Simulation of neutron transmission through materials

Jonathan Ryding

10061683

School of Physics and Astronomy

The University of Manchester

Second year computing report

May 2019

Abstract

Simulated neutron transmission, reflection and absorption properties of water, lead and graphite were compared. Using 10cm slabs of each modelled material, the simulated absorption, reflection and transmission values respectively were 20.10 ± 0.39 %, 80.09 ± 0.43 %, 0.33 ± 0.05 % for water; 10.03 ± 0.34 %, 62.65 ± 0.46 %, 28.56 ± 0.37 % for lead; 0.80 ± 0.08 %, 68.50 ± 0.38 %, 30.79 ± 0.37 % for graphite.

1. Introduction

In particle physics, there is a lot of work in modelling how material properties and geometry affect net behaviour of penetrators. In this project, Monte Carlo methods are used so simulate the random properties of neutron absorption, reflection and transmission [1,2].

For example, the chain reaction of uranium-235 in nuclear reactors requires that the number of neutrons are controlled. This is done using neutron shields. Common materials used are water, lead and graphite [3].

2. Theory

The mean free path is the average distance travelled by a particle until it a collision, where an event, for example scattering, occurs. From the microscopic cross-section and particle number density product, there is the macroscopic cross-sectional area for an event to occur.

$$\lambda = \frac{1}{\Sigma} = \frac{1}{n\sigma} = \frac{M_A N_A}{e\sigma} \tag{1}$$

For multiple events, for example where particle scattering or absorption can occur, the total mean free path is the average distance travelled until either event occurs. The probability of scattering is given by the macroscopic cross-section of scattering per total macroscopic cross section, which is given by the sum of all macroscopic cross sections.

$$\begin{aligned} \Sigma_T &= \Sigma_s + \Sigma_a + \cdots \\ \frac{1}{\lambda_T} &= \frac{1}{\lambda_s} + \frac{1}{\lambda_a} + \cdots \end{aligned} \tag{2}$$

The probability that a particle travels a distance and then an event occurs follows an exponential distribution:

$$p(x) = e^{-\frac{x}{\lambda_T}} \tag{3}$$

It will be assumed that a neutron will randomly scatter in any direction.

The probability of a specific event occurring after travelling is given by:

$$P(a) = \frac{\Sigma_a}{\Sigma_a + \Sigma_b + \cdots} \tag{4}$$

3. Computational approach

Using Python 3.7, neutrons were simulated by using a chain of steps of random length x_i starting from an origin position where neutrons are normal to the surface. Steps lengths are drawn such that

$$x_i = -\lambda_T \, ln \, u_i \tag{5}$$

where u_i is a random number generated between zero and one. This follows from (3).

Steps measured were modelled to be isotropic in all directions of a 3-dimensional system. This required generating uniformly distributed numbers on a sphere. This was done by generating spherical polar coordinate angles θ, ϕ from random distributions:

$$\begin{aligned} \phi_i &= 2\pi u_i \\ \theta_i &= arccos(1 - 2u_i) \end{aligned} \tag{6}$$

A sphere mapped from uniform ϕ_i and θ_i produces a higher density of points towards the poles. Above random distributions (6) mapped produce a uniform sphere, suitable for modelling isotropic scattering.

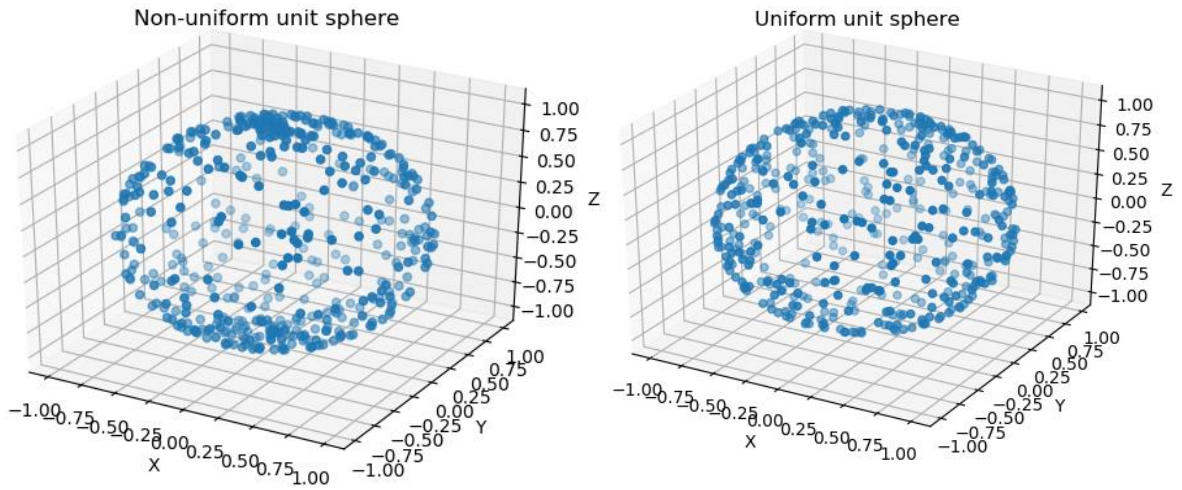

Figure 1. 3D diagrams showing the generation of non-uniform (left) and uniform (right) random numbers on a unit sphere. In the non-uniform sphere, the concentration at the poles is shown.

Slabs of materials were modelled by considering the position of the neutron relative to its starting point at the origin. If a neutron leaves bounds set for example, $0 < x < L$, the neutron has been either reflected or transmitted, depending on the last known position. Examples of each are shown below in Figure 3. It is assumed that the mean free path outside of the slab is infinite so a material that has left the slab cannot renter and undergo a different process. Before moving another step, it is decided where the neutron will now scatter or be absorbed according to Equation (4). If a randomly generated

number is less than $P(absorption)$, then the neutron will be absorbed after travelling that step length. A tally was kept of the event that happens to each neutron to calculate the percentage absorption, reflection and transmission of the material.

