagonal assembly meshes: (a) mesh generated by Cubit; and (b) mesh generated by MOOSE 78
Figure 9-20. Ducted hexagonal assembly mesh cross section at load pad elevation, (left) generated by Cubit, and (right) generated by MOOSE... 78
Figure 9-21. Cubit mesh displacement in m (left), and MOOSE displacement in m (right). . 79
Figure 9-22. Duct thermal deflection displacement (Cubit and MOOSE meshes)................. 80
Figure 9-23. Microreactor core meshes produced by CUBIT (number of nodes: 1.7×106): (a) 1/6 symmetric core mesh, (b) cross-section of the core, and (c) zoom-in region of (b); and MOOSE mesh generators (number of nodes: 1.0×106): (d) 1/6 symmetric core full core mesh, (e) cross-section of the core, and (f) zoom-in region of (e).. 82
Figure 9-24. 2D assembly mesh generation showing moderator, heat pipe, and fuel pin cell being combined into a hexagonal pattern.. 82
Figure 9-25. 2D core mesh generation showing assemblies (fuel, control drum, reflector, dummy) being combined into a core... 83
Figure 9-26. 3D 1/6 symmetric core mesh generation: (a) 2D core mesh; (b) trimmed (dummy blocks removal) and sliced 2D 1/6 symmetric core mesh; (c) extruded 3D 1/6 symmetric core mesh; and (d) completed 3D 1/6 symmetric core mesh after defining axial blocks 83
Figure 9-27. Temperature distribution of the 1/6 symmetric core at the last time step (20 sec) of simulation: (a) simulation based on the mesh generated by CUBIT (b) simulation based on the mesh generated by meshgenerators.. 84
Figure 9-28. Temperature evolution of the heat conduction simulation: (a) average fuel temperature, and (b) average heat pipe surface temperature.. 85
Figure 9-29. Power density comparison of multiphysics simulations using the two different meshes (unit: W/m^3).. 85
Figure 9-30. Temperature comparison of Multiphysics simulations using the two different meshes (unit: K). 86

LIST OF TABLES

Table 3-1. Guidelines for when an extra block is added to the center zone in PolygonConcentricCircleMeshGenerator ... 10
Table 4-1. Geometry parameters needed for MultiControlDrumFunction ... 27
Table 7-1. Description of auto-generated sidesets ... 51
Table 8-1. Generalized algorithm to calculate element centroids ... 59
Table 9-1. Griffin k-effective results between new MOOSE mesh tools and Argonne's Mesh Tools for the ABTR Problem ... 67
Table 9-2. Description of Parameters Used in Input Block for Pin Definitions in Figure 9-8 69
Table 9-3. Griffin k-effective results between new MOOSE mesh tools and Argonne's Mesh Tools for the LFR Assembly Problem ... 72
Table 9-4. Griffin k-effective results between new MOOSE mesh tools and Argonne's Mesh Tools for the LFR Assembly Problem with Coarse Mesh Diffusion Acceleration ... 74
Table 9-5. Comparison of centerline deflection results between the Cubit mesh and the MOOSE hex mesh generator, at the top of the core region, the ACLP and TLP midplanes ... 80
Table 9-6. Key calculated parameters comparison between the two meshes (temperature unit: K) ... 87

1 Introduction

Multiphysics Object Oriented Simulation Environment (MOOSE) (Permann, et al., 2020) is an open-source, parallel finite element framework designed to permit rapid development of robust multi-physics modeling capabilities. The Department of Energy (DOE) Nuclear Energy Advanced Modeling and Simulation Program (NEAMS) program (Stanek, 2019) is tasked with the development, demonstration, and deployment of advanced simulation tools for modeling light water reactors (LWR) and advanced reactors (AR). NEAMS has developed a suite of MOOSE-based physics tools including Griffin (Lee, et al., 2021), BISON (Williamson, et al., 2021), and others which leverage the finite element geometry representation, solvers, and coupling options available in MOOSE. This toolset performs predictive multiphysics simulations of advanced reactors including liquid-metal cooled fast reactors (LMFR), molten salt reactors (MSR), high temperature gas-cooled reactors (HTGR), fluoride-salt cooled high temperature reactors (FHR), and microreactors (MR). The MOOSE MultiApp system (Gaston, et al., 2015) is also used to perform light water reactor simulations through coupling of CASL's VERA suite (VERA, 2021) with MOOSE-based tools.

MOOSE-based physics codes require an input finite element mesh on which the physics solution is calculated, reported, and transferred to other physics codes. The mesh is a discretized representation of the physical geometry onto which material properties are assigned and on which basis functions for representing the solution live. Creation of the finite element mesh is often one of the most time-consuming stages of input preparation during nuclear reactor analysis, particularly for reactor physics in which the entire core geometry generally needs to be modeled rather than isolated components (single pins, subchannels, or ex-core components). High-fidelity geometry modeling (explicit pins, gap regions, cladding) further requires elaborate tracking of groups of elements for material property assignment and output reporting which can be considerably complex for the user to identify and maintain.

MOOSE users have two mesh input options: (1) supply a prepared mesh file generated from an external (non-MOOSE-based) meshing program like Cubit (CUBIT, 2021), or (2) create a mesh in memory (or supply a mesh) using the MOOSE Mesh System available within the framework itself. Use of an external meshing program allows maximum flexibility in terms of geometry support but is time-consuming for the user, requires a steep learning curve to create even basic geometries, and does not permit the usage of additional mesh metadata beyond basic geometry definitions. Furthermore, external meshes cannot be loaded into memory in parallel unless pre-partitioned and may consequently carry memory burden issues on some computing architectures.

Meshes created using MOOSE's Mesh System do not require the mesh to be written to file, permit extra element-wise information and metadata on the mesh, and utilize simple and intuitive input syntax to reduce user burden. The Mesh System allows the user to start with a simple mesh and perform operations on it (e.g. rotation, block deletion, stitching) in order to build up larger meshes. However, while MOOSE's Mesh System contains many useful MeshGenerator objects, many reactor physics users must leverage external software due to lack of support for certain geometry types. During FY21, the NEAMS program made investments to expand the capability of the existing MOOSE Mesh System to support common advanced reactor geometries as well as facilitate the bookkeeping of element groups for both material assignment and output reporting, greatly speeding the reactor analyst's efforts to create input meshes. These expanded capabilities are described in this report.

2 Task Identification

Priorities were gathered from the MOOSE framework team, Griffin developers, and Bison developers. Immediate meshing needs were identified in the reactor physics community, whereas the fuel performance community has needs that can be met with a longer timeframe. Based on those discussions and ongoing collaborations with staff in the NEAMS Application Drivers Technical Area, a series of meshing-related improvements for the MOOSE framework were identified and prioritized to improve reactor physics workflow during FY21:

1. **Hexagonal geometry meshing capability**
    a. Hexagonal pin cells with optional pin representation
    b. Hexagonal assembly (triangular lattice of pins) with optional duct representation
    c. Hexagonal core (triangular lattice of assemblies)
2. **Control drum geometry meshing capability**
    a. Steady state snapshot
    b. Time-dependent rotation
3. **Reporting ID capability, i.e. implementation of element-wise mesh information for regular Cartesian and hexagonal geometries, to identify specific reactor features/zones**
    a. Pin, assembly, and plane "reporting" ids to facilitate output processing
    b. Depletion ids to facilitate tracking of materials in different zones during the solve and output postprocessing
4. **Core periphery meshing capability**
    a. Ability to mesh the peripheral region between a core and cylindrical exterior boundary via a triangle meshing algorithm
5. **Reactor Geometry Mesh Builder capability (employs reactor analyst input language to build up 3D heterogeneous reactor cores more intuitively, analogous to Argonne's Mesh Tools capability)**
    a. Interfaces for Cartesian and Hexagonal cores
    b. Automatic integration of reporting ids
    c. Support for assigning materials during mesh generation

These capabilities are focused on improving MOOSE-based reactor physics workflow which faces the unique challenges of needing to model very large, detailed geometric domains, but the capabilities can be leveraged for other physics tools as well. All of the developed capabilities are planned to be integrated into the open-source MOOSE framework under the "**reactor**" module (moose/modules/reactor). Per MOOSE requirements, all capabilities include test cases and documentation. Verification of the developed capabilities was performed using Griffin and other physics simulations. It should be noted that the first capability is loosely based on capabilities present in Argonne's Mesh Tools kit (Smith & Shemon, 2015) whose adoption led to huge speed ups in input preparation time for the PROTEUS-SN (Shemon, Smith, & Lee, 2016) and PROTEUS-MOC (Jung, Lee, & Smith, 2018) finite element-based reactor physics codes.

As a concise reference, the following objects were created and/or updated for the MOOSE Framework:

Mesh Generator Objects:

- AssemblyMeshGenerator (NEW)
- AzimuthalBlockIDMeshGenerator (NEW)
- CartesianIDPatternedMeshGenerator (NEW)
- CoreMeshGenerator (NEW)
- DepletionIDGenerator (NEW)
- FancyExtruderGenerator (updated)
- HexagonConcentricCircleAdaptiveMeshGenerator (NEW)
- HexIDPatternedMeshGenerator (NEW)
- PatternedHexMeshGenerator (NEW)
- PinMeshGenerator (NEW)
- PlaneIDGenerator (NEW)
- PolygonConcentricCircleMeshGenerator (NEW)
- SimpleHexagonGenerator (NEW)
- TriangulatedMeshGenerator (NEW)

Functions

- MultiControlDrumFunction (NEW)
- Utilities to Facilitate Element Centroid Calculations (NEW)

3 Hexagonal Geometry Meshing Capability

3.1 Reactor Analysis Motivation

Liquid metal-cooled fast reactors (LMFR) such as sodium-cooled (SFR) and lead-cooled fast reactors (LFR) generally employ hexagonal pins, assemblies, and cores as depicted in Figure 3-1.

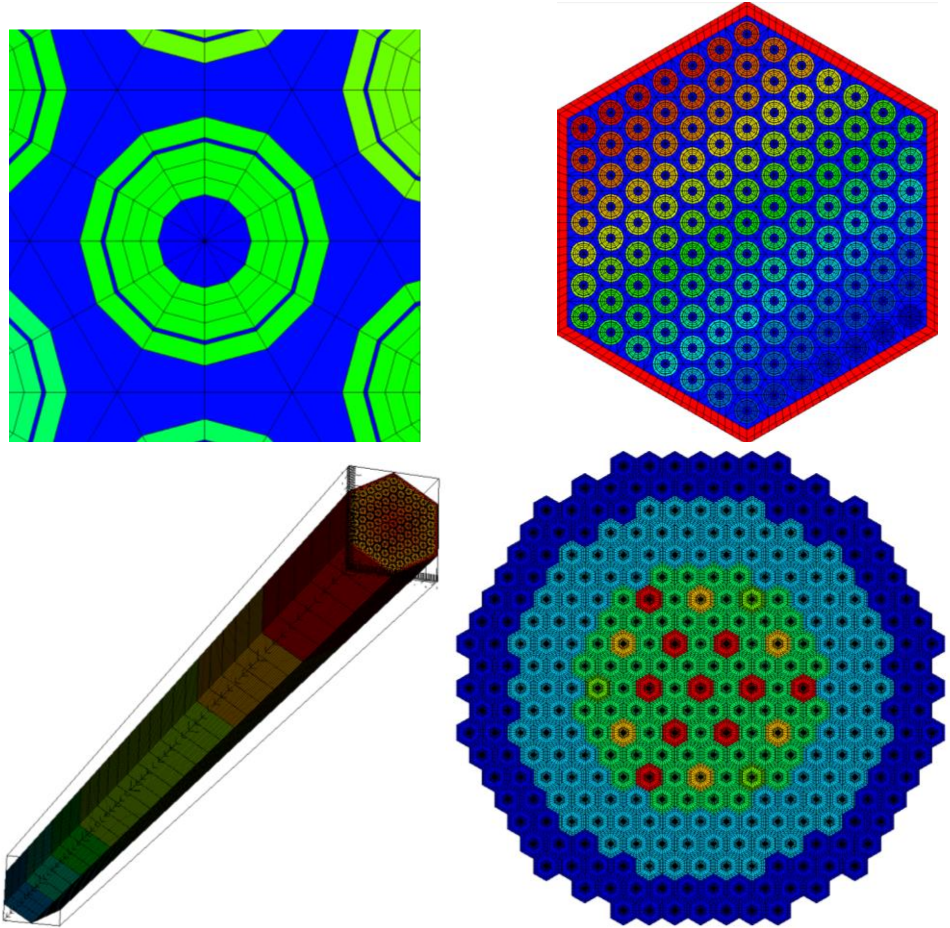


Figure 3-1. Example LMFR geometries: heterogeneous pin cell (top left), heterogeneous ducted assembly (top right), extruded assembly (bottom left), homogeneous assemblies in core.

The fuel or control pins comprise multiple radial regions representing solid or annular fuel, sodium bond, and cladding. The pins are packed into a triangular lattice inside an enclosing hexagonal ducted assembly, which is spaced from other assemblies by an inter-assembly gap containing coolant. The assemblies are patterned in a triangular lattice into a generally hexagonally shaped core, although a few positions on the periphery are typically empty to round off the core boundary. Control assemblies typically contain two ducts: an inner duct enclosing the moving control material, and a stationary outer duct. The ducts are usually separated by liquid metal coolant. Shield and reflector assemblies are typically simpler variations of the fuel assembly geometry.

Within an assembly, all LMFR pins are typically identical, but different assembly types will have different pin sizes. Triangular lattices with hexagonal pins may also occur in other advanced reactor types such as microreactors. LMFR pins, assemblies, and cores can generally be described in 2D and extruded axially (i.e. they do not require modeling of conical regions).

This section describes new capabilities in MOOSE's Mesh System to support the geometries prevalent in LMFRs

3.2 Homogeneous Hexagonal Pin Cells and Assemblies

Homogeneous hexagonal meshes, i.e. hexagonal geometries without explicit pin or duct representation, are typically needed in (1) a nodal-type transport methods, (2) coarse mesh scoping studies, and (3) coarse mesh acceleration methods. Two new mesh generators are available for this purpose: `SimpleHexagonGenerator` and `PolygonConcentricCircleMeshGenerator`. The latter of these is intended for heterogeneous pin cells but may also be used for homogenized pin cells/assemblies.

3.2.1 SimpleHexagonGenerator

`SimpleHexagonGenerator` creates a hexagon subdivided into six equilateral triangles connected at a common (0,0) center node. The resulting mesh represents a homogenized fuel pin or assembly which can be patterned into larger assemblies or cores using the newly developed `PatternedHexMeshGenerator`.

`SimpleHexagonGenerator` has a limited set of input options. The user must specify the hexagon size and style ('apothem' = half-pitch or center-to-flat, or 'radius' = center-to-vertex). The resulting block of elements and external boundary can optionally be renamed and/or renumbered.

```
[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'
   []
[]
```

Figure 3-2. Homogenized pin cell/assembly with basic discretization (SimpleHexagonGenerator)

3.2.2 PolygonConcentricCircleMeshGenerator

`PolygonConcentricCircleMeshGenerator` may be used for homogenized hexagonal cells but has many more options to model explicit pins and ducts. This mesh generator is discussed for heterogenous geometry in Section 3.3 whereas example usage for homogeneous pin cells follows.

This mesh generator is generalized to mesh regular polygons (with 3 or more sides). Therefore, the number of polygon sides should be specified (num_sides). The default num_sides is 6 (hexagon).

The user may specify the number of azimuthal subdivisions per polygon side (num_sectors_per_side) uniquely for each hexagon side. Each entry in this array must be an even integer (minimum value is 2). The coarsest mesh possible is shown in Figure 3-3 which has 12 triangular elements rather than the 6 yielded by `SimpleHexagonGenerator`. Figure 3-4 shows the coarsest all-quad mesh possible. Additional radial subdivisions are possible by specifying background_intervals>1 (Figure 3-5, Figure 3-6).

Important note: If background_intervals>1 (with no rings defined as in Figure 3-5 and Figure 3-6), the background region will be defined as 2 separate blocks (see Table).

The separate block at the center of the meshes in Figure 3-5 and Figure 3-6 is required to accommodate either triangular or quadrilateral elements in the center zone (quad elements are enabled by quad_center_elements = true). Setting background_intervals to 3 or larger will not add additional background blocks (capped at 2). Figure 3-5 and Figure 3-6 demonstrate the additional block_ids and block_names required to name the extra block at the center, compared to Figure 3-3 and Figure 3-4.

```
[Mesh]
   [pccsimplehex1BG]
      type = PolygonConcentricCircleMeshGenerator
      num_sides = 6   # 6 for hexagon is default
      polygon_size = 0.146
      polygon_size_style = 'apothem'
      num_sectors_per_side = '2 2 2 2 2 2' # even; min 2
      # optional
      background_intervals = 1
      background_block_ids = '40'
      background_block_names = 'Fuel'
   []
[]
```

Figure 3-3. Homogenized pin cell/assembly with all tri mesh (PolygonConcentricCircleMeshGenerator)

```
[Mesh]
   [pccsimplehex1BGquad]
      type = PolygonConcentricCircleMeshGenerator
      num_sides = 6   # 6 for hexagon is default
      polygon_size = 0.146
      polygon_size_style = 'apothem'
      num_sectors_per_side = '2 2 2 2 2 2' # even; min 2
      # optional
      background_intervals = 1
      background_block_ids = '40'
      background_block_names = 'Fuel'
      quad_center_elements = true
   []
[]
```

Figure 3-4. Homogenized pin cell/assembly with all quad mesh (PolygonConcentricCircleMeshGenerator)

```
[Mesh]
  [pccsimplehex2BG]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6   # 6 for hexagon is default
    polygon_size = 0.146
    polygon_size_style = 'apothem'
    num_sectors_per_side = '2 2 2 2 2 2' # even; min 2
    # optional
    background_intervals = 2
    background_block_ids = '40 41'
    background_block_names = 'Fuel1 Fuel2'
  []
[]
```

Figure 3-5. Homogenized pin cell/assembly showing extra block required due to >1 background intervals, mixed tri-quad mesh (PolygonConcentricCircleMeshGenerator)

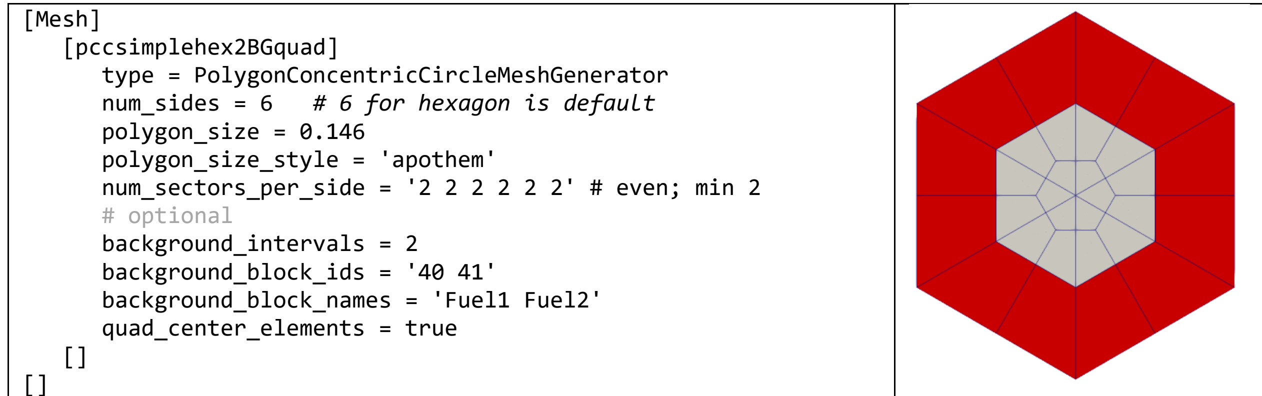


Figure 3-6. Homogenized pin cell/assembly showing extra block required due to >1 background intervals, all quad mesh (PolygonConcentricCircleMeshGenerator)

`PolygonConcentricCircleMeshGenerator` has many additional input options to control block and boundary names and ids, as well as the spacing of nodes on each edge.

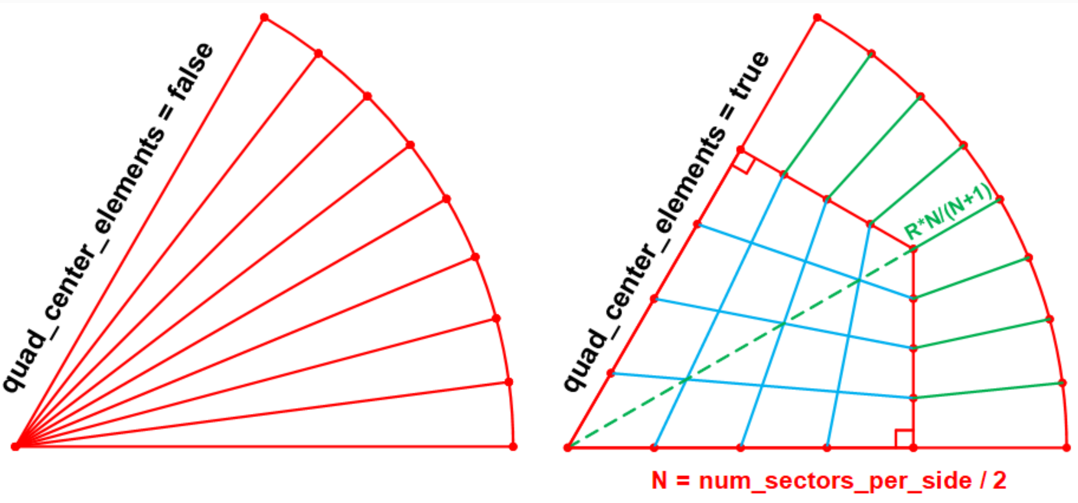


Figure 3-7. Different node algorithms for Tri and Quad central elements

3.3 Heterogeneous Hexagonal Pin Cells

Creation of the heterogeneous hexagonal pin cells commonly found in LMFR and other reactor geometries is performed with the new mesh generator object `PolygonConcentricCircleMeshGenerator`, which was modeled after the existing `ConcentricCircleMeshGenerator` object (the latter creates square pin cells typically used in LWR lattices). The new object is referred to with the 'Polygon' prefix instead of 'Hexagon' because the algorithm is generalized to mesh regular 2D polygons with 3 or more sides and is therefore not limited to hexagons. The Cartesian option (num_sides = 4) includes the functionality of `ConcentricCircleMeshGenerator` along with additional capabilities although it outputs a mesh rotated 45 degrees (vertex up, like a diamond) compared to `ConcentricCircleMeshGenerator`.

We limit our discussion here to hexagonal pin cells (num_sides = 6) which are relevant for advanced reactors.

3.3.1 PolygonConcentricCircleMeshGenerator

`PolygonConcentricCircleMeshGenerator` creates 2D meshes for concentric circles inside a polygon enclosure with or without duct features. Usage of this mesh generator for homogenized pin cells (no pins or ducts) was discussed in the previous section. Doing so requires an extra background block to be defined if background_intervals is larger than 1.

The input options for `PolygonConcentricCircleMeshGenerator` can be categorized into 4 major groups:

- General polygon options (number of sides, size, type of elements to use in center, azimuthal discretization, whether to uniformly space boundary nodes)
- Ring parameters
- Background parameters (region between rings and ducts)
- Duct parameters

The outer radius of each ring is given in the vector ring_radii. Meshing subintervals for the respective rings are given in the vector ring_intervals. The optional ring_block_ids and ring_block_names are used to renumber the block ids and block names to user-defined values. Ring volumes are automatically preserved through the use of preserve_volumes=on. The centermost elements can be either triangle (default) or quad (use_quad_center_elements = on).

The background region is the region enclosed between the outermost ring and the innermost duct region – it is the coolant region for LMFR geometries.

The inner size of each duct is given in the vector duct_sizes, where duct_size_style (apothem or radius) describes the distance being specified. Apothem is simply the ½ pitch, i.e. half the flat-to-flat distance or the center-to-side distance. The term apothem is used because it is applicable to polygons with odd numbers of sides.

A typical hexagon geometry is shown here which include the reference side index numbering.

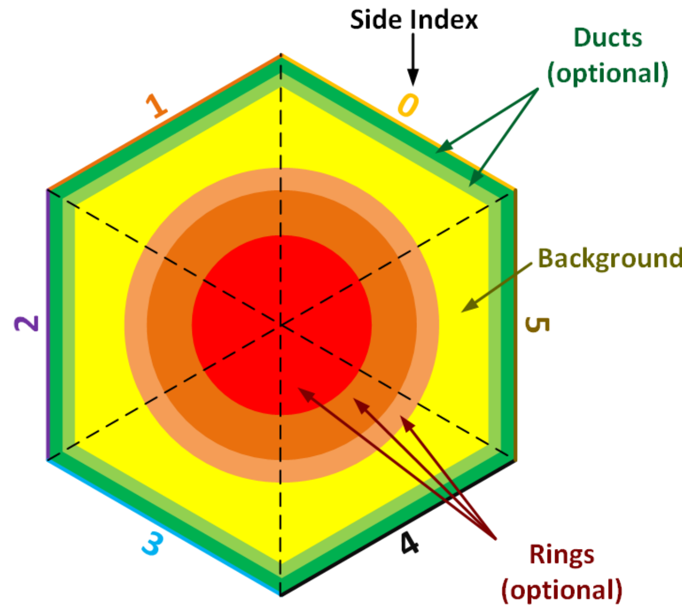


Figure 3-8. A schematic drawing showing the different regions generated by PolygonConcentricCircleMeshGenerator.

In general, concentric blocks of elements are created for each ring, the background region, and each duct. The default block numbering starts at 0 (centermost zone) and increases by 1 with increasing radius of the block. Generally the number of blocks in the resulting mesh is equal to the number of rings + number of ducts + 1.

However, in some cases one extra block is added in the center zone (either 1st ring or background zone depending on whether rings were defined). This extra block is required so that the code can accommodate either triangular or quadrilateral elements for maximum flexibility to the user, while still maintaining as much symmetry as possible.

The extra block appears in two scenarios:

- At least one ring is present, and the first ring has more than 1 subinterval (*ring_intervals*(0)>1), or
- No rings are present, and the background region has more than 1 subinterval (*background_intervals*>1)

The presence of this extra block affects the automated block numbering (center geometrical zone includes subdomains 0 and 1, instead of just 0), as well as the user-defined block numbering and naming input entries which require one extra entry. If the user receives errors regarding size of *background_block_id/names or ring_block_ids/names,* they should first check whether one of the two situations above applies. If so, an extra entry is required to number/name the extra block in the innermost region. The specified block id/name for the extra block must be *different* than the original block. Table 3-1Table summarizes the array sizes required in the 4 possible different situations.

Table 3-1. Guidelines for when an extra block is added to the center zone in PolygonConcentricCircleMeshGenerator

Are Rings Defined?	Center Zone Meshing Intervals	Center Zone Description	Array Sizes
No	*background_intervals* = 1	Center zone comprises 1 block.	*Size(background_block_ids) = size (background_block_names) = 1*
	background_intervals > 1	Center zone comprises 2 blocks.	*Size(background_block_ids) = size (background_block_names) = 2*
Yes	*ring_intervals(0) = 1*	Center zone comprises 1 block.	Size(*ring_block_ids) =* size(*ring_block_names) = size (ring_intervals*)
	ring_intervals(0) > 1	Center zone comprises 2 blocks.	Size(*ring_block_ids) =* size(*ring_block_names) = size (ring_intervals*) + 1

Figure 3-9 and Figure 3-10 show sample inputs for generating pin cell meshes with duct regions. To remove the duct regions, simply omit the duct_* options. Similarly, to remove the ring regions, simply omit the ring_* options.

Figure 3-11 demonstrates the assignment of different azimuthal discretizations on each face. The number of azimuthal sectors per side (num_sectors_per_side) is a vector of even integers of size (num_sides). Volume preservation of the rings is performed with preserve_volumes = on.

