11

Introduction to Monte Carlo Methods

11.1 Analog Physics

In this lecture we'll talk about using pretend neutrons to see what will happen in a system. This process where we use simulated particles that behave analogous to neutrons (or other particles) is an example of analog physics. This gives it a sort retro-sounding ring to the process, like it is something that goes along with vinyl records and psychadelic music.

The way we'll deal with these particles is using random sampling of their interactions, just as quantum mechanics governs the behavior of particles using probabilities. These methods are called Monte Carlo methods. The first modern Monte Carlo methods were developed by Stanislaw Ulam during the Manhattan Project and Nicholas Metropolis coined the name after a famous casino.

11.1.1 Probability Preliminaries

We'll need to know a few things about probabilities before we begin. A cumulative distribution function (CDF) is defined as a function $F(x)$ that is the probability that a random variable from a particular distribution c is less than x. In mathematical form we write this as

$$F(x) = P(c < x).$$

Because probabilities are always in $[0, 1]$ the function $F(x) \in [0, 1]$. Furthermore, the way that the CDF is defined we have two imporant limits:

$$\lim_{x\to\infty} F(x) = 1, \qquad \text{and} \qquad \lim_{x\to-\infty} F(x) = 0.$$

Along with the CDF we will use the probability density function (PDF) which is written as $f(x)$. The PDF is defined such that

$$f(x)dx = \text{The probability that the random variable is in } dx \text{ about } x.$$

The PDF is the derivative of the CDF and they are related by

$$f(x) = \frac{dF}{dx}, \qquad \text{and} \qquad F(x) = \int_{-\infty}^{x} f(x)\, dx.$$

Also, from these relations we get

$$\int_{-\infty}^{\infty} f(x)\, dx = 1,$$

this relation is called the normalization coefficient.

11.1.2 The Exponential Distribution

The most important distribution for Monte Carlo methods for radiation transport is the exponential distribution. This is because the probability that a neutron travels a number of mean-free paths in $d\lambda$ about λ before having a collision is given by

$$f(\lambda)\, d\lambda = e^{-\lambda}\, d\lambda, \text{ for } \lambda > 0$$

To convert this to distance, rather than mean-free paths, we define

$$\Sigma_t x = \lambda, \qquad d\lambda = \Sigma_t dx.$$

Making this substitution we get the PDF for the exponential distribution as

$$f(x) = \Sigma_t e^{-\Sigma_t x}.$$

We can check that this is a proper PDF by integrating over all x and seeing that the integral equals one

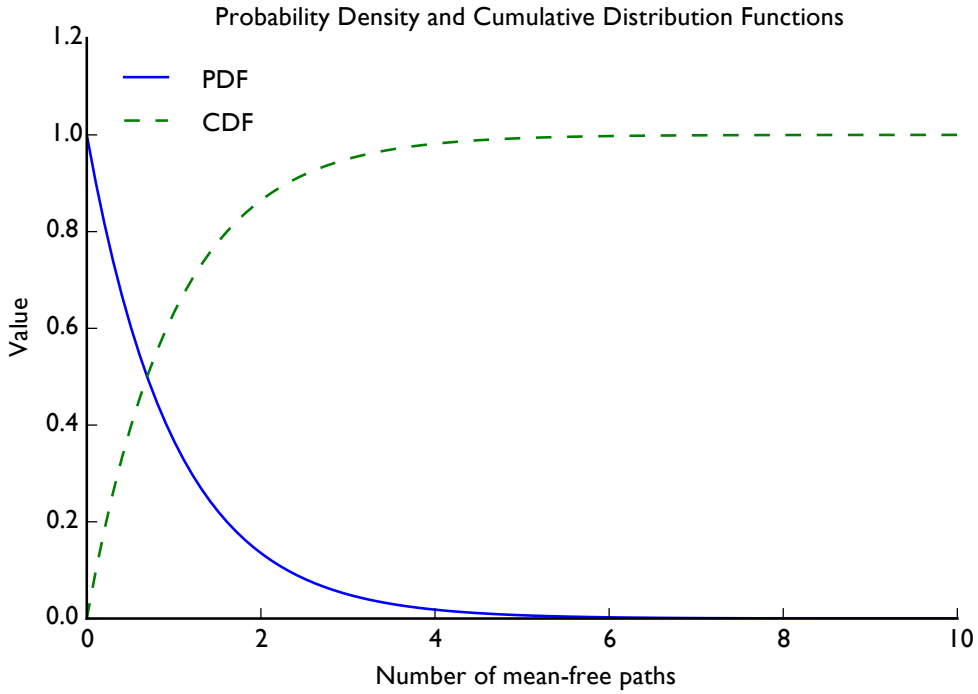
$$\int_0^\infty dx \Sigma_t e^{-\Sigma_t x} = -\, e^{-\Sigma_t x}\Big|_0^\infty = 1.$$

From this we can also define the CDF:

$$F(x) = \int_0^x dx \Sigma_t e^{-\Sigma_t x} = -\, e^{-\Sigma_t x}\Big|_0^x = 1 - e^{-\Sigma_t x}.$$

We can plot the CDF and PDF for this distribution using Python.

```
In [1]: x = np.linspace(0,10,100)
        f = np.exp(-x)
        F = 1-f
        plt.plot(x,f,label="PDF")
        plt.plot(x,F,'--',label="CDF")
        plt.xlabel("Number of mean-free paths")
        plt.ylabel("Value")
        plt.legend(loc='best') #bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
        plt.title("Probability Density and Cumulative Distribution Functions")
        plt.axis([0,10,0,1.2])
```



11.2 Our First Monte Carlo Program

Now we would like to use Monte Carlo solve the following problem: a beam of neutrons strikes a 3 cm thick slab of material with $\Sigma_t = 2.0$ cm^{-1}. What fraction of the neutrons get through the slab without a collision. We know that the answer to this problem is

$$e^{-2\times 3} \approx 0.002478752177.$$

We would like to solve this problem with Monte Carlo using the following procedure

1. Create neutron
2. Sample randomly a distance to collision from the exponential distribution
3. Check to see if the distance to collision is greater than 3.
4. Go back to 1 until we've run ``enough'' neutrons.

At the end of this procedure there the ratio of the number of neutrons that went through the slab to those that we created is the fraction that we are looking for.

