ANS MC2015 - Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method · Nashville, Tennessee · April 19–23, 2015, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2015)

OPENCG: A COMBINATORIAL GEOMETRY MODELING TOOL FOR DATA PROCESSING AND CODE VERIFICATION

William Boyd, Benoit Forget, and Kord Smith
Massachusetts Institute of Technology
Department of Nuclear Science and Engineering
77 Massachusetts Avenue, Cambridge, MA 02139
wboyd@mit.edu; bforget@mit.edu; kord@mit.edu

ABSTRACT

Combinatorial Geometry (CG) is one formulation for computational geometric models that is commonly used in many neutron transport simulation codes. The use of CG is advantageous since it permits an accurate yet concise representation of complex reactor models with a nominal memory footprint. OpenCG is a software package for combinatorial geometry models being developed at the Massachusetts Institute of Technology. The goal for OpenCG is to provide an easy-to-use, physics agnostic library to build geometry models of nuclear reactor cores. OpenCG is a free, open source library with an easy-to-use Python interface to provide nuclear engineers a single, powerful framework for modeling complex reactor geometries. Compatibility modules for commonly used nuclear reactor physics codes, such as OpenMC, OpenMOC, and Serpent, are being concurrently developed for rapid and easy exportation of an OpenCG model directly into the relevant input file format for each code of interest. The present work describes OpenCG and describes some of the novel and useful algorithms included with the software package.

Key Words: combinatorial geometry, code verification, data processing

1 INTRODUCTION

Many neutron transport codes including MCNP [1], Serpent [2], OpenMC [3] and OpenMOC [4] utilize the Combinatorial Geometry (CG) formulation to model complex geometries. CG is also often referred to as Constructive Solid Geometry (CSG) in the neutron transport literature. CG offers many unique advantages as an approach to computational geometry for neutron transport simulations since it:

- Permits description of an *arbitrarily accurate* unstructured geometric mesh
- Provides a *compact representation* with minimal input description
- Represents $\mathcal{O}(n)$ components with $\mathcal{O}(\log n)$ memory requirements
- Utilizes a hierarchical *tree data structure* with scalable $\mathcal{O}(\log n)$ traversals

Although many codes utilize CG, it is overly burdensome to manually write geometric input files for multiple simulation tools for code verification of a single reactor model. In addition, the compact representation is not well suited for data analysis without some of the new advanced

algorithms presented in Sec. 4. For example, burnup calculations for a full-core Pressurized Water Reactor (PWR) with Monte Carlo will require millions of tally depletion zones [5]. To model such a problem, the CG implementations in many codes would have an exorbitant memory footprint and create considerable difficulties for the user to perform tally data analysis. This paper presents a new combinatorial geometry modeling library called OpenCG to accelerate the building of complicated reactor geometries, enable rapid cross-code verification and facilitate large scale data processing.

2 COMBINATORIAL GEOMETRY

Combinatorial geometry allows complex spatial models to be built using boolean operations - such as intersections and unions - of simple surfaces and building blocks termed *primitives*. The CG approach is well suited for reactor models which typically are highly structured and contain repeating patterns. This is the case for commercial PWRs and BWRs whose cores are frequently built out of a rectilinear array of fuel assemblies, each of which is a rectilinear array of fuel pin universes. Similarly, sodium cooled fast reactors (SFRs), high-temperature gas-cooled reactors (HTGRs), and CANDU heavy water reactors make use of hexagonal and cylindrical arrays. This section presents a brief overview of the CG formulation used in the OpenMOC code [6].

In many CG implementations, the most fundamental primitive is termed a *surface*. In this paper, a 3D surface is defined as the set of points that satisfy $f(x, y, z) = 0$ for some function $f(\cdot)$ that will henceforth be termed the *potential function* of the surface. The potential divides the 3D spatial domain into two *halfspaces*. The set of coordinates for which $f(x, y, z) > 0$ is called the *positive* halfspace while those coordinates for which $f(x, y, z) < 0$ collectively form the *negative* halfspace. Typically, neutron transport codes provide *quadratic* surface primitives - such as planes, cylinders, spheres and cones - though any other surface type with a computable potential may also be used.

A *cell* is defined to be the region bounded by a *boolean combination* of surface halfspaces. Surface halfspaces may be composed to form cells with union, intersection and difference operations, as well as rotations and translations. A *universe* is a collection of one or more cells that fill the entirety of the spatial domain. Each cell may be filled with a *material* or a separate *nested* universe. Universes allow unique structures to be created from cells (*e.g.*, fuel pins), and for simple replication of those structures throughout a model by placing them in various locations throughout the geometry.

Lattices are an extremely useful construct for modeling regular, repeating structures. This is especially true for reactor cores which are often composed of rectilinear, hexagonal or cylindrical arrays of fuel pins. For this reason, lattices are a common structure in many neutron transport codes. Each lattice is uniquely specified by the number of array elements along its axes ((x, y, z) for rectilinear, (x', y', z') for hexagonal and (r, z, θ) for cylindrical lattices), the width and height of each lattice cell, and the universe filling each lattice cell. The lattice specification represents a coordinate transformation such that the center of each lattice cell maps to the origin of the universe within it. This allows for a single universe to be replicated in some or all lattice cells without redundantly storing the universe many times in memory.