If different num_sectors_per_side are invoked on different sides of the polygon, then the user must utilize the triangular mesh option for the innermost ring of the pin cell (quad_center_element = false).

```
[Mesh]
   [HetPinCell]
      type = PolygonConcentricCircleMeshGenerator
      num_sides = 6   # 6 for hexagon is default
      num_sectors_per_side = ‘4 4 4 4 4 4’
      polygon_size = 0.05
      polygon_size_style = ‘apothem’
      uniform_mesh_on_sides = true

      ring_radii     =    ‘0.01    0.02 0.025’
      ring_intervals =    ‘1       2    1’
      ring_block_ids =    ‘10      15   20’
      ring_block_names =  ‘hole    fuel cladding’
      preserve_volumes = on
      quad_center_elements = true  #false

      background_intervals = 2
      background_block_ids = ‘40’
      background_block_names = ‘coolant’

      duct_sizes_style = ‘apothem’
      duct_sizes     = ‘0.044 0.0475’
      duct_intervals = ‘1     1’
      duct_block_ids = ‘100   110’
      duct_block_names = ‘duct gap’
   []
[]
```

quad_center_elements = true

quad_center_elements = false

Figure 3-9. Heterogeneous hexagonal cell with pin and duct regions

```
[Mesh]
   [HetPinCell]
      type = PolygonConcentricCircleMeshGenerator
      num_sides = 6   # 6 for hexagon is default
      num_sectors_per_side = '4 4 4 4 4 4'
      polygon_size = 0.05
      polygon_size_style = 'apothem'
      uniform_mesh_on_sides = true

      ring_radii     =   '0.01    0.02 0.025'
      ring_intervals =   '2       2    1'
      ring_block_ids =   '10 11   15   20'
      ring_block_names = 'hole1 hole2 fuel clad'
      preserve_volumes = on
      quad_center_elements = true #false

      background_intervals = 2
      background_block_ids = '40'
      background_block_names = 'coolant'

      duct_sizes_style = 'apothem'
      duct_sizes     = '0.044 0.0475'
      duct_intervals = '1     1'
      duct_block_ids = '100   110'
      duct_block_names = 'duct gap'
   []
[]
```

quad_center_elements = true

quad_center_elements = false

Figure 3-10. Heterogeneous hexagonal cell with pin and duct regions, showing extra block required due to >1 subinterval in center zone

```
[Mesh]
   [HetPinCell]
      type = PolygonConcentricCircleMeshGenerator
      num_sides = 6    # 6 for hexagon is default
      num_sectors_per_side = '4 6 8 8 6 2'
      polygon_size = 0.05
      polygon_size_style = 'apothem'
      uniform_mesh_on_sides = true

      ring_radii     =   '0.02       0.025'
      ring_intervals =   '3          1'
      ring_block_ids =   '15    16   20'
      ring_block_names = 'fuel1 fuel2 clad'
      preserve_volumes = on
      quad_center_elements = false

      background_intervals = 2
      background_block_ids = '40'
      background_block_names = 'coolant'

      duct_sizes_style = 'apothem'
      duct_sizes     = '0.044 0.0475'
      duct_intervals = '1     1'
      duct_block_ids = '100   110'
      duct_block_names = 'duct gap'
   []
[]
```

Figure 3-11. Heterogeneous hexagonal cell with pin and duct regions, showing varied azimuthal discretization per face and preservation of ring volumes

The typical uses of this mesh generator object are therefore to create:

- Homogeneous pin cell or assembly
- Heterogeneous pin cell (rings present)
- Partially heterogeneous assembly (ducts present)
- The initial mesh for a heterogeneous control drum (both rings and duct present), which will be modified to create the absorber arc by another mesh generator.

3.3.2 HexagonConcentricCircleAdaptiveBoundaryMeshGenerator

In some cases, the user may want to develop a hexagonal pin cell or partially homogenized assembly mesh whose nodes will match other already developed meshes. `HexagonConcentricCircleAdaptiveBoundaryMeshGenerator` was developed exactly for this purpose. As its name suggests, this mesh generator creates hexagons rather than general polygons. It therefore has the functionality of `PolygonConcentricCircleMeshGenerator` (with num_sides=6), plus an additional capability to customize the azimuthal discretization to match specified neighbor meshes. An example of this approach is shown in Figure 3-12.

```
[Mesh]
  [fmg]
    type = FileMeshGenerator
```

```
    file = hex_in.e
  []
  [gen]
    type = HexagonConcentricCircleAdaptiveBoundaryMeshGenerator
    num_sides = 6
    num_sectors_per_side = '4 4 4 4 4 4'
    background_intervals = 2
    hexagon_size = 5.0
    sides_to_adapt = 0
    inputs = 'fmg'
  []
[]
```

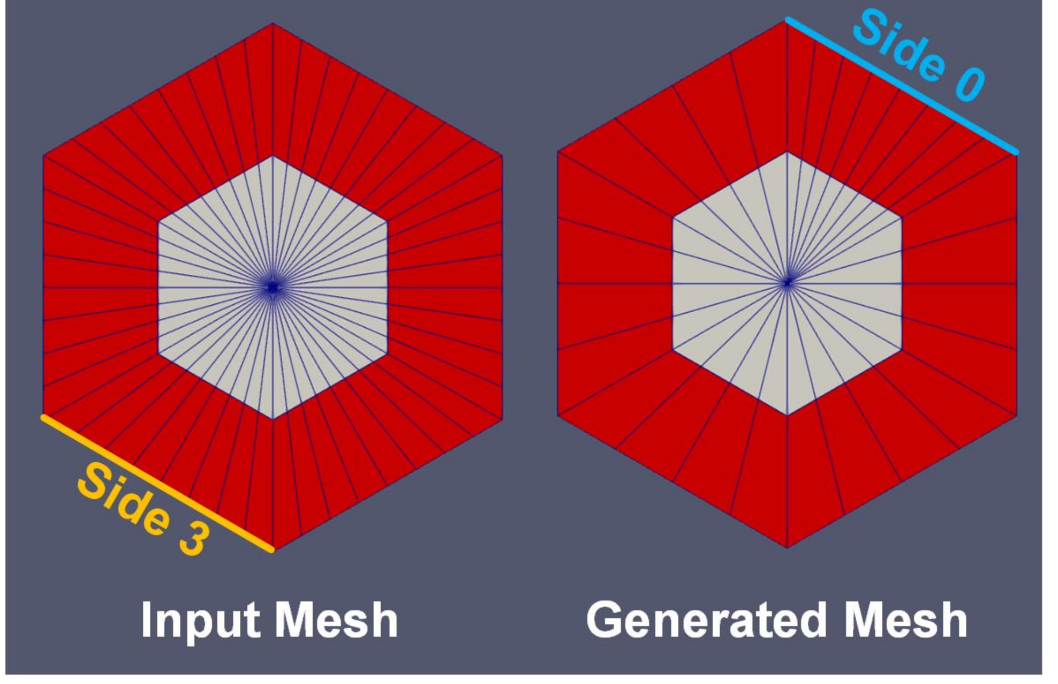


Figure 3-12. A mesh generated by `HexagonConcentricCircleAdaptiveBoundaryMeshGenerator`. Note that the side 0 of the hexagonal mesh adaptively matches side 3 of the input hexagonal mesh.

3.3.3 IDs and Names of Blocks and Boundaries

MOOSE inherits libMesh's ID and Name systems of blocks (subdomains) and boundaries (sidesets/nodesets). Each subdomain, sideset, or nodeset intrinsically have an ID, which is an integer. There is also an option to assign `std::string` type name for each subdomain, sideset, or nodeset. This name assignment is achieved through a std::map type data containing IDs as keys and corresponding names as values.

For both the `PolygonConcentricCircleMeshGenerator` and `HexagonConcentricCircleAdaptiveBoundaryMeshGenerator` objects, the blocks are sequentially numbered from the innermost block to the outmost block by default. By default, the blocks are named by the corresponding block ids. Optionally, for both mesh generators, the users can assign customized block ids and names by setting the following input parameters:

ring_block_ids, ring_block_names, background_block_ids, background_block_names, duct_block_ids, and *duct_block_names*.

The external boundary has an ID of 10,000 and is unnamed by default. The users can assign the customized ID and name for the external boundary by setting input parameter *external_boundary_id* and *external_boundary_name.*

3.4 Hexagonal Assemblies and Cores

This section discusses how to build up a series of hexagonal meshes into a grid pattern to create an assembly or core.

3.4.1 PatternedHexMeshGenerator

`PatternedHexMeshGenerator` assembles multiple 2D hexagonal meshes into a hexagonal grid pattern and optionally adds additional background region and ducts around the periphery of the grid. This mesh generator can be used to mesh a hexagonal assembly containing different pin types or explicit pins, or a reactor core comprising several assemblies.

`PatternedHexMeshGenerator` is the hexagon equivalent to `PatternedMeshGenerator` which stitches together Cartesian geometries. However, `PatternedHexMeshGenerator` includes additional options to treat the grid boundary which is necessary to support hexagonal reactor geometries as mentioned above.

Like `PatternedMeshGenerator`, the user specifies input meshes via the *inputs* parameter, and places these input meshes into a hexagonal grid in the *pattern* array. The *pattern* array must be a perfect hexagonal grid – no missing entries are permitted. Dummy meshes must be used to fill in empty slots and deleted later if the desired pattern is not a perfect hexagon.

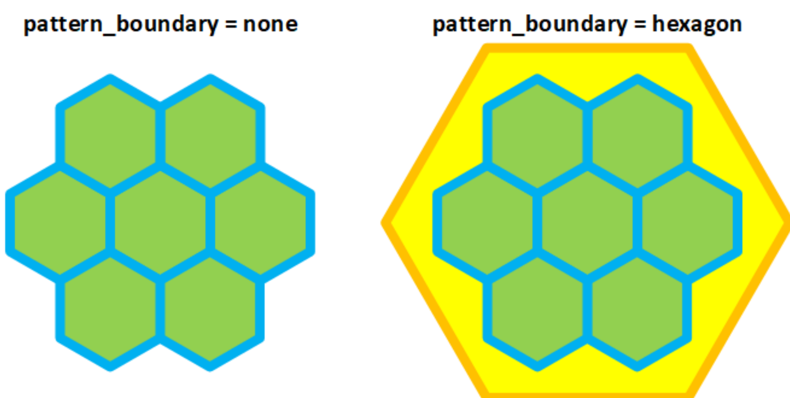


Figure 3-13. A schematic drawing showing the difference between "none" and "hexagon" *pattern_boundary* options.

Finally, the user specifies the boundary around the pattern using *pattern_boundary* (see Figure 3-13). Valid options are *hexagon* or *none.* If *pattern_boundary = none*, the input meshes will be stitched together and the resulting mesh will have a zig-zag boundary (e.g. reactor core). If *pattern_boundary = hexagon,* then an extra layer of background material will be stitched around

the pattern according to *hexagon_size* so that the outer boundary of the pattern is a hexagon rather than a zig-zag boundary (e.g. assembly). Optional duct regions may also be added via *duct_sizes* to specify the inner duct boundaries and *duct_intervals* to describe the meshing resolution.

Note that the constituent input meshes for this mesh generator may be created from `SimpleHexagonGenerator`, `PolygonConcentricCircleMeshGenerator`, `HexagonConcentricCircleAdaptiveMeshGenerator`, and `PatternedHexMeshGenerator` itself (see Figure 3-14 through Figure 3-16 for more details).

```
[Mesh]
  [hex_1]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '4 4 4 4 4 4'
    background_intervals = 2
    ring_radii = 4.0
    ring_intervals = 2
    ring_block_ids = '10 15'
    ring_block_names = 'center_tri center'
    background_block_ids = 20
    background_block_names = background
    polygon_size = 5.0
    preserve_volumes = on
  []
  [pattern]
    type = PatternedHexMeshGenerator
    inputs = 'hex_1'
    pattern = '0 0;
              0 0 0;
               0 0'
    pattern_boundary = hexagon
    background_intervals = 2
    background_block_id = 25
    background_block_name= "assem_block"
    hexagon_size = 18
  []
[]
```

Figure 3-14. A patterned hexagonal mesh based on unit mesh generated by `PolygonConcentricCircleMeshGenerator` with “hexagon” boundary

```
[Mesh]
  [hex_1]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '4 4 4 4 4 4'
    background_intervals = 2
    ring_radii = 4.0
    ring_intervals = 2
    ring_block_ids = '10 15'
    ring_block_names = 'center_tri center'
    background_block_ids = 20
    background_block_names = background
    polygon_size = 5.0
    preserve_volumes = on
  []
  [pattern]
    type = PatternedHexMeshGenerator
    inputs = 'hex_1'
    pattern = '0 0;
              0 0 0;
               0 0'
    pattern_boundary = none
  []
[]
```

Figure 3-15. A patterned hexagonal mesh based on unit mesh generated by `PolygonConcentricCircleMeshGenerator` with "none" boundary

```
[Mesh]
  [hex_1]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '4 4 4 4 4 4'
    background_intervals = 2
    ring_radii = 4.0
    ring_intervals = 2
    ring_block_ids = '10 15'
    ring_block_names = 'center_tri center'
    background_block_ids = 20
    background_block_names = background
    polygon_size = 5.0
    preserve_volumes = on
  []
  [pattern_1]
    type = PatternedHexMeshGenerator
    inputs = 'hex_1'
    pattern = '0 0;
              0 0 0;
               0 0'
    hexagon_size = 15
    background_block_id = 80
    background_block_name = hex_background
  []
  [pattern_2]
    type = PatternedHexMeshGenerator
    inputs = 'pattern_1'
    pattern_boundary = none
    generate_core_metadata = true
    pattern = '0 0;
              0 0 0;
               0 0'
  []
[]
```

Figure 3-16. A patterned hexagonal mesh based on unit mesh generated by `PatternedHexMeshGenerator` with hexagonal boundary

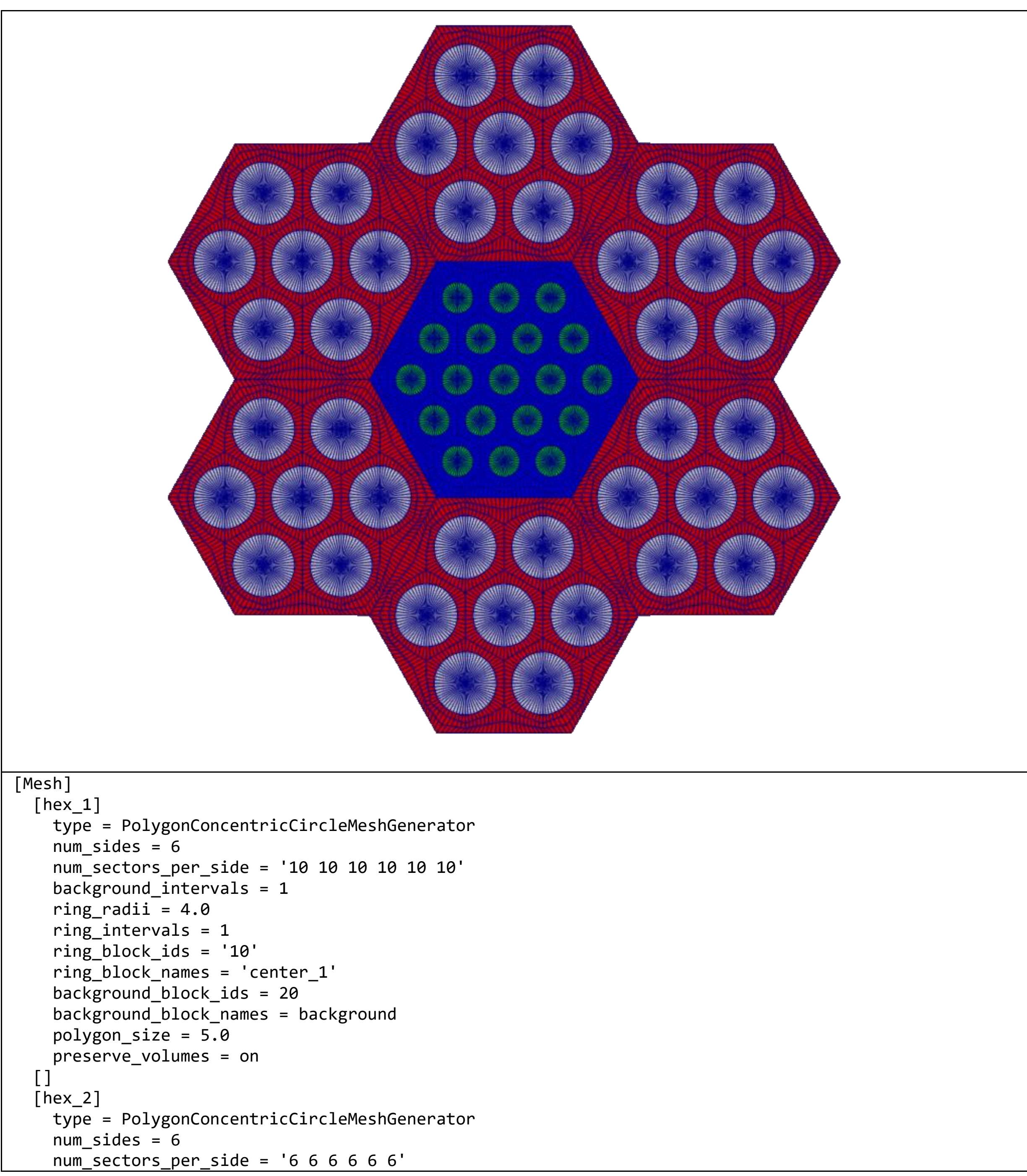
The block ids and names of the input meshes are inherited by `PatternedHexMeshGenerator`. When a mesh with *pattern_boundary = hexagon* is generated, the pattern background block is unnamed with an ID of 1,000 by default. If there are any duct layers, the duct blocks are unnamed with their IDs sequentially numbered starting from 1,001. The users can assign customized IDs and names for background blocks and duct blocks by setting input parameters: *background_block_id, background_block_name, duct_block_ids, and duct_block_names*.

The external boundary has an ID of 10,000 and is unnamed by default. The users can assign the customized ID and name for the external boundary by setting input parameter *external_boundary_id* and *external_boundary_name*.

When the users need to use `PatternedHexMeshGenerator` to recursively combine hexagonal assembly meshes, the Boolean type input parameter *generate_core_metadata = true*. In that case, the appropriate assembly `MeshMetaData` can be utilized for core mesh generation; and a series of reactor core `MeshMetaData` are generated for other MOOSE `Reactor` module objects such as `MultiControlDrumFunction`. To be clear, *generate_core_metadata = true* is required when combining assemblies into a core map, which requires the use of `PatternedHexMeshGenerator` to pattern inputs created from `PatternedHexMeshGenerator`.

3.5 Current Limitations

The adaptive mesh generator only works for single-pin unit mesh now. For assembly unit meshes, if the meshes contain a different number of pins, some special care needs to be taken by the user to utilize least common multipliers to ensure stitchability. An example of such an approach is illustrated in Figure 3-17. When unit assembly meshes with many different numbers of pins are involved, it is impractical to find a reasonably small least common multiplier. Therefore, a more practical and general solution needs to be developed to better handle such scenarios.



```
[Mesh]
  [hex_1]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '10 10 10 10 10 10'
    background_intervals = 1
    ring_radii = 4.0
    ring_intervals = 1
    ring_block_ids = '10'
    ring_block_names = 'center_1'
    background_block_ids = 20
    background_block_names = background
    polygon_size = 5.0
    preserve_volumes = on
  []
  [hex_2]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '6 6 6 6 6 6'
```

```
    background_intervals = 1
    ring_radii = 2.0
    ring_intervals = 1
    ring_block_ids = '110'
    ring_block_names = 'center_2'
    background_block_ids = 120
    background_block_names = background
    polygon_size = 3.0
    preserve_volumes = on
  []
  [pattern_1]
    type = PatternedHexMeshGenerator
    inputs = 'hex_1'
    pattern = '0 0;
              0 0 0;
               0 0'
    hexagon_size = 15
    background_block_id = 20
    background_block_name = hex_background_1
    background_intervals = 2
    uniform_mesh_on_sides = true
  []
  [pattern_2]
    type = PatternedHexMeshGenerator
    inputs = 'hex_2'
    pattern = '0 0 0;
              0 0 0 0;
             0 0 0 0 0;
              0 0 0 0;
               0 0 0'
    hexagon_size = 15
    background_block_id = 120
    background_block_name = hex_background_2
    background_intervals = 2
    uniform_mesh_on_sides = true
  []
  [pattern_3]
    type = PatternedHexMeshGenerator
    inputs = 'pattern_1 pattern_2'
    pattern_boundary = none
    generate_core_metadata = true
    pattern = '0 0;
              0 1 0;
               0 0'
  []
[]
```

Figure 3-17. A patterned hexagonal mesh with unit assembly meshes containing 7 and 19 pins.

4 Control Drum Geometry Meshing Capability

4.1 Reactor Analysis Motivation

Rotating control drums are utilized in microreactor and other core designs for reactivity control during normal operation. Rather than control rods which are inserted and withdrawn axially, control drums can be rotated to position the control material closer or further away from the core center. The control drum contains a ring of material of which only a partial arc contains absorber material. When the control material is closest to the core center, the reactivity feedback is most negative and this is similar to the "control rods inserted" state. A typical control drum geometry is shown in Figure 4-1.

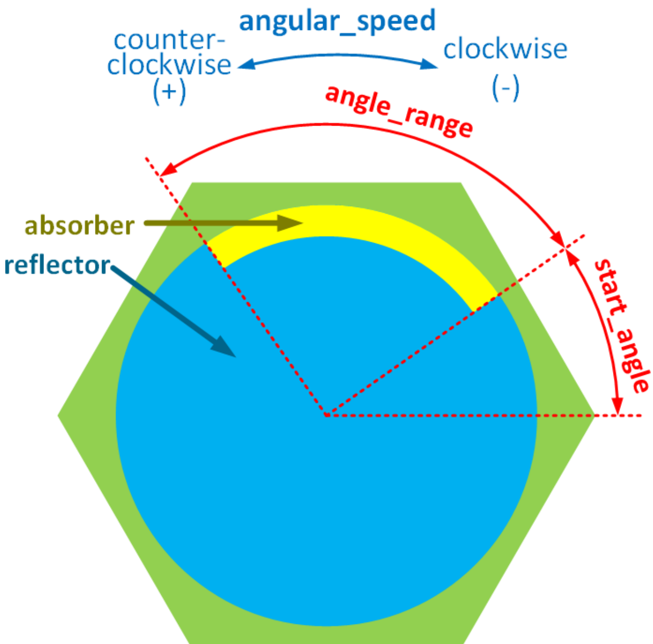


Figure 4-1. A typical control drum structure and important geometrical parameters.

At steady state, the control drum will be in a static position. However, during rotation, the absorber arc is moving and consequently the material-to-geometry mapping is changing with time. Both of these cases are handled with new MOOSE framework mesh generators and functions so that the user can easily model a steady state position or control drum rotation. Currently, the control drum geometry must be contained within a single outer hexagon boundary.

4.2 Steady State Control Drum Position

4.2.1 AzimuthalBlockIDMeshGenerator

To mesh the geometry for a control drum at a single position, the newly developed `AzimuthalBlockIDMeshGenerator` object may be leveraged. This object modifies a hexagonal pin cell mesh generated by either `PolygonConcentricCircleMeshGenerator` or

`HexagonConcentricCircleAdaptiveBoundaryMeshGenerator` by creating new blocks of elements within a user-defined azimuthal sector, given by *start_angle* and *angle_range* in degrees. The user may select multiple radial blocks by *old_block_ids* or *old_block_names* within that sector to modify simultaneously. Azimuthal node positions are moved to exact positions, and new blocks are created within the azimuthal sector for the user-selected radial blocks.

Prior to moving node positions, the algorithm finds the nodes that have azimuthal angles closest to the given azimuthal range and moves those to the exact azimuthal positions. Hexagon corner nodes may not be moved, and the code in that case moves the next nearest nodes. As moving nodes in the azimuthal direction changes the volumes (areas) of the circular blocks, the volume preservation radius correction is made if *preserve_volumes* is set as true.

If the external block (i.e., the block that contains the external boundary of the mesh) is not selected to be modified, the nodes on the external boundary are not altered by this object, which facilitates mesh stitching since the outer boundary will remain unmodified. On the other hand, if the external block is selected, the nodes on the external boundary are moved as well. Figure 4-2 depicts the movement or non-movement of external boundary nodes (orange region) based on whether that external block was selected to be modified. Figure 4-3 shows input syntax examples.

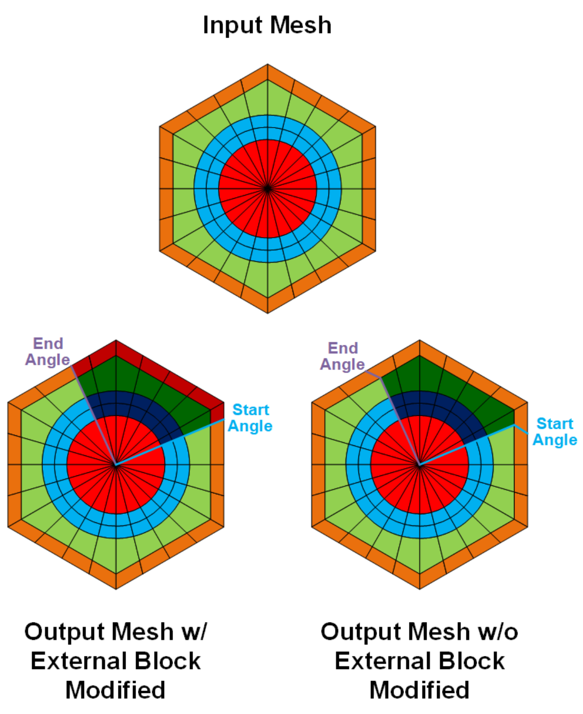


Figure 4-2. A schematic drawing showing the functionalities of this `AzimuthalBlockIDMeshGenerator` object.

```
[Mesh]
  [cd]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '4 4 4 8 4 4'
    background_intervals = 1
    ring_radii = '4.2 4.8'
    ring_intervals = '2 1'
    ring_block_ids = '10 15 20'
    ring_block_names = 'center_tri center cd_ring'
    background_block_ids = 30
    background_block_names = background
    polygon_size = 5.0
    preserve_volumes = true
  []
  [./cd_azi_define]
    type = AzimuthalBlockIDMeshGenerator
    input = cd
    start_angle = 280
    angle_range = 100
    old_block_ids = '10 15 20'
    new_block_ids = '100 150 200'
    new_block_names = 'ctr_tri_new ctr_new cd_ring_new'
    preserve_volumes = true
  []
[]
```

```
[Mesh]
  [cd]
    type = PolygonConcentricCircleMeshGenerator
    num_sides = 6
    num_sectors_per_side = '4 4 4 8 4 4'
    background_intervals = 1
    ring_radii = '4.2 4.8'
    ring_intervals = '2 1'
    ring_block_ids = '10 15 20'
    ring_block_names = 'center_tri center cd_ring'
    background_block_ids = 30
    background_block_names = background
    polygon_size = 5.0
    preserve_volumes = true
  []
  [./cd_azi_define]
    type = AzimuthalBlockIDMeshGenerator
    input = cd
    start_angle = 280
    angle_range = 100
    old_block_ids = '10 15 30'
    new_block_ids = '100 150 200'
    new_block_names = 'ctr_tri_new ctr_new cd_ring_new'
    preserve_volumes = true
  []
[]
```

Figure 4-3. Polygon meshes modified by `AzimuthalBlockIDMeshGenerator` without and with external boundary nodes moved.

4.3 Time-Dependent Control Drum Rotation

In order to simulate the dynamic behavior during control drums rotation during power transients, a MOOSE Functions object `MultiControlDrumFunction` was developed to quantify the real-time volume fraction of absorber/reflector materials in each control drum element. Specifically, the `MultiControlDrumFunction` (1) assesses multiple control drum absorber positions in the core based on metadata from `PatternedHexMeshGenerator`, (2) computes the position of each control drum absorber arc at a given time, and (3) returns a value for each mesh element representing the absorber volume fraction in that element (ranging from 0 to 100 where 0 is no absorber and 100 is pure absorber).

4.3.1 Use of PatternedHexMeshGenerator MeshMetaData

The `MultiControlDrumFunction` object relies on a series of `MeshMetaData` generated by `PatternedHexMeshGenerator` objects that had specified *generate_core_metadata = true*. The metadata includes information about the control drums located in the core:

- `control_drum_positions`: a vector of control drum center positions (i.e., x and y coordinates). This *MeshMetaData* can also be outputted as an ASCII file by setting `generate_control_drum_positions_file` as true and providing `position_file`.

- `control_drum_angles`: a vector of the azimuthal angle (in degrees) of the control drum center position to the center of the core.

- `control_drums_azimuthal_meta`: a two-dimensional vector containing the sorted azimuthal angles of nodes of each individual control drum.

Figure 4-4 depicts the ordering rule for control drum positions; they are ordered from 1 to N based on azimuthal angle from the core center. In addition, `assign_control_drum_id` can be set as true so that the control drum `inputs` meshes can be indexed using an element extra integer called `control_drum_id`. As illustrated, the `control_drum_id` is indexed based on the azimuthal angles of the control drums.

