Monte Carlo Techniques – Penetration of Neutrons through shielding

Tomasz Neska
10294857

School of Physics and Astronomy
The University of Manchester

Second Year Computational Physics Report

May 2020

Abstract

The analysis of the behaviour of a neutron penetrating through shielding has been conducted. This was done by modelling the neutron as a random walk with the utilization of Monte Carlo methods. The materials that were tested included water, lead and graphite. The number densities used are $3.343\ \times 10^{22}\ cm^{-3}$, $6.828 \times 10^{46}\ cm^{-3}$ and $1.985 \times 10^{46}\ cm^{-3}$ respectively. The attenuation lengths for the materials were found to be $1.90 \pm 0.03\ cm$, , $45 \pm 2\ cm$, $3.27 \pm 0.33\ cm$ respectively.

1. Introduction

Monte Carlo techniques encompass a broad range of computational algorithms. These range from integration to weather forecast simulations. They excel in processes which can't be easily solved analytically. The main idea behind those methods involves utilising random sampling to obtain an average which gives insight into the innerworkings of a physical system.[1]

One of the most common applications of Monte Carlo techniques is in random walks. A random walk is a stochastic process. It describes a journey of a particle which travels in randomly chosen directions. It is used to model behaviour of a particle in a liquid or a neutron passing through shielding. Monte Carlo techniques can be used to model the random behaviour of a particle. This is particularly useful in nuclear reactors as the geometry of the shielding can be quite intricate. One of the biggest advantages of Monte Carlo analysis is its simplicity and the ability to obtain results even from highly complex systems. [2]

2. Theory

2.1 Inverse CFD method

The Monte Carlo method is best demonstrated by the example of an integrator. Let there be a sinusoidal curve with a very high frequency of oscillation within the range $0 \leq x \ \leq 2\pi$. If the frequency of the curve is very high, then the step size must be very small to give a reliable result if the integration is done numerically. This can be done using the trapezium-rule or any fixed step-size method. This is evident by the Nyquist theorem. However, if we choose a step size randomly then the points statistically average out to give us the answer over many runs of the integrator. This is the spirit of the Monte Carlo method. It allows us to approach a stochastic process and obtain reliable results over multiple runs of the simulation. This investigation will utilize the random number generator with the help of a probability function to obtain a set of vectors. These vectors will have lengths which are exponentially distributed while the vectors are aimed in random directions. This allows the particle to be tracked. Its interactions with the material are then recorded. Then this single simulation is conducted numerous times until the average is obtained to a gratifying degree of accuracy. [3]

The exponential distribution with the probability generating function is

$$P(x) = e^{-\frac{x}{\lambda}}. \qquad \text{Equation 1}$$

In the context of particles penetrating materials this is known as the Beer-Lambert Law [4]. The variable x indicates the depth of the particle within the material while λ is the mean free path [5][6]. The mean free path is defined as the average distance travelled between successive impacts. This probability function can be derived using

$$R = -\frac{dI}{dx} = nI\sigma. \qquad \text{Equation 2}$$

This relates the rate of absorption per unit thickness, R, with the intensity of the incoming radiation, I. This then gives us that the mean free path is

$$\lambda = n\sigma\ . \qquad \text{Equation 3}$$

The number, n, is the number density and σ is the microscopic cross section. The cumulative probability function (CDF) of Equation1 is

$$CDF(x) = \int_o^x P(x)\, dx = 1 - \ e^{-\frac{x}{\lambda}}. \qquad \text{Equation 4}$$

This allows us to obtain the inverse of the CDF.

$$CDF^{-1}(z) = -\ln(1-z) = -\ln(z). \qquad \text{Equation 5}$$

If a set of random number is inputted into Equation 5 then the outcoming set will be distributed according to the exponential distribution. This equation allows for the generation of steps during the random walk. The change of variable from $1-z$ to z is done due to the numbers being generated being within 0 and 1. Hence the range restriction allows for the variable change. A step length is obtained by multiplying the distributed number by the mean free path.

2.2 Random number generation

The generation of random numbers is essential to the process. It is important to generate a set of equally distributed random numbers to ensure true randomness and a lack of a bias. Otherwise the bias would skew the results of the simulation. The method used in this simulation is obtained from the NumPy library and was tested against other random number generators in Section 3.[7]

It is essential to mention that a pseudo-random generator was also tested to show the inherit flaws in utilizing it. A pseudo-random generator utilizes a modulus function which distributed the numbers equally around a certain range. However, it isn't random as knowing the first number in the sequence generated allows for the discovery of the whole sequence following it. The generation occurs using

$$x_{n+1} = (ax_n + c)\ mod\ m. \qquad \text{Equation 6}$$
$$m < 0; 0 < c < m; 0 \leq a < m;$$
$$0 \leq x_0 < m$$

The variable m being the modulus, c being the increment, a being the multiplier, and x_0 being the seed. This formula allows for a generation of random numbers if c and m are prime. This specific pseudo generator is a Linear Congruential Generator (LCG).[8]

2.3 Neutrons and shielding

The purpose of this experiment is to simulate the transmission of neutrons through shielding and calculate the specific attenuation lengths of water, graphite, and lead. The particles that enter the material are distributed according to Equation 1. When a neutron hits the surface of a material it has a chance to either become absorbed or scatter. The probability of each depends on the multiple of the number density with the microscopic absorption/scattering area. The length at which the intensity of particles drops to $\frac{1}{e}$ is defined as the attenuation length.

3. Method

The process that is being investigated is being modelled as a Markov Chain [9]. The particle is set on the surface of the material with the first step being made in the direction perpendicular to the surface. The length is obtained from Equation 5. The particle moves the step and then generates a probability. According to this probability it is either absorbed or it continues after scattering. This scattering allows it to make another step. This cycle proceeds until the particle is either absorbed, transmitted to the other end, or it reflects to the source. The number of each of those events is counted. Then the thickness of the material is changed, and the simulation is started again. This is done until the numbers of particles at specific thickness agree between runs within a suitable degree of error (See Section 4). Those numbers are then plotted and according to Equation 1 the attenuation length is found.