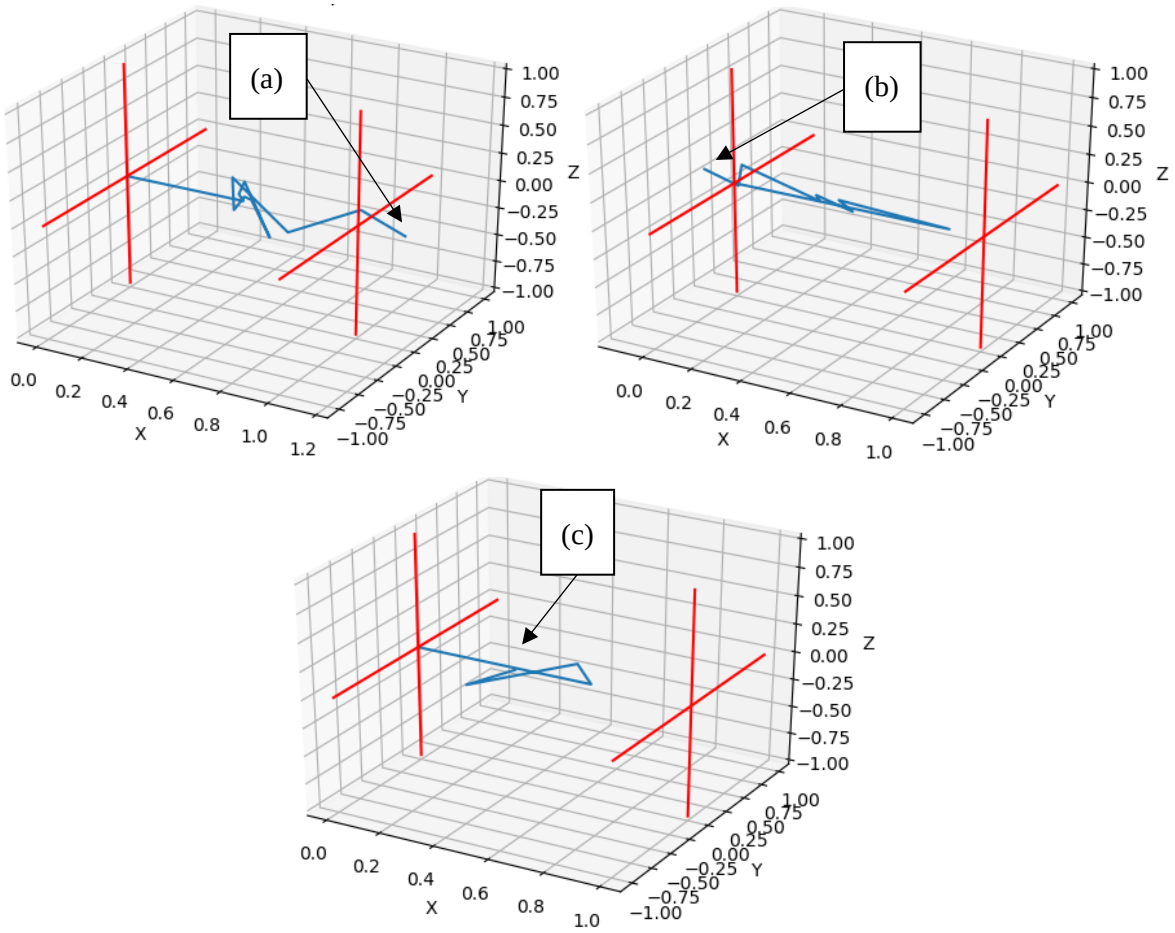

Figure 2. 3D diagrams showing the random walks of neutrons that have been transmitted, reflected and absorbed within a slab of modelled water, of thickness 1 cm.

Using the path lengths calculated by Equation (5) and a known mean free path, a method to calculate attenuation of a material when scattering is included was developed. A neutron with a mean free path of 45cm was used to produce a step length histogram shown below in Figure 3. If a step lies in the range of a bin, it is tallied into that bin. For small bin width, the error due to discreteness is negligible.

The measured mean free path from the linear fit (Figure 4, below) is calculated to be 44.36 ± 3.27 cm, corresponding well to the 45cm used to produce the data. Errors on each point were estimated through $\sqrt{N}$ for large N. The total error on the gradient is given by the square root of the covariance matrix index, calculated by the numpy.polyfit function.

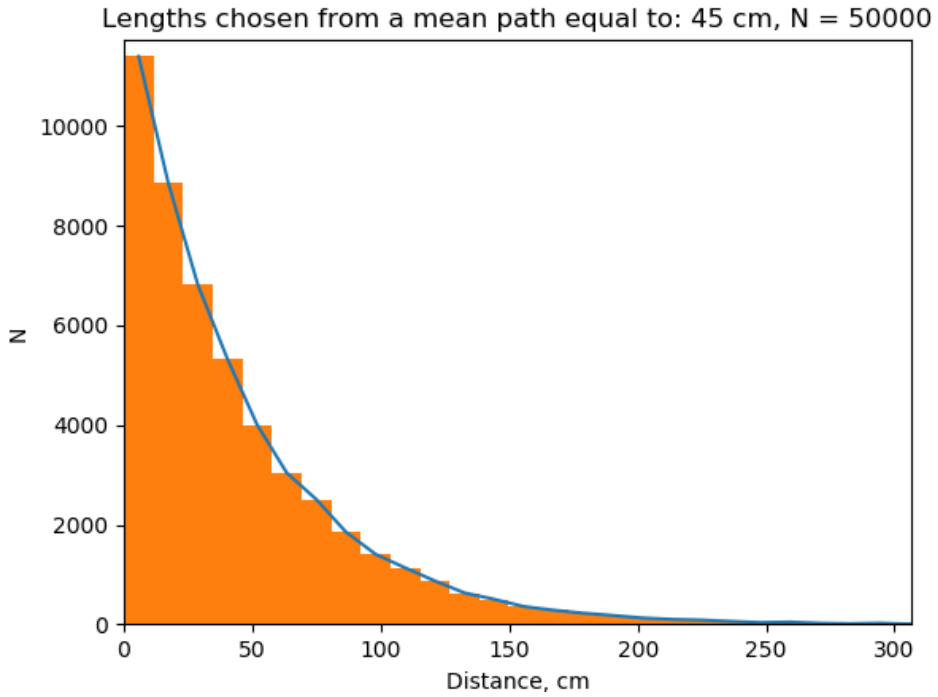

Figure 3. A histogram (shaded) of counted N vs. distance, produced by Equation (5). Lines join the bin centres to produce a curve.

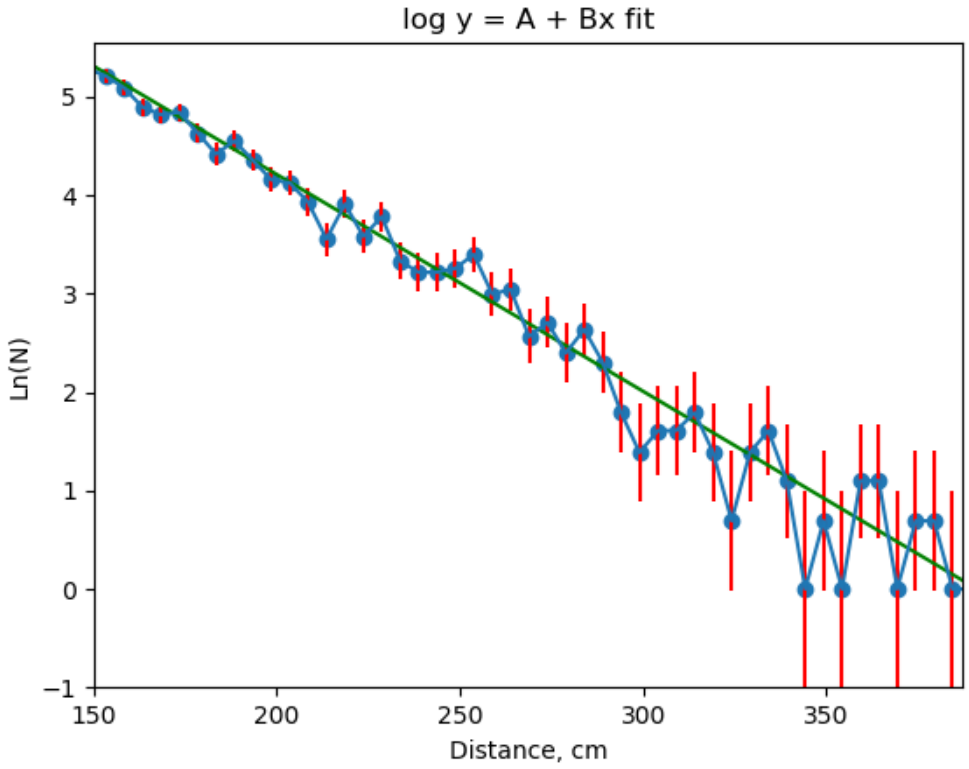

Figure 4. A linear fit to ln(N), from Figure 2. Before 250cm, the line fits well but beyond there is an error due to discrete counting at step lengths. This is noticeable by the increasingly poor fit. This has reduced chi-squared: 1.36. The mean free path is equal to the negative reciprocal of the gradient.

To measure the attenuation length of material slabs, this method is applied to the model including scattering, analysing the number of neutrons transmitted through a slab.