The hard part is that we don't know how to generate a random sample from the exponential distribution. We do, however, know how to get a random number between 0 and 1 using the NumPy's random function. We also know that the CDF, $F(x)$ is always between 0 and 1. Therefore, the following procedure can give me a random number from the exponential distribution:

Pick a random number between 0 and 1, call it θ

Invert the CDF to solve for $F(x) = \theta$, this value of x is my random sample.

In our case we need to solve for

$$\theta = 1 - e^{-\Sigma_t x},$$

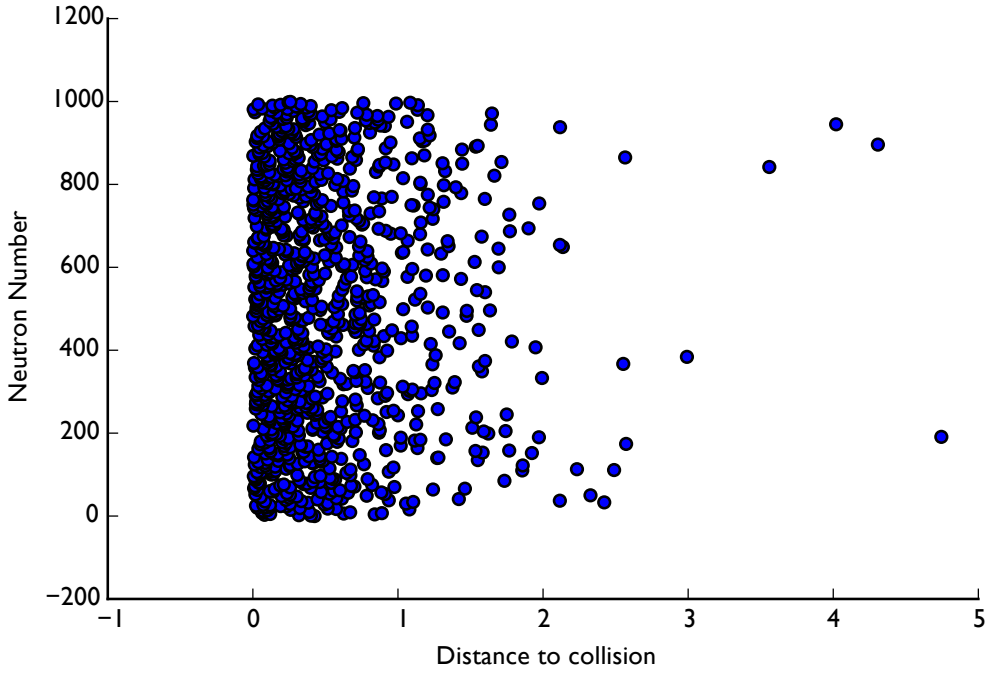
which gives us

$$x = \frac{-\log(1-\theta)}{\Sigma_t}.$$

We can translate this algorithm to python in a few short steps.

```
In [2]: def slab_transmission(Sig_t,thickness,N):
            """Compute the fraction of neutrons that leak through a slab
            Inputs:
            Sig_t:     The total macroscopic x-section
            thickness: Width of the slab
            N:         Number of neutrons to simulate

            Returns:
            transmission:  The fraction of neutrons that made it through
            """
            thetas = np.random.random(N)
            x = -np.log(1-thetas)/Sig_t
            transmission = np.sum(x>thickness)/N

            #for a small number of neutrons we'll output a little more
            if (N<=1000):
                plt.scatter(x,np.arange(N))
                plt.xlabel("Distance to collision")
                plt.ylabel("Neutron Number")
            return transmission
```

```
In [3]: #test the function with a small number of neutrons
        Sigma_t = 2.0
        thickness = 3.0
        N = 1000
        transmission = slab_transmission(Sigma_t, thickness, N)
        print("Out of",N,"neutrons only",int(transmission*N),
              "made it through.\n The fraction that made it through was",transmission)
```




```
Out of 1000 neutrons only 4 made it through.
 The fraction that made it through was 0.004
```

In this case we get a pretty good answer to our question, but let's see if it converges to the correct answer as $N \to \infty$.

```
In [4]:  neuts = np.array([2000,4000,8000,16000,32000,64000,128000,256e3,512e3,1024e3,2056e3])
         for N in neuts:
             transmission = slab_transmission(Sigma_t, thickness, N)
             print("Out of",N,"neutrons only",int(transmission*N),
                   "made it through.\n The fraction that made it through was",transmission)
```

```
Out of 2000.0 neutrons only 6 made it through.
 The fraction that made it through was 0.003
Out of 4000.0 neutrons only 13 made it through.
 The fraction that made it through was 0.00325
Out of 8000.0 neutrons only 29 made it through.
 The fraction that made it through was 0.003625
Out of 16000.0 neutrons only 47 made it through.
 The fraction that made it through was 0.0029375
Out of 32000.0 neutrons only 73 made it through.
```

```
 The fraction that made it through was 0.00228125
Out of 64000.0 neutrons only 165 made it through.
 The fraction that made it through was 0.002578125
Out of 128000.0 neutrons only 341 made it through.
 The fraction that made it through was 0.0026640625
Out of 256000.0 neutrons only 618 made it through.
 The fraction that made it through was 0.0024140625
Out of 512000.0 neutrons only 1285 made it through.
 The fraction that made it through was 0.002509765625
Out of 1024000.0 neutrons only 2528 made it through.
 The fraction that made it through was 0.00246875
Out of 2056000.0 neutrons only 5176 made it through.
 The fraction that made it through was 0.00251750972763
```

We need about 1 million simulated neutrons to get to two digits of the correct answer. Also, if we ran this again we would get different answers because we are using random numbers. The error will go down slowly because the way that Monte Carlo works, the standard deviation of the estimate, that is how much the estimate varies from run to run is proportional to $1/\sqrt{N}$. The standard deviation of the estimate is often called noise. This means that to cut the noise in half we need to quadruple the number of neutrons. Another way to think about it is to say that Monte Carlo methods are one-half order accurate.

11.3 Isotropic Neutrons on a Slab

Even though it takes a lot of particles to make the error in Monte Carlo small, that does not mean it is a bad method. In fact, the nice thing about Monte Carlo is that you have to know very little about the mathematics of the system, all you have to do is be able to push particles around. To demonstrate this let's make our problem a little harder. Now say that the neutrons hitting the slab are not a beam but an isotropic distribution of neutrons where a neutron's path of flight relative to the x-axis is measured by the angle ϕ. Because the distribution is isotropic, any value of $\phi \in [-\pi/2, \pi/2]$ is equally likely.

