Studying the penetration of thermal neutrons through different shielding using Monte Carlo techniques

Monty Kirner

10301768

School of Physics and Astronomy

University of Manchester

Project 3 report

May 2020

Abstract

Monte Carlo techniques were used to study the characteristics of thermal neutron penetration through shielding with varying thickness. The motion of these neutrons was simulated for three different materials: water, lead and graphite. The fraction of normally-incident neutrons transmitted, reflected and absorbed in these three materials were determined. The variation in the fraction of transmitted neutrons for different thicknesses was used to calculate the characteristic attenuation length of each material. These were determined as (1.69 $\pm$ 0.06)cm, (9.9 $\pm$ 0.4)cm and (13.6 $\pm$ 0.9)cm for water, lead and graphite respectively.

1 Introduction

Monte Carlo methods are a class of techniques used to randomly sample a probability distribution [1]. These techniques can be applied in algorithms that model physical systems such as fluids or cellular structures which at a fundamental level consist of simple particle interactions. The first use of Monte Carlo methods was in 1733, when Georges-Louis Leclerc, Comte de Buffon proposed his 'Buffon's needle' problem [2]. The experiment involved dropping needles onto a floor made of parallel strips of wood, each the same width. By calculating the probability that each needle would lie across a line between two strips, an approximation for π can be determined. Another application of these techniques can be found in the nuclear industry when investigating neutron transport - the study of motion and interactions of neutrons within materials [3]. This is crucial when designing nuclear reactors where the shielding must be sufficiently thick to prevent harmful levels of radiation leaking from the reactor. It is therefore important to understand what thickness and type of material are required to achieve this [4]. In this project, the penetration of thermal neutrons through different shielding materials was studied using Monte Carlo methods for a range of thicknesses.

2 Theory

Consider a beam of neutrons with intensity, I, fired normally-incident at a slab of material with thickness L. This slab can be divided into thin layers of unit thickness, dL, consisting of randomly distributed target particles, each with microscopic cross-section, σ. The number of neutrons that interact with particles within the layer is therefore

$$N = n\sigma IL = \Sigma_T IL, \tag{1}$$

where n is the number density of target particles per unit volume and Σ_T is the total macroscopic cross-section of all possible events that can occur. This number density can be written as

$$n = \frac{\rho N_A}{M}, \tag{2}$$

where N_A is Avogadro's constant and ρ, M, are the density and molar mass of the material. The intensity of the beam after passing through a layer will therefore decrease by dI as

$$\frac{dI}{dL} = -\Sigma_T I. \tag{3}$$

By separation of variables and integration of Eq. (3), the intensity of the beam at depth, x, is given as

$$I(x) = I_0 e^{-x/\lambda_T}, \tag{4}$$

where I_0 is the initial intensity and λ_T is the total mean free path equal to $1/\Sigma_T$. In this project it is assumed that λ_T is dependant on two processes: absorption and scattering of the neutrons with microscopic cross-sections σ_a and σ_s respectively [3]. By summation of these microscopic cross-sections and multiplying by n, λ_T can be calculated as

$$\lambda_T = \frac{1}{\Sigma_a + \Sigma_s}, \tag{5}$$

where Σ_a and Σ_s are the respective absorption and scattering macroscopic cross-sections. This assumes that events are modelled as a binomial distribution since the neutron can either absorb or

scatter when hitting a target particle. If the neutron is scattered it will then move a distance, x_i, which can be found by using the inverse cumulative function method. This method produces the required probability distribution by generating random, uniformly distributed numbers. By setting the right side of Eq. (4) as the probability distribution, integrating this with limits [0, x] and then inverting, the inverse cumulative function can be found [5]. As a result, the distance moved is written as

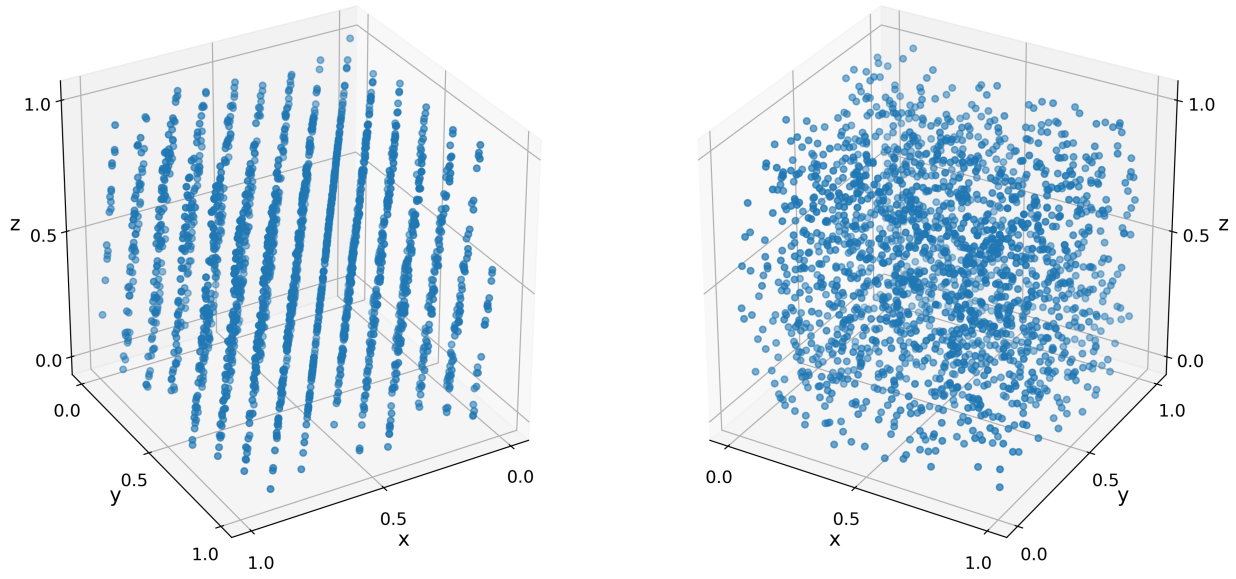
$$x_i = \lambda_T \log(u_i), \tag{6}$$

where u_i are random, uniformly distributed numbers between 0-1. It is also assumed that the neutrons in these simulations are thermalised, in that they have been scattered sufficiently in the material and so reach equilibrium temperature with the medium. This results in isotropic scattering which is independent of the previous motion. This project therefore relies on Monte Carlo methods to simulate the random paths that the neutrons take by repeated random sampling [6].

3 Method

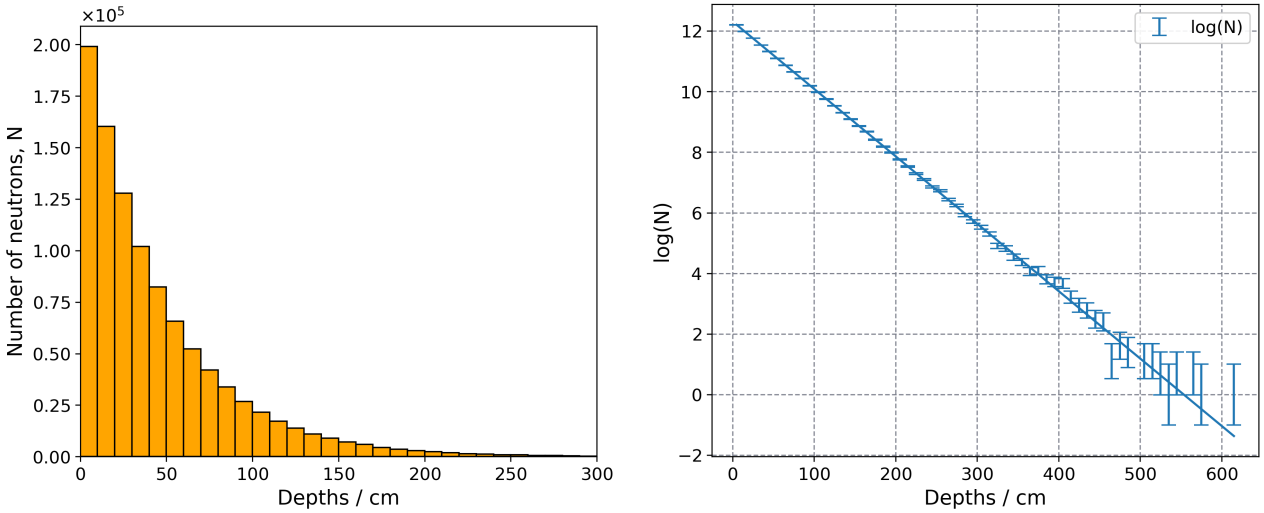
To fully develop a simulation of neutron penetration through various materials, preliminary steps were made to ensure randomness. This physical system is modelled as a collection of random walks that the neutrons take through the material, where at each step random numbers must be drawn to affect the outcome of the following event. The code in Appendix A must therefore be able to successfully generate tables of random numbers. In this case pseudorandom number generators were used, which produce sequences that have no correlation to the process simulated by approximating the properties of random number sequences. These sequences are deterministic, and so can be entirely reproducible, though the outcome is still statistically unbiased. Various pseudorandom number generators were tested to determine which would be most suitable to be used in the simulations. This was achieved by performing spectral tests on the generators. Each generator was tasked with producing a table of points in 3D that could then be plotted. This is shown in Fig. 1 for two different generators. Figure 1(a) used an IBM implementation of a linear congruential generator from the code in Appendix D while Fig. 1(b) used the 'random.uniform' function from the NumPy module. The spectral problem is clearly present in Fig. 1(a) since hyperplanes are formed. The generator in Fig. 1(b) did not display this effect through all rotations of the axes and so was used to produce random numbers for the simulations.

