Introduction to Monte Carlo

Geoffrey Gunow

September 30, 2015

1 Introduction to the Monte Carlo Method

Monte Carlo is a broad method to solve numerical problems through random sampling. In this first exercise, the method will be introduced to calculate π through solving a simple integral. In particular, we are interested in calculating the area of a circle of radius 1.0. This area should of course be π. A depiction of this problem is shown in Figure 1.

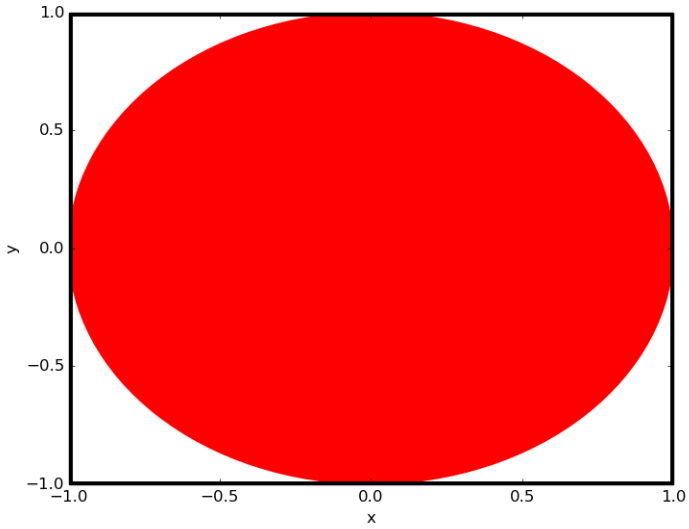

Figure 1: Unit circle inside bounding box of area 4.

Imagine that the circle is a dart board. Assume that we can generate randomly thrown darts over the bounding square of length 2 and area 4. The probability p that the dart will hit the dart board should be equivalent to the ratio of the circle's area to the bounding square's area. In particular this probability can be given by

$$p = A_{\text{circle}}/A_{\text{square}} = A_{\text{circle}}/4. \tag{1}$$

We can re-write this in terms of the area of the circle, which we know to be π as

$$\pi = 4p. \tag{2}$$

Therefore, if we can determine the probability p of the dart hitting the board, we can easily form an estimate of π. To generate the probability we simple generate random numbers. We generate T darts and count how many randomly darts hit the target. If this number is H, the underlying probability p can be estimated as

$$p \approx H/T. \tag{3}$$

Due to the central limit theorem, as the number of darts generated increases, the calculated probability should converge to the true probability.

1.1 Computing Pi

Initial code has been provided in `circle.py`. Complete the remaining code to calculate an estimate of π using Monte Carlo. How many darts does it take to converge the famous first three digits of π?

2 Introduction to Neutron Transport

In neutron transport we are concerned with the neutron population n. In particular we are concerned with where neutrons are located in a reactor core, which direction they are traveling, and their velocity at all times. Another way of expressing this is saying that we are interested in the quantity

$$n(x, y, z, u_x, u_y, u_z, v, t) \tag{4}$$

where (x, y, z) is the location, (u_x, u_y, u_z) is the unit vector describing the direction of travel, v is the velocity, and t is time. In this notation we attempt to express the quantity of interest (neutron population) as a function of its independent variables. If we examine the variables listed, we notice a slight problem. The values (u_x, u_y, u_z) are correlated. Since they describe a unit vector, knowing two of these components yields the third because a unit vector is required to be of length 1.0

$$u_x^2 + u_y^2 + u_z^2 = 1 \tag{5}$$

For instance if I know that a neutron is traveling in direction $u_x = 1.0$ and $u_y = 0.0$,

$$u_z = \sqrt{1 - u_x^2 - u_y^2} \tag{6}$$

meaning that u_z must also equal zero:

$$u_z = \sqrt{1 - 1^2 - 0^2} = 0. \tag{7}$$

For this reason a more correct notation would omit one of these variables, for instance u_y and reduce our notation to

$$n(x, y, z, u_x, u_z, v, t). \tag{8}$$

This notation is rather awkward due to the seemingly mysterious lack of the y component in the unit vector. Fortunately, spherical coordinates of the form (r, θ, ϕ) provide a much better way of expressing directions since a unit vector simply implies $r = 1$. Therefore, we can rewrite this equation as

$$n(x, y, z, \theta, \phi, v, t) \tag{9}$$

which is starting to look more natural. However, it is still not entirely satisfactory since we have now mixed cartesian and spherical coordinate systems in an equation. Therefore, a more general way to write this equation is to let the reader *choose* whichever coordinate system they prefer. For instance, there are some problems in which the cartesian coordinate system is the most practical and there are others where spherical or cylindrical coordinates may be preferred. Therefore we express this equation as

