Nuclear Scattering Model

Authors: Simba Shi, Christopher Whitfeld

Eton College Computational Physics Prize 2022

Contents

1. Introduction

1.1 Pre-Rutherford Problems

The concept of the atom has existed since the days of the Ancient Greeks. These models were at the best of times rudimentary, yet survived in the mainstream scientific thought until the late 1800's.

The first idea that had any semblance to the the modern atomic model was from the Greek Philosopher Demokritos who believed that if you were to say cut up a piece of wood, after many cuts, it would be too small to cut. This is where the name of the atom comes from; "atomos" meaning uncuttable in Greek.

Major developments to this concept came with John Dalton's hypothesis that all matter was made of distinct particles, now named atoms. He imagined them as solid balls, but not composed of even smaller particles

It was only Joseph J. Thomson who discovered the electron. He stipulated that an atom has a structure of a plum pudding. This analogy was determined as he reasoned that there must be positive charge to counterbalance the negative charge of the electron. Therefore, this led to Thomson proposing that atoms could be described as negative particles floating within a 'soup' of positive charges.

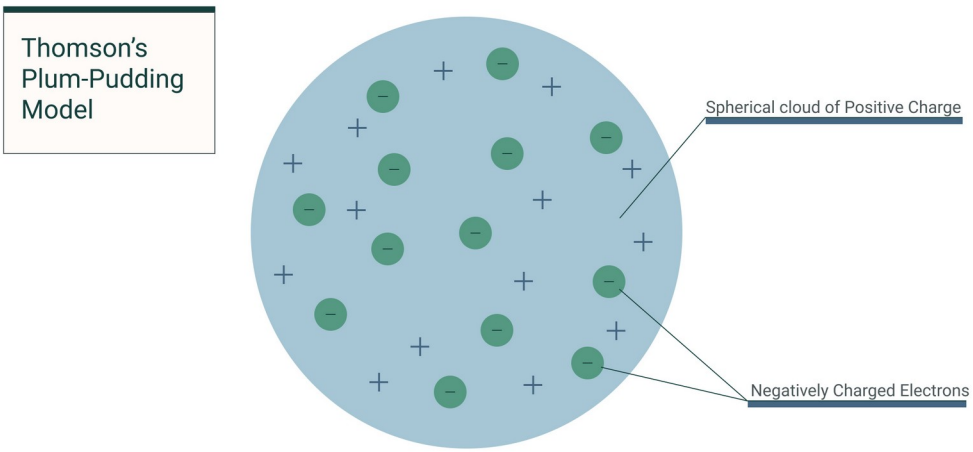


Figure 1.1a, J.J. Thomson's Plum Pudding Model, 1897

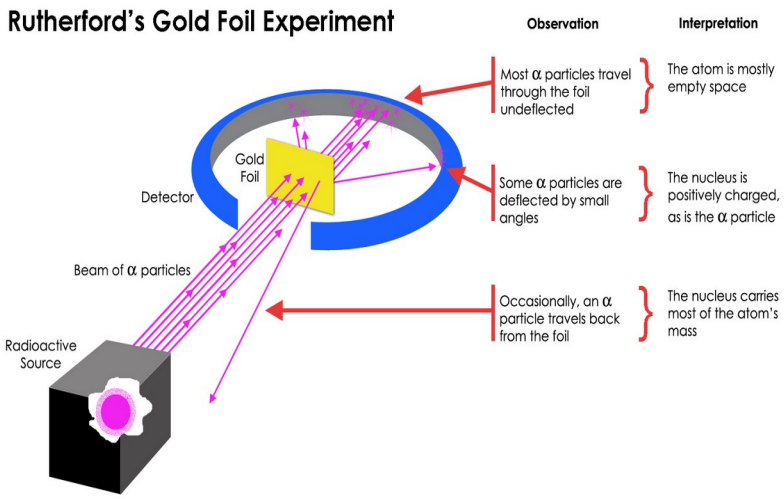
However, this "Plum Pudding Model" failed to explain how a positive charge holds the electrons, with their negative charge. Hence, there was no explanation of the atom's stability.

Equally, it was unable to prove the scattering of alpha particles, as under the model, alpha particles would pass through the gold foil as opposed to being deflected. This is where Rutherford's model comes into play.