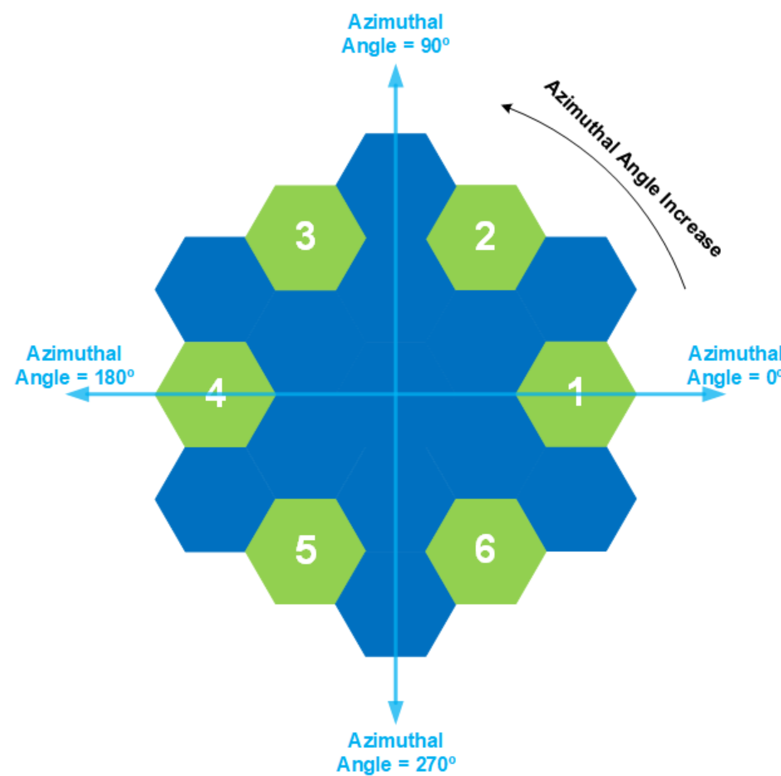


Figure 4-4. A schematic drawing the indexing rule of `control_drum_id` in the `PatternedHexMeshGenerator` object.

4.3.2 MultiControlDrumFunction Rotation of Control Drums

`MultiControlDrumFunction` is a MOOSE Function object that assigns values for the absorber volume fraction in each element of an identified control drum. This object is capable of handling multiple control drums within a single mesh. Referring back to Figure 4-1, a control drum typically has a cylindrical geometry with an outer ring containing both absorber (yellow) and reflector (blue) sections. Using the `MultiControlDrumFunction` object, the rotation of the absorber section can be simulated. The entire ring needs to be contained within a single block. The Function object creates a time- and space-dependent function to represent the volume percentage of the absorber in this ring block. The function is intended to work with either `FunctionAux` to assign values to an elemental auxiliary variable, or `GenericFunctionMaterial` to assign values to a material property. There are three possible scenarios at a given time point:

- If the entire mesh element is within the absorber section, the function value is 100 (percent absorber);
- If the entire mesh element is within the reflector section, the function value is 0 (percent absorber);

- If the mesh element is intercepted by the actual absorber-reflector boundary, the function value is between 0 and 100 and equal to the volume percentage of the absorber part in that mesh element.

To simulate the rotation of the control drums, a set of parameters are needed for each control drum (Table 4-1).

Table 4-1. Geometry parameters needed for MultiControlDrumFunction

start_angle	the azimuthal angle of the starting position of the absorber section at the beginning of the time.
angle_range	the azimuthal angle range of the absorber section.
angular_speed	the rotation speed of the control drum in degree per second (positive values mean counterclockwise rotation; negative values mean clockwise rotation.)

When there are multiple control drums in the reactor core mesh to be simulated, it is important to determine the mesh domain of each control drum in order to utilize the aforementioned algorithm to assign the function values for all the involved control drums. Two options are available for this procedure. When `use_control_drum_id` is true, this `MultiControlDrumFunction` can use the element extra integer `control_drum_id` in the mesh to determine the domain of each control drum. Please refer to Section 4.3.1 for details how `control_drum_id` is assigned. If `use_control_drum_id` is false, the domain of each control drum is determined based on the nearest control drum position. To be specific, for each element in the core mesh, the distances between the element centroid and all the involved control drum positions are calculated. The element belongs to the domain of the control drum that is nearest to the element centroid.

Figure 4-5 (top left) depicts a core mesh containing multiple control drums as colored in brown with pink boundary representing the ring along which the absorber rotates. The top right figure depicts the control drum id (an extra element integer ranging from 1 to 12) as assigned by `PatternedHexMeshGenerator` based on position relative to the core center (i.e., `use_control_drum_id = true`). The bottom left figure shows the control drum domains determined by `MultiControlDrumFunction` based on nearest point algorithm and control drum positions (i.e., `use_control_drum_id = false`). Note that the two control drum domain determination approaches lead to consistent results. The bottom right figure shows the returned absorber fraction (ranging from 0 to 100) in each element of each control drum at a given point in time. Red depicts pure absorber, and blue depicts no absorber.

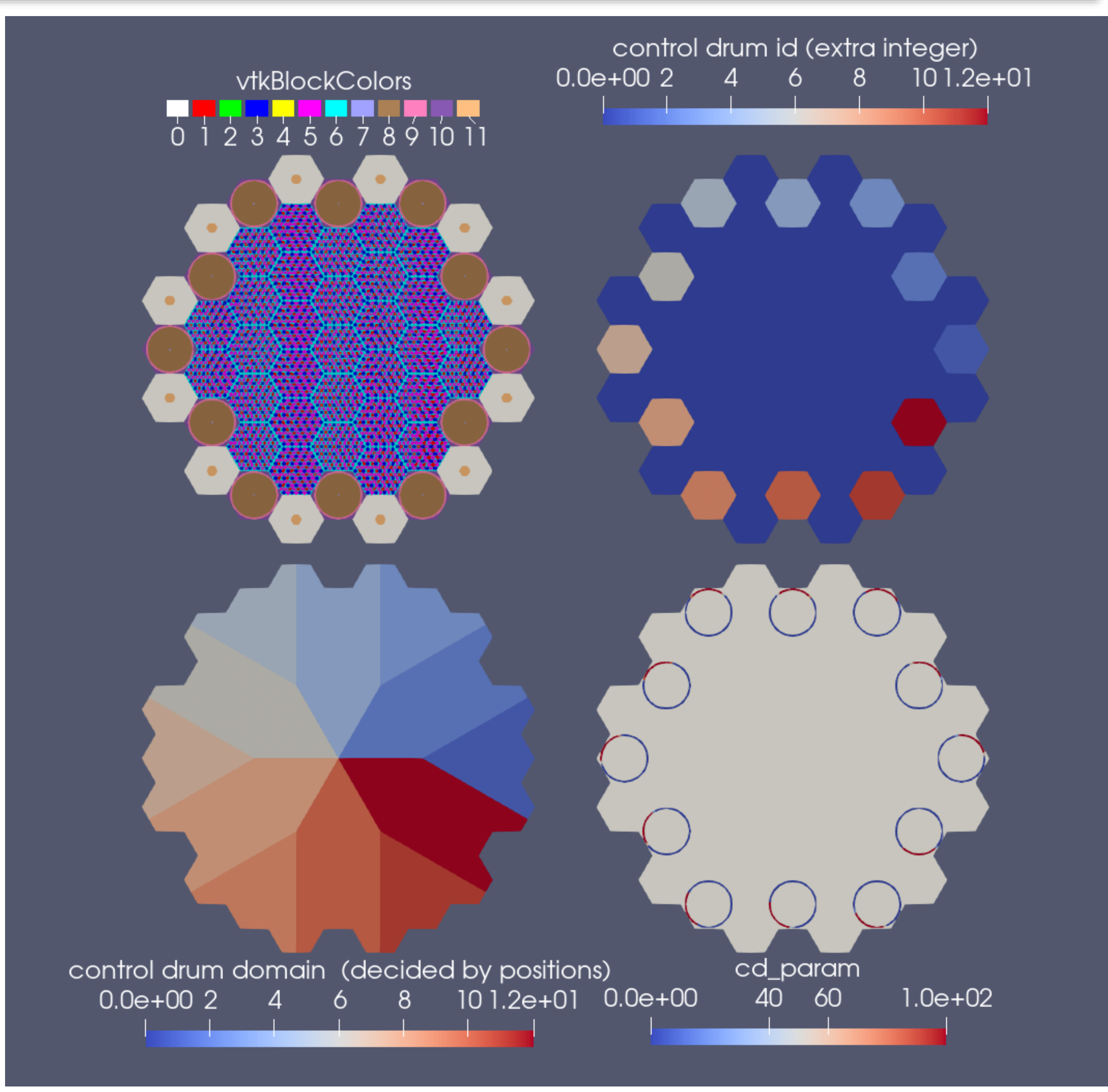


Figure 4-5. An example of control drums simulated by `MultiControlDrumFunction` object.

4.4 Current Limitations

Currently, because the functionalities of the `MultiControlDrumFunction` must rely on the metadata from `PatternedHexMeshGenerator`, only the reactor core meshes generated by `PatternedHexMeshGenerator` can be simulated. That is, the rotation of a standalone control drum mesh generated by `PolygonConcentricCircleMeshGenerator` cannot be simulated using this approach due to the absence of essential metadata. Additionally, due to the limitation in input mesh types of `PatternedHexMeshGenerator`, the control drum geometry must be contained within a single outer hexagon. Other shapes may be supported in the future if needed.

Also, this Function object only provides a mean to quantitatively compute and assign volume fraction values within the MOOSE framework. The usage of the material volume fractions has not been thoroughly tested with a Griffin input.

5 Reporting ID Capability

5.1 Reactor Analysis Motivation

The concept of reporting ID was devised to streamline post-processing for typical reactor cores with structured layout. Typical reactor cores have a hierarchical structure consisting of multiple levels: pin, assembly, core, and axial plane. By assigning corresponding IDs for those levels, referred to the reporting ID here, the regions of interest can be uniquely specified. Thus, the user can easily tally reactor component-wise values, such as pin-by-pin power distribution. The reporting ID capability was embedded into the MOOSE mesh generation capability for Cartesian and hexagonal geometries. For mesh elements belonging to the individual components in each hierarchical level, a reporting ID of the corresponding level is assigned during the mesh generation process using the extra element integer ID function available in MOOSE.

5.2 Assigning Pin, Assembly and Plane Reporting IDs

For typical Cartesian and hexagonal cores, reporting IDs can be setup in the hierarchical mesh build process from the 2D pin to the 3D core. The reporting IDs for individual pins can be assigned when assemblies are built because the IDs for pin level are uniquely determined from the pin arrangement within each assembly type. Similarly, the assembly reporting IDs are assigned in the core construction process. In order to implement the reporting ID capability for pin and assembly levels, the existing mesh generators constructing lattice structure were extended by adding the functionality of ID assignment for each input components in pin or assembly lattice structures.

For Cartesian lattices, `CartesianIDPatternedMeshGenerator` was implemented by extending the existing lattice mesh generator named `PatternedMeshGenerator` in the MOOSE framework. This new mesh generator adopts the existing input structures of `PatternedMeshGenerator` generator for geometry building and uses additional keywords to control the reporting ID assignment. First, a user can select an ID assignment scheme using *assign_type*, and the following schemes are currently available:

- *cell (default)*: Assign IDs for each component in lattice in sequential order.
- *pattern*: Assign IDs for each input component type.
- *manual*: Assign IDs based on user-defined mapping defined in *id_pattern*.

These assignment options are clearly illustrated in Figure 5-1. The name of reporting ID is provided through *id_name* depending on the hierarchical level of component. For example, *pin_id* or *assembly_id* is selected here for assembly or core generations, respectively.

For hexagonal lattices, `HexIDPatternedMeshGenerator` was implemented on top of `PatternedHexMeshGenerator` by adding the additional keywords for controlling the reporting ID generation as shown in Figure 5-2. The input structure of constructing the hexagonal lattice is reused here, and the same input keywords of the Cartesian version are used to control the reporting IDs. Note that separate reporting IDs are generated for the elements on duct regions as shown in Figure 5-2. This allows the user to easily post-process the detailed solutions of interests such as the heating distributions on assembly duct regions. In the double duct structures of control assembly design of SFRs, for example, each duct region has a separate reporting ID.

Note that these generators can be used for both assembly and core generations as illustrated in Figure 5-3 and Figure 5-4. Once the pin mesh information is created, the assemblies are generated with pin ID assignment. Then, the 2D core is assembled by merging the assemblies. Note that the pin ID assignment is readily available in the resulting 2D core mesh because the pin IDs specified in the assembly mesh data is replicated in this process. In this stage, additional assembly IDs are given for individual assemblies on the core lattice.

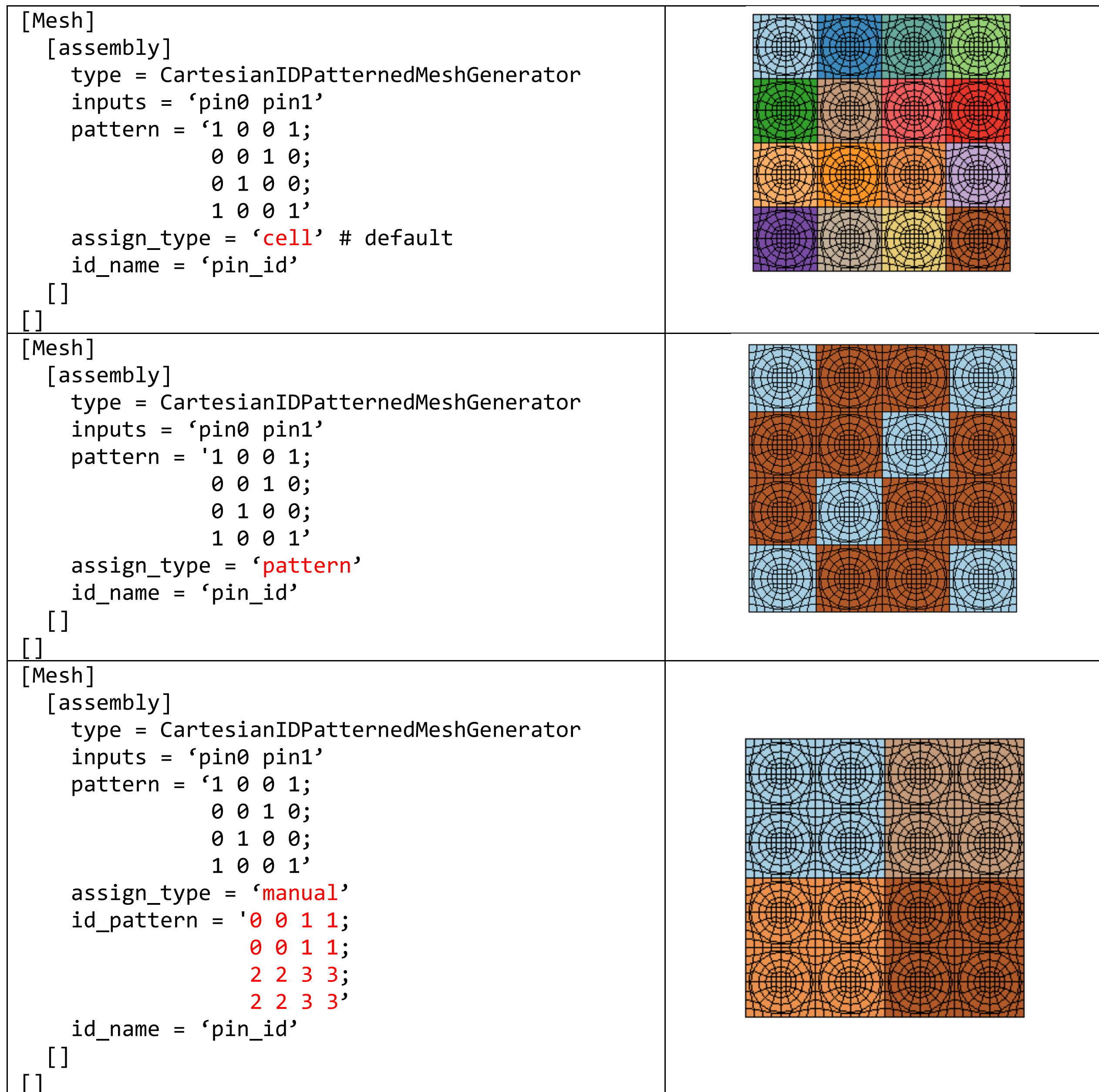
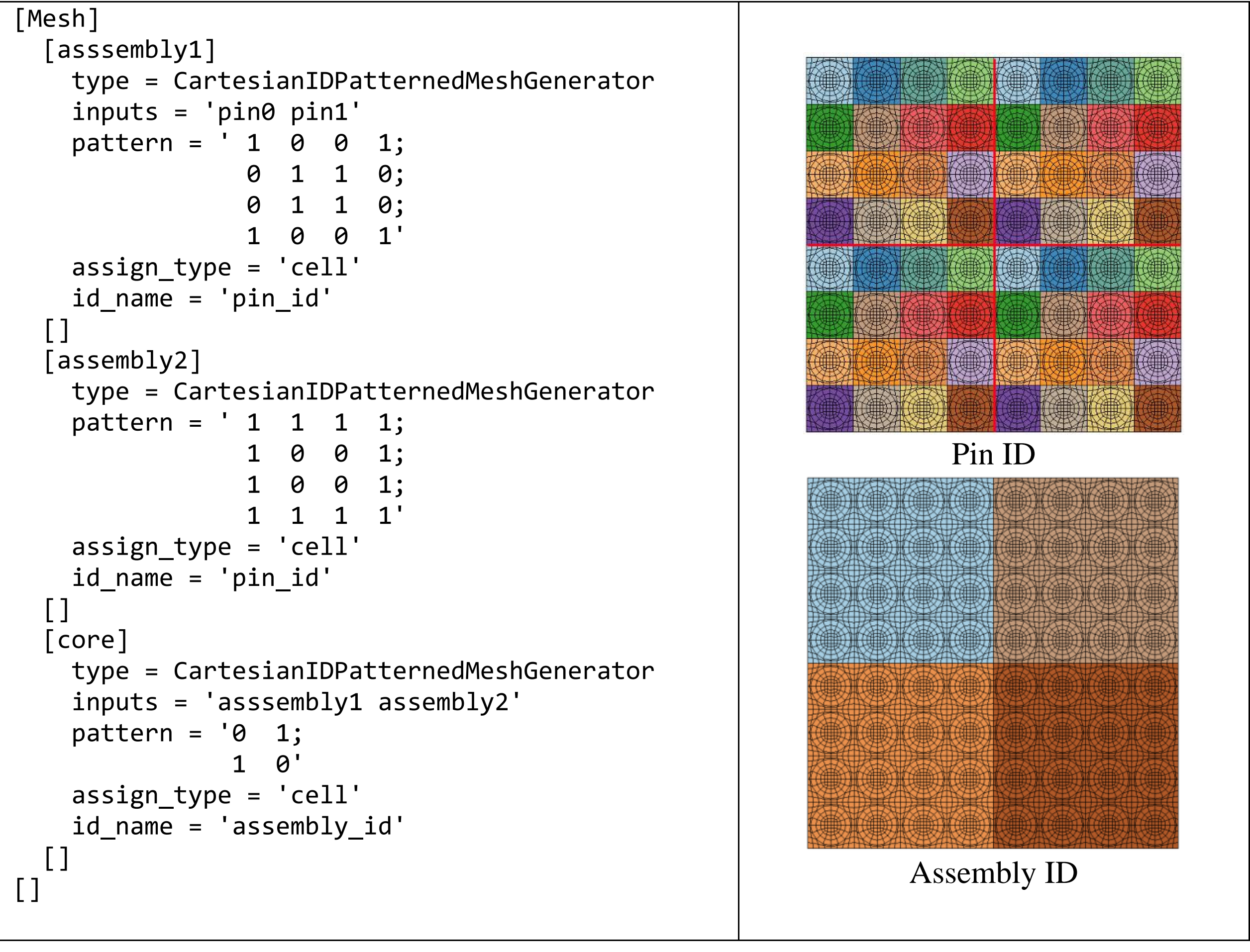


Figure 5-1. Reporting ID Generation for Cartesian Geometry

```
[Mesh]
  [assembly]
    type = HexIDPatternedMeshGenerator
    inputs = 'cell cell2'
    pattern_boundary = hexagon
    pattern = '  1 0 1;
               0 0 0 0;
              1 0 0 0 1;
               0 0 0 0;
                1 0 1'
    hexagon_size = 5.2
    duct_sizes = '2.4 2.5'
    duct_intervals = '1 1'
    id_name = 'pin_id'
    assign_type = ‘cell’
  []
[]
```

Figure 5-2. Reporting ID Generation for Hexagonal Geometry



```
[Mesh]
  [asssembly1]
    type = CartesianIDPatternedMeshGenerator
    inputs = 'pin0 pin1'
    pattern = ' 1  0  0  1;
                0  1  1  0;
                0  1  1  0;
                1  0  0  1'
    assign_type = 'cell'
    id_name = 'pin_id'
  []
  [assembly2]
    type = CartesianIDPatternedMeshGenerator
    pattern = ' 1  1  1  1;
                1  0  0  1;
                1  0  0  1;
                1  1  1  1'
    assign_type = 'cell'
    id_name = 'pin_id'
  []
  [core]
    type = CartesianIDPatternedMeshGenerator
    inputs = 'asssembly1 assembly2'
    pattern = '0  1;
               1  0'
    assign_type = 'cell'
    id_name = 'assembly_id'
  []
[]
```


Figure 5-3. Illustration of Reporting ID Generation for Pins and Assemblies in Cartesian Lattice

```
[Mesh]
  [assembly]
    type = HexIDPatternedMeshGenerator
    inputs = 'cell cell2'
    pattern_boundary = hexagon
    pattern = '  1 0 1;
                0 0 0 0;
               1 0 0 0 1;
                0 0 0 0;
                 1 0 1
    hexagon_size = 2.6
    duct_sizes = '2.4 2.5'
    duct_intervals = '1 1'
    id_name = 'pin_id'
    assign_type = ‘cell’
  []
  [core]
    type = HexIDPatternedMeshGenerator
    inputs = 'assembly'
    pattern_boundary = none
    pattern = '0 0;
              0 0 0;
               0 0'
    generate_core_metadata = true
    id_name = 'assembly_id'
    assign_type = 'cell'
  []
[]
```


Figure 5-4. Illustration of Reporting ID Generation for Pins and Assemblies in Hexagonal Lattice

The default ID numbering for *assign_type = cell* begins at 0 in the top left corner. The ID value increments by one as the pattern is traversed left to right, row by row. The default ID numbering for *assign_type = pattern* matches the pattern numbering shown in the *pattern* input.

In some cases, a user may want to exclude certain regions from being labeled with an ID, for example dummy regions that will later be deleted. This can be accommodated by listing mesh objects in the *exclude_id* input parameter; IDs will not be assigned to these mesh objects. An example of *exclude_id* is shown in Figure 5-5. This option currently works only with *assign_type = cell*.

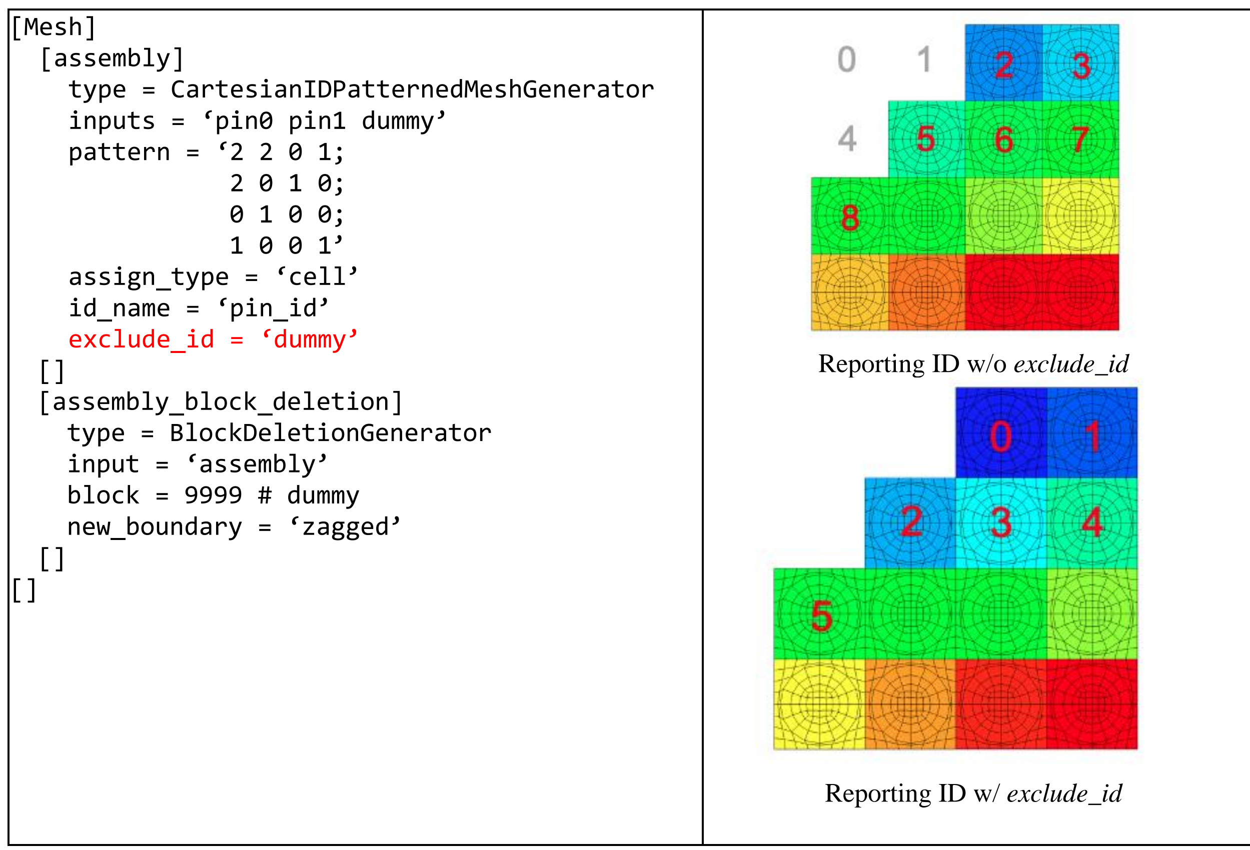


Figure 5-5. Illustration of auto-numbering using exclude_id option

For 3D cases, an additional reporting ID for axial planes can be optionally introduced. The elements in each pin and axial segment can be uniquely specified through the pin, assembly and plane IDs which allow 3D pin-wise distributions to be tallied. For specifying the plane reporting IDs, `PlaneIDGenerator` was implemented, and a sample case is shown in Figure 5-6. Note that this generator only works for 3D extruded geometries where the concept of axial layer is valid. This generator takes a 3D mesh data and its axial layer structure as input. This axial layer structure given in *z_layers* contains a list of z-coordinates defining each layer from bottom to top. If there are N planes, N+1 coordinate points should be defined here. If each axial plane is uniformly sub-divided into layers during the 3D extrusion, users may optionally assign distinct reporting IDs to individual sub-planes by using the *z_sublayers* option to defines the number of sub-layers in each plane to be assigned unique IDs. This is an array of integers corresponding to each plane.

```
[Mesh]
  ...
  [core_3d]
    ...
  []
  [set_plane_id]
    type = PlaneIDGenerator
    input = 'core_3d'
    z_layers = '0.0 2.0 4.0'
    id_name = 'plane_id'
  []
[]
```

Plane ID

Figure 5-6. Illustration of Reporting ID Generation for Plane

5.3 Assigning Depletion IDs

Depletion calculation using Griffin require depletion IDs to specify unique depletion zones where the composition changes over depletion time steps. By making use of the reporting IDs together with the material IDs, the depletion IDs can be automatically prepared for Cartesian and hexagonal cores. For a pin-level depletion case, the individual pins can be identified by the pin and assembly IDs, and the detailed depletion regions within a pin can be further divided by material IDs. Thus, the depletion IDs for the entire problem domains can be specified by finding the unique combinations of assembly, pin and material IDs. For assembly-wise depletion, the user can set up the depletion IDs by combining the assembly and material IDs. The described capability was implemented in `DepletionIDGenerator`. As shown in Figure 5-7, *id_names* lists integer ID names used for setting up the depletion zones. Note that the material ID does not need to be defined in the list because it is included by default. The level of details in depletion zones can be controlled by the integer IDs defined here. For example, one can set up pin-by-pin and axial layer-by-layer arrangement of depletion zone by specifying those three IDs: *pin_id*, *assembly_id* and *plane_id*. Users may optionally provide a list of material IDs to be excluded in the depletion ID generation. For example, a non-fuel region can be excluded in the depletion ID generation using *exclude_material_id*, which creates a more concise depletion ID arrangement. For those materials listed in *exclude_material_id*, the depletion ID is set to zero because the depletion ID should be assigned for the entire region even if not used in the actual depletion calculation.

```
[Mesh]
  ...

  [depletion_id]
    type = DepletionIDGenerator
    input = 'core'
    id_names = 'pin_id assembly_id'
    exclude_material_id = '3 4' # 3:clad 4: coolant
  []
[]
```


Figure 5-7. Depletion ID Generation using Reporting IDs

6 Core Periphery Meshing Capability

6.1 Reactor Analysis Motivation

The outermost assemblies in hexagonal reactor cores are often surrounded by a circular / cylindrical peripheral zone as shown in red in Figure 6-1. While this zone is sometimes ignored in reactor physics analysis, it should be included for completeness and to perform ex-core shielding calculations. This zone takes on an irregular shape as it is the area between the outer cylinder and the jagged outer core boundary. Typically, the user is required to mesh this zone using an external meshing tool (e.g., Cubit). A generic triangle mesher should be implemented in MOOSE framework to handle meshing of this geometry.