Samples were then distributed according to the exponential function in Eq. (4) for the chosen generator. Using 45cm as the mean free path of neutrons in water the number of neutrons, N, as a function of depth was plotted for $N = 1 \times 10^6$ in the absence of scattering. This is shown in Fig. 2(a). By taking the log of N at bin midpoints the mean free path can be found using Eq. 6 from the line of best fit. Errors in the fitted line were propagated from the error in N for each bin. The gradient of the graph in Fig. 2(b) gave a value for the mean free path as (44.96 ± 0.04)cm and so is consistent to one standard deviation. The reduced χ^2 was given to be 1.03 suggesting a good fit with suitable error approximation. It was found that for repeated runs there was a systematic error in the calculated value that was consistently smaller than the input mean free path. This was most likely due to the fitting procedure, where bins with zero neutrons were excluded to avoid infinities in the logarithmic scale.



(a) (b)

Figure 1: Spectral testing of two distinct pseudorandom number generators. (a) IBM implementation of a linear congruential generator. (b) Random numbers generated using the NumPy module. The spectral problem is clearly present in (a) since hyperplanes are formed, however the generator used for (b) does not display this issue.



(a) (b)

Figure 2: (a) Shows how the number of neutrons vary as a function of depth in water where only absorption is considered. The simulation conditions are $N = 1 \times 10^6$. (b) Data from (a) is plotted on a logarithmic scale with a line of best fit to determine the attenuation length.

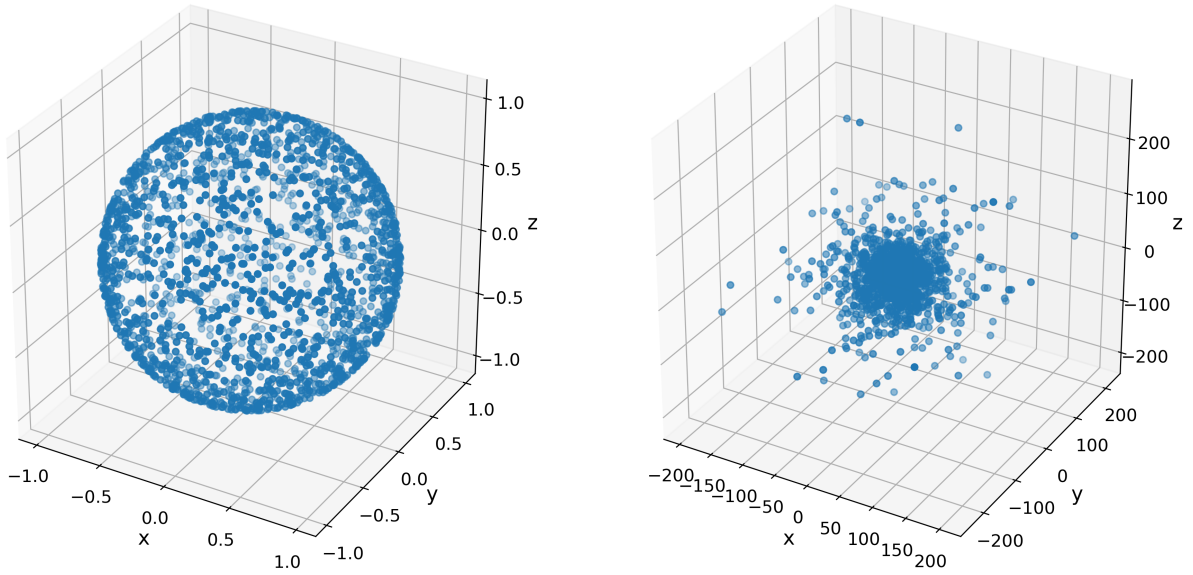
As discussed in Section 2, when a neutron is scattered it will move a random distance in a random direction under the model assumptions made. This random distance can be calculated using Eq. 6. To determine the random direction the neutron takes, a function was made which generates isotropic

points on a sphere. To obtain each point two random numbers, u and v, were generated between 0-1 and then transformed such that

$$\theta = \cos^{-1}(2u - 1) \tag{7}$$

$$\phi = 2\pi v \tag{8}$$

where θ and ϕ correlate to the spherical polar coordinates of a given point. This is shown for a set of uniformly distributed points in Fig. 3(a). By multiplying a random point on the sphere with the random distance previously calculated an isotropic, exponentially distributed random walk can be successfully simulated. A collection of these random steps is shown in Fig. 3(b).



(a) (b)

Figure 3: (a) Isotropic, uniformly distributed points on a sphere. (b) Isotropic, exponentially distributed points.

The random walk of the thermal neutrons is separated into independent steps. When first entering the material the simulated neutrons will all move in the positive x-direction only. This initial step will be exponentially distributed for each neutron. At the end of this step the simulation will then decide whether the neutron is absorbed or scattered. This is based on the σ_a and σ_s values of the material and uses a Monte Carlo technique to choose the process used. The σ values for water, lead and graphite are given in Table 1 along with their respective densities. These values were used to determine Σ_a, Σ_s and the resultant λ_T.

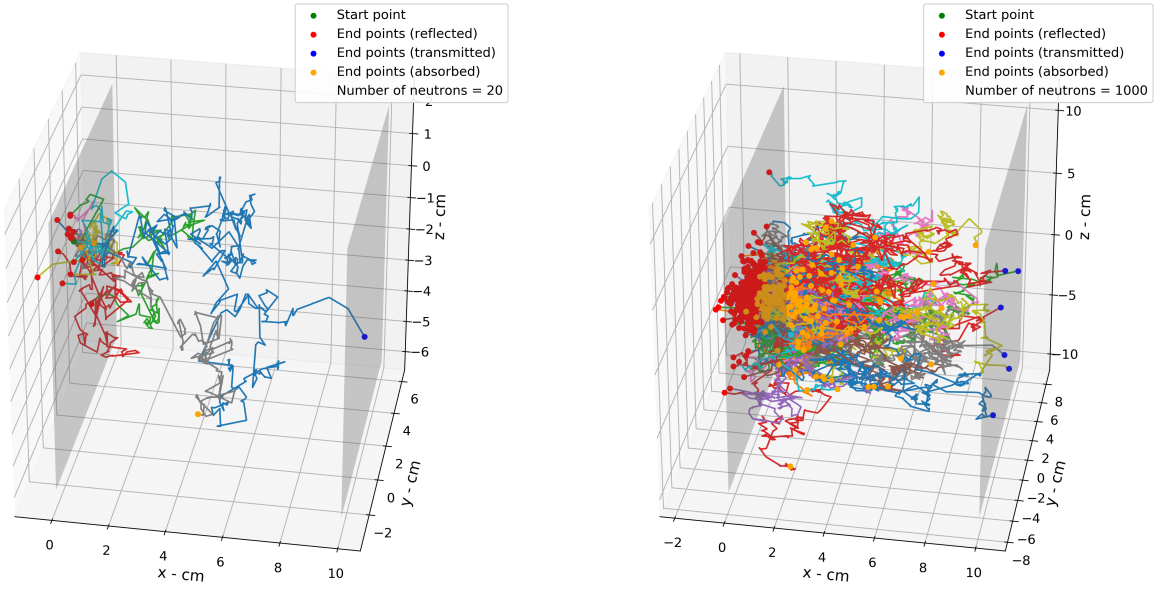
Material type	Absorption σ_a (barn)	Scattering σ_s (barn)	Density ρ (gcm^{-3})
Water	0.6652	103.0	1.0
Lead	0.158	11.221	11.35
Graphite	0.0045	4.74	1.67

Table 1: Required material properties of water, lead and graphite [7].

