Disclaimer

MOCUM code package, examples, tools and other files are for illustrative purposes only and are provided AS IS with absolutely no warranty. This program has not been thoroughly tested under all conditions. The author, therefore, cannot guarantee or imply reliability, serviceability, or functionality of these programs. There is no cost for non-commercial usage of the code. If you intend to use it for commercial purpose, please contact the author.
# Table of Contents

1. Introduction ........................................................................................................................................... 1
2. Installation Guide ................................................................................................................................... 2
   2.1 Windows ........................................................................................................................................... 2
   2.2 Linux ............................................................................................................................................... 3
3. MOCUM-UM: Unstructured Meshing ................................................................................................. 4
   3.1 Geometry ......................................................................................................................................... 4
      3.1.1 Primitives ............................................................................................................................... 4
      3.1.2 Media ...................................................................................................................................... 6
      3.1.3 Boundary ............................................................................................................................... 7
      3.1.4 Array ...................................................................................................................................... 7
   3.2 Triangulation Options ..................................................................................................................... 10
      3.2.1 Triangulation Criteria .......................................................................................................... 10
      3.2.2 Global Circle Sides .............................................................................................................. 10
      3.2.3 Threads for Parallel Computing ............................................................................................ 10
   3.3 Mesh File ........................................................................................................................................ 10
4. MOCUM-MOC: MOC Solver .............................................................................................................. 12
   4.1 Macroscopic Cross Section .............................................................................................................. 12
   4.2 MOC Parameters ............................................................................................................................ 13
      4.2.1 Azimuthal Angles ............................................................................................................... 13
      4.2.2 Polar Angles ......................................................................................................................... 13
      4.2.3 Ray Density ........................................................................................................................... 13
      4.2.4 Convergence Criterion ......................................................................................................... 14
      4.2.5 Boundary Condition ............................................................................................................. 14
      4.2.6 Plots of Flux Distribution ..................................................................................................... 14
      4.2.7 Merge Flux and Reaction Rates ............................................................................................ 15
      4.2.8 Threads .................................................................................................................................. 16
   4.3 Output .............................................................................................................................................. 17
5. Visualization ........................................................................................................................................... 18
   5.1 Mesh Visualization .......................................................................................................................... 18
   5.2 Flux Distribution Visualization ...................................................................................................... 18
6. Examples

6.1 BWR Lattice with Adjacent Gadolinium Burnable Poison Pins

6.2 CANDU6 Fuel Bundle

6.3 MZA Fast Reactor

6.4 KNK-II Core

6.5 HTTR Core

7. Reference
1. Introduction

1.1 MOCUM features

MOCUM \[^{[1,2]}\] stands for Method Of Characteristics Unstructured Meshing.

MOCUM code package solves the steady state Boltzmann transport equation in two dimensional (2-D) arbitrary geometries. The geometry processor allows users to use constructive solid geometry (CSG) to build very complex geometry, which will be further decomposed by conforming Delaunay triangulation (CDT). The created CDT is stored in a plain text file and can be used for other purposes. The fundamental methodology of the neutron flux solver is method of characteristics (MOC). The flux sweeping kernel is fully parallelized by OpenMP 2-level nested loops. MOCUM could calculate multiplication factor \((k_{\text{eff}})\), flux and reaction rate distribution. It also generates figures of meshes and neutron flux profiles. MOCUM has been extensively benchmarked against various neutronics benchmarks representing different types of reactors. Performance shows that MOCUM is highly accurate and efficient in modeling 2-D fission reactors with complex geometries.

1.2 Reading the Manual

The aim of this manual is to describe how to use the code and input deck format. Readers should have basic knowledge of reactor physics and computer/server. The detailed physics and mathematics derivation can be found in Ref. \(^{[1]}\).

This manual is organized in the following way. Chapter 2 presents the installation procedure on Windows and Linux. Chapter 3 describes geometry construction and triangulation. Chapter 4 explains the parameters used in flux sweeping and contents of output. Chapter 5 shows the functions of the MATLAB scripts. The last chapter uses several examples to demonstrate the functionalities of the code.

The entries in the input deck usually contain keyword and value. In this manual, they are presented in the following format:

```
keyword value1 value2 ... # comments
```

Comments follow the symbol of ‘#’. Comments must be placed after command in the same line. Comments should not occupy a line without command.

Card names follow the symbol of ‘@’, such as @geo.

The unit is cm.
2. Installation Guide

MOCUM code package consists of three parts: MOCUM-UM, MOCUM-MOC and MATLAB scripts. MATLAB scripts require MATLAB and they are used for visualizing the meshes and flux distribution. Therefore, MATLAB is optional for running MOCUM, but is highly recommended.

MOCUM-UM is written in C++, and is used for building geometry and performing triangulation. MOCUM-UM depends on CGAL \[^3\] and tree.hh \[^4\]. MOCUM-MOC is written in FORTRAN 2003, and is used for solving the transport equation. They are connected by a mesh file produced by MOCUM-UM. They are cross-platform programs and can run on 32/64-bits Windows XP/Vista/7 and Linux system. MOCUM has been tested with a few compilers and operation systems, listed in Table 2.1. MOCUM is assumed to be able to run on 64bits Windows and 32bits Linux, with other versions of libraries, but they are not tested.