For a neutron traveling in direction ϕ the slab can look thicker than 3.0 cm, because if the neutron is traveling at a grazing angle to the slab it will have to travel through more of the slab to get to the other side. We can express this as

$$\text{thickness}(\phi) = \frac{3.0}{\cos\phi}.$$

A quick check reveals that this gives us what we want: when $\phi = 0$ the neutron is traveling straight through the slab and the thickness to that neutron is 3 cm. Also, when $\phi = \pm\pi/2$ the thickness of the slab is infinite because the neutron is traveling parallel to the slab.

We can make the math easier by defining $\mu = \cos\phi$ and noticing that $\mu \in [0, 1]$. To handle our more complicated problem we make a small change to our Monte Carlo to have each neutron have it's own angle of flight relative to the slab.

1. Create neutron with μ sampled from the uniform distribution $\mu \in [0, 1]$.
2. Sample randomly a distance to collision from the exponential distribution.
3. Check to see if the distance to collision is greater than $3/\mu$.
4. Go back to 1 until we've run ``enough'' neutrons.

The only change is that now we pick μ uniformly between 0 and 1 (recall that each angle was equally likely that is why this is a random distribution). Furthermore, we check to see if the distance to collision is greater than $3/\mu$. Those are the only changes.

The solution to this problem is more complicated to find mathematically, but the answer is expressed by the exponential integral function:

$$\text{transmission} = 2\int_0^1 \frac{d\mu}{\Sigma_t}\, e^{-\Sigma_t x/\mu} = E_2(\Sigma_t x).$$

For our case of $x = 3.0$ and $\Sigma_t = 2$ we get

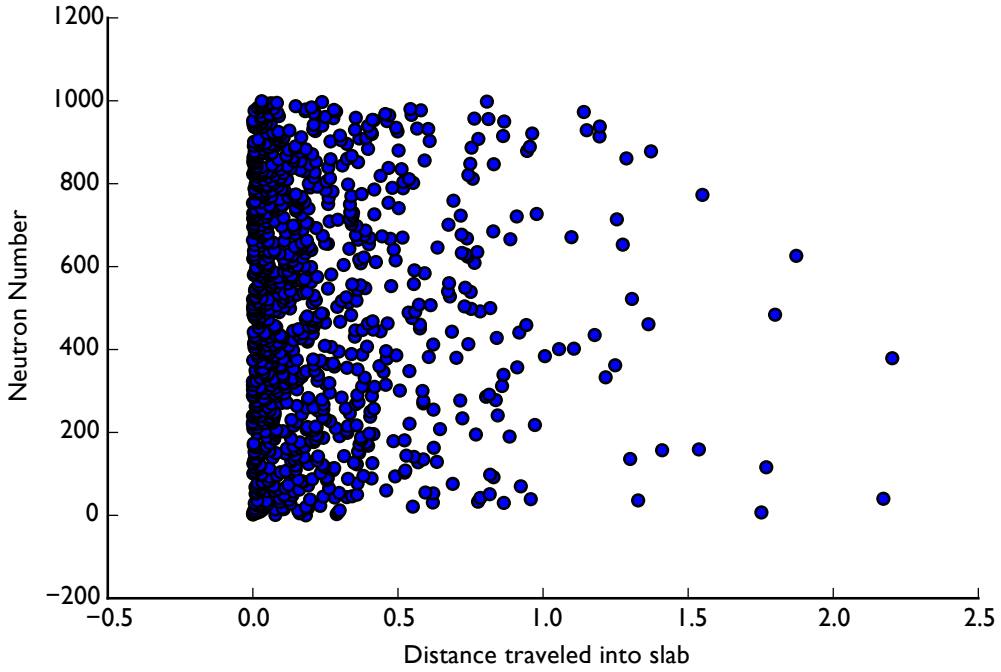
$$E_2(6) \approx 0.000318257.$$

Notice how the transmission went down because most particles will not have $\mu = 1$.

We can simply modify our function above to handle this case.

```
In [5]: def slab_transmission(Sig_t,thickness,N,isotropic=False):
            """Compute the fraction of neutrons that leak through a slab
            Inputs:
            Sig_t:     The total macroscopic x-section
            thickness: Width of the slab
            N:         Number of neutrons to simulate
            isotropic: Are the neutrons isotropic or a beam

            Returns:
            transmission:  The fraction of neutrons that made it through
            """
            if (isotropic):
                mu = np.random.random(N)
            else:
                mu = np.ones(N)
            thetas = np.random.random(N)
            x = -np.log(1-thetas)/Sig_t
            transmission = np.sum(x>thickness/mu)/N

            #for a small number of neutrons we'll output a little more
            if (N<=1000):
                plt.scatter(x*mu,np.arange(N))
                plt.xlabel("Distance traveled into slab")
                plt.ylabel("Neutron Number")
            return transmission
```

```
In [6]: ###test the function with a small number of neutrons
        Sigma_t = 2.0
        thickness = 3.0
        N = 1000
        transmission = slab_transmission(Sigma_t, thickness, N, isotropic=True)
        print("Out of",N,"neutrons only",int(transmission*N),
              "made it through.\n The fraction that made it through was",transmission)
```




```
Out of 1000 neutrons only 0 made it through.
 The fraction that made it through was 0.0
```

```
In [7]:  neuts = np.array([2000,4000,8000,16000,32000,64000,128000,256e3,512e3,1024e3,2056e3])
         for N in neuts:
             transmission = slab_transmission(Sigma_t, thickness, N, isotropic=True)
             print("Out of",N,"neutrons only",int(transmission*N),
                   "made it through.\n The fraction that made it through was",transmission)
```

```
Out of 2000.0 neutrons only 0 made it through.
 The fraction that made it through was 0.0
Out of 4000.0 neutrons only 0 made it through.
 The fraction that made it through was 0.0
Out of 8000.0 neutrons only 3 made it through.
 The fraction that made it through was 0.000375
Out of 16000.0 neutrons only 3 made it through.
 The fraction that made it through was 0.0001875
Out of 32000.0 neutrons only 11 made it through.
 The fraction that made it through was 0.00034375
Out of 64000.0 neutrons only 20 made it through.
 The fraction that made it through was 0.0003125
Out of 128000.0 neutrons only 30 made it through.
 The fraction that made it through was 0.000234375
Out of 256000.0 neutrons only 73 made it through.
 The fraction that made it through was 0.00028515625
Out of 512000.0 neutrons only 140 made it through.
 The fraction that made it through was 0.0002734375
```

```
Out of 1024000.0 neutrons only 324 made it through.
 The fraction that made it through was 0.00031640625
Out of 2056000.0 neutrons only 717 made it through.
 The fraction that made it through was 0.00034873540856
```

