Simulating neutron penetration using Monte Carlo techniques

Adam Coxson

9994414

School of Physics and Astronomy

The University of Manchester

Second year computational physics project report

May 2019

Abstract

Neutron penetration of water, lead and graphite was simulated, using **Python 3. 7. 2**, to determine the proportions of neutrons absorbed, reflected and transmitted for each material. This was done for varied lengths up to the point where neutron transmission was negligible for each material. From this data, a linear fit was applied, and the attenuation lengths of absorption and scattering were calculated.

The attenuation lengths for water, lead and graphite were calculated as (1.89 ± 0.05) cm, (11.4 ± 0.18) cm and (30.6 ± 1.6) cm, with the fit giving reduced chi-squares of 12.7, 8.7 and 41.6, respectively. The large chi-squares were attributed to the model being a poor fit of the data. Additionally, neutron penetration through two different, adjacent materials was also simulated using the Woodcock method, which introduced the concept of fictious steps to effectively model neutron behaviour across material boundaries.

1. Introduction

The simulation of particles can be used to test theoretical models and help inform development of physical experiments. Physical processes are often simulated using Monte Carlo simulations. Monte Carlo methods utilise random sampling to obtain solutions for systems that have a probabilistic interpretation whether they are random or deterministic in nature.

In nuclear physics, Monte Carlo methods can be useful to model the interactions of neutrons within reactors. Fission of uranium-235 produces high energy neutrons which are too energetic to be captured by other U-235 nuclei and instead are more likely to be absorbed by U-238. For further fission of U-235, the neutrons must be slowed down to speeds that correspond to normal thermal energies [1]. Moderators surrounding the reactor scatter the neutrons to lower their energies until they are at thermal equilibrium. The nuclei of the moderators must have high probabilities of scattering the neutrons and low chances of absorption occurring. On the contrary, reactor control rods must be made of materials with high chances of absorption to limit fission if the rate is too high.

In this project, Monte Carlo methods were used to imitate Brownian motion of neutrons through water, lead and graphite. The corresponding fractions of absorbed, reflected and transmitted neutrons as well as the attenuation lengths were determined. These results were used to identify the materials with the highest number of reflections and lowest number of transmissions which can be indicative of the suitability of a material as moderators.

2. Theory

2.1 Neutron interactions

When neutrons pass through a material, a number of interactions can occur between the material's nuclei and the neutrons. In this simulation, only elastic scattering and absorption events were considered. These interactions can be represented by an interaction cross-section, σ. The probability of scattering or absorption occurring is given by the ratio of the interaction's partial cross-section to sum of all the partial cross-sections (total cross-section),

$$P(\text{Scatter event}) = \frac{\sigma_s}{\sigma_s + \sigma_a} = \frac{\sigma_s}{\sigma_T}. \quad (1)$$

where σ_s, σ_a and σ_T are the scatter, absorption and total cross-sections respectively. The macroscopic cross-section, Σ, which is the effective target area of all the nuclei contained in the volume of the material, is defined as

$$\Sigma = n\sigma, \quad (2)$$

where n is the number density of atoms in the material. The mean free path of the neutrons is given by the reciprocal of this expression,

$$\lambda_j = \frac{1}{n\sigma_j} = \frac{1}{\Sigma_j}, \tag{3}$$

where λ is the mean free path and the subscript j can represent scattering, absorption or total interaction [2]. The number of neutron collisions with moderator nuclei, over a distance x, can be defined using the mean free path

$$N_c = \frac{x}{\lambda_T}, \tag{4}$$

where N_c is the number of collisions and λ_T is the total mean free path. Considering $N(x)$ as the number of neutrons that have not collided over x,

$$N(x + dx) = N(x) - N(x)\frac{dx}{\lambda_T}, \tag{5}$$

where $N(x)\frac{dx}{\lambda_T}$ is the number of particles that suffered collisions in interval dx. This expression can be manipulated into the form

$$\frac{dN(x)}{dx} = -\frac{N(x)}{\lambda_T}, \tag{6}$$

which can be solved through separation of variables,

$$N(x) = N_0 e^{-\frac{x}{\lambda_T}}. \tag{7}$$

N_0 is the initial number of incident neutrons [3]. Dividing Equation (7) by N_0 gives, $P(X)$, the probability a neutron undergoes a collision over x before absorption or scattering,

$$P(x) = e^{-\frac{x}{\lambda_T}}. \tag{8}$$

This is the exponential probability distribution function.

2.2 Pseudo-random number generation

There are many physical processes that are not truly random but can be accurately modelled as random systems. To mimic randomness, programs use pseudo-random numbers which are not truly random, they only appear so. One such example of this is the linear congruential random number generator (LCG).

$$x_{i+1} = (ax_i + c) \bmod m, \tag{9}$$

where a, c and m are integer constants, x_i is an integer variable. Given an initial value for x_i, Equation (9) is an iterative process [4]. This sequence of iterated numbers may appear random but suffers from hyperplanes, as shown in Fig. 1. This is a problem experienced by all LCGs, as such, better methods of generating uniform random numbers must be used [5]. In this simulation, functions from the *NumPy* module in **Python** were used for random number generation. This module utilises the Mersenne Twister algorithm which does not have the problem of hyperplanes.

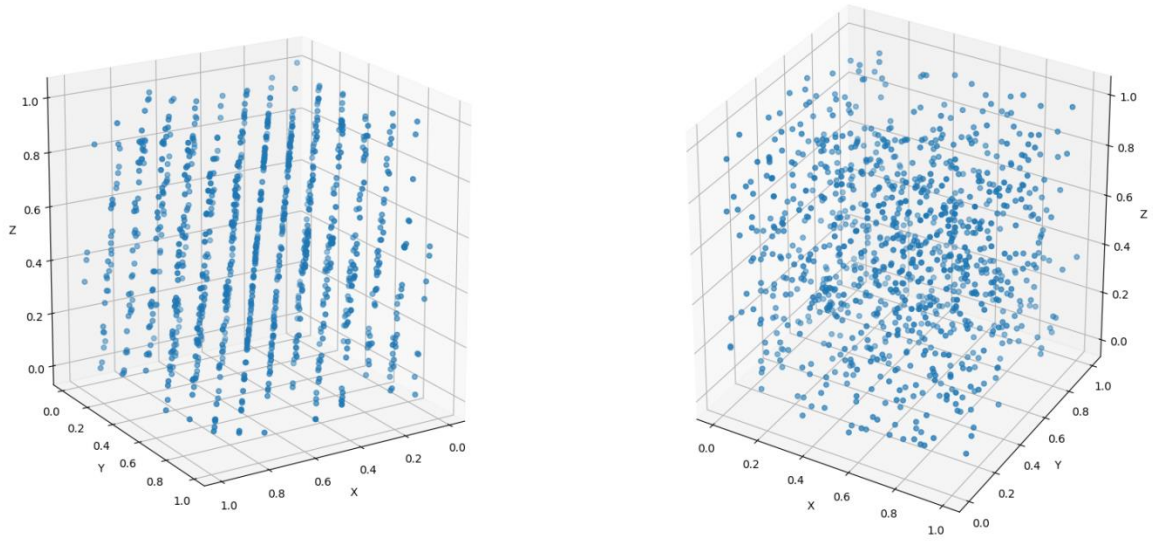

Figure 1: Plots of 3D boxes made up of 1000 pseudo-random numbers. The plot on the left was generated using an LCG, hyperplanes can be seen. The plot on the right used numbers from the Mersenne Twister algorithm, no hyperplanes were found after inspection.

2.3 Woodcock method for material boundaries

When neutrons travel between different materials, different mean free paths must be considered. For complex geometries, this can become difficult to model due to computational limitations. Woodcock, et al. developed a process which worked well for complex geometries of materials and was not as computationally demanding [6]. The method takes all steps according to the smallest mean free path out of all the considered materials. A new probability is introduced, the probability of a fictitious step,

$$P(f) = P(fictious) = 1 - \frac{\Sigma_x}{\Sigma_M}, \tag{10}$$

where Σ_x is the macroscopic area of the material the neutron is currently in and Σ_M is the maximum macroscopic area out of all the materials, i.e. the reciprocal gives the smallest λ. Just before each step, it is checked to see if it is fictitious according to Equation (10). If fictious, a step is taken but in the same direction as the previous step. If $P(f)$ is not satisfied, then the current step is not fictitious and is able to propagate in a different direction. For a neutron in the material whose area is the maximum, $\frac{\Sigma_x}{\Sigma_M}$ is 1 and hence $P(f)$ is 0; fictitious steps are never taken while in that material.

3 Simulation method

3.1 Modelling neutron scattering

Neutrons travel through mediums until they are absorbed or scatter out. This was best modelled as Brownian motion which required use of a Monte Carlo simulation. The program used for this was **Python**. First, isotropic unit spheres were generated using uniformly distributed random numbers. For each point, random numbers between -1 and 1 were assigned to the 3 cartesian co-ordinates x, y, z, representing the components of a unit vector. The length of this vector was calculated from

adding the components in quadrature. If the length was greater than 1 it was discarded, and the co-ordinates were re-sampled. Each component was then normalised by dividing it the modulus length of the vector. This ensured that each unit vector was constrained to a point on the surface of a unit sphere. Generating and plotting many of these points created a uniformly distributed sphere as in shown on the left in Fig. 2.

