Introduction to Monte Carlo

Geoffrey Gunow

October 7, 2015

In the last assignment, a one-group Monte Carlo solver was implemented for a uniformly distributed external source in a homogeneous infinite medium with no fission. In this assignment we will ease these assumptions to create a more realistic solver.

1 Neutron Source

In the previous assignment, we assumed that neutrons started at a position chosen randomly and uniformly throughout the geometry. Physically, this would relate to some neutron source distributed in throughout a reactor core that produced neutrons at a constant rate. In real reactor cores, these type of sources can exist, however they are typically dwarfed by the neutrons formed from fission. Figure 1 shows the typical diagram of the fission reaction which powers nuclear reactors.

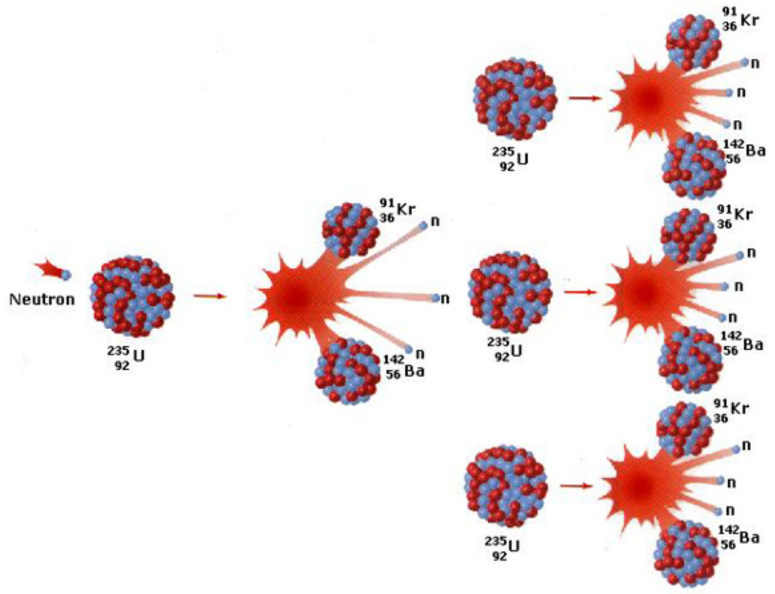

Figure 1: Illustration of the fission process.

In this diagram, an initial neutron can generate a fission which splits an atom and generates more neutrons. The subsequent neutrons can then create more fission events.

This chain reaction would of course go on forever and consume all uranium if the process went unchecked. However, there are physical forces that counteract this. Mainly, neutrons can be absorbed by non-fissile materials as well. This is actually the case in assignment 1 where each absorption event lead to the death of a neutron without generating more neutrons. Once important quantity in reactor physics is the eigenvalue k defined as

$$k = \frac{\text{Neutron Gains}}{\text{Neutron Losses}}. \tag{1}$$

Neutron gains arise primarily from fission whereas neutron losses are due to absorption events and neutrons leaking out of the system. Therefore, we can rewrite our equation as

$$k = \frac{\text{Fission}}{\text{Absorption} + \text{Leakage}}. \tag{2}$$

The eigenvalue k also describes the change in neutron population between "neutron generations." For instance, if in generation 1 there are N neutrons, there will be kN neutrons in generation 2. This leads to a simple relation for the neutrons N in generation n as

$$N = N_0 k^n \tag{3}$$

where N_0 is the initial neutron population in generation 0. Therefore, for a reactor to be in steady-state or *critical*, k must be exactly 1.0. This would seem impossible, but nature has a way of adjusting neutron absorption to force k towards 1.0. In simulations, we assume some absorption cross-section, almost assuredly leading to a calculation of k not equal to 1.0. This computed value is useful because if k is greater than 1.0 our reactor will run hotter than we planned and if k is less than 1.0 it will run colder, and possibly shut down. In this assignment we will include fission as well as a calculation of k using a the simple ratio shown in Eq. 2. Particularly, we will focus on steady-state problems.

Steady-state problems are important for reactor design since they describe the behavior of the reactor mid-operation when the power is being produced. During this time, we would hope that the reactor would produce a constant power output and there would be relatively static behavior. A reactor designer might run a simulation to compute the power output, the peak temperature, etc. All of these require knowing both k and the neutron distribution within the reactor core. This leads to an interesting problem: we are trying to solve for the neutron distribution but our current solver requires that we input some *initial* neutron distribution.

We solve this problem by creating neutron batches. In each batch we assume a neutron distribution that comes from the fission locations of the previous batch. For the first batch we assume a simple neutron distribution, such as a uniform distribution. When we sample a neutron location, we randomly select a fission location from the fission bank. In this manner we have a nested for loop structure – an outer loop over batches and an inner loop over neutron histories.

