Creating Geometry and Mesh Models for Nuclear Reactor Core Geometries Using a Lattice Hierarchy-Based Approach

Timothy. J. Tautges[1] and Rajeev Jain[2]

[1]Argonne National Laboratory, Argonne, IL 60439, USA
tautges@mcs.anl.gov
[2]Argonne National Laboratory, Argonne, IL 60439, USA
jain@mcs.anl.gov

Summary. Nuclear reactor cores are constructed as rectangular or hexagonal lattices of assemblies, where each assembly is itself a lattice of fuel, control, and instrumentation pins, surrounded by water or other material that moderates neutron energy and carries away fission heat. We describe a system for generating geometry and mesh for these systems. The method takes advantage of information about repeated structures in both assembly and core lattices to simplify the overall process. The system allows targeted user intervention midway through the process, enabling modification and manipulation of models for meshing or other purposes. Starting from text files describing assemblies and core, the tool can generate geometry and mesh for these models automatically as well. Simple and complex examples of tool operation are given, with the latter demonstrating generation of meshes with 12 million hexahedral elements in less than 30 minutes on a desktop workstation, using about 4 GB of memory. The tool is released as open source software as part of the MeshKit mesh generation library.

Key words: Reactor Core, Mesh Generation, Lattice-Based Geometry

1 Introduction

Reactor cores can be described as a two-level hierarchy of lattices; the first level of the hierarchy corresponds to fuel assemblies, formed as a lattice of cylindrical pins, while in the second level assemblies of various types are arranged in a lattice to form the reactor core. Generation of geometry and mesh models for reactor cores can be a difficult process. While the struc-

ture inherent in this two-level hierarchy could be used to automate parts of this generation process, experience shows that user interaction is often required at key points of this process as well. The ideal geometry and mesh generation process for these models would balance both lattice-guided automation and opportunities for user interaction at key points in the process. This paper describes a system for generating reactor core geometry and mesh models that balances automation and user interaction in this way.

Nuclear reactor cores are formed by arranging cylindrical pins in a lattice of surrounding material. Pins contain uranium-based fuel, absorbing material for controlling the nuclear chain reaction, or instrumentation. The surrounding material functions as coolant or neutron energy moderator. These materials can be arranged in either a rectangular lattice, used in water-cooled reactors, or a hexagonal lattice, more common in sodium- and gas-cooled reactors. Assemblies vary by degree of uranium enrichment in the fuel material, type of control rod material or function, or other parameters. Assemblies are arranged in a lattice to form the reactor core, either filling space (typical in water- and gas-cooled reactors) or with an interstices material (common in sodium-cooled reactors). Examples of typical assembly and core types are shown later in this paper.

Domain-specific mesh generation tools have been described for various application areas; a few are mentioned here for comparison. The CUBIT meshing toolkit has been used to design tire-specific meshing systems for tire design [1]. Designers are permitted to enter a prescribed list of parameters describing tire and tread model, after which the geometry and mesh are generated automatically. Using external tools such as Distmesh, an open source mesh generator, Fung et al. [2] reported a similar system for electrical impedance tomography. Tools to generate finite element meshes suitable for CFD calculations of blood flow in arteries have been demonstrated by Cebral et al. [3]. Several systems for grid generation for turbomachinery configurations have been reported [4][5]. Some other domain-specific mesh generation tools can be found at the mesh software website [6]. These systems all follow the general idea reported here, where a system allows limited variations in prescribed parameters, from which a geometry and mesh are generated for a specific type of analysis. However, relatively little advantage is taken of repeated structures in the geometric model or of the automation possible through the repeated structures.

In lattice-based model generation, two specific works are similar to our approach: the PSG2 code, which allows specification of model geometry using a unit cell-based approach [7], and the Sabrina and Moritz tools, developed to support radiation transport calculations [8]. However, the models constructed from these descriptions are limited in that they support only a CSG-based representation of the domain, used solely to perform ray trac-

ing for Monte Carlo–based radiation transport calculations. We assert that this general approach has far broader applications, not only in geometry construction but also to support mesh generation.

Hansen and Owen [9] present a general overview of the challenges and opportunities in meshing reactor cores. They mention the need to "develop the tools that are easier to use keeping the user informed about what is occurring inside the tool." They study the current state of meshing and examine various requirements for generating high-quality reactor meshes. They also discuss issues and considerations for creating meshes for multi-physics simulations. However, there is no mention of taking advantage of lattice-type configurations found in most reactor cores.