If scattered, the neutron will then take another random walk step using the techniques discussed. This process continues until the neutron is absorbed, at which point simulation will stop. At each step the simulation will also check if the neutron has left the material via reflection or transmission, and if so it will stop. The path that a neutron takes until it either absorbs or escapes is known as it's particle history.
In this project the fraction of normally-incident neutrons transmitted, reflected, and absorbed in each of the three materials was calculated for different thicknesses. This was achieved by looking at the final positions of the neutron histories to check whether they had reflected, absorbed or transmitted. The fraction of transmitted neutrons was then used to determine the characteristic attenuation lengths for each material. The simulation ran 10 times at each thickness for $N = 1 \times 10^6$. This allowed averaged values to be calculated with errors.

4 Results and discussion

Using the method discussed above, random walks were simulated and plotted to compare the materials. Figures 4 and 5 show examples of random walks for different values of N in water and lead. Similar plots were also made for graphite, and display similar results as the graphs plotted for lead.

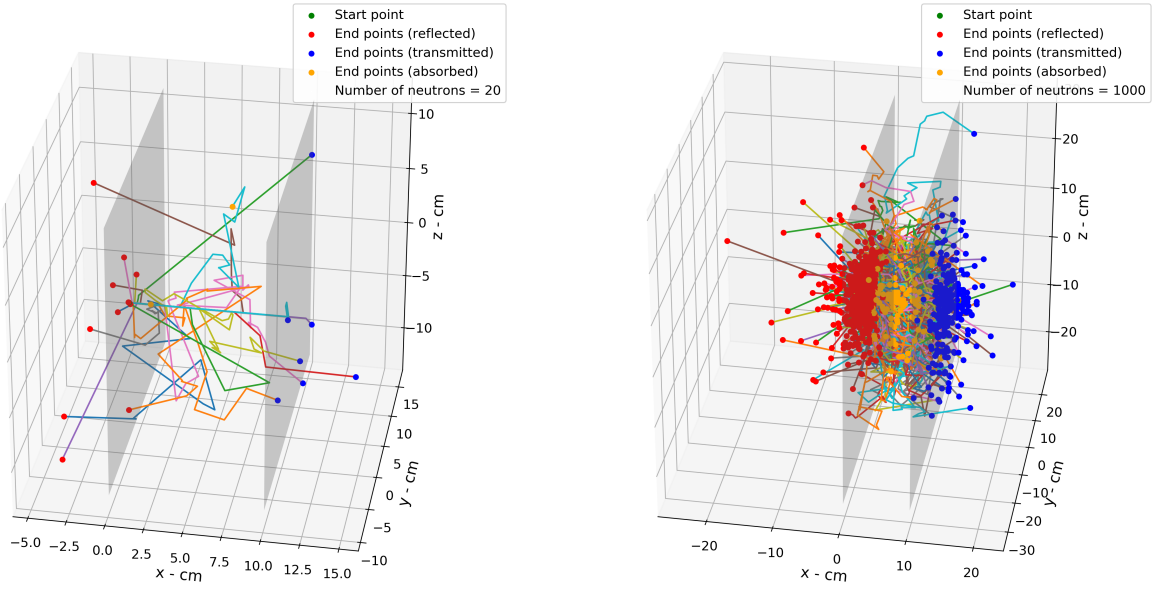


(a) (b)

Figure 4: Random walk simulations in water for (a) $N = 20$ and (b) $N = 1000$. Both plots indicate the start and end points of each walk, as well as the boundaries of the material of thickness 10cm.

These graphs indicate that water has the shortest mean free path when compared to lead and graphite which have similar mean free path values. Comparison of Fig. 4(b) and Fig. 5(b) show that the fraction of transmitted neutrons is much higher in lead than water, where only 6 out of 1000 neutrons are transmitted. From the code in Appendix A, the percentage of normally-incident neutrons transmitted, reflected, and absorbed in each of the three materials was then calculated over a range of 0.5-40cm. The effects of changing N were also studied to determine the best value, where a range of values between 100 and 1×10^7 were tested. This optimum value was determined to be $N = 1 \times 10^6$ since

this balanced the time taken for computation with the accuracy achieved. This gave errors of less than 10%. The results for a thickness of 10cm are shown in Table 2 where values are averaged from the 10 simulations.



(a) (b)

Figure 5: Random walk simulations in lead for (a) $N = 20$ and (b) $N = 1000$. Both plots indicate the start and end points of each walk, as well as the boundaries of the material of thickness 10cm.

Material type	Reflected (%)	Absorbed (%)	Transmitted (%)
Water	79.6 ± 0.2	20.1 ± 0.2	0.32 ± 0.02
Lead	61.8 ± 0.2	10.05 ± 0.07	28.2 ± 0.2
Graphite	68.4 ± 0.2	0.81 ± 0.03	30.8 ± 0.2

Table 2: Percentage of normally-incident neutrons transmitted, reflected and absorbed averaged over 10 simulations where $N = 1 \times 10^6$.

The results in Table 2 show that the majority of neutrons are reflected for all materials simulated. It also confirms that water has the smallest mean free path, which further validates the graphs shown in Fig. 4 and Fig. 5. It can be seen that water is the best absorber of thermal neutrons, followed by lead and finally graphite. This is supported by the fact that the percentage of transmitted neutrons is approximately 100 times larger for lead and graphite. This is due to the water consisting of large amounts of hydrogen, which has mass approximately equal to that of a neutron. This means that collisions will significantly reduce the speed of the incident neutron due to the laws of conservation of energy and momentum. The fraction of transmitted neutrons at the different thicknesses was then used to determine the characteristic attenuation lengths for each material. The graph in Fig. 6 shows how the fraction of transmitted neutrons varies as a function of thickness.

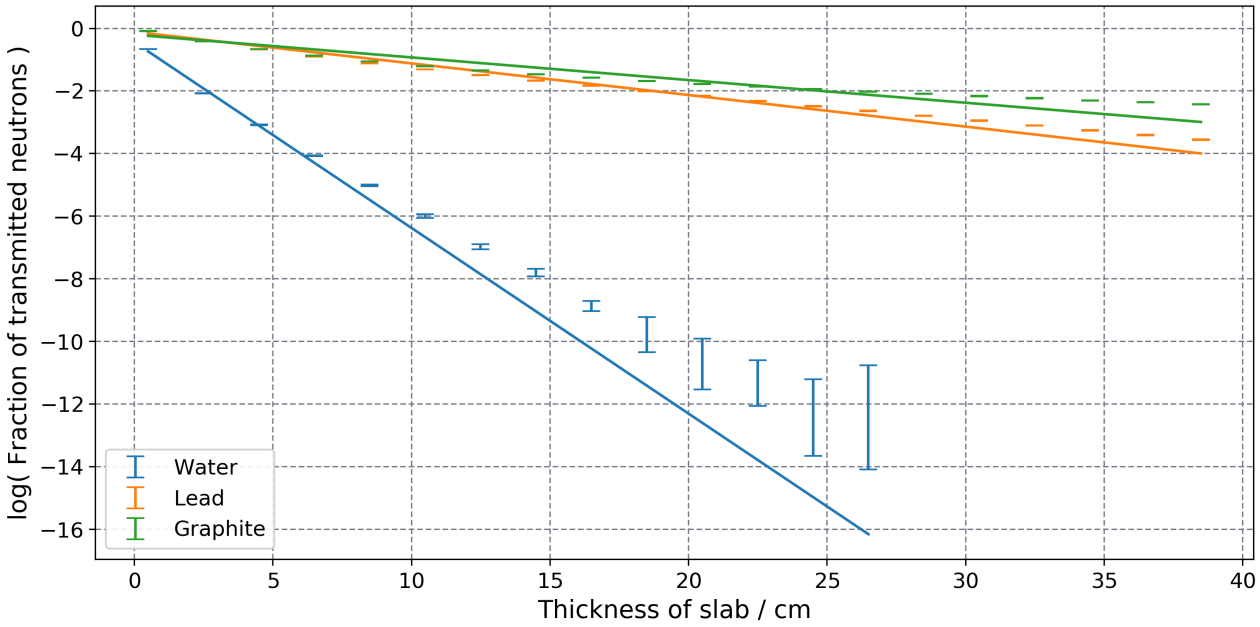


Figure 6: Fraction of transmitted neutrons as a function of thickness plotted on a logarithmic scale with a line of best fit for each material.