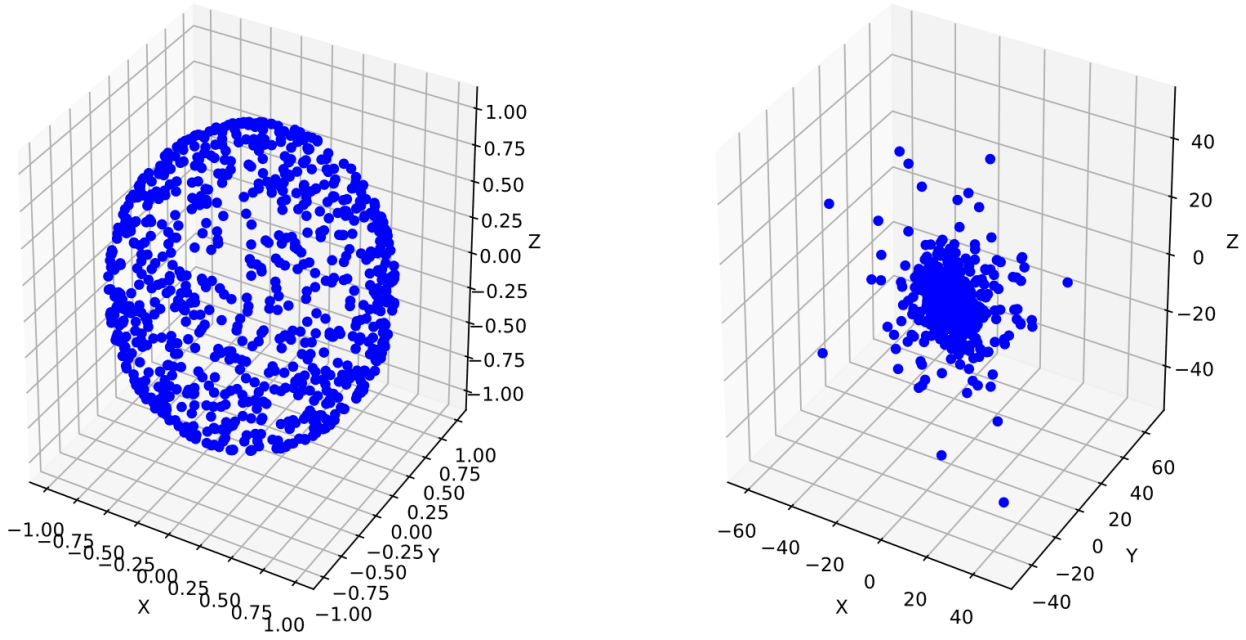

Figure 2: The plot on the left shows a spherical surface populated using uniformly distributed points. This is a unit sphere. The plot on the right shows isotropic steps. This is formed using the same process as the sphere but with each point multiplied by a length drawn from an exponential distribution. Note that most points are clustered near the centre, as large lengths are less probable.

This unit sphere gave the direction the neutron would take after a scatter event. Next required the determination of the step length travelled until the next interaction. From Equation (8), the cumulative distribution function was obtained, step lengths of distance x were calculated according to the inverse of this,

$$x = -\lambda_T \ln(1-u)\,, \tag{11}$$

where u represents uniform random numbers between 0 and 1. Generating the unit sphere and multiplying each point by a different distance from Equation (11) produces isotropic steps of length determined by an exponential distribution, as shown in Fig. 2. In the simulation, this isotropic distribution of exponential step lengths was used to determine the direction taken and distance travelled by a neutron when scattering occurred; this was one step.

After each step, the position of the neutron was checked to see if it had scattered out the material either as a transmission or reflection. If it was still inside the block then a random number was generated between 0 and 1 and compared to the probability of absorption, $P(\text{Absorbed})$, if this probability was greater, the neutron was considered absorbed. If the particle had not been absorbed, it would repeat the scattering step process detailed above. From this process, random walks of neutrons were produced, as shown in Fig. 3, overleaf.

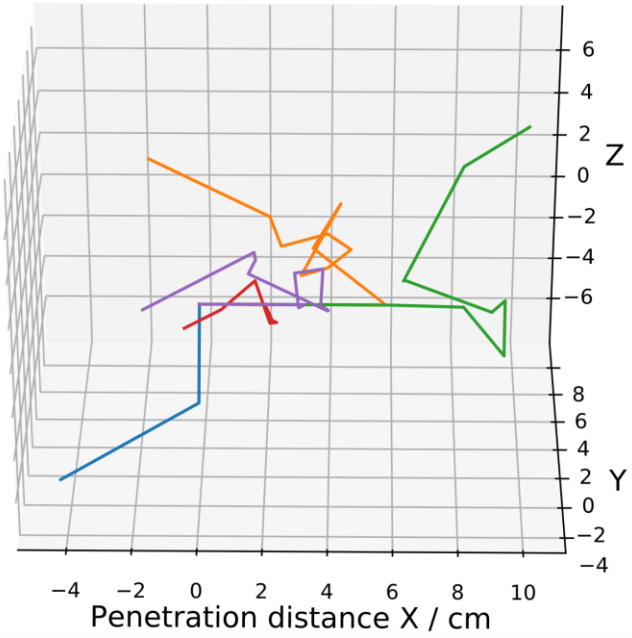

Figure 3: Random walk of 5 different neutrons through 10cm of graphite. The green neutron path shows transmission whereas the others reflected.

3.2 Neutron simulation for mediums of 10cm length

Neutrons at normal incidence to slabs of material were simulated. The thickness of the material along the neutron penetration axis was 10cm and assumed infinite in the axes perpendicular to this, i.e. infinitely wide. The percentages of neutrons absorbed, reflected and transmitted for 10 cm of water, lead and graphite were determined. At first, 100 runs of 500 neutrons were simulated, the mean and standard deviation were recorded. The number of neutrons in each run was then increased by intervals of 500 neutrons until 10000 neutrons were simulated. The standard deviations from the means of repeated runs were plotted against the corresponding number of neutrons. This was also done for the statistical error on counted numbers arising from the Poisson distribution, which is the square root of the counts. The optimal neutron number was decided, a compromise between percentage error and a low enough number of neutrons to have acceptable computation times. This number was then used as the standard for all further calculations. Percentages of transmission, absorption and reflection were then found for 10 cm of each moderator type.

3.3 Calculating attenuation lengths

Neutron transmissions were determined for different lengths of each moderator. The same method as in section 3.2 was applied for each length to get a mean transmission and associated error. The range of lengths used was large enough for the transmission percentage to decay below 1% for each moderator. Rearranging Equation (7) to give the fraction of neutrons transmitted for a given thickness, L, and then taking its natural logarithm,

$$\ln n_T = \ln \frac{N(L)}{N_0} = -\frac{L}{\lambda_T}. \tag{12}$$

Where n_T is the neutron transmission fraction, and λ_T the attenuation length. A linear least-squares fit was applied to the natural logarithm of each transmission fraction with respect to the length of

the moderator. These were plotted, and the negative reciprocal of the gradient gave the attenuation length. This method for calculating attenuation lengths was verified by finding the absorption attenuation length of water, i.e. the mean free path when scattering was ignored, 45 cm. To do this, a set of exponentially distributed random numbers were created using Equation (11), these values were made to represent neutron absorption counts. These were then plotted in a histogram of counts against length travelled, as shown in Fig. 4. The attenuation from this method was calculated as (44.7 ± 0.5) cm, consistent with the theoretical value used to generate the exponential distances, 45 cm.

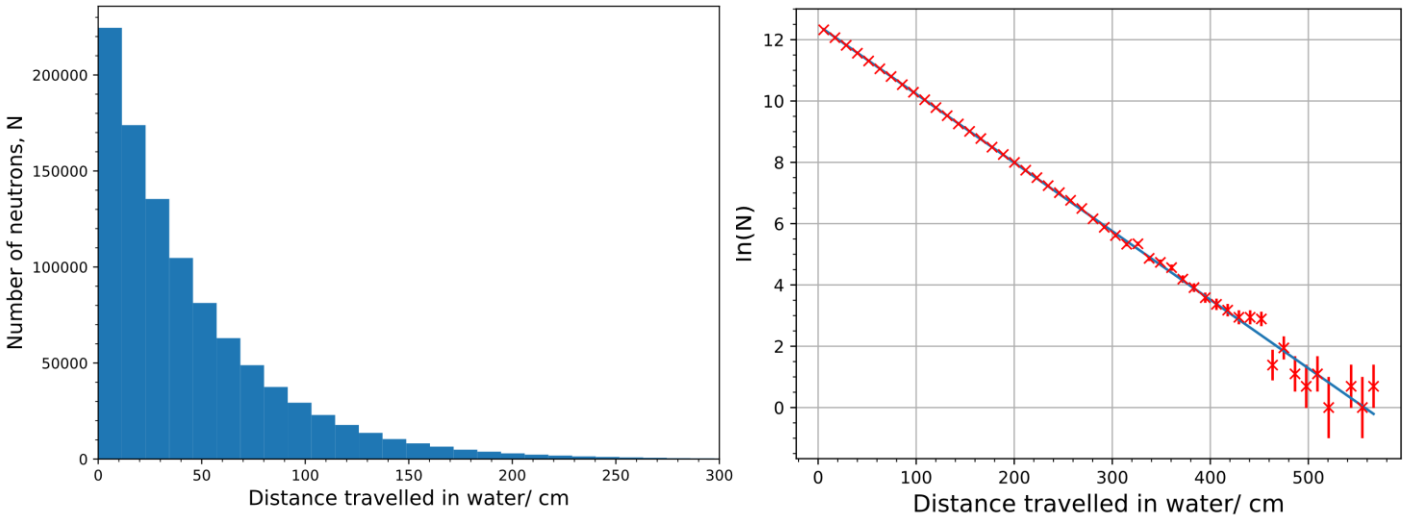

Figure 4: The plot on the left shows a histogram of the absorbed counts against length. The plot on the right is the log of the bin counts plotted against the respective distance each bin is found at. Error bars may be too small to be seen.

3.4 Simulating the Woodcock method

To simulate the Woodcock method, 2 slabs of different material were used, each 10 cm thick along the incidence axis. Before each step, the program determined whether the step was fictious by comparing the fictitious probability from Equation (10) to a random number between 0 and 1. If the random number was greater than the fictious probability then the step was not fictitious and a step in a random direction taken. Otherwise the step was taken in the same direction as defined in the previous non-fictious step. Means and standard deviations were found for the 3 neutron outcomes.