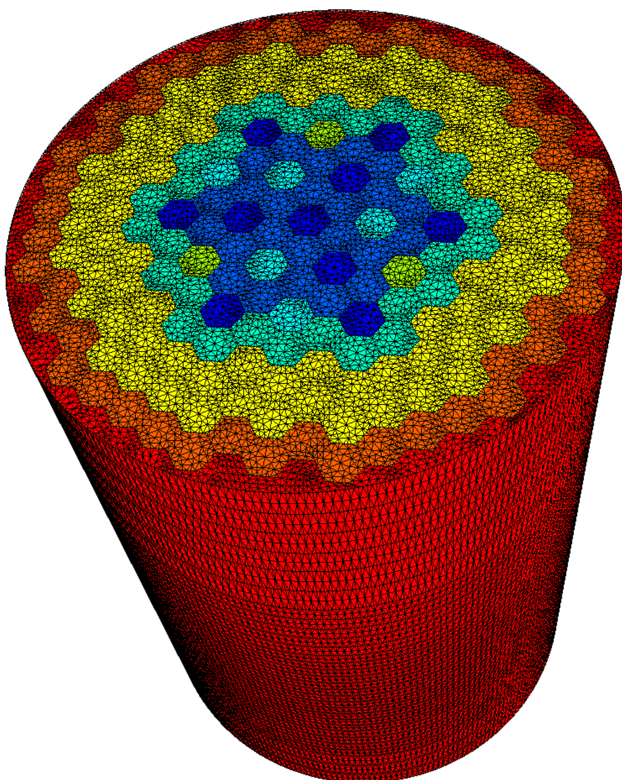


Figure 6-1: Reactor core geometry with meshed core periphery region (Cubit).

6.2 Triangle Licensing Compatibility Issues

It was quickly determined that while extremely useful, writing a new triangle routine specifically for the MOOSE framework would be a large undertaking and not within scope of this project. Therefore, leveraging an open-source triangle mesher would be the best path to success. The primary challenge with leveraging an external code is ensuring that the license for the external code is compatible with MOOSE licensing. Due to the licensing of MOOSE itself, and especially of the more restrictive related projects like BISON, Griffin, etc., any third-party code being used needs to have a relatively open license that allows for integration, distribution, and redistribution under MOOSE's terms. This largely restricts the pool of possible tools to so-called "permissive" licenses,

such as BSD (BSD License, 2021), MIT (MIT License, 2021), and Apache (Apache License, 2021), and potentially the GNU Lesser GPL license (LGPL License, 2021). Codes released under the ordinary GNU GPL license (GPL License, 2021), or their own custom restrictive licenses, are generally not compatible with MOOSE's license and thus not options for this work. Several popular open-source libraries including Triangle (Shewchuk, 2021), TetGen (TetGen, 2021), and gmsh (Geuzaine & Remacle, 2021) were considered, but all but one were eliminated due to their licensing requirements not being compatible with the MOOSE framework.

There are still several triangulation libraries available under permissive licenses, but additionally, a third-party meshing library also needed the ability to mesh around "holes" in the meshing region. Most of the libraries found would triangulate an entire enclosed region, but the use case for MOOSE meshing required the ability to mesh an area around some existing mesh (e.g., a reactor geometry), which additionally limited the library options. The poly2tri code (Hasse, 2021) was selected for implementation into MOOSE after confirming license compatibility with the MOOSE framework developers.

6.3 Core Periphery Triangulation Mesher

The poly2tri library is designed to take a simple 2D polygon region with an outer boundary represented by a polyline and triangulate the region interior to that boundary. Optionally, the library can also be given any number of "holes" within the outer boundary, which are also defined by boundary polylines, and indicate regions within the outer boundary that should not be part of the triangulation.

During triangulation, the library will by default use only the points given in the outer boundary polyline and any hole boundary polylines to perform the triangulation, which can result in lower quality triangles and lead to difficulty when using the triangulation in FEM calculations. The library thus also allows for the input of any number of Steiner points, which are extra triangulation points that can be used to help improve the quality of the triangulation.

The `TriangulatedMeshGenerator` (TMG) was created to connect the functionality of the poly2tri library to the MOOSE mesh generation system. Since the motivation of this effort was to mesh the region between a reactor fuel core and the core periphery (typically a cylinder), TMG was designed to create a circular triangulation outer boundary and use the outer boundary of an existing meshgenerator to define the boundary of a single hole region which the triangulation library will exclude.

The outer circular boundary properties are implemented as parameters in MOOSE input and are defined by the circle radius and the number of segments to discretize the circle into. The inner hole boundary properties are also implemented as parameters in MOOSE input and are defined by the meshgenerator object name and the corresponding outer boundary ID or name. TMG can also optionally take additional circle radius/segment definitions between the core region and the outer boundary to be used as Steiner points to help improve the quality of the triangles produced.

Figure 6-2 shows a simple example triangulation. The core is a simple `GeneratedMeshGenerator` (GMG) square with a `SideSetsAroundSubdomainGenerator` to define an outer boundary sideset that TMG can use. TMG accepts the generated mesh and the sideset name to define the hole region, along with the properties for the outer circle boundary.

```
[Mesh]
  [gmg] # simple internal core
    type = GeneratedMeshGenerator
    dim = 2
    nx = 2
    ny = 2
    xmin = -1
    xmax = 1
    ymin = -1
    ymax = 1
  []

  [gmg_boundary] # create outer boundary sideset
    type = SideSetsAroundSubdomainGenerator
    input = gmg
    block = 0
    new_boundary = gen_outside
  []

  [tmg] # triangulate periphery
    type = TriangulatedMeshGenerator
    subdomain_id = 1

    # inner boundary mesh input
    inner_boundary_mesh = gmg_boundary
    inner_boundary_name = gen_outside

    # outer circle boundary settings
    outer_circle_radius = 3
    outer_circle_num_segments = 20
  []
[]
```

Figure 6-2: Example TMG mesh, simple GMG core.

Figure 6-3 shows a more complex example core geometry, also created with several stitched GMG meshes into a cartesian arrangement. This example also shows one of the main limitations with this library, that although it does create a triangulation of the periphery, the triangles are not always of the quality desired for FEM calculations and would benefit from a quality refinement step.

```
[Mesh]
  [gmg_center] # simple internal core
    type = GeneratedMeshGenerator
    dim = 2
    nx = 20
    ny = 20
    xmin = -0.5
    xmax = 0.5
    ymin = -0.5
    ymax = 0.5
  []
```

```
# copied and stitched GMG meshes removed
# for brevity

  [cmbn_boundary] # create outer boundary sideset
    type = SideSetsAroundSubdomainGenerator
    input = 'stiched_top_bottom'
    block = 0
    new_boundary = 'gen_outer'
    replace = true
  []

  [tmg] # triangulate periphery
    type = TriangulatedMeshGenerator
    subdomain_id = 20

    # inner boundary mesh input
    inner_boundary_mesh = cmbn_boundary
    inner_boundary_name = 'gen_outer'

    # outer circle boundary settings
    outer_circle_radius = 2
    outer_circle_num_segments = 50
  []
[]
```

Figure 6-3: Example TMG mesh, simple cartesian core.

Figure 6-4 shows a hexagonal assembly geometry, created with the `PatternedHexMeshGenerator` and loaded with `FileMeshGenerator`.

```
[Mesh]
  [hex_in] # imported detailed hex geometry
    type = FileMeshGenerator
    file = hex_geometry.e
  []

  [tmg] # triangulate periphery
    type = TriangulatedMeshGenerator
    subdomain_id = 2000

    # inner boundary mesh input
    inner_boundary_mesh = hex_in
    inner_boundary_id = 5001

    # outer circle boundary settings
    outer_circle_radius = 18
    outer_circle_num_segments = 50
  []
[]
```

Figure 6-4: Example TMG mesh, detailed hex assembly.

Like Figure 6-3, Figure 6-4 also shows low quality triangles. Figure 6-5 shows the same hex core with the addition of two rings of Steiner points to improve the quality of the triangles.

```
[Mesh]
  [hex_in] # imported detailed hex geometry
    type = FileMeshGenerator
    file = hex_geometry.e
  []

  [tmg] # triangulate periphery
    type = TriangulatedMeshGenerator
    subdomain_id = 2000

    # inner boundary mesh input
    inner_boundary_mesh = hex_in
    inner_boundary_id = 5001

    # outer circle boundary settings
    outer_circle_radius = 18
    outer_circle_num_segments = 50

    # extra steiner point circles
    extra_circle_num_segments = '200 100'
    extra_circle_radii = '16 17'
  []
[]
```

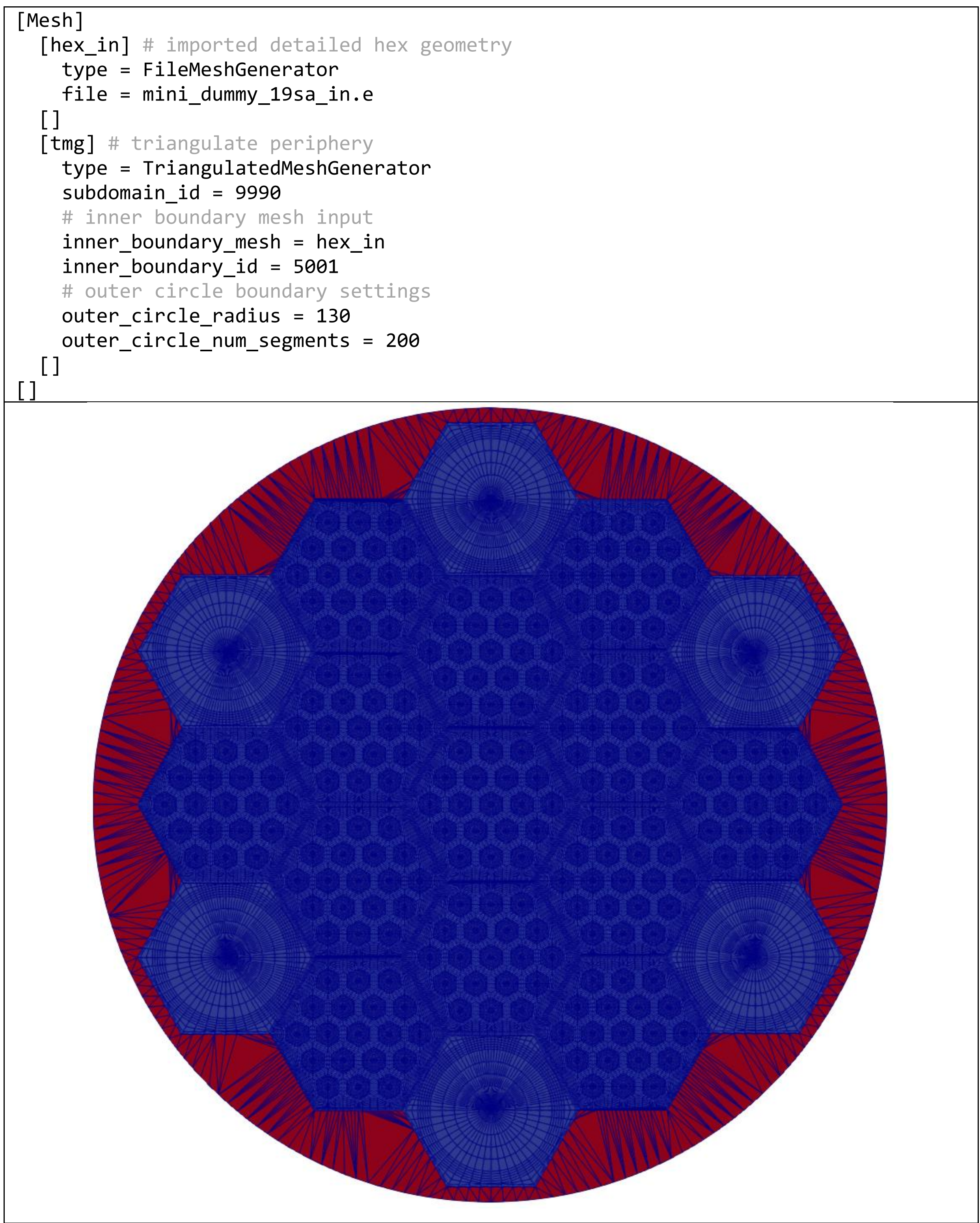
Figure 6-5: Example TMG mesh, detailed hex assembly with Steiner points.

Figure 6-6 shows the example from Figure 6-5 extruded into 3D using `MeshExtruderGenerator` as an example of compatibility of the triangulated mesh with additional mesh generators.

```
[Mesh]
  [hex_in] # imported detailed hex geometry
    type = FileMeshGenerator
    file = hex_geometry.e
  []
  [tmg] # triangulate periphery
    type = TriangulatedMeshGenerator
    subdomain_id = 2000
    # inner boundary mesh input
    inner_boundary_mesh = hex_in
    inner_boundary_id = 5001
    # outer circle boundary settings
    outer_circle_radius = 18
    outer_circle_num_segments = 50
    # extra steiner point circles
    extra_circle_radii = '16 17'
    extra_circle_num_segments = '200 100'
  []
  [extrude] # extrude into 3D
  type = MeshExtruderGenerator
  input = tmg
  num_layers = 5
  extrusion_vector = '0 0 20'
  []
[]
```

Figure 6-6: Example TMG mesh, detailed hex core with Steiner points, and extruded into 3D.

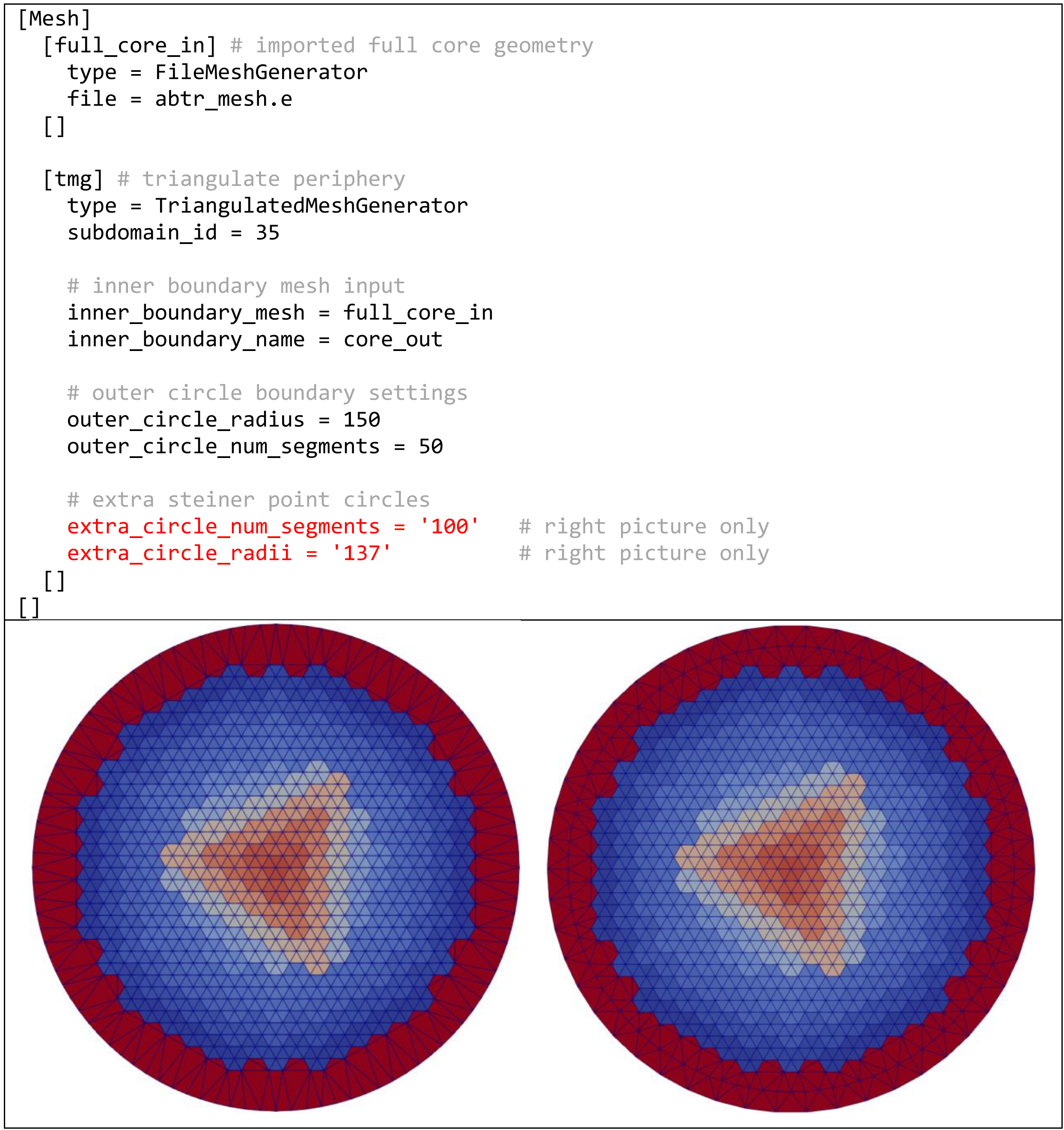
Figure 6-7 shows an example of a multi-assembly core, consisting of 19 subassemblies with different subassembly types created with the `PatternedHexMeshGenerator`.



```
[Mesh]
  [hex_in] # imported detailed hex geometry
    type = FileMeshGenerator
    file = mini_dummy_19sa_in.e
  []
  [tmg] # triangulate periphery
    type = TriangulatedMeshGenerator
    subdomain_id = 9990
    # inner boundary mesh input
    inner_boundary_mesh = hex_in
    inner_boundary_id = 5001
    # outer circle boundary settings
    outer_circle_radius = 130
    outer_circle_num_segments = 200
  []
[]
```

Figure 6-7: Example TMG mesh, multi-assembly core.

Figure 6-8 shows an example of a full core mesh created with `PatternedHexMeshGenerator` (`SimpleHexagonGenerator` was used for individual assembly inputs). This example shows the meh generated with only the outer surface (left), and with the mesh with an extra ring of Steiner points added (right), to show how the quality of the mesh can be improved over the default behavior.



```
[Mesh]
  [full_core_in] # imported full core geometry
    type = FileMeshGenerator
    file = abtr_mesh.e
  []

  [tmg] # triangulate periphery
    type = TriangulatedMeshGenerator
    subdomain_id = 35

    # inner boundary mesh input
    inner_boundary_mesh = full_core_in
    inner_boundary_name = core_out

    # outer circle boundary settings
    outer_circle_radius = 150
    outer_circle_num_segments = 50

    # extra steiner point circles
    extra_circle_num_segments = '100'   # right picture only
    extra_circle_radii = '137'          # right picture only
  []
[]
```

Figure 6-8: Example TMG mesh, full core with (left) only outer boundary nodes, and (right) ring of Steiner points added.

6.4 Other Uses of the Triangulation Mesher

This mesh generator was designed for a relatively specific use case of meshing the area between an existing reactor core geometry and a cylindrical periphery, but the generator could be further expanded. The assumed cylindrical boundary could be extended to allow multiple generated boundary shapes, to accept a polyline as input for an external boundary that could have been generated from an external tool, or to accept an existing mesh generator and boundary ID similar to the internal core boundary.

This mesh generator was also designed based on the assumption that the entire core geometry would already be fully meshed and would be input into the triangulation library as a single "hole", but that could also be extended. The library supports triangulation around multiple holes, so the generator could be extended to accept multiple existing mesh generators and boundary IDs. This could be used for example in a core with multiple assemblies that did not have the region between the assemblies already meshed. The individual assemblies could be specified as a collection of holes to the library, and it would triangulate not only the areas from the core the periphery, but also all of the empty regions between all of the individual assemblies.

6.5 Limitations and Recommendations

The most notable limitation of the poly2tri library is that it only performs triangulation using that points that are already defined by the outer boundary and core boundary. This produces a valid triangulation of the intended region, but not necessarily with triangles of sufficient quality for use in FEM calculations. The library allows for the specification of Steiner points that can be used to help improve quality, but these are entered manually to the library. The most useful addition to this mesh generator would be a refinement algorithm that improves the quality of the triangles to a given threshold. These algorithms often take and existing triangulation as input and go through the process of improving any triangles that are below the quality threshold, so the implementation of this mesh generator should serve as a good starting point for the implementation of such an algorithm.

Given the limited triangulation functionality available in license-compatible open source products, it is recommend that the MOOSE framework consider pursuing development of a native Delaunay triangulation scheme to obtain high quality elements.

7 Reactor Geometry Mesh Builder (RGMB) Capability

7.1 Reactor Analysis Motivation

Reactor analysts for conventional Cartesian and hexagonal reactor cores typically want to specify both geometry and materials simultaneously, as these two parts of the input need to be mapped together to specify the physics problem. Additionally, it is most natural to think of building up a reactor core by pins, then assemblies, then cores. The previous tools described in this work are a huge improvement for MOOSE physics analysts, but additional features are desired to hide unnecessary meshing details from the user, incorporate language specific to reactor design, and assign materials at the meshing stage to abstract away notions of blocks (which the user does not need to know other than for purposes of material assignment). A series of new capabilities has been developed to address these issues and further simplify meshing input for conventional Cartesian and hexagonal reactor pins, assemblies, and cores.

7.2 Building a Repeated Reactor Geometry

Mesh generators have been built that take input in a style preferred by reactor analysts to first define Pins, combine Pins into Assemblies, and finally combine Assemblies into a Core. These MeshGenerators were designed and developed to make use of the existing meshing capabilities in MOOSE and simplify the input required for reactor applications. To help enable this, the MOOSE development team implemented a "subgenerator" capability for MeshGenerators that allows a MeshGenerator to itself call another and have the generated mesh and properties accessible between the two. A global parameters mesh block is used to define dimensionality (2D or 3D), grid style (Cartesian or hexagonal), assembly pitch, and axial geometry and intervals. These properties are assumed to be common across the core and will be inherited by every subsequent pin and assembly.

Material definitions and associations are made at the same time as the pin cell mesh is defined, and this material information is automatically propagated throughout the mesh building process, including 3D extrusion. These three utilities (which call other utilities developed in this work behind the scenes) can generate a full core pin by pin mesh, with automatic ids set for pin, assembly, and planes, as well as material assignment embedded in the mesh. Mesh metadata and extra element integers are heavily leveraged to enable this capability. 3D extrusion may be performed at the pin cell, assembly, or core level. Sidesets are automatically defined for pin, assembly and core outer boundaries, as well as the top and bottom surfaces (for 3D meshes).

A brief discussion follows of each of the new mesh generators.

7.2.1 GlobalMeshParams

A Mesh block calling the `GlobalMeshParams` type is required to begin the RGMB procedure. The input options for this mesh generator are as follows.

GlobalMeshParams

- *dim*: Number of dimensions of the final mesh (Options: 2 or 3, default is 2)
- *geom*: Type of geometry (Options: "Square" or "Hex", default is "Square")
- *assembly_pitch*: Flat-to-flat size of the assembly's outermost boundary
- *axial_regions*: Array of heights of the axial regions (not required for 2D meshes)
- *axial_mesh_intervals*: Array of the number of intervals for each axial region in *axial_regions* (not required for 2D meshes)

- *procedural_subdomain_assignment:* Whether the subdomain IDs should be procedurally generated (true) or derived from region ids (false). If true, each meshing subinterval (radially and axially) will be assigned a subdomain ID that is unique to that pin/assembly type. If false, each meshed subinterval (radially and axially) will be assigned a subdomain id identical to the assigned region ID. (Options: true or false, default is false)

The function of this object is to build an empty (null) mesh object with metadata that can be passed into subsequent mesh generators. Specification of the common parts of the geometry in one location reduces user input errors and ensures axially conformal meshes. This object also ensures that core is defined in a uniform grid based on the *assembly_pitch* which allows for easy definition of irregular core boundary shapes.

Generally, users of the RGMB tool do not need to have awareness of subdomain IDs. However, some further explanation is warranted on what the code is doing behind the scenes regarding subdomain numbering. In order to handle the occasional situations where multiple blocks are created for the center pin or background region (discussed in Table 3-1) of a pin cell, subdomain IDs are created for each meshing subinterval, not just each zone. The generated subdomain IDs are unique to a pin but will repeat if the pin is repeated in the geometry. The choice of *procedural_subdomain_assignment=true* was introduced to allow the user to force unique numbers for each of these subdomains rather than using the region ID assigned to that meshing interval. Generally, it is convenient to simply use the same ID value for both region ID and subdomain ID.

7.2.2 PinMeshGenerator

After defining a mesh object with `GlobalMeshParams`, one may invoke `PinMeshGenerator` to create instances of reactor "Pins". A "Pin" is actually a pin cell containing concentric rings to represent rodded fuel or other materials, a background region, a square or hexagonal boundary, and optional duct regions. Ducts are also available in the case of needing to represent a homogenized assembly (no concentric rings explicitly represented). The geometry parameters are very similar to those required by `PolygonConcentricCircleMeshGenerator`, which is called behind the scenes. In the case of square pins, `TransformGenerator` is also called to rotate the pins to the same orientation that would be expected from ConcentricCircleMeshGenerator. Additionally, material assignments are specified for each axial plane in the pin cell through an array of integers supplied in *region_ids*. Materials are specified from the center of the pin outward, and from the bottom of the pin up. Hence, the first row of the material IDs corresponds to axial bottom of the pin. Reactor analyst terminology like "pitch" is used here. The user has the option to extrude to 3D if desired, in which case `FancyExtruderGenerator` extrudes that mesh based on the specification in the `GlobalMeshParams`. The axial layers are also assigned axial reporting IDs via `PlaneIDGenerator`. This procedure is the same regardless of which RGMB MeshGenerator is used for the extrusion process.

PinMeshGenerator

- *global_params*: The name of the GlobalMeshParams MeshGenerator.
- *pin_type*: A positive integer ID for this pin definition.
- *pitch*: The flat to flat size of the pin.
- *num_sectors*: The number of sectors each side of the pins mesh is divided into.
- *ring_radii*: An array of radii for any rings to have in the pin. (Optional)
- *duct_radii*: An array of apothem distances of the inner wall of any ducts. (Optional)

- *mesh_intervals*: An array of the number of intervals for each radial division of the pin starting from the innermost ring and ending with the outermost duct. Needs to length equal to the number of rings + the number of ducts + 1 for the region separating the rings and ducts.
- *region_ids*: An array of positive integer IDs given to each radial and axial region. The rows indicate radially region and need to be of equal length to *mesh_intervals*, the columns indicate axially region and need to be of equal length to *axial_regions* in the `GlobalMeshParams`. These are set both in the element extra integers and as the subdomain ID for the region if the procedural generation of subdomain IDs is not enabled.
- *extrude*: Whether to extrude at this step if this is to be a 3D mesh. (Options: true or false, default is false)
- *quad_center_elements*: Whether to use quad center elements. If this option is set to false (meaning tri elements used in the center) and *procedural_subdomain_assignment =false*, then the user must define an inner most ring with a unique region ID and only one meshing interval. This is due to the limitations in the exodus format in that only elements with the element type can have the same subdomain ID. (Options: true or false, default is true)

The user should define all pin types in the problem in different mesh generator objects, and differentiate them by unique integers in the *pin_type* card. This will be needed to propagate material assignments during extrusion.