<table>
<thead>
<tr>
<th></th>
<th>Windows XP/Vista/7 (32bits)</th>
<th>Linux (64bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOCUM-UM</td>
<td>Visual Studio 2010 Boost v1.47</td>
<td>GCC v4.4.6, Intel C++ ComposerXE v12.1.5</td>
</tr>
<tr>
<td></td>
<td>CGAL v3.7 or v3.9</td>
<td>CGAL v3.9.1</td>
</tr>
<tr>
<td></td>
<td>Tree.hh v2.65</td>
<td>Tree.hh v2.65</td>
</tr>
<tr>
<td>MOCUM-MOC</td>
<td>Intel Visual Fortran Compiler v12.1.2</td>
<td>Intel Fortran Composer v12.1.5</td>
</tr>
</tbody>
</table>

32bits Windows executable of MOCUM-UM and MOCUM-MOC is provided. Casual users may start from there to avoid installing CGAL which can be problematic for beginners. The MOCUM-UM executable has no known limitation. An ordinary laptop with 3 GB memory can decompose all MOCUM example geometries. The MOCUM-MOC sometimes require large amount of memory. Memory of 3 GB is sufficient to run lattice size problems, or homogenous core with coarse parameters. Heterogeneous full core modeling may require tens of GB of memory. In that case, users must compile MOCUM-MOC on a 64bits server with sufficient memory installed.

2.1 Windows

2.1.1 Running 32bits executable

Unzip the executable into a folder, for example: c:\mocum. You will see two exe files: mesh.exe and moc.exe. They are executable of MOCUM-UM and MOCUM-MOC, respectively. There are two dll files as well: libgmp-10.dll and libiomp5md.dll. They are libraries for mesh.exe and moc.exe, respectively.

If you do not have Microsoft Visual Studio 2010 installed, download and install Microsoft Visual C++ 2010 Redistributable Package (x86).

Click START button. Type cmd and click cmd.exe. In the cmd windows, type the following command:

```
cd c:\mocum\examples\bwr
..\..\mesh.exe bwr.txt
..\..\moc.exe bwr.txt
```

The above commands run the bwr example. It assumes your MOCUM executable is located in c:\mocum. The first argument after the executable name is the input deck filename.
2.1.2 Compilation

**MOCUM-UM**
- Install Microsoft Visual Studio 2010 or Microsoft Visual C++.
- Read CGAL installation guide carefully. Install BOOST library. Download and build CGAL library.
- Download and extract MOCUM Linux compressed package.
- Start a new VS2010 C++ project and add ReactorMeshing.cpp as the existing source file.
- Setup proper CGAL and BOOST header files and library files path.
- Turn on maximum optimization /Ox and OpenMP parallelization /openmp.

**MOCUM-MOC**
- Install Microsoft Visual Studio 2010 or Microsoft Visual C++.
- Install Intel Visual FORTRAN Composer XE.
- Start a new Intel Visual Fortran empty project and add all .f90 files as the existing source files.
- Turn on maximum optimization /O3, multi-file interprocedural optimization /Qipo and OpenMP parallelization /Qparallel.

2.2 Linux

There is no Linux executable, so compilation is always required.

**MOCUM-UM**
- If you have root access, you can simply install CGAL developer package, for example, CGAL-devel-3.9-1.el6.x86_64.rpm. Otherwise, you have to follow the CGAL installation guide to build it from scratch.
- Go to the MOCUM-UM source folder src_meshing.
- If you have gcc installed, type
  
  make

- If you want to use Intel C compiler, type
  
  make -f Makefile.icc

  If icpc is not in the system path, you need to edit the Makefile.icc to specify its location. If CGAL is not installed as root, you need to edit the Makefile or Makefile.icc to specify the locations of the CGAL header files and library files.

  If you are unable to compile MOCUM-UM on Linux, you can always use the Windows executable, which works for very large and very complex geometry, and its runtime is just a few seconds to a few minutes. In this case, you need to manually copy the mesh file and the input file to the folder where you run the MOCUM-MOC.

**MOCUM-MOC**
- You must have Intel FORTRAN Compiler installed. It does not work with gfortran.
- Go to the MOCUM-UM source folder src_moc, and type
  
  make

**Testing**
- Go to MOCUM examples/bwr folder, and type
  
  ../../mesh bwr.txt
  ../../moc bwr.txt
3. MOCUM-UM: Unstructured Meshing

3.1 Geometry

The geometry card starts with `@geo-def`, and ends with `@end`. An example geometry card is illustrated below.

```
@geo-def
unit 1
  box   10 0 1.6 0 1.6
  circle 20 0.8 0.8 0.6
  circle 30 0.8 0.8 0.5
  media 3 10 -20
  media 2 20 -30
  media 1 30
  boundary 10
unit 2
  box   10 0 1.6 0 1.6
  circle 20 0.8 0.8 0.6
  circle 30 0.8 0.8 0.5
  media 3 10 -20
  media 2 20 -30
  media 4 30
  boundary 10
unit 0
  box 10 0 6.4 0 6.4
  array 111 box 4 4 lowerleft 0.0 0.0
  1 1 1 1
  1 2 1 1
  1 1 2 1
  1 1 1 1
  media 0 111
  boundary 10
@end
```

The geometry card consists of several geometry units:

```
unit uid
```

`uid`: Unit index (user defined positive integer or 0).

