An Introduction to OpenMC

Dr. Patrick Shriwise
On behalf of the ONCORE group
August 13th, 2025

Argonne NATIONAL LABORATORY

Today's Webinar

- An introduction to OpenMC
- Geometry
- Materials
- Simulation
- Tallies
- Post-processing
- Wrap-up and questions

Webinar Goals

Today's demonstration will occur in a Jupyter lab instance, but the same applies in any Python environment

- Provide a basic understanding of OpenMC
- Understand different core components
- Understanding of basic simulation execution and result processing

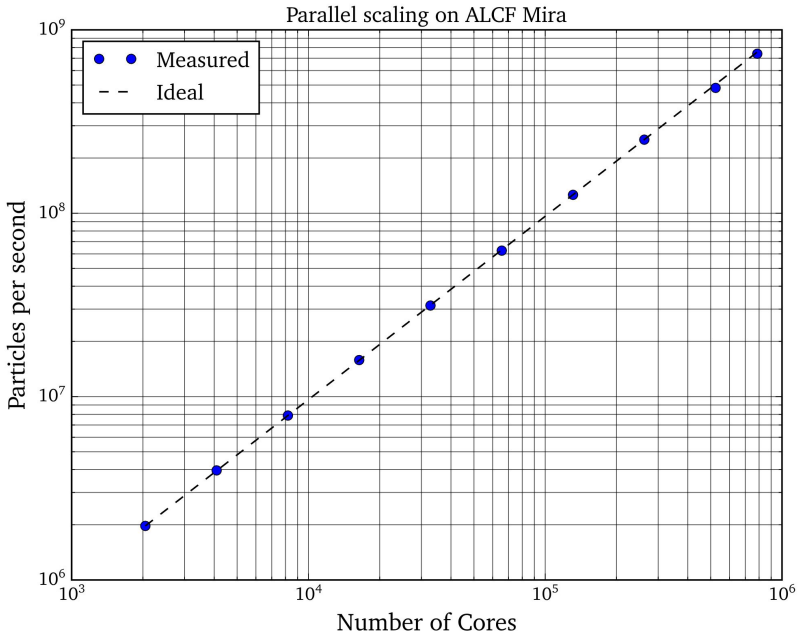
OpenMC

An open source Monte Carlo particle transport code.

- Provides a general solution to the radiation transport equation
- Applies to nuclear reactors, fusion devices, radiation shielding
- Transport executed using C++
- Python interface for model creation and post-processing
- Monte Carlo Advantages:
  - continuous energy treatment of transport
  - no spatial approximations necessary
- Monte Carlo Disadvantages:
  - time to solution/computational cost

What makes OpenMC unique?

- Programming interfaces (C++ & Python)
- Nuclear data interfaces and representation
- Tally abstractions
- Parallel performance
- Development workflow and governance
- Structured text input format (XML)



Navigating the Python API

Naming conventions:

- Module names are `lowercase`

- Functions are `lowercase_with_underscores`

- Classes are `CamelCase`

- Class attributes/variables are `lowercase_with_underscores`

- Top-level (global) variables are `UPPERCASE_WITH_UNDERSCORES`

To give a few specific examples:

- `openmc.deplete` is the depletion *module*

- `openmc.run` is a *function*

- `openmc.MaterialFilter` is a *class*

- `openmc.MaterialFilter.id` is a *class attribute*

- `openmc.data.ATOMIC_NUMBER` is a *top-level variable*

OpenMC Resources

- Code: https://github.com/openmc-dev/openmc
- Docs: https://docs.openmc.org
- Nuclear Data: https://openmc.org
- Forum: https://openmc.discourse.group

Today’s Demonstration

Today we’ll be modeling a simple PWR pincell and performing some basic analysis with user-defined tallies

Software:

- OpenMC (version 0.15.2)
- matplotlib
- pandas
- numpy

Materials

Creating materials

Uses the `openmc.Material` object in the Python module:

```
material = openmc.Material
```

Important methods:

- `Material.set_density`
- `Material.add_nuclide`
- `Material.add_element`
- `Material.add_s_alpha_beta`

Other methods of note:

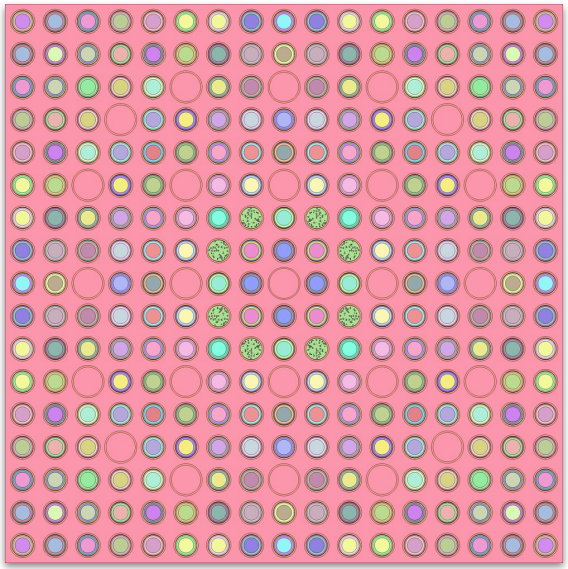
`Material.mix_materials`, `Material.add_elements_from_formula`