To estimate the error on the number transmitted, the main options are to calculate it indirectly using $\sqrt{N}$, but this is only true for large N. In the Results section, the largeness of N required to meet this condition is discussed. Alternatively, the simulation can be run a high enough number of times to directly calculate the standard deviation of results from the mean. This method is the most reliable for trueness of error but makes the program much more CPU intensive.

4. Results

Below, Figure 4. and Figure 5. show the variation of errors with the number of neutrons simulated. Errors were calculated using the standard deviation and through the normal distribution estimation. Generally, the normal distribution appears to be an overestimate on the error. The normal error tends to the standard deviation for large N. The standard deviation tends to a true representation of the random error for many repeats. To reduce CPU usage of the program, the normal error could be used to estimate the percent error at N larger than 5000, with the estimate improving with large N.

Figures 6, 7, 8. below show the absorption, reflection and transmission percentage variation with slab length of each of the three modelled materials: water, lead and graphite. Figure 9. below shows a linear reduced chi-squared fits to the transmission from each material.

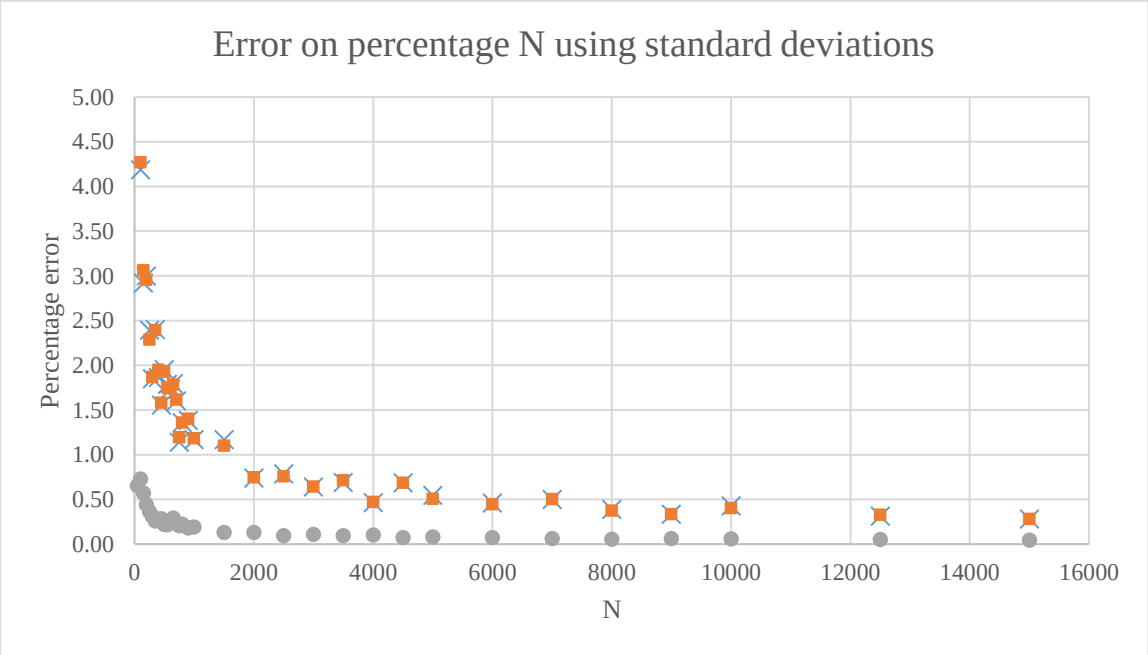

Figure 4. Percentage errors measured by repeating the program 50 times for water. Fractions of reflection: squares, absorption: crosses and transmission: circles.

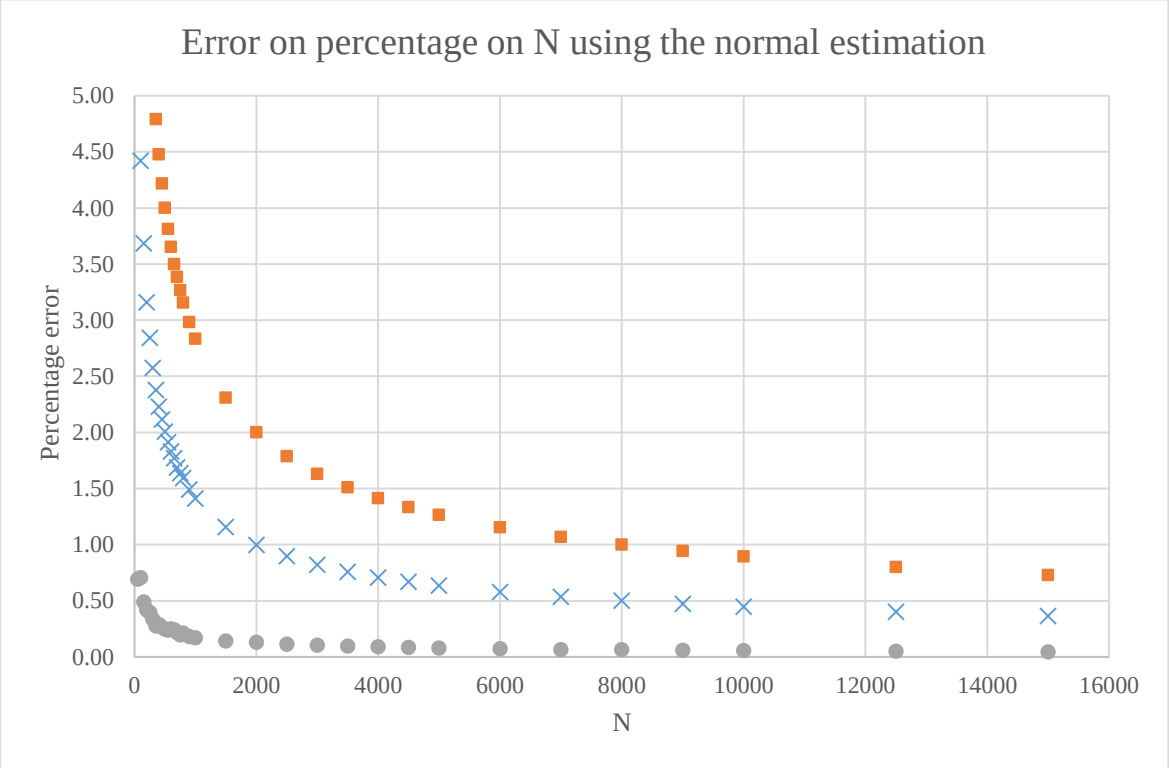

Figure 5. Percentage errors estimated by the normal distribution on the same axis scales as Figure 4. for comparison. Fractions of reflection: squares, absorption: crosses and transmission: circles.