Unit 0 is the global unit, and it must present in every input deck. Each geometry unit is composed of several elemental shapes called primitives. The following primitives are defined in MOCUM code.

3.1.1 Primitives

3.1.1.1 Box
box $p_{idx}$ $xmin$ $xlength$ $ymin$ $ylength$

$p_{idx}$: primitive index;
$xmin, ymin$: lower left vertex coordinate;
$xlength, ylength$: side lengths along x and y axes.

The primitive index is a user defined positive integer. Primitive index must be unique for each primitive inside a unit. Same primitive index can appear in different units.

3.1.1.2 Circle

circle $p_{idx}$ $x0$ $y0$ $r0$ $nsides$

$p_{idx}$: Primitive index;
$x0, y0$: Circle center coordinate;
$r0$: Radius;
$sides$: Optional, number of sides used for approximating the circle.

The circle is approximated by an n-sides regular polygon with same area. The optional value of $nsides$ controls the number of sides of the polygon. The polygon side length should be chosen to equal the average side length of the triangulation to produce same size of triangles, but it can be tuned for special purpose. There is always a polygon vertex at $(x0, y0+r0)$. If this option is omitted, the number of polygon sides will use the circle-side command defined in the @geo-opt card.

3.1.1.3 Horizontal Hexagon

hex $p_{idx}$ $x0$ $y0$ $r0$

$p_{idx}$: Primitive index;
$x0, y0$: Hexagon center coordinate;
$r0$: Hexagon side length or distance between hexagon center and a vertex.

3.1.1.4 Vertical Hexagon

vhex $p_{idx}$ $x0$ $y0$ $r0$

$p_{idx}$: Primitive index;
$x0, y0$: Hexagon center coordinate;
$r0$: Hexagon side length or distance between hexagon center and a vertex.
3.1.1.5 1/6 hexagon type 1

\texttt{hex60-1} \ p\_idx \ x0 \ y0 \ r0

\textit{p\_idx}: Primitive index; \\
x0, y0: Hexagon center coordinate; \\
r0: Hexagon side length or distance between hexagon center and a vertex.

![Figure 3.5. 1/6 hexagon type 1.](image)

3.1.1.6 1/6 hexagon type 2

\texttt{hex60-2} \ p\_idx \ x0 \ y0 \ r0

\textit{p\_idx}: Primitive index; \\
x0, y0: Hexagon center coordinate; \\
r0: Hexagon side length or distance between hexagon center and vertex.

![Figure 3.6. 1/6 hexagon type 2.](image)

3.1.1.7 1/12 hexagon

\texttt{hex30} \ p\_idx \ x0 \ y0 \ r0

\textit{p\_idx}: Primitive index; \\
x0, y0: Hexagon center coordinate; \\
r0: Hexagon side length or distance between hexagon center and a vertex.

![Figure 3.7. 1/12 hexagon.](image)

3.1.2 Media

The \texttt{media} command uses Boolean operation(s) to define a region from primitives. The \texttt{media} command also defines the material index for this region. The \texttt{media} command has the following format:

\texttt{media} \ mid \ p\_idx1 \ p\_idx2 \ p\_idx3 \ ...

\textit{mid}: Material id;
\textit{p\_idx}: Primitive indices defined in the same unit.

The primitive indices form a formula of a serial of Boolean operations. Positive primitive index means union, while negative primitive index means minus. The first primitive index must be positive. Bracket is not supported. The result after each Boolean operation step must be a polygon or a polygon with hole(s) in its interior. It cannot be separated into several parts.
The following example defines a square unit cell. Primitive indices 10, 20 and 30 denote the outer rectangular box, clad outer circle and fuel rod outer circle, respectively. Material 1, 2 and 3 represent fuel, clad and moderator, respectively, whose cross section will be defined in the @xs card.

```
unit 1
  box 10 0 1.6 0 1.6
  circle 20 0.8 0.8 0.6
  circle 30 0.8 0.8 0.5
media 3 10 -20
media 2 20 -30
media 1 30
boundary 10
```

Figure 3.8. A unit cell example.

You can define a few primitives which will not be used in the media command. Users are responsible for filling all regions in a unit. In above example, if the last media command is omitted, the fuel rod region will become undefined, and the code will probably crash or produce wrong results.

### 3.1.3 Boundary

The **boundary** command tells the code the primitive index of the outer boundary of a unit. The boundary must be a single primitive, and cannot be a result of Boolean operation. If a unit is not a global unit (uid > 0), the boundary primitive can be box, hex or vhex. If a unit is a global unit (uid = 0), the boundary can be box, hex, hex60-1, hex60-2 and hex30. The **boundary** command has following format, and it must present in every unit definition.

```
boundary  bp_idx
```

*bp_idx*: Boundary primitive index.