3 OPENCG

A new general purpose software tool called OpenCG is currently under development at the Massachusetts Institute of Technology. OpenCG is a Python Application Programming Interface (API) which provides the basic combinatorial geometry primitives (such as surfaces, cells, universes and lattices) needed to construct complex reactor models. OpenCG is designed using the object-oriented programming paradigm. In addition, OpenCG leverages the extensive Python ecosystem for high performance scientific computing, including `NumPy` [7] for vectorized data array manipulation and `Matplotlib` [8] for visualization.

OpenCG was created to provide a flexible platform for the algorithmic development necessary for combinatorial geometry to be relevant for full core reactor burnup calculations. To the authors' knowledge, no open source combinatorial geometry modeling tool yet exists which is capable of modeling the full 3D complexity of today's Light Water Reactor cores with minimal memory and compute resources. To this end, OpenCG was created to facilitate the following:

- Large scale data processing on irregular CG meshes
- Verification of results between transport codes
- High-fidelity geometric parameter optimization

The OpenCG package implements several useful algorithms to enable large scale data processing as overviewed in Sec. 4. In addition, compatibility modules provide rapid and easy exportation of a Pythonic OpenCG geometry model to common neutron transport code input files as presented in Sec. 5 for code verification. Similarly, the Python software model permits greater freedom for geometric parametrized optimization than can be easily achieved with traditional ASCII input files.

OpenCG also provides meshing features to subdivide cells and/or universes. This can be particularly useful for deterministic neutron transport codes on a highly discretized spatial mesh - such as radially and/or angularly subdivided mesh zones in pin cells, and linearly- or logarithmically-spaced mesh zones in reflector regions.

OpenCG is a particularly useful platform for students in both classroom and research settings. Due to its modular design, OpenCG is highly extensible and will permit the implementation of new, specialized primitive types (*e.g.*, new surface types, cell operations, meshing, etc.). The Python framework allows users and developers to interact with OpenCG through the `IPython` and/or `IPython Notebook` [9] interface. This can be tremendously useful when debugging new geometries or the CG implementation of a particular code.

OpenCG uses the `Git` revision control system and an open source distribution is hosted on GitHub at `https://github.com/mit-crpg/OpenCG`. The build system and configuration management for OpenCG is handled using Python's `Distutils` package, which is provided by default with all modern Python distributions. The source code for OpenCG will be reviewed by the MIT Technology Licensing Office for open source release under the MIT License in 2015.

4 UNIQUE ALGORITHMS

Combinatorial Geometry has existed as a field of mathematics and computer science for many decades. Over that time, a great many useful algorithms have been developed which leverage the CG formulation for various computational applications [10, 11]. OpenCG implements a small subset of those algorithms which are most useful for neutron transport reactor physics applications. In this section, an overview of the most unique CG algorithms in OpenCG are presented, including arbitrary cell volume calculations (Sec. 4.1) and unique region classification (Sec. 4.2). To the authors' knowledge, this paper also presents novel algorithms to identify local neighbor symmetries (Sec. 4.3) and to perform region differentiation (Sec. 4.4) in CG.

4.1 Cell Volume Calculation

Many reactor physics calculations which utilize depletion solvers or thermal hydraulic feedback require *cell volumes*. Some transport codes can calculate cell volumes stochastically or analytically for certain components, but do not provide volume calculators for arbitrarily formed CG cells. Although some tools allow/require a user to input cell volumes based on *a priori* knowledge of the model, this is less than desirable since it is not a scalable or flexible workflow model.

Recently, Millman et al. developed a hybrid analytic-stochastic divide-and-conquer algorithm to compute arbitrary cell volumes with a high degree of accuracy [12]. This algorithm is incorporated into the MC21 transport code. An implementation of Millman's algorithm is in progress in OpenCG and is expected to be publicly available in an early release version of the library. However, a purely stochastic algorithm is currently included in OpenCG which can compute cell volumes with arbitrary accuracy (albeit slowly).

The current volume calculation algorithm in OpenCG is described by Alg. 1, Alg. 2 and Alg. 3. This algorithm uses a recursive technique to traverse the CG tree data structure of universes, lattices, and cells and compute the *accessible volume* at each level of the hierarchy. The algorithm accomplishes this by implicitly and recursively subdividing and passing bounding box volumes to each bounded primitive (cells, universes or lattices) child node in the CG tree. These bounding boxes are used by a rejection sampling technique to compute arbitrary cell volumes in Alg. 3. It should be noted that this algorithm will compute the area rather than the volume for geometries with an infinite extent in one dimension (*e.g*, an infinitely long fuel assembly along the z-axis).