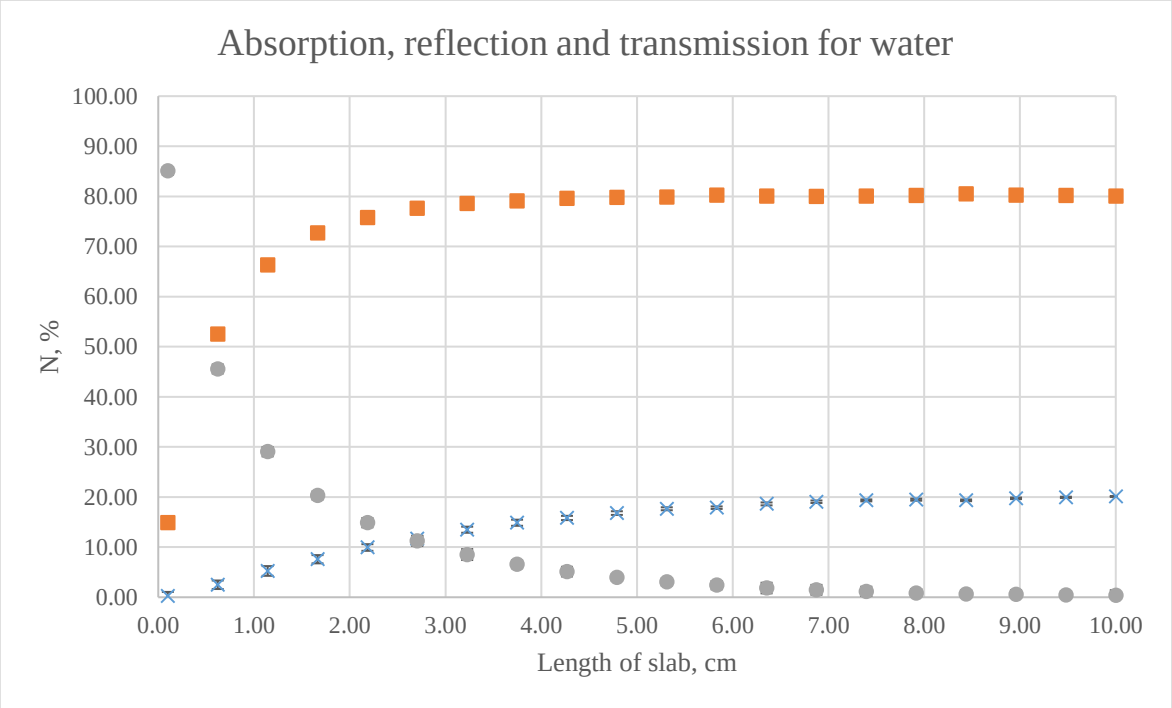

Figure 6. Absorption, reflection and transmission percentages with varying water slab lengths. Percentage reflection: squares, absorption: crosses and transmission: circles.

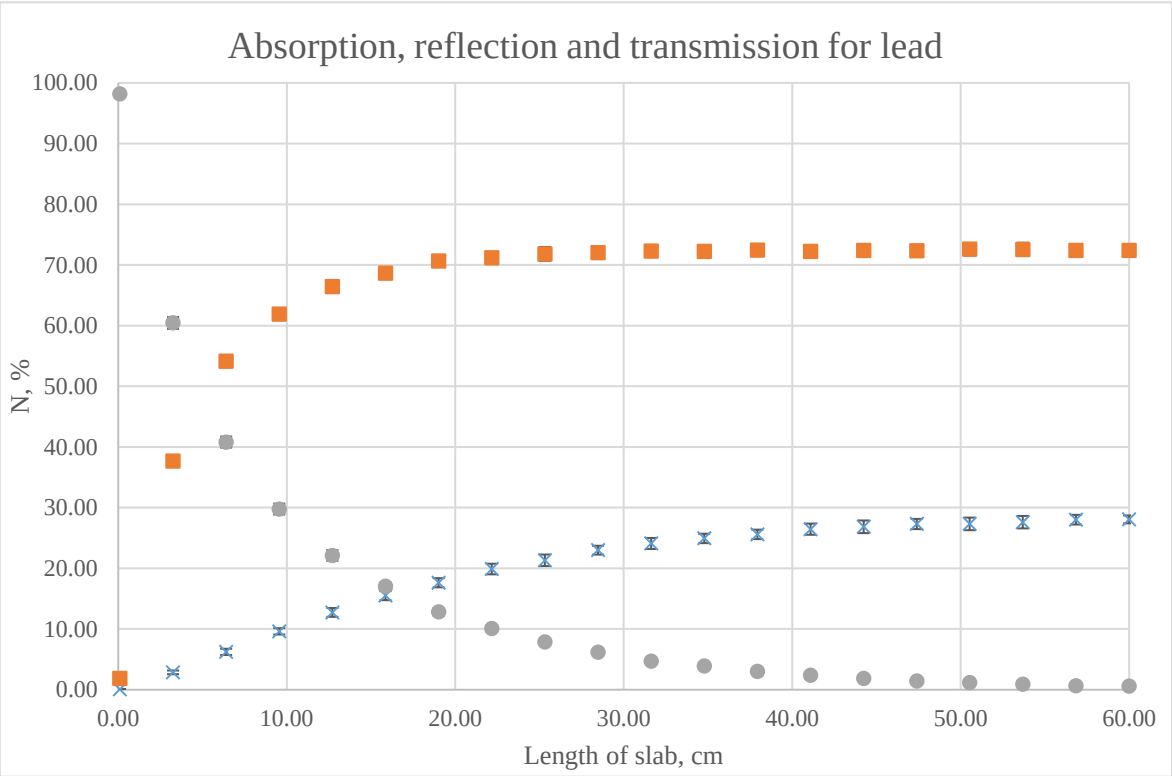

Figure 7. Absorption, reflection and transmission percentages with varying water slab length. Percentage reflection: squares, absorption: crosses and transmission: circles.

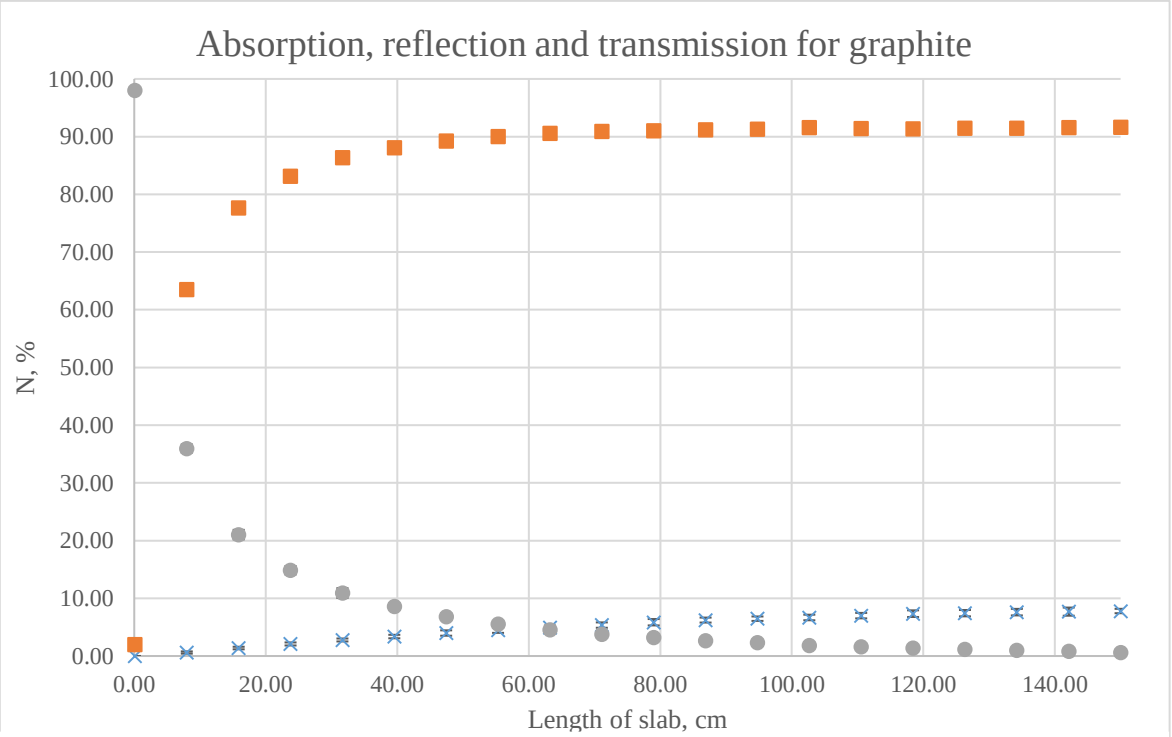

Figure 8. Absorption, reflection and transmission percentages with varying water slab length. Percentage reflection: squares, absorption: crosses and transmission: circles.