The object-oriented approach has been implemented due to the clear separation between random number generation and the experimental methods. This allows for the reusability of code along with the modular application of other classes like the LCG class. This also allows the experiments to be self-contained without the need for global variables. The inheritance of classes was used to ensure legibility in the code. The code doesn't possess a user interface due to its redundancy.

Material name	Water	Lead	Graphite
Absorption area/ $barn$	0.6652	0.158	0.0045
Scattering area/ $barn$	103	11.221	4.74
Density/ gcm^{-1}	1	11.35	1.67
Number density/ cm^{-1}	3.34×10^{22}	1.34×10^{23}	3.296×10^{22}
Macroscopic absorption cross section area/ cm^2	0.0222	0.0212	0.00015
Scattering cross section/ cm^2	3.44×10^{-10}	1.50	0.156
Total mean free path/ cm	45.0	0.656	6.395

Figure 1 – The table summarising the scattering constants used.

The first part of the simulation calculates value for the attenuation length of water for thickness of $10\,cm$. This is done to check the capabilities of the simulation as the theoretical value is $\sim 45\,cm$. The second part deals with the complete model which includes the scattering of neutrons. It then measures the attenuation lengths for the three respective materials by plotting the intensities of transmission against the thickness.

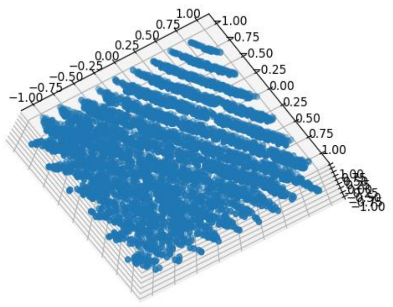
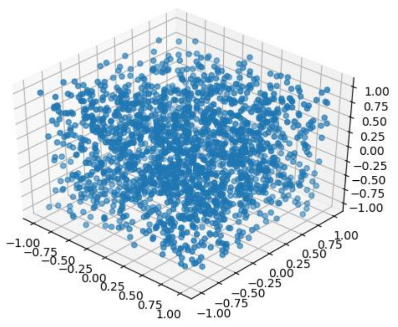



Figure 2 – The random distribution generated using a pseudo-random generator (left) and a uniform random number generation (right). The planes visible on the left show the correlation between the generated numbers while the plot on the right shows no correlation.

Testing was conducted to ensure the uniformity of the generated numbers as visible in Figure 2. The method with a uniform distribution was chosen. As a further test of the generator a sphere with uniform distribution of points was obtained by a use of a transformation [10]. This was done to ensure that the random walk moved in isotropic steps. The sphere is visible in Figure 3 (left). The method utilised to produce the sphere was then utilized to produce a set of isotropic vectors with lengths distributed according to the exponential distribution. This along with the iterative implementation allows for the random walk to be simulated.

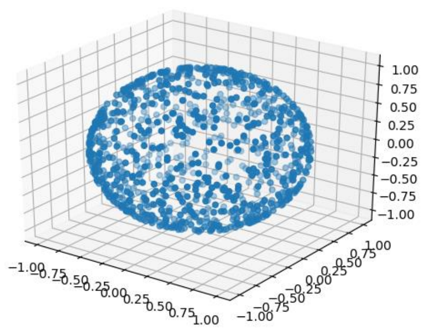
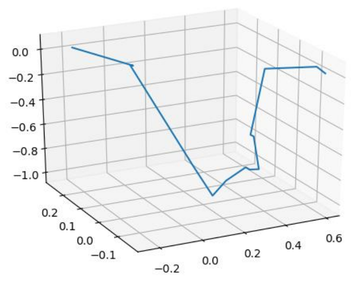
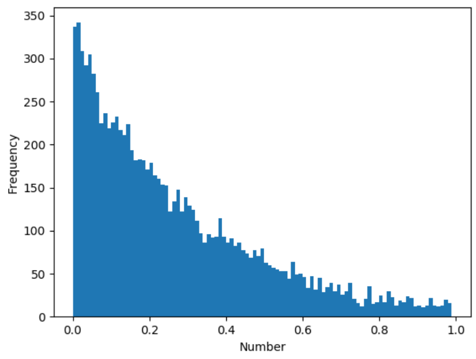



Figure 3 – An example or a random walk (right) and uniformly distributed points on a surface of a sphere (left).

The last part of the implementation introduced the Woodcock method which allows for the simulation of a particle in a system with two materials touching. [11]

Material	Water	Graphite	Lead	Water and Lead
Total neutrons	5000	5000	5000	1000
No. of reflected neutrons/ number (%)	$3995.4\ \pm 0.62\,\%$	$2429.4 \pm 0.6\,\%$	$3616.8 \pm 0.5\%$	$0.74 \pm 0.1\%$
No. of transmitted neutrons	$19.4\ \pm 0.09\,\%$	$2557.4 \pm 0.6\,\%$	$123.1 \pm 0.2\,\%$	0
No. of absorbed neutrons	$985.2\ \pm 0.7\,\%$	$13.2\ \ \pm 0.07\%$	$1260.1 \pm 0.2\%$	$999.26 \pm 0.01\%$

Figure 4 – The results of a simulation with 5000 neutrons attempting to penetrate a barrier 10 cm thick. The percentage errors were given for each number. The rightmost column includes results for the Woodcock method with thickness being 5 cm for both materials.



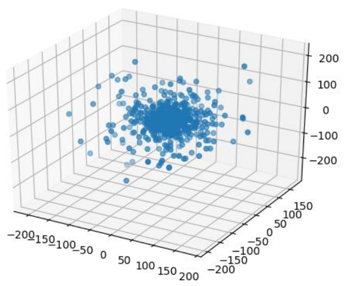


Figure 5 – Random numbers distributed according to the exponential distribution (left) and exponentially distributed steps (right). The mean free path used in calculation is $45\ cm$.