A volume calculation for all bounded primitives in a geometry can be initiated by a single function call in the OpenCG API. In particular, Alg. 1 is called with the *root* universe $univ$ - *e.g.*, the universe at the root node of the CG tree data structure. In addition, the user must define the total volume vol of the geometry - which is generally much easier to compute *a priori* than the individual cell volumes - as well as the tolerance tol or maximum relative uncertainty acceptable for each volume. The algorithm then computes the volumes for all cells within the root universe (Alg. 3), which may each have a nested universe or lattice fill. The termination case for the algorithm is reached for cells filled with materials rather than nested universes or lattices.

Algorithm 1 Universe Volume Calculation

procedure COMPUTEUNIVERSEVOLUME(universe $univ$, volume vol, tolerance tol)
  for all $cell$ **in** $univ$ **do**
    $cell\_vol \leftarrow$ COMPUTECELLVOLUME($cell$, vol, tol) ▷ Alg. 3
    $fill \leftarrow cell.fill$
    if type($fill$) **is** LATTICE **then**
      COMPUTELATTICEVOLUME($fill$, $cell\_vol$, tol) ▷ Alg. 2
    else if type($fill$) **is** UNIVERSE **then**
      COMPUTEUNIVERSEVOLUME($fill$, $cell\_vol$, tol) ▷ Recursively call Alg. 1
    end if
  end for
end procedure

Algorithm 2 Lattice Volume Calculation

procedure COMPUTELATTICEVOLUME(lattice $latt$, volume vol, tolerance tol)
  $vol \leftarrow {}^{vol}/_{latt.num\_cells}$ ▷ Lattice cell bounding box volume
  for all $univ$ **in** $latt$ **do**
    COMPUTEUNIVERSEVOLUME($univ$, vol, tol) ▷ Alg. 1
  end for
end procedure

Algorithm 3 Cell Volume Calculation

procedure COMPUTECELLVOLUME(cell $cell$, volume vol, tolerance tol)
  $count \leftarrow 0$
  $samples \leftarrow 0$
  $unc \leftarrow \infty$
  $box\_vol \leftarrow cell.width\_x \times cell.width\_y \times cell.width\_z$ ▷ Bounding box volume
  while $unc \geq tol$ **do**
    $samples \leftarrow samples + 1$
    $x \sim \mathcal{U}(cell.min\_x, cell.max\_x)$ ▷ Uniformly sample random variable
    $y \sim \mathcal{U}(cell.min\_y, cell.max\_y)$ ▷ Uniformly sample random variable
    $z \sim \mathcal{U}(cell.min\_z, cell.max\_z)$ ▷ Uniformly sample random variable
    if (x, y, z) **in** $cell$ **then**
      $count \leftarrow count + 1$
    end if
    $frac \leftarrow {}^{count}/_{samples}$ ▷ Volume fraction within bounding box
    $unc \leftarrow box\_vol \times ({}^{frac - frac^2}/_{samples})^{\frac{1}{2}}$ ▷ Compute relative uncertainty
  end while
  $vol \leftarrow vol \times {}^{count}/_{samples}$ ▷ Compute cell volume
  $cell.vol \leftarrow vol$ ▷ Set cell volume
  return vol
end procedure

4.2 Unique Region Classification

High-fidelity transport simulations of full core reactors will require millions of depletion zones. This presents a number of challenges for CG for full core reactor simulations. In an ideal scenario, a unique geometric cell should only be stored once in memory but may be referred to in multiple locations in the geometry. Each cell *instance* may experience different radiation and/or thermal hydraulic conditions and this must be properly taken into account. Hence, in order for CG be effective, the code must classify (or index) each unique instance of the cell throughout the geometry.

An algorithm was recently developed by Lax et al. to classify unique regions (*e.g.*, cell instances) in combinatorial geometries [13] with a minimal memory overhead. Lax's algorithm has been implemented in OpenMC for *distributed tallies* for each instance of a cell. OpenMC's distributed tallies have been used to compute pin powers for the BEAVRS benchmark [14] where each fuel pin cell has $\sim$20,000 unique instances throughout the core. In addition, a variation of the algorithm is used by OpenMOC to categorize unique cell instances as flat source regions [4].

OpenCG also includes an implementation of this region classification algorithm to assign each region a unique positive integer ID. The OpenCG API provides the following functionality:

- Compute the total number of regions in a geometry
- Build a *path* for a given region ID
- Find the region ID for a given *path*
- Find the region ID given (x, y, z) coordinates
- Find all region IDs for a given cell

The *path* for a given region ID represents the path taken from the root universe node through the CG tree data structure of universe, lattice and cell nodes to reach some leaf node (*e.g.*, an instance of a cell filled by a material). Paths may be represented as a linked list with each node in sequence. The OpenCG API for region classification is a key workflow element for downstream data processing of distributed tallies from the OpenMC code. In addition, the region classification API can be used for data processing of reaction rates on OpenMOC's spatially discretized flat source region CG mesh.