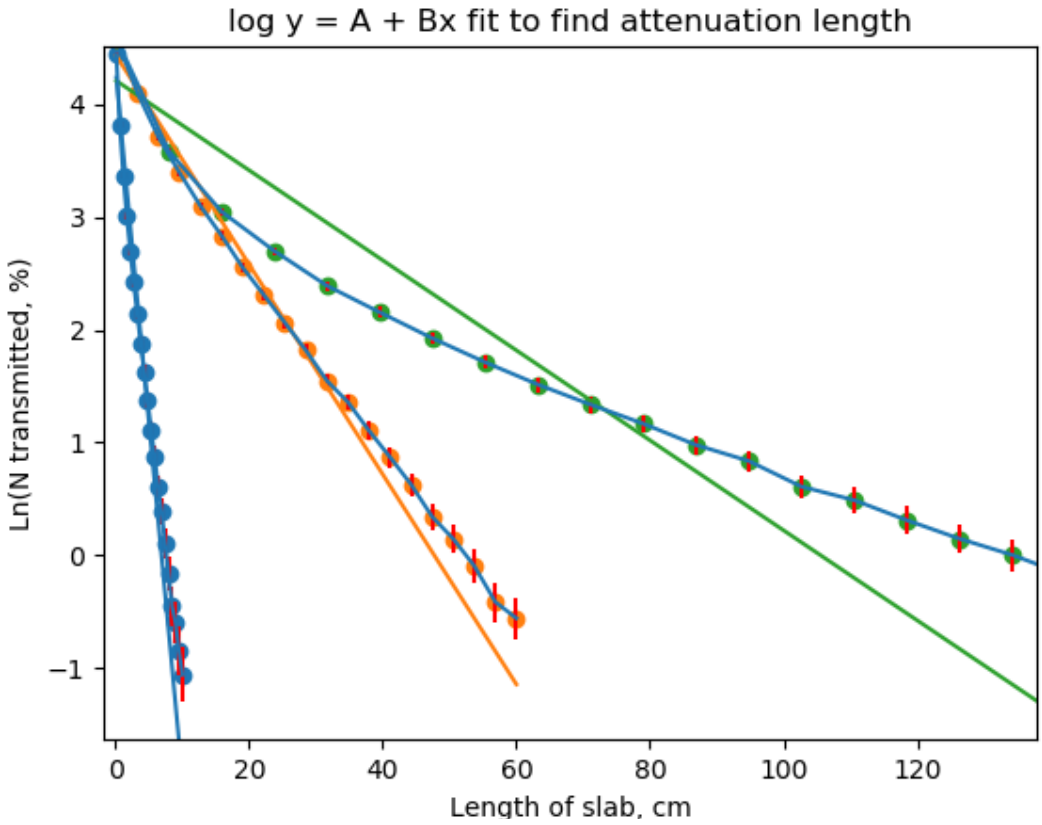

Figure 9. A linear fit to the natural log of the percentage transmission varying with slab lengths for each material. From left to right, water (steepest, blue), lead (orange) and graphite (most shallow, green).

The exponential fit becomes increasingly poor for the greater attenuation lengths measured. For larger attenuation lengths, the exponential fit to the transmission percentage is poorer. This is reflected by the increasingly reduced chi-squared values across the substances, summarised in Table 1. below.

Material	Mean free path of neutrons calculated, cm	Attenuation length, cm	Reduced chi-squared
Water	0.29	1.60 ± 0.34	17.83
Lead	2.66	10.75 ± 1.98	27.82
Graphite	2.52	25.06 ± 7.55	118.96

Table 1. A summary of the attenuation lengths of each material determined through the linear fits to the transmission in Figure 9.

At 10 cm, 10000 neutrons were simulated to calculate the absorption, reflection and transmittance percentages of each material. These are shown in Table 2. below.

Slab material, 10 cm length	Percentage absorbed (%)	Percentage reflected (%)	Percentage Transmitted (%)	$\sigma_{absorption}$ (Barns)	$\sigma_{scattering}$ (Barns)
Water	20.10 ± 0.39	80.09 ± 0.43	0.33 ± 0.05	0.6652	103.0
Lead	10.03 ± 0.34	62.65 ± 0.46	28.56 ± 0.37	0.158	11.22
Graphite	0.80 ± 0.08	68.50 ± 0.38	30.79 ± 0.37	0.0045	4.74

Table 2. A summary of the absorption, reflection and transmittance percentages of each material determined.

For this length of slab, water is the best choice of a material to absorb, reflect and prevent transmission of neutrons. Graphite offers the lowest absorption percentage at this length.

5. Conclusions

The absorption, reflection and transmission percentages of 10 cm slabs of water, lead and graphite were determined using a simulation of isotropic particle scattering and known values of microscopic cross-sections of absorption and scattering. The material attenuation lengths were determined to be 1.60 ± 0.34 cm, 10.75 ± 1.98 cm and 25.06 ± 7.55 cm, each respectively.

References

[1] Metropolis, N. & Ulam, S., "The Monte Carlo Method", *Journal of the American Statistical Association*, Volume 44, Issue 247, 1949.

[2] K.S. Krane, *Introductory Nuclear Physics*, Wiley, 1988.
[3] G.F. Knoll, *Radiation Detection and Measurement*, Wiley, 1989.

Appendix

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 15 12:54:45 2019
Jonathan Ryding
2nd yr Computational Physics, Project 3: Neutron attenuation through Monte Carlo methods
This program models neutrons using random numbers from distributions to model neutron absorption, reflection and transmission of materials.

@author: Jonathan
"""

#Use of random numbers and maths
import numpy as np
# and graphs
import matplotlib.pyplot as plt



#other useful packages for this project:
import math
from mpl_toolkits.mplot3d import Axes3D
import time



def randssp(p,q):
    #  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.

    #used to see spectral planes problem
    global m, a, c, x

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

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

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

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

    return r

def step_length(mean_free_path):
    #function generates a step length from the exponential distribution to model steps of neutron.
    step = -mean_free_path * np.log(np.random.uniform())
    return step

def scatter_vector(track):
    #function to generate the isotropic unit vectors to model scattering
    u = np.random.uniform()
    v = np.random.uniform()
    x, y, z = 0, 0, 0

    thi = 2 * np.pi * u #transformation to produce a uniform sphere
    theta = np.arccos(1 - 2*v)

    x = np.sin(theta) * np.cos(thi) #converted thi, theta into x,y,z cartesians,

    if track is True: #y, z vectors not needed to be calculated if not being tracked as slab is being modelling infinite
        y = np.sin(theta) * np.sin(thi)
```

```
    z = np.cos(theta)


  return x ,y, z


def simulate(x_section_a, x_section_s, Mmolar, density, n_neutrons, n_times, length, track):
  #function that uses earlier defined step_length and scatter_vector to simulate and count the absorption, reflection and transmission of materials.
  #Input the qualities of the material to model different material.


  n =  density * NA / Mmolar #(10^23 cm^2))number density of water cm^-3
  Za = n* x_section_a *0.1  #macroscopic absoprtion cross section of water, cm^-1
  Zs = n* x_section_s *0.1
  Ztot = Za +Zs


  total_mean_free_path =  1/Ztot #calculate mean free path
  Nt_array = np.zeros(Q) #initial empty arrays for the function to add to as the simulation of n_neutrons progresses. Repeated n_times, results from each are stored in indices of each array.
  Na_array = np.zeros(Q)
  Nr_array = np.zeros(Q)


  for j in range(0, n_times):
    print("Run" + str(j+1) ) #gives user an idea of how long each run takes, how many left until the program finishes.
    Nt = 0
    Na = 0
    Nr = 0


    for i in range(0, n_neutrons): #for N neutrons
```

```
Xpos0 = 0 #initial position at origin

Xpos1 = step_length(total_mean_free_path) #first step normal to surface, of random length drawn from exponential distribution


if track is True:

    Xpos = []

    Ypos = []

    Zpos = []

    Ypos0 = 0

    Zpos0 = 0


    Xpos.append(Xpos0)

    Ypos.append(0)

    Zpos.append(0)

    Xpos.append(Xpos1)

    Ypos.append(0)

    Zpos.append(0)


while (0 < Xpos1 < length): #while the neutron is in the slab, follow steps below


    Xunit, Yunit, Zunit = scatter_vector(track) #isotropic unit vectors, (normalised so that x^2 + y^2 +z^2 = 1, for a unit sphere.)