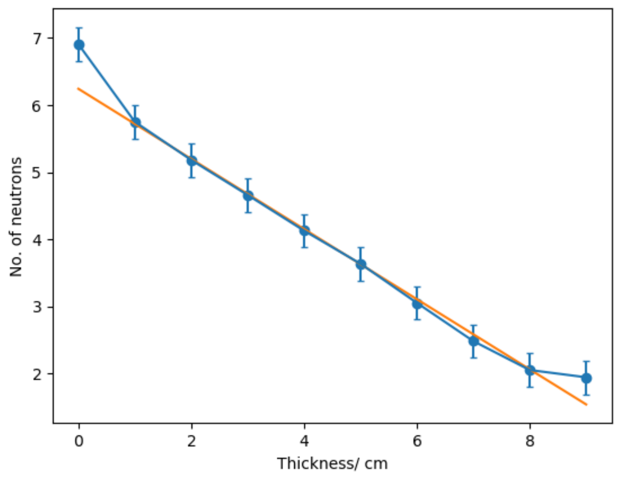


Figure 6 – The linear fit of the intensity of neutrons against thickness. The error bars are estimated using data spread as described in Section 4.

4. Analysis and Results

The first simulation measures the attenuation length for a sample of thickness $10\ cm$ composed of water. The length obtained was $42.55 \pm 14.47\ cm$.This is less than one standard deviation within the predicted value. The number of neutrons used allowed for an increase in accuracy but gave diminishing returns as expected from Monte Carlo methods. This value was obtained with only absorption being considered. The error decreased nearly exponentially with the number of neutrons used. For a sample of ten the error was 66% while for a sample of hundred the error was 15%. It requires approximately ten times more neutrons for the accuracy to increase two-fold. However, this increase flattens out at approximately 1000 neutrons. The error then is $\sim 1\%$.

Figure 4 shows the numbers of neutrons obtained at 10 cm of thickness. This was repeated 5 times and the numbers were averaged. The standard deviation gave the error in each value. The rightmost column gives the result for the Woodcock method with the first material being water and the second being lead. Both materials had were 10cm thick.

Next the simulation would be run across a range of values of thickness. The range was chosen for each material as indicated by the preliminary runs. This was due to the difference in mean free path. The attenuation lengths obtained are $1.90 \pm 0.03\ cm$, $3.27 \pm 0.33\ cm$, and $45 \pm 2\ cm$ for water, lead and graphite respectively.

The large macroscopic cross-section of neutrons in water is due to the large amount of hydrogen. The number of neutrons that reflect is very high for all materials. This is quite important when the materials are utilised in shielding as excessive absorption will heat the material.

The last part of the experiment was the utilization of the Woodcock method to calculate the number of neutrons that pass across a barrier made from water and lead. The results are visible in Figure 3.

5. Error Analysis

The main sources of error in this simulation arose from the truncation error and random fluctuation error. The errors caused by random fluctuations are inherit to the Monte Carlo methods as random number generation is necessary. This was controlled by running each simulation multiple times and finding an average. The key to minimizing errors was the organization of the routines. The routine responsible for running a single experiment at a specified thickness was ran 5 times. This allowed to verify the error on the tally of neutrons that were absorbed or transmitted. The standard deviation between those values gave the error. The main routine that obtained the thickness against number plot used those errors in the linear fitting of $\log(N)$ against thickness T. This didn't reduce the fluctuation but gave a more accurate value of the error on the values obtained. Hence this gave the more accurate value of the attenuation length.

It is important to mention that the 7 decimal place accuracy of Python allows for the numbers between 0 and 1 to be generated and then multiplied by any range necessary but this can cause issues if the range is higher than 10^7. This will cause groupings to occur as the resolution of the generator is smaller than the range over which it generates. Therefore, the truncation error can cause large discrepancies with random number simulations. This was avoided by generating in a larger range but by also not using a generator with an accuracy at least 10^2 larger than the range.

6. Conclusion

The investigation into Monte Carlo methods can be classified as successful. The algorithmic generation of an isometric random walk allowed for the simulation of a neutron inside a barrier. This gave the ability to analyse the capabilities of different materials as shielding. For the three materials tested the attenuation lengths were obtained. These are $1.90 \pm 0.03\ cm$, $3.27 \pm 0.33\ cm$, $45 \pm 2\ cm$ for water, lead and graphite respectively.

7. References

[1] SOBOL, I., 2017. *PRIMER FOR THE MONTE CARLO METHOD*. [Place of publication not identified]: CRC Press.

[2] Bronglinghaus, M., 2020. *Mean Free Path*. [online] Web.archive.org. Available at: <https://web.archive.org/web/20111105115303/http://www.euronuclear.org/info/encyclopedia/m/mean-fee-path.htm> [Accessed 14 May 2020].

[3] Grzelak, L., Witteveen, J., Suarez-Taboada, M. and Oosterlee, C., 2014. The Stochastic Collocation Monte Carlo Sampler: Highly Efficient Sampling from 'Expensive' Distributions. *SSRN Electronic Journal*,.

[4] Swinehart, D., 2020. The Beer-Lamber Law. *University of Oregon*, 39(7), pp.333-335.

[5] PROSCHAN, M., 2019. *ESSENTIALS OF PROBABILITY THEORY FOR STATISTICIANS*. [S.l.]: CHAPMAN & HALL CRC.

[6] Kennard, E., 1938. *Kinetic Theory Of Gasses*. New York: McGraw-Hill.

[7] Khan Academy. 2020. *Pseudorandom Number Generators (Video) | Khan Academy*. [online] Available at: <https://www.khanacademy.org/computing/computer-science/cryptography/crypt/v/random-vs-pseudorandom-number-generators> [Accessed 14 May 2020].