All combinations of the three moderator materials were tested. From the results, the combination of two moderators that acted as the best neutron moderator was identified. This was the combination with the lowest transmission and highest reflectivity in order to contain as many neutrons as possible; in the context of retaining neutrons for fission reactions and acting as reactor shielding.

4. Results

4.1 Neutron simulation for mediums of 10cm length

For 10 cm of water, the percentage errors on the neutron count outcomes were plotted against the number of neutrons simulated, as shown in Fig. 5.

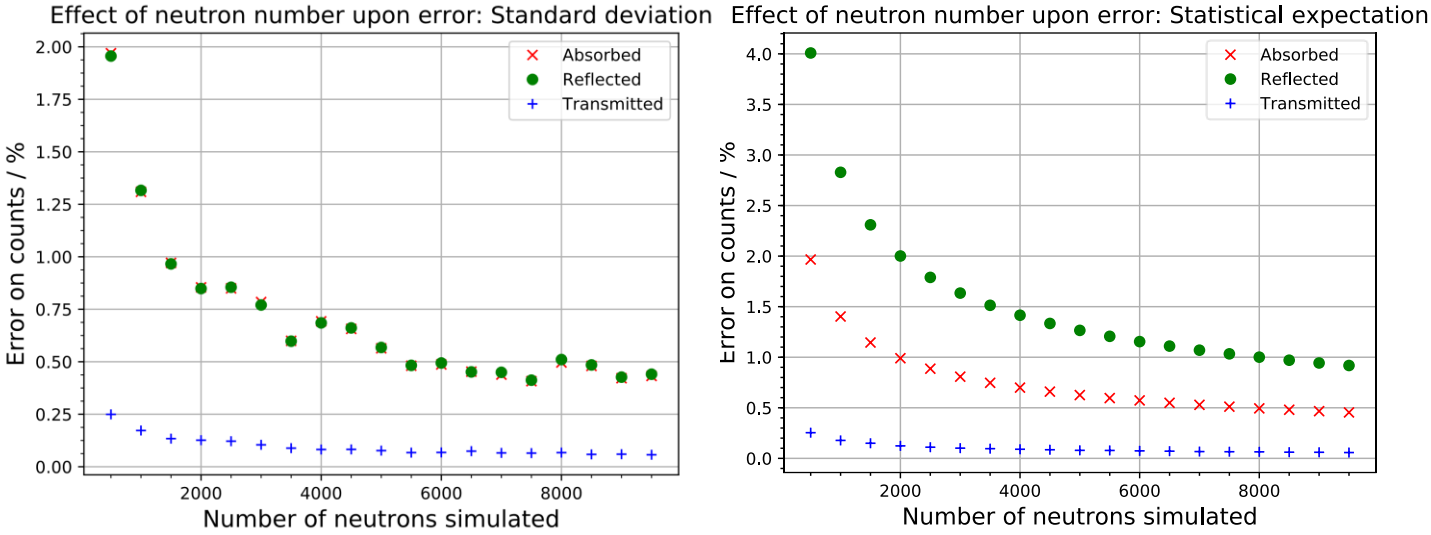

Figure 5: Graphs of percentage error against neutrons simulated for 10 cm of water. The left plot is from the standard deviation error, note that reflected and absorbed points may over plot. Right is the expected error.

At 5000 neutrons, the absorbed percentage error was approximately 0.60 %. More neutrons could be simulated but at the cost of computation time so 5000 neutrons were chosen as the standard number for further calculations since it had an error less than 1 %. This process was also repeated for lead and graphite, similar results to water were obtained and as a result, the same conclusion to simulate 5000 neutrons was made.

The neutron outcomes for 10 cm slabs of water, lead and graphite are shown in Table 1. These outcomes were obtained from averaging 100 runs of 5000 neutrons each.

Material (10cm)	**Absorbed / %**	**Reflected / %**	**Transmitted / %**
Water	19.6 ± 0.6	80.1 ± 0.6	0.32 ± 0.08
Lead	8.8 ± 0.4	62.6 ± 0.7	28.6 ± 0.4
Graphite	0.72 ± 0.1	68.5 ± 0.8	30.8 ± 0.8

Table 1: Percentage of neutrons absorbed, reflected and transmitted for 10 cm thick slabs of water, lead and graphite. Values are from 100 runs of 5000 simulated neutrons each.

This data shows that 10 cm of water had the highest absorption and reflection percentages and lowest transmission percentages. Lead and graphite had relatively high transmission rates. Furthermore, the random walks often showed the neutrons had more scattering events in water than the lead and graphite. This suggested 10 cm of water was a better material for a moderator

since neutron transmission was negligible and it had a higher number of collisions, necessary to thermalize neutrons.

The errors in Table 1 were obtained from the standard deviation of the 100 runs for each value. They were approximately the same as $\sqrt{N}$, the statistical estimate of error, where *N* is the number of neutrons counted for one particular outcome. The statistical error expectation was a result of the neutron outcomes following a Poisson distribution. However, the actual errors were slightly smaller since they follow a binomial distribution. This can be seen in the reflection data in Fig. 5, the statistical expectation values were approximately double the errors from the standard deviation.

4.2 Attenuation lengths

The attenuation lengths for water, lead and graphite were determined using the procedure as described in section 3.3. The lengths are shown in Table 2. Neutron outcome percentages were calculated for different lengths and plotted. These are shown for water, lead and graphite in Fig. 6, overleaf.

Material	**Total length simulated / cm**	**Attenuation Length / cm**	**Reduced Chi-squared**
Water	10	(1.89 ± 0.05)	12.7
Lead	100	(11.4 ± 0.2)	8.7
Graphite	150	(30.6 ± 1.6)	41.6

Table 2: Attenuation lengths obtained for water, lead and graphite. Errors on the lengths and reduced chi-squares from the fitting procedure are given. The total length simulated was the maximum range of values required for a graph to be produced that could be accurately fitted to.

Water had the smallest attenuation length which is consistent with the negligible transmission rate in Table 1. The reduced chi-squares were outside of the optimal range, 0.5 – 2. These inconsistences were attributed to the linear fit being a poor model of the data. Deviation below the fit line for the first half and increasing deviation above the fit is evident on all 3 logarithm graphs in Fig. 6.

For each material type, the total length to simulate was determined by the point of saturation. This was defined as the length where the percentage of neutrons absorbed and reflected had become constant and transmission was negligible. For example, in water, neutron transmission dropped to 0 % and absorption was constant at just under 20 % beyond 8 cm. Similarly, for lead the transmission was 0 % and absorption constant at approximately 27 % beyond 80 cm.

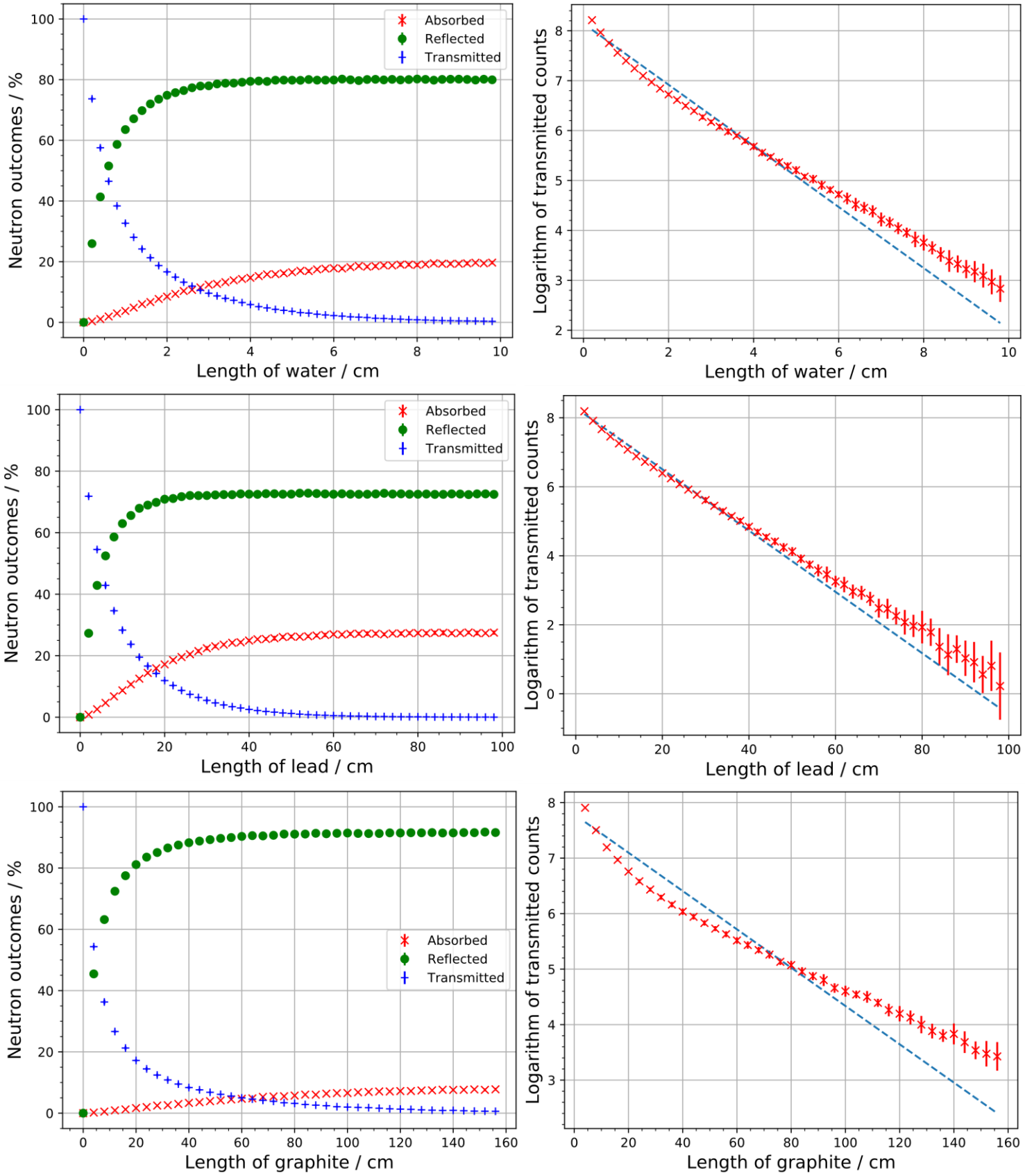

Figure 6: Graphs of neutron outcome percentage on the left column and linearized transmitted counts on the right. The top 2 graphs are for water, middle 2 for lead and bottom 2 for graphite. Error bars may be too small to be seen.