The regular unit and global unit have independent coordinate systems, and the primitive listed in the boundary command should be defined in the correct position. For box type unit, the lower left corner should be placed at the origin; for hex or vhex type unit, the center should be placed at the origin. The boundary primitive of the global unit can be placed anywhere.

### 3.1.4 Array

The **array** command is the only way to link different units. Units are mapped into another unit through **array** command. An array can be considered as a special primitive. A unit can have multiple arrays. MOCUM support rectangular, vertical hexagonal and horizontal hexagonal arrays.

### 3.1.4.1 Rectangular array

```
array  a_idx box I J lowerleft x0 y0
```
uid(1,1)  uid(2,1)  uid(3,1) ... uid(I,1)
uid(1,2)  uid(2,2)  uid(3,2) ... uid(I,2)
...
uid(1,J)  uid(2,J)  uid(3,J) ... uid(I,J)

*a_idx:* Array index (similar to the primitive index, any positive integer);
*box:* Array type, rectangular array;
*I, J:* Number of entries along X and Y;
*lowerleft:* Reference point; keep it unchanged;
*x0, y0:* Array lower left point coordinate.
*uid:* Unit indices. Array dimension: I×J.

The lower left unit in a rectangular array has lowest *x* and *y* coordinates among all units in the array. The rectangular array can have rectangular units (not a square) as its entries. It is the users’ responsibility to make sure the dimensions of all units along a row or a column are identical. Different rows or columns can have different dimension. Array’s outermost entries can be empty, and they are denoted by 0. Interior array entries cannot be empty (a hole inside an array is not allowed). A rectangular array example is shown below.

```plaintext
array 135 box 4 3 lowerleft 10.0 20.10
1 2 3 2
0 4 5 4
1 2 0 2
```

![Figure 3.9. A rectangular array example.](image)

The defined array must be referenced by the `media` command. Array can also participate Boolean operations. If 135 denotes a defined array, the following 3 lines are valid. If the Boolean operation result is an array, or part of an array, the material id must be 0.

```plaintext
media 0 135
media 0 135 -10 -20
media 3 30 -135
```

The rectangular array can be nested. For example, unit 1 is mapped into unit 2 via array 11, and unit 2 is then mapped into unit 3 via array 22.

### 3.1.4.2 Vertical Hexagonal Array

```plaintext
array a_idx vhex I J center x0 y0
uid(1,1), uid(2,1), ..., uid((I+1)/2,1)
uid(1,1), uid(2,1), ..., uid((I+1)/2+1,1)
...
uid(1,(J+1)/2), uid(2,(J+1)/2), ..., uid(I,(J+1)/2)
...
uid(1,J), uid(2,J), ..., uid((I+1)/2,J)
```

*a_idx:* Array index (similar to the primitive index, any positive integer);
**vhex:** Array type, vertical hexagonal array;

*I, J:* Number of columns in the center row, number of rows. *I* must equal to *J, I* and *J* must be odd integers;

**center:** Reference point; keep it unchanged;

*x0, y0:* Array center point coordinate.

**uid:** Unit indices.

The units in the vertical hexagonal array must be equal size vertical hexagons (vhex). A vertical hexagonal array example is shown below.

```
array 135 vhex 5 5 center 3.5 6.0
   0  2  0
   4  5  6  7
   8  9 10 11 12
  13 14  0  0
   0 18 19
```

![Figure 3.10. A vertical hexagonal array example.](image)

### 3.1.4.3 Horizontal Hexagonal Array

```
array a_idx hex I J center x0 y0
uid(1,1), uid(2,1), ..., uid((I+1)/2,1)
uid(1,1), uid(2,1), ..., uid((I+1)/2+1,1)
...
uid(1,(J+1)/2), uid(2,(J+1)/2), ..., uid(I,(J+1)/2)
...
uid(1,J), uid(2,J), ..., uid((I+1)/2,J)
```

**a_idx:** Array index (similar to the primitive index, any positive integer);

**hex:** Array type, horizontal hexagonal array;

*I, J:* Number of columns in the center row, number of rows. *I* must equal to *J, I* and *J* must be odd integers;

**center:** Reference point; keep it unchanged;

*x0, y0:* Array center point coordinate.

**uid:** Unit indices.

The units in the horizontal hexagonal array must be equal size horizontal hexagons (hex). The first row starts from the lower left corner and goes up. A horizontal hexagonal array example is shown below.

```
array 135 hex 5 5 center 3.5 6.0
   0  2  0
   4  5  6  7
   8  9 10 11 12
  13 14  0  0
   0 18 19
```

![Figure 3.10. A horizontal hexagonal array example.](image)
3.2 Triangulation Options

The triangulation option card starts from @geo-opt and ends with @end.

3.2.1 Triangulation Criteria

criteria 0.125 /

0.125: Shape criterion, do not change;
/: All triangles’ side lengths are shorter than l. This value controls the mesh size.

3.2.2 Global Circle Sides

circle-side nsides

nsides: Number of regular polygon sides used for circle approximation. The circle area is preserved. This value can be override by the nsides value defined in the circle command.

