Assignment 8: Tracking Neutrons in a Hollow Spherical Shell

Overview

A Monte Carlo Simulation (MCS) uses random sampling to solve complex problems. In fact, MCS is used to model the probability of different outcomes to understand the impact of risk and uncertainty in forecasting models. To make the uncertainty in our solution small, we need to average over a very large number of simulations, which is computationally expensive. However, the nice thing about the Monte Carlo is that you don't need to know a lot about mathematics. All you need to do is push particles around.

In this Assignment, we will track the neutrons transiting a hollow spherical shell. For the nuclear engineers this could be neutrons transiting a shield meant to protect you from them, or a fissionable material meant to produce power. For the aeronautical and mechanical engineers this could be a neutron source used for nondestructive testing. In the manufacturing of a safety-related structural part (e.g., casting a wing strut) it is possible to accidentally introduce voids (air bubbles) in the part which will invisibly reduce the strength of the part. Shining radiation through the part can reveal such voids by detecting a greater than expected transmission of neutrons, indicating that there is less material there absorbing some fraction of the neutrons.

Repeatedly running MCS with some number of neutrons doesn't give the same result since the standard deviation for each random run scales as (neutrons)$^{-1/2}$. Therefore, in order to reduce the uncertainty by half we need to increase the number of neutrons we simulate by a factor of four – computationally expensive, but simpler calculations for an overall savings.

Introduction:

Consider a hollow spherical shell of inner radius, R_i, and outer radius, R_o, around an isotropic source of neutrons. Assume that neutrons inside the inner radius are essentially in a vacuum and will not interact with any air molecules there: the neutrons will travel in straight lines until they enter the spherical shell material at R_i. Neutrons outside the outer radius are also in air (vacuum) and will not scatter back into the sphere. These neutrons could be biologically harmful so we want to know how many leave the sphere. Neutrons in the solid material of the sphere will interact with the atoms there either by the atom absorbing the neutron and stopping its existence as a free neutron, or by isotropically scattering the neutron to continue its journey in a different direction.

.

Our objective is to compute the fraction of source neutrons that escape through the outer radius. For this purpose, you need to create three different module files that should be defined as follow:

Part 1: Tracking neutrons:

Create a module file named **shell_transmission.py.** In this file, you need to define five functions:

1) **random_trajectory** which returns a random 3D trajectory vector,
2) **random_distance_to_collision** which computes a random path length to interaction,
3) **DxDyDz** which returns changes in coordinates based on path length and trajectory,
4) **collision_event** which determines whether a collision was an absorption or scatter event, and
5) **shell_transmission** which simulate the random walk of neutron through a sphere and returns the number of neutrons transmitted.

These functions need to be defined as follows.

def random_trajectory():

- When a neutron leaves the source or scatters off of an atom in the shell it proceeds in a random direction. This direction is shown in the figure below as the vector Ω, which is defined by the angles θ, the "polar angle", and φ, the "azimuthal angle". (Some references reverse the symbols for these angles so take care!) The direction is isotropic, meaning that all trajectories are equally possible.
- Neutrons leaving the source at (0.0, 0.0, 0.0) travel in direction Ω until they encounter the inner radius of the shell at, R_i. Neutrons that have scattered off of an atom in the shell travel in direction Ω through the shell material.

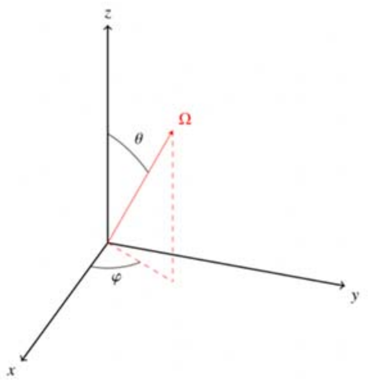


Figure 1: a random trajectory Ω. The domain of θ is $[0, \pi]$ and the domain of φ is $[0, 2\pi]$.