2 Geometry

In addition to updating the neutron source, we will also update the geometry to contain boundary information. We will be concerned with two boundary conditions – vacuum and reflective. A vacuum boundary condition implies that all neutrons that leave the geometry

Table 1: Distances to surface intersections

Surface	Distance
$x = -1$	$(-1 - 0.7) \,/\, \left(-\sqrt{1/2}\right) = 2.48$
$x = +1$	$(1 - 0.7) \,/\, \left(-\sqrt{1/2}\right) = -0.35$
$y = -1$	$(-1 - (-0.5)) \,/\, \left(-\sqrt{1/5}\right) = 1.12$
$y = +1$	$(1 - (-0.5)) \,/\, \left(-\sqrt{1/5}\right) = -3.35$
$z = -1$	$(-1 - 0.5) \,/\, \left(\sqrt{1/5}\right) = -3.35$
$z = +1$	$(1 - 0.5) \,/\, \left(\sqrt{1/5}\right) = 1.12$

through the boundary are eliminated from consideration and never return to the geometry. They are simply tallied as leaked neutrons and the neutron history is terminated.

For reflective boundaries, the neutron is sent back at the reflected angle in the geometry, much like a billiard ball hitting an edge. We will limit ourselves to rectangular geometries for simplicity. The reflected angles can be calculated simple by negating components of the unit vector. For instance, if a neutron is traveling in direction $\left\langle \sqrt{1/3}, \sqrt{1/3}, \sqrt{1/3} \right\rangle$ and reflects off the surface $x = 1$, the reflected direction would be $\left\langle -\sqrt{1/3}, \sqrt{1/3}, \sqrt{1/3} \right\rangle$. If needed the reflected angles (θ and ϕ) can be calculated using simple trigonometry. When writing the code for this, beware of edge and corner cases. When hitting a x-y edge, both the x and y components need to be negated. When hitting a corner (x-y-z), all components need to be negated.

2.1 Example

Suppose we are solving a homogeneous cube with reflected boundaries at the planes $x = -1$, $x = 1$, $y = -1$, $y = 1$, $z = -1$, and $z = 1$. We sample a neutron at location $(0.7, -0.5, 0.5)$ traveling in direction $\left\langle -\sqrt{1/2}, -\sqrt{1/5}, \sqrt{1/5} \right\rangle$. Then we sample a distance to collision of 1.5. If we calculate the endpoint given the direction vector and the distance to collision, we find the coordinates $(-0.31, -1.17, 1.17)$ which is clearly outside the boundaries of the geometry. Therefore, there must have been an intersection with a boundary surface. Systematically, we calculate the distance to each surface in the direction of travel in Table 1.

Negative distances are disregarded since they imply an intersection not in the direction of travel. Therefore the shortest distance is 1.12, which is shorter than the sampled distance to collision. Therefore, the particle is moved to the intersection. Since it is a y-z intersection, both the y and z components of the unit vector are negated and the new direction of travel becomes $\left\langle -\sqrt{1/2}, \sqrt{1/5}, -\sqrt{1/5} \right\rangle$. We attempt to travel the remaining $1.5 - 1.12 = 0.38$ in that new direction.

3 Code Structure

In this assignment we will begin to implement object oriented programming. This type of programming structure allows for transparency, readability, and ease in altering components of the code. The basic idea in object oriented programming is to divide the data into various classes that communicate with each other via some interface. To start we will introduce the usage of classes without an object oriented framework. Then we will start implementing the updated Monte Carlo solver with this type of code structure.

3.1 Using Classes

In the provided code, various Monte Carlo functions have been grouped and moved to separate files. Additionally, the `Material`, `Boundaries`, `Coords` classes have been added for your convenience. Restructure `monte_carlo.py` and `distributions.py` to use these classes. Why might this structure be preferred over the structure in assignment 1?

3.2 Including Geometric Boundaries

Update `monte_carlo.py` to include the ability to handle vacuum and reflective boundary conditions. This will require checking for boundary crossings as described in Section 2 as well as handling reflected directions.

3.3 Creating a Tally Class

Create a new file named `tally.py` and add it to the git repository by typing `git add tally.py` from the command prompt. In this file, include a class named `Tally` which records all the relevant tallies. This will include the mean crow distance, as calculated in assignment 1. Also include leakage, absorption, and fission tallies. Whenever a neutron leaks from a vacuum boundary in the geometry, increment the leakage tally. Likewise whenever a neutron is absorbed, increment the absorption tally. Do not worry about updating the fission tally for the moment. All these tallies should be initialized to zero. Update `monte_carlo.py` to make use of the `Tally` class.