3.2.3 Threads for Parallel Computing

thread nthread

nthread: Number of threads used for parallel computing. If this command is absent from the input deck, the code will use maximum number of threads on the computer. If nthread > maximum number of threads on a system, it will be reset to the maximum number of threads. If nthread = -1, nthread will be set to maximum number of threads -1.

Most of the MOCUM-UM is serial, because CGAL library is not thread safe. Therefore, increasing the number of threads has little effect on reducing the runtime.

3.3 Mesh File

The triangulation process will produce a temporary file, Core_Diagnostics. This file is produced by CGAL library, and can be deleted after the execution of MOCUM-UM.

MOCUM-UM will also produce a mesh file containing the vertex coordinates and neighbor information. MOCUM-UM attaches an extension name of “.mesh” to the input file name as the mesh file name. For example, if your input file name is “MyInput.txt”, the mesh file name will be “MyInput.txt.mesh”. If the mesh file already exists, the old mesh file will be overwritten.

If you do not intend to use this mesh file for other purposes, you can jump to the next chapter.

The mesh file is a plain text file, and has the following format:
Points: npts
1 x1 y1
2 x2 y2
Meshes: \textit{nmeshes}
1 \textit{mid1} \textit{pid1}_1 \textit{pid1}_2 \textit{pid1}_3
2 \textit{mid2} \textit{pid2}_1 \textit{pid2}_2 \textit{pid2}_3
\ldots

Neighbors and area: \textit{nmeshes}
1 \textit{nb1}_1 \textit{nb1}_2 \textit{nb1}_3 \textit{area1}
2 \textit{nb2}_1 \textit{nb2}_2 \textit{nb2}_3 \textit{area2}
\ldots

Boundary: \textit{type} \textit{value1} \textit{value2} \textit{value3} \ldots \textit{l}

\textit{npts}: Number of unique points (integer);
\textit{x}, \textit{y}: Point coordinates (float);
\textit{nmeshes}: Number of meshes (integer);
\textit{mid}: Material id in each mesh (integer);
\textit{pid}: Point index of three vertices (integer);
\textit{nb}: Mesh index of three neighbors (integer). Zero means an edge is a core boundary;
\textit{area}: Mesh area (float);
\textit{type}: Core boundary shape defined in the boundary command in unit 0 (text string). It can be \textit{box}, \textit{hex}, \textit{hex60}-1, \textit{hex60}-2, \textit{hex30};
\textit{value}: any values used in primitive definition command. For example, \textit{xmin} \textit{xlength} \textit{ymin} \textit{ylength} in \textit{box} command, or \textit{x0} \textit{y0} \textit{r0} in \textit{hex}, \textit{hex60}-1, \textit{hex60}-2, and \textit{hex30} commands.
\textit{l}: All triangles’ side lengths are shorter than \textit{l}.

If the core boundary is hexagon type, there will be very slim triangles (three vertices of a triangle almost fall on a straight line) along the diagonal boundaries caused by the imperfection of the square root routines in CGAL. Those slim meshes can be easily found by spotting the small area, and they are properly treated by MOCUM-MOC. If you intend to use the mesh file for other purposes, you need to keep it in mind.
4. MOCUM-MOC: MOC Solver

The MOCUM-MOC code reads two cards: the macroscopic cross section card @xs and MOC parameter card @solver.

4.1 Macroscopic Cross Section

The cross section card starts from @xs and end with @end. It has following format.

```
@xs
type macro
group G
nmat nm
mid 1
  Σ''_1  Σ''_2  ...  Σ''_G
  Σ'_1  Σ'_2  ...  Σ'_G
  νΣ'_1  νΣ'_2  ...  νΣ'_G
  χ_1  χ_2  ...  χ_G
  Σ"_1  Σ"_2  ...  Σ"_G
  Σ^a_1  Σ^a_2  ...  Σ^a_G
  Σ^s_1→i  Σ^s_1→2  ...  Σ^s_1→G
  Σ^s_2→i  Σ^s_2→2  ...  Σ^s_2→G
  ...
  Σ^s_G→1  Σ^s_G→2  ...  Σ^s_G→G
mid 2
...
@end
```

type macro: Keep them unchanged;
G: Number of energy groups;
nm: Number of materials;
Σ'': Transport corrected total macroscopic cross section;
Σ': Fission macroscopic cross section;
ν: Average number of neutrons emitted per fission reaction;
χ: Fission spectrum;
Σ": Absorption cross section;
Σ^s: Scattering cross section.

The flux solver does not use fission and absorption cross section. They are only used for reaction rate calculation. Therefore, if any of them is not available, you can simply put all 0.0 there.
In some cases, if void/vacuum regions are used in the geometry, user can define a special material, whose total and in-group scattering cross sections are all 1E-16 (any very small number), and all other cross sections are 0.0. The small number is used for preventing “divided by zero” error.

### 4.2 MOC Parameters

The MOC parameter card starts from @solver and ends with @end. This is the last card in the input deck.

#### 4.2.1 Azimuthal Angles

naz naz

naz: Number of azimuthal angles.

