Modelling Nuclear Scattering

James Tcheng (BJH)

Eton College Computational Physics Prize 2022

Table of Contents

All code for the project is linked here: https://github.com/doggie007/NuclearScattering

1. Introduction

1.1 Early Development of Atomic Theory

The idea of atoms dates back to 5th century BC in Ancient Greece, in fact the word atom comes from *atomos* in Ancient Greek and translates to "uncuttable". Many attribute the beginning of atomism to Democritus and his mentor Leucippus, who believed that everything is composed of atoms, which are tiny, uniform, incompressible, and indestructible objects that are always in motion through empty space. Atoms differ from one another by shape and size which determined the properties of matter. Though rudimentary, their ideas laid the foundation for future atomic models.

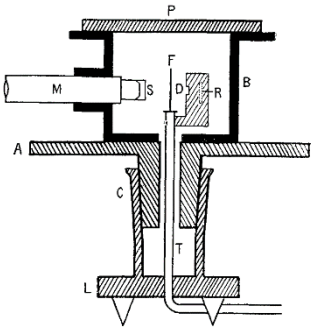
In 1803, chemist John Dalton drew upon the idea of the Ancient Greeks. Atoms of a given element are identical to each other, and compounds are a combination of different types of atoms in defined ratios. Chemical reactions resulted in the rearrangement of these atoms.

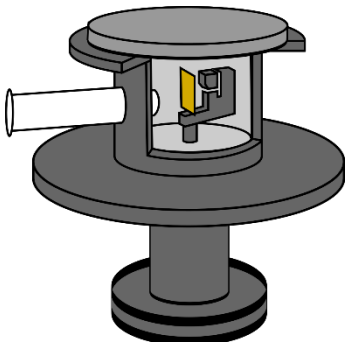
In 1897, J. J. Thomson discovered the electron, first indicating the atom had an internal structure. He proposed the "plum pudding" model in 1904, where positive charge is spread uniformly across the atom and tiny negatively charged electrons are distributed throughout. His model was widely accepted for a short period of time until his student Rutherford proved him wrong.

1.2 Rutherford Gold Foil Experiment

> "*It was quite the most incredible event that has ever happened to me in my life. It was almost as incredible as if you fired a 15-inch shell at a piece of tissue paper and it came back and hit you*."–Ernest Rutherford

Between 1908 and 1913, at the behest of Rutherford, Geiger and Marsden performed a series of experiments where they directed a narrow pencil of alpha particles emitted from a radium source at a thin metal foil through a vacuum, and measured the scattering pattern using a zinc-sulphide screen. Every time an alpha particle struck the fluorescent screen it produced a flash of light called a scintillation, which was viewable via a microscope attached to the back of the screen. The microscope could be rotated to observe the scattering in different directions.





According to Thompson's model, nearly all alpha particles should pass straight through, with the exception of tiny deflections being a result of accrued deflections from the passage through successive atoms. Surprisingly, a number of alpha particles underwent large angle scattering, some even diffusely reflected. This led Rutherford to propose a theory in 1911 that an atom consisted of a tiny, central concentrated charge with orbiting electrons, suggesting most of the atom was empty space.

2. Project Overview

2.1 Aims

We aim to model Rutherford scattering by applying Coulomb's law and Newton's equations of motion. Specifically, we aim to recreate the Geiger-Marsden experiments detailed in their 1913 paper named "The Laws of Deflexion of α Particles through Large Angles", which sought to verify the mathematical equations governing this type of scattering developed by Rutherford in his 1911 paper "The Scattering of α and β Particles by Matter and the Structure of the Atom". We will compare the results we achieve to Rutherford's analytical solutions.

As was experimented in the Geiger-Marsden experiments, we will mainly investigate whether the number of deflected α particles observed at a specified angle θ is proportional to:

1. $\csc^4\left(\frac{\theta}{2}\right)$
2. Thickness of the scattering material t
3. Square of the central charge Q
4. Velocity of α particles $\frac{1}{v^4}$

2.2 Mathematical details

2.2.1 Coulomb's Law

The force between two-point charges Q_1 and Q_2 separated by distance r is given by:

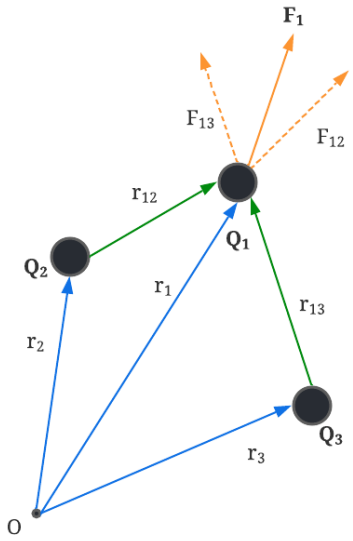
$$F = \frac{kQ_1Q_2}{r^2}$$

where $k = \frac{1}{4\pi\varepsilon_0}$ is Coulomb's constant

Written in vectorized form:

$$\vec{F} = \frac{kQ_1Q_2}{|r|^2}\,\hat{r} = \frac{kQ_1Q_2}{|r|^3}\,\vec{r}$$

where $\vec{r}$ is the position vector.



The force between multiple charges can be calculated by:

$$\overrightarrow{F_1} = \overrightarrow{F_{12}} + \overrightarrow{F_{13}} + \cdots + \overrightarrow{F_{1n}}$$

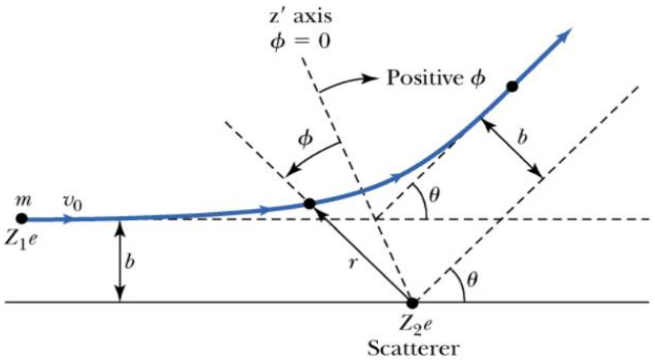
$$\overrightarrow{F_1} = \frac{kQ_1Q_2}{|r_{12}|^3}\overrightarrow{r_{12}} + \frac{kQ_1Q_3}{|r_{13}|^3}\overrightarrow{r_{13}} + \cdots + \frac{kQ_1Q_n}{|r_{1n}|^3}\overrightarrow{r_{1n}}$$

$$\overrightarrow{F_1} = kQ_1\sum_{i=2}^{n}\frac{Q_i}{|r_{1i}|^3}(\overrightarrow{r_1} - \overrightarrow{r_i})$$

(Note that $\overrightarrow{F_{12}}$ denotes the force exerted on $Q_1$by Q_2 and $\overrightarrow{r_{12}} = \overrightarrow{r_1} - \overrightarrow{r_2}$ is the vector from Q_2 to Q_1)

2.2.2 Rutherford Scattering Formula

In his 1911 paper, Rutherford derived a formula describing the number of α particles scattered from a thin metal foil at a given angle from the Coulomb force, which follows a hyperbolic trajectory as shown.



b is the impact parameter, θ is the scattering angle, v_0 is the starting velocity of the alpha particle, and Z is the atomic number of the respective particles

Rutherford's assumptions:

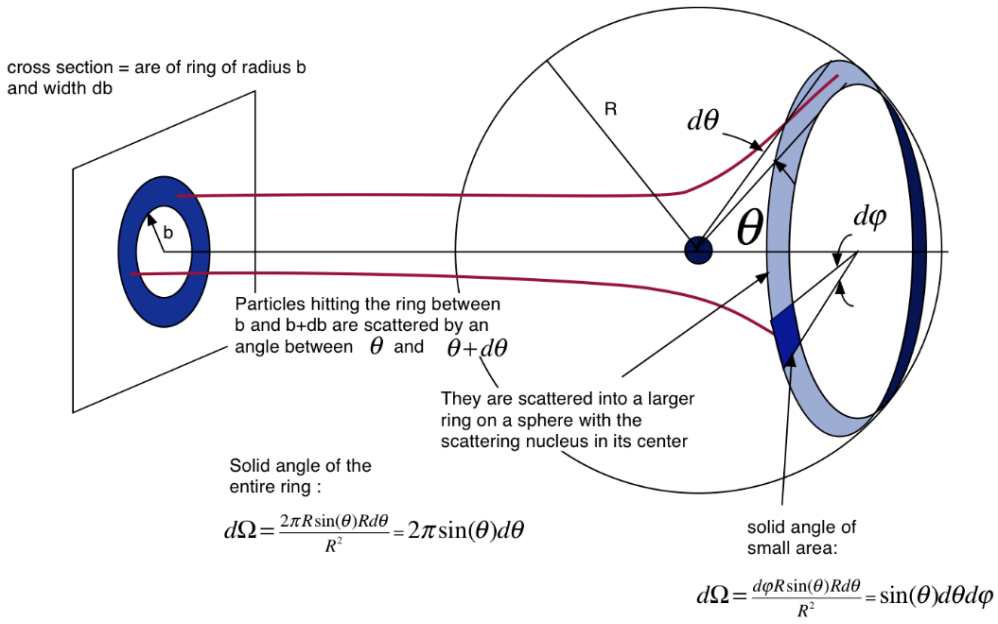
1) α particles and the nucleus are small enough to be treated as point masses and charges
2) Coulomb force between the charges is the only force present and laws of classical mechanics are applied
3) Nucleus is large enough that it does not recoil
4) Target is so thin that only a single scattering occurs

The relationship between impact parameter and scattering angle:

$$b(\theta) = \frac{kQ_1Q_2}{m{v_0}^2}\cot\left(\frac{\theta}{2}\right) \quad (1)$$

The "Rutherford scattering formula" can be written in the form of the differential cross section, which is the probability that a particle passing through an area $d\sigma$ before scattering can be found within the solid angle $d\Omega$ after scattering:

$$\frac{d\sigma}{d\Omega} = \frac{b}{\sin\theta}\left|\frac{db}{d\theta}\right| = \left(\frac{kQ_1Q_2}{2m{v_0}^2}\right)^2 csc^4\left(\frac{\theta}{2}\right) \quad (2)$$



Note. Image taken from: https://wanda.fiu.edu/boeglinw/courses/Modern_lab_manual3/Rutherford.html

The Rutherford scattering formula can also be written as the angular distribution of the scattering rate $N(\theta)$:

$$N(\theta) = \frac{N_0 n L Z^2 e^4}{(8\pi\varepsilon_0 KE^2)^2 \sin^4\left(\frac{\theta}{2}\right)} \quad (3)$$

- N_0 is the incident alpha particle rate
- n is the number of atoms per unit volume in the foil
- L is the thickness of the foil
- Z is the atomic number of the scattering material
- KE is the kinetic energy of an alpha particle

3. Modelling Single Atom Scattering

We model an α particle's trajectory with a velocity of v_0 in the x-axis starting at a reasonable position *starting_x* in the negative x-axis, with an impact parameter *b* in the z-axis (at this stage we keep y-axis constant / 0 which does not make a difference to the scattering angle) approaching a single metal atom positioned at the origin. Taking suitably small timesteps *dt*, we update the position and velocity of the alpha particle. The simulation ends when the α particle reaches the simulation boundary, which we set as a large sphere with radius *sd* from the origin. Recording the angle of the final velocity vector, we calculate the scattering angle by comparing it with the incident velocity vector. We will observe differences when we alter the element of the atom, using gold, tin, silver, copper, and aluminium.

We will investigate whether or not it is necessary to use Newton's 3rd law but will come to the conclusion it does not make a noticeable difference as Rutherford observed, and treat particles as point masses and charges.

Simulation parameters

We take v_0 as the average speed of an alpha particle which has a kinetic energy of ≈5 MeV.

Through experimentation, we set dt as $5 \times 10^{-22}\ s$ and sd as $1 \times 10^{-8}\ m$ for single scattering.

As the impact parameter increases, the analytical solution for the scattering angle (see below) decreases rapidly and diminishes to an insignificant value ≈ 0. As Rutherford's equation assumes the alpha particles start from an infinite

distance, the *starting_x* alters the percentage error for large impact parameters / very small scattering angles. We will deem a scattering angle of $< 0.05°$ as insignificant since our bin size will be significantly larger. A scattering angle of $0.05°$, alpha particle with kinetic energy of 5 MeV, and the strongest charged nuclei gold, corresponds to an impact parameter of $\approx 3.1432 \times 10^{-11}$m, meaning we only need to test impact parameters from 0 to $< 10^{-10}$m and confirm the scattering angle tends to zero as impact parameter further increases. Through experimentation, a *starting_x* of $5 \times 10^{-9}m$ satisfies these conditions.