Before embarking on the effort reported here, we explored generation of reactor core geometry and mesh using the more general CUBIT meshing tool. However, these efforts met challenges in multiple areas. Various methods were explored. In the first method tried, constructing the whole reactor core geometry and meshing it afterwards required a great deal of user interaction to fine-tune the mesh scheme and intervals in the various assembly types. The process was also brittle: small changes to meshing parameters caused the overall meshing process to fail or generate unexpected results. Generating geometry and mesh for individual assemblies, copy/moving them into the overall core model, and then merging geometry and mesh for the whole collection proved more robust; however, this approach also required large amounts of memory (6 GB for a model of < 1 million hexahedral elements) and execution time (several hours on a desktop Linux-based workstation). From this experience, we conclude that while it might be possible to use tools such as CUBIT for this type of problem, we can probably do better by considering the inherent structure of the hierarchy of lattices present in reactor core models. We believe that the results later in this paper show this to be the case.

In this paper, we describe a three-stage process for generating core lattice geometry and mesh. First, assembly models of various types are generated, based on input describing the content of unit cells, the arrangement of unit cells in the lattice, and the extent of the lattice and any surrounding material. This stage also outputs two CUBIT journal files that can be used to automatically mesh the assembly model. Second, the assembly geometry is meshed with CUBIT, either automatically based on the journal files output in the first stage, or after interactive modifications of that meshing procedure. The first two stages are repeated for each assembly type in the core model. Third, after all assembly meshes have been constructed, they are arranged in a core model using a copy/move/merge process operating on the mesh only. The resulting mesh has material assignments and boundary condition groupings that can be used in typical reactor core analysis.

In many cases, the three-stage process described here is fully automatic, resulting in a meshed core model (the most significant barrier to automation is reliable meshing of the cross section of each assembly; this is discussed Section 3.2). At the same time, the user has several opportunities for making targeted adjustments to the process, while being able to execute the process as before for unmodified parts. In this way, both automation and the opportunity for interactivity are balanced. This approach offers several advantages to the analyst: (1) the time required to update the model parameters and create a new model is considerably lower compared to traditional approaches; (2) creation of different meshes for sensitivity analysis is very easy; (3) some non-lattice-based features are incorporated in this methodology to support specific needs in rectangular and hexagonal core geometries; (4) the same set of tools can be used to accommodate different types of reactors; and (5) the process can be automatically re-executed by a makefile automatically created by the tools described in this paper, regenerating only those parts of the model depending on the specific changes.

The primary contribution of this paper is the demonstration of a method for constructing geometry and mesh models for problems appearing as a two-level hierarchy of lattices. Using the lattice information simplifies input, improves automation, and is more efficient in both time and memory. Our implementation of this method maintains a careful balance between automation and interactivity. Moreover, the method used to propagate material and boundary condition information through the copy/move process, based on the entity set abstraction used to access mesh, is a demonstration of how to handle application-specific metadata abstractly; this approach will prove useful in other meshing contexts, for example, adaptive mesh refinement and topological mesh improvement.

The remainder of this paper is organized as follows. Section 2 describes the types of lattices addressed by the tools in this paper. Section 3 describes in detail the tools for generating assembly geometry and core meshes. Section 4 gives implementation details. Section 5 describes the propagation of mesh metadata during the core mesh generation stage. Section 6 describes example core models created using these tools, along with performance data. Section 7 discusses our conclusions.

2 Reactor Core Lattices

The term "lattice" refers to a collection of shapes arranged in some regular pattern. The shapes composing the lattice are called unit cells; in this work we also refer to these as pin cells. In the fully general case, a lattice can be

formed from unit cells of varying shape; for example, the surface of a soccer ball is formed from a collection of pentagons and hexagons. Here, we restrict ourselves to lattices with a single unit cell shape. The internal structure of unit cells is allowed to vary, however, resulting in assembly/core models composed of multiple unit cell/assembly types, respectively.

Nuclear reactor cores have been designed with a wide variety of fuel pin and assembly arrangements. Two lattice types are most common: rectangular and hexagonal lattices. Rectangular or square lattices are used mostly in light water reactor designs, while hexagonal lattices are most common in gas- and sodium-cooled reactors.