    R = step_length(total_mean_free_path) #length of step, equal to sphere


    Xstep = R * Xunit

    Ystep = R * Yunit

    Zstep = R * Zunit #x^2 + y^2 +z^2 = R^2 true
```

```
            Xpos0 = Xpos1  #update previous position
            Xpos1 = Xpos0 + Xstep #update new position

            if track is True:

                Xpos1 = Xpos0 + Xstep #update new position
                Ypos1 = Ypos0 + Ystep #update new position
                Zpos1 = Zpos0 + Zstep #update new position

                Xpos.append(Xpos1)
                Ypos.append(Ypos1)
                Zpos.append(Zpos1)

            if np.random.uniform() > x_section_s/(x_section_s + x_section_a): #does the neutron get absorbed after moving to the new step?

                #probability of being absorbed is macro x section of absorption / macro x sections summed (scatter & absorption)
                Na = Na + 1
                #print("absorbed")
                break # absorbed, end while loop

            #if it has left the slab, while loop ends and a tally is added to the relevant group
            #if not, while loop restarts
        #if the position is past the slab, it has been transmitted
        #if the position is outside the slab, it has been reflected.
        #assuming mean free path outside the slab is infinite.
```

```
    #print("left")

    if (Xpos1 > length):
        #print("trans")
        Nt = Nt + 1

    if (Xpos1 < 0):
        #print("reflected")
        Nr = Nr + 1

Nt_array[j] = Nt
Na_array[j] = Na
Nr_array[j] = Nr

if track is True:
    #tracking option to produce plots of the neutrons through the material.
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    # For each set of style and range settings, plot n random points in the box
    # defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].

    ax.plot(Xpos, Ypos, Zpos)
    ax.plot([0, 0, 0], np.linspace(-1,1,3), [0,0,0], color = "r")
    ax.plot([0, 0, 0], [0,0,0], np.linspace(-1,1,3), color = "r")

    ax.plot([length, length, length], np.linspace(-1,1,3), [0,0,0], color = "r")
    ax.plot([length, length, length], [0,0,0], np.linspace(-1,1,3), color = "r")
```

```
            ax.set_xlabel('X')
            ax.set_ylabel('Y')
            ax.set_zlabel('Z')
            plt.title("Simulated neutron path")
            plt.show()

    return Nt_array, Na_array, Nr_array

def errorcomparison(x_section_a, x_section_s, Mmolar, density, n_neutrons, n_times, length):
#varying number of neutrons  with an array, using indices and loops
    I = len(n_neutrons)

    #comparing normal error with stdev error at varying n_neutrons
    percent_NtErrNormal = np.zeros(I)
    percent_NaErrNormal = np.zeros(I)
    percent_NrErrNormal = np.zeros(I)

    percent_NtErrStd = np.zeros(I)
    percent_NaErrStd = np.zeros(I)
    percent_NrErrStd = np.zeros(I)


    for i in range(0,I):
      print("again")
```

```
    Nt_array, Na_array, Nr_array = simulate(x_section_a, x_section_s, Mmolar, density, n_neutrons[i], n_times, length, False)


    #estimates of variance
    percent_NtErrNormal[i] = 100*np.sqrt(np.mean(Nt_array))/n_neutrons[i]
    percent_NaErrNormal[i] = 100*np.sqrt(np.mean(Na_array))/n_neutrons[i]
    percent_NrErrNormal[i] = 100*np.sqrt(np.mean(Nr_array))/n_neutrons[i]


    percent_NtErrStd[i] = 100* np.std(Nt_array)/n_neutrons[i]
    percent_NaErrStd[i] = 100*np.std(Na_array)/n_neutrons[i]
    percent_NrErrStd[i] = 100*np.std(Nr_array)/n_neutrons[i]
  """
  plt.scatter(n_neutrons, percent_NtErrNormal)
  plt.plot(n_neutrons, percent_NtErrStd)
  plt.show()


  plt.scatter(n_neutrons, percent_NaErrNormal)
  plt.plot(n_neutrons, percent_NtErrStd)
  plt.show()


  plt.scatter(n_neutrons, percent_NtErrNormal)
  plt.plot(n_neutrons, percent_NrErrStd)
  plt.show()
  """
  return percent_NtErrNormal, percent_NaErrNormal, percent_NrErrNormal, percent_NtErrStd, percent_NaErrStd, percent_NrErrStd
```

```
def attenuation(x_section_a, x_section_s, Mmolar, density, n_neutrons, n_times, length):


#varying lengths  with an array, using indices and loops
    I = len(length)
    print(I)
# if n is larger than 5000, could just use normal error to speed up calculation


    percent_Nt = np.zeros(I)
    percent_Na = np.zeros(I)
    percent_Nr = np.zeros(I)


    percent_NtErrNormal = np.zeros(I)
    percent_NaErrNormal = np.zeros(I)
    percent_NrErrNormal = np.zeros(I)


    percent_NtErrStd = np.zeros(I)
    percent_NaErrStd = np.zeros(I)
    percent_NrErrStd = np.zeros(I)



    for i in range(0,I):
      print("again")


     Nt_array, Na_array, Nr_array = simulate(x_section_a, x_section_s, Mmolar, density, n_neutrons, n_times, length[i], False)
                                    #estimates of variance
```

```
    percent_Nt[i] = 100*(np.mean(Nt_array))/n_neutrons
    percent_Na[i] = 100*(np.mean(Na_array))/n_neutrons
    percent_Nr[i] = 100*(np.mean(Nr_array))/n_neutrons

    percent_NtErrNormal[i] = 100*np.sqrt(np.mean(Nt_array))/n_neutrons
    percent_NaErrNormal[i] = 100*np.sqrt(np.mean(Na_array))/n_neutrons
    percent_NrErrNormal[i] = 100*np.sqrt(np.mean(Nr_array))/n_neutrons

    percent_NtErrStd[i] = 100* np.std(Nt_array)/n_neutrons
    percent_NaErrStd[i] = 100*np.std(Na_array)/n_neutrons
    percent_NrErrStd[i] = 100*np.std(Nr_array)/n_neutrons


  return percent_Nt, percent_Na, percent_Nr, percent_NtErrNormal, percent_NaErrNormal, percent_NrErrNormal, percent_NtErrStd, percent_NaErrStd, percent_NrErrStd




def reducedChiSquared(Ydata, Yfit, Yerr, Nparameters):
#calculates the reduced chi-squared of a general fit to data step-by-step by summing the square of each residual and dividing by the number of degrees of freedom of the fit.
#Yfit is the array of velocities from the fitted curve at the same angle as Ydata.
#chi-squared is an indicator of how appropriate the fit is to the data.

  chiold = 0 #initial chi-squared value begins at 0.