3.1. Computational Methods

The motion of the alpha particle is governed by Newton's equations of motion, which can be written as two coupled first-order differential equations with initial conditions mentioned above:

($\vec{r}$ is the position vector of the α particle)

$$\frac{d\vec{r}}{dt} = v(t), \frac{d\vec{v}}{dt} = \frac{F(\vec{r})}{m}$$

We can solve these coupled ODEs using a variety of algorithms. Several algorithms were experimented with, including the Euler-Cromer method, Euler-Richardson method, Half-step method, and different Runge-Kutta methods using the *Scipy* library.

By rearranging equation (1), we can solve for the analytical solution of the scattering angle given impact parameter b and compare it with the angle our code achieves to find the most accurate algorithm.

$$\theta(b) = 2\tan^{-1}\left(\frac{kQ_1Q_2}{m{v_0}^2 b}\right) \quad (3)$$

To find the scattering angle the code achieves, we use the formula:

$$\theta = \cos^{-1}\left(\frac{\vec{r} \cdot \overrightarrow{v_0}}{|r||v_0|}\right)$$

where $\vec{r}$ is the position vector of the end position of the particle.

The code is run on Python 3.9.1. We make use of the *numba* library to significantly speed up *numpy* computations by compiling the code rather than interpreting every line each time a function is run.

Initial Setup

```
#dependencies
import math
import matplotlib.pyplot as plt
import numpy as np
from scipy import constants
from scipy.integrate import solve_ivp
from functools import partial
import random
#numba significantly speeds up computations by compiling function beforehand
from numba import jit
#stop runtime warnings
import warnings
warnings.filterwarnings('ignore')

#constants
charge_of_alpha = 2 * constants.e
mass_of_alpha = constants.physical_constants["alpha particle mass"][0]
coulomb_constant = 1 / (4 * math.pi * constants.epsilon_0)

#class for access to properties of an element
class Element():
    def __init__(self, symbol, atomic_number, atomic_mass):
        self.symbol = symbol
        self.charge = atomic_number * constants.e
        self.mass = atomic_mass * constants.physical_constants["unified atomic mass unit"][0]

#elements
gold = Element("Au", 79, 196.96657)
tin = Element("Sn", 50, 118.71)
silver = Element("Ag", 47, 107.8682)
copper = Element("Cu", 29, 63.546)
aluminium = Element("Al", 13, 26.981539)
all_elements = [gold, tin, silver, copper, aluminium]


def MeV_to_joules(MeV):
    return MeV * 1e6 * constants.e

def energy_to_velocity(MeV):
    energy_in_joules = MeV_to_joules(MeV)
    return math.sqrt(energy_in_joules * 2 / mass_of_alpha)

#simulation parameters
average_energy_of_alpha = 5.0 #average kinetic energy of alpha particle in MeV
v0 = energy_to_velocity(average_energy_of_alpha)  #starting velocity in x-axis
sd = 1e-8   #simulation domain: radius of measurement sphere from the origin
dt = 5e-22  #timestep
starting_x = 5e-9 #distance from the closest atom in x-axis to start simulation
atom_positions = np.array([[0.0,0.0,0.0]]) #single atom at origin

@jit(nopython=True, fastmath=True)
def coulomb_force(alpha_position, atom_positions, element_charge):
    #params: position vector [x,y,z] of alpha particle, 2D array of the positions of the scattering atoms, charge of scattering atom
    #returns: 2D array of forces [[Fx, Fy, Fz], ...] on the alpha particle from each atom

    #vectors from atom positions to alpha particle position
```

```
    r_to_alpha = alpha_position - atom_positions
    #|r| ^ 2
    sum_distances_squared = (r_to_alpha ** 2).sum(axis = 1)
    #1 / (|r| ^ 3)
    inv_r3 = sum_distances_squared ** (-1.5)
    #coulomb force calculation: each row in r_to_alpha matrix multiplied by corresponding scalar in inv_r3
    force = (r_to_alpha.T * inv_r3).T * (coulomb_constant * element_charge * charge_of_alpha)
    return force

@jit(nopython=True, fastmath=True)
def scattering_angle(initial_v, final_v):
    #returns the angle (in radians) between two vectors
    costheta = np.dot(final_v, initial_v)  / (np.linalg.norm(final_v) * np.linalg.norm(initial_v))
    theta = math.acos(costheta)
    return theta

def expected_scattering_angle(b, v0, element):
    #returns analytical solution to scattering angle (in radians) given impact parameter
    if b == 0.0: return math.pi #avoid division by zero
    return 2 * math.atan((coulomb_constant * charge_of_alpha * element.charge) / (mass_of_alpha * (v0 ** 2) * b))

def percentage_error(expected_value, actual_value):
    return ((actual_value - expected_value) / expected_value) * 100
```

Euler-Cromer Method

We use this simple method as a solid basis to check whether the coulomb force function works as expected and whether particle trajectories make sense, as well as to estimate suitable simulation parameters. We will not implement Newton 3rd law for this initial method.

Details for Euler methods can be found here: https://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node2.html

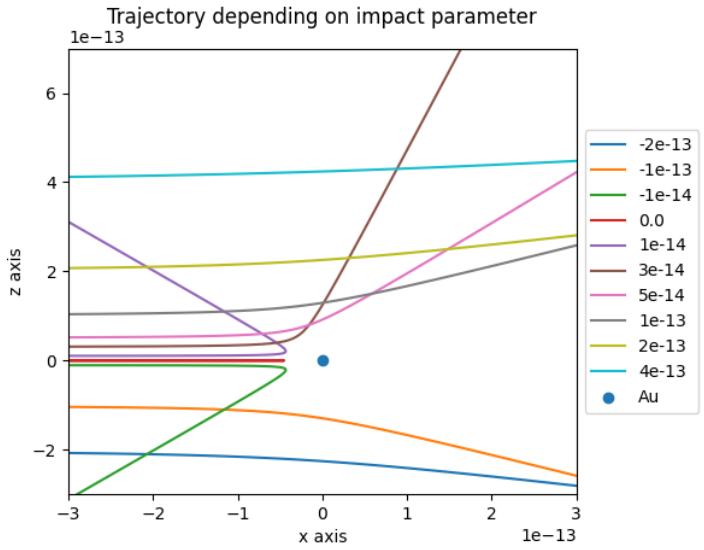
```
@jit(nopython=True, fastmath=True)
def euler_cromer(b, v0, element_charge, atom_positions):
    #returns trajectory and scattering angle given impact parameter (applied in z-axis), charge of scattering material, atom positions using euler-cromer method
    trajectory = []                                 #appends position (x,y,z) at every timestep
    position = np.array([-starting_x, 0.0, b])      #current particle position
    initial_v = np.array([v0, 0.0, 0.0])            #store initial particle velocity for final angle calculation
    velocity = initial_v.copy()                     #current particle velocity
    #until it goes beyond simulation domain
    while np.linalg.norm(position) <= sd:
        trajectory.append(position.copy())
        acc = coulomb_force(position, atom_positions, element_charge).sum(axis = 0) / mass_of_alpha  #find acceleration on a-particle
        velocity += dt * acc        #update velocity
        position += dt * velocity   #update position
    theta = scattering_angle(initial_v, velocity) #scattering angle between final and initial velocity vectors
    return (trajectory, theta)
```

A simple wrapper for plotting the following trajectories using this Euler-Cromer method.

```
def plot_trajectories_wrapper(func):
    #wrapper helper function to avoid repetitive code for plotting trajectories
    def wrap(*args, **kwargs):
        #setting up plot
        ax = plt.subplot(1,1,1)
        plt.xlabel("x axis")
        plt.ylabel("z axis")
        plt.title(kwargs["title"], pad=15)
        ax.set_xlim((-3e-13, 3e-13))
        ax.set_ylim((-3e-13, 7e-13))
        kwargs["ax"] = ax
        #call function to plot
        func(*args, **kwargs)
        #plot legend on the side
        box = ax.get_position()
        ax.set_position([box.x0, box.y0, box.width * 0.85, box.height])
        ax.legend(loc="center left", bbox_to_anchor=(1,0.5))
        plt.show()
    return wrap
```

Dependence on impact parameter

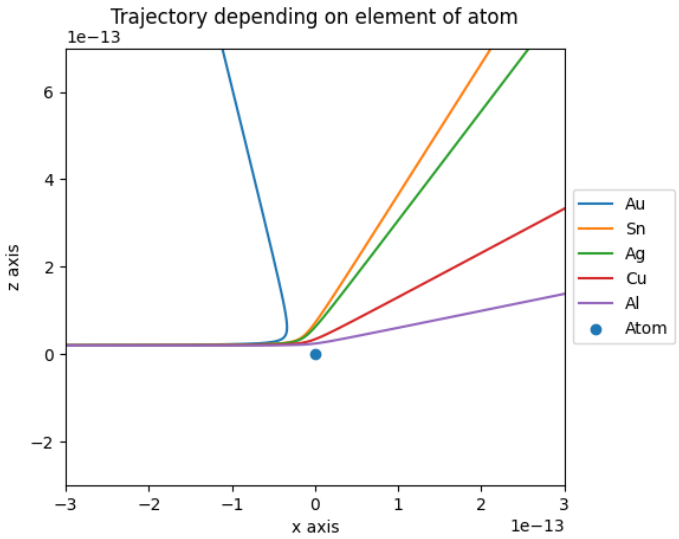
```
@plot_trajectories_wrapper
def dependence_on_impact_params(element, v0, bs, title, ax=None):
    ax.scatter(0,0,label=element.symbol) #plot atom
    for b in bs:
        trajectory = np.array(euler_cromer(b, v0 = v0, element_charge=element.charge,
atom_positions=np.array([[0.0,0.0,0.0]]))[0])
        #plot x and z axis
        ax.plot(trajectory[:,0],trajectory[:,2], label = b)
bs_to_plot = [-2e-13, -1e-13,-1e-14, 0.0, 1e-14, 3e-14, 5e-14, 1e-13, 2e-13, 4e-13]
dependence_on_impact_params(element=gold, v0=v0, bs=bs_to_plot,title="Trajectory depending on impact parameter")
```



The smaller the impact parameter, the larger the deflection.

Dependence on element of scattering atom

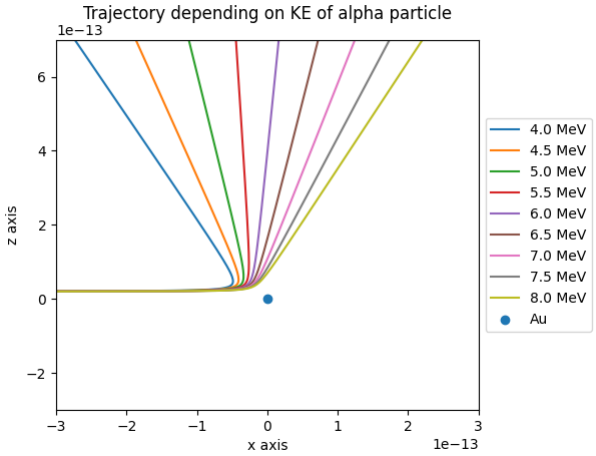
```
@plot_trajectories_wrapper
def dependence_on_element(elements, v0, b, title, ax=None):
    ax.scatter(0,0,label="Atom")
    for element in elements:
        trajectory = np.array(euler_cromer(b, v0 = v0, element_charge=element.charge,
atom_positions=atom_positions)[0])
        #plot x and z axis
        ax.plot(trajectory[:,0],trajectory[:,2], label = element.symbol)
dependence_on_element(elements=all_elements, v0=v0, b=2e-14, title="Trajectory depending on element of atom")
```



The more strongly-charged the nuclei, the greater the deflection.