We do start to get to the correct answer, but since so few neutrons get through we have to simulate a lot of them. Let's try 10 million:

```
In [8]:  N = 1e7
         transmission = slab_transmission(Sigma_t, thickness, N, isotropic=True)
         print("Out of",N,"neutrons only",int(transmission*N),
               "made it through.\n The fraction that made it through was",transmission)
```

```
Out of 10000000.0 neutrons only 3164 made it through.
 The fraction that made it through was 0.0003164
```

We're getting several digits of accuracy, but it took a lot of neutrons. Of course in real life 10 million neutrons is not very many. We typically talk about neutrons in numbers like 10^{10} or greater. This is the price to pay with analog physics: the number of our pretend neutrons are always going to be smaller that the actual neutrons.

11.4 Our First Monte Carlo Shielding Calculation

Both problems that we solved above can be solved pretty easily by hand. We would like to solve a problem that is much more difficult. To make the problem more difficult we can add some scattering. We want to know what fraction of the neutrons get through the slab before being absorbed. This is a typical question in radiation shielding.

In particular let's say that the slab is made up of a material that has $\Sigma_t = 2.0$ cm^{-1}, $\Sigma_s = 0.75$ cm^{-1}, and $\Sigma_a = 1.25$ cm^{-1}. Also, let's say that the neutrons are scattered isotropically when they scatter, that is the direction can change to any other direction upon scattering. This problem cannot be solved very well by diffusion (remember diffusion is an approximation). We can solve it by modifying our procedure from before.

We'll need to add the fact that a collision can be a scatter or an absorption. We will still sample a distance to collision, the difference is that when the neutron collides, we sample whether it is absorbed or scattered based on the scattering ratio: Σ_s/Σ_t. If it scatters, we sample another μ for it and keep following it. Otherwise, we stop following the neutron because it has been absorbed.

The algorithm for this problem just builds on what we did before.

1. Create a counter, $t = 0$ to track the number of neutrons that get through.
2. Create neutron with μ sampled from the uniform distribution $\mu \in [0, 1]$. Set $x = 0$
3. Sample randomly a distance to collision, l, from the exponential distribution.
4. Move the particle to $x = x + l\mu$.
5. Check to see if $x > 3$. If so $t = t + 1$. Check if $x < 0$ if so go to 2.
6. Sample a random number s in [0,1], if $s < \Sigma_s/\Sigma_t$, the particle scatters and sample $\mu \in [-1, 1]$ and go to step 3. Otherwise, continue.
7. Go back to 2 until we've run ``enough'' neutrons.

In this case we need to check to make sure that the neutron does not scatter and exit the slab at $x = 0$. If that happens, to our mind that is the same as absorption because that neutron is not going to leak through the slab.

Our algorithm is going to have to change a lot in this case, but all we're really doing is modifying the steps of the simple algorithm.

```
In [9]:  def slab_transmission(Sig_s,Sig_a,thickness,N,isotropic=False):
             """Compute the fraction of neutrons that leak through a slab
             Inputs:
             Sig_s:     The scattering macroscopic x-section
             Sig_a:     The absorption macroscopic x-section
             thickness: Width of the slab
             N:         Number of neutrons to simulate
             isotropic: Are the neutrons isotropic or a beam

             Returns:
             transmission:  The fraction of neutrons that made it through
             """
             Sig_t = Sig_a + Sig_s
             transmission = 0.0
             N = int(N)
             for i in range(N):
                 if (isotropic):
                     mu = np.random.random(1)
                 else:
                     mu = 1.0
                 x = 0
                 alive = 1
                 while (alive):
                     #get distance to collision
                     l = -np.log(1-np.random.random(1))/Sig_t
                     #move particle
                     x += l*mu
                     #still in the slab?
                     if (x>thickness):
                         transmission += 1
                         alive = 0
                     elif (x<0):
                         alive = 0
                     else:
                         #scatter or absorb
                         if (np.random.random(1) < Sig_s/Sig_t):
                             #scatter, pick new mu
                             mu = np.random.uniform(-1,1,1)
                         else: #absorbed
                             alive = 0
             transmission /= N
             return transmission
```

As a test, this should do the same thing as the previous example if we set $\Sigma_s = 0$ and $\Sigma_a = 2$.

```
In [10]: N = 100000
         Sigma_s = 0.0
         Sigma_a = 2.0
         transmission = slab_transmission(Sigma_s,Sigma_a, thickness, N, isotropic=True)
         print("Out of",N,"neutrons only",int(transmission*N),
               "made it through.\n The fraction that made it through was",transmission)
```

```
Out of 100000 neutrons only 25 made it through.
 The fraction that made it through was 0.00026
```

That seems to be working. Now let's try it with $\Sigma_s = 0.75$ and $\Sigma_a = 1.25$. In this case we would expect the transmission rate to go up.

```
In [11]: N = 1000000
         Sigma_s = 0.75
         Sigma_a = 2.0 - Sigma_s
         transmission = slab_transmission(Sigma_s,Sigma_a, thickness, N, isotropic=True)
         print("Out of",N,"neutrons only",int(transmission*N),
              "made it through.\n The fraction that made it through was",transmission)
```

```
Out of 1000000 neutrons only 861 made it through.
 The fraction that made it through was 0.000861
```