The geometry of a lattice can be fully described by three types of parameters: the unit cell shape or lattice shape; the lattice pitch, which is the distance between unit cell centers in the pattern determined by the lattice shape; and the lattice dimension, which is a measure of the extent of the lattice, or how many unit cells form the lattice. The description of rectangular and hexagonal lattices in terms of these parameters is discussed in the following sections.

2.1 Rectangular Lattice

In a rectangular lattice, both the unit cells and the graph formed by joining unit cell centers with straight lines are rectangular. There are two lattice pitches, P_x and P_y, one each in the logical X and Y directions. The extent of a rectangular lattice is given by the number of unit cells in the X and Y directions, N_x and N_y. The total number of unit cells in a rectangular lattice is N_xN_y. The specification of unit cell types is as a vector of integer types; by convention, these integers identify unit cell types starting at the top left and proceeding right and down in the lattice. We place the origin of a rectangular lattice at the centroid of the bottom left cell. Fig. 2 shows an example specification of a rectangular lattice in these terms, defining its pitches, dimensions, and unit cell types. This method for specifying rectangular lattices is used as input to the tools described Section 3.

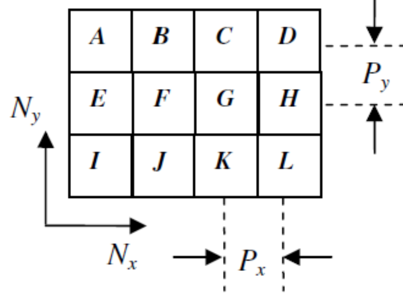
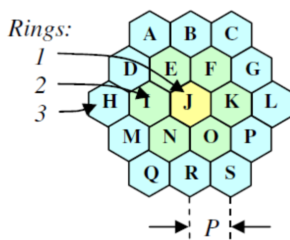


Fig. 1. Rectangular lattice with three unit cell parameters required for its specification.

2.2 Hexagonal Lattice

In a hexagonal lattice, the unit cells are hexagonal, while the lines joining unit cell centers form triangles. We restrict ourselves to a single hexagonal lattice pitch, P. The dimension of a hexagonal lattice is indicated by the number of rings, N_r, formed around a central unit cell, with the central cell identified as the first ring. The number of unit cells in a hexagonal lattice of N_r rings is $3(N_r)(N_r-1) + 1$. By convention, the central unit cell hexagon is oriented such that two of its vertices are placed on the +y and –y axes, and the +x and –x axes intersect two edges or "flats" of the unit cell. The unit cell types for a full hexagonal lattice can be specified similar to the specification of a rectangular lattice, starting with the top left unit cell and proceeding right and down. Fig. 3 shows an example specification for a hexagonal lattice.



$$Hexagonal, Symmetry = 1$$

$$N_{rings} = N_r = 3$$

$$P = 1.0$$

$$N_{TOT} = 3N_r\left(N_r - 1\right) + 1 = 19$$

$$Celltypes = ABC..$$

Fig. 2. Full hexagonal lattice with three unit cell parameters required for its specification.

For hexagonal lattices, a "full" lattice specifies a full 360 degrees; partial lattices can also be defined, where some whole-number division of 360

degrees is defined. We refer to these as “symmetry=x” lattices, where x is the number of divisions of the 360-degree lattice into the partial lattices. Both 60-degree (symmetry=6) and 30-degree (symmetry=12) partial lattices are supported. These are the two common types often used in reactor core simulations.

Two symmetric 60-degree sections can be described for a hexagonal lattice, depending on the orientation of the central unit cell. A “HexFlat” type has the central unit cell oriented in the conventional manner, with the +x axis bisecting a hexagon edge, while in a “HexVertex” type lattice a vertex of the central unit cell falls on the +x axis. HexFlat and HexVertex type lattices are shown in Figs. 4 and 5, respectively. The two types differ in the number of unit cells appearing in the partial lattice and in their arrangement around the central unit cell. For partial lattices, unit cell types are more easily specified by ring, starting from +360/x to zero degrees, from the central unit cell outward. Therefore, the unit cells for the lattice in Fig. 3 are specified in order of increasing letter.