4.3 Local Neighbor Symmetry (LNS) Identification

In some common use cases, it may be beneficial for a CG formulation to identify *neighbor* cells, or pairs of cells which are adjacent to one another. For example, Monte Carlo transport, as well as some deterministic methods (*e.g.*, MOC, CPM), rely heavily on ray tracing of neutrons through a geometric model. Ray tracing in CG may be optimized through the use of neighbor cells as a heuristic to predict the sequence of cells that each ray intersects. Furthermore, the concept of neighbor cells can be used in lattice physics calculations to identify regions which may experience similar spectral self-shielding effects. In particular, neighbor cells provide a heuristic to identify fuel pins which may have similar multi-group cross-sections in a fuel assembly (*e.g.* discriminate pins next to 0, 1, or 2 guide tubes, etc.). A novel algorithm to systematically accomplish both of these objectives, termed *Local Neighbor Symmetry* (LNS) identification, is implemented in OpenCG.

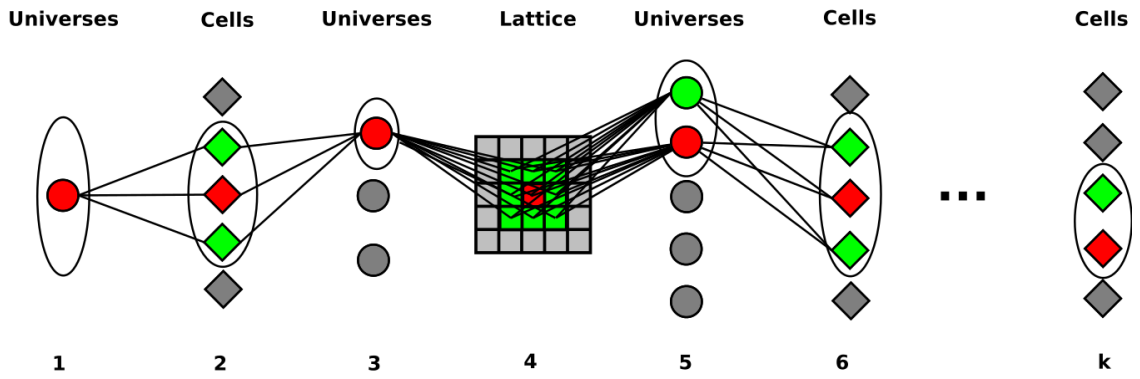

Figure 1. An example k-partite graph structure used to identify local neighbor symmetries. Red nodes correspond to the universes/cells encapsulating a region of interest, green nodes correspond to the neighbors of that region, and gray nodes correspond to universes/cells which are not neighbors. Red and green nodes at each level are combined into an argument for a hash function to generate a local neighbor symmetry identifier for the region of interest.

The LNS identification algorithm is described in Alg. 4 to identify the unique symmetry for the path to a region in a combinatorial geometry. LNS identification performs a *Breadth-First Search* (BFS) to find neighbors on each level of the CG tree. For example, BFS is used to find neighbor cells for a particular cell within a universe. Similarly, BFS is used to find neighbor universes adjacent to a particular lattice cell. The neighbor cells and universes on each of the k levels of a CG tree are connected to form a *k-partite graph* as depicted in Fig. 1. Finally, the k-partite graph is used as an argument to a *hash function* to compute the LNS identifier (*e.g*, a non-negative integer) for the particular region represented by the path. This algorithmic formulation is general to any arbitrary combinatorial geometry, including those with nested rectilinear, hexagonal or cylindrical lattices.

Algorithm 4 Local Neighbor Symmetry Identification

procedure COMPUTENEIGHBORSYMMETRY($path$)
  $G \leftarrow \emptyset$ ▷ Initialize empty set for graph
  $k \leftarrow$ **length**($path$) ▷ Find number of independent sets
  for $i := 1, k$ **do**
    if type($path[i]$) **is** UNIVERSE **then**
      $G \leftarrow G \cup \{path[i]\}$ ▷ Append universe to graph
    else if type($path[i]$) **is** LATTICE **then**
      $N \leftarrow$ BREADTHFIRSTSEARCH($path[i]$) ▷ Find lattice cell neighbors
      $G \leftarrow G \cup \{N\}$ ▷ Append neighbors to graph
    else if type($path[i]$) **is** CELL **then**
      $N \leftarrow$ BREADTHFIRSTSEARCH($path[i]$) ▷ Find cell neighbors
      $G \leftarrow G \cup \{N\}$ ▷ Append neighbors to graph
    end if
  end for
  return HASH(G) ▷ Return k-partite graph hash
end procedure