Azimuthal angles are uniformly distributed on the $X$-$Y$ plane. The first azimuthal angle is $360/naz/2$ degrees. If all core boundary conditions are vacuum/free, any azimuthal angle number can be used. If reflective boundary condition is used on any core boundary that is parallel to $X$ or $Y$ axes, the number of azimuthal angles must be a multiple of 4. If reflective boundary condition is used on any core boundary whose slope is $±\sqrt{3}$ (four diagonal edges of a horizontal hexagon, left and right edge of hex60-1), the number of azimuthal angles must be a multiple of 6. If reflective boundary condition is used on any core boundary whose slope is $±\sqrt{3}/3$ (upper left edge of hex30, upper left and lower left edge of hex60-2), the number of azimuthal angles must be a multiple of 12. Therefore, a multiple of 12, such as 12, 24, and 36 ..., is always safe with any core shape. Usually, you should use at least 8 azimuthal angles for accurate results.

#### 4.2.2 Polar Angles

npl pl_type npl


npl: Number of polar angles.

The maximum number of polar angles for Leonard type is 3, and the maximum number of polar angles for Gauss-Legendre type is 6. Usually, three Leonard type polar angles can produce the most accurate results.

#### 4.2.3 Ray Density

ray-density r_rho

r_rho: Ray density (unit: cm$^{-1}$).

Ray density means the number of characteristics rays within 1 cm. The inverse of the ray density is ray separation (unit: cm), distance between two adjacent rays. Usually, a zone should be covered by more than 10 rays for accurate results.
4.2.4 Convergence Criterion

\[ k_{\text{tol}} \]

\[ k_{\text{eff_tol}} \]

\[ k_{\text{eff_tol}} \] : \( k_{\text{eff}} \) convergence criterion.

4.2.5 Boundary Condition

\[ \text{bc bc1 bc2 bc3 bc4 bc5 bc6} \]

\( \text{bc} \) : Boundary conditions on core edges. Reflective: 1; vacuum: 0.

You must put 6 values after the \( \text{bc} \) command, regardless how many edges the core has. If the number of core edges is smaller than 6, only the first a few values are used.

Numbering sequence:
- \( \text{box} \): bottom \( \rightarrow \) counterclockwise;
- \( \text{hex} \): bottom \( \rightarrow \) counterclockwise;
- \( \text{hex60-1} \): bottom \( \rightarrow \) counterclockwise;
- \( \text{hex60-2} \): lower left \( \rightarrow \) counterclockwise;
- \( \text{hex30} \): bottom \( \rightarrow \) counterclockwise.

4.2.6 Plots of Flux Distribution

If the \text{plot} command is present, the code will generate one or more files containing the plot information. The plot filenames have extension name of “.plot.01”, “.plot.02” ... attached to the input filename.

\text{plot npplot}
\text{plot_type sizex sizey xmin xmax ymin ymax}

\( \text{npplot} \): Number of plots;
\( \text{plot_type} \) : all: draw entire core; local: draw a portion of the core;
\( \text{sizex, sizey} \) : Resolution along \( X \) and \( Y \) axes;
\( \text{xmin, xmax, ymin, ymax} \) : These values are required for local plot type only. Coordinates of the portion of the core user intends to draw.

The following example defined 2 plots. The first plot asks the code to produce a plot file with a resolution of 500 \times 400. The plot region is from (10.0, 100.0) to (60.0, 140.0). The second plot is for the entire core, and the resolution is 800 \times 800.

\text{plot 2}
\text{local 500 400 10.0 60.0 100. 140.0}
\text{all 800 800}

The plot file is a plain text files, and has the following format:
sizex sizex xmin xmax ymin ystep ymax ng
Block of group 1 flux (sizex by sizex)
Block of group 2 flux (sizex by sizex)
...

sizex, sizex: Resolution along X and Y axes;
xmin, xmax, ymin, ymax: Coordinates of the portion of the core this file includes.
xstep, ystep: Distance between two adjacent dots along X and Y axes.

4.2.7 Merge Flux and Reaction Rates

User can define a few rectangular or hexagonal arrays to extract the flux and reaction rate information in each array cell. The merge command has the following format.

merge nmerge

nmerge: Number of merge commands.

The merge definition has three array types: box, vhex and hex. The rectangular array has following format:

box nx ny all
box nx ny local xmin xmax ymin ymax

nx, ny: Number of subdivision along X and Y axes.
xmin xmax ymin ymax: Coordinates of the region of interest.

The region of interests will be subdivided into nx by ny pieces. Each subdivision is a rectangular with equal size. If the all option is used, the entire core is considered as the region of interest. If local option is used, the coordinates of the region of interest are required.

There are two types of hexagonal arrays for the merge command:

vhex ncol r0 x0 y0
hex ncol r0 x0 y0

ncol: Number of rows or number of columns in the center row;
r0: Distance between hexagon center and one vertex;
x0, y0: Coordinates of the center of the center hexagon in the array.