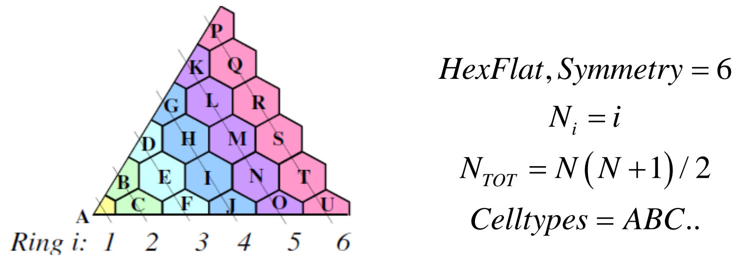


Fig. 3. HexFlat lattice type and specification.

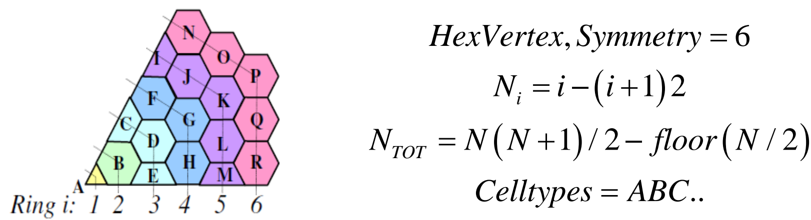


Fig. 4. HexVertex lattice type and specification.

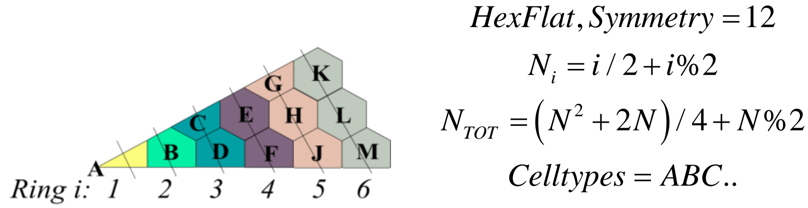


Fig. 5. HexFlat type describing a 30-degree (symmetry=12) lattice type and specification.

Only one symmetric 30-degree section can be described for a hexagonal lattice. Both the HexFlat and HexVertex schemes result in the same model with rotational symmetry. The HexFlat lattice type is used to describe a 30-degree or 1/12th symmetry of hexagonal lattice. N_i represents the number of unit cells in the *i*th ring. Unit cell types are specified from +30 to zero degrees, from the central unit cell outwards. Unit cells are specified as before, in order of increasing letter for the lattice in Fig. 5.

3 Generation of Assembly and Core Models

A mesh of a reactor core is produced by generating assembly geometry and mesh models for each unique assembly type, then copying/moving these assembly meshes into place in a core lattice. Our approach uses a sequence of tools to accomplish these tasks, so that the user can manually adjust output of one tool and input of the next tool at each stage of the process. The workflow for running these tools is depicted in Figs. 6 and 7. The first stage of this process uses the AssyGen tool, which generates an assembly geometry based on an assembly input file. This file describes the assembly as a lattice of cylindrical pins, annular regions surrounding each pin (e.g., for pin cladding), and overall assembly dimensions and surrounding duct wall characteristics. AssyGen also generates a meshing script for meshing the assembly.

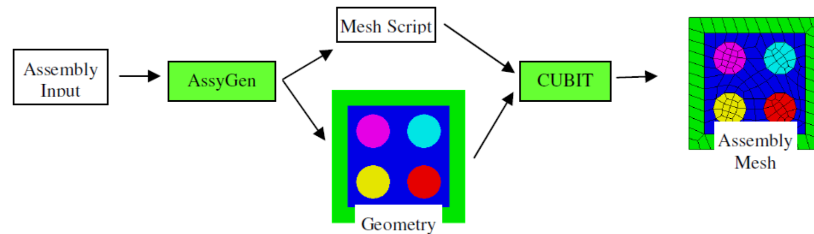


Fig. 6. First two stages of the geometry/mesh process, where AssyGen and CUBIT are executed for each assembly type.

In the second stage of execution, the meshing script and geometry output from AssyGen are used by CUBIT to generate a mesh for that assembly type. In many cases, this process is automatic, requiring no further input from the user. If users desire custom modifications to the mesh or if the geometry cannot be meshed automatically, the user can modify the mesh script to tailor the mesh generation to their individual needs or substitute their own script for generating the assembly mesh. This approach may be desirable, for example, if the user wants more mesh concentrated around fuel pins with the mesh size transitioning to the coarser mesh on the assembly boundary. The only explicit requirement on the resulting mesh is that the mesh on outer boundaries of the assembly match that of neighboring assemblies, or of the interstices material in cases where assemblies do not completely fill the core space.