Dependence on energy of alpha particle

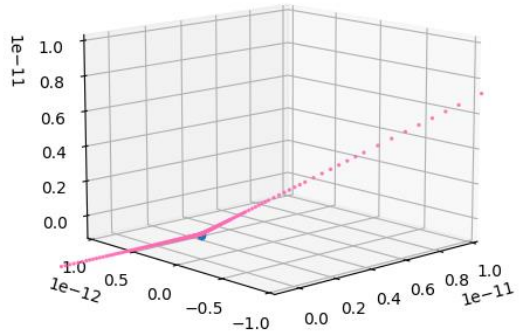
```
@plot_trajectories_wrapper
def dependence_on_ke(element, energies, b, title, ax=None):
    ax.scatter(0,0,label=element.symbol)
    for ke in energies:
        v0 = energy_to_velocity(ke)
        trajectory = np.array(euler_cromer(b, v0 = v0, element_charge=element.charge,
atom_positions=atom_positions)[0])
        #plot x and z axis
        ax.plot(trajectory[:,0],trajectory[:,2], label = f"{ke} MeV")
energies = np.arange(4.0, 8.5, 0.5) #list of energies in MeV to plot from 4.0 - 8.0
dependence_on_ke(element=gold, energies=energies, b=2e-14, title="Trajectory depending on KE of alpha particle")
```



The greater the initial velocity, the lesser the deflection it experiences.

Runge-Kutta Methods

We can improve on the simple Euler-Cromer to achieve greater accuracy and greater efficiency using Runge-Kutta methods implemented using the *Scipy* module, using the *scipy.integrate.solve_ivp* function (documentation here: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html) which solves initial value problems for a system of ODEs. It has an **adaptive time step** (takes shorter steps as it approaches the atom and longer steps when it is far), is well-tested, and fairly efficient in terms of CPU resources and time. The function offers six Runge-Kutta integration methods, including 3 explicit Runge-Kutta methods meant for non-stiff problems, and implicit Radau and BDF, meant for stiff problems, and LSODA as a universal method. It is clear this problem is stiff, meaning numerical methods for the solving the equation are unstable unless the timestep is sufficiently small, which is obvious given Coulomb's inverse-square law where the force can increase/decrease very rapidly as the α particle approaches/moves away from the atom. Therefore, we should only choose from Radau, BDF, and LSODA methods for this problem.



Scattering from a gold nucleus, where the blue marker is the nucleus and pink markers represent the evaluated trajectory of the alpha particle. We can see the adaptive stepping occurring as the timestep gets very small as it approaches the nucleus and gradually gets larger as the force decreases when it moves away.

To solve the ODE using *solve_ivp*, we need to define a function $f(t, S)$ that computes the derivative of the system variables S (the right-hand-side) at time t, which looks like this if we do not include Newton's 3rd law:

$$\vec{S} = \begin{bmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{bmatrix} \quad \Rightarrow \quad \frac{d\vec{S}}{dt} = \vec{f}(t, \vec{S}) = \begin{bmatrix} v_x \\ v_y \\ v_z \\ \frac{F_x(x,y,z)}{m} \\ \frac{F_y(x,y,z)}{m} \\ \frac{F_z(x,y,z)}{m} \end{bmatrix}$$

We will also implement a RHS function that uses Newton's 3rd law and applies a force on the atom which is simply the negative of the force vector on the alpha particle to compare the differences. We will get to choose which RHS function to use when calling the function.

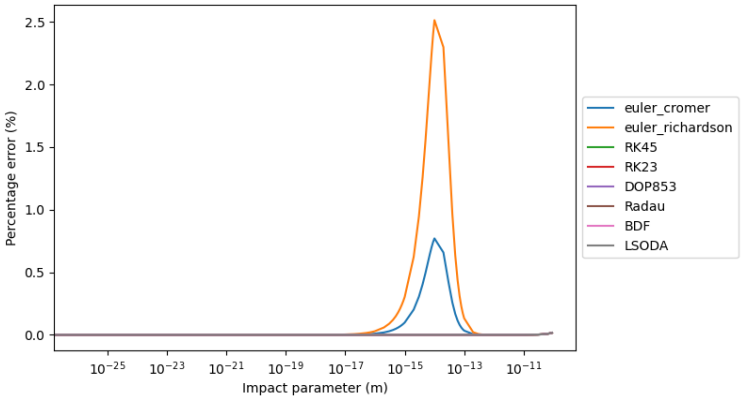
We estimate the maximum time interval as simply the expected time taken to travel $2 \times sd$, and set an event that terminates the solving when the alpha particle crosses the simulation domain, and set the initial time step same as dt. Instead of setting a maximum time step which loses the benefits of having an adaptive time step and effective computation, we need to carefully control the *rtol* (relative tolerance) and *atol* (absolute tolerance) parameters for the function which we first set as 10^{-7} respectively for single scattering.

```
@jit(nopython=True, fastmath=True, cache=True)
def reached_boundary(t, S):
    #terminates solve_ivp when alpha particle reaches simulation boundary which causes a sign-change in the function
    position, _ = np.array_split(S, 2)
    return sd - np.linalg.norm(position)
reached_boundary.terminal = True
```

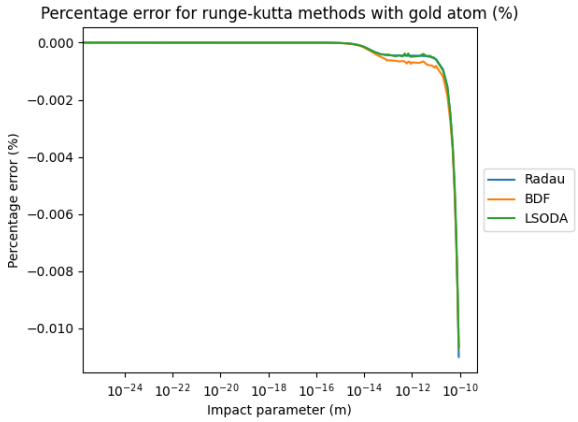
```
7.
8.  @jit(nopython=True, fastmath=True, cache=True)
9.  def rhs(t, S, element_charge, atom_positions):
10.     alpha_position, velocity = np.array_split(S, 2)
11.     force = coulomb_force(alpha_position, atom_positions, element_charge)
12.     acceleration = force.sum(axis = 0) / mass_of_alpha
13.     return np.hstack((velocity, acceleration))
14.
15. @jit(nopython=True, fastmath=True, cache=True)
16. def rhs_3rd_law(t, S, element_charge, element_mass):
17.     #uses newton 3rd law
18.     alpha_position, alpha_velocity, atoms = np.split(S, [3,6]) #splits array into 3 chunks, the last one
    representing information about the atoms
19.     atom_positions, atom_velocities=  np.split(np.reshape(atoms.copy(), (-1,3)), 2) #reshape atoms information
    into 2D and split into 2 chunks
20.     force = coulomb_force(alpha_position, atom_positions, element_charge)
21.     alpha_acceleration = force.sum(axis = 0) / mass_of_alpha #sum of all forces on a-particle
22.     atom_accelerations = -(force) / element_mass
23.     return np.hstack((alpha_velocity, alpha_acceleration, atom_velocities.ravel(), atom_accelerations.ravel()))
24.
25.
26. def runge_kutta(initial_pos, v0, element, atom_positions, method, third_law = False, rtol=1e-7, atol=1e-7):
27.     initial_v = np.array([v0, 0.0, 0.0])
28.
29.     #cut-off time if it does not reach simulation boundary
30.     time_interval = sd * 2 / v0
31.
32.     if third_law:
33.         initial_atom_positions = atom_positions.ravel()
34.         initial_atom_velocities = np.zeros(3 * len(atom_positions))
35.         initial_S = np.hstack((initial_pos, initial_v, initial_atom_positions, initial_atom_velocities))
36.         # print(initial_S)
37.         sol = solve_ivp(fun = lambda t,S: rhs_3rd_law(t, S, element.charge, element.mass), t_span = (0,
    time_interval), y0 = initial_S, events = reached_boundary, method = method, first_step = dt, rtol= rtol,
    atol=atol)
38.     else:
39.         initial_S = np.hstack((initial_pos, initial_v))
40.         sol = solve_ivp(fun = lambda t,S: rhs(t, S, element.charge, atom_positions), t_span = (0,
    time_interval), y0 = initial_S, events = reached_boundary, method = method, first_step = dt, rtol= rtol,
    atol=atol)
41.
42.     if not sol.success:
43.         raise Exception("ODE solver failed to integrate")
44.     #number of time steps taken
45.     num_points = len(sol.t)
46.     #values of solution at each time step
47.     values = sol.y
48.     #retrieve final velocity
49.     final_v = values[:,num_points-1][3:6]
50.     theta = scattering_angle(initial_v, final_v)
51.     return theta
```

Selecting the best algorithm

We can plot the percentage error / deviation of each method against the impact parameters from 0 to $10^{-10}m$ by using the expected value given from equation (3): $\theta(b) = 2\tan^{-1}\left(\frac{kQ_1Q_2}{mv_0{}^2b}\right)$.



It is clear that Runge-Kutta methods implemented using Scipy ODE solver are superior to the modified Euler methods. Not only are they more accurate, the adaptive step size means runtime is significantly reduced. A closer look at the stiff solvers shows they all perform very well even with the strongly-charged gold atom.



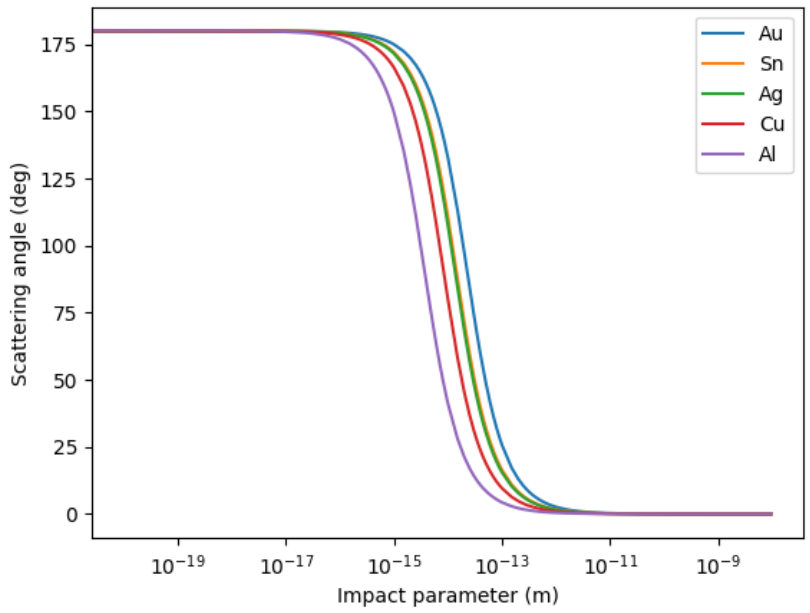
LSODA performs comparatively faster than the other two so we will use this method.

Relationship between impact parameter and scattering angle

We plot the relationship between impact parameter and scattering angle for all the elements.

```
1.  plt.xlabel("Impact parameter (m)")
2.  plt.ylabel("Scattering angle (deg)")
3.  bs = [0.0] + [coeff * 10 ** p for p in range(-20,-8) for coeff in np.arange(1,10,0.5)]
4.  third_law_method = partial(runge_kutta, method = "LSODA", v0 = v0, atom_positions = atom_positions,
    third_law=True)
5.  for element in all_elements:
```

```
6.      thetas = np.degrees([third_law_method(initial_pos=np.array([-starting_x, 0.0, b]), element=element) for b in
    bs])
7.      plt.plot(bs, thetas, label=element.symbol)
8.  plt.legend()
9.  plt.xscale("log")
10. plt.show()
```