1.2 Rutherford's Gold Foil Experiment

Seeing the flaws in J.J. Thomson's model for the atom, Ernest Rutherford, a physicist from New Zealand, with his undergraduate Ernest Marsden and the physicist Hans Geiger undertook a series of experiments between 1908 and 1913.

The experiment that is being used in this challenge is in fact a summarisation of these experiments.



In short, the experiment consisted of aiming beams of positively charge alpha particles at very thin gold foil through a small point in a block of lead. Following Thomson's model, the particles should have passed straight through this gold foil, however it was noticed that although most alpha particles passed through the foil without change, a few were deflected slightly and a fraction by greater angles.

These unexpected alpha particles were detected by and a screen coated with zinc sulfide to render it fluorescent served as a counter to detect alpha particles. Each alpha particle struck the fluorescent screen producing a scintillation (burst of light), which was visible through a viewing microscope attached to the back of the screen.

Therefore, Rutherford concluded that an atom must be mainly empty as most particles didn't deviate. The fraction that did therefore must have been deflected by a strong repulsive force, likely in the centre of the atom. This resulted in his model where an atom consists of one central nucleus and a cloud of electrons.

Figure 1.2b, Thomson's model compared to Rutherfords model of an atom

1.3 Impact on Modern Science

Though the model was not perfect, it laid the groundwork for further developments as it was technically impossible according to Newtonian Physics. Therefore further developments were made by Niels Bohr and the now fully accepted Quantum Mechanical Model, set out by Erwin Schrödinger.

2. Project Overview

2.1 Fundamental Aims

This project, inspired by the Geiger-Marsden gold foil experiment which started in 1908, aims to simulate the scattering of alpha particles when fired at gold atoms, and to reproduce the results obtained in the original experiment. At each time step, we will use Coulomb's law to calculate the force of attraction between a given alpha particle and the gold atoms, and update the alpha particle's position and velocity using the kinematic equations. The trajectories of the alpha particles will be plotted out on a graph.

2.2 Algorithm Summary

The algorithmic flow can be summarised as follows:

1. Select the number of alpha particles to fire at the gold nuclei, as well as their starting positions.
2. Start moving the alpha particles towards the gold nuclei.
3. Calculate the distances between the alpha particle and each of the gold nuclei using Pythagoras Theorem: $r=\sqrt{x^2+y^2}$, where x and y are the distances on the x and y axes respectively.
4. Calculate the force of attraction between the alpha particle and each of the gold nuclei using Coulomb's law $F=k\frac{q_1 q_2}{r^2}$, where r is the distance calculated in the previous step.
5. For each gold nucleus, calculate the alpha particle's acceleration towards it, using Newton's second law of motion $a=\frac{F}{m}$, where F was calculated in the previous step.
6. Update the alpha particle's position and velocity using the kinematic equations:
7. Plot out the alpha particle's new coordinates on the graph.
8. Repeat steps 3-7 until the alpha particle runs off the graph.

2.3 Mathematical Formulae

In order to model the scattering of the alpha particles, three key sets of equations need to be applied.

2.3.1 Coulomb's law

Shown by the formula:

$$F = k\frac{q_1 q_2}{r^2}$$

Where q_1 and q_2 are the charges of the gold nucleus and the alpha particle. r is the distance between them. And where k is the Coulomb's constant, given by:

$$k = \frac{1}{4\pi\epsilon_0}$$

Subsequently, ϵ_0 is the electric constant which is in turn given the value:

$$\epsilon_0 = 8.854\,187\,8128 \times 10^{-12}\,F\,m^{-1}$$

Instead of working out the raw force with Coulomb's Law, we instead calculated the acceleration caused by the interactions between the alpha particles and nucleus by dividing the result by the mass of the alpha particle. This is so as to apply the result in the kinematic equations

2.3.2 Newton's second law of motion

This was used in order to work out the acceleration. Given by:

$$Force = mass \times acceleration$$

Simple rearranging gives:

$$acceleration = \frac{Force}{mass}$$

As the force was worked out in Coulomb's Law, it is very simple therefore to apply this equation to work out the acceleration of the Alpha Particle.