7.2.3 AssemblyMeshGenerator

Pin cells are patterned into an "Assembly" using `AssemblyMeshGenerator`. This mesh generator calls the automatic reporting id patterned mesh generators (`HexIDPatternedMeshGenerator`, `CartesianIDPatternedMeshGenerator`) to combine pins into an assembly. Ducts can be applied around hexagonal patterns of pin cells and are currently disabled for Cartesian grids. Along with the definition of ducts, hexagonal assemblies have the added requirement of the definition of a background region. The background region is the region surrounding the array of hexagonal pins and inside the inner boundary of any ducts. If no duct exists, the background region extends to the outer boundary of the assembly. Any number of duct layers can be defined via the *duct_radii* parameter, however the outermost duct will always terminate at the assembly boundary that is defined in the GlobalMeshParams.

AssemblyMeshGenerator

- *inputs*: An array of the names of the pins that you want to reference in the pattern.
- *assembly_type*: A positive integer ID for this assembly definition.
- *pattern*: A 2D array of pins in *inputs* indicating location of the pins in the assembly. The shape of the pattern must match the shape expected for the geometry type defined in GlobalMeshParams.
- *background_intervals:* The radial meshing intervals for the background region. (Required for "Hex" geometry)
- *background_regions_id:* An array of the region IDs for each axial division of the background region. (Required for "Hex" geometry)
- *duct_radii:* An array of sizes of ducts to place around the assembly, inside of the boundary, given by apothem. (Option only for "Hex" geometry)
- *duct_intervals:* An array of the radial meshing intervals for the ducts. This parameter must be of equal length to *duct_radii.* (Option only for "Hex" geometry)
- *duct_region_ids:* An array of the region IDs for each axial division of the ducts defined in *duct_radii.* (Option only for "Hex" geometry)

- *extrude*: Whether to extrude at this step if this is to be a 3D mesh. (True or false, defaults to false)

During the stitching process of pins into an assembly, automatic pin numbering is applied and stored in extra element integers labeled *pin_id*. The user has the option to extrude to 3D at this point in the process if desired. In the case of the hexagonal assemblies, the background region and any ducts defined are given the assembly type ID, and they are also given a pin type ID of the maximum value of their integer type (uint_16) to designate them as not belonging to any pin. This is designation is required during extrusion for material information propagation.

The user should define all assembly types in the problem in different mesh generator objects, and differentiate them by unique integers in the *assembly_type* card. This will be needed to propagate material assignments.

7.2.4 CoreMeshGenerator

Assemblies are patterned into a "Core" using the `CoreMeshGenerator` object. This mesh generator calls the automatic reporting id patterned mesh generators (`HexIDPatternedMeshGenerator`, `CartesianIDPatternedMeshGenerator`) to combine assemblies into a core. Unlike the underlying patterned mesh generators, the `CoreMeshGenerator` object permits "empty" spaces in the core map through use of a reserved name for a dummy assembly. This allows the user to bypass definition of extra dummy assemblies and subsequent deletion. (Behind the scenes, this is what the code is doing, but the user does not need to concern themself with this.)

CoreMeshGenerator

- *inputs*: An array of the names of the assemblies that you want to reference in the pattern. The array can also include the *empty_position_name*.
- *empty_position_name*: The name used in "*inputs*" to indicate a dummy assembly position that should be left empty. This can be treated the same as any real mesh name in the input.
- *pattern*: A 2D array of pins in "*inputs*" indicating location of the pins in the assembly. The shape of the pattern must match the shape expected for the geometry type defined in GlobalMeshParams.
- *extrude*: Whether to extrude at this step if this is to be a 3D mesh. (Options: true or false, default is false)

During the stitching process of assemblies into a core, automatic assembly numbering is applied and stored in extra element integers called *assembly_id*. The user has the option to extrude to 3D at this point in the process if desired. Once extrusion is performed, reporting IDs to identify the plane are automatically stored in extra element integers called *plane_id.*

7.2.5 Automatic Sideset Generation

The MeshGenerator objects automatically assign a predictable sideset ID and name for the outer boundary of the geometry, and top and bottom surfaces (if extrusion to 3D is requested). The *pin_type* and *assembly_type* integer values are evaluated and used in the naming and numbering process. Table 7-1 describes the sidesets which are autogenerated with these tools.

Table 7-1. Description of auto-generated sidesets

Object	Sideset Description	Sideset ID	Sideset Name
`PinMeshGenerator`	Nodes comprising the outer boundary of a pin cell (hex or square)	20,000 + {*pin_type*}	'outer_pin_ {*pin_type*}'
`AssemblyMeshGenerator`	Nodes comprising the outer boundary of an assembly (hex or square)	2,000 + {*assembly_type*}	'outer_assembly_ {*assembly_type*}'
`CoreMeshGenerator`	Nodes comprising the outer boundary of a core (jagged edge permitted, hex or square lattice)	200	'outer_core'
`PinMeshGenerator`, `AssemblyMeshGenerator`, `CoreMeshGenerator`	Top surface of a 3D mesh	201	'top'
`PinMeshGenerator`, `AssemblyMeshGenerator`, `CoreMeshGenerator`	Bottom surface of a 3D mesh	202	'bottom'

For example, if `PinMeshGenerator` has been given the input card *pin_type* = 6 input, the resulting 2D pin cell mesh will have an external boundary sideset called 'outer_pin_6' with ID 20006.

If `AssemblyMeshGenerator` has been given the input card *assembly_type* = 13, the resulting 2D assembly mesh will have an external boundary sideset called 'outer_assembly_13' with ID 2013.

The `CoreMeshGenerator` external boundary sideset is 200.

If the mesh is extruded at any stage (Pin, Assembly, or Core), then additional sidesets 201 ('top') and 202 ('bottom') are defined.

7.3 Cartesian RGMB Example

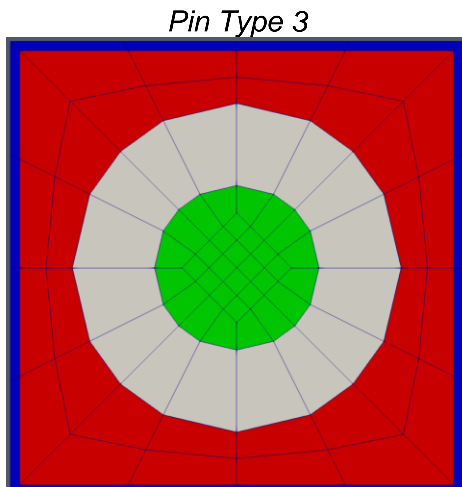
The following figure demonstrates the simplicity of defining a 3D Cartesian core containing 24 assemblies (3 assembly types), null spaces around the core perimeter, and each assembly contains 3 pin types with 3 axial zones. Less than 100 lines of input is required in this example and the material mapping is already done and stored on each mesh element. Additionally, the reporting ids for pin, assembly, and plane are automatically applied for ease of post-processing.

In this example, global parameters are first set to define a square lattice, the lattice pitch, and three axial regions with 2 subintervals each. Then, three pin types are defined. The second row of the table generates the 3 assembly types, and the third row of the table generates the core.

```
[Mesh]
  [./gmp]
    type = GlobalMeshParams
    dim = 3
    geom = "Square"
    assembly_pitch = 7.10315
    axial_regions = '5.0 20.0 5.0'
    axial_mesh_intervals = '2 2 2'
  []

  [./pin1]
    type = PinMeshGenerator
    global_params = gmp
    pin_type = 1
    pitch = 1.42063
    num_sectors = 4
    ring_radii = '0.3818'
    region_ids = '1 2;
                  11 12;
                  1 2'
    mesh_intervals = '1 1'
  []

  [./pin2]
    type = PinMeshGenerator
    global_params = gmp
    pin_type = 2
    pitch = 1.42063
    num_sectors = 4
    region_ids = '3;
                  13;
                  3'
    mesh_intervals = '1'
  []
  [./pin3]
    type = PinMeshGenerator
    global_params = gmp
    pin_type = 3
    pitch = 1.42063
    num_sectors = 4
    ring_radii = '0.254 0.508'
    duct_radii = '0.68'
    region_ids = '4 1 2 5;
                  12 11 12 15;
                  4 1 2 5'
    mesh_intervals = '1 1 2 1'
  []
```



<table>
<tr>
<td>

```
  [./amg]
    type = AssemblyMeshGenerator
    assembly_type = 1
    inputs = 'pin1 pin2 pin3'
    pattern = '1 1 0 1 1;
               1 2 0 2 1;
               0 0 2 0 0;
               1 2 0 2 1;
               1 1 0 1 1'
  []

  [./amg1]
    type = AssemblyMeshGenerator
    assembly_type = 2
    inputs = 'pin1 pin2 pin3'
    pattern = '1 1 1 1 1;
               1 1 1 1 1;
               1 1 1 1 1;
               1 1 1 1 1;
               1 1 1 1 1'
  []

  [./amg2]
    type = AssemblyMeshGenerator
    assembly_type = 3
    inputs = 'pin1 pin2 pin3'
    pattern = '1 1 1 1 1;
               1 0 0 0 1;
               1 0 2 0 1;
               1 0 0 0 1;
               1 1 1 1 1'
  []
```

</td>
<td>
Assembly Type 1
</td>
</tr>
<tr>
<td>

```
  [./cmg]
    type = CoreMeshGenerator
    inputs = 'amg amg1 amg2 empty'
    empty_position_name = empty
    pattern = '3 3 1 1 3 3;
               3 2 0 0 2 3;
               1 0 2 0 0 1;
               1 0 0 2 0 1;
               3 2 0 0 2 3;
               3 3 1 1 3 3'
    extrude = true

  []

[]
```

</td>
<td>
3D Core with Irregular Boundary Shape
</td>
</tr>
</table>

Figure 7-1: 3D Cartesian core input and mesh built using MOOSE's new reactor geometry mesh builder.

7.4 Hexagonal RGMB Example

The following figure demonstrates the simplicity of defining a 3D hexagonal core using the new reactor geometry mesh builder. The 55-assembly core contains 3 assembly types and null spaces, and each assembly contains up to 3 pin types. Less than 120 lines of input is required and the material mapping is already done and stored on each mesh element. Additionally, the reporting ids for pin, assembly, and plane are automatically applied for ease of post-processing.

The multi-page figure begins on the following page.

The first row of the figure depicts the setting of global parameters using the `GlobalMeshParams` mesh generator, and the definitions of 3 pin types (labeled with unique *pin_type* values 1, 2, and 3). The material ids for each axial plane are assigned in the *region_id* input, which has 3 rows since there are 3 axial regions in the input `GlobalMeshParams` object. The meshes are 2D at this point but store all the necessary metadata for extrusion and material mapping.

The second row of the figure depicts the creation of 3 assembly types (labeled with unique *assembly_type* values 1, 2, and 3). Note the addition of background and duct zones and materials. The assembly pitch was inherited from the mesh metadata on the pin inputs. A graphic of the first assembly mesh is shown as generated by the mesh generator (vertex up). The connection between the assembly pattern and the mesh can be seen more easily by rotating this figure 90 degrees in the clockwise direction.

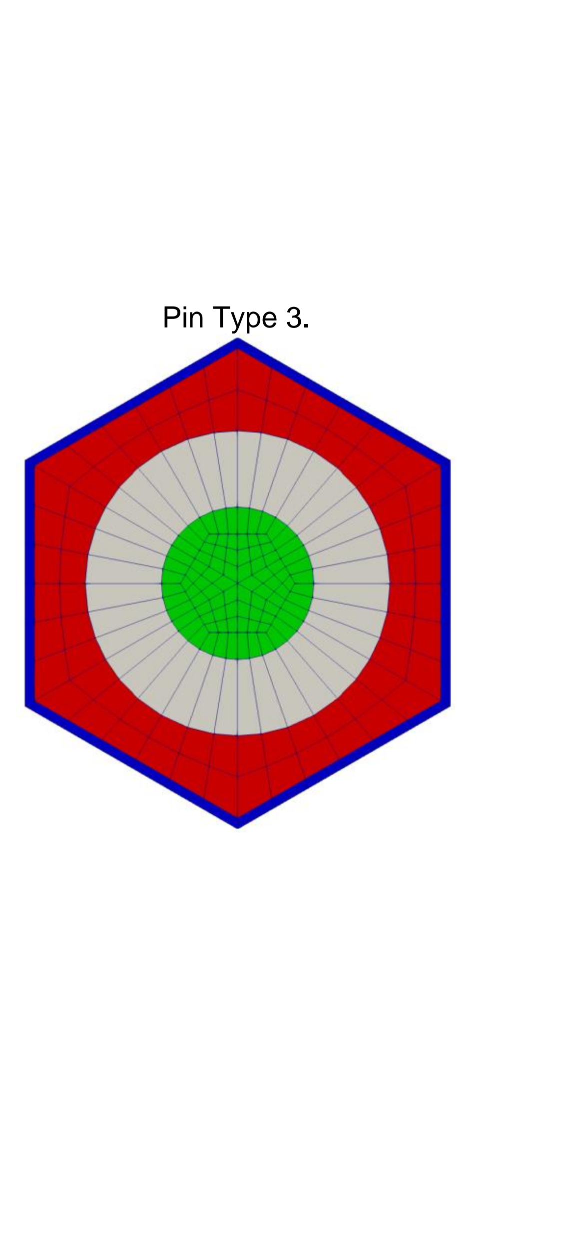
The third row of the figure depicts the creation of the core including axial extrusion. Note the ability to slot in dummy assemblies which the code generates and deletes automatically. At the end of this step, a 3D mesh exists with material IDs assigned on each subdomain, as well as reporting IDs and sidesets for later boundary condition assignment.

```
[Mesh]
  [./gmp]
    type = GlobalMeshParams
    dim = 3
    geom = "Hex"
    assembly_pitch = 7.10315
    axial_regions = '5.0 20.0 5.0'
    axial_mesh_intervals = '2 2 2'
  []

  [./pin1]
    type = PinMeshGenerator
    global_params = gmp
    pin_type = 1
    pitch = 1.42063
    num_sectors = 6
    ring_radii = '0.3818'
    region_ids = '1 2;
                 11 12;
                  1 2'
    mesh_intervals = '1 1'
  []

  [./pin2]
    type = PinMeshGenerator
    global_params = gmp
    pin_type = 2
    pitch = 1.42063
    num_sectors = 6
    region_ids = '3;
                 13;
                  3'
    mesh_intervals = '1'
  []

  [./pin3]
    type = PinMeshGenerator
    global_params = gmp
    pin_type = 3
    pitch = 1.42063
    num_sectors = 6
    ring_radii = '0.254 0.508'
    duct_radii = '0.68'
    region_ids = ' 4  1  2  5;
                  14 11 12 15;
                   4  1  2  5'
    mesh_intervals = '1 1 2 1'
  []
```

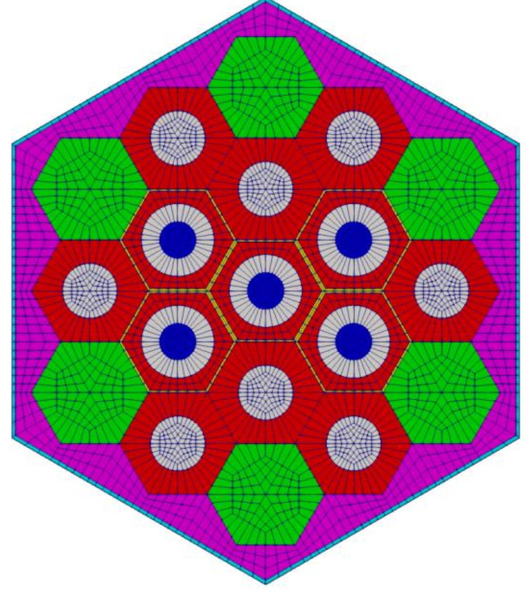


```
[./amg]
   type = AssemblyMeshGenerator
   assembly_type = 1
   inputs = 'pin1 pin2 pin3'
   pattern = ' 1 0 1 ;
              0 2 2 0;
             1 0 2 0 1;
              0 2 2 0;
               1 0 1'
   background_intervals = 1
   background_region_id = '30
                           31
                           30 '
   duct_radii = '3.5'
   duct_intervals = '1'
   duct_region_ids = '40;
                      41;
                      40'
 []

 [./amg1]
   type = AssemblyMeshGenerator
   assembly_type = 2
   inputs = 'pin1 pin2 pin3'
   pattern = ' 1 1 1 ;
              1 1 1 1;
             1 1 1 1 1;
              1 1 1 1;
               1 1 1'
   background_intervals = 1
   background_region_id = '30
                           31
                           30 '
   duct_radii = '3.5'
   duct_intervals = '1'
   duct_region_ids = '40;
                      41;
                      40'
 []

 [./amg2]
   type = AssemblyMeshGenerator
   assembly_type = 3
   inputs = 'pin1 pin2 pin3'
   pattern = '1 1 1;
             1 0 0 1;
            1 0 2 0 1;
             1 0 0 1;
              1 1 1'
   background_intervals = 2
   background_region_id = '30
                           31
                           30 '
   duct_radii = '3.5'
   duct_intervals = '1'
   duct_region_ids = '40;
                      41;
                      40'
 []
```

Assembly Type 1
(to more easily visualize the pattern, rotate clockwise by 90 degrees)



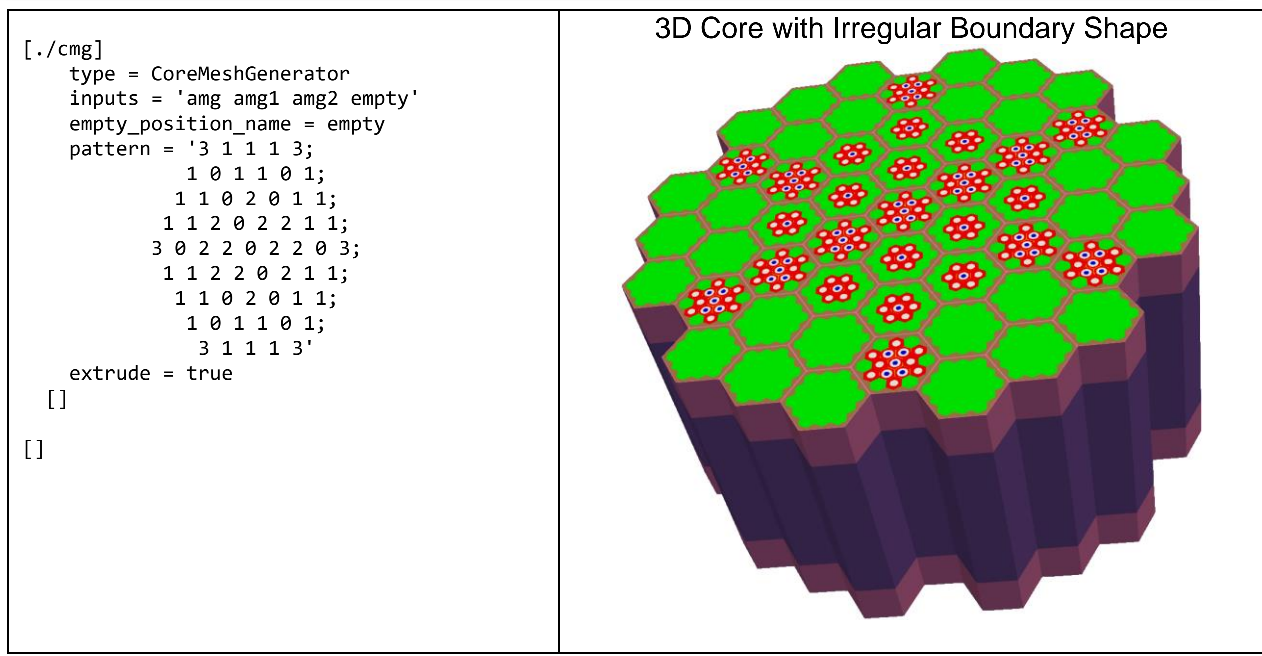


Figure 7-2. 3D Hexagonal core input and mesh built using MOOSE's new reactor geometry mesh builder.

7.5 Limitations and Recommendations

The RGMB capability in its current form lacks some capabilities that are believed to be needed in the future. The most important of these limitations is that it currently does not perform conformal meshing checks or fix an attempt to stitch non-conformal meshes. The responsibility is on the user to ensure pins and assemblies can be stitched together. Generally, non-conformal mesh stitching is an issue when assemblies with different numbers of pins neighbor each other, but the issue can be avoided if the user adjusts meshing interval settings in the constituent assembly meshes. Regardless, this has been identified as an important area of future work. Another limitation is the asymmetric capability of assembly ducts. While they are only currently an option for hexagonal assemblies, the capability should be added to accommodate assembly ducts in Cartesian assemblies as well. However, this would require the development of an unrelated MeshGenerator for concentric squares and the focus of this work is on the application to fast reactor geometries. Lastly, neither the `PinMeshGenerator` nor the `AssemblyMeshGenerator` currently work with `SimpleHexagonGenerator` which may be preferred for defining simple cores.

Additional upgrades needed include the ability to include control drums in the core, the ability to collapse subdomain IDs at the end automatically (not needed once materials have been assigned), and development of a consistent naming/numbering scheme for sidesets. The sidesets here are generated only on the external boundaries, and these may not be sufficient for all physics calculations.

Depending on user feedback, some input names may be hidden in the future to streamline user input and other, currently hidden, inputs may be added as optional parameters to enable greater user control. The initial version only exposes the options needed to create symmetric pins and stitched meshes for both geometries and handles the advanced and optional inputs internally.

8 Miscellaneous Enhancements

Aside from the systematic improvements in MOOSE mesh generators and related objects as discusses in the other chapters in this report, some miscellaneous updates have also been made to enhance the functionalities of existing MOOSE mesh generators and to provide some utility tools to help users to better use the meshing related features in MOOSE. These updates are summarized in this chapter.

8.1 Updating of FancyExtruderGenerator

During the development of the control drum related objects and the reporting ID features, it was found that the original version of `FancyExtruderGenerator` did not retain element extra integers and subdomain/nodeset/sideset names during the extrusion procedure. Consequently, the information assigned to the 2D mesh is lost if the user employed `FancyExtruderGenerator` to extrude into 3D. In addition, `FancyExtruderGenerator` has a useful functionality called “swap”, which allows the users to reassign subdomain IDs for different extruded layers. It is necessary to expand this “swap” feature to the element extra integers.

The `FancyExtruderGenerator` was updated by adding the following new features:

- Retain all existing element extra integers of the input mesh during extrusion.
- Retain all existing subdomain, nodeset and sideset name maps of the input mesh during extrusion.
- Enable element extra integer "swap", which is similar to the existing “subdomain_swaps”.

This update enables the implementation of a series of new features developed in this project that are discussed in other chapters of this report, including reporting ID and control drum meshing and manipulation. It also generally enhances the functionalities of `FancyExtruderGenerator` for other applications.

8.2 Element Centroid Locater (2D 1st Order Elements)

During the mesh generator development, sometimes it is necessary to determine the center of mass (i.e., centroid) of a mesh to help calculate other geometry parameters. Generally, the centroid coordinate of a mesh can be calculated by means of averaging the centroid coordinates of all the elements using element volumes (areas for 2D meshes) as weights. That is,

$$\vec{x}_{centroid}^{mesh} = \frac{1}{N}\sum_{i=1}^{N} \vec{x}_{centroid}^{elem} \cdot V_{elem} \qquad \text{Eq. 8-1}$$

In libMesh, both centroid and volume of an element are defined as member functions of the `Elem` class: libMesh::Elem::volume() and libMesh::Elem::centroid(). Theoretically, it should be straightforward to apply Eq. 8-1 using existing member functions. However, the libMesh::Elem::centroid() does not actually calculate an element’s center of mass. Instead, it only calculates the average of all vertices of that element. This approach can calculate the centroid of a

triangular element (2D) or a tetrahedron element (3D) correctly. For other type of elements, the centers of mass must be calculated using a different algorithm.

For a quadrilateral element, the real centroid coordinate can be calculated using the following steps:

(1) Divide the quadrilateral element into two triangles using one of its diagonals and find the centroids of these two triangles using the arithmetic average of the three vertices (labelled as points A and B).

(2) Divide the quadrilateral element into another two triangles using the other diagonal and find the centroids of these two triangles using the arithmetic average of the three vertices (labelled as points C and D).

(3) The intersect of the line segments AB and CD is the centroid of the quadrilateral element.

This algorithm has been implemented in the polygon mesh generator classes developed in this project and can be used by other MOOSE programmers.

More generally, the following approaches can be used to calculate centroids of different types of elements, which adopts Eq. 8-1 to expand the centroid algorithm of simple geometry element to that of the complex ones (see Table 8-1).

Table 8-1. Generalized algorithm to calculate element centroids

Element Type	Centroid	Volume
TRI	$(\vec{p}_0 + \vec{p}_1 + \vec{p}_2)/3$	$\lvert(\vec{p}_1 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)\rvert/2$
QUAD	Eq. 8-1 for two TRIs	Summation of two TRIs
TET	$(\vec{p}_0 + \vec{p}_1 + \vec{p}_2 + \vec{p}_3)/4$	$\lvert(\vec{p}_0 - \vec{p}_3) \cdot [(\vec{p}_1 - \vec{p}_3) \times (\vec{p}_2 - \vec{p}_4)]\rvert/6$
PYRAMID	Eq. 8-1 for two TETs	Summation of two TETs
PRISM	Eq. 8-1 for three TETs	Summation of three TETs
HEX	Eq. 8-1 for two PRISMs	Summation of two PRISMs

Generalized member functions MOOSE::PolygonGeneratorBase::centroid_calculator() and MOOSE::PolygonGeneratorBase::volume_calculator() have been developed in a local repository and may be merged into MOOSE after testing and optimization.

9 Physics Code Verification of Developed Capabilities

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. Results were compared to pre-existing results that used externally generated meshes from either CUBIT or Argonne's Mesh Tools system.

9.1 Griffin Verification: Homogenized Fast Reactor Example

The first example to demonstrate the functionality of the new MOOSE mesh tools involves simulation of the Advanced Burner Test Reactor (ABTR) model (Shemon, Grudzinski, Lee, Thomas, & Yu, 2015). This full-core reactor model features a 3-D core with 10 rings of hexagonal assemblies, where each assembly is discretized into 12 axial layers. Figure 9-1 illustrates the top-down and side view of reactor geometry. Each hexagonal prismatic region in the assembly is further discretized into six triangular regions with equivalent block ids, and all reactor neutronics properties are homogenized over these triangular prism elements. Furthermore, several assemblies in the outermost ring have been deleted from the core. The exact core geometry specifications are based on a mesh that was previously created by Argonne's Mesh Tools system for neutronic analysis of the ABTR problem. The goal of this section is to show close agreement in Griffin neutronics results when the input mesh is defined by the new MOOSE mesh tools when compared to using a mesh generated by Argonne's Mesh Tools system. Section 9.1.1 discusses the exact mesh generation process in MOOSE for this problem, and Section 9.1.2 defines relevant parameters used in the Griffin neutronics simulation and compares the simulation results between using MOOSE mesh tools and Argonne's mesh tools as the input mesh. The Griffin input files discussed in this section and in Section 9.2 are not included in this report but can be made available upon request. Moreover, there are plans currently to include these input files into the main Griffin source code repository as examples of how the mesh tools described in this report can be used to define geometry meshes for representative reactor problems.