4.3 Simulating the Woodcock method

The Woodcock method was simulated for all 2 slab combinations. For each combination, 100 repeats of 5000 neutrons were run to get the percentages of neutron outcomes. The data is shown in Table 3 below.

Materials: 1st - 2nd	**Absorbed / %**	**Reflected / %**	**Transmitted / %**
Water - Lead	19.7 ± 0.5	80.1 ± 0.5	0.158 ± 0.061
Water - Graphite	19.7 ± 0.5	80.0 ± 0.5	0.265 ± 0.064
Lead - Water	26.2 ± 0.6	73.6 ± 0.6	0.163 ± 0.054
Lead - Graphite	12.12 ± 0.5	68.5 ± 0.7	19.3 ± 0.5
Graphite - Water	17.6 ± 0.6	82.1 ± 0.6	0.273 ± 0.085
Graphite - Lead	8.85 ± 0.39	71.9 ± 0.6	19.2 ± 0.5

Table 3: Data of neutron outcomes from Woodcock simulations.

From this data it was concluded that the best moderator combination to surround a fission reactor core was graphite – water, graphite on the inside followed by a layer of water. This was because it had the highest reflection percentage, 82.1 %, and one of the lowest transmission percentages. Water – lead had the lowest transmission percentage, but this was only smaller than the percentage for graphite – water by 0.11 %. The action of the fictious steps were clearly visible on the random walks through the materials as shown for graphite in Fig. 7.

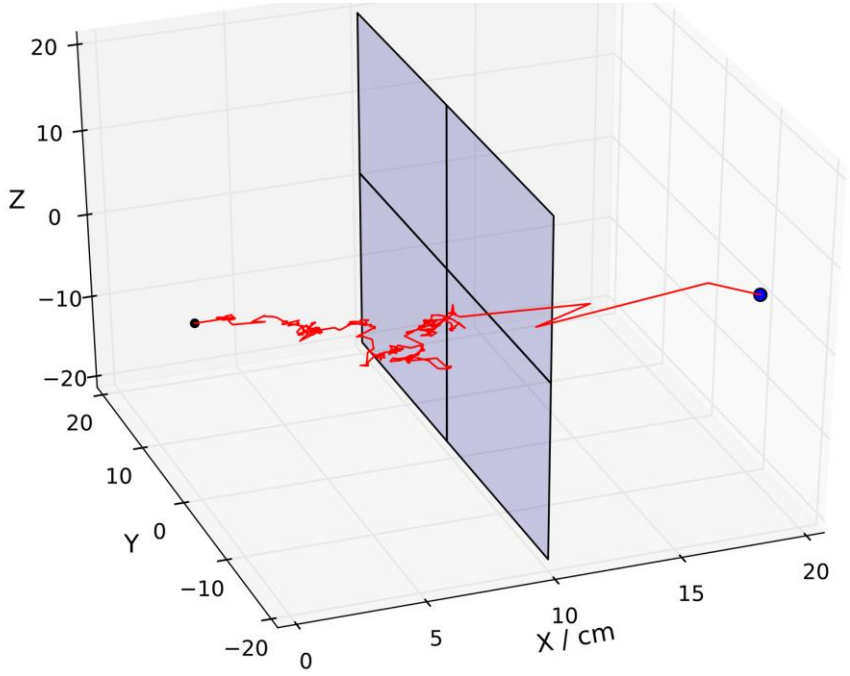

Figure 7: A plot of a random walk through two different materials with the Woodcock method applied. The moderator boundary is shown at X = 10cm. Left of this was water, right of this was graphite. Entry point was the small black dot, the neutron outcome was a transmission shown by the blue dot.

From Fig. 7, Water had a smaller mean free path than graphite and, as soon as the neutron crossed the material boundary into graphite, fictious steps occurred in that region.

5. Further Discussion

Monte Carlo methods proved effective at simulating neutron penetration through different materials. The percentage outcomes of neutrons through 10 cm of each moderator and the results from the Woodcock method clearly show which materials give the highest rates of reflection and lowest rates of transmission. It can be argued that an ideal moderator would reflect as many neutrons as possible and transmit negligible amounts; keeping the neutrons inside to undergo further fission while shielding the surroundings from high energy neutrons. Additionally, in literature 100 collisions with nuclei is often quoted as the approximate number required to reduce a high energy neutron to a thermal neutron [1]. This means a moderator with a small mean free path ensures the neutron undergoes enough collisions. From the simulation it was concluded that, per cm, water was the best moderator as it had the lowest transmission rates and highest reflectivity, however this does vary as moderator length increases. A more optimized setup would be to use an inner graphite and outer water layer to give higher reflectivities and negligible transmission.

The fitted graphs used to determine the attenuation length in Fig. 6 show that the model was not the best fit of the data. This was represented by the reduced chi-squares and the visible data deviation from the fitted lines. To get better fits, the method would need to be extended to include more interactions, not just scattering and absorption. Other possible interactions include phonon scattering with solid lattice structures or collisions with other penetrating neutrons [7]. Another assumption of the employed method was that each neutron was already thermal, and the simulated collisions were elastic. In actual materials, some collisions would be inelastic, so neutron energy may not remain constant after every collision.

6. Conclusion

The attenuation lengths of water, lead and graphite were (1.89 ± 0.05), (11.4 ± 0.2) and (30.6 ± 1.6) cm, with reduced chi-squares of 12.7, 8.7 and 41.6, respectively. These inconsistences were attributed to the linear fits being a poor model of the data since the employed method only considered absorption and elastic scattering.

References

[1] Gadioli E. and Hodgson P. E., Introductory Nuclear Physics, Oxford University Press, 1997.

[2] Beckhurts K. H. and Wirtz K., Neutron Physics, Berlin: Springer-Verlag, 1958.

[3] Feynman R., “Diffusion, Vol 1 Chp 43,” in *The Feynman Lectures on Physics*, Narosa Publishing House, 1963.

[4] Newman M., Computational Physics, Michigan, 2013.

[5] Marsaglia G., “Random Numbers Fall Mainly in the Planes,” *Proceedings of the National Academy of the United States of America,* no. 61, pp. 25-28, 1968.

[6] Woodcock E. R. et al., “Techniques used in the GEM codes for Monte Carlo Neutronics calculations in reactors and other systems of complex geometry,” Argonne National Laboratory , 1965.

[7] Lovesey S., Theory of Neutron Scattering from Condensed Matter, Oxford: Clarendon Press, 1984.