From the lines of best fit produced in Fig. 6, the attenuation lengths of water, lead and graphite were determined as (1.69 ± 0.06)cm, (9.9 ± 0.4)cm and (13.6 ± 0.9)cm respectively. These attenuation lengths are significantly larger than the λ_T values which were calculated as 0.29cm, 2.66cm and 2.52cm respectively. However, reduced χ^2 values of 256.7, 1037.8 and 2872.2 indicate that the model is incompatible with the physics of the system. The assumption that the data should follow an exponential decrease is therefore not valid for these data sets. For neutrons that can only be absorbed, the path taken is always in one direction with the probability of absorption being constant for all thin layers, dx. If scattering is included in the model then the neutrons can now travel in any direction and so the distribution is no longer exponential and as such a different model is needed. Furthermore, this model assumes that the neutrons are thermalised, however as the neutrons move their energy changes which results in fluctuation of the σ_a value. Another issue with the model is that it only considers neutrons. Successful particle transport codes are also able to model other particles all interacting with each other, where cross-sections for each type of particle will be different. Incident particles may also cause target particles to scatter sufficiently so that they must now be considered in the simulation. To improve this project these effects could be taken into account to change the model used to further advance simulation.

However, true Monte Carlo simulations do not produce exact solutions since they rely on the code terminating after a finite number of particle histories have been processed. As such these simulations will contain intrinsic statistical errors. One advantage of using Monte Carlo techniques for neutron transport is that they are simple to model since they directly apply the physics to the simulation of individual particle histories. If few assumptions are made each neutron can be simulated such that they accurately follow the underlying physics. As a result, the simulations are capable of producing systems that are almost void of any type of truncation error. Despite this, making an advanced simulation such as this is expensive to run at an acceptable efficiency. Therefore, although the simulation might be free of truncation errors, it is difficult to reduce statistical error [8]. In the case of this project statistical error can also be reduced by increasing N or the number of simulations.

Another type of error in this project is due to discretisation errors when approximating the random walks as a series of steps when it is in fact a continuous process. Furthermore, computing errors such as rounding error caused the accuracy of the simulations to decrease. This is due to decimal numbers being stored as binary representations and so when a sequence of calculations with inputs involving rounding errors are made, errors will accumulate.

5 Conclusion

This project was carried out to investigate how the penetration of thermal neutrons through different shielding materials varies using Monte Carlo methods. By simulating the random walks of neutrons within water, lead and graphite, the fraction of normally-incident neutrons transmitted, reflected and absorbed in each of these materials were determined. From this the characteristic attenuation lengths of water, lead and graphite were determined as (1.69 ± 0.06)cm, (9.9 ± 0.4)cm and (13.6 ± 0.9)cm respectively. To improve the project, the model used to simulate neutron transport could be revised to include other physical properties of the system such as the inclusion of different particle types within the materials used. To reduce statistical error more simulations could be run at larger N values.

References

[1] J. Spanier and M. Gelbard, *Monte carlo principles and neutron transport problems*. Addison-wesley, 1969.

[2] G. Buffon, *Histoire de l'Acad. Roy. des Sci.*, pp. 43–45. Royal Academy of Sciences Paris, 1733.

[3] V. F. Sears, "Neutron scattering lengths and cross sections," *Neutron News*, vol. 3, no. 3, pp. 26–37, 1992.

[4] J. Shultis and R. Faw, "Radiation shielding technology," *Health Physics*, vol. 88, no. 6, pp. 587–612, 2005.

[5] E. D. Cashwell, *A practical manual on the Monte Carlo method for random walk problems*. Los Alamos Scientific Laboratory of the University of California,, 1957.

[6] Y. A. Shreider, *The Monte Carlo method: The Method of Statistical Trials*, ch. 3, pp. 137–183. Pergamon Press, 1966.

[7] G. W. C. Kaye and T. H. Laby, *Tables of physical and chemical constants and some mathematical functions*. Longman, 15 ed., 1986.

[8] E. W. Larsen, "An overview of neutron transport problems and simulation techniques," in *Computational Methods in Transport* (F. Graziani, ed.), pp. 513–534, Springer Berlin Heidelberg, 2006.