[8] Bolte, J., 2020. *Linear Congruential Generators - Wolfram Demonstrations Project*. [online] Demonstrations.wolfram.com. Available at: <https://demonstrations.wolfram.com/LinearCongruentialGenerators/> [Accessed 14 May 2020].

[9] Karlin, S. and Taylor, H., 2014. *A First Course In Stochastic Processes*. Saint Louis: Elsevier Science.

[10] Pegg Jr, E., 2011. *Random Points On A Sphere - Wolfram Demonstrations Project*. [online] Demonstrations.wolfram.com. Available at: <https://www.demonstrations.wolfram.com/RandomPointsOnASphere/> [Accessed 14 May 2020].

[11] Woodock, E., Murphy, T., Hemmings, P. and Longworth, T., 2020. *Techniques Used In The GEM Code For Monte Carlo Neutronics Calculations In Reactors And Other Systems Of Complex Geometry*. United Kingdom Atomic Energy Authority.

[12] Neska, T. (2020). *SplitSky/Scientific_Programming*. [online] GitHub. Available at: https://github.com/SplitSky/Scientific_Programming [Accessed 20 April. 2020].

8. Appendix - Code

The history of changes can be found in the GitHub repository [12].

```
# -*- coding: utf-8 -*-
"""
Author: Tomasz Neska
Date: 13/05/2020
Description: Project 3 - Monte Carlo techniques - Penetration of neutrons through shielding
"""
# initialisation
from math import *
import numpy as np
import matplotlib.pyplot as plt
import random
import time
import cmath
import json
from scipy import optimize
from mpl_toolkits.mplot3d import Axes3D
from numba import jit

plt.rcParams.update({'font.size': 14})
plt.style.use('default')
figure = plt.figure()
plt.rcParams.update({'errorbar.capsize': 2})


def linearFit(x, flux, meanFreePath, plot, ey):
    ey = np.array(ey)
    '''
    :param x:
    :param flux:
    :param meanFreePath:
    :param plot:
    :return: gradient and the error

    variable name               type                description
    temp                        float               stores the temporary values
    flux                        float               stores the parameters being evaluated
    x                           numpy array         stores the x-axis values for plotting
    y                           numpy array         stores the y-axis values for plotting
    fit_m                       float               stores the gradient
    fit_c                       float               stores the intercept
    variance_m                  float               stores the variance of the gradient
    variance_c                  float               stores the variance of the intercept
    sigma_m                     float               stores the standard error of the
gradient
    sigma_c                     float               stores the standard error of the
intercept
    figure                      object              stores the figure object
    axes                        object              stores the subplot object
    '''

    temp = []
    for counter in range(0, len(flux), 1):
        if (flux[counter] == 0):
            temp.append(counter)
    flux = np.delete(flux, temp)
    x = np.delete(x, temp)
    ey = np.delete(ey, temp)
    y = np.log(flux)

    fit_parameters, fit_errors = np.polyfit(x, y, 1, w=ey, cov=True)
    fit_m = fit_parameters[0]
    fit_c = fit_parameters[1]
    variance_m = fit_errors[0][0]
    variance_c = fit_errors[1][1]
    sigma_m = np.sqrt(variance_m)
    sigma_c = np.sqrt(variance_c)

    if plot:
        figure = plt.figure()
        axes = figure.add_subplot(111)
        axes.plot(x, fit_m * x + fit_c)
        axes.scatter(x, y)
        axes.set_xlabel("Number")
        axes.set_ylabel("Frequency")

        print('Linear np.polyfit of y = m*x + c')
        print('Gradient  m = {:04.10f} +/- {:04.10f}'.format(fit_m, sigma_m))
        print('Intercept c = {:04.10f} +/- {:04.10f}'.format(fit_c, sigma_c))
        print("mean free path -1/lambda =" + str(-1 / meanFreePath))

    return [fit_m, sigma_m], [x, y, ey]


class LCG:  # taken directly from the course material
    """
    A general linear congruential generator
    """

    def __init__(self, m, a, c):
        self.m = m
        self.a = a
        self.c = c
        self.seed = 0
        self.this_sample = self.seed  # Set initial sequence value to be the seed
        # Can return the original seed value if we want to!

    def sample(self):
        # Generate the sample (between 0 and m)
        self.this_sample = (self.a * self.this_sample + self.c) % self.m
        # Return the sample (between 0 and 1)
        return self.this_sample / self.m

    # Allow the seed value to be set explicitly
    def set_seed(self, seed_val):
        self.seed = seed_val
        self.this_sample = self.seed


class Random_Generator(object):
    def __init__(self):
        '''
        This is the initialisation function
            variable name               type                    description
            self.array                  list                    stores the values obtained
from random number generator methods
            self.x                      list                    stores x-values for plotting
            self.expDist                nested list             stores the exponential
distibution
            self.length                 numpy array             stores the length of an
evaluated vector
            self.vectors                float                   store the complete set of
vectors
        '''
        self.array = []
        self.x = []
        self.expDist = []
        self.length = []
        self.vectors = []

    def badGenerateArray(self, N):
        # generates numbers between 0 and 1.

        random_set = np.array([])
        # This is a pretty terrible way to generate 1000 random numbers, but it explicitly
shows that we can
        # generate 1 random number at a time and put 1000 of them into an array
        for i in range(N):
            random_set = np.append(random_set, np.random.random())
        self.array = random_set

    def weightedAverage(self, y, ey):
        y = np.array(y)
        ey = np.array(ey)
        weighted_mean = np.sum(y / ey ** 2) / np.sum(1 / ey ** 2)
        error = np.sqrt(1 / np.sum(1 / ey ** 2))