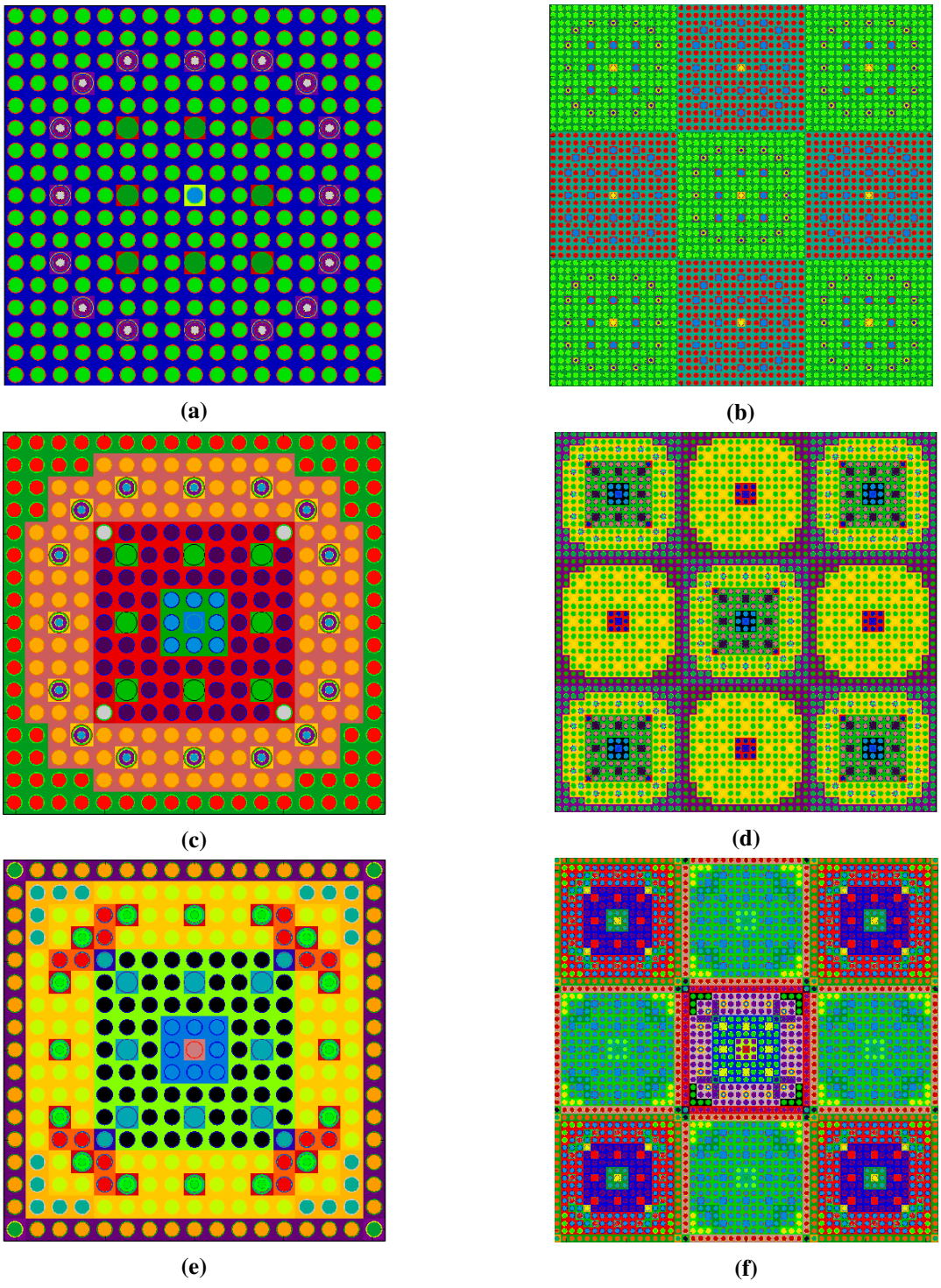

Figure 2. Two rectilinear lattice geometries are depicted to illustrate the use of local neighbor symmetry identification. The *cells* are depicted for (a) a 17×17 PWR lattice and (b) a 3×3 colorset of two different 17 × 17 PWR assemblies each with burnable absorbers, guide tubes and instrument tubes. The *unique neighbor* symmetry identifiers are color-coded in (c) and (d) for the assembly and colorset, respectively. Likewise, the *general neighbor* symmetry identifiers are color-coded in (e) and (f).

The LNS identification algorithm in OpenCG uses Python's `hashlib.sha1` implementation of the SHA-1 hash function [15]. The Cell/Universe IDs and Lattice cell indices which represent the red and green nodes in the k-partite graph are assembled into an immutable and hashable NumPy "view" desriptor which is used as the argument to the hash function. It should be noted that OpenCG's LNS identification does not explicitly handle hash collisions - *e.g.*, scenarios in which two cells with different neighbors hash to the same value. The SHA-1's 160-bit hash value makes a hash collision extremely unlikely even for the most complicated reactor models with thousands of unique LNS identifiers. However, it would be a straightforward modification to account for the possibility of hash collisions should issues arise in the future.

Several parameters are incorporated into OpenCG's LNS API to provide the user with various methods to adjust the number of symmetries discovered by the algorithm. For example, OpenCG includes *general neighbor* and *unique neighbor* identifiers. General neighbors includes all neighbor cells/universes found in BFS - including duplicates - as separate nodes in the k-partite graph. For example, in a lattice, a single universe may be placed multiple times around a lattice cell, each of which will be independently discovered by BFS and replicated as a distinct node within the graph. On the contrary, unique neighbors does not include duplicate cells/universes - only unique cells/universes are represented by distinct nodes in each independent set in the k-partite graph. A few diagrams of general and unique LNS applied to two geometries are depicted in Fig. 2.