A Appendix: Project-3.py file:

```
'''
------------------------------------------------------------------------------

PHYS20762 Project 3 - Monte Carlo Techniques

Monty Kirner - 10301768 - 24/04/20 - University of Manchester

------------------------------------------------------------------------------

This code simulates the penetration of thermal neutrons through different
shielding using Monte Carlo techniques.

------------------------------------------------------------------------------
'''
#------------------------------------------------------------------------------
# Initialisation
#------------------------------------------------------------------------------

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import constants as pc
from numba import jit

# Imported files (must be in same directory as this .py file):
import Plot_Data as plot
import Stats as stats
from randssp import randssp

#------------------------------------------------------------------------------
# Constants and input data
#------------------------------------------------------------------------------

BARN = 10**-24

# Data formatted as absorption (σa/barn), scattering (σs/barn),
# density (gcm-3) and atomic mass (u):
WATER_DATA = np.array([0.6652, 103.0, 1.0, 18.015])
LEAD_DATA = np.array([0.158, 11.221, 11.35, 207.2])
GRAPHITE_DATA = np.array([0.0045, 4.74, 1.67, 12.0107])

#------------------------------------------------------------------------------
# Functions
#------------------------------------------------------------------------------

@jit(nopython = True)
def calculate_mean_free_paths(material_data):
    ''' Calculates the three mean free paths for a given data set and returns
    these values in a numpy array.
    '''
    # Gets number of moles where M = material_data[3]:
    n_moles = ( pc.Avogadro * material_data[2] ) / material_data[3]
```

```
    # Finds absorption and scattering cross section:
    sigma_a = material_data[0] * BARN
    sigma_s = material_data[1] * BARN

    # Calculates macroscopic cross sections:
    macro_sigma_a = n_moles * sigma_a
    macro_sigma_s = n_moles * sigma_s

    # Finds the mean free paths:
    lambda_a = 1 / macro_sigma_a
    lambda_s = 1 / macro_sigma_s
    lambda_t = 1 / ( macro_sigma_a +  macro_sigma_s )

    return np.array([lambda_a, lambda_s, lambda_t,
                     macro_sigma_a, macro_sigma_s])

#-------------------------------------------------------------------------

def get_mean_free_paths_df():
    ''' Calculates the three mean free paths for the three different materials
    and returns these values in a df.
    '''
    water_lambda_array = calculate_mean_free_paths(WATER_DATA)
    lead_lambda_array = calculate_mean_free_paths(LEAD_DATA)
    graphite_lambda_array = calculate_mean_free_paths(GRAPHITE_DATA)

    # Finds the mean free paths and converts to df:
    all_mean_free_paths_df = pd.DataFrame({
        'Water' : water_lambda_array,
        'Lead' : lead_lambda_array,
        'Graphite' : graphite_lambda_array },
        index = ['lambda_a / cm', 'lambda_s / cm', 'lambda_t / cm',
                 'macro_sigma_a / cm', 'macro_sigma_s / cm'])

    return all_mean_free_paths_df

#-------------------------------------------------------------------------

@jit(nopython = True)
def random_uniform(array_length):
    ''' Required to use np.random.uniform function with numba, use this to call
    the function.
    '''
    random_array = np.empty(array_length, dtype = np.float64)
    for i in range(array_length):
        random_array[i] = np.random.uniform(0, 1)

    return random_array

#-------------------------------------------------------------------------

@jit(nopython = True)
def get_rand_exp(mean_free_path, array_length):
    ''' Uses a random number generator to generate samples distributed
    according to an exponential function exp(- u_data / path_length).
```

```
    '''
    u_data = random_uniform(array_length)
    s_data = - mean_free_path * np.log(u_data)

    return s_data

#-------------------------------------------------------------------------

@jit(nopython = True)
def get_rand_vector(radius, array_length):
    ''' Generates isotropic vectors, r = (x, y, z), of specified radius using
    spherical polar co-ordinates and returns these values.
    '''
    # Uses random_uniform since numba can't compile np.random.uniform:
    random_theta_array = random_uniform(array_length)
    random_phi_array = random_uniform(array_length)

    theta = np.arccos( 2 * random_theta_array - 1 )
    phi = 2 * np.pi * random_phi_array

    x = radius * np.sin(theta) * np.cos(phi)
    y = radius * np.sin(theta) * np.sin(phi)
    z = radius * np.cos(theta)

    return x, y, z

#-------------------------------------------------------------------------

@jit(nopython = True)
def get_rand_exp_vector(mean_free_path, array_length):
    ''' Generates uniform, isotropic, randomly distributed vectors with lengths
    distributed as exp(-x/l).
    '''
    # Sets up empty array:
    rand_exp_vectors = np.zeros( (array_length, 3) )

    # Creates set of s_data (lengths):
    s_data = get_rand_exp(mean_free_path, array_length)

    # Creates set of random unit vectors:
    x, y, z = get_rand_vector(1, array_length)

    for i in range(len(s_data)):

        # Multiplies each unit vector x, y, z by the random length:
        rand_exp_vectors[i][0] = x[i] * s_data[i]
        rand_exp_vectors[i][1] = y[i] * s_data[i]
        rand_exp_vectors[i][2] = z[i] * s_data[i]

    return rand_exp_vectors

#-------------------------------------------------------------------------

@jit(nopython = True)
def particle_walk(total_mean_free_path, macro_sigma_a, macro_sigma_s,
```

```
                thickness, track_positions):
    ''' Simulates a particle walk for one neutron through a slab of certain
    thickness. If track_positions = True, the output will consist of full array
    of the particles position history, else it will return the final position.
    '''
    particle_absorbed = False
    particle_killed = False

    # Sets up initial starting point:
    position = np.array([0., 0., 0.])

    step = 0

    # Adds first x-step to array and keeps track:
    if track_positions == True:

        # Use np.absolute to only take +ve direction:
        new_position = np.absolute(get_rand_exp(total_mean_free_path, 1))
        position[0] += new_position[0]

        tracked_positions = np.vstack((np.array([0., 0., 0.]), position))
        step = 1

    while not particle_absorbed and not particle_killed:

        # First step without particle tracking:
        if step == 0:

            new_position = np.absolute(
                get_rand_exp(total_mean_free_path, 1))
            position[0] += new_position[0]
            step = 1

        # Following steps:
        else:
            # next_step has shape (1, 3):
            next_step = get_rand_exp_vector(total_mean_free_path, 1)

            # position has shape (3,):
            position[0] += next_step[0][0]
            position[1] += next_step[0][1]
            position[2] += next_step[0][2]

        reshaped_position = position.reshape((1, 3))

        # So that all positions are added to array (slower):
        if track_positions == True:

            tracked_positions = np.vstack((tracked_positions,
                                           reshaped_position))
        # Used for absorbed elif statement:
        random_number = random_uniform(1)[0]

        if position[0] < 0:
```

```
            kill_type = 'Reflected'
            particle_killed = True

        elif position[0] > thickness:

            kill_type = 'Transmitted'
            particle_killed = True

        elif random_number < ( macro_sigma_a / (macro_sigma_a +
                                                macro_sigma_s) ):
            kill_type = 'Absorbed'
            particle_absorbed = True

        step += 1

    if track_positions == True:

        return tracked_positions, kill_type

    else:
        return reshaped_position, kill_type

#-------------------------------------------------------------------------------

@jit(nopython = True)
def get_particle_walk_arrays(number_of_neutrons, thickness,
                             track_positions, total_mean_free_path,
                             macro_sigma_a, macro_sigma_s):
    ''' Runs the particle_walk function for the number of neutrons chosen, and
    also counts the number of reflected / absorbed / transmitted neutrons to
    then return in console.
    '''
    reflected = 0
    absorbed = 0
    transmitted = 0

    # Numba cannot use np.array([]) so use lists:
    random_walk_x = []
    random_walk_y = []
    random_walk_z = []

    for neutron in list(range(0, number_of_neutrons)):

        positions, kill_type = particle_walk(
            total_mean_free_path, macro_sigma_a, macro_sigma_s,
            thickness, track_positions)

        if kill_type == 'Reflected':
            reflected += 1

        elif kill_type == 'Absorbed':
            absorbed += 1

        elif kill_type == 'Transmitted':
            transmitted += 1
```

```
        # Appends positions array, so that random_walk_x[0] is first neutron:
        random_walk_x.append(positions[:,0])
        random_walk_y.append(positions[:,1])
        random_walk_z.append(positions[:,2])

    return (random_walk_x, random_walk_y, random_walk_z,
            reflected, absorbed, transmitted)

#-------------------------------------------------------------------------

def roundup_max_to_10(data):
    ''' Finds the maximum value of an array and then rounds this up to the
    next multiple of 10 (ie. 342 -> 350). Use this to choose even bin size.
    '''
    max_point = np.amax(data)
    rounded_max = ( np.ceil( max_point / 10.0) * 10 )

    return rounded_max

#-------------------------------------------------------------------------
#-------------------------------------------------------------------------
# Main code
#-------------------------------------------------------------------------
#-------------------------------------------------------------------------

# For first few example plots:
small_samples = 2000
large_samples = 1000000

# Set to True to display initial setup plots / histogram & log plots:
plot_initial_graphs = True

plot_exp_hist_graph = True

# Finds the mean free paths for all materials:
all_mean_free_paths_df = get_mean_free_paths_df()

#-------------------------------------------------------------------------

# Change these values depending on what you want to test for random walks:

# How many neutrons simulated for each run:
number_of_neutrons = 1000000

# How many times you simulate the randoms walks:
number_of_simulations = 10

# Either 'Water' / 'Lead' / 'Graphite' for which material tested:
material_name = 'Water'

# If True, the simulation will keep a full array of the particles position
# history, if False it will only return the final position:
track_positions = True

# Returned values:
```

```
output = True

# To plot final simulations graph (also set thickness_range = False):
plot_random_walks = False

# Use this when only looking at 10cm:
thickness_values = np.array([10.0])

# If you want to choose a range, set to True:
thickness_range = False

if thickness_range == True:

    # In cm:
    thickness_values = np.arange(0.5, 40.5, 2)

    # Clutters console otherwise:
    output = False

#------------------------------------------------------------------------------

# Random generator that produces tables of points in 3 dimensions (x,y,z):
x_np = np.random.uniform(size = small_samples)
y_np = np.random.uniform(size = small_samples)
z_np = np.random.uniform(size = small_samples)

if plot_initial_graphs == True:

    # Diplays plot in 3D using Plot_Data.py:
