Brief script description:

* **simulate\_tokamak.py**: Uses a 3D spherical geometry of embedded layers with different materials and lengths that may be controlled to run an OpenMC simulation and tally information about the neutronics in the simulation. A particle source in the center of the geometry emits neutrons.
* **plot\_simulations\_2d.py**: Uses results from **simulate\_tokamak.py** to plot a graph comparing variables that are controlled against chosen tally scores. Points can be highlighted over to check the conditions of the simulation.

README:

In **simulate\_tokamak.py**, we start off by defining materials which we can use in the simulation. If we want to add just one element, we would apply the following code segment (using lead as an example).

pb = openmc.Material(name="pb")

pb.add\_element('Pb',1.0,"ao")

pb.set\_density("g/cm3",11.34)

The fields labeled 1.0 and “ao” specify that this material is 100% lead. Alternatively, if we were to add another element, such as Be, to create a mixture, we could split the content of the mixture to be half and half, labeling Pb with 1.0/2.0 and “ao” and Be with 1.0/2.0 and “ao”. We could also use “wo” instead of “ao” if we wanted Be and Pb to each take up half of the total weight of the mixture. If we want to add a molecule, we would use the openmc.Material.add\_nuclide method instead, using the same parameters.

flibe = openmc.Material(name="flibe")

flibe.add\_nuclide("F19",4.0/7.0,"ao")

flibe.add\_nuclide("Li6",0.40\*2.0/7.0,"ao")

flibe.add\_nuclide("Li7",(1-0.40)\*2.0/7.0,"ao")

flibe.add\_nuclide("Be9",1.0/7.0,"ao")

flibe.set\_density("g/cm3",1.94)

We can add all these materials to the openmc.Materials class.

We then build the geometry for our reactor and apply the materials we create to different components of the reactor. The following segment creates a 3D geometry with concentric spheres of different lengths in some confined region (here it is listed as 500 cm x 500 cm x 500 cm, but should be as big as it needs to be to fit all other reactor components within this box). We make the boundary type of this confinement box a vacuum so that when the simulation is ultimately run, neutrons do not scatter off the edges of the box and reenter the reactor components. All these neutrons that reach the edge of the box will vanish. However, neutrons will scatter off particles inside the different components where we can detect interesting physics.

box = openmc.model.rectangular\_prism(500,500,boundary\_type='vacuum') \

& -ZPlane(250, boundary\_type='vacuum') \

& +ZPlane(-250, boundary\_type='vacuum')

inner = openmc.Sphere(r=5)

multiplier = openmc.Sphere(r=5+multiplier\_thickness)

breeder = openmc.Sphere(r=5+multiplier\_thickness+breeder\_thickness)

surfaces = [inner, multiplier, breeder]

s1, s2, s3, s4 = openmc.model.subdivide(surfaces)

outside = box & ~openmc.Union((s1, s2, s3))

regions = [s1, s2, s3, outside]

The openmc.model.subdivide takes the individual spheres we created and makes them into concentric spheres so that we have multiple layers in which to fill material. The variable labeled s2 corresponds only to the neutron multiplier layer, and s3 only the breeder layer. The s4 layer is an infinitely large layer outside the breeder that is spit out, but we can ignore it. We then have to fill each of the regions of the geometry with the different materials we defined earlier.

fills = [None,pb,flibe,None]

names = [mat.name if hasattr(mat,"name") else "" for mat in fills]

cells = [openmc.Cell(region=reg,fill=fill,name=name) for reg, fill, name in zip(regions,fills,names)]

We can now export this geometry to an xml file to be saved for running the simulation.

root = openmc.Universe(cells=cells)

geom = openmc.Geometry(root)

geom.export\_to\_xml()

We then must define the source of neutrons and how they’re being emitted. OpenMC is not a fusion reactor library: it does not simulate plasmas. It just emits neutrons from a point source and examines the interactions between the neutrons and other materials. We use the openmc.Source class to identify the location of the source and the energy of each neutron from the source as well as how the source is emitted (Discrete, Isotropic, etc.). The openmc.Settings class identifies other conditions for the simulation:

* batches → how many simulations will be run; this reduces the standard deviation of the tally scores
* inactive → how many of the batches will not be counted in the simulation; this calibrates the simulation for the later batches that will be counted
* particles → how many neutrons will be emitted from the source

point = openmc.stats.Point((0, 0, 0))

src = openmc.Source(space=point)

src.energy=openmc.stats.Discrete([14.0E6],[1.0])

settings = openmc.Settings()

settings.run\_mode = 'fixed source'

settings.source = src

settings.batches = batches

settings.inactive = 5

settings.particles = nps

settings.export\_to\_xml()

There's a number of different "tallies" that can be analyzed. A tally simply defines a variable you want to examine in the simulation. For example, let’s say we want to examine TBR. TBR requires us to know how much tritium is produced vs tritium burned. Since no tritium is burned because there is no plasma, we can estimate the TBR by instead analyzing how much tritium is produced vs how many initial neutrons we emit from our source. In a plasma, for each tritium burned, a neutron is produced, which we can model as our source neutrons: T (tritium) + D (deuterium) → n + He-4. In that case, we can use the score ‘205’ or ‘(n,t)’, which grabs how many (n,T) reactions occur per source particle, equaling the TBR value.