2.3.3 Kinematic equations

Kinematics, dealing with the motions and movements of objects is a major part of the mathematical calculations in the project. We have used multiple equations from this branch of physics to ultimately work out the position of an alpha particle every iteration.

$$x(t+\Delta t) = x(t) + v(t)\,\Delta t$$

Used to update the location of the alpha particles. The v calculated with:

$$v(t+\Delta t) = v(t) + a(t)\,\Delta t$$

3. The Rutherford Scattering Models

3.1 List of scattering models

Below is a list of scattering models which we will implement in this project:

- Scattering from 1 stationary nucleus, on a 2D surface.
- Scattering from a lattice, on a 2D surface.
- Scattering from a lattice, in a 3D space.

3.2 Initialisation

Before we begin modelling Rutherford scattering, we first must choose a programming language to use, as well as a few essential libraries to help with calculations and graph plotting.

3.2.1 Programming language

In this project, we will use Python as our programming language; Python is an interpreted high-level programming language, and is widely used in many domains, including data science and research. Python is easily integrated into Jupyter Notebooks, which is the file format we are using.

3.2.2 Libraries

In order to complete the first model of the Gold Foil Experiment, an array of Python Libraries need to be imported.

Math:

The Math library is a built-in Python library, and it allows us to use various constants such as π and operations such as square rooting. This is therefore important to calculating alpha particle trajectories.

Numpy:

Numpy serves to perform more efficient mathematical operations. Particularly when dealing with vectors and matrices, we have used numpy arrays that consume less memory than Python lists.

Matplotlib:

By far the most convenient way to represent results on a graph, we have applied Matplotlib so that all results are presented visually.

3.2.3 Imported Libraries

In the code block below, we import the necessary Python modules.

```
import math

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

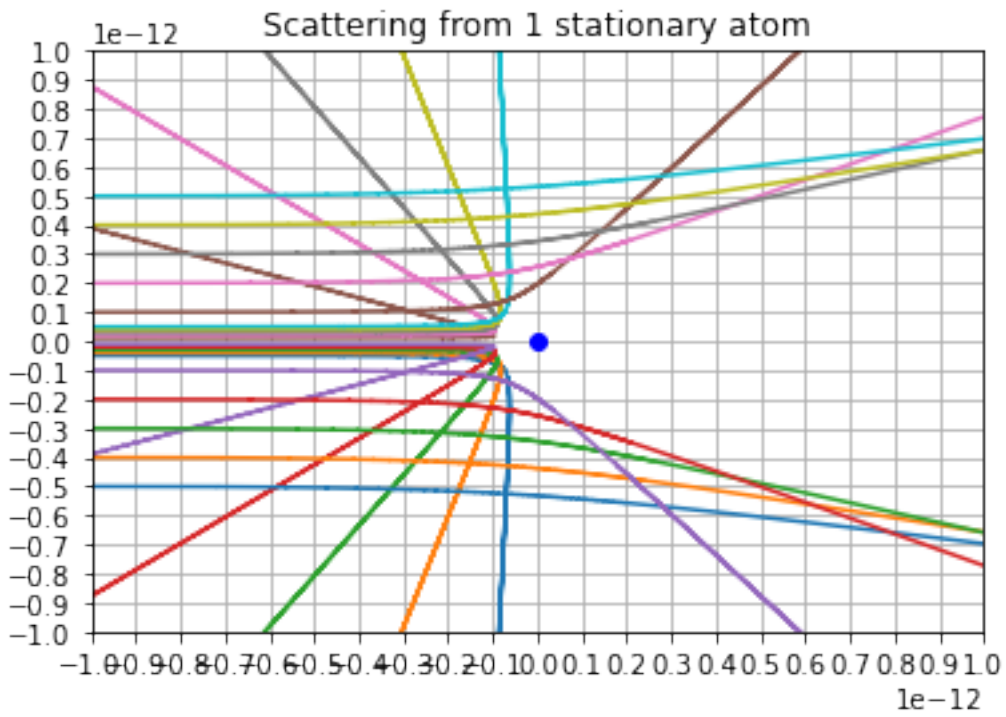
3.3 Scattering from 1 stationary nucleus

For our first model, we shoot alpha particles at a single stationary gold nucleus, and their trajectories are plotted on a 2D graph.