The numbering sequences of three array types are illustrated in Fig. 4.1.
The averaged flux in each cell is the integrated flux which is calculated by the following equation:

$$\Phi_{\text{avg}} = \int_{\text{cell}} \Phi_g V$$

The merge command also calculates averaged flux for each material separately. This function is very useful for extracting the flux in different material inside the same cell (for example, flux in the fuel pin and flux in the moderator inside the same cell). This function is also useful if the fuel pins are not arranged in above three types of arrays. Similar to the above equation, the averaged flux for each material will only integrate over the region where certain material presents.

The absorption and fission reaction rates in each cell are calculated according to the following equation:

$$\overline{R^a} = \sum_{g=1}^{G} \left( \int_{\text{cell}} \Phi_g \Sigma_f^g V \right)$$
$$\overline{R^f} = \sum_{g=1}^{G} \left( \int_{\text{cell}} \Phi_g \Sigma_f^f V \right)$$

Note: some cells or portion of a cell in the array can fall outside the core.

### 4.2.8 Threads

The `threads` command controls the parallel computing. It has the following format.

```
threads nto ntg ntaz
```

- `nto`: Number of threads used for any loops other than flux sweeping;
- `ntg`: Number of threads used for energy group parallelization;
- `ntaz`: Number of threads used for azimuthal angle parallelization.

Value of `nto` controls the number of parallel threads used for auxiliary loops other than the MOC kernel, such as zone or ray loops. The value of `nto` usually equals to the maximum number of threads on the system. The MOC kernel will consume more than 95% runtime of MOCUM-MOC. Therefore, the nested loops of energy group and azimuthal angle are both parallelized. The number of threads used for MOC kernel is `ntg x ntaz`, and the maximum number of threads can be used is the product of the energy group and azimuthal angles.
Example for choosing the parallel scheme: There are 64 cores in the server. The calculation has 6 energy groups and 12 azimuthal angles. To maximize the computation speed, the threads command can be: 64 3 12. If possible, try to parallelize only one loop instead of both energy group and azimuthal angle loops. For above case, if the server has 12 cores, “12 1 12” will be faster than “12 3 4”, because threads management of nested loop takes extra time.

4.3 Output

The output file has an extension name of “.out” attached to the end of the input filename. The output mainly contains 3 parts. All of them are self-explanatory. The first part is the geometry processing statistics. The second part is the $k_{\text{eff}}$ convergence process. The last part is the averaged flux and reaction rates, if the merge command presents in the input file.

On the screen output, at the end of the execution, it should print “Normal End”. Otherwise, the execution is interrupted.
5. Visualization

5.1 Mesh Visualization

The MATLAB script “plotmesh.m” is used for visualizing the unstructured meshes. Place this file in the same folder in where the mesh file is. Edit the mesh filename in the script. Run the script by press F5 in MATLAB. By editing the script, user can change the color of different materials, and mesh boundary color.

5.2 Flux Distribution Visualization

The MATLAB script “plotflux.m” is used for visualizing the flux distribution. Place this file in the same folder in where the plot file is. Edit plot filename in the script. Run the script by press F5 in MATLAB.
6. Examples

The code package includes 6 sets of examples/benchmarks. All of them were used for MOCUM validation and are well documented in Ref. [1, 2]. They represent various core geometries that demonstrate the MOCUM capabilities. The summary of all examples is listed in Table 6.1. The user may need to edit the `threads` command to fit user’s computer system.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reactor types</th>
<th>Geometry type</th>
<th>Scale</th>
<th>Cases</th>
<th>Memory requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>BWR</td>
<td>BWR</td>
<td>Rectangular</td>
<td>Lattice</td>
<td>1</td>
<td>&lt; 1 GB</td>
</tr>
<tr>
<td>CANDU6</td>
<td>PHWR</td>
<td>Irregular</td>
<td>Fuel bundle</td>
<td>1</td>
<td>&lt; 1 GB</td>
</tr>
<tr>
<td>MZA</td>
<td>Fast reactor</td>
<td>Plate-type</td>
<td>Core</td>
<td>8</td>
<td>2 GB / 17 GB</td>
</tr>
<tr>
<td>KNK-II</td>
<td>Fast reactor</td>
<td>Hexagonal</td>
<td>Core</td>
<td>2</td>
<td>2 GB</td>
</tr>
<tr>
<td>HTTR assembly</td>
<td>VHTR (Gen-IV)</td>
<td>Hexagonal</td>
<td>Assembly</td>
<td>7</td>
<td>2 GB</td>
</tr>
<tr>
<td>HTTR core</td>
<td>VHTR (Gen-IV)</td>
<td>Hexagonal</td>
<td>Core</td>
<td>3</td>
<td>17 GB</td>
</tr>
</tbody>
</table>

6.1 BWR Lattice with Adjacent Gadolinium Burnable Poison Pins.

*This example demonstrates:*
- Primitive/unit definition;
- Boolean operation;
- Rectangular array;
- Rectangular merge.

This example is a 4 by 4 BWR lattice with two Gadolinium pins. The geometry description is depicted in Fig. 6.1. The regular fuel is 3%w UO$_2$, while the poisoned fuel contains 3%w UO$_2$ and 3%w Gd$_2$O$_3$. The fuel is wrapped by Zircaloy-2 clad and moderated by water. The reflective boundary condition is specified on all four sides. There are 2 energy groups.