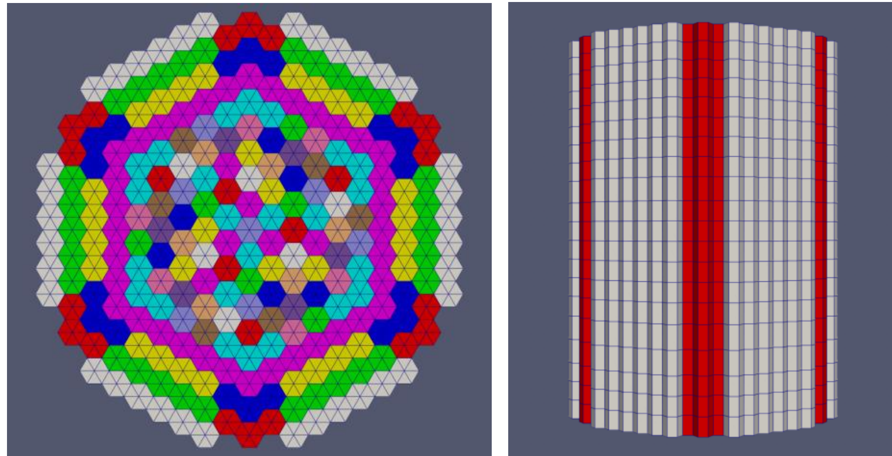


Figure 9-1. Top-down (left) and side (right) views of ABTR homogeneous model, generated by Argonne's Mesh Tools

9.1.1 ABTR Mesh Generation in New MOOSE Mesh Tools

The general steps for producing the input mesh for the ABTR problem in the new MOOSE mesh tools can be summarized as follows:

1. Define each unique 2-D assembly
2. Define layout of assemblies in 2-D and dummy assembly deletion
3. Extrude 2-D geometry to 3-D
4. Define core sidesets

Note that this example was prepared prior to availability of the Reactor Geometry Mesh Builder (which also does not currently permit usage of the `SimpleHexagonGenerator` mesh object used here.) In step 1, each 2-D assembly with a unique block id is defined using the `SimpleHexagonGenerator` type, which discretizes each assembly in the core into 6 triangular prisms. Here, the block id of each assembly and the size of the hexagon is defined, and the dummy assemblies which will be removed from the core layout are also defined in this section. An arbitrary block id is given to the dummy assembly, which will be the input in step 2 to specify that assemblies with this block id should be removed from the core layout. The code snippet for step 1 is shown below, and each of the code blocks presented is contained within the overarching `[Mesh]` block. For brevity, only two assemblies definitions are shown, but this step is repeated for each assembly with a unique block id in the 2-D assembly layout.

```
[assembly1]
  type = SimpleHexagonGenerator
  hexagon_size = 7.3425
  hexagon_size_style = 'apothem'
  block_id = '1'
[]
[assembly2]
  type = SimpleHexagonGenerator
  hexagon_size = 7.3425
  hexagon_size_style = 'apothem'
  block_id = '2'
[]
... # Repeat for all other assemblies
[dummy]
  type = SimpleHexagonGenerator
  hexagon_size = 7.3425
  hexagon_size_style = 'apothem'
  block_id = '997'
[]
```

Figure 9-2. ABTR 2-D Assembly Definition Using SimpleHexagonGenerator

In step 2, the core layout of the assemblies provided in step 1 is defined using the `PatternedHexMeshGenerator` type, and the code snippet for this step is shown in Figure 9-3. Here, the outer sideset boundary of the core is defined as *"core_out"*, and the layout is specified using the *pattern* parameter. It should be noted here that the values used to define the 2-D assembly layout correspond to the index order of the assemblies defined in the *inputs* parameter. The exact core layout of assemblies for this problem are chosen to mimic the layout of assemblies in the mesh

created by Argonne's Mesh Tools, but this is by no means the only way that the layout of assemblies can be defined. The current implementation of MOOSE mesh tools allows for a flexible definition of the core layout and provides a straightforward way to re-shuffle these assemblies simply by changing the value of the *pattern* block.

```
[core_lattice]
  type = PatternedHexMeshGenerator
  inputs = 'assembly31 assembly29 assembly30 assembly24 assembly25
            assembly26 assembly28 assembly16 assembly17 assembly18
            assembly19 assembly27 assembly23 assembly9  assembly10
            assembly11 assembly12 assembly13 assembly20 assembly22
            assembly15 assembly6  assembly7  assembly8  assembly21
            assembly14 assembly5  assembly3  assembly4  assembly1
            assembly2  assembly0  dummy'
  pattern_boundary = none
  external_boundary_name = core_out
  generate_core_metadata = false
  generate_control_drum_positions_file = false
  pattern =      '32  32  31  31  31  31  31  31  32  32;
                 32  29  29  30  30  30  30  30  29  29  32;
               31  29  27  27  28  28  28  28  27  27  29  31;
             31  30  27  26  26  26  26  26  26  26  27  30  31;
           31  30  28  26  21  21  25  24  21  21  26  28  30  31;
         31  30  28  26  21  13  20  19  18  17  21  26  28  30  31;
       31  30  28  26  22  14   7  12  11  10  16  23  26  28  30  31;
     31  30  28  26  23  15   8   3   6   5   9  15  22  26  28  30  31;
   32  29  27  26  21  16   9   4   1   2   4   8  14  21  26  27  29  32;
 32  29  27  26  21  17  10   5   2   0   1   3   7  13  21  26  27  29  32;
   32  29  27  26  21  18  11   6   1   2   6  12  20  21  26  27  29  32;
     31  30  28  26  24  19  12   3   4   5  11  19  25  26  28  30  31;
       31  30  28  26  25  20   7   8   9  10  18  24  26  28  30  31;
         31  30  28  26  21  13  14  15  16  17  21  26  28  30  31;
           31  30  28  26  21  21  22  23  21  21  26  28  30  31;
             31  30  27  26  26  26  26  26  26  26  27  30  31;
               31  29  27  27  28  28  28  28  27  27  29  31;
                 32  29  29  30  30  30  30  30  29  29  32;
                   32  32  31  31  31  31  31  31  32  32'
[]
[del_dummy]
  type = BlockDeletionGenerator
  block = 997
  input = core_lattice
  new_boundary = core_out
[]
```

Figure 9-3. ABTR 2-D Core Lattice Definition and Dummy Deletion

In step 3, the `FancyExtruderGenerator` type is used to extrude the 2-D core into 3-D and specify the core axial discretizations. Here, *heights* is used to indicate where axial discretizations occur in the z-plane, and *num_layers* defines the number of axial subdivisions for each layer. *top_boundary* and *bottom_boundary* represent the sideset ids given to top and bottom sidesets that result from the

3-D extrusion, and these ids will be used to define the core outer boundaries in Griffin along with *core_out* in step 2. Finally, *subdomain_swaps* is used to set the block ids for each layer, since the typical extrusion process preserves the same block ids from 2-D into each layer in 3-D. These block ids are set in this specific manner to match the block ids set by Argonne's Mesh Tools for this problem, but once again these block ids can be defined in 3-D with a more ordered numbering. Figure 9-4 shows the code snippet for step 3.

```
[extrude]
  type = FancyExtruderGenerator
  input = del_dummy
  heights = '50.24 42.32 17.98 16.88 16.88 16.88 16.89 16.88 19.76 65.66 31.14 30.15'
  num_layers = '3 2 1 1 1 1 1 1 1 4 2 2'
  direction = '0 0 1'
  top_boundary = 998
  bottom_boundary = 999
  subdomain_swaps = ' 31  31;
                      31  61  29  59  30  60  24  54  25  55  26  56  28  58  16  46  17  47
                      18  48  19  49  27  57  23  53   9  39. 10  40  11  41  12  42  13  43
                      20  50  22  52  15  45   6  36   7  37   8  38  21  51  14  44   5  35
                       3  33   4  34   2  32;
                      31  65  29  59  30  60  24  54  25  64  26  56  28  58  16  46  17  47
                      18  48  19  49  27  57  23  53   9  39  10  40  11  62  12  42  13  43
                      20  50  22  63  15  45   6  36   7  37   8  38  21  51  14  44   5  35
                       3  33   4  34   2  32;
                      31  65  29  84  30  85  24  80  25  64  26  81  28  83  16  74  17  75
                      18  76  19  49  27  82  23  79   9  68  10  69  11  62  12  70  13  71
                      20  77  22  63  15  73   6  36   7  66   8  67  21  78  14  72   5  35
                       3  33   4  34   2  32;
                      31  65  29 104  30 105  24 100  25  64  26 101  28 103  16  94  17  95
                      18  96  19  49  27 102  23  99   9  88  10  89  11  62  12  90  13  91
                      20  97  22  63  15  93   6  36   7  86   8  87  21  98  14  92   5  35
                       3  33   4  34   2  32;
                      31  65  29 124  30 125  24 120  25  64  26 121  28 123  16 114  17 115
                      18 116  19  49  27 122  23 119   9 108  10 109  11  62  12 110  13 111
                      20 117  22  63  15 113   6  36   7 106   8 107  21 118  14 112   5  35
                       3  33   4  34   2  32;
                      31  65  29 144  30 145  24 140  25  64  26 141  28 143  16 134  17 135
                      18 136  19  49  27 142  23 139   9 128  10 129  11  62  12 130  13 131
                      20 137  22  63  15 133   6  36   7 126   8 127  21 138  14 132   5  35
                       3  33   4  34   2  32;
                      31 169  29 167  30 168  24 162  25 163  26 164  28 166  16 155  17 156
                      18 157  19  49  27 165  23 161   9 148  10 149  11 150  12 151  13 152
                      20 158  22 160  15 154   6  36   7 146   8 147  21 159  14 153   5  35
                       3  33   4  34   2  32;
                      31 193  29 191  30 192  24 186  25 187  26 188  28 190  16 179  17 180
                      18 181  19  49  27 189  23 185   9 172  10 173  11 174  12 175  13 176
                      20 182  22 184  15 178   6  36   7 170   8 171  21 183  14 177   5  35
                       3  33   4  34   2  32;
                      31 193  29 212  30 213  24 208  25 187  26 209  28 211  16 202  17 203
                      18 204  19  49  27 210  23 207   9 196  10 197  11 174  12 198  13 199
                      20 205  22 184  15 201   6  36   7 194   8 195  21 206  14 200   5  35
                       3  33   4  34   2  32;
                      31 217  29 212  30 213  24 208  25 216  26 209  28 211  16 202  17 203
                      18 204  19  49  27 210  23 207   9 196  10 197  11 214  12 198  13 199
                      20 205  22 215  15 201   6  36   7 194   8 195  21 206  14 200   5  35
                       3  33   4  34   2  32;
                      31 247  29 245  30 246  24 240  25 241  26 242  28 244  16 232  17 233
                      18 234  19 235  27 243  23 239   9 225  10 226  11 227  12 228  13 229
                      20 236  22 238  15 231   6 222   7 223   8 224  21 237  14 230   5 221
                       3 219   4 220   2 218'
[]
```

Figure 9-4. ABTR 3-D Extrusion Process

Finally, in step 4, the boundary sidesets are renamed as a single name "VOID", in order to define the core periphery as a single entity. This name can be used in Griffin directly to specify the problem boundary conditions. Moreover, this avoids the need to track sideset ids directly from Exodus files, which can be a cumbersome process especially in Argonne's Mesh Tools where boundary sideset id assignments can occur in a non-intuitive manner. Figure 9-5 shows the code snippet for step 4.

```
[rename_sidesets]
  type = RenameBoundaryGenerator
  input = extrude
  old_boundary = 'core_out 998 999'
  new_boundary = 'VOID VOID VOID'
[]
```

Figure 9-5. ABTR Sideset Renaming Process

To illustrate the differences in how input files are laid out when the new MOOSE mesh tools and Argonne's Mesh Tools are used respectively, Figure 9-6 shows the `[Mesh]` block for these two frameworks. In the case of MOOSE mesh tools, the mesh block is defined using internal MOOSE mesh generators, while the mesh must be imported as a *FileMeshGenerator* from an Exodus file when using Argonne's Mesh Tools. In this light, any changes to the underlying mesh when running downstream MOOSE App calculations can be made in a single step by directly modifying the input file when MOOSE mesh tools are used. On the other hand, when external mesh generator platforms such as Argonne's Mesh Tools are used, changes to the underlying mesh are conducted in three steps – first, by running the mesh generator application to create a PROTEUS-formatted mesh file, second, converting to Exodus file format, and third, by running the MOOSE App on the updated Exodus file.

```
[Mesh]
  [assembly1]
    # From Figure 9-2
  []
  [assembly2]
    # From Figure 9-2
  []
  ...
  [dummy]
    # From Figure 9-2
  []
  [core]
    # From Figure 9-3
  []
  [del_dummy]
    # From Figure 9-3
  []
  [extrude]
    # From Figure 9-4
  []
  [rename_sidesets]
    # From Figure 9-5
  []
[]
```

```
[Mesh]
  [fmg]
    type = FileMeshGenerator
    file = mesh.e
    # mesh.e is produced externally in Argonne Mesh Tools with a separate syntax, not shown here. Conversion utilities are required to convert the AMT mesh to Exodus format to import here.
  []
[]
```

Figure 9-6. MOOSE Mesh block when using MOOSE mesh tools (left) vs. Argonne's Mesh Tools (right)

9.1.2 Griffin Neutronics Parameters for ABTR Problem and Comparison of Neutronics Results between New MOOSE Mesh Tools and Argonne's Mesh Tools

The Griffin code (Lee, et al., 2021) is used to simulate the neutronic behavior for the ABTR problem, and both the new MOOSE mesh tools and Argonne's Mesh Tools are used for mesh generation in order to show that downstream calculations yield comparable results between these two frameworks for mesh generation. Since Griffin is a MOOSE-based application, the Mesh blocks defined in Figure 9-6 can be used directly in the Griffin input file. All other input blocks that specify the neutronics problem parameters are kept the same between the two Griffin simulation runs. More precisely, 33 energy groups are used to solve the steady-state neutronics problem with a solver scheme that utilizes the self-adjoint angular flux (SAAF) formulation with continuous finite element method (CFEM) and the discrete ordinates (S_N) method. Diffusion synthetic acceleration (DSA) with identical mesh and energy discretization as the transport solver is used to accelerate problem convergence, and vacuum boundary conditions are applied to the core periphery. 15 neutronics materials are present in the ABTR problem. Since these material assignments are made in the Griffin input according to the block ids of the mesh elements, the block id assignments in the mesh based on MOOSE mesh tools are made to be identical to the block id assignments implemented by Argonne's Mesh Tools. This ensures that the material assignment block can be kept the same between the Griffin input files for these two mesh generation frameworks.

K-effective is used as a metric for comparing neutronics results between the MOOSE-based and Argonne Mesh Tools-based frameworks for mesh generation. Table 9-1 summarizes the results between these two frameworks and it can be seen that the two cases produce identical k-effective values irrespective of the tool used to generate the input mesh. These results indicate that the two mesh generation frameworks produce identical mesh discretizations for the homogeneous 3-D full-core problem.

Table 9-1. Griffin k-effective results between new MOOSE mesh tools and Argonne's Mesh Tools for the ABTR Problem

Mesh Generation Tool	Griffin Solver Scheme	K-Effective
New MOOSE Mesh Tools	SAAF-CFEM-SN with DSA Diffusion Acceleration	1.036481
Argonne's Mesh Tools	SAAF-CFEM-SN with DSA Diffusion Acceleration	1.036481

9.2 Griffin Verification: Heterogeneous Fast Reactor Assembly Example

Section 9.1 showed that equivalent neutronics results were achieved between Argonne's Mesh Tools and the new MOOSE mesh generators when defining the mesh for a homogeneous 3-D full core problem with a hexagonal assembly lattice. In this section, neutronics verification is extended to the 3-D heterogeneous assembly problem in order to show how the MOOSE framework meshing enhancements can be applied to problems with circular pincell regions. Here, the assembly geometry is based on a candidate lead-cooled fast reactor (LFR) assembly design (Grasso, Levinsky, Franceschini, & Ferroni, 2019), which features a seven-ring pin lattice with an assembly length of 353.42 cm and an axial discretization with 10 unique layers. Figure 9-7 shows the top-down and side views of the mesh that was previously created with Argonne's Mesh Tools for this problem (Shemon, Yu, & Kim, 2020). Once again, the goal for this section is to replicate this mesh as closely as possible with the internal MOOSE mesh generators described in this report, and to show comparable neutronics results between traditional frameworks and new MOOSE-based frameworks for mesh generation. Section 9.2.1 defines the necessary steps in the MOOSE input file to generate the LFR assembly mesh, while Section 9.2.2 summarizes the neutronics results between using the new MOOSE mesh tools and Argonne's Mesh Tools as the input mesh. Finally, Section 9.2.3 describes how the new MOOSE-based meshing capabilities can additionally be implemented to define a separate coarse mesh that can be used for diffusion acceleration in neutronics problems. This example was again developed prior to availability of the RGMB capability which could streamline the mesh generation procedure shown here.

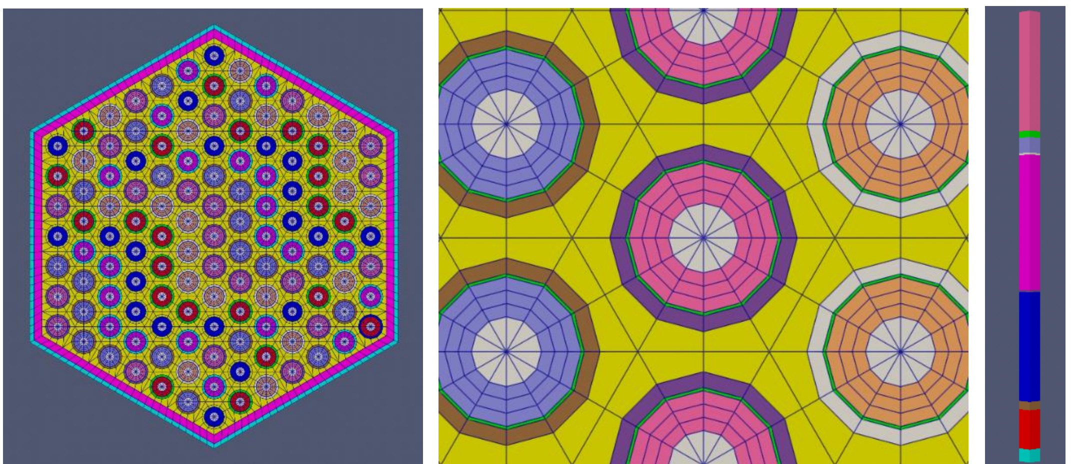


Figure 9-7. Top-down view of entire assembly (left), zoomed top-down view of central pins (middle), and side view (right) of the LFR heterogeneous assembly model, generated by Argonne's Mesh Tools

9.2.1 LFR Mesh Generation in New MOOSE Mesh Tools

The mesh generation process for the LFR problem follows a similar procedure as described in Section 9.1.1, where the lattice pattern is used to define pins instead of assemblies in a ring pattern. The following steps are taken to generate the LFR mesh using MOOSE mesh generators:

1. Define each unique 2-D pin
2. Define layout of pins in 2-D
3. Extrude 2-D geometry to 3-D
4. Define assembly sidesets

In step 1, each unique 2-D pin is defined using the `PolygonConcentricCircleMeshGenerator` type, which discretizes the pin cell into a hexagon with a single pin and background region. The code snippet for this step is shown in Figure 9-8, and is repeated for each pin in the problem with a unique block id. For reference, Table 9-2 includes the definition of each of the input parameters in the pin blocks. For this problem, ring and background block ids are set to match the block ids in the mesh created by Argonne's Mesh Tools. Defining numerous identical pins with different block numbers is necessary in this case because this mesh was intended for use in hot channel factor calculations whereby individual pin properties (e.g. materials) could be perturbed.

```
[pin001]
  type = PolygonConcentricCircleMeshGenerator
  num_sides = 6
  num_sectors_per_side = '2 2 2 2 2 2'
  polygon_size = 0.67125
  ring_radii = '0.2020 0.4319 0.4495 0.5404'
  ring_intervals = '1 3 1 1'
  ring_block_ids = '0 129 2 130'
  background_intervals = 1
  background_block_ids = 4
  preserve_volumes = on
[]
[pin002]
 type = PolygonConcentricCircleMeshGenerator
  num_sides = 6
  num_sectors_per_side = '2 2 2 2 2 2'
  polygon_size = 0.67125
  ring_radii = '0.2020 0.4319 0.4495 0.5404'
  ring_intervals = '1 3 1 1'
  ring_block_ids = '0 131 2 132'
  background_intervals = 1
  background_block_ids = 4
  preserve_volumes = on
[]
... # Repeat for all other pins
```

Figure 9-8. LFR 2-D Pin Definition Using PolygonConcentricCircleMeshGenerator

Table 9-2. Description of Parameters Used in Input Block for Pin Definitions in Figure 9-8

Input Parameter	Description
num_sides	Number of sides in background polygon shape
num_sectors_per_side	Number of sectors for each side of background polygon
polygon_size	Size of background polygon region
ring_radii	Radii of rings in circular pin region
ring_intervals	Number of subdivisions per ring in circular pin region
ring_block_ids	Block id's for each ring in circular pin region
background_intervals	Number of subdivisions in background region
background_block_ids	Block id's for each background region subdivision
preserve_volumes	Whether or not to preserve volume during pin discretization

Similar to the ABTR problem in Section 9.1.1, in step 2 the pin layout of the LFR assembly is defined using the `PatternedHexMeshGenerator` type. The code snippet for this step is shown in Figure 9-9. Since a duct region is present in this problem, duct properties are also defined with the

parameters *duct_sizes*, *duct_block_ids*, and *duct_intervals*. Once again, the exact pin layout for this problem is chosen to mimic the layout of pins in the mesh created by Argonne's Mesh Tools.

```
[assembly_lattice]
  type = PatternedHexMeshGenerator
  inputs = 'pin001 pin002 pin003 pin004 pin005 pin006 pin007 pin008 pin009
            pin010 pin011 pin012 pin013 pin014 pin015 pin016 pin017 pin018
            pin019 pin020 pin021 pin022 pin023 pin024 pin025 pin026 pin027
            pin028 pin029 pin030 pin031 pin032 pin033 pin034 pin035 pin036
            pin037 pin038 pin039 pin040 pin041 pin042 pin043 pin044 pin045
            pin046 pin047 pin048 pin049 pin050 pin051 pin052 pin053 pin054
            pin055 pin056 pin057 pin058 pin059 pin060 pin061 pin062 pin063
            pin064 pin065 pin066 pin067 pin068 pin069 pin070 pin071 pin072
            pin073 pin074 pin075 pin076 pin077 pin078 pin079 pin080 pin081
            pin082 pin083 pin084 pin085 pin086 pin087 pin088 pin089 pin090
            pin091 pin092 pin093 pin094 pin095 pin096 pin097 pin098 pin099
            pin100 pin101 pin102 pin103 pin104 pin105 pin106 pin107 pin108
            pin109 pin110 pin111 pin112 pin113 pin114 pin115 pin116 pin117
            pin118 pin119 pin120 pin121 pin122 pin123 pin124 pin125 pin126
            pin127'
  pattern_boundary = hexagon
  background_intervals = 1
  hexagon_size = 8.20825
  duct_sizes = '7.6712 8.0245'
  duct_sizes_style = apothem
  duct_intervals = '1 1'
  background_block_id = 4
  duct_block_ids = '257 258'
  external_boundary_id = 997
  pattern = '103  102  101  100   99   98   97;
           104   71   70   69   68   67   66   96;
         105   72   45   44   43   42   41   65   95;
       106   73   46   25   24   23   22   40   64   94;
     107   74   47   26   11   10    9   21   39   63   93;
   108   75   48   27   12    3    2    8   20   38   62   92;
 109   76   49   28   13    4    0    1    7   19   37   61   91;
   110   77   50   29   14    5    6   18   36   60   90  126;
     111   78   51   30   15   16   17   35   59   89  125;
       112   79   52   31   32   33   34   58   88  124;
         113   80   53   54   55   56   57   87  123;
           114   81   82   83   84   85   86  122;
             115  116  117  118  119  120  121'
[]
```

Figure 9-9. LFR Assembly 2-D Pin Lattice Definition

In step 3, the `FancyExtruderGenerator` type is once again used to extrude the 2-D core into 3-D and specify the core axial discretizations. The extrusion process is identical to step 2 in Section 9.1.1. For brevity, the entire input to *subdomain_swaps* is not shown. However, for this problem, each of the 259 block ids in each 2-D axial layer is remapped to a unique value so as to create the same block id mapping with that of the mesh created by Argonne's Mesh Tools.

```
[extrude]
  type = FancyExtruderGenerator
  input = assembly_lattice
  heights = '10.07 30.79 6.56 85.85 1.52 106.07 1.51 12.13 5.05 93.87'
  num_layers = '1 3 1 9 1 20 1 2 1 9'
  direction = '0 0 1'
  top_boundary = 998
  bottom_boundary = 999
  subdomain_swaps = '0 0;
                     0 259  1 260  2 261 ...  ;
                     0 518  1 519  2 520 ...  ;
                     0 777  1 778  2 779 ...  ;
                     0 1036 1 1037 2 1038 ... ;
                     0 1295 1 1296 2 1297 ... ;
                     0 1554 1 1555 2 1556 ... ;
                     0 1813 1 1814 2 1815 ... ;
                     0 2072 1 2073 2 2074 ... ;
                     0 2331 1 2332 2 2333 ... ;
[]
```

Figure 9-10. LFR Assembly 3-D Extrusion Process

Finally, in step 4, the boundary sidesets are renamed so that the top, bottom, and side sidesets are are renamed to ASSEMBLY_TOP, ASSEMBLY_BOTTOM, and ASSEMBLY_SIDE respectively, as shown in Figure 9-11. These sideset names will be inputted directly into Griffin to define boundary conditions for the neutronics problem.

```
[rename_sidesets]
  type = RenameBoundaryGenerator
  input = extrude
  old_boundary = '998 999 997'
  new_boundary = 'ASSEMBLY_TOP ASSEMBLY_BOTTOM ASSEMBLY_SIDE'
[]
```

Figure 9-11. LFR Assembly Sideset Renaming Process

9.2.2 Comparison of Griffin Neutronics Results between New MOOSE Mesh Tools and Argonne's Mesh Tools for LFR problem

Similar to Section 9.1.2, neutronics results for the LFR problem are compiled from the Griffin code and summarized in Table 9-3. For this problem, 9 energy groups are used to solve the steady-state neutronics problem with a solver scheme that utilizes the discontinuous finite element method (DFEM) with the discrete ordinates (S_N) method. 33 neutronics materials are present in the LFR problem, and vacuum boundary conditions are applied to the top and bottom surfaces of the assembly while reflecting boundary conditions are applied to the outer radial surfaces. Diffusion acceleration is not leveraged for the simulations in this section but will be the topic of exploration for Section 9.2.3. Both Argonne's Mesh Tools and the new MOOSE mesh tools are used to define the input mesh to Griffin. Table 9-3 highlights a 20pcm difference in the computed Griffin eigenvalue between these two meshing frameworks. This slight discrepancy is due to the fact that

the mesh discretization scheme used in the background region between the outermost pins and the duct differs between Argonne's Mesh Tools and the new MOOSE mesh tools, and this disparity can be seen by comparing the yellow background regions of the plots in Figure 9-12.

Table 9-3. Griffin k-effective results between new MOOSE mesh tools and Argonne's Mesh Tools for the LFR Assembly Problem

Griffin Solver Scheme	New MOOSE Mesh Tools K-Effective	Argonne's Mesh Tools K-Effective	K-Effective Difference (pcm)
Direct DFEM-SN	1.17139	1.17119	20

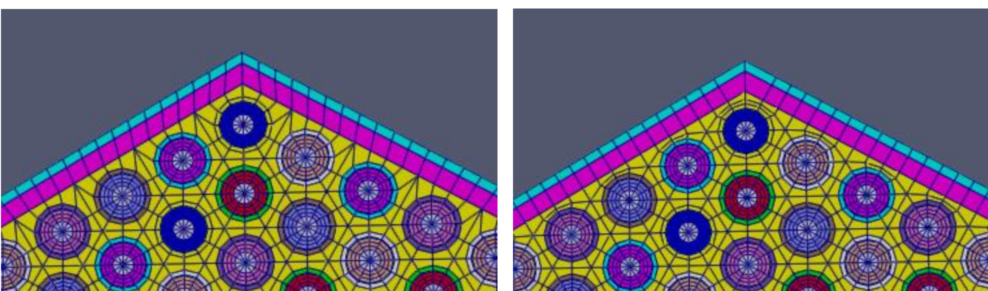


Figure 9-12. Zoomed top-down view of background region between outermost pins and duct for LFR heterogeneous assembly model, generated by Argonne's Mesh Tools (left) and new MOOSE mesh tools (right)

9.2.3 Extension of New MOOSE Meshing Capabilities to LFR Problem with Coarse Mesh Diffusion Acceleration

A further extension of the new MOOSE meshing capabilities is to define the coarse mesh layout for neutronics diffusion acceleration schemes. Instead of importing a separate mesh to use for diffusion acceleration, the new MOOSE mesh generator framework allows for the coarse mesh to be defined directly in the `[Mesh]` block of the diffusion acceleration input. Figure 9-13 depicts the fine and coarse mesh constructed by the new MOOSE mesh generators for the LFR problem that are then passed to Griffin in order to solve full neutronics transport with diffusion acceleration.

The procedure for coarse mesh input generation mirrors that of the fine mesh input generation described in Section 9.2.1. However, the coarse mesh is homogenized over each hexagonal pin region, so the pin definition described in step 1 of Section 9.2.1 is modified to reflect this change. Moreover, given the irregularity of the background discretization of the outermost pins, the coarse mesh in this region is homogenized over each pincell while keeping background discretization identical to that of the fine mesh. The pin definitions for the coarse mesh LFR problem are given in Figure 9-14, where *innerpin* defines the homogenized hexagonal region for all inner pins, while *outerpin* defines the homogenized pincell with discretized background region representing the pincells in the outermost ring. Once again, the block ids are selected so as to match the block ids from the analogous coarse mesh defined by Argonne's Mesh Tools.

