Section 9

Monte Carlo Simulation

Some physical systems are so complicated that it is either too difficult to write down the equations describing them, or these equations may be too difficult to solve. In these circumstances a computer method called *Monte-Carlo simulation* may be a way of making predictions or confirming that a theory adequately accounts for an observed phenomenon. Programs for this are often short and simple to write, but they can consume very large quantities of computer time and very accurate results are difficult to achieve.

Q9.1 For this section you should keep detailed notes of what you do and the results you find, together with copies of your code. Your mark will be based on all of these.

9.1 Estimation of the critical mass of Uranium 235

The chain reaction in a nuclear reactor or nuclear bomb goes as follows. A uranium atom is hit by a stray primary neutron and undergoes fission, with the release of energy, and the production of several more neutrons. These neutrons may escape from the piece of uranium or they may hit other nuclei and initiate further fissions. In a small piece of uranium most neutrons will escape but above some critical volume a self-sustaining chain reaction will occur.

Rather than try to follow the fate of all the neutrons produced in all the subsequent fissions, we will track just those neutrons produced in the first fission, and count how many secondary fissions they initiate. Since this number will fluctuate, we will have to repeat our simulation for some large number of initial fissions to find the average number of secondary fissions per initial fission.

9.2 A one-dimensional model

To begin with we will simplify to a one-dimensional model. Suppose the uranium extends along a line from 0 to L. The initial fission may occur anywhere along the line, so we generate a random number uniformly-distributed in the interval from 0 to L using `L*random.random()` (after importing `random` from numpy). The number of extra neutrons produced in each fission varies; on *average* it is 2.5, but for the moment we will simply take it to be 2 in every case.

These neutrons move along the line, colliding with uranium nuclei. In most cases they simply bounce off, but there is a chance of initiating another fission before they diffuse out of the end of the line. It can be shown that the average (rms) distance that a neutron diffuses away from its starting point before causing a fission is $R = (2ab)^{1/2}$ where a is the mean free path between the neutron hitting another nucleus and bouncing off, and b the mean free path between the neutron hitting a nucleus and causing fission. Experimental values are $a = 1.7\,\text{cm}$ and $b = 21\,\text{cm}$. For the moment assume that all the neutrons travel exactly the average distance.

The procedure is now to generate some number, say 100, of initial fissions. In your code you will need an outer loop counting these 100 initial fissions. Within the outer loop you will need to deal with each of the two secondary neutrons and the easiest way is to use a loop counting up to 2. For the moment assume that all the neutrons travel exactly the average distance. They may be travelling to the right or the left (remember we are in one dimension) and we ignore the possibility that their directions might be correlated. So for each of the two neutrons generate a random number `random.random()`. If it is greater than 0.5 take the neutron moving to the right, otherwise to the left. Add or subtract the average diffusion length from the initial position (according to whether the neutron is going right or left) to find the possible positions for the next two fission events. If a calculated position is outside the range 0 to L then the neutron will escape without causing further fission. Use a variable to count the secondary fissions as they occur. Increase it by one for each neutron that causes a secondary fission inside the bar.

Notice that the number of secondary fissions fluctuates if you run the program several times.

Before attempting accurate measurements on the critical size of the uranium we should allow for the fluctuations in the number of secondary neutrons. The function `neutrons()` has been designed to give integer values with average 2.5 and with a distribution resembling that observed experimentally. It is available in the file `neutrons.py`, together with another function you will need later.

Start with $L = 0.1\,\text{m}$ and vary this to determine the critical value. Decide whether

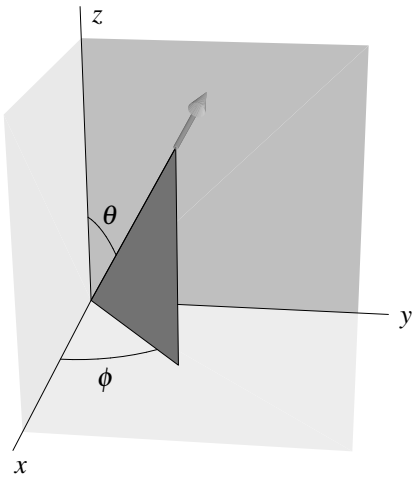

Figure 9.1 Specifying a direction in three dimensions with angles θ and ϕ.

you need to use more than 100 initial neutrons.

9.3 Extension to three dimensions

One dimensional models cannot be taken very seriously but are often used to get an order of magnitude estimate when the full three dimensional calculation is beyond the capacity of the computer available. In this case, a three dimensional simulation is feasible.

Consider a cube of uranium of side L. The initial fission requires 3 coordinates (x,y,z) chosen uniformly between 0 and L. The secondary neutrons are emitted in random directions. Think of the fission occurring at the centre of a sphere and the direction of an emitted neutron being specified by the latitude and longitude where its track crosses the sphere. We conventionally use the *co-latitude*, θ, that is the angle from the North pole, *not* from the equator, as shown in figure 9.1. All angles of longitude ϕ are obviously equally likely so we can generate a random value for ϕ with `2.0*pi*random.random()`. In contrast, values of θ are not equally likely; we want the neutron paths to be uniform over the surface of the sphere, and a small range of θ near the pole corresponds to a much smaller area than the same range near the equator. It can be shown that $\cos\theta$ is uniformly distributed, so we can generate θ from

```
acos(2.0*random.random()-1.0)
```

The model for the diffusion of the neutrons can be improved. It can be shown that the probability that the neutron diffuses a distance s before causing fission is proportional to $s^2\exp(-3s^2/2R^2)$. A function `diffusion()` (in file `neutrons.py`) generates a random number s with this distribution. These numbers have average value of unity so need to be scaled by $R = (2ab)^{1/2}$ before use. The coordinates of the next fission are then given by:

$$\begin{aligned} x' &= x + s\sin\theta\cos\phi \\ y' &= y + s\sin\theta\sin\phi \\ z' &= z + s\cos\theta \end{aligned}$$

Now test whether this point is inside the cube.

Vary the value of L and find the critical value and hence the critical mass. The density of uranium is $18.7\,\mathrm{Mg\,m^{-3}}$.