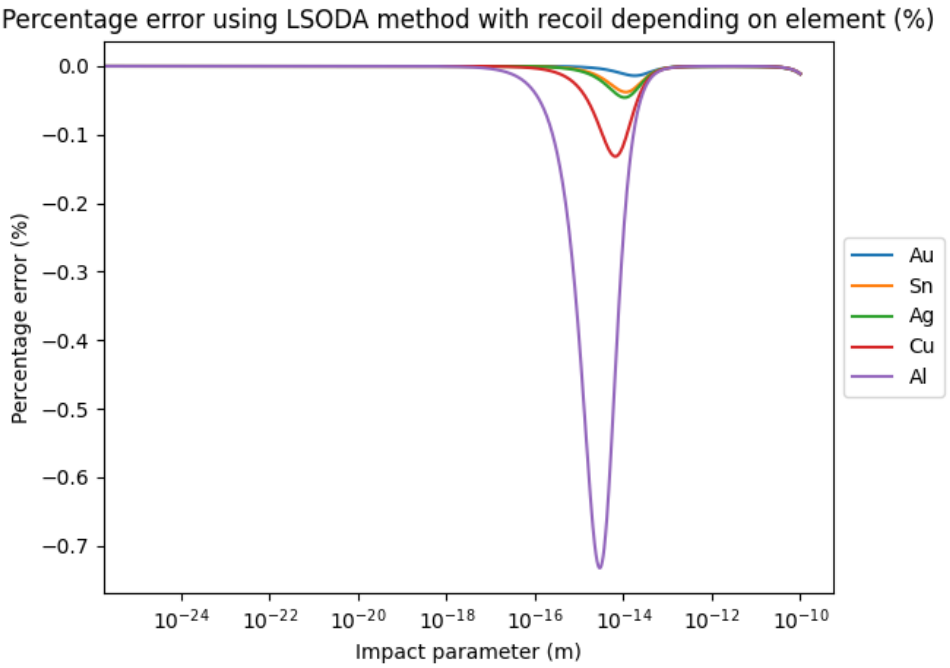


3.2 To Newton's 3rd Law or to not?

We will compare applying Newton's 3rd Law to our ODE (the atom recoils) vs using Rutherford's analytical solution and see how big of a difference the results are.

```
1.  ax = plt.subplot(1,1,1)
2.  plt.xlabel("Impact parameter (m)")
3.  plt.ylabel("Percentage error (%)")
4.  plt.title("Percentage error using LSODA method with recoil depending on element (%)")
5.  ax.set_xscale("log")
6.  bs = [0.0] + [coeff * 10 ** p for p in range(-25,-10) for coeff in range(1,10)]
7.  method = partial(runge_kutta, method = "LSODA", v0 = v0, atom_positions = atom_positions,
    third_law=True)
8.  for element in all_elements:
9.      p_errors = []
10.     for b in bs:
11.         starting_position = np.array([-starting_x, 0.0, b])
12.         correct = math.degrees(expected_scattering_angle(b, v0, element))
13.         theta = math.degrees(method(starting_position, element=element))
14.         p_error = percentage_error(correct, theta)
15.         p_errors.append(p_error)
16.     ax.plot(bs, p_errors,label=element.symbol)
```

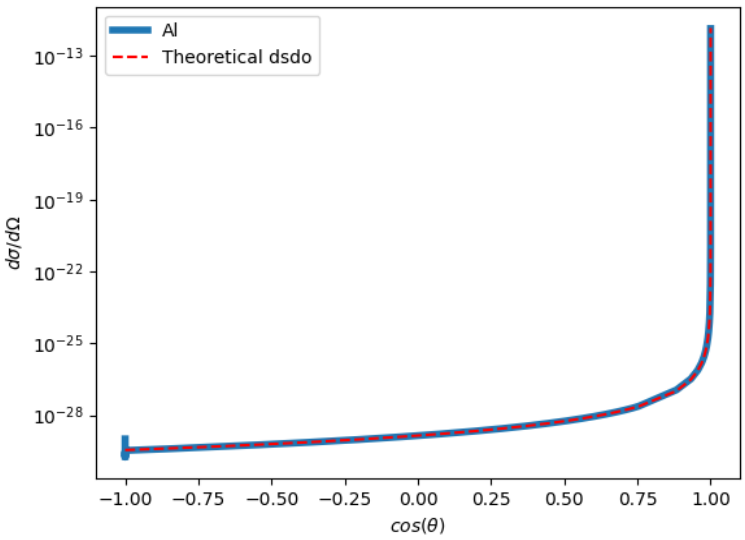
```
box = ax.get_position()
ax.set_position([box.x0, box.y0, box.width * 0.85, box.height])
ax.legend(loc="center left", bbox_to_anchor=(1,0.5))
plt.show()
```



Understandably being the lightest element, the aluminium atom experiences the most recoil, consequently its scattering angle to impact parameter relationship deviates most from Rutherford's formula. That being said, the percentage error sits comfortably under 0.8%, which is well within acceptable percentage deviation and nearly unnoticeable to not applying Newton's 3rd law so Rutherford's assumption was valid for single scattering. Interestingly, the scattering angle at which there is maximum deviation from the Rutherford's formula is similar for all elements when the scattering angle is between 103° to 104°.

3.3 Differential Cross Section

The plot of the differential cross section against the cosine of the scattering angle serves as a signature for scattering off a point target in which no structure is evident. Here is a plot for an aluminium atom if we compare our numerical results from applying LSODA with Newton's 3rd Law against Rutherford's formula:

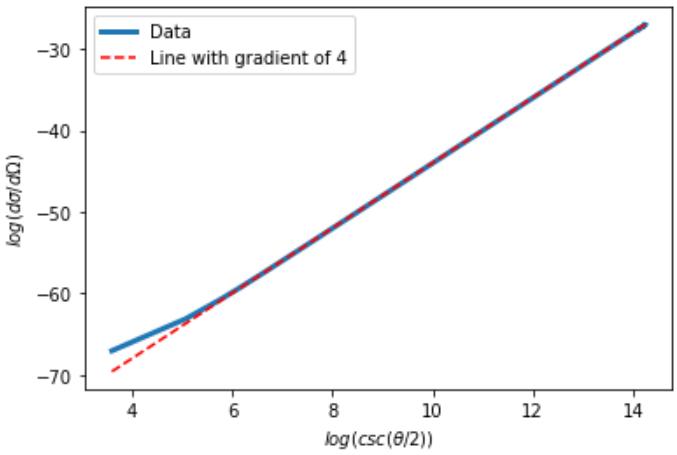


These results are clearly in good agreement with each other, another justification for why applying Newton's 3rd law is not that important.

```
1.  bs = [coeff * 10 ** p for p in range(-25,-10) for coeff in np.arange(1,10,0.5)]
2.
3.  #rutherford formula
4.  calculate_theoretical_dsdo = lambda thetas, element, v0:((charge_of_alpha * element.charge) / (8 * math.pi *
    constants.epsilon_0 * mass_of_alpha * v0 ** 2 * np.sin(thetas/2.0) ** 2)) ** 2
5.  #expression for scattering cross-section
6.  calculate_dsdo = lambda bs, thetas: (bs / np.sin(thetas)) * np.abs(np.gradient(bs, thetas))
7.
8.  thetas = np.array([runge_kutta(initial_pos=np.array([-starting_x, 0.0,
    b]),v0=v0,element=aluminium,third_law=True,atom_positions=atom_positions, method = "LSODA") for b in bs])
9.  costhetas = np.cos(thetas)
10. #calculate the differential cross section for both experimental and theoretical
11. dsdo = calculate_dsdo(bs, thetas)
12. theoretical_dsdo = calculate_theoretical_dsdo(thetas, aluminium, v0)
13. #plot against costheta
14. plt.plot(costhetas, dsdo,label=aluminium.symbol, linewidth=4.0)
15. plt.plot(costhetas, theoretical_dsdo, "r--", label="Theoretical dsdo")
16.
17. plt.xlabel(r"$cos(\theta)$")
18. plt.ylabel(r"$d\sigma/d\Omega$")
19. plt.yscale("log")
20. plt.legend()
21. plt.show()
```

The differential cross section is proportional to $csc^4\left(\frac{\theta}{2}\right)$. We will validify this relationship holds even when applying Newton's 3rd law to aluminium by plotting $\log\left(\frac{d\sigma}{d\Omega}\right)$ against $\log\left(csc\left(\frac{\theta}{2}\right)\right)$ which should result in a straight line with a gradient of 4.

```
#fitting functions
def straight_line_fit(thetas, m, c):
    return m * thetas + c

def cosec_half_4_fit(thetas, a):
    #csc(theta/2)**4
    return a / (np.sin(np.radians(thetas) / 2)**4)

def cosec_half_1_fit(thetas, a):
    #csc(theta/2)
    return a / (np.sin(np.radians(thetas) / 2))

def log_dsdo_log_cosec(bs, thetas, algorithm=""):
    #check whether differential cross section is really proportional to csc(theta/2)^4
    dsdo = calculate_dsdo(bs, thetas)
    log_dsdo = np.log(dsdo)
    log_cosec = np.log(cosec_half_1_fit(thetas, 1))
    m, c = curve_fit(straight_line_fit, log_cosec, log_dsdo)[0]
    print(f"Slope using {algorithm} is {m}, deviation is {abs(percentage_error(4.0, m))}%")
    plt.plot(log_cosec, log_dsdo, label="Data",linewidth=3)
    plt.plot(log_cosec, straight_line_fit(log_cosec, 4, c), "r--", label="Line with gradient of 4")
    plt.ylabel(r"$log(d\sigma/d\Omega)$")
    plt.xlabel(r"$log(csc(\theta/2))$")

testing_element = aluminium
bs = np.linspace(1e-20, 1e-10, 10000)
thetas = np.array([runge_kutta(initial_pos=np.array([-starting_x, 0.0,
b]),v0=v0,element=testing_element,third_law=True,atom_positions=atom_positions, method = "LSODA") for b in bs])
log_dsdo_log_cosec(bs, thetas, "LSODA with Newton 3rd law")
```

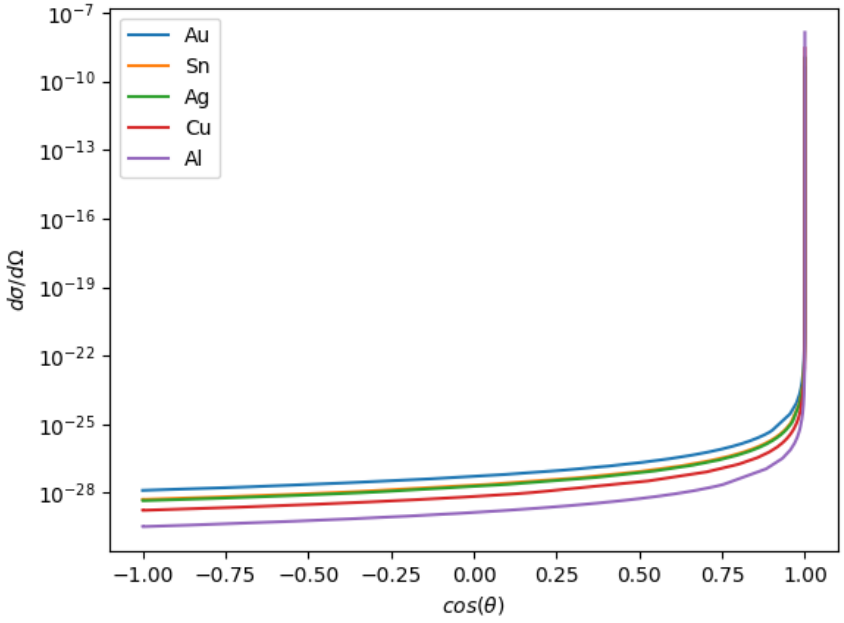


The relationship holds true. Note the very slight curve also occurs even if using the exact Rutherford's formula for $\theta(b)$. Sampling many more points should eventually straighten out the curve.

Variation of differential cross section with different target nucleus

We plot the differential cross section for different target nuclei.

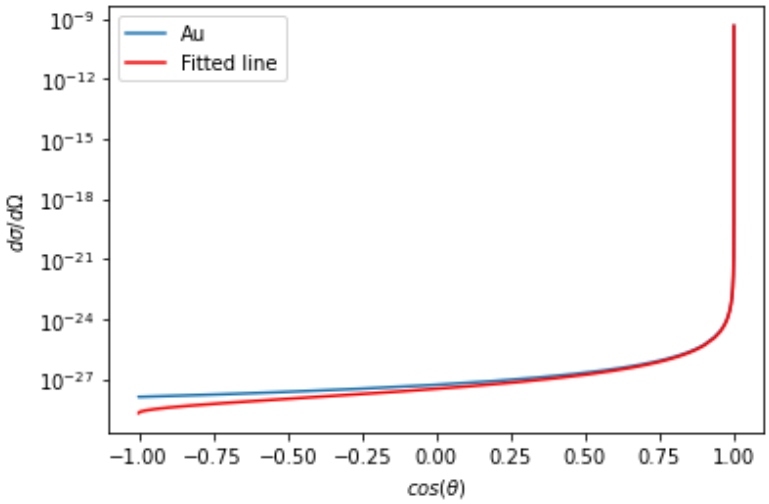
```
1.  bs = [coeff * 10 ** p for p in range(-20,-9) for coeff in np.arange(1,10,0.5)]
2.  for element in all_elements:
3.      thetas = np.array([third_law_method(initial_pos=np.array([-starting_x, 0.0, b]),element=element) for b in
    bs])
4.      costhetas = np.cos(thetas)
5.      #calculate the differential cross section
6.      dsdo = (bs / np.sin(thetas)) * np.abs(np.gradient(bs, thetas))
7.      #plot against costheta
8.      plt.plot(costhetas, dsdo,label=element.symbol)
9.
10. plt.xlabel(r"$cos(\theta)$")
11. plt.ylabel(r"$d\sigma/d\Omega$")
12. plt.yscale("log")
13. plt.legend()
14. plt.show()
```



As expected, the probability of deflection at large angles is higher for more strongly-charged nuclei.