        return weighted_mean, error

    def randssp(self, p, q):
        # q - how many sets of random numbers
        # p - how many numbers in each set

        global m, a, c, x

        try:
            x
        except NameError:
            m = pow(2, 31)
            a = pow(2, 16) + 3
            c = 0
            x = 123456789

        try:
            p
        except NameError:
            p = 1
        try:
            q
        except NameError:
            q = p

        r = np.zeros([p, q])

        for l in range(0, q):
            for k in range(0, p):
                x = np.mod(a * x + c, m)
                r[k, l] = x / m

        return r

    def betterGenerateArray(self, N):
        random_set = np.random.uniform(0, 1, N)
        self.array = random_set

    def testRandomGenerator(self, N):
        # plots the distribution of the random numbers generated
        figure = plt.figure()
        axes = figure.add_subplot(131)
        axes2 = figure.add_subplot(132)
        axes3 = figure.add_subplot(133)

        bins = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
        # generate numbers bad
        self.badGenerateArray(N)
        bin_frequencies, bin_locations = np.histogram(self.array, bins=bins)
        axes.hist(self.array, bins=bins)
        axes.set_xlabel("Number")
        axes.set_ylabel("Frequency")
        mean = np.mean(bin_frequencies)
        std = np.std(bin_frequencies)
        # generate numbers good
        self.betterGenerateArray(N)
        bin_frequencies, bin_locations = np.histogram(self.array, bins=bins)
        axes2.hist(self.array, bins=bins)
        axes2.set_xlabel("Number")
        axes2.set_ylabel("Frequency")
        mean2 = np.mean(bin_frequencies)
        std2 = np.std(bin_frequencies)

        # pseudo random
        temp = self.randssp(1, N)
        temp = temp[0]
        bin_frequencies, bin_locations = np.histogram(self.array, bins=bins)
        axes3.hist(temp, bins=bins)
        axes3.set_xlabel("Number")
        axes3.set_ylabel("Frequency")
        mean3 = np.mean(bin_frequencies)
        std3 = np.std(bin_frequencies)

        # compare number generators
        # binomial calculation
        probablity = 1 / (len(bins) - 1)
        binomial_mean = N * probablity
        binomial_std = np.sqrt(binomial_mean * (1 - binomial_mean / N))
        print("The expected value from the binomial distribution is: {:0.9} with fluctuation
of  {:0.9}".format(
            binomial_mean, binomial_std))
        print("The first generator gave a mean of {:0.9} +/- {:0.9}".format(mean, std))
        print("The second generator gave a mean of {:0.9} +/- {:0.9}".format(mean2, std2))
        print("The pseudo random generator gave a mean of {:0.9} +/- {:0.9}".format(mean3,
std3))

    def spectraPhen(self):
        # Samples using the inbuilt generator
        nsamples = 2000
        x_np = np.zeros(nsamples)
        y_np = np.zeros(nsamples)
        z_np = np.zeros(nsamples)
        for i in range(nsamples):
            x_np[i] = 2 * np.random.random() - 1
            y_np[i] = 2 * np.random.random() - 1
            z_np[i] = 2 * np.random.random() - 1

        fig = plt.figure()
        ax1 = fig.add_subplot(111, projection='3d')
        ax1.scatter(x_np, y_np, z_np)

        # Makes samples using RANDU algorithm
        randu = LCG(2 ** 31, 65539, 0)
        nsamples = 5000
        x_np = np.zeros(nsamples)
        y_np = np.zeros(nsamples)
        z_np = np.zeros(nsamples)
        randu.set_seed(28538)
        for i in range(nsamples):
            x_np[i] = 2 * randu.sample() - 1
            y_np[i] = 2 * randu.sample() - 1
            z_np[i] = 2 * randu.sample() - 1

        fig = plt.figure()
        ax1 = fig.add_subplot(111, projection='3d')
        ax1.scatter(x_np, y_np, z_np)

    def plotDistribution3D(self, x, y, z):
        fig = plt.figure()
        ax1 = fig.add_subplot(111, projection='3d')
        ax1.scatter(x, y, z)

    def quickHist(self, data, step):
        bins = np.arange(0, 1, step)
        figure = plt.figure()
        axes = figure.add_subplot(111)
        axes.set_ylabel("Frequency")
        axes.set_xlabel("Number")
        axes.hist(data, bins=bins)

    def generate_vectors(self, accuracy, plot, r=1):
        # generates isotropic vectors distributed on a surface of a sphere
        # range of u is -1 to 1
        # range of theta is 0 to 2 pi
        self.betterGenerateArray(accuracy)
        u = 2 * self.array - 1
        self.betterGenerateArray(accuracy)
        theta = 2 * np.pi * self.array

        x = r * np.cos(theta) * np.sqrt(1 - u ** 2)
        y = r * np.sin(theta) * np.sqrt(1 - u ** 2)
        z = r * u
        self.length = np.sqrt(x ** 2 + y ** 2 + z ** 2)
        if plot:
            self.plotDistribution3D(x, y, z)
        self.vectors = [x, y, z]

    def generateExponentialDist(self, N, plot, meanFreePath):

        step = 0.1
        bins = np.arange(0, 1, step)
        self.betterGenerateArray(N)
        self.expDist = -1 * meanFreePath * np.log(self.array)
        histogram = np.histogram(self.expDist)
        y = histogram[0]
        x = histogram[1][:len(histogram[1]) - 1] + step / 2
        y = np.log(y)
        # eliminate zeroes
        temp = []
        for counter in range(0, len(y), 1):
            if y[counter] == -np.inf:
                temp.append(counter)
        y = np.delete(y, temp)
        x = np.delete(x, temp)

        fit_parameters, fit_errors = np.polyfit(x, y, 1, cov=True)
        fit_m = fit_parameters[0]
        fit_c = fit_parameters[1]
        variance_m = fit_errors[0][0]
        variance_c = fit_errors[1][1]
        sigma_m = np.sqrt(variance_m)
        sigma_c = np.sqrt(variance_c)

        if plot:
            figure = plt.figure()
            axes = figure.add_subplot(111)
            axes.plot(x, fit_m * x + fit_c)
            axes.scatter(x, y)
            axes.set_xlabel("Number")
            axes.set_ylabel("Frequency")
            axes.errorbar(x, y, yerr=np.ones(len(y)) * 0.4, fmt='b+')

            print('Linear np.polyfit of y = m*x + c')