    plot.scatter_3d(x_np, y_np, z_np, -60, 'numpy-uniform-3D-plot',
                    axes = ['x', 'y', 'z'], fig_num = 1)

#------------------------------------------------------------------------------

# Confirming the spectral problem present using randssp:
# Creates 3D array of shape (3, samples):
randssp = randssp(3, small_samples)

x_randssp = randssp[0]
y_randssp = randssp[1]
z_randssp = randssp[2]

if plot_initial_graphs == True:

    plot.scatter_3d(x_randssp, y_randssp, z_randssp, 60, 'randssp-3D-plot',
                    axes = ['x', 'y', 'z'], fig_num = 2)

#------------------------------------------------------------------------------

# Generates isotropic unit vectors:
x_unit, y_unit, z_unit = get_rand_vector(1, small_samples)

if plot_initial_graphs == True:

    plot.scatter_3d(x_unit, y_unit, z_unit, -60, 'rand-unit-vectors-3D-plot',
```

```
                    axes = ['x', 'y', 'z'], fig_num = 3)

#-------------------------------------------------------------------------------

# Generates randomly distributed isotropic exponential vectors:
rand_exp_vectors = get_rand_exp_vector(
    all_mean_free_paths_df.loc['lambda_a / cm', 'Water'], small_samples)

if plot_initial_graphs == True:

    plot.scatter_3d(rand_exp_vectors[:,0], rand_exp_vectors[:,1],
                    rand_exp_vectors[:,2], -60,
                    'rand-exp-vectors-3D-plot',
                    axes = ['x', 'y', 'z'], fig_num = 4)

#-------------------------------------------------------------------------------

# Exponential function random number generator:

# Get exponential function for absorption:
water_absorp_s_data = get_rand_exp(all_mean_free_paths_df.loc[
    'lambda_a / cm', 'Water'], large_samples)

# Finds largest data point and round this up to next 10:
water_absorp_s_bin_max = roundup_max_to_10(water_absorp_s_data)

water_absorp_bins = np.arange(0, water_absorp_s_bin_max, 10)

if plot_exp_hist_graph == True:

    plot.histogram(water_absorp_s_data, water_absorp_bins,
                   'water-absorption-exponential-hist',
                   axes = ['Depths / cm', 'Number of neutrons, N'], fig_num =5)

#-------------------------------------------------------------------------------

# Finds the number of neutrons per bin, N:
water_absorp_hist, water_absorp_bins = np.histogram(water_absorp_s_data,
                                                    water_absorp_bins)

# Finds the error for each bin, N_error:
water_absorp_errors = np.sqrt( water_absorp_hist *
                               ( 1 - (water_absorp_hist / large_samples)) )

# Finds the bin midpoints (removes initial 0 since shifting by -5):
water_absorp_bin_midpoints = water_absorp_bins[1::] - 5

water_absorp_neutrons_df = pd.DataFrame({
    'Depths / cm' : water_absorp_bin_midpoints,
    'N' : water_absorp_hist,
    'N_error' : water_absorp_errors })

# Removes rows where N <= 0 (ie. no neutrons in bin):
water_absorp_remove_zeroes_df = water_absorp_neutrons_df[
    water_absorp_neutrons_df['N'] > 0]
```

```
# Renames for plotting:
water_absorp_log_neutrons_df = pd.DataFrame({
    'log(N)' : np.log(
        water_absorp_remove_zeroes_df['N']) })

water_absorp_log_errors = (water_absorp_remove_zeroes_df['N_error']
                           / water_absorp_remove_zeroes_df['N'])

water_absorp_log_errors_df = pd.DataFrame({
    'log(N)' : water_absorp_log_errors })

#-------------------------------------------------------------------------------

# Finds the fitted line and results:
water_absorp_fit, water_absorp_fit_results_df = stats.linear_fit(
    water_absorp_remove_zeroes_df['Depths / cm'],
    water_absorp_log_neutrons_df['log(N)'],
    water_absorp_log_errors_df['log(N)'],
    'water absorption')

water_absorp_fit_df = pd.DataFrame({
    'log(N)' : water_absorp_fit })

water_atten_length = -1 / water_absorp_fit_results_df.loc[
    'Value', 'Gradient']

water_atten_length_error = (
    water_absorp_fit_results_df.loc['Error', 'Gradient'] /
    (water_absorp_fit_results_df.loc['Value', 'Gradient'])**2 )

# Use Stats.py's value rounder:
(water_atten_length_rounded,
 water_atten_length_error_rounded) = stats.value_rounder(
     water_atten_length, water_atten_length_error)

print('''\nAttenuation length of water without \
scattering: ({:g} ± {:g})cm'''.format(
water_atten_length_rounded, water_atten_length_error_rounded))
print('\n' + '-' * 70)

if plot_exp_hist_graph == True:

    plot.scatter(water_absorp_remove_zeroes_df['Depths / cm'],
                 water_absorp_log_neutrons_df,
                 water_absorp_log_errors_df,
                 water_absorp_fit_df,
                 'water-absorption-log',
                 axes = ['Depths / cm', 'log(N)'], fig_num =6)

#-------------------------------------------------------------------------------

# Calculating random walks and plotting:

# To append mean value of fraction transmitted for each thickness:
fraction_trans_array = np.zeros((1, len(thickness_values)) )
```

```
fraction_trans_errors = np.zeros((1, len(thickness_values)) )

thickness_index = 0

for thickness in thickness_values:

    percent_reflected_array = np.zeros((1, number_of_simulations))
    percent_absorbed_array = np.zeros((1, number_of_simulations))
    percent_transmitted_array = np.zeros((1, number_of_simulations))

    # Runs required functions for each simulation:
    for simulation in range(0,number_of_simulations):

        (random_walk_x, random_walk_y, random_walk_z,
         reflected, absorbed, transmitted) = get_particle_walk_arrays(
             number_of_neutrons, thickness, track_positions,
             all_mean_free_paths_df.loc['lambda_t / cm', material_name],
             all_mean_free_paths_df.loc['macro_sigma_a / cm', material_name],
             all_mean_free_paths_df.loc['macro_sigma_s / cm', material_name] )

        random_walk_x = np.asarray(random_walk_x)
        random_walk_y = np.asarray(random_walk_y)
        random_walk_z = np.asarray(random_walk_z)

        percent_reflected = (reflected / number_of_neutrons) * 100
        percent_absorbed = (absorbed / number_of_neutrons) * 100
        percent_transmitted = (transmitted / number_of_neutrons) * 100

        # Adds above values to array:
        percent_reflected_array[0][simulation] = percent_reflected
        percent_absorbed_array[0][simulation] = percent_absorbed
        percent_transmitted_array[0][simulation] = percent_transmitted

    #-------------------------------------------------------------------------

    # Final results and plotting:

    # Calculates the mean and std for each type of process:
    percent_reflected_mean = np.average(percent_reflected_array)
    percent_absorbed_mean = np.average(percent_absorbed_array)
    percent_transmitted_mean = np.average(percent_transmitted_array)

    percent_reflected_std = np.std(percent_reflected_array)
    percent_absorbed_std = np.std(percent_absorbed_array)
    percent_transmitted_std = np.std(percent_transmitted_array)

    # Convert to a fraction:
    fraction_trans_array[0][thickness_index] = percent_transmitted_mean/100
    fraction_trans_errors[0][thickness_index] = percent_transmitted_std/100

    thickness_index += 1

    # Rounds all values to 1 sig fig of error:
    reflected_mean_rounded, reflected_std_rounded = stats.value_rounder(
        percent_reflected_mean, percent_reflected_std)
```

```
    absorbed_mean_rounded, absorbed_std_rounded = stats.value_rounder(
        percent_absorbed_mean, percent_absorbed_std)
    transmitted_mean_rounded, transmitted_std_rounded = stats.value_rounder(
        percent_transmitted_mean, percent_transmitted_std)

    if output == True:

        print('\n' + '-' * 70)
        print(('\nTransmission of neutrons through thickness = ' +
               str(thickness) + 'cm for ' + str(number_of_simulations)
               + ' simulations'))
        print('\n' + '-' * 70)
        print('\nType of material = ' + str(material_name) )
        print('\nTotal neutrons for each simulation = '
              + str(number_of_neutrons) )
        print('\nPercentage reflected = ({:g} ± {:g})%'.format(
            reflected_mean_rounded, reflected_std_rounded) )
        print('\nPercentage absorbed = ({:g} ± {:g})%'.format(
            absorbed_mean_rounded, absorbed_std_rounded) )
        print('\nPercentage transmitted = ({:g} ± {:g})%'.format(
            transmitted_mean_rounded, transmitted_std_rounded) )
        print('\n' + '-' * 70)

#------------------------------------------------------------------------------

if thickness_range == True:

    fraction_trans_df = pd.DataFrame({
        'Thickness of slab / cm' : thickness_values,
        'Fraction transmitted' : fraction_trans_array[0],
        'Fraction transmitted error' : fraction_trans_errors[0] })

    # Removes rows where N <= 0 (ie. no neutrons in bin):
    fraction_trans_remove_zeroes_df = fraction_trans_df[
        fraction_trans_df['Fraction transmitted'] > 0]

    # Rename for plotting:
    log_fraction_trans_df = pd.DataFrame({
        material_name : np.log(
            fraction_trans_remove_zeroes_df['Fraction transmitted']) })

    log_fraction_trans_errors = fraction_trans_remove_zeroes_df[
        'Fraction transmitted error'] / fraction_trans_remove_zeroes_df[
            'Fraction transmitted']

    log_fraction_trans_errors_df = pd.DataFrame({
        material_name : log_fraction_trans_errors})

    #--------------------------------------------------------------------------

    # Finds the fitted line and results:

    fraction_trans_fit, fraction_trans_fit_results_df = stats.linear_fit(
    fraction_trans_remove_zeroes_df['Thickness of slab / cm'],
    log_fraction_trans_df[material_name],
```

```
    log_fraction_trans_errors_df[material_name],
    'range of thicknesses')

    fraction_trans_fit_df = pd.DataFrame({
        material_name : fraction_trans_fit })

    material_atten_length = -1 / fraction_trans_fit_results_df.loc[
    'Value', 'Gradient']

    material_atten_length_error = (
        fraction_trans_fit_results_df.loc['Error', 'Gradient'] /
        (fraction_trans_fit_results_df.loc['Value', 'Gradient'])**2 )

    # Use Stats.py's value rounder:
    (material_atten_length_rounded,
     material_atten_length_error_rounded) = stats.value_rounder(
         material_atten_length, material_atten_length_error)

    print('\nAttenuation length of '+ str(material_name) +
          ': ({:g} ± {:g})cm'.format(material_atten_length_rounded,
                                     material_atten_length_error_rounded))
    print('\n' + '-' * 70)

#------------------------------------------------------------------------------

if plot_random_walks == True and not thickness_range:

    if track_positions == True:
        save_name = (material_name + '-' + str(number_of_neutrons) +
                     '-neutrons-tracking-random-walk-3D-plot')
    else:
        save_name = (material_name + '-' + str(number_of_neutrons) +
                     '-neutrons-no-tracking-random-walk-3D-plot')

    plot.particle_walk_3d(random_walk_x, random_walk_y, random_walk_z,
                          number_of_neutrons, thickness, -80, save_name,
                          axes = ['x - cm', 'y - cm', 'z - cm'], fig_num = 8)

if thickness_range == True:

    plot.scatter(fraction_trans_remove_zeroes_df['Thickness of slab / cm'],
                 log_fraction_trans_df,
                 log_fraction_trans_errors_df,
                 fraction_trans_fit_df,
                 'thickness-range-log',
                 axes = ['Thickness of slab / cm',
                         'log( Fraction of transmitted neutrons )'],
                 fig_num = 7)

#------------------------------------------------------------------------------
```