Python Code attached as Appendix

```
# -*- coding: utf-8 -*-
"""
@author: Adam Coxson, Undergrad,
School of Physics and Astronomy, University of Manchester.
PHYS20762 Computational Physics
Project 3 - Penetration of Neutrons through shielding
Submitted - 05/2019


The Project task is to simulate the absorption and scattering  of
thermal neutrons using Monte Carlo techniques. This is done for slabs of
materials of known thicknesses.

"""
""" IMPORTS """
import numpy as n  # NumPy for array manipulation
import matplotlib as mpl # matlab plotting
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # 3D plots
from random import random # Random numbers

mpl.rc('axes', labelsize = 14) # set matplotlib graph axes labels to size 14 font
mpl.rc('axes', titlesize = 14) # set titles to 14


""" FUNCTIONS """


def randssp(p,q):
    """ Function to generator Pseudo-random numbers from a linear congruential
    generator LCG. These random numbers suffer from hyperplane problem.
    """
    #global m, a, c, x
    m = pow(2, 31)    # Integer constants
    a = pow(2, 16) + 3
    c = 0
    x = 123456789 # some initial number

    try: p            # Ensuring inputted arguments are valid
    except NameError:
        p = 1         # Default to 1 if invlaid
    try: q
    except NameError:
        q = p       # Default to p if invalid

    r = n.zeros([p,q]) # Set a numpy array of zeroes the size of wanted array

    for l in range (0, q): # looping over elements of arrays for rand nums
        for k in range (0, p):
            x = n.mod(a*x + c, m) # iterative equation to generate numbers
            r[k, l] = x/m        # Divide by the mod to normalise
    return r; # Return set of random numbers


def ParticleDistance(MeanFreePath):
    """ Takes in a mean free path, generates a random probablity which is then
    subject to an inverse exponential distribution. This works to generate a
    value whose magnitude is determined by the mean free path and the
    exponential distribution. Returns this value as the length travelled
    """
    RandNum = n.random.uniform(0, 1) # Generate random number between 0 and 1
    length = -MeanFreePath*n.log(1 - RandNum) # inverse exp subject to MFP
    return length;          # Length respective of exponential distribution


def GetDirectionSphere():
    """ Function to generate 3 cartesian unit vectors. It then uses these as
    the components to calculate a length. Each unit vector is normalised
    w.r.t the length to constrain them to a unit sphere. This enables a point
```

```
    which would replicate the surface of a sphere is iterated many times.
    The normalised unit vectors are returned to the program for further use.
    I.e. acts as a random way of choosing direction in 3D position space.
    """
    valid = False
    while valid == False:
        # Generating a uniform random number between -1 and 1
        RandomNum = n.random.uniform(-1, 1) # Generating a uniform random number between -1 and 1
        x = RandomNum                    # Assigning this as an x-coordinate
        RandomNum = n.random.uniform(-1, 1)
        y = RandomNum                    # y - coordinate
        RandomNum = n.random.uniform(-1, 1)
        z = RandomNum                    # z - coordinate
        RandomNum=n.random.uniform(0, 1)    # Random number between 0 and 1
        kill = RandomNum                 # Random number

        LengthModulus = n.sqrt(x**2 + y**2 + z**2) # Finding the length of vector
        if LengthModulus <= 1:      # Constraining length to a unit Sphere
            valid=True              # If within unit sphere, break loop
            Modx= x / LengthModulus # Normalise each coordinate
            Mody= y / LengthModulus # this ensures each one represents a component of a
            Modz= z / LengthModulus # unit vector

    return Modx, Mody, Modz, kill; # Return components of unit vector and absorption probability


def SpherePlot(NumOfPoints, MeanFreePath):
    """ Function to plot uniformly distributed points to generate a spherical surface.
    This is just for visualisation purposes. Spheres of 500 points or less are quick
    to respond but look sparse, anymore can cause stutter in interactive plots.
    """
    figSP = plt.figure()                # Graph plotting setup
    bx = figSP.add_subplot(121, projection='3d')# A single 3D graph
    for i in range(0,NumOfPoints):          # Loop for desired points
        x, y, z, temp = GetDirectionSphere()    # Find unit vector components
        bx.scatter(x,y,z, c='b')            # add unit vector to plot
        bx.set_xlabel('X')              # Graph labels
        bx.set_ylabel('Y')
        bx.set_zlabel('Z')

    bx2 = figSP.add_subplot(122, projection='3d')# A single 3D graph
    for i in range(0, NumOfPoints):         # Loop for desired points
        x, y, z, temp = GetDirectionSphere()    # Find unit vector components
        dist = ParticleDistance(MeanFreePath)
        x, y, z = x*dist, y*dist, z*dist
        bx2.scatter(x,y,z, c='b')           # add unit vector to plot
        bx2.set_xlabel('X')             # Graph labels
        bx2.set_ylabel('Y')
        bx2.set_zlabel('Z')
    return 0;


def UniformBoxComparison(NumOfRandNums):
    """ This function generates and plots 3D boxes of random numbers. Two
    methods are used. One is uniform random numbers. The other random numbers
    are generated from the Linear Congruential Generator method. These plots
    allow a comparison for hyperplanes to be identifed in the latter method.
    """

    RandData = n.random.uniform(size=(3,NumOfRandNums)) # Create 3 sets of  random numbers
    Temp = randssp(3,NumOfRandNums)         # Using LCG func. create 3 sets rand num
    fig = plt.figure()          # Setup for plotting

    ax = fig.add_subplot(121, projection='3d') # Setup for 3D plotting
    ax.scatter(Temp[0], Temp[1], Temp[2])     # 3D scatter plot with hyperplanes
    ax.set_xlabel('X')              # Graph labels
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.show()
```

```
    ax2 = fig.add_subplot(122, projection='3d') # Add a second 3D plot
    ax2.scatter(RandData[0], RandData[1], RandData[2]) # Plot using uniform data
    ax2.set_xlabel('X')
    ax2.set_ylabel('Y')
    ax2.set_zlabel('Z')
    plt.show()
    return 0;


def ExponentialHistogram(lam, NumOfRandNums):
    """ Sorts a set of random numbers into a random exponential distribution
    using the inverse Cumulative distribution function.
    """
    x = n.random.uniform(size=(NumOfRandNums,1)) # Set of random numbers
    s = -1*lam*n.log(1-x) # inverse exponential distribution

    fig3 = plt.figure()        # Setting up 3D plot preliminaries
    ax5 = fig3.add_subplot(111)
    N, bins, patches = ax5.hist(s, bins = 50) # histogram
    plt.xlabel('Distance travelled in water/ cm' )
    plt.ylabel('Number of neutrons, N')
    plt.xlim(0,300)           # Bins are too small past 300cm
    plt.minorticks_on()
    plt.show()

    bins2 = n.diff(bins)/2 + n.delete(bins,-1) # Finding bin midpoints
    Histarray = n.array([bins2, N], dtype = n.float64)
    newHist = Histarray[:,Histarray[1]!=0] # removing zeroes from N

    # Polyfit of logged data to get the attenuation length without scattering
    p,cov = n.polyfit(newHist[0], n.log(newHist[1]), 1, cov=True)
    fit = n.polyval(p, newHist[0])

    plt.figure() # Plotting the fitted data on a logarithm graph
    plt.plot(newHist[0],fit) # fit line
    plt.errorbar(newHist[0],n.log(newHist[1]), yerr = 1/n.sqrt(newHist[1]),
                 fmt= 'rx')
    plt.xlabel('Distance travelled in water/ cm' )
    plt.ylabel('ln(N)')
    plt.minorticks_on()
    plt.grid(True)
    plt.show()

    AttenuationLength =  -1/p[0] # Return the attenuation length from the data
    AttenError = ((n.sqrt(cov[0][0]))/p[0])*(1/p[0]) # error from covariant
    print("Attenuation length:", AttenuationLength, u"\u00B1", AttenError)

    return AttenuationLength, AttenError;


def ProgressCheck(CurrentValue, MaxEndValue):
    """ Crude function to output the current stage of a process. Useful for
    looping code that can take a long time to execute. Can get a cuppa in the
    mean-time, or do some tutorial sheets. This doesn't work properly if
    the Current Value isn't roughly 25%, 50% or 75%.
    """
    if CurrentValue == round(MaxEndValue/4): # If 1/4 of final value
        print("\n25% complete")
    elif CurrentValue == round(MaxEndValue/2): # 1/2
        print("50% complete")
    elif CurrentValue == round(3*MaxEndValue/4): # 3/4
        print("75% complete")
    elif CurrentValue == round(MaxEndValue): # All done!
        print("100% complete :D\n")
    else:
        pass
    return 0;
```

```
def BinNumberComparison(lamda, NumOfRandNums):
    """ This function generates random numbers according to an exponential
    distribution. It then sorts these into a histogram and fits the logarithm.
    The function loops through an array of different bin numbers so the effect
    of bin number upon the accuracy of the fit can be quantified.
    """
    PathLength = n.array([]) # Empty array for storing bin counts
    PathError = n.array([])         # Array for length errors
    binSelection = n.arange(5,100,5) # Array of diffrent bin counts
    #fig = plt.figure()
    #ax = fig3.add_subplot(111) # Plotting variables for the histogram

    for i in range(0,len(binSelection)): # Looping over the different bins
        #for j in range(0,100):      # Iterations for a mean length to be taken

        x = n.random.uniform(size=(NumOfRandNums,1)) # Set of Random values
        s = -1*lamda*n.log(1-x)  # Applying an inverse exponential distribution
        N, bins = n.histogram(s, bins = binSelection[i], normed=True) # Histogram