Additional user options in the OpenCG LNS API include the ability to truncate the first i and/or last j independent sets ($i + j \leq k$) from the graph before hashing it to compute the LNS identifier. This option has the effect of localizing LNS to particular levels of the CG tree. Finally, the user may set the depth for the BFS at each level in the CG tree. This allows LNS to search for cells/universes which may be further removed (*e.g*, not adjacent) from a particular region of interest.

4.4 Region Differentiation

As previously noted, one of the advantages of combinatorial geometry is that it can take advantage of patterned structures with repeating primitives. In certain use cases, however, it may be necessary to replicate certain primitives which have the same geometric properties but experience very different radiation and/or thermal hydraulic conditions, and hence have different material properties in a transport simulation. For example, the Local Neighbor Symmetry algorithm is useful for identifying groups of fuel pin cell instances which may have similar multi-group cross-sections in lattice physics calculations. However, in order for a transport code to make use of LNS, a replica of each cell must be made to represent each of its different LNS identifiers.

The process of manually constructing a combinatorial geometry with many replicated but geometrically identical cells is very time consuming and prone to errors. This paper presents a novel algorithm termed *Region Differentiation* which efficiently and systematically reconstructs a combinatorial geometry with replicated cells. A characterization of the Region Differentiation algorithm applied to a CG tree data structure is shown in Fig. 3.

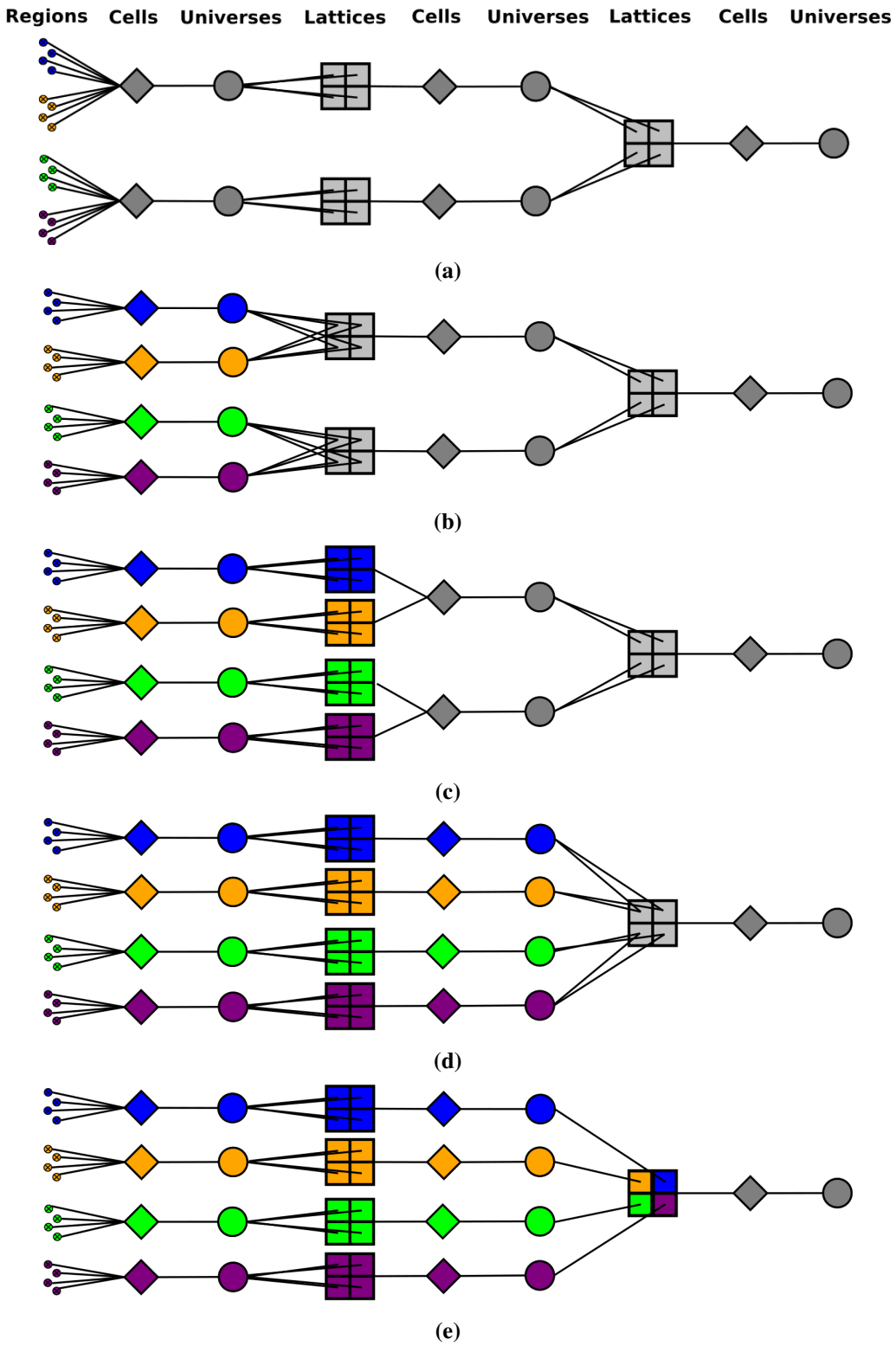

Figure 3. A few of the stages of the region differentiation algorithm. The regions (cell instances) to be differentiated are grouped and colored blue, orange, green and purple in (a). The first levels of cells and universes for each region group are differentiated in (b). The same is done for the lattices in (c). The algorithm continues to recursively differentiate cells, universes and lattices until no region groups collide at any level of the CG tree in (e).