- This function returns random scalar values of θ (theta) and φ (phi) in radians, in that order.
- Compute random values of θ and φ by generating uniformly distributed random numbers on the interval [0, 1) and scaling them appropriately.

def random_distance_to_collision(Sig_t):

- When the neutron is in the sphere material it will travel some random distance s before interacting with an atom. The probability *p(s)* that a neutron travels a distance *s* through the material is exponentially distributed as $p(s) = e^{-\Sigma_t s}$ where Σ_t is the total probability of interaction per unit length of travel: many neutrons interact within a short distance, and a few neutrons can go very long distances before interacting according to this decaying exponential function. We need to sample random distances *s* according to this distribution to follow neutrons around from one collision to the next.
- Sig_t (Σ_t) is the total probability of interaction per unit path length traversed and is the sum of the probabilities of scatter and absorption: $\Sigma_t = \Sigma_s + \Sigma_a$.
- This function returns a random scalar path length to interaction, *s*.
- *s* can be sampled from the exponential distribution by computing $s = -\ln(1 - x) / \Sigma_t$ where x is a uniform random number in the range [0, 1).
- In effect, the probability p(s) is sampled by a random number *x* between 0.0 (0% probability) and 1.0 (100% probability). Given a probability *x*, the path length can be solved for as $s = -\ln(x) / \Sigma_t$. However, typical random number generators produce values in the interval [0, 1) that are inclusive of 0. This means that there is a chance that ln(x) could raise an exception since ln(0) is

undefined. A uniformly distributed random number x in [0, 1) is indistinguishable from a random number $1-x$ in (0, 1].

- The function could still fail if Σ_t is zero and is meaningless if Σ_t is negative. Your function must appropriately test input parameter Σ_t and raise a ValueError with a useful error message if it is not positive.

def DxDyDz(s, theta, phi)

- A neutron that travels a distance of magnitude s in direction Ω will change its Cartesian (x, y, z) coordinates by

$$\Delta x = s\ sin\theta\ cos\varphi$$
$$\Delta y = s\ sin\theta\ sin\varphi$$
$$\Delta z = s\ cos\theta$$

- This function returns a 1D numpy array with three elements in the order Δx, Δy, and Δz.

def collision_event(Sig_s, Sig_a):

- This function determines whether the collision was an absorption or scatter event. If the neutron is absorbed that neutron's journey ends and a new neutron is tracked from the source. If the neutron is scattered it continues its random journey to the next collision event.
- Sig_s (Σ_s) measures the probability that the neutron scatters, while Sig_a (Σ_a) measures the probability that the neutron is absorbed. Since these are the only two outcomes for a neutron collision in this material the fraction of neutrons that are scattered is Σ_s / (Σ_s + Σ_a) = Σ_s / Σ_t. The remaining fraction is the fraction absorbed.
- This function returns a bool that is True if the neutron was scattered and will continue its journey and returns False if the neutron was absorbed.

def shell_transmission(Sig_s, Sig_a, Ri, Ro, N)

- This function uses the above four functions to simulate the random walk of **N** neutrons in the sphere. This function returns a 1D numpy integer array of size N where each element n is the total number of neutrons transmitted past R_o after simulating n neutrons.
- As above, Sig_s (Σ_s) is the scattering probability per unit path length and Sig_a (Σ_a) is the absorption probability per unit path length.
- R_i and R_o are the inner and outer radius of the shell, respectively.
- N is the integer number of neutrons to simulate.

For each neutron simulated:

- Start the neutron at Cartesian coordinates (0.0, 0.0, 0.0).
- Isotropically transport the neutron to the inner face of the shell at R_i. (Hint: the neutron will travel a distance R_i from the origin to the inner face.)
- Each time the neutron moves the Cartesian coordinates of the neutron (x, y, z) should be updated.
- From the inner face of the shell, transport the neutron in the shell for as long as the neutron continues to scatter around in the shell. Each neutron transport to new coordinates can result in one of three occurrences:
  1) The coordinates locate the neutron outside the outer radius of the sphere, R_o, meaning the neutron has transmitted through the sphere, the transmission array needs to be incremented by one in elements n through to the end of the array. A new neutron is then started from the origin up to the N neutrons being simulated.
  2) The neutron stays in the sphere material and either A) scatters and continues its journey with a new random trajectory and random path length, or B) is absorbed and ends its journey and a new neutron is started from the origin up to the N neutrons being simulated.
  3) The coordinates locate the neutron inside the inner radius in the hollow void. This neutron has nothing to collide with and so will continue its journey along the same trajectory until it strikes the inner radius again at the other side of the shell. To understand the coordinates where the neutron reenters the shell on the other side of the central void we need to find the value of *s* such that