  for i in range(0, len(Ydata)):
```

```
        #calculating the  weighted sum of the residuals squared for each data point

        chi_step = pow( ((Ydata[i] - Yfit[i])/Yerr[i]) , 2 )
        chiSquared = chi_step + chiold
        chiold = chiSquared

    reduChiSquare = (chiSquared / (len(Ydata) - Nparameters) )
    #To calculate the reduced chi-squared, divide the chi-squared by the number of degrees of freedom. This is equal to the number of data points used minus the number of parameteres of the fit.

    return reduChiSquare

def nozeros(array_required, array2, array3):
    #function that removes the indices across 3 arrays to remove 0s from a target array

    non0 = np.nonzero(array_required) #labels indices of nonzeros to be saved

    new_array_required = np.zeros(len(non0)) #empty new ones
    new_array2 = np.zeros(len(non0))
    new_array3 = np.zeros(len(non0))

    for i in range(0, len(non0)): #loop across indices that want to be saved

        new_array_required = array_required[non0[i]]
        new_array2 = array2[non0[i]]
        new_array3 = array3[non0[i]]
```

```
    return new_array_required, new_array2, new_array3 #output new arrays


def plot3(Lplot, percent_Nt, percent_Na, percent_Nr, percent_NtErr, percent_NaErr, percent_NrErr):
    #plotting function


    plt.plot(Lplot, percent_Nt)
    plt.errorbar(Lplot, percent_Nt, yerr = percent_NtErr)


    plt.plot(Lplot, percent_Na)
    plt.errorbar(Lplot, percent_Na, yerr = percent_NaErr)


    plt.plot(Lplot, percent_Nr)
    plt.errorbar(Lplot, percent_Nr, yerr = percent_NrErr)


    plt.show()
    return
   #fit e^x/lambda to transmitted to get attenuation length
#remove  possible zero indices of arrays


def attenuationlength(Lplot, percent_Nt, percent_NtErr):
    #function to plot and analyse the fit to log N = A + B * length of slab


    percent_Nt_no0, L_no0, err_no0 = nozeros(percent_Nt, Lplot, percent_NtErr)
    err_no0, percent_Nt_no0, L_no0 = nozeros(err_no0, percent_Nt_no0, L_no0)
#allows use of stdev for low N
#removing potential breaks by zeros from plotting
```

```
logN = np.log(percent_Nt_no0)

#Could calculate error with stdev, but for N past 5000 it is approximately the normal error.
logNerr = np.zeros(len(logN))
for i in range(0, len(logN)):

    logNerr[i] = err_no0[i]/percent_Nt_no0[i]

#line fitting
(coef, covr) = np.polyfit(L_no0, logN, 1 , cov = True, w = (1/logNerr))
#weights used by polyfit are reciprocal of error
covr = np.array(covr).ravel()

fitline = np.zeros(len(L_no0))

for i in range(0, len(L_no0)):
     fitline =  coef[0]*L_no0 + coef[1]

#plotting fitted line to data



plt.plot(L_no0, percent_Nt_no0)
plt.xlabel("Length of slab, cm")
plt.ylabel("Ln(N transmitted, %)")
plt.errorbar(L_no0, percent_Nt_no0, yerr = err_no0, ecolor = "r")
plt.show()
```

```
    plt.scatter(L_no0, logN)
    plt.plot(L_no0, fitline)
    plt.title("log y = A + Bx fit to find attenuation length")
    plt.xlabel("Length of slab, cm")
    plt.ylabel("Ln(N transmitted, %)")
    plt.errorbar(L_no0, logN, yerr = logNerr, ecolor = "r")
    plt.show()

    #caulcate reduced chisquared
    reduChiSqu = reducedChiSquared(logN, fitline, logNerr, 2)

    A = - 1/coef[0]
    Aerr = A * np.sqrt( np.absolute(np.sqrt(covr[0])/coef[0] ))
    #output results
    print("Attenuation length: (%5.2f ± %5.2f)"%(A, Aerr) )
    print("The best fitted line is: log(percentNt)(L) = (%5.2f ± %5.2f) X + (%5.2f ± %5.2f)" %(coef[0], np.sqrt(covr[0]), coef[1], np.sqrt(covr[3]) ))
    print("This has reduced chi-squared: %5.2f" %reduChiSqu)

    return A, Aerr

N = 600

u, v, r, thi, theta, x, y, z = np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N)
```

```
for i in range(0, N):

    #using random numbers to generate thi, theta
    thi[i] = 2 * np.random.uniform() *np.pi
    theta[i] = np.random.uniform() *np.pi

    x[i] = np.sin(theta[i]) * np.cos(thi[i])
    y[i] = np.sin(theta[i]) * np.sin(thi[i])
    z[i] = np.cos(theta[i])
    #plotting a unit sphere

plt.scatter(thi,theta)
plt.title("log y = A + Bx fit to find attenuation length")
plt.xlabel("thi, azimuthal")
plt.ylabel("theta, polar angle")
plt.title("uniform thi, theta space")
plt.show()



fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(x, y, z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
```

```
plt.grid(b=None, which='major', axis='both')

plt.title("Non-uniform unit sphere")

plt.show()

#uniform thi-theta space produces a non-isotropic sphere


#preforming inverse transform sampling


s, u, v, thi, theta, x, y, z = np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N), np.zeros(N)


for i in range(0, N):


    s[i] = np.random.uniform()

    u[i] = np.random.uniform()

    v[i] = np.random.uniform()



    thi[i] = 2 * np.pi * u[i] #transformation to produce a uniform sphere

    theta[i] = np.arccos(1 - 2*v[i])


    x[i] = np.sin(theta[i]) * np.cos(thi[i]) #converted thi, theta into x,y,z cartesians,

    y[i] = np.sin(theta[i]) * np.sin(thi[i])

    z[i] = np.cos(theta[i])


R = np.zeros(N)


for i in range(0, N ):
```

```
        R[i] = np.sqrt( x[i]**2 + y[i]**2 +z[i]**2)
        #checking unit vectors


#plt.scatter(u,v)
#plt.title("Uniform random numbers, u,v test.")
#plt.show()


#plt.scatter(thi,theta)
#plt.title("Transformed random thi, theta")
#plt.show()


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
ax.scatter(x, y, z)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.title("Uniform unit sphere")
plt.grid(b=None, which='major', axis='both')
plt.show()


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
```

```
ax.scatter(s, u, v)

ax.set_xlabel('X Label, s')

ax.set_ylabel('Y Label, u')

ax.set_zlabel('Z Label, v')

plt.title("Checking random number generation")

plt.grid(b=None, which='major', axis='both')

plt.show()



#random path lengths taken from exponential distribution


M = 5000

S = np.zeros(M)

Lambda = 45 #cm

Bins = 50

for i in range(0, M):


    S[i] = - Lambda* np.log(np.random.uniform())



Y, X = np.histogram(S, bins = Bins)

X = X[1:] - X[-1]/(2*Bins) #move X to centre of bins, and plot against Y to show exponential distribution


plt.plot(X, Y)

plt.title("Lengths chosen from a mean path equal to: " + str(Lambda) + " cm, N = "  + str(M))
```

```
plt.xlabel("Distance, cm")
plt.ylabel("N" )
plt.show()
plt.hist(S, bins = Bins)
plt.show()

Y, X, Yerr = nozeros(Y, X, np.sqrt(Y))
logY = np.log(Y)

logYerr = np.zeros(len(logY))
for i in range(0, len(logY)):

    logYerr[i] = Yerr[i]/Y[i]

(coef, covr) = np.polyfit(X, logY, 1 , cov = True, w = (1/logYerr))
covr = np.array(covr).ravel()

fitline = np.zeros(len(X))

for i in range(0, len(X)):
    fitline =  coef[0]*X + coef[1]

plt.scatter(X, logY)
plt.plot(X, fitline, color = "g")
plt.title("log y = A + Bx fit to find attenuation length")
plt.xlabel("Length travelled, cm")
plt.ylabel("Ln(N)")
```