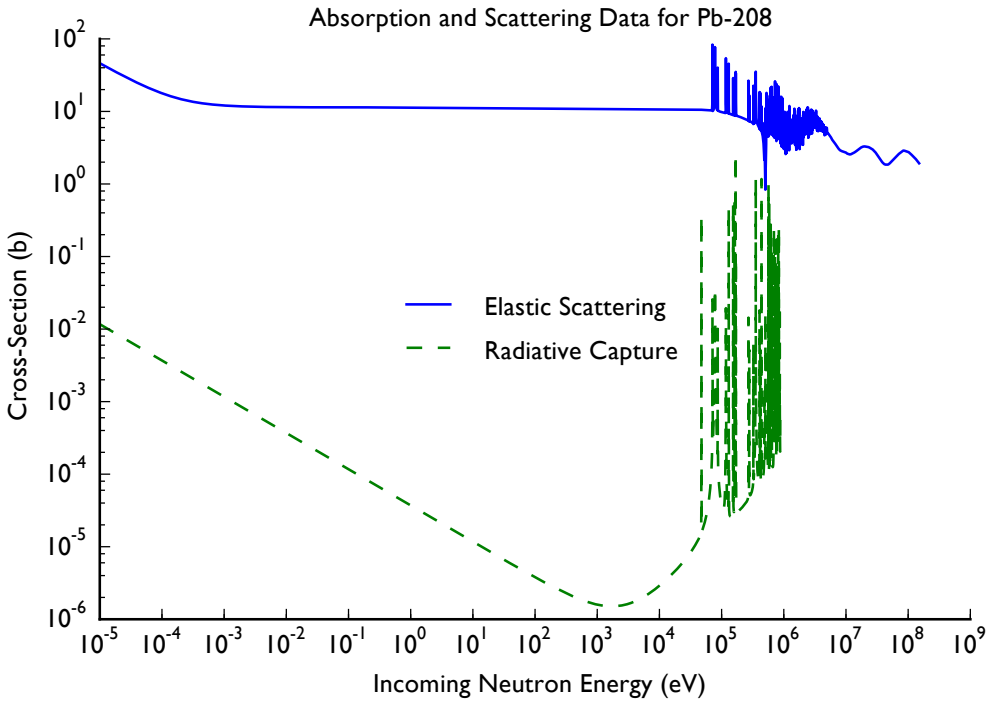
About a factor three increase. With scattering it takes much longer to do the simulation because we might have to follow each neutron for several steps because we have to follow it until it is absorbed or leaves the slab.

This is a problem where we cannot write down the answer easily. As I mentioned, diffusion cannot solve this problem accurately because it is a boundary driven problem with not that much scattering. To derive the full solution to the transport equation is beyond the scope of this class, and is covered in PhD. level courses when we talk about singular eigenfunction expansions. It is not a strech to say in this case that the Monte Carlo approach is much easier.

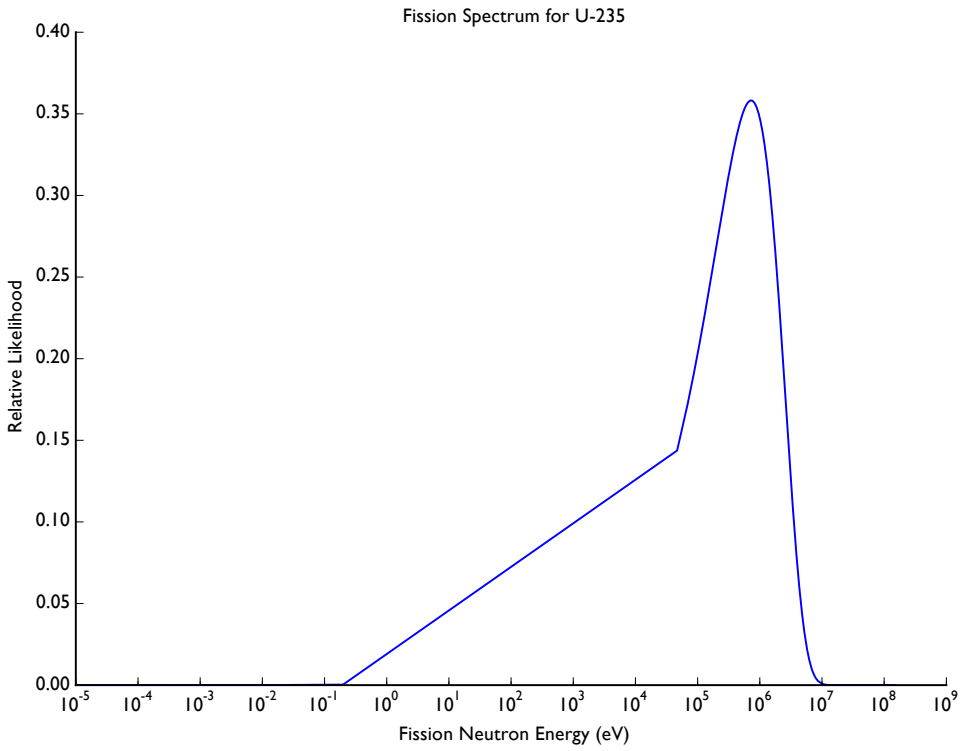
11.5 A Real Shielding Problem

We will now take a large step forward. We will try to solve the problem of designing a lead-208 shield for a bare reactor made of uranium-235. We will use the full energy dependent cross-sections for lead and the actual fission spectrum of U-235. To do this we will have to read in the data for the lead cross-sections and the fission spectrum. Luckily, we learned how to do this in Lecture 5.

```
In [12]: import csv
         lead_s = [] #create a blank list for the x-sects
         lead_s_energy = [] #create a blank list for the x-sects energies
         #this loop will only execute if the file opens
         with open('pb_scat.csv') as csvfile:
             pbScat = csv.reader(csvfile)
             for row in pbScat: #have for loop that loops over each line
                 lead_s.append(float(row[1]))
                 lead_s_energy.append(float(row[0]))
         lead_scattering = np.array([lead_s_energy,lead_s])
         lead_abs = [] #create a blank list for the x-sects
         lead_abs_energy = [] #create a blank list for the x-sects energies
         #this loop will only execute if the file opens
         with open('pb_radcap.csv') as csvfile:
             pbAbs = csv.reader(csvfile)
             for row in pbAbs: #have for loop that loops over each line
                 lead_abs.append(float(row[1]))
                 lead_abs_energy.append(float(row[0]))
         lead_absorption = np.array([lead_abs_energy,lead_abs])

         plt.loglog(lead_scattering[0,:], lead_scattering[1,:],label="Elastic Scattering")
         plt.loglog(lead_absorption[0,:], lead_absorption[1,:],'--',label="Radiative Capture")
         plt.xlabel('Incoming Neutron Energy (eV)')
         plt.ylabel('Cross-Section (b)')
         plt.title('Absorption and Scattering Data for Pb-208')
         plt.legend(loc="best")
```




```
In [13]: #experimentally derived fission spectrum
         expfiss = lambda E: 0.453*np.exp(-1.036*E/1.0e6)*np.sinh(np.sqrt(2.29*E/1.0e6))
         plt.semilogx(lead_scattering[0,:], expfiss(lead_scattering[0,:]))
         plt.xlabel('Fission Neutron Energy (eV)')
         plt.ylabel('Relative Likelihood')
         plt.title('Fission Spectrum for U-235')
```



To solve this problem we will have to change our algorithm somewhat. Firstly, we will have to generate neutrons with energies sampled from the fission spectrum. Secondly, we will have to change how we scatter particles. When a particle scatters we will have to sample a new energy for the post-scattered neutron. We will tackle each of these next.