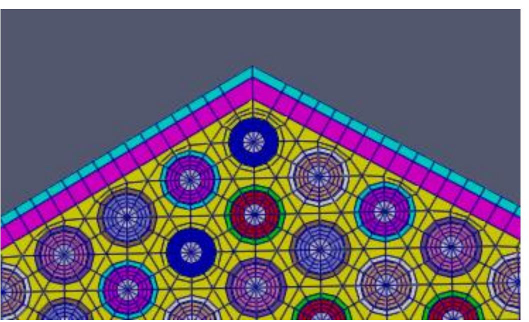
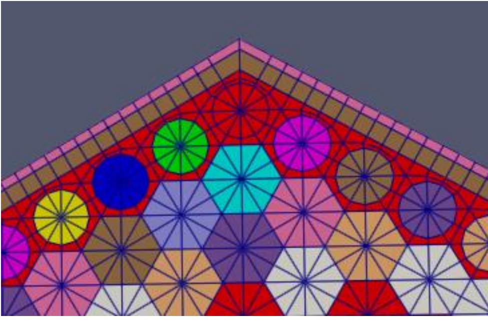



Figure 9-13. Zoomed top-down view of fine-mesh LFR heterogeneous assembly (left) and coarse-mesh LFR heterogeneous assembly (right), generated by new MOOSE mesh tools (right)

```
[innerpin001]
  type = PolygonConcentricCircleMeshGenerator
  num_sides = 6 # must be six to use hex pattern
  num_sectors_per_side = '2 2 2 2 2 2'
  polygon_size = 0.67125
  background_intervals = 1
  background_block_ids = 1
[]
... # Repeat for all other inner pins

[outerpin001]
  type = PolygonConcentricCircleMeshGenerator
  num_sides = 6 # must be six to use hex pattern
  num_sectors_per_side = '2 2 2 2 2 2'
  polygon_size = 0.67125
  ring_radii = '0.5404'
  ring_intervals = '1'
  ring_block_ids = '2'
  background_intervals = 1
  background_block_ids = 3
  preserve_volumes = on
[]
... # Repeat for all other outer pins
```

Figure 9-14. Coarse Mesh LFR 2-D Pin Definition Using PolygonConcentricCircleMeshGenerator

Currently, Griffin offers multiple approaches for diffusion acceleration: Nonlinear Diffusion Acceleration (NDA), NDA with Diffusion Synthetic Acceleration (DSA) stabilizing scheme, and Coarse Mesh Finite Differences (CMFD) acceleration NDA is a generalized form of the CMFD acceleration scheme, while DSA is a linearized form of NDA (Lee, et al., 2021). The CMFD implementation in Griffin is relatively new and was not explored in this example. All acceleration schemes require a coarse mesh to be defined, which must completely contain all elements in the fine mesh. Both Argonne's Mesh Tools and the new MOOSE mesh tools are used to define the coarse mesh for the NDA approach, and Table 9-4 summarizes the k-effective results for Griffin

simulations with and diffusion acceleration. The results in first row are identical to those found in Table 9-3 and represent the case without diffusion acceleration, while the second row aggregates the results when the NDA diffusion acceleration scheme is used. Consistent eigenvalues were obtained for the direct and diffusion-accelerated solvers. More importantly, for a given Griffin solver scheme, a roughly 20pcm difference when the new MOOSE mesh tools framework is used over Argonne's Mesh Tools framework for fine and coarse mesh generation. Once again, this minor difference is due to the variation in the discretization schemes employed by these frameworks in the background region between the outermost pins and the duct regions. However, these results demonstrate close agreement in neutronics results between meshes generated by new MOOSE mesh tools and other mesh generation frameworks such as Argonne's Mesh Tools.

Table 9-4. Griffin k-effective results between new MOOSE mesh tools and Argonne's Mesh Tools for the LFR Assembly Problem with Coarse Mesh Diffusion Acceleration

Griffin Solver Scheme	New MOOSE Mesh Tools K-Effective	Argonne's Mesh Tools K-Effective	K-Effective Difference (pcm)
Direct DFEM-SN	1.17139	1.17119	20
DFEM-SN with NDA Diffusion Acceleration	1.17142	1.17122	20

9.3 MOOSE Tensor Mechanics Verification: Ducted Hexagonal Assembly Example

To demonstrate applicability to other physics codes, a hollow ducted hexagonal assembly (with load pads) mesh was generated for use in a structural mechanics verification problem using MOOSE Tensor Mechanics (analyzed graciously by N. Wozniak at Argonne National Laboratory). The benchmark problem comes from IAEA Verification Problem 1 (VP1) from the working group for the verification and validation of Liquid Metal Fast Breeder Reactor (LMFBR) analysis codes organized by The International Atomic Energy Agency (IAEA) called the International Working Group on Fast Reactors (IWGFR). The coordinated work was performed by eleven participating agencies in nine different countries (Verification and Validation of LMFBR Static Core Mechanics Codes Part I, 1990).

This IAEA VP1 example examines the free (unrestrained) thermal bowed deformation of a single, hexagonal assembly with above core load pad (ACLP). The assembly is 4000 mm in height and is fixed in position at the bottom. The active core region extends from z=1500 to z=2500 mm, and the load pad extends from z=2950 to 3050 mm. A cross section of the assembly geometry is shown in Figure 9-15 which depicts the duct flat-to-flat (D = 132.9 mm) and the load pad flat-to-flat (D_{ACLP} = 138.4 mm). The assembly duct wall thickness is 3 mm. While the load pad modeling is not critical for IAEA VP1, later IAEA verification problems indeed rely on explicit load pad modeling to simulate contact, and therefore the load pad was modeled here as specified in the benchmark problem.

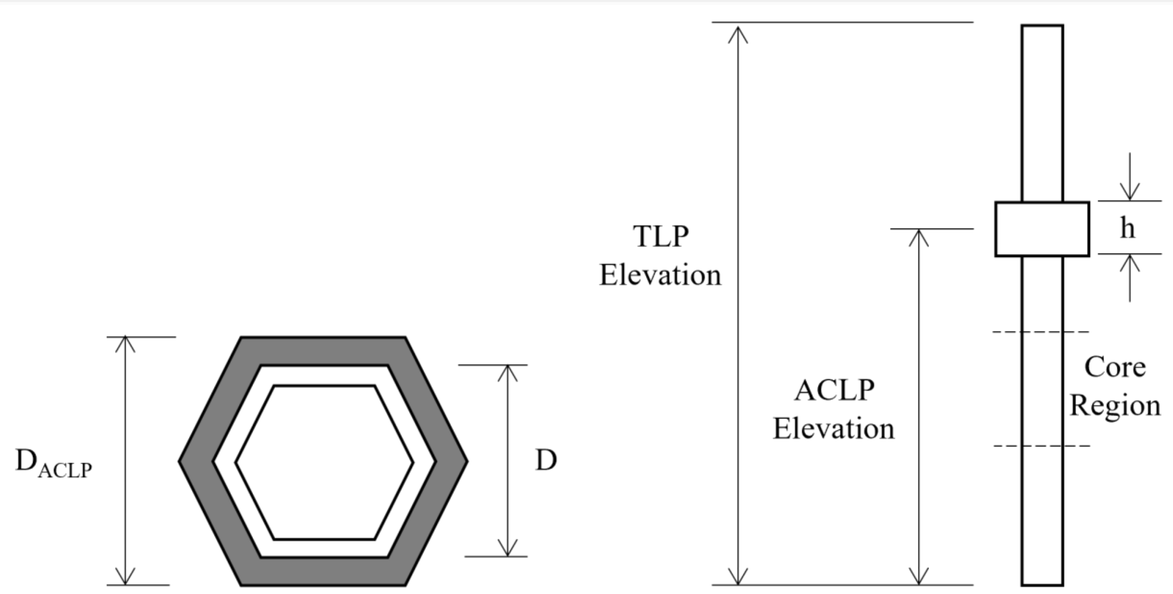


Figure 9-15. Geometry for hexagonal ducted assembly: (left) 2D cross section at load pad, and (right) vertical cross section showing location of load pad and active core.

In this verification problem, a linearly varying thermal gradient is applied along the axial direction in the active core region, shown in Figure 9-16. Below the core, all the duct corners have the same temperature of 400°C (752°F). The duct temperature from core inlet to core outlet varies linearly to 550°C (1022°F) at Corner 4, 537.5°C (999.5°F) at Corners 3 and 5, 512.5°C (954°F) at Corners 2 and 6, and 500°C (932°F) at Corner 1; the temperature remains at these constant temperatures from the core outlet to the top of the duct, shown in Figure 9-17.

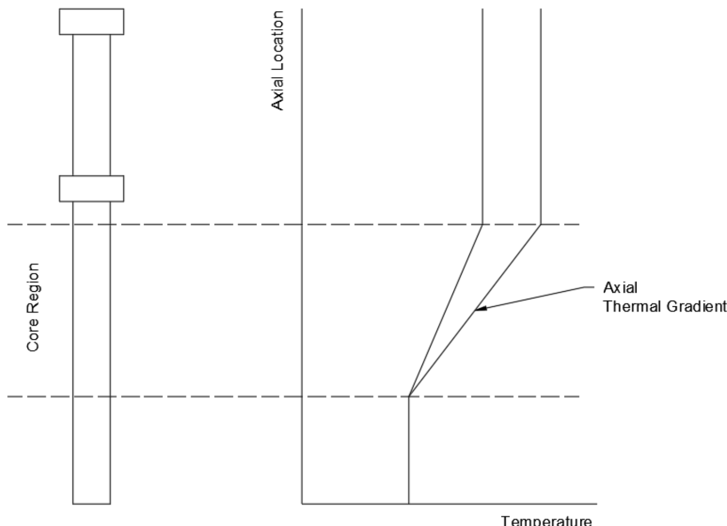


Figure 9-16. Schematic showing the thermal gradient developed axially along the core region of the duct, showing the temperature difference for different corners of the duct.

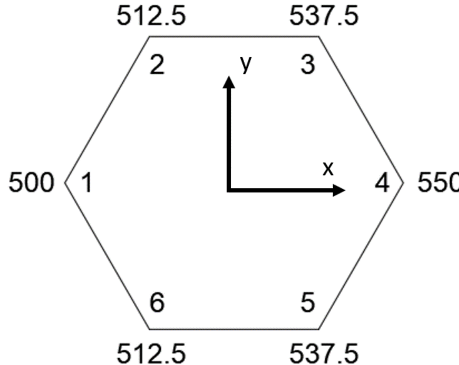


Figure 9-17. Maximum corner temperatures at the top of the core region

9.3.1 Ducted Assembly Mesh Generation with MOOSE

The mesh for this problem was created using a combination of several MOOSE mesh generators, including the recently developed `PolygonConcentricCircleMeshGenerator`, as well as `BlockDeletionGenerator`, `MeshExtruderGenerator`, `TransformGenerator`, and `CombinerGenerator`. The detailed mesh generation process is shown graphically in Figure 9-18. The following steps were taken to generate the ducted hexagonal assembly mesh:

1. Generate a hexagonal 2D pin-cell structure for the assembly duct cladding using `PolygonConcentricCircleMeshGenerator`
2. Remove the inside mesh and keep the duct mesh using `BlockDeletionGenerator`
3. Extrude the duct mesh using either `MeshExtruderGenerator` or `FancyExtruderGenerator`
4. Generate a hexagonal 2D pin-cell structure for the load pad using `PolygonConcentricCircleMeshGenerator`
5. Remove the inside mesh and keep the load pad mesh using `BlockDeletionGenerator`
6. Extrude the loading pad duct mesh using either `MeshExtruderGenerator` or `FancyExtruderGenerator`
7. Move the load pad duct mesh along axial direction (z axis) to the needed location, using `TransformGenerator`
8. Combine or connect the cladding and loading pad duct meshes using `CombinerGenerator` (when connection is not needed, and cladding and loading pad duct meshes are standalone) or `StitchedMeshGenerator` (when connection is needed)
9. Utilize `RenameBoundaryGenerator` to rename sideset faces (optional, but convenient for user to rename as “face1”, etc rather than a numeric value).

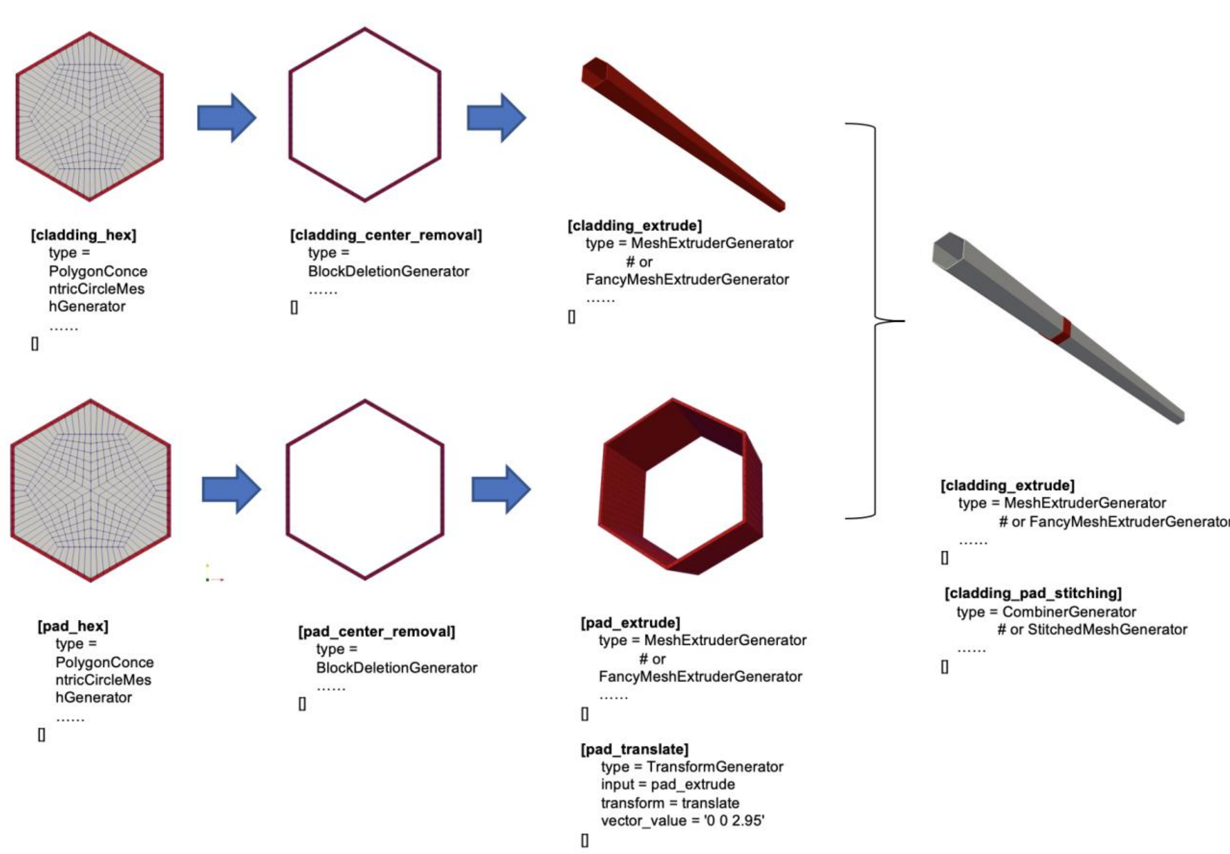


Figure 9-18 Ducted hexagonal assembly mesh generation

The following figures compare the ducted hexagonal assembly meshes generated by Cubit and MOOSE. The MOOSE mesh contains more elements radially (a user choice).

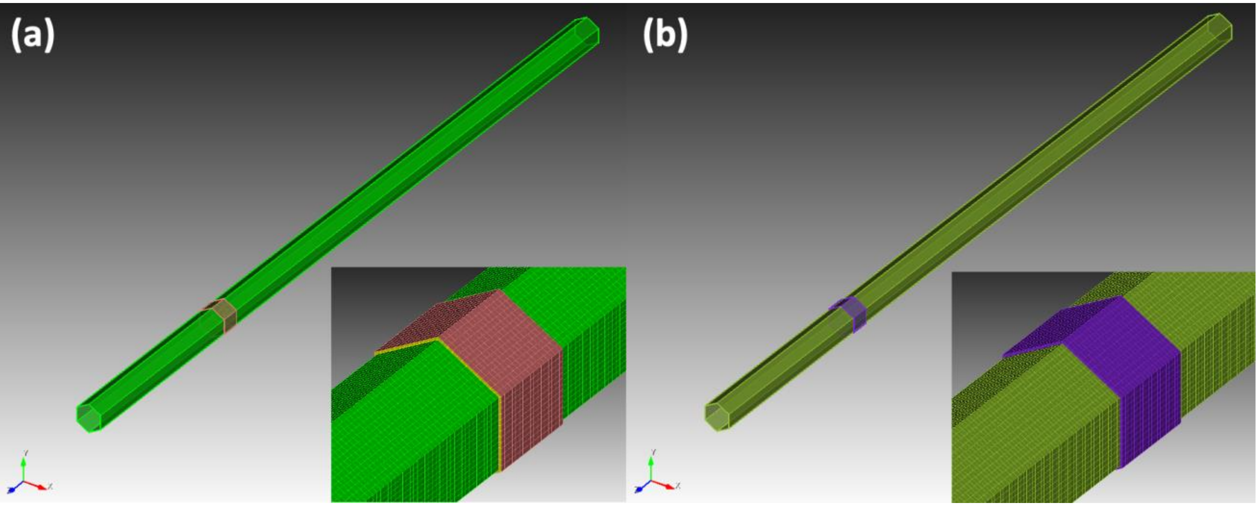


Figure 9-19 Ducted hexagonal assembly meshes: (a) mesh generated by Cubit; and (b) mesh generated by MOOSE

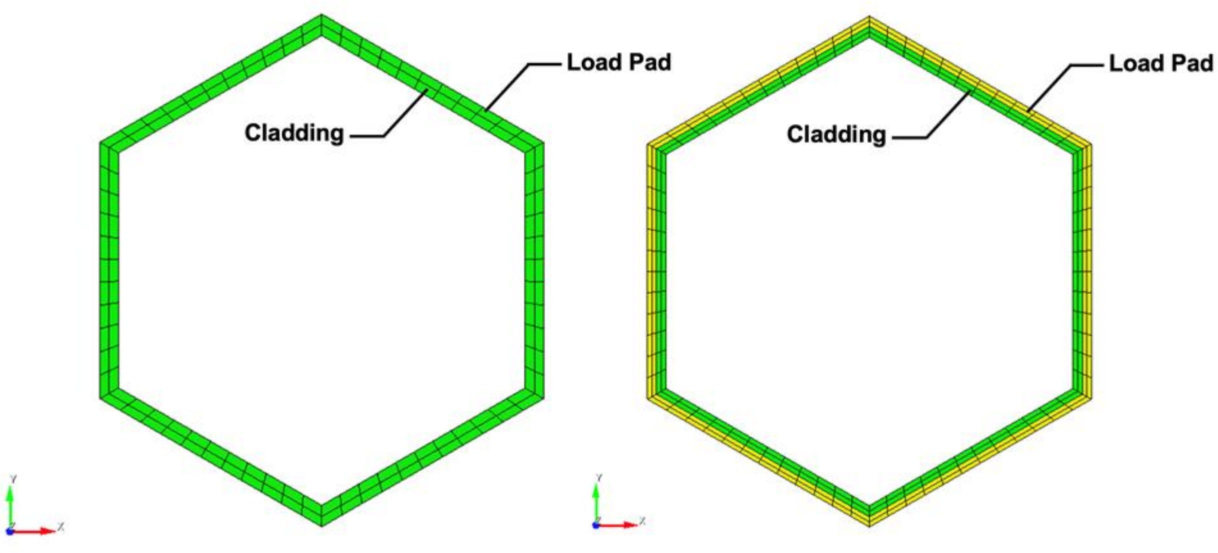


Figure 9-20. Ducted hexagonal assembly mesh cross section at load pad elevation, (left) generated by Cubit, and (right) generated by MOOSE

We note that there may be other ways of building this mesh within MOOSE but the final mesh must contain the appropriate sidesets (one per face of the hexagon so that temperature can be applied).

9.3.2 Comparison of Results using MOOSE vs. Cubit meshes

The temperature distribution was prescribed on each outside face of the hexagonal cross-section. The calculated displacements are shown in Figure 9-21. The view presented in the figure is along the (6-5) face view from Figure 9-17, which has a maximum temperature on the right side (positive x axis) of 550°C and a minimum temperature on the left side (negative x axis). This tends to bow the duct from right to left in the positive Y direction. The comparison of the centerline deflection curves is provided in Figure 9-22 with error percentages provided in Table 9-5 for deflection values at the top of the core region, the ACLP midplane, and the TLP midplane locations.

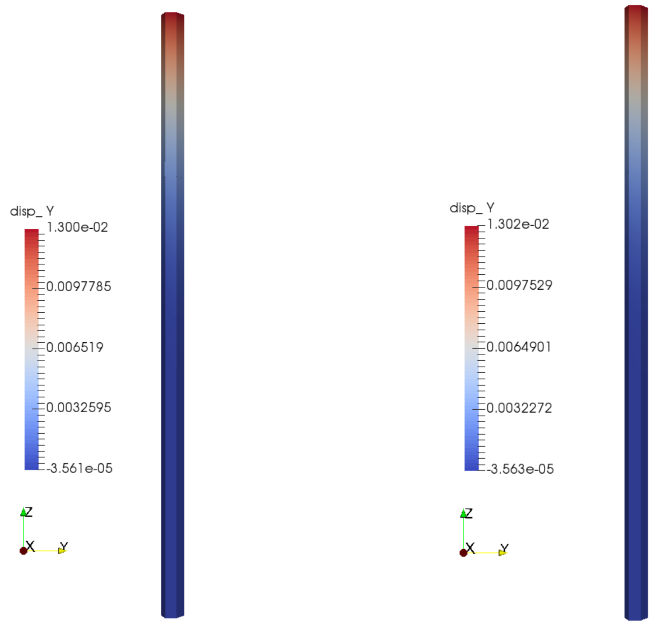


Figure 9-21. Cubit mesh displacement in m (left), and MOOSE displacement in m (right).

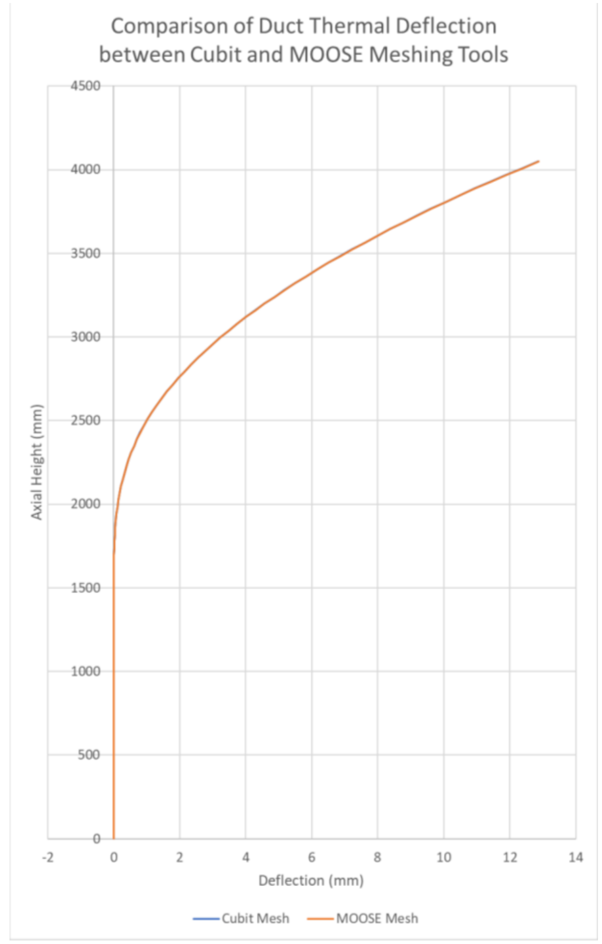


Figure 9-22. Duct thermal deflection displacement (Cubit and MOOSE meshes)

The table below summarizes the deflection calculations at the top of the core region, the ACLP midplane, and the TLP midplane. Excellent agreement between the different mesh methods is observed at each location with a maximum difference of 0.08% at the TLP. Collectively, these results indicate that the MOOSE hex mesh generator can accurately mesh a hexagonal thin-walled duct with load pads for a more efficient workflow than using Cubit.

Table 9-5. Comparison of centerline deflection results between the Cubit mesh and the MOOSE hex mesh generator, at the top of the core region, the ACLP and TLP midplanes

	Cubit (mm)	MOOSE (mm)	Difference (%)
Core Top	1.00	1.00	0.0
ACLP midplane	3.25	3.25	0.0
TLP midplane	12.26	12.27	0.08

9.4 Multiphysics Verification: Microreactor Example

Microreactor concepts have received significant attention in the United States due to the unique features of lower capital investment, siting flexibility and high mobility. Under NEAMS, an on-going effort to demonstrate microreactor multiphysics simulation capabilities by coupling multiple MOOSE based codes BISON (fuel performance analysis) Griffin (neutronics), Sockeye (heat pipe analysis), and SAM (system analysis) is underway (Stauff, et al., 2021). Cubit was originally employed to generate the full core mesh. Cubit is a powerful software toolkit for 2D and 3D mesh generations, with a comprehensive set of meshing schemes. Cubit users can generate and visualize complex meshes with high flexibility. However, Cubit does not provide a template or scripts to generate reactor geometries. Users are required to start from drawing circles (for fuel pellets/pins/claddings) and squares/hexagons (for pin cells/assemblies) one by one, followed by extrusion of 2D structures to 3D, and defining massive numbers of blocks and boundaries at the end. The mesh generation process can be extremely time consuming. For an inexperienced Cubit user, any change in reactor structure may induce substantial efforts on mesh re-generation. Moreover, the intrinsic meshing schemes in Cubit do not support volume preservation, which can cost additional effort for the users to tune the circular structures in order to preserve their volumes.

9.4.1 Microreactor Core Mesh Generation with MOOSE Mesh Generators

In the microreactor core example taken from the NEAMS Microreactor analysis activity, typical heat pipe micro-reactor components including control drums, reflectors, graphite monolith, heat pipes, moderators, fuel, and air gaps within the reactor were modeled. Figure 9-23(a), (b) and (c) shows the 1/6 symmetric core mesh generated by Cubit and used in that analysis.

With the newly developed mesh generators, MOOSE is now capable of generating the full core microreactor mesh analogous to the one generated by Cubit. Figure 9-23(d), (e) and (f) show the 1/6 symmetric core mesh generated by MOOSE. All the microreactor components in the Cubit mesh were reproduced and meshed. Details of the mesh generation process are shown graphically in Figure 9-24, Figure 9-25, and Figure 9-26. The following steps were taken to generate the 1/6 symmetric core mesh:

1. Generate hexagonal 2D pin-cell structures, including moderator (Mod_hex), heat pipe (HP_hex) and fuel (Fuel_hex), using `PolygonConcentricCircleMeshGenerator` (Figure 9-24)
2. Stitch pin-cell structures generated in step 1 together to produce an assembly using `PatternedHexMeshGenerator` (Figure 9-24)
3. Generate hexagonal 2D pin-cell structures, including control drums, reflectors, dummy, and air center block, using `HexagonConcentricCircleAdaptiveBoundaryMeshGenerator` and `AzimuthalBlockIDMeshGenerator`. Noted that `AzimuthalBlockIDMeshGenerator` is only needed for generating control drum mesh (Figure 9-25)
4. Assemble pin-cell structures generated in step 3 and the assembly mesh generated in step 2 into the core mesh using `PatternedHexMeshGenerator` (Figure 9-25)
5. Remove the dummy blocks and slice the 2D full core mesh into 1/6 symmetric core mesh using `BlockDeletionGenerator` assisted with `ParsedSubdomainMeshGenerator` (Figure 9-26)
6. Extrude the 2D 1/6 symmetric core mesh to 3D using either `MeshExtruderGenerator` or `FancyMeshExtruderGenerator` (Figure 9-26)
7. Define the top and bottom reflector using `ParsedSubdomainMeshGenerator` (Figure 9-26)
8. Generate and rename sidesets using `SideSetsBetweenSubdomainsGenerator` and `RenameBoundaryGenerator`.