In the final stage, shown in Fig. 7, the CoreGen tool reads an input file describing the arrangement of assemblies in the core lattice and locations of meshes for each assembly type; this information is used to generate the overall core mesh. CoreGen also produces a makefile, which can be used to automatically rerun parts of the process affected by changes to any of the input files.

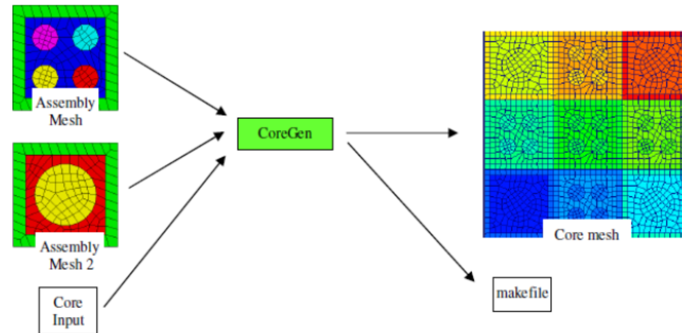


Fig. 7. Third stage of the geometry/mesh process, where CoreGen is executed.

3.1 Generating Assembly Models

A reactor core assembly is a lattice of unit cells. Each unit cell has zero or more concentric cylindrical layers of material, cut from a background material defined for each unit cell or for the assembly as a whole. The assembly can be surrounded by one or more layers of duct wall material. Information is input in a simple text-based language. An example input file for AssyGen is shown in Fig. 8. The lines with the GeometryType, Materials, and Dimensions keywords give the type and extent of the assembly and the material types used in unit cells. Multiple pincell types can be defined, each with one or more concentric cylinders of material and a background material for the cell. Following pincell input, the Assembly keyword line gives the overall lattice extents, followed by the pincell alias for each unit cell in that lattice. The overall assembly model can be modified based on the Center, Rotate, and Section keywords, and mesh sizes can be specified by using the RadialMeshSize and AxialMeshSize keywords. The top assembly in Fig. 7 was generated based on this input. For a full description of the AssyGen input language, see [10].

A few other options allow variations in the assembly model. The CellMaterial keyword results in unit cell boundaries being generated explicitly in the geometric model of the assembly; this allows finer control of the mesh in each unit cell. Cylinders input for individual pin cells can be larger than the unit cell dimensions; these volumes will overlap neighboring pincell regions when present. The Intersect keyword can be used to request that pincell contents be intersected with the pincell region, such that they do not overlap neighboring regions. Empty pin cells can be specified in the

assembly lattice by using the X or XX pincell alias; these are useful in regions neighboring those with overlapping contents.

AssyGen names the surfaces at the top, bottom, and sides of the assembly. They are named by using the material name with "_top," "_bot," and "_side" suffixes. This naming can be useful to the analyst. For example, for an assembly with upward flow, surfaces whose names end in "_bot"/"_top" can be placed in groups for application of inlet/outlet boundary conditions, respectively. Surfaces can be filtered by containing the volume using set Booleans, for example, to select fluid regions outside of cylindrical fuel pins.

```
Geometry Volume
GeometryType Rectangular
Materials 3 PWR_FC_F_01 M1 Mat_Block G1 Mat_Coolant C1
Dimensions 2 0.0 0.0 0 124.0 18.0 18.0 23.5 23.5  G1 C1
Pincells 1
! Standard fuel pin cell
PWR_FC_01 FC 2
Pitch  9.0 9.0 124.0
Cylinder 1 0.0 0.0  0.0 124.0   3.0  M1
!
Assembly 2 2
FC FC
FC FC
Center
RadialMeshSize 2.0
```

Fig. 8. Example AssyGen input file.

3.2 Meshing Assembly Geometric Models

In addition to generating the geometric model, AssyGen writes two meshing scripts. The first contains the basic commands necessary to mesh the assembly model and define various material groups and boundary condition sets. The second defines several parametric variables used in the first meshing file; parameters are defined for mesh sizes and various user-selectable mesh schemes.

In many cases, the user will need only to specify a mesh size in the AssyGen input file, after which the meshing process for an assembly will be completely automatic. In other cases, for example, if a relatively coarse mesh size is requested or finer-grained control over mesh density or quali-

ty is required, users can modify the meshing scripts file as necessary before running CUBIT on that file. The syntax used with these journal files can be found in [10].