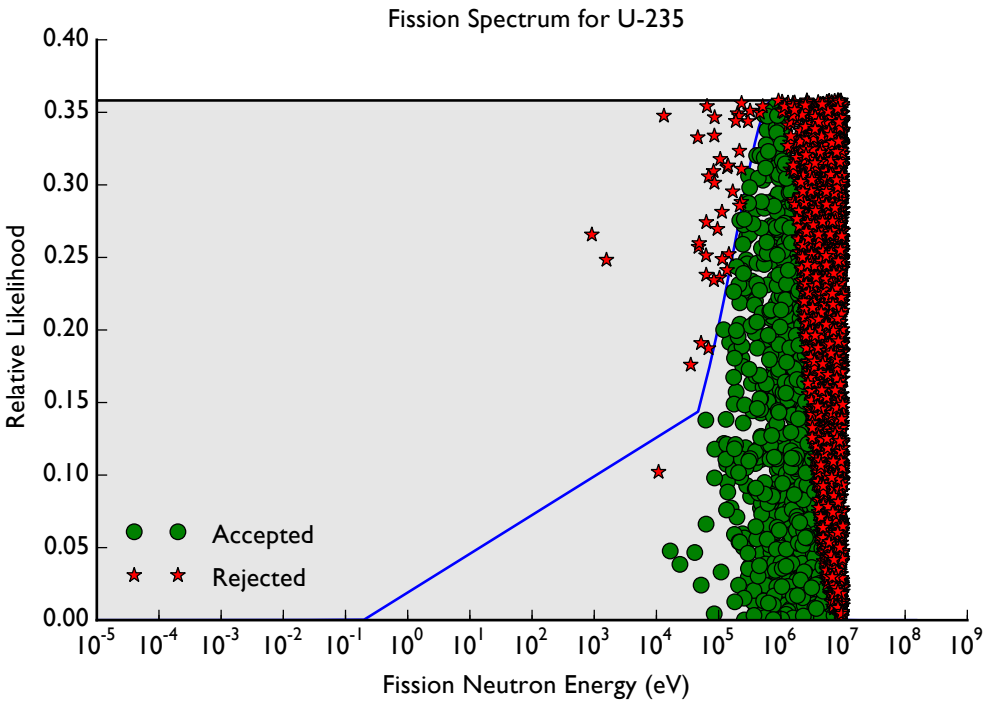
11.6 Rejection Sampling

Previously, we discussed how to sample from a distribution by inverting the CDF and then using a random number between 0 and 1 to give us the inverse CDF value corresponding to our sample. With the fission spectrum we can't do this. We don't have a CDF, we just have a distribution that gives a relative likelihood of a fission neutron being born with a certain energy. To sample from this we use rejection sampling. The idea of rejection sampling is to draw a box around the distribution we want to sample from, and pick points in the box. If the point is below the curve, we accept the point, otherwise we reject it and sample again.

We'll demonstrate the idea of rejection sampling with our fission spectrum data.

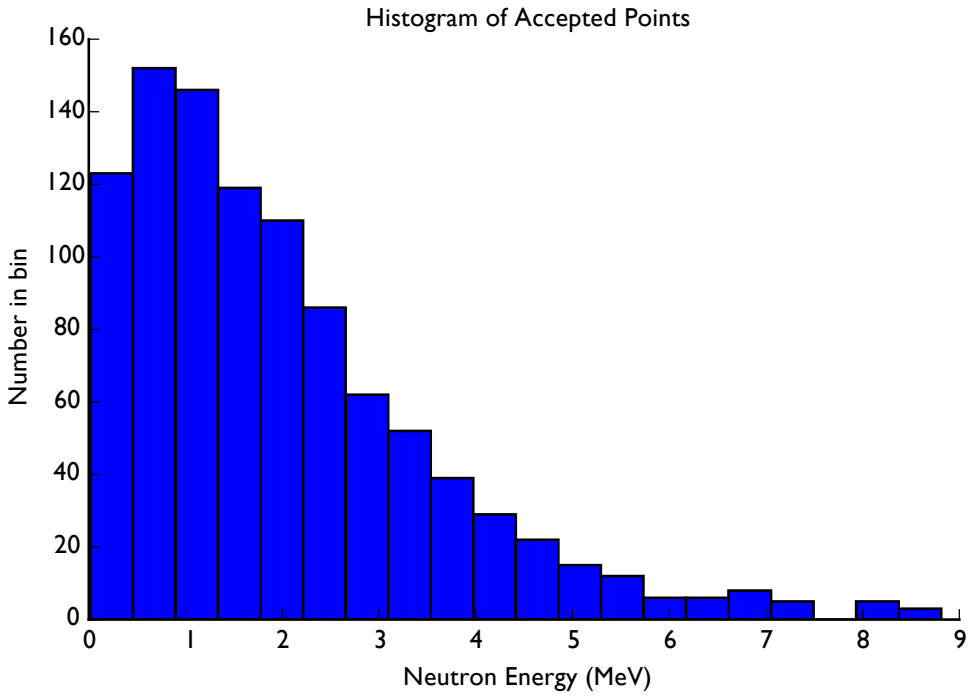
```
In [14]: #make samples
         min_eng = np.min(lead_scattering[0,:])
         max_eng = 1e7
         max_prob = np.max(np.max(expfiss(lead_scattering[0,:])))
         N = 1000
         accepted_x = []
         accepted_y = []
         rejected_x = []
         rejected_y = []
         for i in range(N):
```

```
    rejected = 1
    while (rejected):
        #pick x
        x = np.random.uniform(min_eng,max_eng,1)
        y = np.random.uniform(0,max_prob,1)
        rel_prob = expfiss(x)
        if (y <= rel_prob):
            accepted_x.append(x)
            accepted_y.append(y)
            rejected = 0
        else:
            rejected_x.append(x)
            rejected_y.append(y)
accepted = np.array([accepted_x, accepted_y])
rejected = np.array([rejected_x, rejected_y])

from matplotlib.patches import Polygon
ax = plt.subplot(111)
plt.semilogx(lead_scattering[0,:], expfiss(lead_scattering[0,:]))
plt.xlabel('Fission Neutron Energy (eV)')
plt.ylabel('Relative Likelihood')
plt.title('Fission Spectrum for U-235')
verts = [(min_eng,0),(min_eng,max_prob), (max_eng,max_prob),(max_eng,0)]
poly = Polygon(verts, facecolor='0.9', edgecolor='k')
ax.add_patch(poly)
plt.plot(accepted[0,:], accepted[1,:],'o',label='Accepted')
plt.plot(rejected[0,:], rejected[1,:],'*',label='Rejected')
plt.legend(loc=3)
```