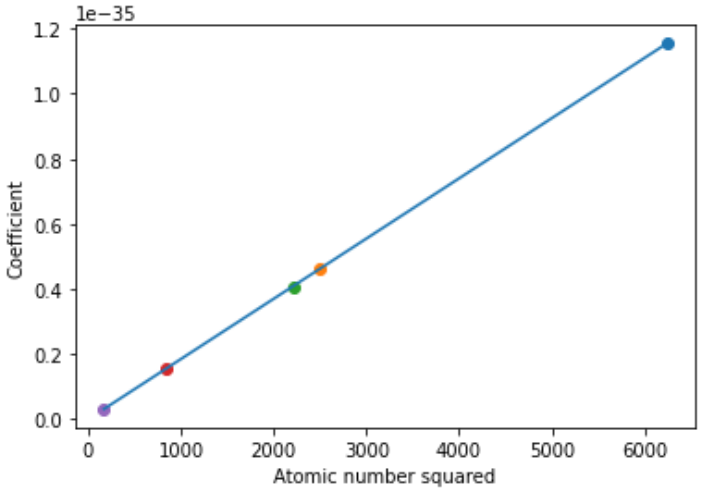
We can verify that the differential cross section is indeed proportional to Q^2. We do this by plotting the square of the atomic number of the target nuclei against the coefficient k from fitting the data points to $k \times csc^4\left(\frac{\theta}{2}\right)$, which should result in a straight line.

```
bs = [coeff * 10 ** p for p in range(-20,-9) for coeff in np.arange(1,10,0.25)]

coeffs = [] #array of k where line k * csc(theta/2)^4 fits data for each element

for element in all_elements:
    thetas = np.array([third_law_method(initial_pos=np.array([-starting_x, 0.0, b]),element=element) for b in bs])
    costhetas = np.cos(thetas)
    dsdo = calculate_dsdo(bs, thetas)
    plt.plot(costhetas, dsdo, label=element.symbol)
    #fit data
    k = curve_fit(cosec_half_4_fit, thetas, dsdo)[0][0]
    coeffs.append(k)
    plt.plot(costhetas, cosec_half_4_fit(thetas, k), color="red", label="Fitted line")
    plt.yscale("log")
    plt.xlabel(r"$cos(\theta)$")
    plt.ylabel(r"$d\sigma/d\Omega$")
    plt.legend()
    plt.show()
```



A plot of the measured differential cross section for gold versus its fitted line of $csc^4\left(\frac{\theta}{2}\right)$.

```
xs = []
for element, k in zip(all_elements, coeffs):
    xs.append(element.atomic_number ** 2)
    plt.scatter(element.atomic_number ** 2, k)
popt = curve_fit(straight_line_fit,xs, coeffs)[0]
plt.plot(xs, straight_line_fit(np.array(xs), *popt))
plt.ylabel("Coefficient")
plt.xlabel("Atomic number squared")
plt.figsize()
plt.show()
```



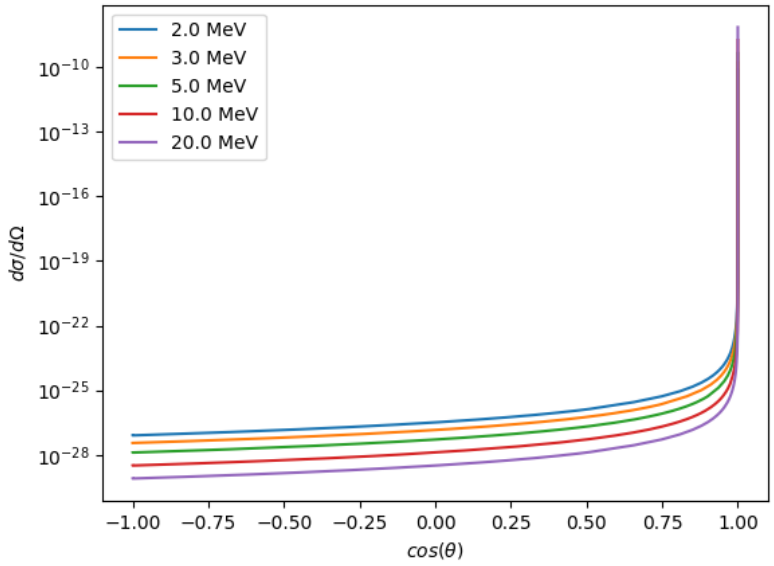
This shows that the relationship is valid.

Variation of differential cross section with kinetic energy of alpha particle

We plot the differential cross section for different energies for a single gold nucleus.

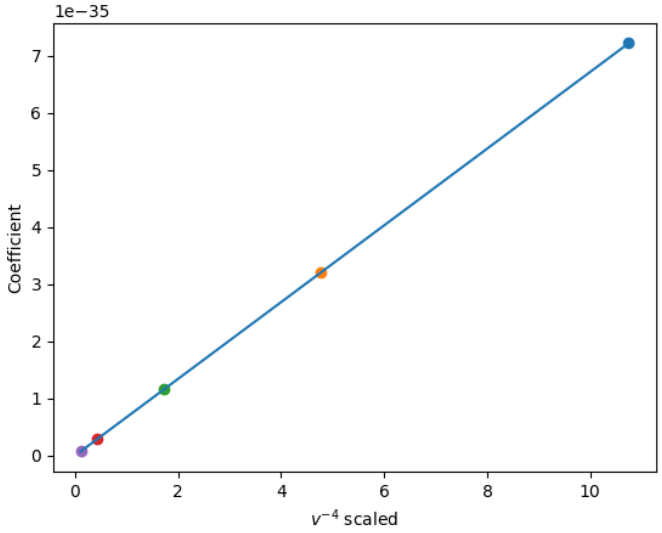
We can similarly verify that the differential cross section is indeed proportional to $\frac{1}{v^4}$. Again we do this by plotting $\frac{1}{v^4}$. against the coefficient k from fitting the data points to $k \times csc^4\left(\frac{\theta}{2}\right)$, which should result in a straight line.

```
1.  bs = [coeff * 10 ** p for p in range(-20,-9) for coeff in np.arange(1,10,0.25)]
2.  coeffs = [] #array of k where line k * csc(theta/2)^4 fits data for each element
3.  energies = [2.0, 3.0, 5.0, 10.0, 20.0] #in MeV
4.  velocities = list(map(energy_to_velocity, energies))
5.  testing_element = gold
6.  for energy, velocity in zip(energies,velocities):
7.      thetas = np.array([runge_kutta(initial_pos=np.array([-starting_x, 0.0,
    b]),v0=velocity,element=testing_element,third_law=True,atom_positions=atom_positions, method = "LSODA") for b in
    bs])
8.      costhetas = np.cos(thetas)
9.      dsdo = calculate_dsdo(bs, thetas)
10.     plt.plot(costhetas, dsdo, label=f"{energy} MeV")
11.     #fit data
12.     k = curve_fit(cosec_half_4_fit, thetas, dsdo)[0][0]
13.     coeffs.append(k)
14. plt.yscale("log")
15. plt.xlabel(r"$cos(\theta)$")
16. plt.ylabel(r"$d\sigma/d\Omega$")
17. plt.legend()
18. plt.show()
```



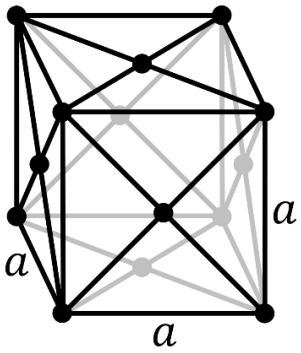
As expected, a higher kinetic energy results in a lower probability of deflection.

```
1.  xs = [] #stores scaled 1/v^4
2.  for velocity, k in zip(velocities, coeffs):
3.      #x axis needs to be scaled or else value is too small for fitting
4.      scaled_inv_velocity_4 = 1 / (velocity ** 4) * 1e29
5.      xs.append(scaled_inv_velocity_4)
6.      plt.scatter(scaled_inv_velocity_4, k)
7.  popt = curve_fit(straight_line_fit,xs, coeffs)[0]
8.  plt.plot(xs, straight_line_fit(np.array(xs), *popt))
9.  plt.ylabel("Coefficient")
10. plt.xlabel(r"$v ^{-4} $ scaled")
11. plt.show()
```



We also verify this relationship holds true.

4. Modelling Scattering from a Crystal Lattice

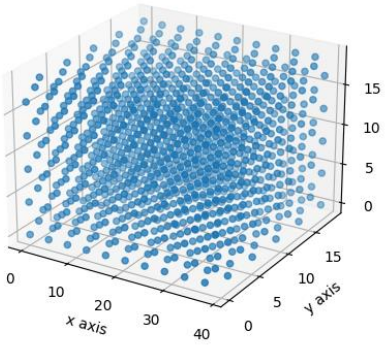


We will investigate whether the Rutherford scattering formula holds true for crystal lattices. The metals tested above (excluding Tin) have a face-centred cubic structure (FCC), which looks like this where there is an atom at the centre of all 6 faces. We assume the entire lattice is built out of this structure and is a perfect lattice.

We use the *Atomic Simulation Environment (ASE)* library for generating large simple FCC crystal lattices. We only need to supply the element and the number of atoms in the x, y, z directions. It will automatically retrieve the experimental lattice parameter of that element for us. This creates a lattice object and we can simply get the positions of all the atoms in a 2D array.

We will set a cut-off distance for the coulomb force and only require the simulation to find the force on the alpha particle within a certain radius. We use a K-d Tree provided by the *scipy* library (scipy.spatial.KDTree — SciPy v1.8.0 Manual) which offers logarithmic lookup time for the k-nearest neighbours and nearest neighbours within a given radius given the lattice structure is massive. A radius of $10^{-9}m$ seems reasonable as using equation (3) achieves a miniscule scattering angle even with gold nuclei.

During the period before the alpha particle enters the lattice and the period after leaves the lattice, there are no atoms within the radius, therefore we will resort to using the force exerted from an arbitrary number of closest atoms so that *solve_ivp* will take smaller timesteps as force increases, which we have set as 4, the same as the number of atoms in an FCC unit cell. Generating a full metal foil is clearly computationally difficult—just a single gram of gold contains more than 10^{21} atoms. Given the thickness is an important factor, we can keep the thickness of the foil the same, but make the cross section relatively small. This assumption should likely only affect cases of large angle scattering which are very rare. We set the cross section to be 30 by 30 in number of atoms. To simulate the beam of alpha particles, we generate a random y and z position on a sufficiently small circle (uniformly) so that it is definitely confined within the cross section of the lattice, which we set as a radius of $10^{-9}m$. All particles start with the same x-position which is $-(x_{max} + starting\_x)$ where x_{max} is the maximum x coordinate of the lattice. To the right is a picture of a 10-atoms thick (x-axis), 5 by 5 atoms wide (y and z axes) gold FCC structure. (Coordinates in angstroms)



Runge-Kutta Method for Lattice Scattering

```
@jit(nopython=True)
def split_array(S):
    #Faster jitted splitting of array into alpha particle position and velocity components
    return np.array_split(S,2)

@jit(nopython=True, fastmath=True)
def rhs_fast_helper(alpha_position, alpha_velocity, element_charge, closest_positions):
    #Faster jitted helper function which returns the RHS
    force = coulomb_force(alpha_position, closest_positions, element_charge)
    acceleration = force.sum(axis = 0) / mass_of_alpha
    return np.hstack((alpha_velocity, acceleration))

def rhs_closest_positions(t, S, element_charge, atom_positions, tree):
    #rhs function that calculates force by finding nearest atoms (within a radius)
    #unable to jit as KDTree is incompatible with Numba so uses helper functions to maintain good speed
    force_radius = 1e-10
    num_closest = 4
    alpha_position, alpha_velocity = split_array(S)
    #find closest points within radius
    indices_closest_positions = tree.query_ball_point(alpha_position, force_radius)
    #find closest num_closest points if nothing is within radius or else solve_ivp will fail
    if len(indices_closest_positions) == 0:
        indices_closest_positions = tree.query(alpha_position, num_closest)[1]
    #returns the derivatives
    return rhs_fast_helper(alpha_position, alpha_velocity, element_charge,
atom_positions[indices_closest_positions])

def runge_kutta_lattice(initial_pos, v0, element, atom_positions, method, tree, rtol=1e-4, atol=1e-6):
    #returns scattering angle by simulating a-particle trajectory through a lattice
    #uses KD tree to find closest points within radius of particle
    initial_v = np.array([v0, 0.0, 0.0])
    #cut-off time if it does not reach simulation boundary
    time_interval = sd * 2 / v0
    initial_S = np.hstack((initial_pos, initial_v))
    sol = solve_ivp(fun = lambda t,S: rhs_closest_positions(t, S, element.charge, atom_positions, tree), t_span
= (0, time_interval), y0 = initial_S, events = reached_boundary, method = method, first_step = dt, rtol= rtol,
atol=atol)
    if not sol.success:
        raise Exception("ODE solver failed to integrate")
    #number of time steps taken
    num_points = len(sol.t)
    #values of solution at each time step
    values = sol.y
    #retrieve final velocity
    final_v = values[:,num_points-1][3:6]
    theta = scattering_angle(initial_v, final_v)
    return theta
```