Figure 6.2 illustrates the unstructured meshes visualized by the supplied MATLAB script. Figure 6.2 is for illustrative purpose only, and it may not represent the best mesh size for this example.
6.2 CANDU6 Fuel Bundle

This example demonstrates:
- Take the advantage of the symmetry;
- Irregular geometry;
- Use different number of \textit{nsides} to approximate circles with different sizes;
- Extract flux in irregular geometry by defining multiple materials of same type.

This test case represents the typical CANDU type cell with a Cartesian moderator region. Natural abundance UO$_2$ fuel rods are cooled and moderated by heavy water. The fuel cell structure is illustrated in Fig. 6.3, but the outside heavy water moderator region (square shape with a side length of 28.575cm) is not shown in the figure. There are four rings of fuel rods in the central D$_2$O coolant tube, and the numbers of the fuel rods in each ring are 1, 6, 12 and 18, respectively. The reflective boundary conditions are specified on all outer boundaries. There are 2 energy groups.

MOCUM only models the upper right $\frac{1}{4}$ portion of the geometry to reduce the computation time. A magnified view of the unstructured meshing of the CANDU-6 type cell is illustrated in Fig. 6.4.

![Figure 6.3. Geometry of the CANDU-6 annular cell.](image)

![Figure 6.4. A magnified view of the unstructured meshing of the CANDU-6 type cell.](image)

6.3 MZA Fast Reactor

This example demonstrates:
- Plate-type reactor calculation.

The reactor core of this problem is the simplified MZA core. The plate type unit fuel cell is composed of four uranium dioxide plates, two plutonium plates, six sodium plates and two carbon and steel plates. These material plates are piled along the axial direction. In addition, a voided-fuel lattice is also used to simulate a sodium-voided situation. Fig. 6.5 (a) shows the specifications of the unit fuel lattice and the unit voided-fuel lattice. Blanket and reflector lattices are treated homogeneously in this problem. Four core configurations were created. Case 1 is a reference configuration while the other three are sodium-voided configurations. The core specifications of the four cases are shown in Fig. 6.5 (b). There are two versions for each case: the
heterogeneous version models the exact representation of the plates in the fuel regions, while the homogeneous version considers the fuel as one homogenized region. The strong flux gradient in the heterogeneous fuel plate region presents difficulty for deterministic methods and requires fine meshes. There are 4 energy groups.

Figure 6.5. 2-D MZA fast reactor configuration: (a) unit lattice specification; (b) core specification (4 cases).

6.4 KNK-II Core

This example demonstrates:
Vertical hexagonal array;
Vertical hexagonal merge;
1/12 core symmetry;
Use void region to treat the conjugated external boundary.

This fast reactor KNK-II core model is considered of hexagonal subassemblies. As illustrated in Fig. 6.6, the core is composed of eight hexagonal rings with 169 homogeneous assemblies in the core. The transport effect is emphasized due to small size of the core and the local insertion of the control rods.

MOCUM modeled the 1/12 core to take advantage of the geometry symmetry. The vacuum boundary condition is specified on the conjugated outer boundary, while the reflective boundary condition is used on the bottom and top left boundaries. Two configurations were calculated: all-rods-in and all-rods-out, and there are 4 energy groups.
6.5 HTTR Core

This example demonstrates:
1/6 core symmetry (type 2);
Horizontal hexagonal merge.

As shown in Fig. 6.7(c), the core of the HTTR benchmark problems is consisted of prismatic blocks with a uniform size and a pitch of 36 cm. The core periphery is filled with half blocks along the core sides and 1/3 blocks at the corners, resulting in a core side length of 252 cm. As depicted in Fig 6.7(c), there are seven unique types of hexagonal blocks: 4 types of fuel blocks, filled and empty control rod blocks, replaceable and permanent reflector blocks.

The two heterogeneous blocks are illustrated in Fig. 6.7(a, b). Fig. 6.7(a) shows the structure of the fuel blocks. There are 33 identical fuel pins and 3 burnable poison rods in each fuel block with a pitch of 5.15 cm. Control rod blocks are shown in Fig. 6.7(b), and each block contains a single removable control rod at its center. The diameters of fuel pin, BP rods and CR are 4.1, 1.5 and 12.3 cm, respectively. Other regions in fuel and control rod blocks are filled by graphite. All reflector blocks are homogeneous.

Three core configurations were constructed corresponding to various control rod insertion patterns: case 1: all-rods-in; case 2: partially controlled (in Fig. 6.7(c), the rightmost control rod in the center row is withdrawn; all other control rods are inserted); case 3: all-rods-out.
Figure 6.7. (a) Fuel block configuration; (b) Control rod (CR) block configuration; (c) Core configuration (1/6 core): 1-4: fuel blocks with increasing enrichment, 5/6: filled and empty control rod blocks, 7 and 8: replaceable and permanent reflector block.

The flux profile of all six energy groups of the second case, partially controlled is illustrated in Fig. 6.8 by the MATLAB script.

Figure 6.8. Flux distribution of HTTR benchmark partially controlled case.
7. Reference