Looking at the histogram of the accepted samples looks like the fission spectrum:

```
In [15]: plt.hist(accepted[0,:]/1.0e6, bins=20)
         plt.title("Histogram of Accepted Points")
         plt.xlabel("Neutron Energy (MeV)")
         plt.ylabel("Number in bin")
```



11.7 Looking Up Energies

To look up the cross-section for lead at different energies we need a function that can return the value of the cross-section for a given energy. We'll make this happen by using by finding the energy in the data set that is closest to the input energy. We'll do this using the NumPy function argmin.

```
In [16]: def energy_lookup(data_set, inp_energy):
             """look up energy in a data set and return the nearest energy in the table
             Input:
             data_set:   a vector of energies
             inp_energy: the energy to lookup

             Output:
             index: the index of the nearest neighbor in the table
             """
             #argmin returns the indices of the smallest members of an array
             #here we'll look for the minimum difference between the input energy and the table
             index = np.argmin(np.fabs(data_set-inp_energy))
             return index
```

```
In [17]: #let's check our implementation
         index = energy_lookup(lead_scattering[0,:],1.0e6)
         lead_scattering[:,index]

Out[17]: array([  1.00000000e+06,   4.26881000e+00])
```

11.8 Elastic Scattering

When a particle scatters elastically, the energy of the scattered neutron, E', of a neutron with an initial energy E is governed by a probability given by

$$P(E \rightarrow E') = \begin{cases} \frac{1}{1-\alpha} & \alpha E \leq E' \leq E \\ 0 & \text{otherwise} \end{cases},$$

where

$$\alpha = \frac{(A-1)^2}{(A+1)^2},$$

with A the mass of the nucleus. Therefore, we can sample the value of the scattered neutron's energy from a uniform distribution from αE to E.

11.9 Lead Shielding of Reactor Algorithm and Code

The algorithm will be an enhanced version of the one before. We will assume isotropic, elastic scattering and radiative capture as the only reactions in the lead, though this simplification could be modified. The algorithm now looks like:

1. Create neutron with μ sampled from the uniform distribution $\mu \in [0,1]$ and an energy sampled from the fission spectrum via rejection sampling. Set $x = 0$
2. Sample randomly a distance to collision, l, from the exponential distribution.
3. Move the particle to $x = x + l\mu$.
4. Check to see if x is greater than the shield thickness, if so stop following the neutron. Check if $x < 0$ if so go to 1.
5. Sample a random number s in [0,1], if $s < \Sigma_s/\Sigma_t$, the particle scatters and sample $\mu \in [-1,1]$ and an energy in based on the formula above and go to step 3. Otherwise, continue.
6. Go back to 1 until we've run ``enough'' neutrons.

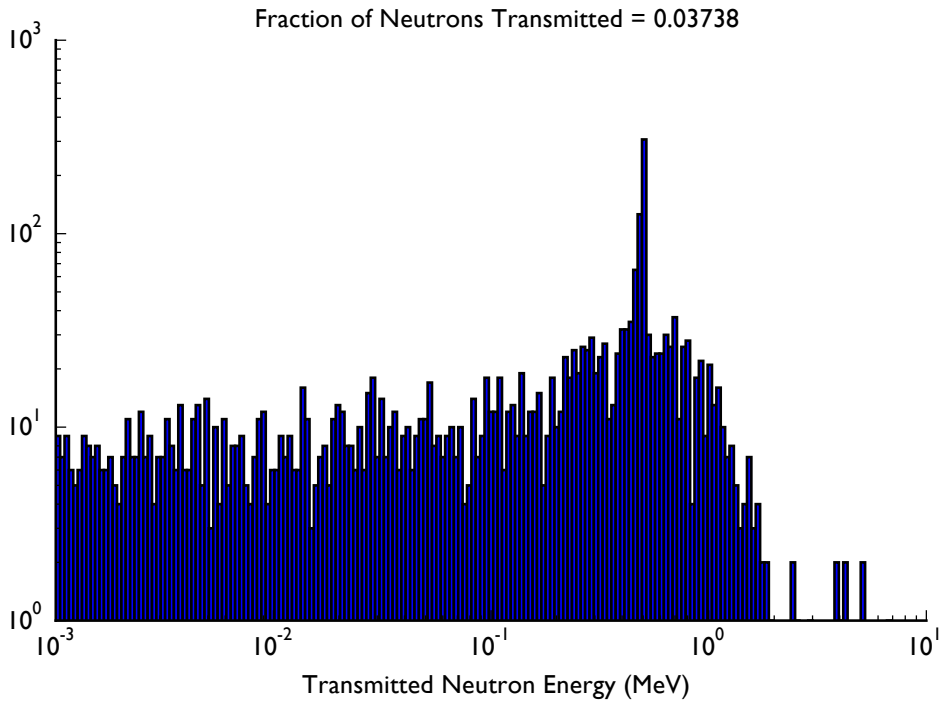
```
In [18]: def slab_reactor(Sig_s,Sig_a,thickness,density,A,N,isotropic=False):
             """Compute the fraction of neutrons that leak through a slab
             Inputs:
             Sig_s:     The scattering macroscopic x-section array in form Energy, X-sect
             Sig_a:     The absorption macroscopic x-section
             thickness: Width of the slab
             density:  density of material in atoms per cc
             A:         atomic weight of shield
             N:         Number of neutrons to simulate
             isotropic: Are the neutrons isotropic or a beam

             Returns:
             transmission:  energies of neutrons that leak through
             created:  energies of neutrons that were born
             """
             alpha = (A-1.0)**2/(A+1.0)**2
```

```
    transmission = []
    created = []
    N = int(N)
    for i in range(N):
        #sample direction
        if (isotropic):
            mu = np.random.random(1)
        else:
            mu = 1.0
        #compute energy via rejection sampling
        rejected = 1
        while (rejected):
            #pick x
            x = np.random.uniform(min_eng,max_eng,1)
            y = np.random.uniform(0,max_prob,1)
            rel_prob = expfiss(x)
            if (y <= rel_prob):
                energy = x
                rejected = 0
        #initial position is 0
        x = 0
        created.append(energy[0])
        alive = 1
        while (alive):
            #get distance to collision
            scat_index = energy_lookup(Sig_s[0,:],energy)
            abs_index = energy_lookup(Sig_a[0,:],energy)
            cur_scat = Sig_s[1,scat_index]
            cur_abs = Sig_a[1,abs_index]
            Sig_t = cur_scat + cur_abs
            l = -np.log(1-np.random.random(1))/Sig_t
            #move particle
            x += l*mu
            #still in the slab
            if (x>thickness):
                transmission.append(energy[0])
                alive = 0
            elif (x<0):
                alive = 0
            else:
                #scatter or absorb
                if (np.random.random(1) < cur_scat/Sig_t):
                    #scatter, pick new mu and energy
                    mu = np.random.uniform(-1,1,1)
                    energy = np.random.uniform(alpha*energy,energy,1)
                else: #absorbed
                    alive = 0
    return transmission, created
```