$$n(\vec{r}, \vec{\Omega}, v, t) \tag{10}$$

where $\vec{r}$ now refers generally to the position (equivalent to (x, y, z) in the old notation) and $\vec{\Omega}$ refers generally to the direction (equivalent to (θ, ϕ) in the old notation). This expression is now far more general and compact but it is important to remember that the vector quantities $\vec{r}$ and $\vec{\Omega}$ implicitly include multiple variables where $\vec{r}$ describes three independent variables and $\vec{\Omega}$ describes two independent variables. One last change we will make to this equation is to convert from velocity v to energy E. Since in classical physics

$$E = \frac{1}{2}mv^2 \tag{11}$$

and m is the known mass of a neutron, there is a clear one-to-one correspondence between velocity and energy. From the velocity it is simple to calculate the energy. Often, data is presented in terms of energy rather than velocity so the neutron population is likewise described in terms of energy leading to the final description of the neutron population

$$n(\vec{r}, \vec{\Omega}, E, t). \tag{12}$$

The arguments $\vec{r}$, $\vec{\Omega}$, E, and t are often referred to as the *phase space*. To run a simulation using Monte Carlo, we attempt to simulate as many particles covering the phase space as possible. The idea is to simply "think like a neutron." Many neutron lifetimes are simulated from "birth" until "death."

Often, we are concerned with steady-state problems in which the time variable is no longer a factor. Examples of these type of problems would include calculating the neutron distribution during operation, which we design to be relatively time-invariant. In our first Monte Carlo problem, we will make a few more assumptions. First we will assume that all neutrons have the same energy. This is an extreme approximation but we will correct it later. Second, we will assume all neutrons are created from an *external* source distributed uniformly throughout the problem. Normally, the bulk of the neutrons are created from fission but we will see later how this can cause additional simulation concerns. Lastly, we will assume an infinite and homogeneous medium.

For this heavily simplified problem, we will create a Monte Carlo solver in Python. This solver will follow neutrons from birth, through scattering interactions, until final absorption. In any neutron transport Monte Carlo code, you can follow neutrons through the geometry and end up calculating nothing meaningful. In order to extract meaningful information, we need to implement *tallies*.

For this first problem, we will implement a simple tally – the mean crow distance. This is the average distance between the start point and the end of the neutron histories. To implement this, temporarily save the start point of the neutron. Upon absorption, calculate the distance from the initial start point. Accumulate all of these tallies and take the average to calculate the mean crow distance.

In order to solve any neutron transport problem, we need to know the geometry and material properties, specifically cross-sections. Cross-sections dictate the probability of a neutron interaction with the material. However, the units are a bit strange – inverse distance. Cross-sections are essential to calculate anything in neutron transport. For this problem we need to know three cross sections – the total cross-section Σ_t, the scattering cross-section Σ_s, and the absorption cross-section Σ_a. As these names would imply, the total cross-section describes the total interaction probability (the probability of any interaction), the scattering cross-section describes the probability of scattering, and the absorption cross-section describes the probability of absorption. In general,

$$\Sigma_t = \Sigma_s + \Sigma_a \tag{13}$$

So from the total and scattering cross-sections, we can calculate the absorption cross-section. The geometry will be a simple rectangular prism (3D rectangle) described by bounds $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, $z_{\min}$, and $z_{\max}$. However, since we are assuming an infinite homogeneous geometry these bounds will only be used to compute starting locations. The structure of the program will resemble the physics that the neutron sees, as shown in Fig. 2.

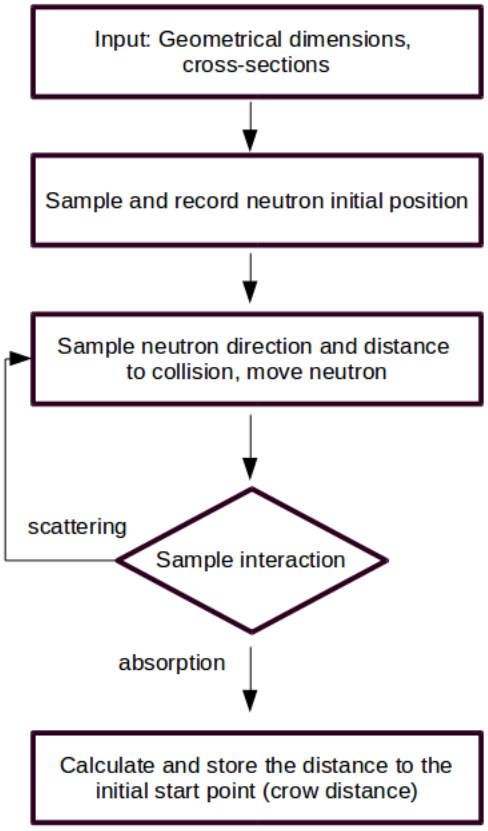

Figure 2: Simplified Monte Carlo algorithm.

2.1 Understanding Angular Sampling

2.2 First Monte Carlo Solver

Sample code has been provided in `monte_carlo.py`. Use and extend this code to implement your first Monte Carlo solver. Much of the structure you need has already been implemented but you will need to complete the function `transport_neutron`. Note that you will need to create variables to store and update the neutron location. You will also need to remember how to use polar and azimuthal angles with a distance to update the position.