B Appendix: Plot_Data.py file:

```
"""
-------------------------------------------------------------------------------

Plotting Data Functions

Monty Kirner - 10301768 - 01/05/20 - University of Manchester

-------------------------------------------------------------------------------

This code contains various functions required to plot differnt types of graphs.

-------------------------------------------------------------------------------
"""
#-------------------------------------------------------------------------------
# Initialisation
#-------------------------------------------------------------------------------

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import ticker

# Sets up scientific notation for 3D plots:
formatter = ticker.ScalarFormatter(useMathText = True)
formatter.set_scientific(True)
formatter.set_powerlimits((-4,4))

params = {'legend.fontsize': 12,
          'axes.labelsize': 14,
          'xtick.labelsize':12,
          'ytick.labelsize':12 }
plt.rcParams.update(params)

#-------------------------------------------------------------------------------
# Functions
#-------------------------------------------------------------------------------

def histogram(x_data, bins, save_name, axes, fig_num):
    ''' Creates a histogram for x_data with desired bin size.
    '''
    x_label = axes[0]
    y_label = axes[1]

    fig = plt.figure(num = fig_num, figsize = (6, 5) )

    # Makes 1 subplot of figure:
    ax = fig.add_subplot()

    #ax.grid(True, c = '#777B88', dashes=[4,2])

    ax.hist(x_data, bins, color = 'orange', edgecolor='black')
```

```
    #-------------------------------------------------------------------------

    # Formatting, saving and displaying plots:
    ax.set_xlabel(r'{}'.format(x_label) )
    ax.set_ylabel(r'{}'.format(y_label) )

    ax.set_xlim(0, None)

    # If ticks sizes are >= 1e4, ticks are changed to scientific:
    ax.ticklabel_format(style = "sci", scilimits = (-4,4), useMathText = True)

    plt.tight_layout(pad = 0.5)
    plt.savefig('{}-graph.png'.format(save_name), dpi = 300 )

    return plt.show()

#-----------------------------------------------------------------------------

def scatter(x_data, y_data, y_error, y_fit, save_name, axes, fig_num):
    ''' Creates a scatter graph for multiple datasets with errorbars and lines
    of best fit. y_data_df and y_fit_df must be in pandas dataframe format with
    the same column names, where the column names are also the labels used in
    the legend.
    '''
    x_label = axes[0]
    y_label = axes[1]

    fig = plt.figure(num = fig_num, figsize = (6, 5) )

    # Makes 1 subplot of figure:
    ax = fig.add_subplot()

    ax.grid(True, c = '#777B88', dashes=[4,2])

    # Assigns colours to each set of data:
    plot_colours = ['#1F77B4', '#FF7F0E', '#2CA02C', '#D62728', '#9467BD']

    # Iterates through y_data and y_fit to plot each method:
    for column in y_data:

        # Gets the data of one column in y_data:
        column_y_values = y_data.loc[ : , column ]

        # Gets the errorbars of one column in y_error:
        column_y_error_values = y_error.loc[ : , column ]

        # Gets the fitted data of one column in y_fit:
        column_y_fit_values = y_fit.loc[ : , column ]

        # Finds the index of the column:
        column_index = y_data.columns.get_loc(column)

        # Gets correct label to use in legend:
        labels = r'{:}'.format(column)
```

```
        # Plots data with errorbars:
        ax.errorbar(x_data, column_y_values, yerr = column_y_error_values,
                    ecolor = plot_colours[column_index], fmt = 'none',
                    capsize = 5, label = labels)

        # Add the linear fit line:
        ax.plot(x_data, column_y_fit_values, '-',
                c = plot_colours[column_index])

    #-------------------------------------------------------------------------

    # Formatting, saving and displaying plots:
    ax.set_xlabel(r'{}'.format(x_label) )
    ax.set_ylabel(r'{}'.format(y_label) )

    # If ticks sizes are >= 1e4, ticks are changed to scientific:
    ax.ticklabel_format(style = "sci", scilimits = (-4,4), useMathText = True)

    ax.legend(loc = 'best', framealpha = 0.9)

    plt.tight_layout(pad = 0.5)
    plt.savefig('{}-graph.png'.format(save_name), dpi = 300 )

    return plt.show()

#-----------------------------------------------------------------------------

def scatter_3d(x_data, y_data, z_data, view_angle, save_name, axes, fig_num):
    ''' Creates a 3d scatter plot of given data.
    '''
    x_label = axes[0]
    y_label = axes[1]
    z_label = axes[2]

    fig = plt.figure(num = fig_num, figsize = (6, 6) )

    # Makes 1 3D subplot of figure:
    ax = fig.add_subplot(projection = '3d')

    ax.scatter(x_data, y_data, z_data, c = '#1F77B4')

    #-------------------------------------------------------------------------

    # Formatting, saving and displaying plots:
    ax.set_xlabel(r'{}'.format(x_label) )
    ax.set_ylabel(r'{}'.format(y_label) )
    ax.set_zlabel(r'{}'.format(z_label) )

    # If ticks sizes are >= 1e4, ticks are changed to scientific:
    ax.xaxis.set_major_formatter(formatter)
    ax.yaxis.set_major_formatter(formatter)
    ax.zaxis.set_major_formatter(formatter)

    # For unit vector plots:
    if np.amax(x_data) <= 1 or np.amax(y_data) <= 1 or np.amax(z_data) <= 1:
```

```
        ax.xaxis.set_major_locator(ticker.MultipleLocator(0.5))
        ax.yaxis.set_major_locator(ticker.MultipleLocator(0.5))
        ax.zaxis.set_major_locator(ticker.MultipleLocator(0.5))

    ax.view_init(azim = view_angle)

    plt.tight_layout(pad = 0.5)
    plt.savefig('{}-graph.png'.format(save_name), dpi = 300 )

    return plt.show()

#-------------------------------------------------------------------------

def particle_walk_3d(x_data, y_data, z_data, number_of_neutrons, thickness,
                     view_angle, save_name, axes, fig_num):
    ''' Creates a 3d plot of particle walks where x/y/z_data is in df format.
    '''
    x_label = axes[0]
    y_label = axes[1]
    z_label = axes[2]

    fig = plt.figure(num = fig_num, figsize = (8, 8) )

    # Makes 1 3D subplot of figure:
    ax = fig.add_subplot(projection = '3d')

    #-------------------------------------------------------------------------

    # Plots starting point to use in legend:
    ax.scatter(0, 0, 0, c = 'green', label = 'Start point')

    # Adds end points for legend:
    ax.scatter([], [], [], c = 'red', label = 'End points (reflected)')
    ax.scatter([], [], [], c = 'blue', label = 'End points (transmitted)')
    ax.scatter([], [], [], c = 'orange', label = 'End points (absorbed)')

    # Add number of neutrons to legend:
    plt.scatter([], [], [], c = 'white', label = ('''Number of neutrons = '''
                                        + str(number_of_neutrons)) )

    min_y_array = []
    max_y_array = []
    min_z_array = []
    max_z_array = []

    # Iterates through x_data to plot each neutron:
    for neutron in list(range(0, number_of_neutrons)):

        # Gets the data of one column in each df:
        x_values = x_data[neutron]
        y_values = y_data[neutron]
        z_values = z_data[neutron]

        # Plots data with connector lines:
        ax.plot(x_values, y_values, z_values)
```

```
        # Finds the last value of arrays:
        x_last = x_values[-1]
        y_last = y_values[-1]
        z_last = z_values[-1]

        # Finds min/max y and z values:
        min_y_value = np.amin(y_values)
        max_y_value = np.amax(y_values)
        min_z_value = np.amin(z_values)
        max_z_value = np.amax(z_values)

        min_y_array.append(min_y_value)
        max_y_array.append(max_y_value)
        min_z_array.append(min_z_value)
        max_z_array.append(max_z_value)

        if x_last < 0:
            ax.scatter(x_last, y_last, z_last, c = 'red')

        elif x_last > thickness:
            ax.scatter(x_last, y_last, z_last, c = 'blue')

        else:
            ax.scatter(x_last, y_last, z_last, c = 'orange')

    #-------------------------------------------------------------------------

    # Plot slab:

    # Gets min and max values of all y / z values:
    min_y = np.amin(min_y_array)
    max_y = np.amax(max_y_array)

    min_z = np.amin(min_z_array)
    max_z = np.amax(max_z_array)

    y = np.linspace(min_y, max_y, 2)
    z = np.linspace(min_z, max_z, 2)

    Y, Z = np.meshgrid(y, z)

    # Add planes for slabs:
    ax.plot_surface(0, Y, Z, color = 'lightgrey', alpha = 0.3)
    ax.plot_surface(thickness, Y, Z, color = 'lightgrey', alpha = 0.3)

    #-------------------------------------------------------------------------

    # Formatting, saving and displaying plots:
    ax.set_xlabel(r'{}'.format(x_label) )
    ax.set_ylabel(r'{}'.format(y_label) )
    ax.set_zlabel(r'{}'.format(z_label) )

    # If ticks sizes are >= 1e4, ticks are changed to scientific:
    ax.xaxis.set_major_formatter(formatter)
    ax.yaxis.set_major_formatter(formatter)
```

```
    ax.zaxis.set_major_formatter(formatter)

    ax.view_init(azim = view_angle)

    ax.legend(loc = 'best', framealpha = 0.9)

    plt.tight_layout(pad = 0.5)
    plt.savefig('{}-graph.png'.format(save_name), dpi = 300 )

    return plt.show()

#-------------------------------------------------------------------------
```