3.4 Adding Fission

Now add fission to the program. This will require making yet another new file named `fission.py` including a class named `Fission` which records fission sites (where the fissions occur) and keeps a fission bank for sampling initial neutron positions. Update `monte_carlo.py` to include fission events and include a loop over batches as discussed in Section 1. The new code structure will have a loop on the outside over batches and the inner loop will be over neutron histories used in assignment 1. At the end of each batch, the fission cites will be copied to the fission bank to sample a neutron position. Use the function `sample_fission_site` to choose a neutron location. When an absorption occurs, sample whether the event is a fission event using `sample_fission`. When the event is a fission event, sample the number of fission neutrons using `sample_num_fission`. Store n copies of the neutron location to the fission sites where n is the number of sampled neutrons. Also, add n to the fission tally. A description of the new program structure with fission is given in Figure 2.

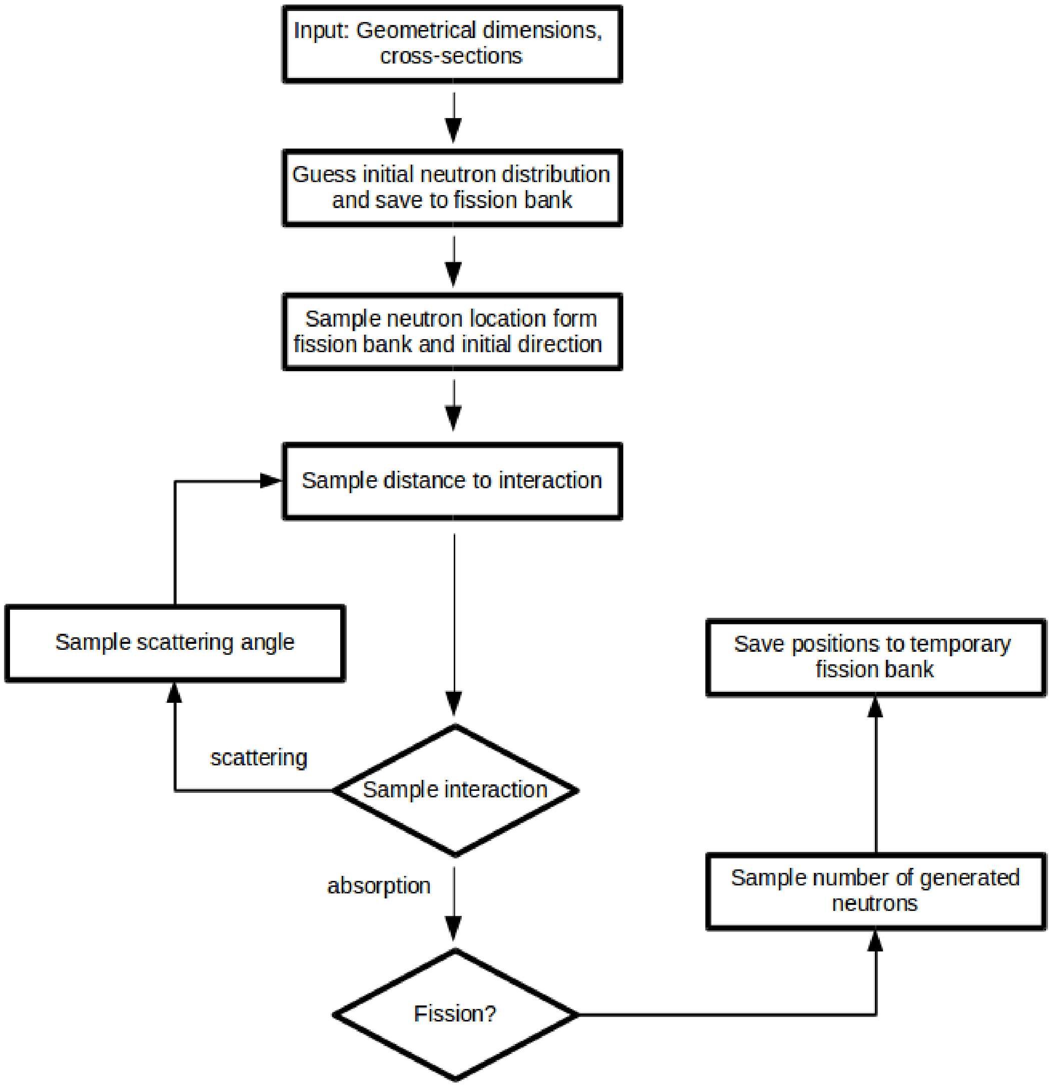
Input: Geometrical dimensions, cross-sections
Guess initial neutron distribution and save to fission bank
Sample neutron location form fission bank and initial direction
Sample distance to interaction
Sample scattering angle
Save positions to temporary fission bank
scattering
Sample interaction
absorption
Sample number of generated neutrons
Fission?

Figure 2: Monte Carlo program with fission.

3.5 Calculating k

Now the code should be ready to calculate a real value for k. Simply calculate k for each batch using the fission, absorption, and leakage tallies with Eq. 2. Do this for cross-sections of you choosing with a reflective boundary conditions on every surface. For this case, the value of k should converge to

$$k = \frac{\nu\Sigma_f}{\Sigma_a}. \tag{4}$$