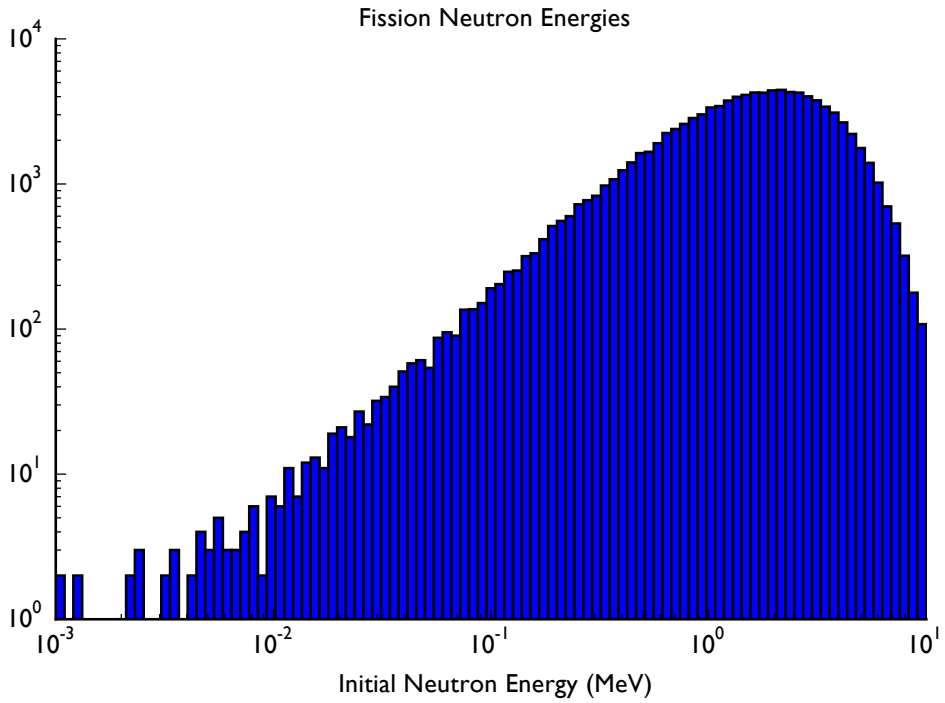
```
In [19]: N = 1e5
         density = 11.34/208*6.022e23
         thickness = 5
         transmission,created = slab_reactor(
```

```
        lead_scattering,lead_absorption, thickness,density,208, N, isotropic=True)
```

In [20]:
```
plt.hist(np.array(transmission)/1.e6,bins = 10 ** np.linspace(np.log10(0.001), np.log10(10), 200))
plt.gca().set_xscale("log")
plt.gca().set_yscale("log")
plt.xlabel("Transmitted Neutron Energy (MeV)")
plt.title("Fraction of Neutrons Transmitted = " + str(len(transmission)/N))
```



In [21]:
```
plt.hist(np.array(created)/1.e6,bins = 10 ** np.linspace(np.log10(0.001), np.log10(10), 100))
plt.gca().set_xscale("log")
plt.gca().set_yscale("log")
plt.xlabel("Initial Neutron Energy (MeV)")
plt.title("Fission Neutron Energies")
```



Coda

We have demonstrated that we can solve complicated problems by ``rolling dice'' if we roll many, many dice and move particles around based on these random numbers. One feature of this approach is that it requires little in the way of mathematical sophistication, with the tradeoff that the convergence is slow (remember that the noise in the solution decays as the number of samples to the negative one-half power). Nevertheless, Monte Carlo methods are attractive and they are widely used in nuclear engineering and other fields.

Short Exercises

For each problem write a python code that accomplishes the task. You may use any of the algorithms presented in this chapter in your solutions.

1. The Maxwell-Boltzman distribution, often called just a Maxwellian distribution, gives the distribution of speeds of particles in a gas by the formula

$$f(v) = \sqrt{\frac{m}{2\pi kT}} \exp\left[\frac{-mv^2}{2kT}\right],$$

where m is the mass of the particles, T is the temperature, and k is the Boltzmann constant. Consider a gas of deuterium at $kT = 1$ keV $= 1.60218 \times 10^{-16}$ J. Sample particle speeds from the Maxwellian using rejection sampling. From your sampled points, compute the mean speed and the square-root of the mean speed squared (i.e., compute the mean value of the speed squared and then take the square root, aka the root-mean square speed). The mean speed should be

$$\int_0^\infty dv\, v f(v) = \sqrt{\frac{8kT}{\pi m}},$$

and the root-mean square speed should be

$$\sqrt{\int_0^\infty dv\, v^2 f(v)} = \sqrt{\frac{3kT}{m}}.$$

Compute these quantities using sample numbers of $N = 10, 10^2, 10^4, 10^6$ and discuss your results.

2. Modify the shielding code to consider neutrons of a single energy impinging on the shield and to tally the energy of the absorbed neutrons. Assume the neutrons are all 2.5 MeV and are produced from the fusion of deuterium. Plot the distribution of transmitted and absorbed neutrons with a large enough number of sample particles.