        bins2 = n.diff(bins)/2 +  n.delete(bins,-1) # Finding the midpoint of each bin
        Histarray = n.array([bins2, N], dtype = n.float64) # forming new arrays of floats
        newHist = Histarray[:,Histarray[1]!=0]      # Removing zeroes from N

        # fitting bins and their populaions
        p,cov = n.polyfit(newHist[0], n.log(newHist[1]), 1, cov=True)
        PathLength = n.append(PathLength, -1/p[0]) # Repeated lengths for current bin
        PathError = n.append(PathError, ((n.sqrt(cov[0][0]))/p[0])*(1/p[0]))

    print("Bin number comparison: bins, attenuation length in water, error")
    print(n.dstack([binSelection, PathLength, PathError]))
    return PathLength, PathError;

def Find_Residual(yfit, y, yErr):
    """  # Finds residuals for a varaible after a fitting procedure.
    Also produces a chi-square. Gives dy and deltaf for residual plots
    """
    residual = dy = y - yfit
    deltaf = n.sqrt(n.sum(n.power(dy, 2)) / (len(yfit)-3))
    X2chisqR = n.sum((dy/yErr)**2)/(len(yfit)-2)
    return  residual, deltaf, X2chisqR;


def AttentuationLength(lengths, counts, countErr):
    """ Determines attenuation length for logged data by removing any zeroes
    which are NaN and then polyfits. Also produces a reduced chi-square.
    """
    # Remove the value at 100% transmission, it skews the fit
    ShortCounts, ShortLengths = n.delete(counts, 0) , n.delete(lengths, 0)
    ShortErr =  n.delete(countErr, 0)
    # Removing the zeros
    TempCounts = n.array(ShortCounts[ShortCounts != 0])
    TempLengths = n.array(ShortLengths[ShortCounts != 0])
    FracCountErr = (n.array(ShortErr[ShortCounts != 0])/TempCounts)
    weight = 1/(FracCountErr)# Weighting for polyfit
    # Fitting  the logged count data
    p,cov = n.polyfit(TempLengths, n.log(TempCounts), 1, w = weight,  cov=True)
    #p,cov = n.polyfit(TempLengths, n.log(TempCounts), 1,  cov=True)
    fit = n.polyval(p,TempLengths)

    AttenLength = -1/p[0] # Getting the attenuation length from co-efficient
    AttenError = ((n.sqrt(cov[0][0]))/p[0])*(1/p[0]) # error on co-efficient
    # Finding a chi-square
    dy,df, X2chiR = Find_Residual(fit, n.log(TempCounts),FracCountErr)

    return TempLengths, AttenLength, AttenError, fit, X2chiR;
```

```
def SimulateNeutron(MaxLength, AbsorbArea, ScatterArea, density, mass, toRecord):
    """ Simulates a neutron, or neutral particle, travelling through a material
    this is achieved by considering interaction cross sections and the mean
    free paths. Particle outcomes and their paths are recorded.
    """
    ATotal = AbsorbArea + ScatterArea            # Total area, scatter and absorption
    A_N = 6.022141            # Avagrados constant e23 is accounted for in density
    meanFreePath = mass*10 / (density*ATotal*A_N) # Mean free path from macro/mirco areas
    # Probability of absorption defined by scatter/absorb area ratios
    absorbProb = AbsorbArea / ATotal

    x = y = z = 0          # Initial co-ords at penetration point

    xHistory = n.array([x]) # Setting up the variables to record particle history
    yHistory = n.array([y])
    zHistory = n.array([z])
    scatterCount = 0       # Recording number of times scattered
    alive = True           # Condition for particle still active in medium
    absorbed = reflected = transmitted = False # Simulation end conditions

    while alive:               # While particle still active
        vx, vy, vz, kill = GetDirectionSphere()  # Generate random unit vector components
        dist = ParticleDistance(meanFreePath) # Distance from exponential distribution

        if scatterCount == 0: # For the very first step
            vx, vy, vz = 1, 0, 0 # Paticle incident normal to medium boundary (along x)
            dist = ParticleDistance(meanFreePath) # Dist from exponential distribution
            kill = 1           #  Particle will not be absorbed right at the start

        scatterCount += 1    # Next step counter
        x += vx*dist         # update particle position
        y += vy*dist
        z += vz*dist

        if toRecord:       # Save positions for particle history
            xHistory = n.append(xHistory, x)
            yHistory = n.append(yHistory, y)
            zHistory = n.append(zHistory, z)

        transmitted = x > MaxLength  # Particle position, transmitted through medium
        reflected = x < 0            # Particle scattered back out the medium
        # If particle still inside block, check absorbed condition
        if not( transmitted or reflected ):
            absorbed = absorbProb > kill    # If absorbed
        alive = not(absorbed or transmitted or reflected) # ah-ah-ah-ahh, Stayin' Alive
        # Particle history 2D array of x,y,z
    PathHistory = n.array([xHistory, yHistory, zHistory])

    return absorbed, reflected, transmitted, PathHistory;


def MultiNSim(MaxLength, AbsorbArea, ScatterArea, density, mass, NumOfNeutrons,
              SimRuns, toShow):
    """ This function applies the neutron simulation but then loops it
    for a specified number of runs each of so many neutrons. Finds the means
    and standard deviations and outputs to user. Also can plot random walks.
    """
    if toShow == True:
        figW = plt.figure()   # Figure Setup
        wx=Axes3D(figW)
        wx.set_xlabel("Penetration distance X / cm")
        wx.set_ylabel("Y")
        wx.set_zlabel("Z")

    Absorbed, Reflected, Transmitted = [],[],[] # Empty lists

    for j in range(0,SimRuns):
        AbsorbCount = ReflectCount = TransmitCount = 0
        for i in range(0,NumOfNeutrons):
```

```
            # Calling neutron simulation
            absorb, reflect, transmit, Path = SimulateNeutron(MaxLength,
                        AbsorbArea, ScatterArea, density, mass,True)
            AbsorbCount += int(absorb) #  Counting up the neutron results
            ReflectCount += int(reflect)
            TransmitCount += int(transmit)
            # Plot 5 random walks from first run of neturons
            if i % (NumOfNeutrons/5) == 0 and j == 1 and toShow == True:

                wx.plot(Path[0],Path[1],Path[2]) # plotting coords for walk

        Absorbed.append(AbsorbCount)      # Total neutron outcomes
        Reflected.append(ReflectCount)    # Each element is for one length
        Transmitted.append(TransmitCount)
        Ab, Ref, Tr = n.array([Absorbed]), n.array([Reflected]), n.array([Transmitted])
        if toShow == True:
            ProgressCheck(j, SimRuns)

    # Sorting the means and their errors into arrays for return outputs
    CountData = n.array([n.mean(Ab), n.std(Ab),n.mean(Ref), n.std(Ref),
                    n.mean(Tr), n.std(Tr)])
    CountDataPerc = (100*CountData)/NumOfNeutrons # Percentage conversion

    # Particle simulation end result outputs
    if toShow == True:
        # From a single run
        print("\nFrom a single run of",NumOfNeutrons,
                    "neutrons. Errors are statistical, root of counts")
        print("Absorbed:",(100*AbsorbCount)/NumOfNeutrons , u"\u00B1",
            n.round((100*n.sqrt(AbsorbCount))/NumOfNeutrons, 4), "%")
        print("Reflected:",(100*ReflectCount)/NumOfNeutrons , u"\u00B1",
            n.round((100*n.sqrt(ReflectCount))/NumOfNeutrons, 4), "%")
        print("Transmitted:",(100*TransmitCount)/NumOfNeutrons , u"\u00B1",
            n.round((100*n.sqrt(TransmitCount))/NumOfNeutrons, 4), "%")
        # Comparing data from many runs
        print("\nFrom averaging",SimRuns ,"runs of",NumOfNeutrons,"neutrons.")
        print("Absorbed:",CountDataPerc[0] , u"\u00B1",n.round(CountDataPerc[1], 4), "%")
        print("Reflected:",CountDataPerc[2] , u"\u00B1",n.round(CountDataPerc[3], 4), "%")
        print("Transmitted:",CountDataPerc[4] , u"\u00B1",n.round(CountDataPerc[5], 4), "%")

    return CountData, CountDataPerc;


def VariedLengthSim(Lengths, AbsorbArea, ScatterArea, density, mass,
                                    NumOfNeutrons, SimRuns):
    """ This function runs the neutron simulation multiple times for different
    lengths and gets the total counted outcomes of the neutrons for each length.
    The data is then passed into a func. that polyfits and finds the
    attentuation length w.r.t. neutron transmission.
    """
    Absorbed, Reflected, Transmitted = [],[],[] # Empty lists
    AbErr, RfErr, TrErr = [],[],[] # Empty lists

    toShow =  False # Not plotting, don't need path histories
    for i in range(0,len(Lengths)):       # Loop over different lengths
        CountsTemp, CountPercTemp = MultiNSim(Lengths[i], AbsorbArea,
                ScatterArea, density, mass, NumOfNeutrons, SimRuns, toShow)

        Absorbed.append(CountsTemp[0])     # Means of neutron outcomes
        AbErr.append(CountsTemp[1])        # Errors of neutron outcomes
        Reflected.append(CountsTemp[2])
        RfErr.append(CountsTemp[3])
        Transmitted.append(CountsTemp[4])

        if CountsTemp[4] == NumOfNeutrons: # Add 0.1 as error for full
            TrErr.append(0.1)              # transmission to stop divide by 0
        else:
            TrErr.append(CountsTemp[5]) # Add std as per usual
```

```
        ProgressCheck(Lengths[i], Lengths[-1]) # output of progress for debug

    counts = n.array([Absorbed, AbErr, Reflected, RfErr, Transmitted, TrErr])
    # Converting counts to percentages
    percentages = (100/NumOfNeutrons)*(counts)

    # Finding attenuation length from transmitted neutrons
    NewLengths, AttenLength, AttenErr, fit, X2chiR = AttentuationLength(Lengths,
                                        counts[4], counts[5])
    AttenuationData = n.array([NewLengths, fit, AttenLength, AttenErr, X2chiR])