The region differentiation algorithm is implemented in OpenCG and presents an interface which takes in a set of arbitrarily formed *region groupings*. A region grouping is the set of all region IDs that reference a particular cell in the geometry. Two or more region groupings corresponding to the same cell designate specific cell instances that should be replicated. The algorithm replicates cells, universes, and even lattices for each region grouping.

A naive or brute-force implementation of the region differentiation algorithm would scale as $\mathcal{O}(kn!)$ in both memory and time for k nested universe/cell levels and n region groupings. The reason is that *a priori*, the algorithm does not know from which region groupings various cell instances will combine with one another to form universes, lattices and/or cells, which must themselves be differentiated for each possible combination of region groupings. To avoid factorial scaling, OpenCG makes use of *dynamic programming* to efficiently differentiate intrinsics one level at a time within the CG tree.

The region differentiation algorithm iterates over each level of nested universes and cells. At each step, the paths for each region starting from the current level in the CG tree and ending with the root universe node are hashed and stored in a hash table linked to the region grouping. Next, the algorithm manages *intrinsic collisions* when two or more region groupings with the same path in the hash table point to the same intrinsic (cell, universe or lattice). To resolve intrinsic collisions, the algorithm differentiates the intrinsic for each region grouping involved in the collision. As intrinsic collisions are resolved, the algorithm *merges* any region groupings with paths that hash to same value in the hash table. The algorithm's termination condition is reached when the hash table only has one entry - *i.e.*, paths for all regions hash to the same value.

5 COMPATIBILITY MODULES

OpenCG provides compatibility modules to export geometries to many different transport codes' input file formats as depicted in Fig. 4. The concept behind the compatibility modules is to allow users to build a single geometry using OpenCG's Python API and to export that geometry to whichever transport code a user may wish to use. This can greatly accelerate verification and/or uncertainty quantification between codes since only a single geometry must be constructed. In addition, OpenCG compatibility modules facilitate a dynamic workflow of data processing and transfer between multiple codes. Fig. 5 illustrates how OpenCG may be used to generate input files for multi-group cross-section calculations in OpenMC, and then transfer that data on a CG mesh for a deterministic OpenMOC calculation.

A transport code must have a combinatorial geometry API (Python is the preferred language) as a key pre-requisite for the OpenCG compatibility module concept. For example, OpenMC includes a Python API with object-oriented CG primitives. This API's primitives include routines to directly export themselves to the XML input file format used by OpenMC. The OpenCG-OpenMC compatibility module allows OpenCG's primitives to be transformed into the corollaries within the OpenMC Python API, and vice versa. The compatibility modules enable easy exportation of a user's physics-agnostic OpenCG geometry directly to OpenMC (or any other supported code).

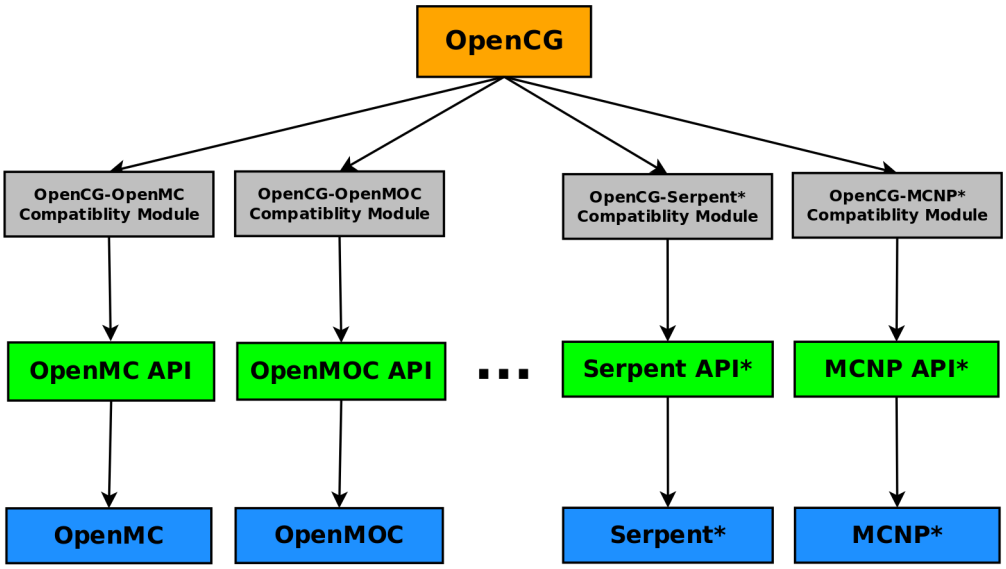

Figure 4. OpenCG compatibility modules for various neutron transport codes. The compatibility modules for OpenMC and OpenMOC will be released in future public distributions of each code, while modules for Serpent and MCNP are in progress at the time of this writing.