This process of constructing assembly meshes, while somewhat automated, can be sensitive to mesh size input by the user. This problem is manifested particularly in the generation of unstructured quadrilateral meshes for the top surface in an assembly, which is cut by large numbers of cylindrical rods. The test problem input for the more complex example in this paper contains a carefully chosen radial mesh size that, if made larger (giving a coarser mesh), will cause mesh generation for that surface to fail. This situation is common in mesh generation technology, where mesh generation robustness grows worse for coarser meshes. This is one of the reasons for splitting the core mesh generation process into several steps: to allow user intervention midway through the process. The general solution for this type of failure would be either to make the mesh size finer (producing many more elements), or to perform geometry decomposition of the assembly top surface such that the resulting smaller, less-complex surfaces are meshable with more primitive algorithms. Because CUBIT is a closed-source code, efforts are being made to support lattice-type mesh generation with the MeshKit library described in Section 4.

3.3 Generating Core Model

A reactor core is formed by placing various assembly types in a lattice, possibly surrounded by material in the interstices regions. CoreGen supports construction of full rectangular, full hexagonal, 60-degree hexagonal, and 30-degree hexagonal lattices. Two 60-degree hexagonal variants, HexVertex and HexFlat, are supported; these lattice types are described in Section 2.

Input for defining a core is similar to that of an assembly. An example input file for the CoreGen program is shown in Fig. 9. CoreGen uses Geometry and GeometryType to specify the dimension and the type of core model specified, respectively. The symmetry keyword is used to indicate the desired symmetry. Assemblies and Lattice keywords provide information about the assembly meshes and dimensions and the arrangement of the assemblies forming the core, respectively. The CoreGen program also generates a makefile. If AssyGen, CoreGen, and CUBIT are in the user's path, this makefile automates generation of the overall mesh; after making changes to either of the AssyGen or CoreGen input files, the user can rerun "make" to rebuild the entire core model. We note here that CoreGen operates on mesh and makes no use of the geometric models pro-

duced by AssyGen; those models are for the sole purpose of generating the mesh model for each assembly type. After the assembly meshes have been copied/moved into the core lattice, coincident nodes are merged, resulting in a contiguous mesh. Syntax and keywords used to specify input to CoreGen and the makefile can be found in the MeshKit repository [10].

```
Geometry Volume
GeometryType Rectangular
Symmetry 1
Assemblies 2   23.5 23.5
s1.cub S1
s2.cub S2
Lattice 3 3
S2 S1 S2 &
S1 S1 S1 &
S2 S1 S2
END
```

Fig. 9. Example CoreGen input file.

4 Implementation

The tools described in this paper rely on geometry and mesh libraries developed as part of the Interoperable Tools for Advanced Petascale Simulations project. The Common Geometry Module (CGM) [11] provides functions for constructing, modifying, and querying geometric models in solid model-based and other formats. While CGM can evaluate geometry from several underlying geometry engines, this work relies mostly on ACIS [12], with an Open.Cascade-based [13] version under development. Finite-element mesh and mesh-related data are stored in the Mesh-Oriented datABase (MOAB) [14]. MOAB provides query, construction, and modification of finite-element meshes, plus polygons and polyhedra. Mesh generation is performed by using a combination of tools. The CUBIT mesh generation toolkit [15] provides algorithms for both tetrahedral and hexahedral mesh generation. MeshKit, an open-source mesh generation library under development at Argonne National Laboratory, provides efficient algorithms for mesh copy/move/merge [16]. CGM and MOAB are accessed through the ITAPS iGeom and iMesh interfaces, respectively.

The iMesh concept of sets and tags is important to the implementation of these tools. A set is an arbitrary collection of entities and other sets; a tag is data annotating entities, sets, or the interface itself. The combination of sets and tags is a powerful mechanism used to describe boundary conditions, material types, and other types of metadata commonly found with mesh.

5 Metadata Handling

Mesh files for any simulation contain boundary condition, material type and other user-specified information. When different assembly meshes are copied/moved to form the core, the metadata associated with individual assembly meshes must be correctly propagated to the core model. For example, if an assembly mesh defines materials for elements in the mesh, these definitions should be propagated with the copies of those elements.