α-particle beam

```
def random_position(radius):
    #initial position of alpha particle within a uniform circle of specified radius
    radius = radius * math.sqrt(random.random())
    theta = 2 * math.pi * random.random()
    z = radius * math.cos(theta)
```

```
        y = radius * math.sin(theta)
        coordinates = np.array([-(x_max + starting_x), y, z])
        return coordinates
```

Generating Lattice + Lattice Scattering Simulation

Note that the multiprocessing is used, which does not work in Jupyter Notebook and so was run in a separate Python file. We will simulate firing a narrow stream of α particles at a lattice, recording the scattering angle for each α particle in degrees. Then we will save this data using a text file.

```
if __name__ == "__main__":
    element = gold
    atoms_thick = 10
    cross_section_atoms = 30 #number of atoms across y and z direction
    beam_radius = 1e-9
    num_alpha = int(1e5)
    NUM_PROCESSES = 8  #number of process to happen concurrently

    #Generate FCC lattice object
    atoms = FaceCenteredCubic(element.symbol, size=(atoms_thick, cross_section_atoms, cross_section_atoms))
    #2D array of atom positions
    atom_positions = atoms.get_positions()
    #to angstrom scale
    atom_positions *= 1e-10
    #centre to origin
    x_max,y_max,z_max = atom_positions.max(axis = 0)
    atom_positions[:,0] -= x_max / 2
    atom_positions[:,1] -= y_max / 2
    atom_positions[:,2] -= z_max / 2
    #reupdate the max atom positions
    x_max,y_max,z_max = atom_positions.max(axis = 0)
    #construct KD tree
    tree = cKDTree(atom_positions)

    list_of_starting_positions = [random_position(beam_radius) for _ in range(num_alpha)]
    #multiprocessing to get scattering angle given initial positions
    with Pool(NUM_PROCESSES) as p:
        thetas = p.map(partial(runge_kutta_lattice, tree=tree,
v0=v0,element=element,atom_positions=atom_positions,method="LSODA",rtol=1e-4,atol=1e-6),
list_of_starting_positions)
    thetas_in_deg = np.degrees(thetas)

    #saved to text file
    with open("scattering_data/____.txt", "a") as file:
        for theta in thetas_in_deg:
            file.write(f"{theta}\n")
```

4.1 Investigation into Rutherford's Formula

Reading the data from the text files, we will convert the lines of the text file into an array of angles, then place each angle in their corresponding bins which is implemented using *np.histogram*. We experiment with bin sizes to best reflect the data, but we will generally choose a bin size of less than 0.5°. The centre of each bin is taken as the angle recorded and the frequency is the number of α particles landing at that angle.

Useful Code for Data Analysis

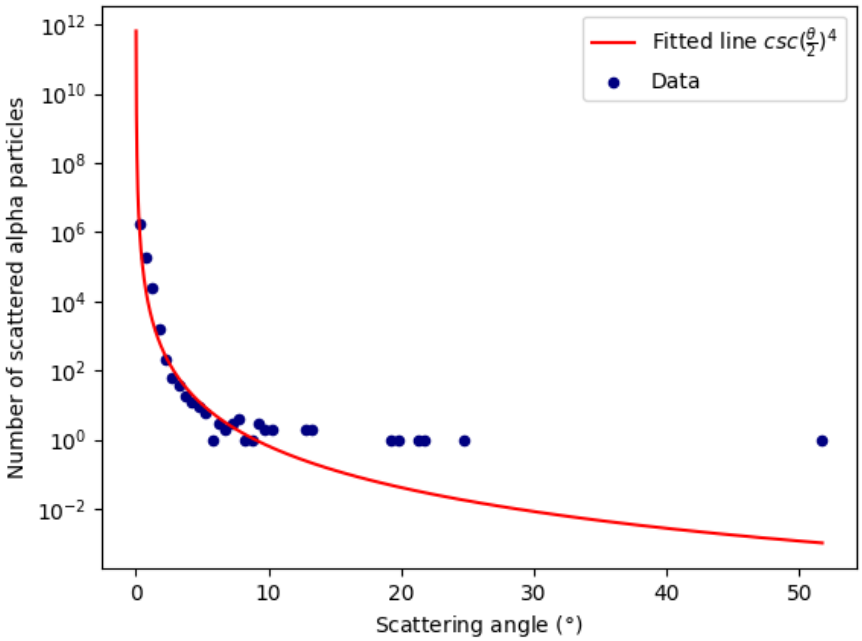
```
def file_to_arr(file, num_data=-1):
    #reads text file and translates into list of length num_data of scattering angles
    #if num_data is -1 choose all of the data points
    arr = []
    file_lines = file.readlines()
    #start from the 5th line
    if num_data == -1: file_lines = file_lines[4:]
    else: file_lines = file_lines[4:num_data+4]
    for line in file_lines:
        arr.append(float(line.strip("\n")))
    if num_data != -1 and num_data != len(arr):
        print("Insufficient data to match specified number")
    return arr

def file_to_counts(file, num_data=-1, binwidth=0.5):
    #returns an array of angles (bin centres) and corresponding array of no. of a particles landing at that angle
    #chooses num_data data points from text file (-1 means all data)
    #chooses bins of size binwidth
    thetas_arr = file_to_arr(file, num_data)
    counts, bins = np.histogram(thetas_arr,bins=np.arange(0.0, max(thetas_arr) + binwidth, binwidth))
    binscenters = np.array([0.5 * (bins[i] + bins[i+1]) for i in range(len(bins)-1)])
    return binscenters, counts


#fitting functions
def straight_line_fit(thetas, m, c):
    return m * thetas + c

def cosec_half_4_fit(thetas, a):
    #csc(theta/2)**4
    return a / (np.sin(np.radians(thetas) / 2)**4)

def cosec_half_1_fit(thetas, a):
    #csc(theta/2)
    return a / (np.sin(np.radians(thetas) / 2))

def plot_line_of_best_fit(xs, ys, ax):
    #plots straight line onto ax and returns the gradient
    xs, ys = np.array(xs), np.array(ys)
    m, c = curve_fit(straight_line_fit, xs, ys)[0]
    ax.plot(xs, straight_line_fit(xs, m, c), color="red", label="Fitted line")
    return m
```

Variation in no. of scattered α-particles with scattering angle

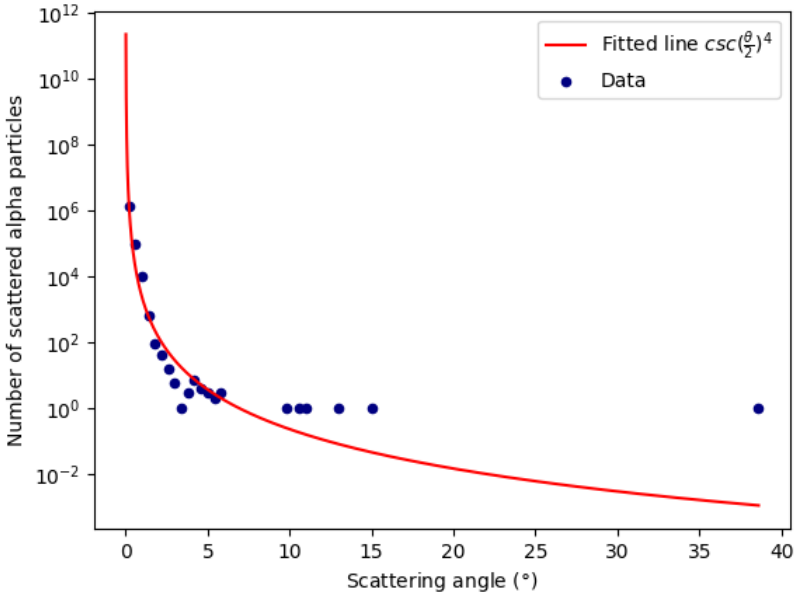
Geiger and Marsden found this should be proportional to $csc^4\left(\frac{\theta}{2}\right)$.

Below is the plot after firing nearly 2×10^6 alpha particles at a gold foil of 10 atoms thick, alongside a fitted line.

Bin size taken as 0.5.



Another plot after firing 1.5×10^6 alpha particles at a silver foil of 10 atoms thick. Bin size taken as 0.4.



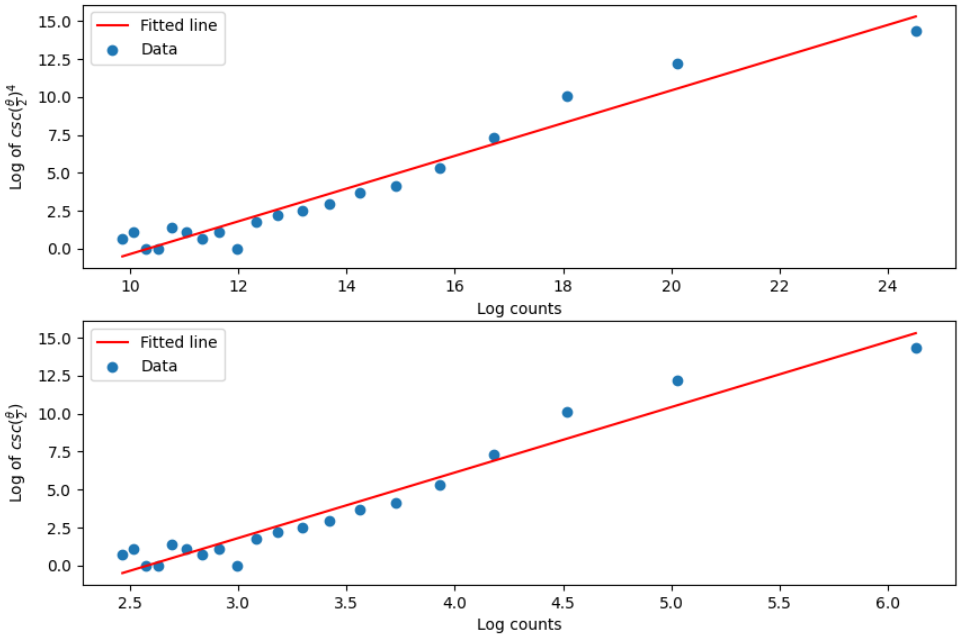
The graphs follow Rutherford's prediction quite well. Continuing to fire more particles should result in a smoother curve and achieve better data for larger angles.

```
def plot_num_scattered_to_angle(file_name, binwidth=0.5):
    with open(file_name) as file:
        #plot actual data as scatter plot
        binscenters, counts  = file_to_counts(file, binwidth=binwidth)
        plt.scatter(binscenters, counts, color="navy", label="Data", s=20)

        #plot fitted line of k * csc(theta/2) ^ 4
        plotting_angles = np.arange(0.0, max(binscenters)+0.01,0.01) #generate enough x-values for plotting best fit
        popt, _ = curve_fit(cosec_half_4_fit, xdata=binscenters, ydata=counts) #get optimal params
        plt.plot(plotting_angles, cosec_half_4_fit(plotting_angles, *popt), color="red", label=r"Fitted line $csc(\frac{\theta}{2})^4$")

        plt.yscale("log")
        plt.legend()
        plt.show()

plot_num_scattered_to_angle('scattering_data/gold_10.txt', binwidth=0.5)
```