C Appendix: Stats.py file:

```
"""
-------------------------------------------------------------------------

Statistical functions - Linear fitting / chi squared

Monty Kirner - 10301768 - 01/05/20 - University of Manchester

-------------------------------------------------------------------------

This code contains statistical tools such as least squares fitting and
chi-squared analysis.

-------------------------------------------------------------------------
"""
#-------------------------------------------------------------------------
# Initialisation
#-------------------------------------------------------------------------

import numpy as np
import pandas as pd

#-------------------------------------------------------------------------
# Functions
#-------------------------------------------------------------------------

def chi_squared(x_data, y_data, y_err, y_fit):
    ''' Returns chi squared for a polynomial fit of input data.
    '''
    return np.sum( ( (y_data - y_fit) / y_err)**2 )

#-------------------------------------------------------------------------

def value_rounder(value, error_in_value):
    ''' Rounds final value to 1 sig fig of error. Returns these values.
    '''
    error_sig_fig = float('{:.1g}'.format(error_in_value))
```

```
    # Turns error into a string with 6 d.p:
    error_sig_fig_str = str( '{:f}'.format(error_sig_fig))

    # If number is greater than zero:
    if '.000000' in error_sig_fig_str:

        # Gets rid of zeros after dp:
        after_decimal_point_removed = error_sig_fig_str.replace('.000000', '')

        # Finds number of sig figs:
        sig_fig_number = len(after_decimal_point_removed)

        # Rounds calculated value to same as error sig fig:
        value_sig_fig = round(value, -(sig_fig_number - 1) )

    else:
        # Rounds calculated value to same as error sig fig:

        # Finds number of decimal places:
        dp_number = len( str(error_sig_fig) ) - 2

        # Rounds calculated value to same as error sig fig:
        value_sig_fig = round(value, (dp_number) )

    return value_sig_fig, error_sig_fig

#-------------------------------------------------------------------------------

def linear_fit(x_data, y_data, y_error, data_name):
    ''' Performs linear LS fitting of data and finds fitting parameters.
    Returns fitting parameters and array of values for LOBF.
    '''
    weights = 1 / y_error

    fit_parameters, fit_errors = np.polyfit(x_data, y_data, 1,
                                            cov = True, w = weights)
    # Create set of fit values for graph:
    y_fit = np.polyval(fit_parameters, x_data)

    # The fit parameters are returned in fit_parameters:
    fit_m = fit_parameters[0]
    fit_c = fit_parameters[1]

    # The errors are returned in fit_errors:
    fit_sigma_m = np.sqrt(fit_errors[0][0])
    fit_sigma_c = np.sqrt(fit_errors[1][1])

    #---------------------------------------------------------------------------

    # Computes chi squared:
    fit_chi_squared = chi_squared(x_data, y_data, y_error, y_fit)

    # Computes reduced chi squared where denom = number of data points - 2,
    # since two degrees of freedom, a & b for linear fit:
    fit_reduced_chi_squared = ( fit_chi_squared / ( len(y_data) - 2) )
```

```
    #-------------------------------------------------------------------------

    # Rounds all values to 1 sig fig of error:
    fit_m_rounded, fit_sigma_m_rounded = value_rounder(fit_m, fit_sigma_m)
    fit_c_rounded, fit_sigma_c_rounded = value_rounder(fit_c, fit_sigma_c)

    #-------------------------------------------------------------------------

    # Summarise and print the results for the fit:
    print('\n' + '-' * 70)
    print('\nResults for linear fit of {} data:'.format(data_name) )
    print('\nGradient,  m = {:g} ± {:g}'.format(fit_m_rounded,
                                              fit_sigma_m_rounded) )
    print('Intercept, c = {:g} ± {:g}'.format(fit_c_rounded,
                                             fit_sigma_c_rounded) )
    print('\nReduced χ² = {:3.2f}\n'.format(fit_reduced_chi_squared) )
    print('-' * 70)

    #-------------------------------------------------------------------------

    fit_results_df = pd.DataFrame({
        'Gradient' : [fit_m, fit_sigma_m],
        'Intercept' : [fit_c, fit_sigma_c],
        'Reduced chi-squared' : [fit_reduced_chi_squared, 0]},
        index = ['Value', 'Error'] )

    return y_fit, fit_results_df

#-----------------------------------------------------------------------------
```

D Appendix: randssp.py file:

```
import numpy as np

#  RANDSSP Multiplicative congruential uniform random number generator.
#  Based on the parameters used by IBM's Scientific Subroutine Package.
#  The statement
#     r = randssp(m,n)
#  generates an m-by-n random matrix.
#  The function can not accept any other starting seed.
#
#  This function uses the "bad" generator parameters that IBM
#  used in several libraries in the 1960's.  There is a strong
#  serial correlation between three consecutive values.

def randssp(p,q):

    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
```