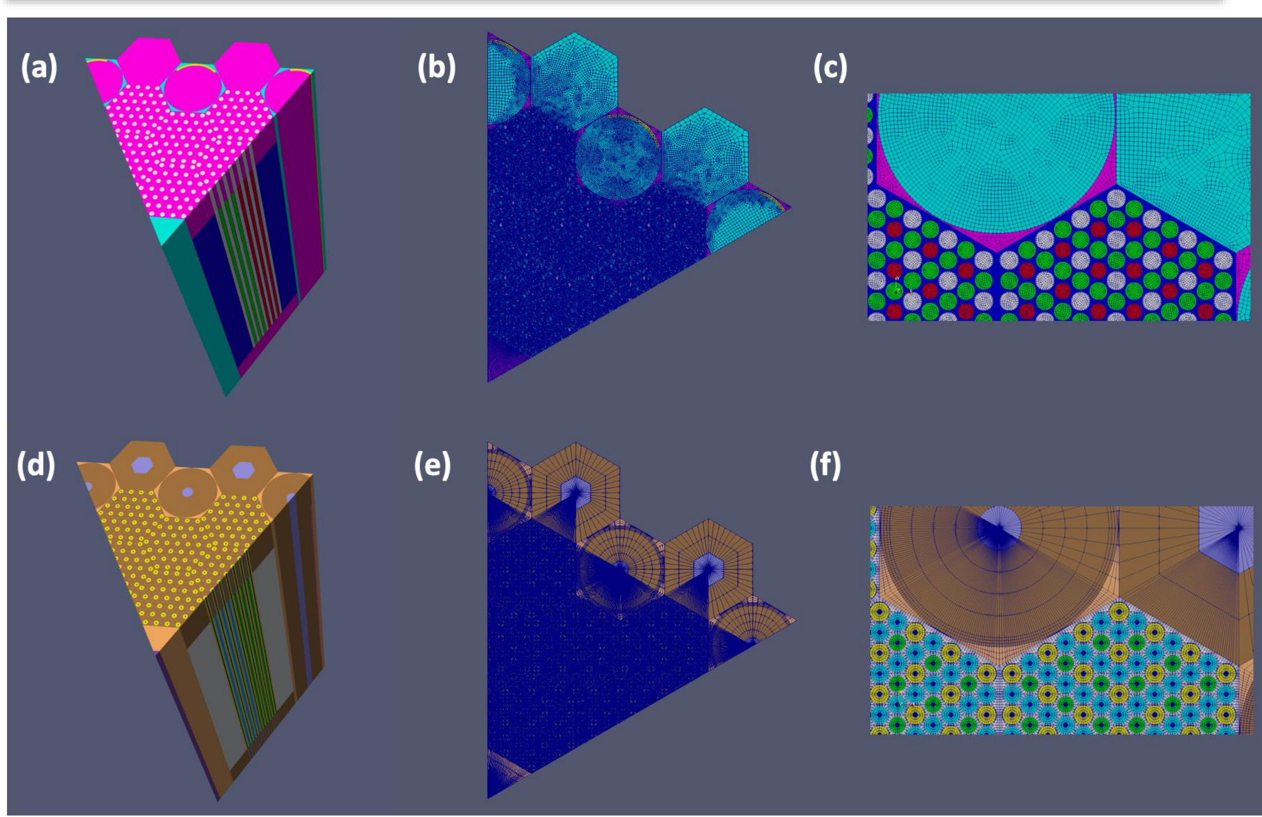


Figure 9-23. Microreactor core meshes produced by CUBIT (number of nodes: 1.7×10^6): (a) 1/6 symmetric core mesh, (b) cross-section of the core, and (c) zoom-in region of (b); and MOOSE mesh generators (number of nodes: 1.0×10^6): (d) 1/6 symmetric core full core mesh, (e) cross-section of the core, and (f) zoom-in region of (e)

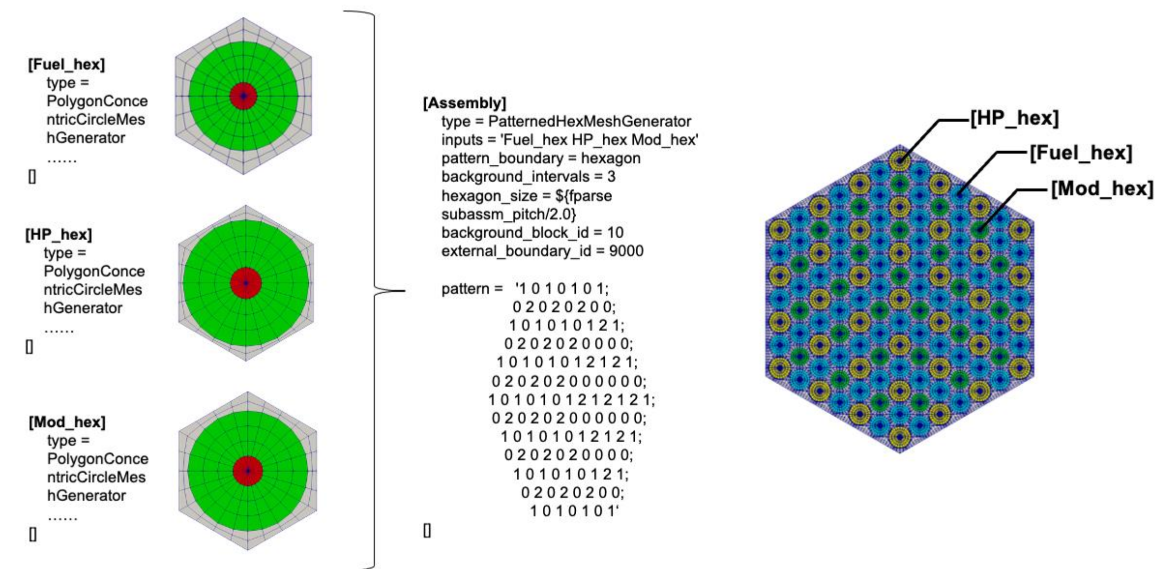


Figure 9-24. 2D assembly mesh generation showing moderator, heat pipe, and fuel pin cell being combined into a hexagonal pattern

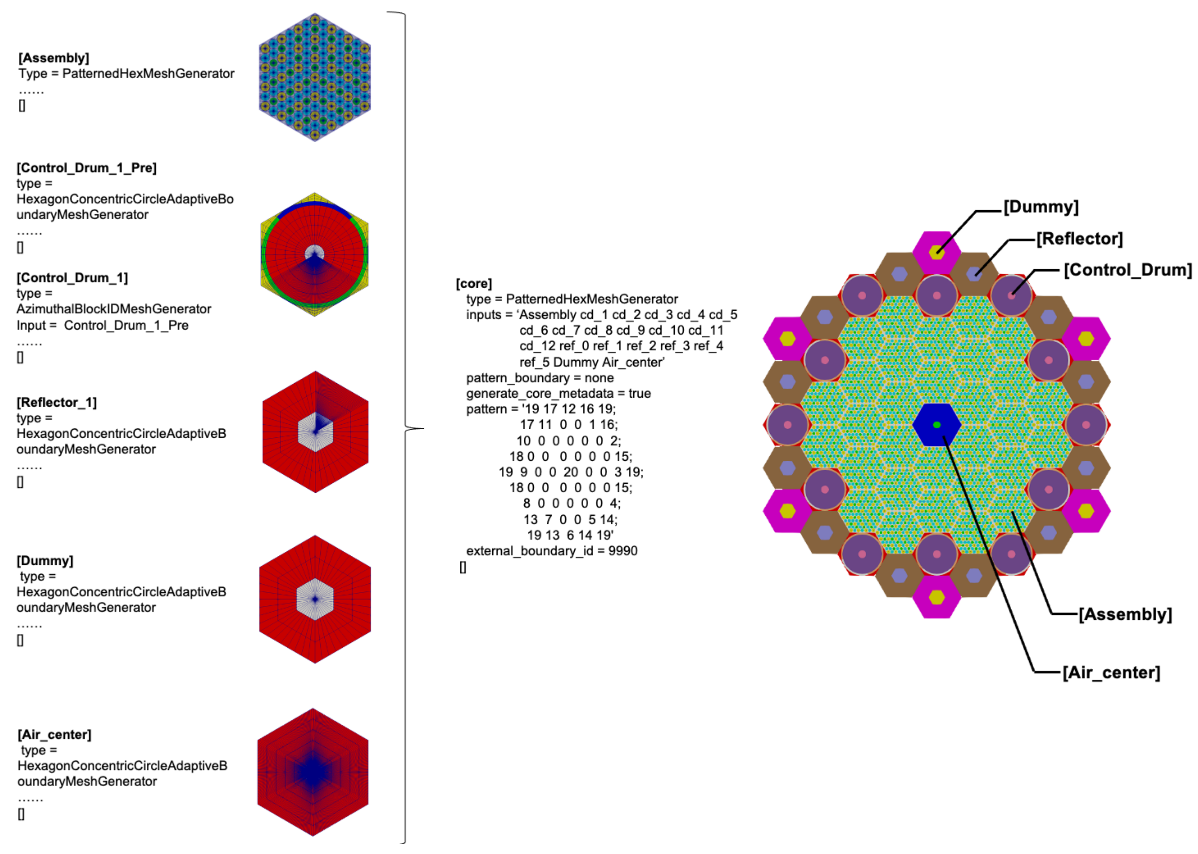


Figure 9-25. 2D core mesh generation showing assemblies (fuel, control drum, reflector, dummy) being combined into a core

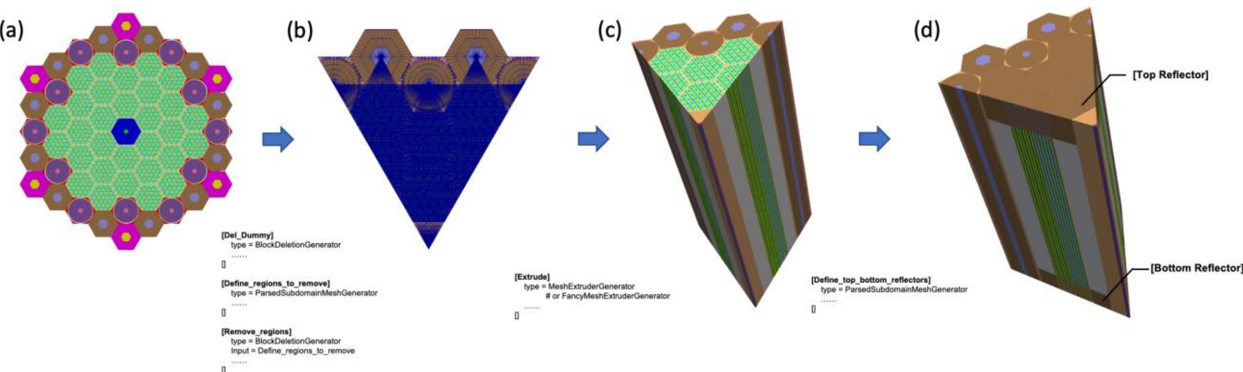


Figure 9-26. 3D 1/6 symmetric core mesh generation: (a) 2D core mesh; (b) trimmed (dummy blocks removal) and sliced 2D 1/6 symmetric core mesh; (c) extruded 3D 1/6 symmetric core mesh; and (d) completed 3D 1/6 symmetric core mesh after defining axial blocks

Two simulations were conducted to examine the performance of 3D 1/6 symmetric core mesh generated by MOOSE mesh generators in comparison to that generated by Cubit: (1) a heat conduction simulation using BISON, and (2) a multiphysics simulation with coupling BISON-Griffin-Sockeye.

9.4.2 BISON Heat-Conduction Simulation with MOOSE mesh

The first simulation is a simple heat conduction problem with no input power. The core was simulated to establish a “steady-state” which is also the starting status of the multiphysics simulation. The temperature of the core at the initial state was 300K uniformly. Without input power, the temperature distribution relies on the boundary conditions: the heat pipe surfaces with T_{ext} =800K are the only heat sources, and external surfaces (including top, bottom and outer surfaces of reflector, and top surface of helium gap) with T_{ext} =300K become the heat sink.

Figure 9-27 shows the temperature distribution of 1/6 symmetric core at the last time step (20 sec) of simulation. Similar temperature distributions were developed using the meshes generated by Cubit and MOOSE mesh generators. Temperature evolutions of fuel and heat pipe surfaces are selected to compare computations with the meshes generated by Cubit and MOOSE mesh generators (Figure 9-28). As seen, the difference in temperature evolutions is negligible, indicating both meshes are reliable in the heat conduction simulation.

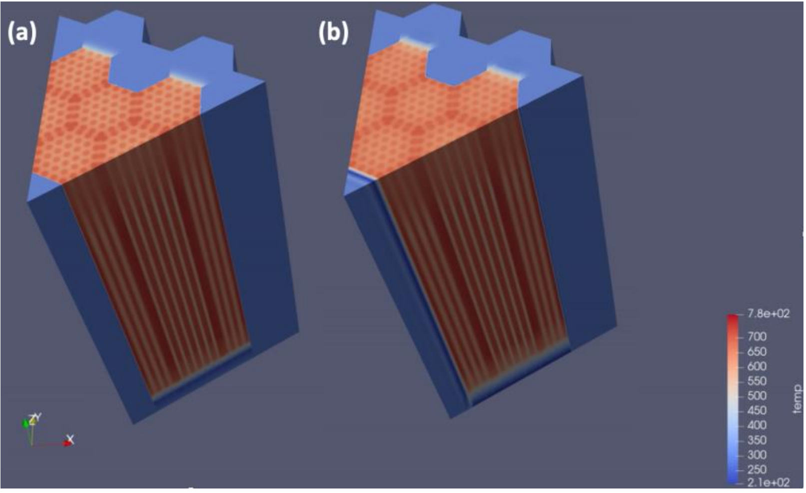


Figure 9-27. Temperature distribution of the 1/6 symmetric core at the last time step (20 sec) of simulation: (a) simulation based on the mesh generated by CUBIT (b) simulation based on the mesh generated by meshgenerators

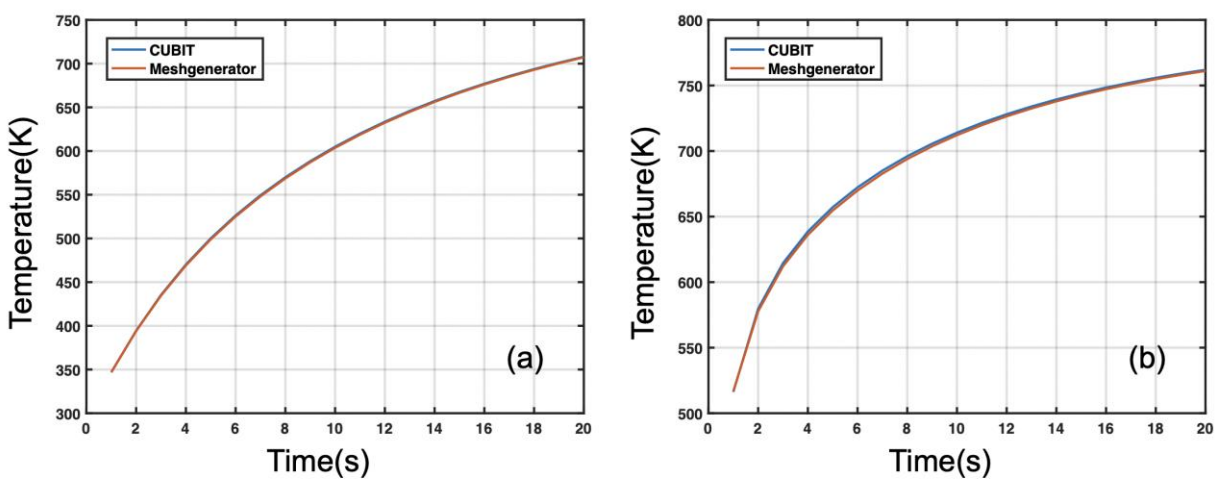


Figure 9-28. Temperature evolution of the heat conduction simulation: (a) average fuel temperature, and (b) average heat pipe surface temperature

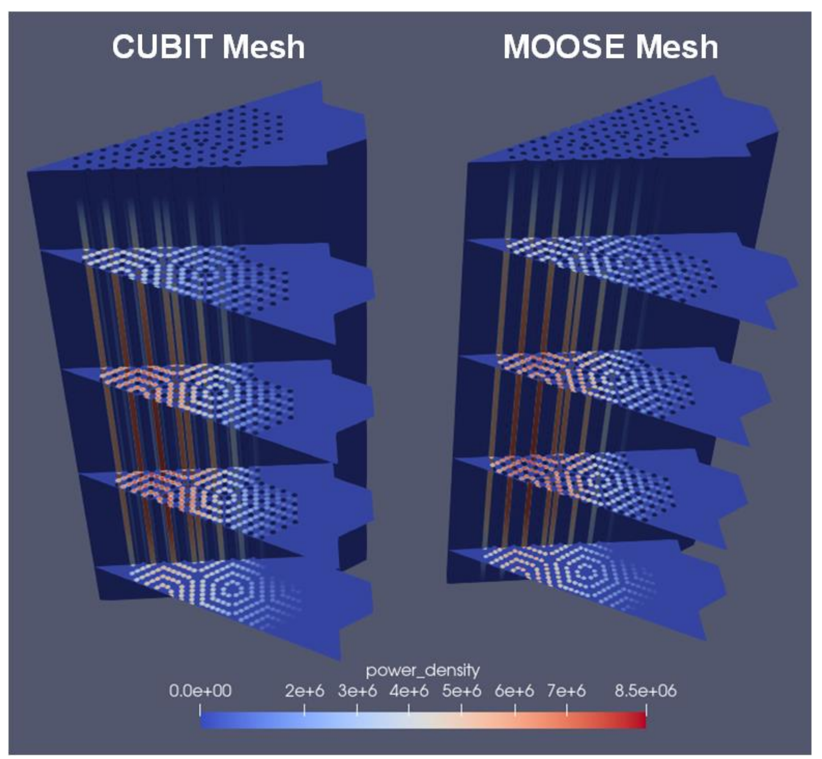


Figure 9-29. Power density comparison of multiphysics simulations using the two different meshes (unit: W/m^3).

9.4.3 Griffin-BISON-Sockeye Multiphysics Simulations

The consistency between the CUBIT generated mesh and MOOSE generated mesh was further investigated using a Multiphysics approach. The Multiphysics simulation was performed by coupling three MOOSE applications: Griffin, BISON and Sockeye. Here, Griffin was used as the main application that performs steady state eigenvalue calculation using the generated 3D core meshes and diffusion theory. The main application is coupled with a BISON sub-application using the same meshes through Picard iteration. The Griffin main application provides power density distribution to the BISON sub-application and gets fuel temperature profile back during each iteration. Additionally, for each single heat pipe, the BISON sub-application has its own second-level Sockeye sub-application to simulate the heat pipe cooling performance. The power density and temperature profile at the steady state predicted by the Griffin-BISON-Sockeye approach using the two meshes are illustrated in Figure 9-29 and Figure 9-30, respectively. Use of Cubit and MOOSE meshes produced consistent power and temperature prediction.

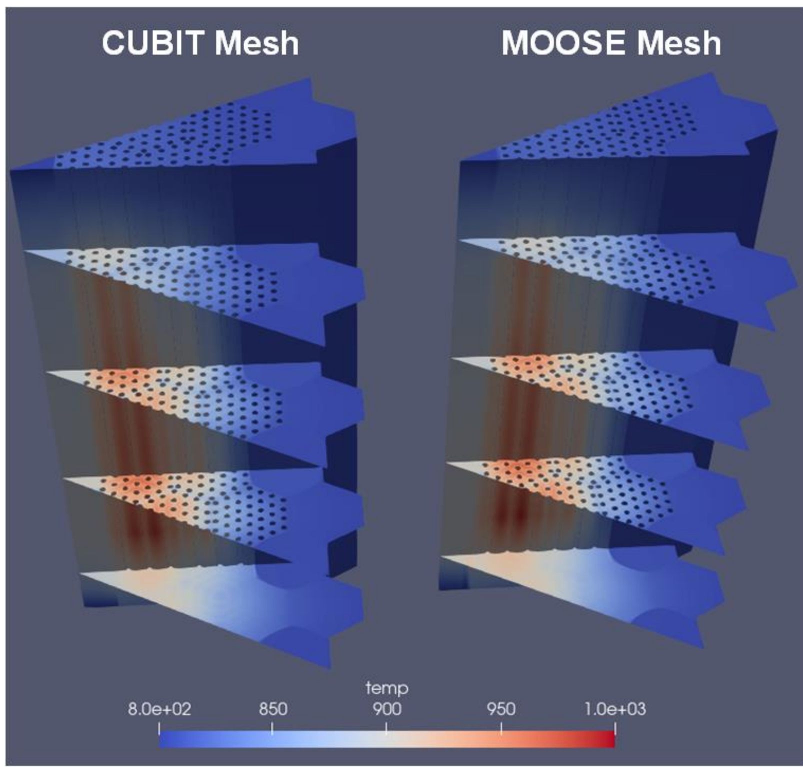


Figure 9-30. Temperature comparison of Multiphysics simulations using the two different meshes (unit: K).

Table 9-6. Key calculated parameters comparison between the two meshes (temperature unit: K)

	CUBIT	MOOSE
k_{eff}	0.93078	0.93165
T_{max}^{fuel}	982.50	987.62
T_{avg}^{fuel}	873.07	874.01
T_{min}^{fuel}	804.41	804.53
$T_{max}^{moderator}$	971.59	975.58
$T_{avg}^{moderator}$	870.92	871.89
$T_{min}^{moderator}$	806.11	806.34
$T_{max}^{monolith}$	978.38	983.18
$T_{avg}^{monolith}$	866.66	867.64
$T_{min}^{monolith}$	803.62	803.79

A more specific comparison is made in Table 9-6. The eigenvalue differs by less than 100 pcm by using the Cubit mesh and MOOSE mesh. The temperature difference is usually lower than 5 K for different components of the reactor core. This is excellent agreement considering that the Cubit and MOOSE meshes have vastly different discretization (owing to the “pave” algorithm in Cubit which is not easily controlled by the user). The run-time for the simulation using Cubit mesh is greater than that using MOOSE mesh due to the difference in mesh density (Figure 9-23).

9.5 Summary of Physics Tests

Verification tests performed with Griffin, MOOSE Tensor Mechanics, BISON, and coupled Griffin-Bison-Sockeye simulations demonstrate that the newly developed mesh tools provide consistent physics solutions compared to similar meshes generated outside MOOSE. Minor differences were observed and are expected since the meshes tested were not exactly identical.

The new MOOSE meshing tools offer improved user workflow by avoiding the use of external tools and being able to access and control mesh data more easily to facilitate physics inputs. In particular, avoiding CUBIT is ideal since this tool has a steep learning curve and also does not have the capability to preserve volume (meshed volume may not equal geometrical volume) in cylindrical fuel pins and absorber regions. Cubit meshes therefore must be used in conjunction with a density correction in the physics code which can be tedious to perform. The MOOSE tools used to generate pin cells conserve circular/cylindrical volumes automatically unless the user overrides this option. Use of MOOSE meshing tools can also mesh the geometry more optimally than Cubit for certain geometries and result in a fewer nodes and lower computational time.

The new tools offer flexibility in defining core layout, block id assignment in 3D extrusion, and they allow boundary sidesets to be named descriptively instead of arbitrary sideset ids. Mesh sensitivity analysis can be done directly by changing input file directly instead of having to re-generate Exodus file through external mesh tools for each mesh adjustment under consideration. In fact, the Stochastic Tools Module of MOOSE can also access and perturb mesh parameters which permits additional parameterization studies which were previously not possible with an external mesh. Finally, coarse meshes can be defined directly instead of having to import separate exodus mesh files from an external program.

10 Summary

A series of new MOOSE mesh generators have been developed to support advanced reactor geometry meshing needs. The new mesh generators support five primary priorities:

1. Hexagonal geometry meshing capability including pin cells, ducted assemblies, and cores
2. Control drum geometry meshing capability including static and rotating absorber
3. Reporting ID capability, i.e., implementation of element-wise mesh information for regular Cartesian and hexagonal geometries, to identify specific reactor features/zones
4. Core periphery meshing capability
5. Reactor Geometry Mesh Builder (RGMB) capability that employs reactor analyst input language to build up 3D Cartesian or hexagonal heterogeneous reactor cores more intuitively including material assignment

The first two sets of mesh generators have been tested using Griffin, MOOSE Tensor Mechanics, BISON, and coupled Griffin-Bison-Sockeye simulations on a variety of fast reactor and microreactor problems. Solutions compared well to previous results which relied on external meshing software, demonstrating that the MOOSE-based meshes are performing correctly.

At time of this report writing, the mesh generators are in the process of being integrated into the MOOSE framework and are expected to be available to the general user within the "reactor" module by late 2021.

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.

- Physics verification and output testing of capabilities 3, 4 and 5, as well as the transient control drum capability in 2
- Enable proper modifications on the elements of the outmost layer of the hexagon meshes so that meshes with different numbers of nodes on the external boundaries can be stitched together after such modifications.
- Enable a single meshing input block for meshes required for all levels of MultiApps system. 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.
- Rearrange node positions near a specified plane or surface to prepare a mesh for clipping.
- Addition of refinement algorithm to `TriangulatedMeshGenerator` to improve quality of triangulated triangles
- Extend outer boundary of `TriangulatedMeshGenerator` to additional shapes besides circle, set of points (e.g., from external program), and/or boundary of existing meshgenerator mesh (if core periphery is part of another mesh)
- Improve the mesh generators which define sidesets/boundary names and IDs, such as `SideSetsFromPointsGenerator` and `SideSetsFromNormalsGenerator` to allow selections of specific blocks by users

- Auto-generate sideset IDs during 3-D extrusion based on number of unique id's in 2D map
- Develop consistent naming and numbering convention for sidesets and boundaries to avoid conflicts between mesh generators
- Development of a MOOSE native Delaunay triangulation routine
- Additional testing is needed of control drum routines, reporting_id, `TriangleMeshGenerator` and the RGMB mesh generators that were developed
- Implement a coarse mesh generation capability based on mesh meta data and reporting IDs to support a coarse mesh based acceleration technique.
- Add additionally functionality to RGMB (Cartesian ducts, ability to create a control drum object, ability to auto-collapse blocks, enhanced sideset control)

REFERENCES

Apache License. (2021). *Apache License, Version 2.0*. Retrieved from Open Source Initiative: https://opensource.org/licenses/Apache-2.0

BSD License. (2021). *The 3-Clause BSD License*. Retrieved from Open Source Initiative: https://opensource.org/licenses/BSD-3-Clause

CUBIT. (2021). Retrieved from The CUBIT Geometry & Meshing Toolkit: https://cubit.sandia.gov/

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.

Geuzaine, C., & Remacle, J.-F. (2021). *Gmsh - A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*. Retrieved from https://gmsh.info/

GPL License. (2021). *GNU General Public License v3*. Retrieved from Open Source Initiative: https://opensource.org/licenses/GPL-3.0

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

Jung, Y., Lee, C., & Smith, M. (2018). *PROTEUS-MOC User Manual.* Argonne, IL: ANL/NSE-18/10, Argonne National Laboratory. doi:10.2172/1483947

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.

LGPL License. (2021). *GNU Lesser General Public License v3*. Retrieved from Open Source Initiative: https://opensource.org/licenses/LGPL-3.0

MIT License. (2021). *The MIT License*. Retrieved from Open Source Initiative: https://opensource.org/licenses/MIT

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

Shemon, E. R., Grudzinski, J. J., Lee, C. H., Thomas, J. W., & Yu, Y. Q. (2015). *Specification of the Advanced Burner Test Reactor Multi-Physics Coupling Demonstration Problem.* Argonne, IL: ANL/NE-15/43, Argonne National Laboratory.

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

Shemon, E., Yu, Y., & Kim, T. (2020). *Demonstration of NEAMS Multiphysics Tools for Fast Reactor Applications.* Argonne, IL: ANL/NSE-20/25, Argonne National Laboratory.

Shewchuk, J. R. (2021). *Triangle - A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*. Retrieved from https://www.cs.cmu.edu/~quake/triangle.html

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., 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.

TetGen. (2021). *TetGen - A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator*. Retrieved from https://wias-berlin.de/software/index.jsp?id=TetGen

VERA. (2021). Retrieved from The Virtual Environment for Reactor Applications: https://vera.ornl.gov/

(1990). *Verification and Validation of LMFBR Static Core Mechanics Codes Part I.* Vienna, Austria: IWGFR/75, International Atomic Energy Agency.

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