            print('Gradient  m = {:04.10f} +/- {:04.10f}'.format(fit_m, sigma_m))
            print('Intercept c = {:04.10f} +/- {:04.10f}'.format(fit_c, sigma_c))
            print("mean free path -1/lambda =" + str(-1 / meanFreePath))

        self.expDist = [x, y]

    def generateDistVectors(self, meanFreePath, N):
        r = np.random.uniform(0, 1, N)
        r = -1 * meanFreePath * np.log(r)  # makes the lengths exponentially distributed

        number = np.random.uniform(0, 1, N)

        u = 2 * number - 1

        number = np.random.uniform(0, 1, N)

        theta = 2 * np.pi * number

        x = r * np.cos(theta) * np.sqrt(1 - u ** 2)
        y = r * np.sin(theta) * np.sqrt(1 - u ** 2)
        z = r * u

        self.vectors = [x, y, z]
        self.plotDistribution3D(x, y, z)

    def genDistVector(self, meanFreePath):
        r = np.random.uniform(0, 1, 1)
        r = -1 * meanFreePath * np.log(r)  # makes the lengths exponentially distributed

        self.betterGenerateArray(1)
        u = 2 * self.array - 1
        self.betterGenerateArray(1)
        theta = 2 * np.pi * self.array

        x = r * np.cos(theta) * np.sqrt(1 - u ** 2)
        y = r * np.sin(theta) * np.sqrt(1 - u ** 2)
        z = r * u

        return [x.tolist()[0], y.tolist()[0], z.tolist()[0]]

    def getLength(self, vector):
        return np.sqrt(vector[0] ** 2 + vector[1] ** 2 + vector[2] ** 2)

    def generateDirection(self, r, vector):
        length = self.getLength(vector)  # normalizes the vector -> length 1
        vector[0] = vector[0] / length
        vector[1] = vector[1] / length
        vector[2] = vector[2] / length

        for counter in range(0, 3, 1, ):  # extends the vector appropriately
            vector[counter] *= r

        return vector


class Experiment(Random_Generator):
    def __init__(self, material1, N, thickness, thickness2, name, material2):
        self.data = []
        self.absArea = material1[0]
        self.scatterArea = material1[1]
        self.density = material1[2]  # number density
        self.thickness = thickness
        self.thickness2 = thickness2
        self.N = N
        self.meanFreePath = 1 / (self.density * self.absArea + self.density *
self.scatterArea)
        self.history = []
        self.gradients = [[], [], []]  # stores gradients form many experiments
        # absorbed, reflected, transmitted
        self.name = name
        self.material2 = material2
        self.abs_err = []
        self.pass_err = []
        self.ref_err = []
        self.x = []

        super().__init__()

    def randomWalk(self, T):
        vector = self.genDistVector(self.meanFreePath)
        # T -> thickness
        SigmaT = 0
        SigmaT += self.density * self.absArea
        SigmaT += self.density * self.scatterArea
        self.meanFreePath = 1 / SigmaT

        prob_absorption = self.density * self.absArea / (self.density * self.absArea +
self.density * self.scatterArea)
        is_absorbed = 0
        i = 0
        x = 0
        self.history = [[], [], []]
        while is_absorbed == 0:
            if i == 0:
                vector[0] += self.getLength(vector)  ## adds the vectors
            else:
                vector = self.addVectors(self.genDistVector(self.meanFreePath), vector)

            self.history[0].append(vector[0])
            self.history[1].append(vector[1])
            self.history[2].append(vector[2])
            x = vector[0]
            if (x < 0):
                return "reflected", self.history
            elif (x > T):
                return "passed", self.history

            # evaluate what happens to the particle
            probability = np.random.uniform(0, 1, 1)[0]
            if probability <= prob_absorption:
                # gets absorbed
                is_absorbed = 1
                return "absorbed", self.history
            else:
                # scatters
                i += 1

    def compressData(self, array):
        # compresses experiment data
        absorbed = []
        passed = []
        reflected = []
        for entry in array:
            absorbed.append(entry[0])
            passed.append(entry[1])
            reflected.append(entry[2])

        self.abs_err.append(np.std(absorbed))
        self.pass_err.append(np.std(passed))
        self.ref_err.append(np.std(reflected))

        return [np.mean(absorbed), np.mean(passed), np.mean(reflected)]

    def addVectors(self, vector1, vector2):
        temp = [0, 0, 0]
        for counter in range(0, 3, 1):
            temp[counter] = vector1[counter] + vector2[counter]

        return temp

    def plotRandomWalk(self):
        x = self.history[0]
        y = self.history[1]
        z = self.history[2]
        fig = plt.figure()
        ax1 = fig.add_subplot(111, projection='3d')
        ax1.plot(x, y, z)

    def experiment(self, N, thickness):
        # performs an experiment N times
        histories = []
        results = [0, 0, 0]
        for i in range(0, N, 1):
            result, temp = self.randomWalk(thickness)
            histories.append(result)

        for entry in histories:
            # counts the outcomes
            if entry == "absorbed":
                results[0] += 1
            elif entry == "passed":
                results[1] += 1
            elif entry == "reflected":
                results[2] += 1  #
        self.data = results
```

```
def experimentWoodcock(self, N, T1, T2):
    # performs an experiment N times
    histories = []
    results = [0, 0, 0]
    for i in range(0, N, 1):
        result, temp = self.woodcockMethod()
        histories.append(result)

    for entry in histories:
        # counts the outcomes
        if entry == "absorbed":
            results[0] += 1
        elif entry == "passed":
            results[1] += 1
        elif entry == "reflected":
            results[2] += 1  #
    self.data = results

def thicknessPlot(self, N, min_T, max_T, step):
    array2 = []
    results = []
    thickness = np.arange(min_T, max_T, step=step)
    for entry in thickness:
        temp = []
        for i in range(5):
            self.experiment(N, entry)