MeshKit abstracts the handling of metadata by using the concepts of “copy set” and “expand set.” For any set designated as a copy set, a copy of that set is made and populated with element copies whose original element was in the original set. Any set designated as an expand set receives copies of original entities already in that set. Together, these abstractions simplify the implementation of meshing algorithms such as copy/move. The application just needs to identify characteristics used to identify each set type; the copy/move implementation then finds sets with those charac\teristics and acts accordingly. The abstraction enables this behavior without requiring the copy/move implementation to understand the semantics of those sets, for example, what purpose a material set fills for an application. The copy/expand set concept also could be used in other meshing applications, for example, adaptive mesh refinement or topological mesh improvement.

6 Examples

In this section we present two examples; one a simple hexagonal assembly and the other a more complex core model.

6.1 Single Hexagonal Assembly

Our first example, a single Very High Temperature Reactor (VHTR) assembly, has six pincell types, with three sizes of pins representing fuel, burnable poison, and coolant regions; a structure at the center describes a fuel handling hole, and a larger hole in the lower half of the assembly represents a control rod and guide tube. Empty pin cells are specified in cell positions surrounding these larger holes. The resulting geometric model is shown in Fig. 10 (left). We note that generating an all-hexagonal mesh for this model is difficult because of the many cylindrical pins cut out of the top surface of the block. A fine mesh can be generated without too much difficulty; however, generating a coarse mesh for this model would require substantial effort to make targeted modifications to the assembly geometry and mesh parameters. The model generated by AssyGen was used as input to UNIC, a neutron transport code [17] developed at Argonne National Laboratory. Fig. 10 (right) shows the thermal neutron flux computed. The flux is much lower in the region of the large control rod and slightly lower around the six burnable poison pins (located at corners of the hexagonal assembly), as expected. Note that in this case a tetrahedral mesh was used, which was generated by modifying input to the CUBIT journal file output by AssyGen.

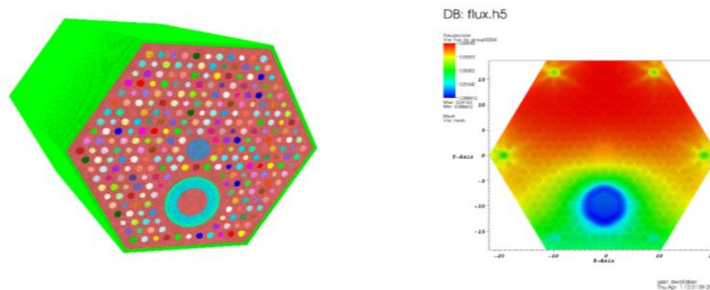


Fig. 10. Hexagonal assembly geometry and mesh constructed by AssyGen (left); thermal neutron flux computed by the UNIC neutron transport calculation for this hexagonal assembly (right).

6.2 Complex Core Model

Fig. 11 and Fig. 12 show a more complex example, a core model created using the three-stage process described in this paper. Fig. 11 shows four of the 11 assembly types used in this model. Keyword-based input files were used with AssyGen to generate the assembly geometry. CUBIT was then used to mesh the assemblies using the journal files created by AssyGen. The 1/6 core model in Fig. 12 was generated by CoreGen. This model has approximately 12 million hexahedral elements and 14 million mesh vertices. The tools were run on a desktop Linux-based workstation with a clock speed of 2.5 GHz and 11.8 GB RAM. It took 4 minutes to generate the geometries, 5 minutes to create the assembly meshes, and 20 minutes to copy/move and merge the assemblies and create the core.

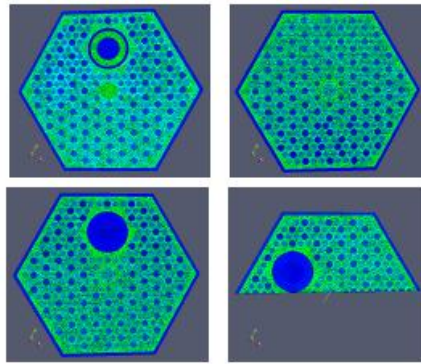


Fig. 11. Four assembly types produced by AssyGen.

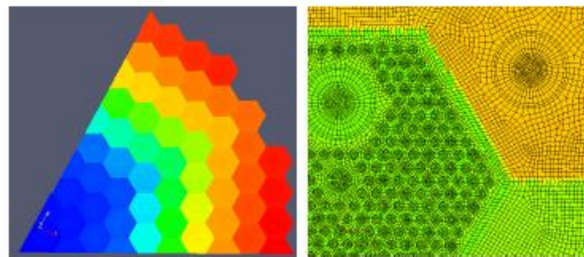


Fig. 12. One sixth of a VHTR core model generated using CoreGen (left); a closeup of several assemblies in this model (right).