```
plt.errorbar(X, logY, yerr = logYerr, ecolor = "r")

plt.show()


reduChiSqu = reducedChiSquared(logY, fitline, logYerr, 2)


A = - 1/coef[0]

Aerr = A * np.sqrt( np.absolute(np.sqrt(covr[0])/coef[0] ))

print("Attenuation length: (%5.2f ± %5.2f)"%(A, Aerr) )

print("The best fitted line is: log(Y) = (%5.2f ± %5.2f) X + (%5.2f ± %5.2f)" %(coef[0], np.sqrt(covr[0]), coef[1], np.sqrt(covr[3]) ))

print("This has reduced chi-squared: %5.2f" %reduChiSqu)

"""


 #Example results:

Lambda = 1cm, M = 5000, Bins = 100

Attenuation length: ( 1.08 ±  0.14)

The best fitted line is: log(percentNt)(L) = (-0.93 ±  0.02) X + ( 6.06 ±  0.02)

This has reduced chi-squared:  1.22 (in range 0.8, 1.2)


Attenuation length: (46.40 ±  3.25)

The best fitted line is: log(Y) = (-0.02 ±  0.00) X + ( 6.32 ±  0.01)

This has reduced chi-squared:  1.13

You get what you put in out

"""


####################
```

#Main simulation of neutrons through water, lead and graphite

####################

NA = 6.0221 #avogadro, x10^23 global

N = 1000

Q = 5

L = 10 #cm

Na, Nr, Nt = 0,0,0

#A = np.polyfit(X, Y, deg = 15)

#taylor expansion e^-x/a = 1 - x/a +....

#log0 undefined, could take out all 0 points and fit a linear line: log y = C + Bx

#water

water_sigmaA = 0.6652 #barns

water_sigmaS = 103.0 #barns 10^-24 cm^2

water_density = 1.00 #g/cm^3

water_molar_mass = 18.01528 #g/mol

#lead

lead_sigmaA = 0.158

lead_sigmaS = 11.221

lead_density = 11.35

lead_molar_mass = 207.2

```
#graphite

graphite_sigmaA = 0.0045

graphite_sigmaS = 4.74

graphite_density = 1.67

graphite_molar_mass = 12.011



#outputs percent_NtErrNormal, percent_NaErrNormal, percent_NrErrNormal, percent_NtErrStd, percent_NaErrStd, percent_NrErrStd

"""

Nplot = [50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650] #700, 750, 800, 900, 1000, 1500, 2000, 2500, 3000, 3500 , 4000,4500, 5000, 6000, 7000, 8000, 9000, 10000, 12500, 15000]


print("starting water")

start = time.time()

Wpercent_NtErrNormal, Wpercent_NaErrNormal, Wpercent_NrErrNormal, Wpercent_NtErrStd, Wpercent_NaErrStd, Wpercent_NrErrStd = errorcomparison(water_sigmaA, water_sigmaS, water_molar_mass, water_density, Nplot, Q, L)

print("starting lead")

end = time.time()

timetaken = (end - start)

print(timetaken)

lead_percent_NtErrNormal, lead_percent_NaErrNormal, lead_percent_NrErrNormal, lead_percent_NtErrStd, lead_percent_NaErrStd, lead_percent_NrErrStd = errorcomparison(lead_sigmaA, lead_sigmaS, lead_molar_mass, lead_density, Nplot, Q, L)

print("staring graphite")

graphite_percent_NtErrNormal, graphite_percent_NaErrNormal, graphite_percent_NrErrNormal, graphite_percent_NtErrStd, graphite_percent_NaErrStd, graphite_percent_NrErrStd =
```

errorcomparison(graphite_sigmaA, graphite_sigmaS, graphite_molar_mass, graphite_density, Nplot, Q, L)

WLplot = np.linspace(0.1, 10, num = 20)

LLplot = np.linspace(0.1, 60, num = 20)

GLplot = np.linspace(0.1, 150, num = 20)

Wpercent_Nt, Wpercent_Na, Wpercent_Nr, Wpercent_NtErrNormal, Wpercent_NaErrNormal, Wpercent_NrErrNormal, Wpercent_NtErrStd, Wpercent_NaErrStd, Wpercent_NrErrStd = attenuation(water_sigmaA, water_sigmaS, water_molar_mass, water_density, N, Q, WLplot)

Lpercent_Nt, Lpercent_Na, Lpercent_Nr, Lpercent_NtErrNormal, Lpercent_NaErrNormal, Lpercent_NrErrNormal, Lpercent_NtErrStd, Lpercent_NaErrStd, Lpercent_NrErrStd = attenuation(lead_sigmaA, lead_sigmaS, lead_molar_mass, lead_density, N, Q, LLplot)

Gpercent_Nt, Gpercent_Na, Gpercent_Nr, Gpercent_NtErrNormal, Gpercent_NaErrNormal, Gpercent_NrErrNormal, Gpercent_NtErrStd, Gpercent_NaErrStd, Gpercent_NrErrStd = attenuation(graphite_sigmaA, graphite_sigmaS, graphite_molar_mass, graphite_density, N, Q, GLplot)

"""

print("water")

#plot3(WLplot, Wpercent_Nt, Wpercent_Na, Wpercent_Nr, Wpercent_NtErrNormal, Wpercent_NaErrNormal, Wpercent_NrErrNormal)

#WA, WAerr =attenuationlength(WLplot, Wpercent_Nt, Wpercent_NtErrNormal)

print("lead")

#plot3(LLplot, Lpercent_Nt, Lpercent_Na, Lpercent_Nr, Lpercent_NtErrNormal, Lpercent_NaErrNormal, Lpercent_NrErrNormal)

#LA, LAerr = attenuationlength(LLplot, Lpercent_Nt, Lpercent_NtErrNormal)

print("graphite")

#plot3(GLplot, Gpercent_Nt, Gpercent_Na, Gpercent_Nr, Gpercent_NtErrNormal, Gpercent_NaErrNormal, Gpercent_NrErrNormal)

```
#GA, GAerr = attenuationlength(GLplot, Gpercent_Nt, Gpercent_NtErrNormal)


#for 10 cm slabs of all

"""

WNt_array, WNa_array, WNr_array = simulate(water_sigmaA, water_sigmaS, water_molar_mass, water_density, N, Q, L, False)

LNt_array, LNa_array, LNr_array = simulate(lead_sigmaA, lead_sigmaS, lead_molar_mass, lead_density, N, Q, L, False)

GNt_array, GNa_array, GNr_array = simulate(graphite_sigmaA, graphite_sigmaS, graphite_molar_mass, graphite_density, N, Q, L, False)


#print("Transmitted %: " + str(100*np.mean(water_Nt_array)/N) + " normal error: " + str(100*NtErrNormal/N) + " std error: " + str(100*NtErrStd/N) )

#print("Absorbed %: " + str(100*np.mean(water_Na_array) / N) + " normal error: " + str(100*NaErrNormal/N) + " std error: " + str(100*NaErrStd/N))

#print("Reflected %: " + str(100*np.mean(water_Nr_array) / N) + " normal error: " + str(100*NrErrNormal/N) + " std error: " + str(100*NrErrStd/N))  #estimates of variance


#tracking a few path

#Wt_array, WNa_array, WNr_array = simulate(water_sigmaA, water_sigmaS, water_molar_mass, water_density, N, 1, 1, True)

Wt_array, WNa_array, WNr_array = simulate(graphite_sigmaA, graphite_sigmaS, graphite_molar_mass, graphite_density, N, Q, 1, True)


"""
```