            temp.append(self.data)

        temp = self.compressData(temp)
        results.append(temp)

    self.data = results

    self.thickness = thickness  # array storing thickness

    absorbed = []
    transmitted = []
    reflected = []

    for entry in self.data:
        absorbed.append(entry[0])
        transmitted.append(entry[1])
        reflected.append(entry[2])

    figure = plt.figure()

    # plotting for absorption
    flux = np.array(absorbed)
    marker = [0, 0, 0]
    temp = 0
    for entry in absorbed:
        if entry == 0:
            temp += 1

    if not temp == len(absorbed):
        temp, array = linearFit(thickness, flux, self.meanFreePath, False, self.abs_err)
        self.gradients[0].append(temp)
        ax1 = figure.add_subplot(131)
        ax1.scatter(thickness, flux, label="Absorption")
        ax1.plot(thickness, flux)
        ax1.errorbar(thickness, flux, yerr=self.abs_err, fmt='b+')
        array2.append(array)
    else:
        marker[0] = 1

    temp = 0
    for entry in reflected:
        if entry == 0:
            temp += 1

    if not temp == len(reflected):
        # plotting for reflection
        flux = np.array(reflected)
        temp, array = linearFit(thickness, flux, self.meanFreePath, False, self.ref_err)
```

```
            self.gradients[1].append(temp)
            ax2 = figure.add_subplot(132)
            ax2.scatter(thickness, flux, label="Reflection")
            ax2.errorbar(thickness, flux, yerr=self.ref_err, fmt='b+')
            array2.append(array)
        else:
            marker[1] = 1

        temp = 0
        for entry in transmitted:
            if entry == 0:
                temp += 1
        if not temp == len(transmitted):
            # plotting for transmission
            flux = np.array(transmitted)
            temp, array = linearFit(thickness, flux, self.meanFreePath, False, self.pass_err)
            self.gradients[2].append(temp)
            ax3 = figure.add_subplot(133)
            ax3.scatter(thickness, flux, label="Transmission")
            ax3.plot(thickness, flux)
            ax3.errorbar(thickness, flux, yerr=self.ref_err, fmt='b+')
            array2.append(array)

        else:
            marker[2] = 1

        if np.array(marker).sum() != 3:
            figure.legend()
            temp = -1 * 1 / self.gradients[2][0][0]
            temp2 = self.gradients[2][0][1] / self.gradients[2][0][0]
            temp2 *= temp
            temp2 = np.abs(temp2)
            print(
                "The attenuation length for " + str(self.name) + " is " + str(temp) + str("
+/- ") + str(temp2) + " cm")
        else:
            print("no data obtained")

    def percentageAbsorption(self, N):
        # check variation of error with number of neutrons used
        thickness = 10  # cm
        results = []
        for counter in range(10):
            self.experiment(N, thickness)
            results.append(self.data)

        absorbed = []
        transmitted = []
        reflected = []

        for entry in results:
            absorbed.append(entry[0])
            transmitted.append(entry[1])
            reflected.append(entry[2])

        print(absorbed)
        print(transmitted)
        print(reflected)

        mean_absorbed = np.mean(absorbed) / N * 100
        mean_transmitted = np.mean(transmitted) / N * 100
        mean_reflected = np.mean(reflected) / N * 100

        std_abs = np.std(absorbed) / np.mean(absorbed) * mean_absorbed
        std_tra = np.std(transmitted) / np.mean(transmitted) * mean_transmitted
        std_ref = np.std(reflected) / np.mean(reflected) * mean_reflected

        print("--------------------------------------------------------------------------------
---------")
        print("Transmission through a fixed Thickness - " + str(self.name))
        print("Thickness : 10 cm ")
        print("Total Neutrons: " + str(N))
        print("Neutrons Reflected: " + str(np.mean(reflected)))
        print("Neutrons Transmitted: " + str(np.mean(transmitted)))
        print("Neutrons Absorbed: " + str(np.mean(absorbed)))
        print("Percentage Transmitted: " + str(mean_transmitted))

        print(
            'The average percentage of neutrons that were absorbed is: {:04.10f} % +/-
{:04.10f}'.format(mean_absorbed,

std_abs))
        print('The average percentage of neutrons that were transmitted is: {:04.10f} % +/-
{:04.10f}'.format(
            mean_transmitted, std_tra))
        print('The average percentage of neutrons that were reflected is: {:04.10f} % +/-
{:04.10f}'.format(
            mean_reflected, std_ref))

    def woodcockMethod(self):
        # Material 1 is always the first block encountered
        T1 = self.thickness
        T2 = self.thickness2 + T1
        marker = 0
        Sigma1 = 1 / self.meanFreePath
        Sigma2 = self.material2[0] * self.material2[2] + self.material2[1] * self.material2[2]
        SigmaT = 0
        if Sigma1 > Sigma2:
            marker = 2
            SigmaT = Sigma1
            prob_fictitious = 1 - Sigma2 / SigmaT
        else:
            marker = 1
            SigmaT = Sigma2
            prob_fictitious = 1 - Sigma1 / SigmaT

        self.meanFreePath = 1 / SigmaT

        # T -> thickness
        prob_absorption1 = self.density * self.absArea / (self.density * self.absArea +
self.density * self.scatterArea)
        prob_absorption2 = self.material2[0] * self.material2[2] / (
                self.material2[0] * self.material2[2] + self.material2[1] * self.material2[2])

        is_absorbed = 0
        i = 0
        x = 0
        self.history = [[], [], []]

        r = np.random.uniform(0, 1)
        r = -1 * self.meanFreePath * np.log(r)
        vector = [r, 0, 0]

        while is_absorbed == 0:
            if i == 0:
                vector[0] += self.getLength(vector)
            else:
                vector = self.addVectors(self.genDistVector(self.meanFreePath), vector)

            self.history[0].append(vector[0])
            self.history[1].append(vector[1])
            self.history[2].append(vector[2])
            x = vector[0]