7 Conclusions

An input-file-based methodology has been developed that can generate a lattice-based reactor core mesh with minimal user input. The methodology operates in three stages. First, assembly models of various types are generated by the AssyGen tool, based on input describing the content of pin cells, the arrangement of pin cells in the lattice, and the extent of the lattice and any surrounding material. The assembly model or models then are meshed with the CUBIT mesh generation toolkit, optionally based on a journal file output by AssyGen. After one or more assembly model meshes have been constructed, they are arranged in a core model by using the CoreGen tool. Although this approach is ideally suited for lattice-based core geometries, it also offers flexibility to incorporate non-lattice-based features.

ACKNOWLEDGMENTS

We thank M. A. Smith, A. Wollaber, and J. H. Thomas in the Nuclear Engineering Division at Argonne National Laboratory for helpful discussions and feedback for creating the input file language used in these tools. We also thank the Fathom group at Argonne, who maintain the libraries required by this tool. This work is sponsored by the U.S. Dept. of Energy Office of Nuclear Energy GenIV Program; by the U.S. Dept. of Energy Office of Scientific Computing Research, Office of Science, and by the US Department of Energy's Scientific Discovery through Advanced Computing program, under Contract DE-AC02-06CH11357.

REFERENCES

1. Sandia news notes (2005) http://www.cs.sandia.gov/newsnotes/2005newsnotes.html#Goodyear
2. S. Fung, A. Adler, and A. D. C. Chan (2010) "Using Dismesh as a mesh generating tool for EIT," Journal of Physics, Conference Series.
3. J. R. Cebral and R. Lohner (1999) "From medical images to CFD meshes," in Proc. 8th International Meshing Rountable, pp. 321-331.
4. B. K. Sony and M. H. Shih (1991) "TIGER: Turbomachinery Interactive Grid GenERation," in Proc. Third International Conference of Numerical Grid Generation in CFD, Barcelona, Spain.

5. R. V. Chima (2008) TCGRID website, http://www.grc.nasa.gov/WWW/5810/rvc/tcgrid.htm
6. List of meshing software website http://www-users.informatik.rwth-aachen.de/~roberts/software.html
7. PSG2 / Serpent website (2010) http://montecarlo.vtt.fi/
8. K.A. V. Riper (1993) "SABRINA: Three-dimensional geometry visualization code system," PSR-242, ORNL-RSICC, Oak Ridge, TN.
9. G. Hansen and S. Owen (2008) "Mesh generation technology for nuclear reactor simulation; Barriers and opportunities," Journal of Nuclear Engineering and Design. pp. 2590-2605.
10. MeshKit README website (2010) http://trac.mcs.anl.gov/projects/fathom/browser/MeshKit/trunk/rgg/README
11. T. J. Tautges (2005) "CGM: A geometry interface for mesh generation, analysis and other applications," Engineering with Computers, 17, pp. 486-490.
12. Spatial website (2010) http://www.spatial.com/
13. Open CASCADE Technology website (2000-2010) http://www.opencascade.org.
14. C. Ollivier-Gooch, L. F. Diachin, M. S. Shephard, and T. Tautges (2007) "A language-independent API for unstructured mesh access and manipulation," in Proc. 21st International Symposium on High Performance Computing Systems and Applications, IEEE, p. 22
15. G. D. Sjaardema, T. J. Tautges, T .J. Wilson, S. J. Owen, T. D. Blacker, W. J. Bohnhoff, T. L. Edwards, J. R. Hipp, R. R. Lober, and S. A. Mitchell (1994) CUBIT mesh generation environment, volume 1: Users manual, Sandia National Laboratories, Albuquerque, NM.
16. MeshKit website (2010) http://trac.mcs.anl.gov/projects/fathom/browser/MeshKit
17. M. A. Smith, C. Rabiti, G. Palmiotti, D. Kaushik, A. Siegel, B. Smith, T. Tautges, and W. S. Yang (2007) "UNIC: Development of a new reactor physics analysis tool," in Proc. Winter Meeting on International Conference on Making the Renaissance Real, American Nuclear Society, pp. 565-566.