Plotting $\log(N)$ against $\log\left(csc\left(\frac{\theta}{2}\right)\right)$ where N is the number of scintillations we should get a straight line with gradient of 4. Similarly, plotting $\log(N)$ against $\log\left(csc^4\left(\frac{\theta}{2}\right)\right)$ we should get a straight line with gradient of 1. We will not use angles above 10° as we have not fired enough particles to achieve good data for larger angles.

```
with open('scattering_data/gold_10.txt') as file:
    binwidth = 0.5
    max_angle = 10.0 #maximum angle for data used
    max_index = int(max_angle / binwidth) #will only use first max_index number of data points
    counter = file_to_counter(file, binwidth=binwidth)
    binscenters = list(counter.keys())
    counts = list(counter.values())

    counts = counts[:max_index]
    binscenters = binscenters[:max_index]
    fig, axes = plt.subplots(2, figsize=(8,8)) #axes[0] for cosec4, axes[1] for cosec1

    log_counts = np.log(counts)
    #replace -inf with 0
    log_counts = np.nan_to_num(log_counts, neginf=0)

    log_cosec_4 = np.log(cosec_half_4_fit(binscenters,1))
    log_cosec_1 = np.log(cosec_half_1_fit(binscenters,1))
    axes[0].scatter(log_cosec_4, log_counts, label="Data")
    axes[1].scatter(log_cosec_1, log_counts, label="Data")

    cosec4_gradient = plot_line_of_best_fit(log_cosec_4, log_counts, axes[0])
    print(f"Cosec4 gradient: {cosec4_gradient}")
    cosec1_gradient = plot_line_of_best_fit(log_cosec_1, log_counts, axes[1])
    print(f"Cosec1 gradient: {cosec1_gradient}")

    axes[0].set_xlabel("Log counts")
```

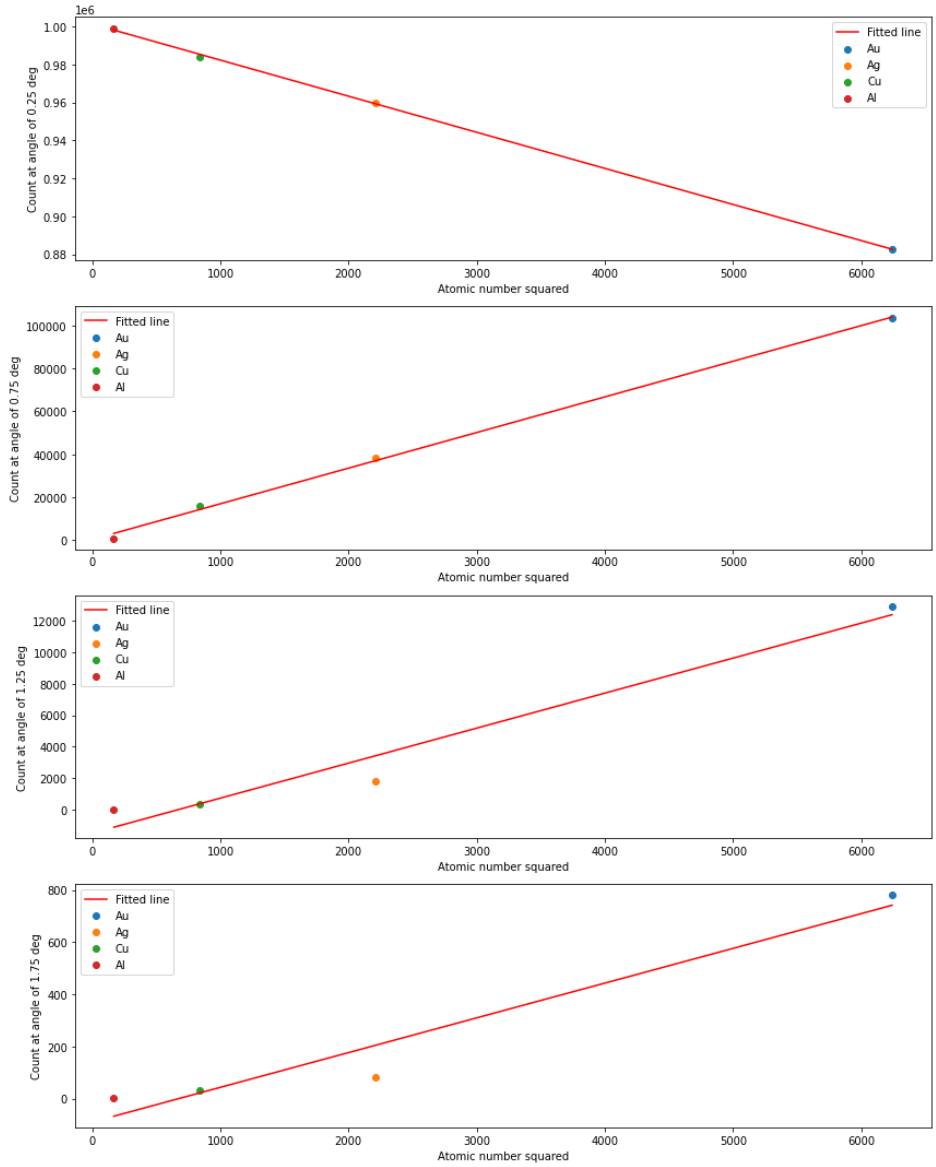
```
28.     axes[1].set_xlabel("Log counts")
29.     axes[0].set_ylabel(r"Log of $csc(\frac{\theta}{2})^4$")
30.     axes[1].set_ylabel(r"Log of $csc(\frac{\theta}{2})$")
31.     axes[0].legend()
32.     axes[1].legend()
33.     plt.show()
```



We get gradients of ≈ 1.08 for $\log\left(csc^4\left(\frac{\theta}{2}\right)\right)$ and ≈ 4.31 for $\log\left(csc\left(\frac{\theta}{2}\right)\right)$. Though not exactly perfect, it does show that this $csc^4\left(\frac{\theta}{2}\right)$ relationship should exist.

Variation in no. of scattered α-particles with the charge of nucleus

Geiger and Marsden found this should be proportional to $Q_n{}^2$. For a specific angle where there is sufficient data, we plot the number of scattered particles at that angle against the square of the atomic number of the target nuclei. We take data from firing 10^6 alpha particles at 10-atom thick lattices of gold, silver, copper, and aluminium, and plot for angles 0.75°, 1.25°, and 1.75°. We also plot the extreme 0.25° which we would expect to be decreasing. Bin size chosen as 0.5.

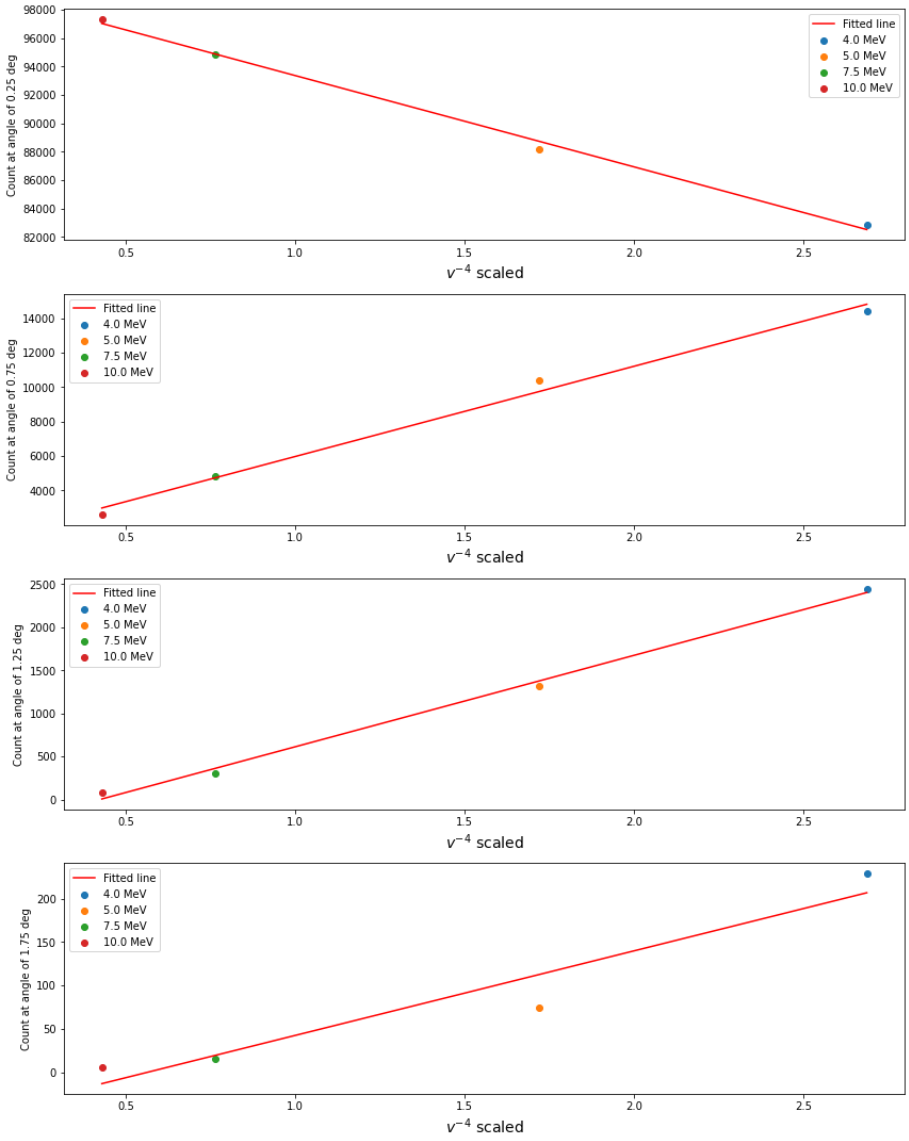


The points follow a reasonably straight line to show this proportionality relationship is followed. Certainly, more particles can be fired to achieve even better results.

Variation in no. of scattered α-particles with the kinetic energy of particles

The formula predicts this should be proportional to KE^2 or $\frac{1}{v^4}$.

We take data from firing 10^5 alpha particles with energies of 4.0, 5.0, 7.5, and 10.0 MeV at a 10-atom thick gold lattice and plot for angles 0.75°, 1.25°, and 1.75°. We also plot the extreme 0.25° which we again would expect to be decreasing.

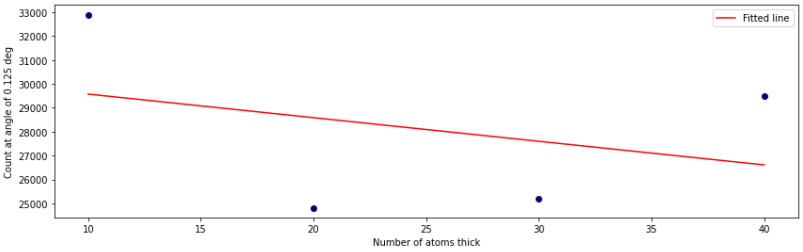


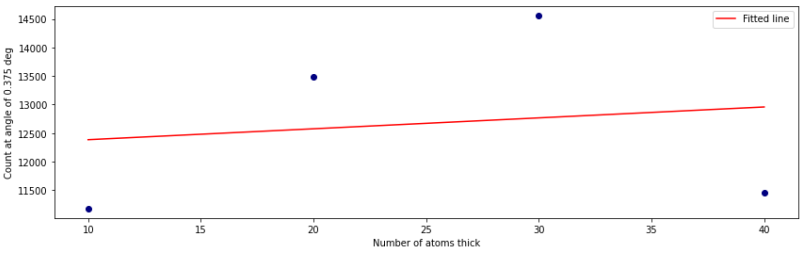
The points also follow a reasonably straight line to show this proportionality relationship is followed. Again, more particles could be fired to obtain better results.

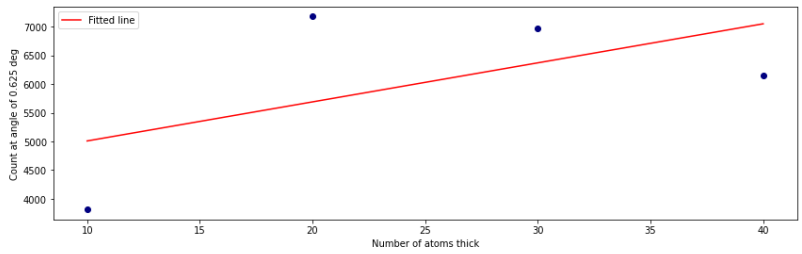
Variation in no. of scattered α-particles with the thickness of the foil

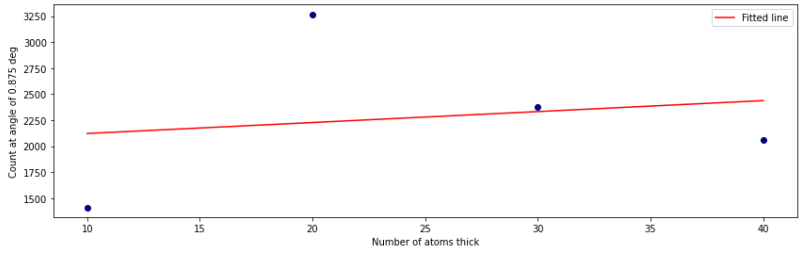
This should be proportional to the thickness itself. For a specific angle where there is sufficient data, we plot the number of scattered particles at that angle against the number of atoms thick the lattice is.