Geometry

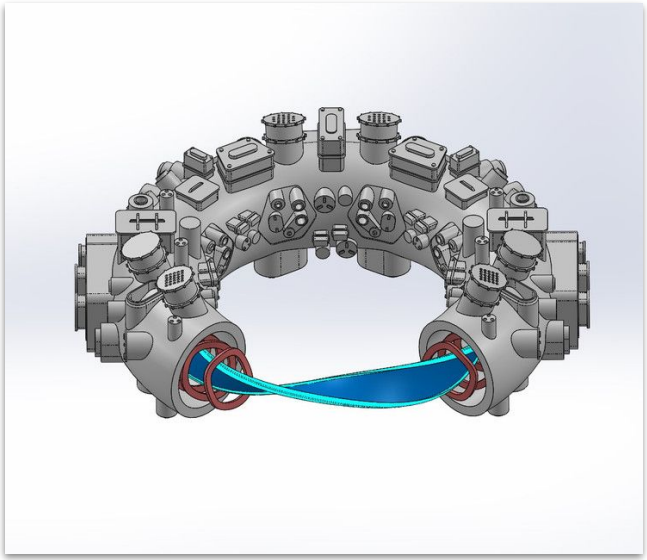
Geometry

OpenMC relies on two types of geometry representations:

Constructive solid geometry (CSG)



CAD-based surface tessellations (DAGMC)



Constructive Solid Geometry (CSG)

A plane perpendicular to the x axis:

$$x - x_0 = 0$$

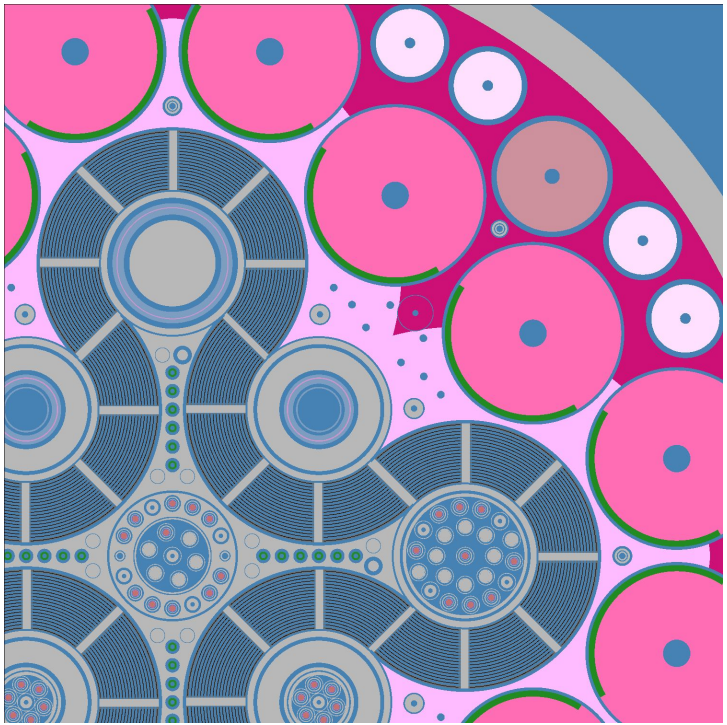
A cylinder parallel to the z axis:

$$(x - x_0)^2 + (y - y_0)^2 - R^2 = 0$$

A sphere:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - R^2 = 0$$

Full surface list



Advanced Test Reactor

Making geometry in OpenMC

Surfaces: definitions of the boundaries between materials

`openmc.Sphere`, `openmc.ZCylinder`, `openmc.XPlane`, `openmc.Plane`, …

Regions: compositions of surface half-spaces

`-` → negative halfspace, `+` → positive halfspace

`&` → intersection; `|` → union; `~` → complement

```
upper_hemisphere = -sphere & +midplane
```

Cells: the connection of a region and material

```
cell = openmc.Cell(fill=uranium_oxide, region=inside_sphere)
```

Universes: collections of cells that can be re-used/repeated

```
universe = openmc.Universe(cells=[cell])
```

Boundary Conditions

The following boundary conditions are supported in OpenMC

- vacuum
- reflective
- periodic (must be set on two surfaces)
- white

Temperatures

Temperatures can be set for either cells or materials (in K).

If a temperature is set on cell and the material being used to fill the cell, the cell temperature will apply.

Cross section data is evaluated a specific temperatures, so temperature treatment is handled in a couple of ways in OpenMC:

- nearest temperature
- temperature interpolation
- windowed multipole

Simulation

Executing OpenMC

There are several ways to execute OpenMC

From the terminal:*

```
$ openmc
```

A Python module-level call*:

```
openmc.run()
```

A call on a Python `openmc.Model` object:

```
model = openmc.Model
...
model.run()
```

* both of these methods expect that XML files are present in the current directory

A note on parallelism

OpenMC utilizes two types of parallelism: distributed memory (MPI) and shared memory (threading)

When used in tandem, the number of CPU cores used is the number of MPI processes *multiplied* by the number of cores. For example, the following terminal command would use 40 cores.

```
$ mpiexec -n 4 openmc -s 10
```

equivalent command in Python

```
>>> model.run(mpi_args=['mpiexec','-n','4'], threads=10)
```

Tallies & Post-Processing

OpenMC Tallies

Any tally in OpenMC can be described with the following form:

$$X = \underbrace{\int d\mathbf{r} \int d\mathbf{\Omega} \int dE}_{\text{filters}} \underbrace{f(\mathbf{r}, \mathbf{\Omega}, E)}_{\text{scores}} \psi(\mathbf{r}, \mathbf{\Omega}, E)$$

where filters set the limits of the integrals and the scoring function is convolved with particle information (e.g. reaction type, current material, etc.).

List of filters List of Scores

Statepoint files

Statepoint files are binary files containing the results of an OpenMC simulation, including any specified tallies.

Object creation:

```
statepoint = openmc.StatePoint('statepoint.10.h5')
...
statepoint.close()
```

Context manager (recommended):

```
with openmc.StatePoint('statepoint.10.h5') as sp:
    simulation_time = sp.runtime['simulation']
```

Wrap-up and Questions