            if (x < 0):
                return "reflected", self.history
            elif (x > T2):
                return "passed", self.history

            # evaluate what happens to the particle
            probability = np.random.uniform(0, 1)

            # check the region the particle is in:
            if (x < T1) and (x > 0):  # it's in region 1
                if marker == 1:
                    if probability > prob_fictitious:
                        r = np.random.uniform(0, 1)
                        r = -1 * self.meanFreePath * np.log(r)
                        direction = self.generateDirection(r, vector)
                        vector = self.addVectors(direction, vector)  # makes the step
                        self.history[0].append(vector[0])
                        self.history[1].append(vector[1])
                        self.history[2].append(vector[2])
                        i += 1
                    else:
                        probability = np.random.uniform(0, 1, 1)[0]
                        if probability >= prob_absorption1:
                            # gets absorbed
                            is_absorbed = 1
                            return "absorbed", self.history
                        else:
                            # scatters
                            i += 1
                else:
                    # normal behaviour in region 1
                    probability = np.random.uniform(0, 1, 1)[0]
                    if probability >= prob_absorption1:
                        is_absorbed = 1
                        return "absorbed", self.history
                    else:  # scatters
                        i += 1

            elif (x < T2) and (x >= T1):  # it's in region 2
                if marker == 2:
                    if probability > prob_fictitious:
                        r = np.random.uniform(0, 1)
                        r = -1 * self.meanFreePath * np.log(r)
                        direction = self.generateDirection(r, vector)
                        vector = self.addVectors(direction, vector)  # makes the step
                        self.history[0].append(vector[0])
                        self.history[1].append(vector[1])
                        self.history[2].append(vector[2])
                        i += 1
                    else:
                        probability = np.random.uniform(0, 1)
                        if probability >= prob_absorption2:
                            is_absorbed = 1
                            return "absorbed", self.history
                        else:
                            # scatters
                            i += 1
                else:
                    # normal behaviour in region 2
                    probability = np.random.uniform(0, 1, 1)[0]
                    if probability >= prob_absorption2:
                        is_absorbed = 1
                        return "absorbed", self.history
                    else:  # scatters
                        i += 1

    def percentageWoodcock(self, N):
        # check variation of error with number of neutrons used
        thickness = 10  # cm
        results = []
        for counter in range(50):
            self.experimentWoodcock(N, 10, 10)
            results.append(self.data)

        absorbed = []
        transmitted = []
        reflected = []

        for entry in results:
            absorbed.append(entry[0])
            transmitted.append(entry[1])
            reflected.append(entry[2])

        print(absorbed)
        print(transmitted)
        print(reflected)

        mean_absorbed = np.mean(absorbed) / N * 100
        mean_transmitted = np.mean(transmitted) / N * 100
        mean_reflected = np.mean(reflected) / N * 100

        std_abs = np.std(absorbed) / np.mean(absorbed) * mean_absorbed
        std_tra = np.std(transmitted) / np.mean(transmitted) * mean_transmitted
        std_ref = np.std(reflected) / np.mean(reflected) * mean_reflected

        print("-----------------------------------------------------------------------------
---------")
        print("Transmission through a fixed Thickness - " + str(self.name))
        print("Thickness : 10 cm ")
        print("Total Neutrons: " + str(N))
        print("Neutrons Reflected: " + str(np.mean(reflected)))
        print("Neutrons Transmitted: " + str(np.mean(transmitted)))
        print("Neutrons Absorbed: " + str(np.mean(absorbed)))
        print("Percentage Transmitted: " + str(mean_transmitted))

        print(
            'The average percentage of neutrons that were absorbed is: {:04.10f} % +/-
{:04.10f}'.format(mean_absorbed,

std_abs))
        print('The average percentage of neutrons that were transmitted is: {:04.10f} % +/-
{:04.10f}'.format(
            mean_transmitted, std_tra))
        print('The average percentage of neutrons that were reflected is: {:04.10f} % +/-
{:04.10f}'.format(
            mean_reflected, std_ref))


def main():
    lead = [0.158E-24, 11.221E-24, 1.34E23]
    graphite = [0.0045E-24, 4.74E-24, 3.296E22]
    water = [0.6652E-24, 103.0E-24, 3.343E22]

    gen1 = Experiment(water, 1000, 10, 10, "water",
                      lead)
    gen2 = Experiment(lead, 1000, 10, 10, "lead", graphite)
    gen3 = Experiment(graphite, 1000, 10, 10, "graphite", water)

    gen1.generateDistVectors(45,1000)
    gen1.generate_vectors(1000, True)
    gen1.spectraPhen()
    gen1.testRandomGenerator(1000)
    gen1.randomWalk(100)
    gen1.plotRandomWalk()


    gen1.generateExponentialDist(10,True,45)
    gen1.generateExponentialDist(100, True, 45)
    gen1.generateExponentialDist(1000, True, 45)
    gen1.generateExponentialDist(5000, True, 45)
    gen1.generateExponentialDist(10000, True, 45)

    print("10 cm spectra")
    gen1.percentageAbsorption(5000)
    gen2.percentageAbsorption(5000)
    gen3.percentageAbsorption(5000)

    print("Woodcock")
    gen1.percentageWoodcock(1000)
    gen3.percentageWoodcock(1000)

    print("Thickness spectra")
    gen1.thicknessPlot(500, 0, 10, 1)
    gen2.thicknessPlot(1000, 0, 20, 1)
    gen3.thicknessPlot(1000, 0, 100, 5)

main()
plt.show()
```