At first glance this relationship seems to be followed for these smaller thicknesses, however with further testing as the thickness increases there does not seem to be an increase in number scattered for a certain angle, even looking like random data. This is due to the setup and possibly a limitation of our simulation: we assume the FCC is a perfect crystal structure, and we positioned it so that it is not rotated in any way, which results in lines of atoms when looking parallel to the x-axis (or any axis). Although this would have had very little effect on the proportionality results obtained above where the number of atoms thick is irrelevant, this setup would certainly not be suitable for testing the relationship with varying thicknesses.

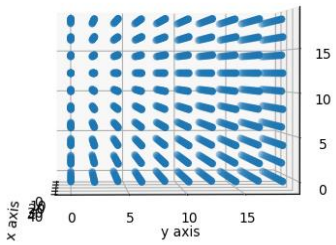






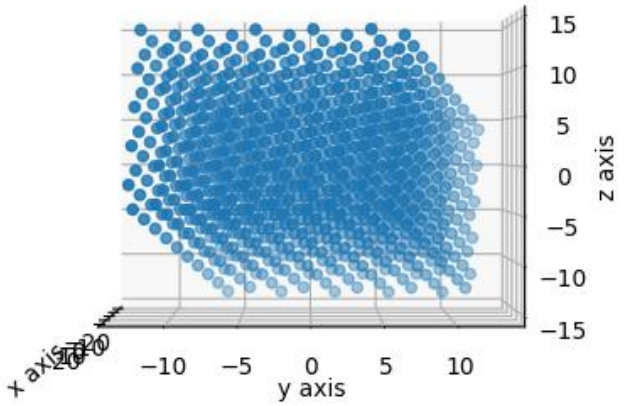


Parallel side view of lattice



Gold lattice scattering results show very unexpected results, leading us to believe the orientation of the lattice plays a significant role. Bin size taken as 0.25.

As a temporary fix, by rotating the lattice via matrix multiplication, we ensure there are not parallel lines of atoms, and so an increase in the thickness should result in increasing the probability of larger angle scattering.



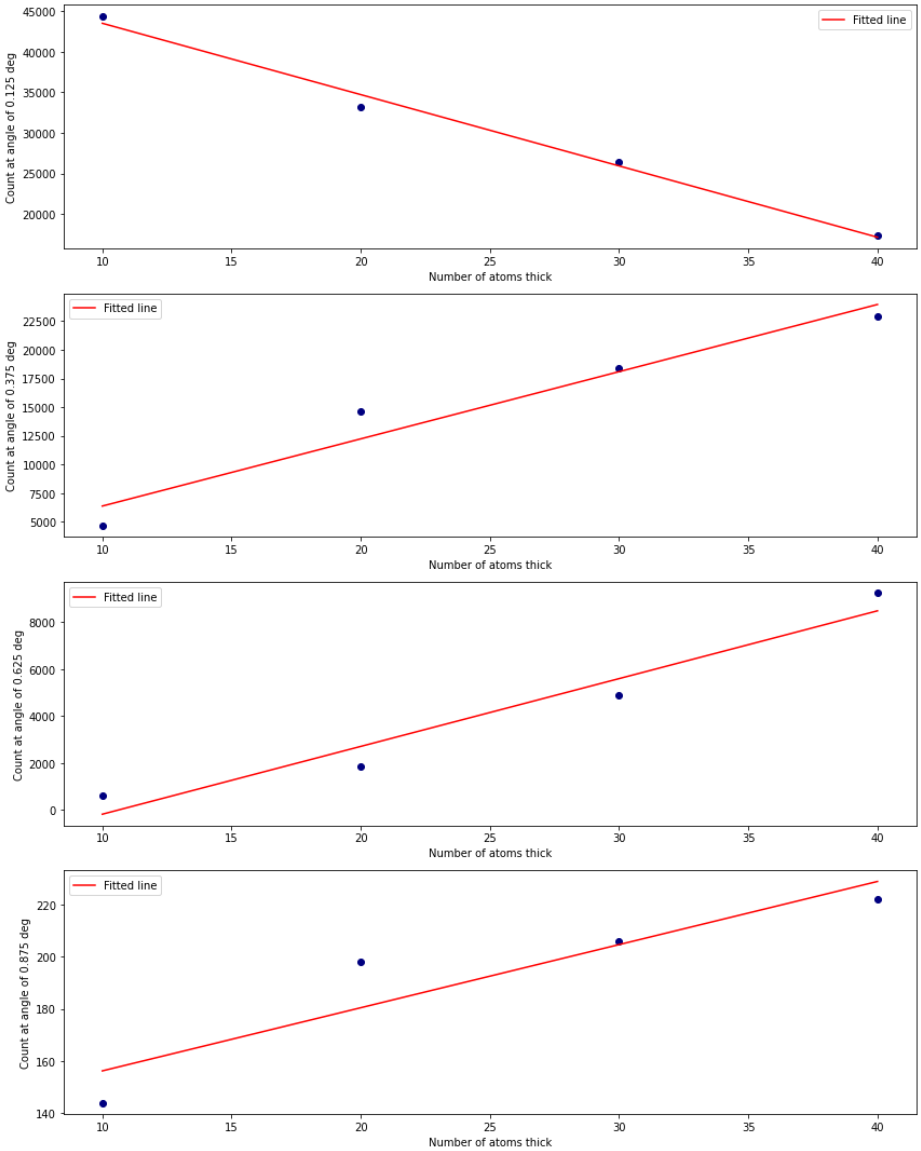
Parallel side view of matrix rotated 15° about the y-axis and 10° about the z-axis.

Matrix rotation code

```
def rotate_y(theta):
        return np.matrix([[ math.cos(theta), 0, math.sin(theta)],
                          [ 0              , 1, 0           ],
                          [-math.sin(theta), 0, math.cos(theta)]])

def rotate_z(theta):
    return np.matrix([[ math.cos(theta), -math.sin(theta), 0],
                      [ math.sin(theta),  math.cos(theta), 0],
                      [ 0              , 0               , 1]])
```

Following code inserted after the lattice has been centred at the origin:

```
#rotate the atom positions in y and z
atom_positions = np.matmul(atom_positions, rotate_y(math.radians(15)))
atom_positions = np.matmul(atom_positions, rotate_z(math.radians(10)))

#turn back from matrix to numpy array after matrix multiplication
atom_positions = np.squeeze(np.asarray(atom_positions))
```

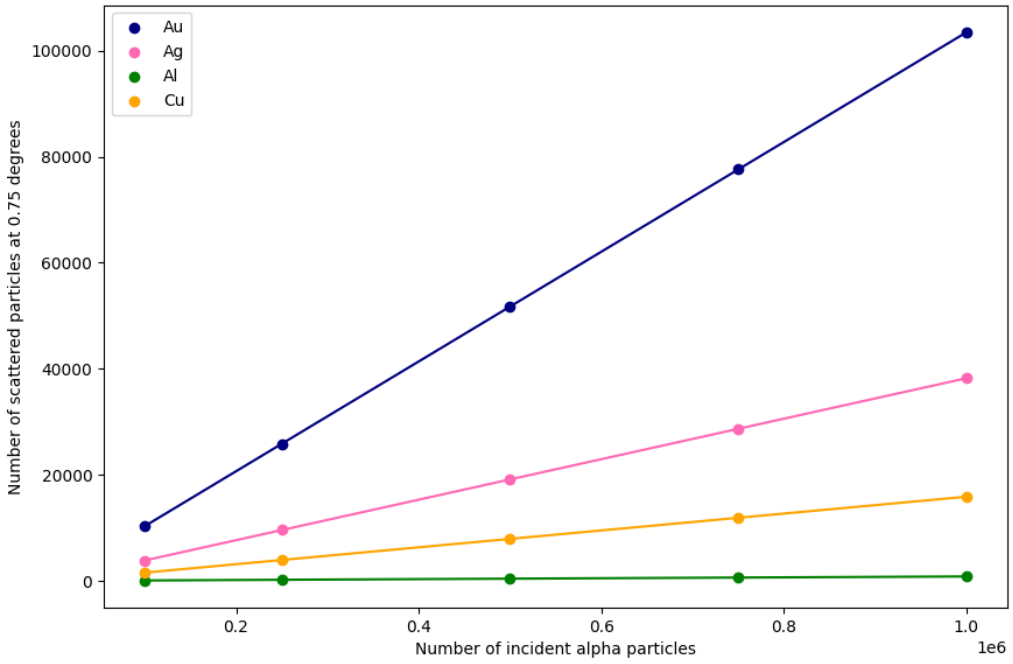


We can observe a much clearer proportionality relationship, where the number of counts increases as the thickness increase. However, we have yet to confirm this for thicker lattices and a better approach is needed to confirm this relationship.

Variation in no. of scattered α-particles with the number of incident particles

Though one might argue not to be a relationship at all, the number of scattered alpha particles at a specified angle should be proportional to the number of incident particles fired N_0.

We choose a bin size of 0.5 and an angle of 0.75°, and plot using data from firing at 10 atom thick foils of varying elements.



Clearly, this proportionality relationship is observed.

5. Conclusion

Due to time constraints and limitations of our model, comparing the results with real world data was not possible. Even so, small systematic errors in experiments can lead to widely varying results. Nonetheless, it would be plausible to assume that since nearly all scattering experimental results agree with Rutherford's formula, testing against the formula would suffice.

Overall, we have achieved most of the stated aims for investigating whether nuclear scattering agrees with the mathematical findings of Rutherford, as was carried out in the 1913 Geiger-Marsden experiments. Further investigation is required for stronger evidence to confirm the proportionality relationship with thickness.

Limitations and Future Improvement

- The most challenging part was simulating lattice scattering and finding ways to speed up computations due to having to "fire" alpha particles many times. Perhaps other computational techniques could be employed (caching), or exploiting the structure of the crystal and creating a new algorithm so that the entire lattice does not need to be generated. The slow speed of Python certainly does not help us in carrying out these numerically-intensive functions, and other lower-level languages like C++ would be favourable. Also, at the moment the width of the lattice is smaller than its length when it should be much larger, to account for particles scattering off to the side, so better lattice generation techniques need to be implemented
- A way for retaining Newton's 3rd Law without significantly reducing computational efficiency remains yet to be implemented, and investigating whether there might be noticeable differences in very large lattices
- A non-uniform nucleus could be constructed instead of having a point mass and charge, although one might suspect it would not make a noticeable difference
- Much improvement can be made to better simulate the ionising nature of the alpha particles and simulate the way the "beam" works
- A way to generate a non-uniform or non-perfect crystal structure to better reflect reality
- Accounting for the nuclear strong force to account for particles of extreme energies, showing the scattered intensity depart from the Rutherford scattering formula

- Other types of materials and metals with different crystal structures like BCC could be tested to investigate any differences they make
- In an environment without time constraints, we could simulate firing many more alpha particles which should offer even stronger evidence to confirm these relationships

References

Geiger, H., & Marsden, E. (1913). LXI. *The laws of deflexion of a particles through large angles*. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, *25*(148), 604–623. https://doi.org/10.1080/14786440408634197

Wikipedia contributors. (2022a, April 1). *Rutherford scattering*. Wikipedia. https://en.wikipedia.org/wiki/Rutherford_scattering

Wikipedia contributors. (2022, May 4). *Geiger–Marsden experiments*. Wikipedia. https://en.wikipedia.org/wiki/Geiger%E2%80%93Marsden_experiments

Rutherford Scattering. Hyperphysics. http://hyperphysics.phy-astr.gsu.edu/hbase/rutsca.html

Rutherford scattering: Measuring the scattering rate as a function of the scattering angle and the atomic number. Physics Leaflets. https://www.ld-didactic.de/literatur/hb/e/p6/p6521_e.pdf

GeeksforGeeks. (2021, June 24). *Coulomb's Law*. https://www.geeksforgeeks.org/coulombs-law/#:%7E:text=Coulomb%20studied%20the%20force%20between,a%20line%20that%20connects%20them.

Rutherford Scattering — Modern Lab Experiments documentation. (2022). Werner Boeglin. https://wanda.fiu.edu/boeglinw/courses/Modern_lab_manual3/Rutherford.html