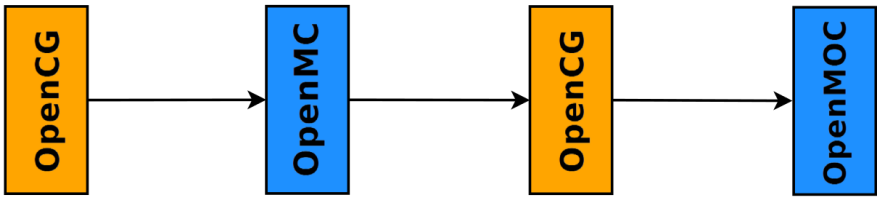

Figure 5. A workflow utilizing OpenCG and multiple compatibility modules. In this example, a geometry is built in OpenCG and exported to OpenMC for a simulation to compute multi-group cross-sections on the irregular CG tally mesh. The OpenMC tally mesh is exported to OpenCG for data processing of the tallies. Finally, the tallied multi-group cross-sections are transferred on the OpenCG mesh to OpenMOC for a deterministic calculation.

6 CONCLUSIONS

A new combinatorial geometry modeling tool called OpenCG has been developed by the Computational Reactor Physics Group at MIT. OpenCG provides an easy-to-use Python API for large scale processing of data from neutron transport codes which utilize CG. Several unique algorithms in OpenCG - including Local Neighbor Symmetry identification and Region Differentiation - were presented in this paper. In addition, compatibility modules for various codes, including OpenMC and OpenMOC, are implemented to enable rapid cross-code verification of a single geometric model in OpenCG. It is the hope of those involved with OpenCG that the nuclear engineering community will utilize the open source codebase to complement the rich and growing set of Python-based tools for nuclear engineering - including PyNE [16] - for their own research interests.

7 ACKNOWLEDGMENTS

Special thanks to Joshua Richard, Nicholas Horelik, Samual Shaner and Bryan Herman for their advice and feedback during the initial development of OpenCG. Davis Tran, Logan Abel and Qicang Shen were each instrumental in developing various components of OpenCG. The first author was supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374. This research was supported in part by the Center for Exascale Simulation of Advanced Reactors (CESAR), a co-design center under the U.S. Department of Energy Contract No. DE-AC02-06CH11357.

8 REFERENCES

[1] X-5 Monte Carlo Team, "MCNP – A General Monte Carlo N-Particle Transport Code, Version 5," Technical Report LA-UR-03-1987, Los Alamos National Laboratory (2008).

[2] J. Leppanen, "Serpent - A Continuous Energy Monte Carlo Reactor Physics Burnup Calculation Code," 2013, http://montecarlo.vtt.fi/download/Serpent_manual.pdf.

[3] P. K. Romano and B. Forget, "The OpenMC Monte Carlo Particle Transport Code," *Annals of Nuclear Energy*, **51**, pp. 274–281 (2013).

[4] W. Boyd, S. Shaner, L. Li, B. Forget and K. Smith, "The OpenMOC Method of Characteristics Neutral Particle Transport Code," *Annals of Nuclear Energy*, **68**, pp. 43–52 (2014).

[5] K. Smith, "Reactor Core Methods," *M&C 2003*, Gatlinburg, Tennessee, April 6–10, 2003, Plenary Presentation.

[6] "OpenMOC User Documentation: Constructive Solid Geometry," 2014, https://mit-crpg.github.io/OpenMOC/methods/constructive_solid_geometry.html.

[7] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science & Engineering*, **13**, pp. 22–30 (2011).

[8] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing In Science & Engineering*, **9**, *3*, pp. 90–95 (2007).

[9] F. Pérez and B. E. Granger, "IPython: A System for Interactive Scientific Computing," *Computing in Science & Engineering*, **9**, *3*, pp. 21–29 (2007).

[10] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer (1987).

[11] J. Pach and P. K. Agarwal, *Combinatorial Geometry*, John Wiley & Sons (2011).

[12] D. L. Millman, D. P. Greisheimer, B. R. Nease, and J. Snoeyink, "Robust Volume Calculations for Constructive Solid Geometry (CSG) Components in Monte Carlo Transport Calculations," *PHYSOR*, Knoxville, TN, USA, 2012.

[13] D. Lax, W. Boyd, and N. Horelik, "An Algorithm for Identifying Unique Regions in Constructive Solid Geometries," *PHYSOR*, Kyoto, Japan, 2014.

[14] N. Horelik, B. Herman, B. Forget, and K. Smith, "Benchmark for Evaluation and Validation of Reactor Simulations (BEAVRS), v0.1.1," *Intl'l Conf. Math. and Comp. Methods Appl. to Nucl. Sci. & Eng.*, Sun Valley, ID, USA, 2013.

[15] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)," 2001, RFC 3174.

[16] A. M. Scopatz, P. K. Romano, P. H. Wilson, and K. D. Huff, "PyNE: Python for Nuclear Engineering," *Trans of the Amer. Nucl. Soc.*, **107**, pp. 985 (2012).