    print("Attenuation length:", AttenLength, u"\u00B1", AttenErr,
          "Reduced χ2:", X2chiR)
    return counts, percentages, AttenuationData;


def PercentageErrorComparisons(MaxLength, AbsorbArea, ScatterArea, density,
                                        mass, SimRuns):
    """ function to compare the effect of simulating more neutrons on error.
    This can be used to identify the smallest possible neutron numbers for
    fast computation time while retaining enoguh accuracy.
    """
    VariednNum = n.arange(500,10000,500) # neutron numbers to loop through
    AbsorbCountErrVN,ReflCountErrVN,TransCountErrVN = [],[],[] # empty lists
    AbsorbErrVN,ReflErrVN,TransErrVN = [],[],[]

    for i in range (0, len(VariednNum)): # looping over number of neutrons
        ProgressCheck(i, len(VariednNum))
        # Running multi sim toget means and std for current NumOfNeutrons
        Counts_W2, Percents_W2 = MultiNSim(MaxLength, AbsorbArea, ScatterArea,
                        density, mass, VariednNum[i], SimRuns, False)

        # Appending the count errors and mean std errors
        AbsorbCountErrVN.append(n.sqrt(Counts_W2[0])/VariednNum[i])
        AbsorbErrVN.append(Counts_W2[1]/VariednNum[i])
        ReflCountErrVN.append(n.sqrt(Counts_W2[2])/VariednNum[i])
        ReflErrVN.append(Counts_W2[3]/VariednNum[i])
        TransCountErrVN.append(n.sqrt(Counts_W2[4])/VariednNum[i])
        TransErrVN.append(Counts_W2[5]/VariednNum[i])

    plt.figure() # Error percentages using standard deviation of means
    plt.plot(VariednNum, 100*n.array(AbsorbErrVN),'rx', label = 'Absorbed')
    plt.plot(VariednNum, 100*n.array(ReflErrVN),  'go', label = 'Reflected')
    plt.plot(VariednNum, 100*n.array(TransErrVN), 'b+', label = 'Transmitted')
    plt.title('Effect of neutron number upon error: Standard deviation')
    plt.xlabel('Number of neutrons simulated' )
    plt.ylabel('Error on counts / %')
    plt.minorticks_on()
    plt.grid(True)
    plt.legend()
    plt.show()

    plt.figure() # Error percentages using root of counts
    plt.plot(VariednNum, 100*n.array(AbsorbCountErrVN),'rx', label = 'Absorbed')
    plt.plot(VariednNum, 100*n.array(ReflCountErrVN),  'go', label = 'Reflected')
    plt.plot(VariednNum, 100*n.array(TransCountErrVN), 'b+', label = 'Transmitted')
    plt.title('Effect of neutron number upon error: Statistical expectation')
    plt.xlabel('Number of neutrons simulated' )
    plt.ylabel('Error on counts / %')
    plt.minorticks_on()
    plt.grid(True)
    plt.legend()
    plt.show()

    return 0;


def WoodCockSim(Length, Areas, density_arg, mass_arg, toRecord):
    """ Function to simulate neutron penetration through two adjacents blocks
```

```
of different material. This uses the Woodcock method rather than usual
geometrical considerations to define how the neutrons should interact
wtih the material. Note material 1 is defined as the material the neutron
first enters.
"""
 # Defining the lengths from start to slab boundary and out the 2nd block
Boundary, MaxLength = Length[0],Length[1]
# Redefining the areas, masses and densities for each material from args
AbsorbArea1, AbsorbArea2 = Areas[0],Areas[1]
ScatterArea1, ScatterArea2 = Areas[2],Areas[3]
mass1, mass2 = mass_arg[0], mass_arg[1]
density1, density2 = density_arg[0], density_arg[1]

ATotal1 = AbsorbArea1 + ScatterArea1  # Total area = scatter and absorption
ATotal2 = AbsorbArea2 + ScatterArea2  # for the 1st and 2nd blocks
A_N = 6.022141                        # Avagrados constant

absorbProb1 = AbsorbArea1 / ATotal1 # Probability of absorption
absorbProb2 = AbsorbArea2 / ATotal2 # defined by scatter/absorb area ratios
Max_ATotal = max([ATotal1, ATotal2]) # Finding max out of the 2 values

meanFreePath1 = mass1*10 / (density1*ATotal1*A_N)
meanFreePath2 = mass2*10 / (density2*ATotal2*A_N)
Min_MFP = min([meanFreePath1, meanFreePath2])

FictRegion1 = 1 - ATotal1/Max_ATotal # Defining fictious probabilities
FictRegion2 = 1 - ATotal2/Max_ATotal # = 0 for region with applied MFP


x = y = z = 0              # Initial co-ords at penetration point
xHistory = n.array([x])    # Setting up the variables to record particle history
yHistory = n.array([y])
zHistory = n.array([z])
scatterCount = 0           # Recording number of times scattered
alive = True               # Condition for particle still active in medium
absorbed = reflected = transmitted = False # Simulation end conditions

while alive:               # While particle still active

    if x >= Boundary:     # If particle has entered 2nd material from 1st
        FictRegion = FictRegion2 # redefine current fict. prob.
        absorbProb = absorbProb2 # and absorbtion prob.
        #print("Region: 2", x)
    else:
        FictRegion = FictRegion1 # If particle is in the 1st material
        absorbProb = absorbProb1
        #print("Region: 1")

    fictProb = n.random.uniform(0, 1) # Create a probability for this loop
     # If satisfied, neutron will scatter and change direction
    if fictProb > FictRegion and scatterCount != 0: # take fictious step
        # Generate random unit vector components
        vx, vy, vz, kill = GetDirectionSphere()
        scatterCount += 1    # Next step counter

    dist = ParticleDistance(Min_MFP) # Distance from exponential distribution

    if scatterCount == 0: # For the very first step
        vx, vy, vz = 1, 0, 0 # Paticle incident normal to medium boundary (along x)
        dist = ParticleDistance(Min_MFP) # Dist from exponential distribution
        kill = 1           #  Particle will not be absorbed right at the start
        scatterCount += 1 # Ensures prev if statement can activate

    x += vx*dist  # update particle position
    y += vy*dist  # If fictprob < FictRegion(prob) then particle hasn't
    z += vz*dist  # changed it's direction, fictious step taken

    if toRecord:       # Save positions for particle history
        xHistory = n.append(xHistory, x)
```

```
            yHistory = n.append(yHistory, y)
            zHistory = n.append(zHistory, z)

        transmitted = x > MaxLength # Particle transmitted through medium
        reflected = x < 0           # Particle scattered back out the medium
        # If particle still inside block, check absorbed condition
        if not( transmitted or reflected ):
            absorbed = absorbProb > kill    # If absorbed
        alive = not(absorbed or transmitted or reflected)

        # Particle history 2D array of x,y,z
    PathHistory = n.array([xHistory, yHistory, zHistory])

    # Return count totals and particle path
    return absorbed, reflected, transmitted, PathHistory;

def MultiWoodCock(Length, Areas, densities, masses, NumOfNeutrons,SimRuns):
    """ This function applies the Woodcock simulation but then loops it
    for a specified number of runs each of so many neutrons. Finds the means
    and standard deviations and outputs to user
    """
    toRecord = False # Plotting is done separately from this function
    Absorbed, Reflected, Transmitted = [],[],[] # Empty lists

    for j in range(0,SimRuns):
        AbsorbCount = ReflectCount = TransmitCount = 0
        for i in range(0,NumOfNeutrons):
            # Calling neutron woodcock simulation
            absorb, reflect, transmit, Path = WoodCockSim(Length, Areas,
                                    densities, masses, toRecord)
            AbsorbCount += int(absorb) #  Counting up the neutron results
            ReflectCount += int(reflect)
            TransmitCount += int(transmit)

        Absorbed.append(AbsorbCount)      # Total neutron outcomes
        Reflected.append(ReflectCount)    # Each element is for one length
        Transmitted.append(TransmitCount)
        Ab, Ref, Tr = n.array([Absorbed]), n.array([Reflected]), n.array([Transmitted])

    # Sorting the means and their errors into arrays for return outputs
    CountData = n.array([n.mean(Ab), n.std(Ab),n.mean(Ref), n.std(Ref),
                         n.mean(Tr), n.std(Tr)])
    CountDataPerc = (100*CountData)/NumOfNeutrons # Percentage conversion

    # Particle simulation end result outputs
    print("\nFrom averaging",SimRuns ,"runs of",NumOfNeutrons,"neutrons.")
    print("Absorbed:",CountDataPerc[0] , u"\u00B1",n.round(CountDataPerc[1], 4), "%")
    print("Reflected:",CountDataPerc[2] , u"\u00B1",n.round(CountDataPerc[3], 4), "%")
    print("Transmitted:",CountDataPerc[4] , u"\u00B1",n.round(CountDataPerc[5], 4), "%")

    return CountData, CountDataPerc;



""" MAIN """

print("\n---------- PREPARTORY CODE ----------\n")
NumOfRandNums, NumOfRandNums2 = 1000, 1000000
WaterMFP = 45 # For steps according to exponential

# Plot 2 3D boxes, one with hyperplanes, one without to show the effect of LCG
UniformBoxComparison(NumOfRandNums)
# Plot a sphere surface of uniform points
SpherePlot(NumOfRandNums, WaterMFP)
# Comparing the effect of bin number upon attentuation length
BinNumberComparison(WaterMFP,  NumOfRandNums2)
# Plotting histogram used to determine attenuation length
ExponentialHistogram(WaterMFP,  NumOfRandNums2)
```

```
""" VARIABLES """
# neutrons to simulate as determined by the error comparisons
NumOfNeutrons, SimRuns = 5000, 100
NumOfNeutrons2, SimRuns2 = 200, 10 # for debug/fast, remove number 2
MaxLength   =  10 # 10 cm of materials

# Properties of Water
AbsorbArea_W  = AbAW =  0.6652
ScatterArea_W = ScAW = 103.0
density_W     = DW   = 1.0
mass_W        = MW   = 18.01528

# Properties of Lead
AbsorbArea_L  = AbAL = 0.158
ScatterArea_L = ScAL = 11.221
density_L     = DL   = 11.35
mass_L        = ML   = 207.2

# Properties of Graphite
AbsorbArea_G  = AbAG = 0.0045
ScatterArea_G = ScAG = 4.74
density_G     = DG   = 1.67
mass_G        = MG   = 12.011

print("\n---------- WATER ----------\n")