$$(x_i + s\ sin\theta\ cos\varphi)^2 + (y_i + s\ sin\theta\ sin\varphi)^2 + (z_i + s\ cos\theta)^2 = R_i{}^2$$

     where (x_i, y_i, z_i) is the location of the neutron in Cartesian coordinates. This quadratic equation can be cast as a residual function f(s) = 0 to find the unique root *s* that will move the neutron to the other side of the interior of the shell. Use a lambda function to define this residual function and use your bisection method with `xtol = 1.0e-10` and appropriate lower and upper bounds that bracket the root to compute s. Then move the neutron this distance *s* unimpeded along its trajectory through the central void to the other side of the inside of the shell. The neutron then continues its journey from the inner face with a new random direction and path length.

Part 2: Bisection to solve quadratic equation:

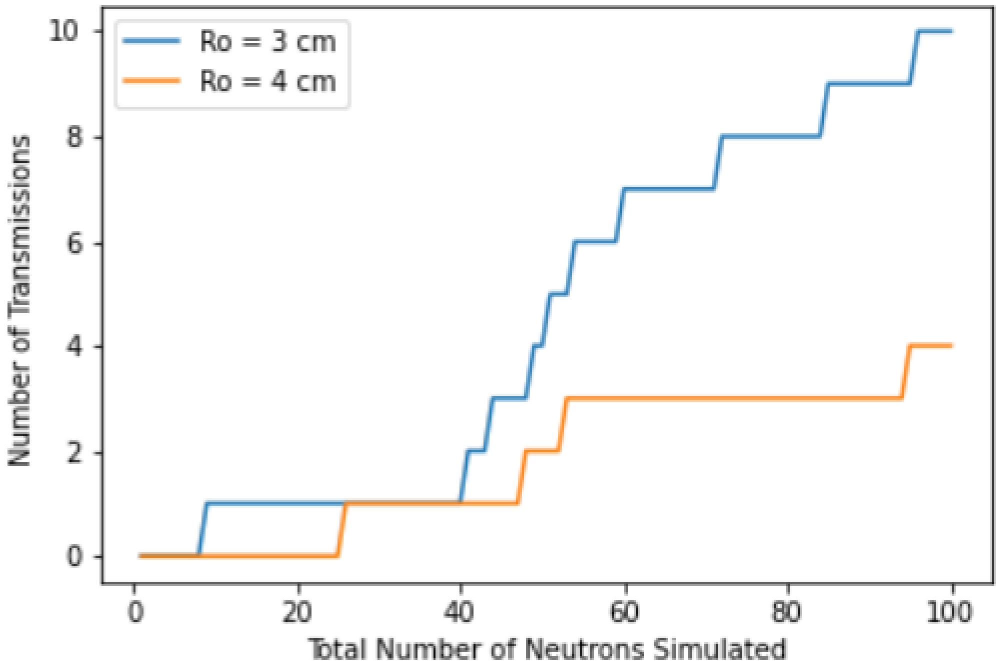
Create a file named **find_root.py**. In this file you need to write a function named **bisection**. You already have worked on this code as an in-class activity in Lesson 9.

def bisection(func, x_lo, x_hi, xtol=1e-7, ftol=1e-12, maxiter=100):

This function returns the root of func(x) = 0 within the range of [x_lo, x_hi]. The given x value must satisfy func(x_lo)*func(x_hi)<0.

Part 3: Outcomes

- Create a file named **hw08.py**.
- Simulate the transmission of **100 neutrons** through a shell of **thickness 1 cm** and **inner radius 2cm** while the **scattering and absorption probability** are each set to **1.0**.
- The number of neutrons that are transmitted should be shown on the console with a useful print statement.
- Plot the number of transmissions versus the total number of neutrons simulated.
- Change the shell thickness to **2 cm** and show the results on the same plot.
- Your results will look something like the following. Try rerunning your code to get a feel for how randomly variable the results can be.



Out of 100 neutrons, 10 made it through with outer radius of 3 cm.
Out of 100 neutrons, 4 made it through with outer radius of 4 cm.

If you wish further details on **lambda** functions you may refer to the following source: https://realpython.com/python-lambda/