We also need to define the filters to determine which reactor components (or cells as OpenMC calls them) we are considering using openmc.CellFilter and to filter for reactions with just neutrons using openmc.ParticleFilter, which can only filter for neutrons, photons, electrons, or positrons. There is no reaction that begins with a neutron and ends with a triton that also involves one of these other 3 particles, so adding this particle filter is redundant, but for other reactions or tally scores it may not be.

particle\_filter = openmc.ParticleFilter('neutron')

flibe\_cell = openmc.CellFilter(cells[2])

tbr = openmc.Tally(name='TBR')

tbr.filters = [flibe\_cell,particle\_filter]

tbr.scores = ['205']

The score number corresponds to an ENDF-MT number from the <https://www.nndc.bnl.gov/endf/help.html> site. However, OpenMC does not tally cross sections, even though on this site, the number often says it corresponds to a cross section for a certain reaction. Instead, OpenMC tallies get the number of reactions for said reaction per source particle. A list of tally scores and units is also described accurately in the OpenMC documentation: <https://docs.openmc.org/en/stable/usersguide/tallies.html>.

If we want to tally, say, the energy of neutrons that pass through a certain surface, we will first need to specify an openmc.SurfaceFilter using the openmc.Sphere objects we generated for the geometry. The score ‘current’ tallies the number of neutrons that pass through the surface. However, if we want the energies of these neutrons, we’ll need to apply an openmc.EnergyFilter. OpenMC keeps track of the energy of each neutron individually, so for simplicity, it allocates the neutrons into energy bins rather than spitting out the actual energy value of each neutron. If we later print out the results of this tally, we’ll just get the number of neutrons within a given energy range per source neutron allocated to their appropriate energy bins. We can add more bins and reduce their sizes to get a more precise energy value range in which the neutron can be.

surface\_filter\_before\_multi = openmc.SurfaceFilter(inner)

surface\_filter\_after\_multi = openmc.SurfaceFilter(multiplier)

energy\_bins = np.linspace(0, 24e6, 25)

energy\_filter = openmc.EnergyFilter(energy\_bins)

energybefore = openmc.Tally(name='energy\_before\_neutronmultiplier')

energybefore.filters = [surface\_filter\_before\_multi,particle\_filter,energy\_filter]

energybefore.scores = ['current']

energyafter = openmc.Tally(name='energy\_after\_neutronmultiplier')

energyafter.filters = [surface\_filter\_after\_multi,particle\_filter,energy\_filter]

energyafter.scores = ['current']

To add these tallies to our simulation, we add them to the openmc.Tallies class.

tallies = openmc.Tallies([tbr, energybefore, energyafter])

tallies.export\_to\_xml()

We now have all components to run the OpenMC simulation.

model = openmc.model.Model(geom, mats, settings, tallies)

model.run()

After the simulation is finished, we’ll want to prepare a .json file to be returned so that we can use the control variables and tallies to be used for plotting later. All variables that are to be examined for plotting, other than tallies, must be put in the json\_output list below.

json\_output= {'breeder\_thickness':breeder\_thickness,

'multiplier\_thickness':multiplier\_thickness}

OpenMC outputs the results in a statepoint.#.h5 file, where the ‘#’ is replaced by the number of batches executed. If we ran 50 batches, then this file would be labeled as statepoint.50.h5. A summary.h5 file is also created that gathers results from all batches combined. In general, we will only care about the statepoint file since the final results in this file have the least amount of standard deviation and also provide more information about the simulation. We can use openmc.Statepoint to extract the results of the tally and then put them in the json file.

sp = openmc.StatePoint('statepoint.'+str(batches)+'.h5')

tallies\_to\_retrieve = ['TBR']

for tally\_name in tallies\_to\_retrieve:

tally = sp.get\_tally(name=tally\_name)

tally\_result = tally.sum[0][0][0]/batches

#for some reason the tally sum is a nested list

tally\_std\_dev = tally.std\_dev[0][0][0]/batches

#for some reason the tally std\_dev is a nested list

json\_output[tally\_name] = {'value': tally\_result,

'std\_dev':tally\_std\_dev}

For the neutron energy tallies, however, we have to follow a slightly different procedure. Recall that, unlike for the ‘TBR’ case which prints out just a single value, the energy tallies print out a value per energy bin. In our example, there are 24 energy bins. So, to plot just one result for the initial conditions the user specifies, we’ll want to compute the average neutron energy.

spectra\_tallies = ['energy\_before\_neutronmultiplier','energy\_after\_neutronmultiplier']

for spectra\_name in spectra\_tallies:

spectra\_tally = sp.get\_tally(name=spectra\_name)

spectra\_tally\_result = [entry[0][0] for entry in spectra\_tally.mean]

spectra\_tally\_std\_dev = [entry[0][0] for entry in spectra\_tally.std\_dev]

sumr = np.sum(spectra\_tally\_result)

energy = 1e6

sumenergy = 0

sumstddevenergy = 0

for elem in spectra\_tally\_result:

sumenergy += elem\*energy/sumr

energy = energy + 1e6

energy = 1e6

for elem in spectra\_tally\_std\_dev:

sumstddevenergy += elem\*energy/sumr

energy = energy + 1e6

json\_output[spectra\_name] = {'value': sumenergy,

'std\_dev': sumstddevenergy,

'energy\_groups':list(energy\_bins)}

We can loop through different initial conditions to run the simulation, which is implemented at the bottom of the script, and then we output the json file to be used by one of the plotting scripts.

results = []

c=0

for i in tqdm(range(0,40)):

c=c+1

for breeder in breeder\_material\_name:

multiplier = c/2

result = make\_geometry\_tallies(batches=30,

nps=10000,

breeder\_thickness=40,

multiplier\_thickness=multiplier,

)

results.append(result)

output\_filename = 'simulation\_results.json'

with open(output\_filename, mode='w', encoding='utf-8') as f:

json.dump(results, f)

The actual **simulate\_tokamak.py** file may have additional parameters built in, namely pressure and temperature variables, for the user’s convenience. The **plot\_simulations\_2d.py** script generates the html files by opening the name of the json file, grabbing the desired observables, and plotting them.