# Properties of Water

VariedLength_W = VL_W = n.arange(0, 10, 0.2)

# Error comparions for varied neutron numbers
PercentageErrorComparisons(MaxLength, AbsorbArea_W, ScatterArea_W,
                           density_W, mass_W, SimRuns)

# Neutron simulations for 10cm of water
Counts_W, Percents_W = MultiNSim(MaxLength, AbsorbArea_W, ScatterArea_W,
                           density_W, mass_W, NumOfNeutrons, SimRuns, True)

# Varying length of moderator
ARTcounts_W, ARTpercents_W, AttenData_W = VariedLengthSim(VL_W, AbsorbArea_W,
                    ScatterArea_W, density_W, mass_W, NumOfNeutrons, SimRuns)

plt.figure() # Neutron outcomes in water
plt.errorbar(VariedLength_W, ARTpercents_W[0], yerr = ARTpercents_W[1],
                                        fmt='rx', label = 'Absorbed')
plt.errorbar(VariedLength_W, ARTpercents_W[2], yerr = ARTpercents_W[3],
                                         fmt='go', label = 'Reflected')
plt.errorbar(VariedLength_W, ARTpercents_W[4], yerr = ARTpercents_W[5],
                                        fmt='b+', label = 'Transmitted')
#plt.title('Neutron outcomes for increasing length of water')
plt.xlabel('Length of water / cm' )
plt.ylabel('Neutron outcomes / %')
plt.minorticks_on()
plt.grid(True)
plt.legend()
plt.show()

# Removing the first element at 100% since it skews the logarithm fit
VariedLength2_W = n.delete(VariedLength_W, 0)# removing initial
Tcounts2_W, TcountErrs_W = n.delete(ARTcounts_W[4], 0), n.delete(ARTcounts_W[5], 0)

plt.figure() # Neutron transmissions, logged, fitted and now graphed
plt.errorbar((VariedLength2_W), n.log(Tcounts2_W), yerr= (TcountErrs_W/Tcounts2_W),
                                                fmt = 'rx')
plt.plot(AttenData_W[0], AttenData_W[1], '--') # Fitted data from polyfit and corresponding length■
■s
#plt.title('Log graph of neutron transmissions in water')
plt.xlabel('Length of water / cm' )
plt.ylabel('Logarithm of transmitted counts')
plt.minorticks_on()
```

```
plt.grid(True)
plt.show()


print("\n---------- LEAD ----------\n")

VariedLength_L = VL_L = n.arange(0, 100, 2)

# Error comparions for varied neutron numbers
PercentageErrorComparisons(MaxLength, AbsorbArea_L,
                        ScatterArea_L, density_L, mass_L, SimRuns)

# Neutron simulatinos for 10cm of lead
Counts_L, Percents_L = MultiNSim(MaxLength, AbsorbArea_L,
                ScatterArea_L, density_L, mass_L, NumOfNeutrons, SimRuns, True)

# Varying length of moderator
ARTcounts_L, ARTpercents_L, AttenData_L = VariedLengthSim(VL_L, AbsorbArea_L,
                    ScatterArea_L, density_L, mass_L, NumOfNeutrons, SimRuns)

plt.figure() # Neutron outcomes in lead
plt.errorbar(VariedLength_L, ARTpercents_L[0], yerr = ARTpercents_L[1],
                                        fmt='rx', label = 'Absorbed')
plt.errorbar(VariedLength_L, ARTpercents_L[2], yerr = ARTpercents_L[3],
                                        fmt='go', label = 'Reflected')
plt.errorbar(VariedLength_L, ARTpercents_L[4], yerr = ARTpercents_L[5],
                                       fmt='b+', label = 'Transmitted')
#plt.title('Neutron outcomes for increasing length of lead')
plt.xlabel('Length of lead / cm' )
plt.ylabel('Neutron outcomes / %')
plt.minorticks_on()
plt.grid(True)
plt.legend()
plt.show()

# Removing the first element at 100% since it skews the logarithm fit
VariedLength2_L = n.delete(VariedLength_L, 0)
Tcounts2_L, TcountErrs_L = n.delete(ARTcounts_L[4], 0), n.delete(ARTcounts_L[5], 0)

plt.figure() # Neutron transmissions, logged, fitted and now graphed
plt.errorbar((VariedLength2_L), n.log(Tcounts2_L), yerr= (TcountErrs_L/Tcounts2_L),
                                                fmt = 'rx')
plt.plot(AttenData_L[0], AttenData_L[1], '--') # Fitted data from polyfit and corresponding length■
■s
#plt.title('Log graph of neutron transmissions in lead')
plt.xlabel('Length of lead / cm' )
plt.ylabel('Logarithm of transmitted counts')
plt.minorticks_on()
plt.grid(True)
plt.show()



print("\n---------- GRAPHITE ----------\n")
# Properties of Graphite

VariedLength_G = VL_G = n.arange(0, 160, 4)

# Error comparions for varied neutron numbers
PercentageErrorComparisons(MaxLength, AbsorbArea_G, ScatterArea_G,
                        density_G, mass_G, SimRuns)

# Neutron simulatinos for 10cm of graphite
Counts_G, Percents_G = MultiNSim(MaxLength, AbsorbArea_G,
                ScatterArea_G, density_G, mass_G, NumOfNeutrons, SimRuns, True)

# Varying length of moderator
ARTcounts_G, ARTpercents_G, AttenData_G = VariedLengthSim(VL_G, AbsorbArea_G,
                    ScatterArea_G, density_G, mass_G, NumOfNeutrons, SimRuns)
```

```
# Plots for varied lenght
plt.figure() # Neutron outcomes in graphite, percentages
plt.errorbar(VariedLength_G, ARTpercents_G[0], yerr = ARTpercents_G[1],
                                        fmt='rx', label = 'Absorbed')
plt.errorbar(VariedLength_G, ARTpercents_G[2], yerr = ARTpercents_G[3],
                                       fmt='go', label = 'Reflected')
plt.errorbar(VariedLength_G, ARTpercents_G[4], yerr = ARTpercents_G[5],
                                     fmt='b+', label = 'Transmitted')
#plt.title('Log graph of neutron transmissions in graphite')
plt.xlabel('Length of graphite / cm' )
plt.ylabel('Neutron outcomes / %')
plt.minorticks_on()
plt.grid(True)
plt.legend()
plt.show()

# Removing the first element at 100% since it skews the logarithm fit
VariedLength2_G = n.delete(VariedLength_G, 0)
Tcounts2_G, TcountErrs_G = n.delete(ARTcounts_G[4], 0), n.delete(ARTcounts_G[5], 0)

plt.figure() # Neutron transmissions, logged, fitted and now graphed
plt.errorbar((VariedLength2_G), n.log(Tcounts2_G), yerr= (TcountErrs_G/Tcounts2_G),
                                                fmt = 'rx')
plt.plot(AttenData_G[0], AttenData_G[1], '--') # Fitted data from polyfit and corresponding length■
■s
#plt.title('Log graph of neutron transmissions in graphite')
plt.xlabel('Length of graphite / cm' )
plt.ylabel('Logarithm of transmitted counts')
plt.minorticks_on()
plt.grid(True)
plt.show()


print("\n---------- WOODCOCK SIMULATIONS ----------")

Lengths = [10,20] # 10cm is boundary between the blocks. 20cm is outside of both
# Variable lists to represent all different material combinations
# W_to_L = Water first block, lead second block

print("\n---------- WATER - LEAD ----------")
AreasW_to_L     = [AbAW,AbAL,ScAW,ScAL] # Absorbtion and scatter areas
massesW_to_L    = [MW, ML]              # Masses
densitiesW_to_L = [DW, DL]              # Densities
MultiWoodCock(Lengths, AreasW_to_L, densitiesW_to_L,massesW_to_L,
            NumOfNeutrons,SimRuns)   # Calling woodcock simulation

print("\n---------- WATER - GRAPHITE ----------")
AreasW_to_G     = [AbAW,AbAG,ScAW,ScAG]
massesW_to_G    = [MW, MG]
densitiesW_to_G = [DW, DG]
MultiWoodCock(Lengths, AreasW_to_G, densitiesW_to_G,massesW_to_G,
            NumOfNeutrons,SimRuns)

print("\n---------- LEAD - WATER ----------")
AreasL_to_W     = [AbAL,AbAW,ScAL,ScAW]
massesL_to_W    = [ML, MW]
densitiesL_to_W = [DL, DW]
MultiWoodCock(Lengths, AreasL_to_W, densitiesL_to_W,massesL_to_W,
            NumOfNeutrons,SimRuns)

print("\n---------- GRAPHITE - WATER ----------")
AreasG_to_W     = [AbAG,AbAW,ScAG,ScAW]
massesG_to_W    = [MG, MW]
densitiesG_to_W = [DG, DW]
MultiWoodCock(Lengths, AreasG_to_W, densitiesG_to_W,massesG_to_W,
            NumOfNeutrons,SimRuns)

print("\n---------- GRAPHITE - LEAD ----------")
```

```
AreasG_to_L     = [AbAG,AbAL,ScAG,ScAL]
massesG_to_L    = [MG, ML]
densitiesG_to_L = [DG, DL]
MultiWoodCock(Lengths, AreasG_to_L, densitiesG_to_L,massesG_to_L,
            NumOfNeutrons,SimRuns)

print("\n---------- LEAD - GRAPHITE ----------")
AreasL_to_G     = [AbAL,AbAG,ScAL,ScAG]
massesL_to_G    = [ML, MG]
densitiesL_to_G = [DL, DG]
MultiWoodCock(Lengths, AreasL_to_G, densitiesL_to_G,massesL_to_G,
            NumOfNeutrons,SimRuns)

print("\n---------- WOODCOCK RANDOM WALK ----------")


ys = n.linspace(-20, 20, 20) # equal Y and Z co-ordinates
zs = n.linspace(-20, 20, 20)

Y, Z = n.meshgrid(ys, zs) # Forming the Y and Z's into 2D mesh
X  = 10                   # Material boundary at x = 10cm

"""Note that here I have looped 100 times and set it to only plot for
transmissions. This is simply because I wanted a pretty plot of a neutron
going through water and then graphite for my report. Can safely omit the
loop and the if t == True: if statement """

for j in range(0,100): # loop 100 times
   # Call woodcock simulation for a single particle
   a, r, t, Path = WoodCockSim(Lengths, AreasW_to_G, densitiesW_to_G,
                           massesW_to_G ,True)

   if t == True: # If this sim was transmitted, plot
      figW = plt.figure() # 3D plot setup
      wx=Axes3D(figW)
      wx.set_xlabel("X / cm")
      wx.set_ylabel("Y")
      wx.set_zlabel("Z")
      wx.plot(Path[0],Path[1],Path[2], 'r') # Plot neutron history
      wx.plot_surface(X, Y, Z, alpha = 0.2) # plot 10cm boundary

      if a == True:                 # Conditions for different outcomes
         OutcomeColour = 'red'      # Plot them different for
      elif r == True:               # Absorption, Reflection and Trans
         OutcomeColour = 'green'
      else:
         OutcomeColour = 'blue'

      for i in range (0, (len(Path[0]))): # loop over all individual points
         x,y,z = float(Path[0][i]),float(Path[1][i]),float(Path[2][i])
         if i == 0:             # plot entry point with black blobl
            wx.scatter(x,y,z, 'x', c= 'black')
         elif i == (len(Path[0])-1):# plot final point with colour blob
            wx.scatter(x,y,z, 'ro',c = OutcomeColour, s = 50)
         else:
            pass
""" END OF PROGRAM """
```