```
# Create MatplotLib graph.
fig, ax = plt.subplots()

# Parameters and constants.
axis_range = 1e-12  # range of the axes
delta_time = 1.0e-24  # time step for kinematic equations
velocity_x0 = 1.0e7  # initial velocity of the alpha particle, along
the x-axis
velocity_y0 = 0  # initial velocity of the alpha particle, along the
y-axis
proton_charge = 1.6e-19  # electric charge of 1 proton (in coulombs)
atomic_mass_constant = 1.661e-27  # atomic mass constant
x0 = -axis_range  # initial x coordinate of alpha particles
coulombs_constant = 1 / (4 * math.pi * 8.85e-12)  # Coulomb's constant
alpha_particle_mass = 4 * atomic_mass_constant  # alpha particle's
(helium nucleus) atomic mass
gold_nucleus_charge = 79 * proton_charge  # electric charge of a gold
nucleus (in coulombs)
alpha_particle_charge = 2 * proton_charge  # electric charge of an
alpha particle (in coulombs)


# Calculation of the Coulomb force acceleration.
def get_coulomb_acceleration(nucleus1_charge, nucleus2_charge, r1, r2,
mass):
    return coulombs_constant * nucleus1_charge * nucleus2_charge /
(math.sqrt(r1 ** 2 + r2 ** 2)) ** 2 / mass


def single_nucleus(plot_title):
    params = []
    # The range of alpha particles y coordinates to fire at nucleus.
Values can be changed.
    for i in range(-5, 6, 1):
        if i != 0:
            params.append(i*1e-14)
    for i in range(-50, 51, 10):
        if i != 0:
            params.append(i*1e-14)
```

```
    # Calculating the course of the trajectories of molecules.
    # x0 is the starting x coordinate of alpha particles, and y0 the
starting y coordinate.
    for y0 in params:
        # List of x coordinates the alpha particle has travelled
through.
        x_coordinates = [x0]
        # List of y coordinates the alpha particle has travelled
through.
        y_coordinates = [y0]
        # Calculate next x coordinate after time step (h) passes using
kinematic equation.
        # x(t+Δt)=x(t)+v(t)Δt
        x = x0 + delta_time * velocity_x0
        y = y0 + delta_time * velocity_y0

        # v_x = initial velocity + acceleration * interval
        velocity_x = velocity_x0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * x / math.sqrt(x ** 2 + y ** 2) *
delta_time
        velocity_y = velocity_y0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * y / math.sqrt(x ** 2 + y ** 2) *
delta_time

        # Save previous velocities in x and y axes.
        prev_velocity_x = velocity_x0
        prev_velocity_y = velocity_y0

        # Save previous positions for x and y axes.
        prev_x = x0
        prev_y = y0

        # We record new positions and velocities as long as the alpha
particle is within range.
        while np.isfinite(y) and -axis_range < x < axis_range:
            # Add current x and y positions to the lists recording the
histories of where the alpha particle has been.
            x_coordinates.append(x)
            y_coordinates.append(y)

            constAx = get_coulomb_acceleration(gold_nucleus_charge,
alpha_particle_charge, x, y, alpha_particle_mass) * x / math.sqrt(x **
2 + y ** 2)
            constAy = get_coulomb_acceleration(gold_nucleus_charge,
alpha_particle_charge, x, y, alpha_particle_mass) * y / math.sqrt(x **
2 + y ** 2)
```

```
            # Update position values.
            # x(t+Δt)=x(t)+v(t)Δt
            temp = x
            x = prev_x + prev_velocity_x * delta_time
            prev_x = temp

            temp = y
            y = prev_y + prev_velocity_y * delta_time
            prev_y = temp

            # Updating the velocity of particles moving along the
axis.
            temp = velocity_x
            velocity_x = prev_velocity_x + constAx * delta_time
            prev_velocity_x = temp

            temp = velocity_y
            velocity_y = prev_velocity_y + constAy * delta_time
            prev_velocity_y = temp

        # Title of the graph.
        plt.title(plot_title)
        # Generating points on the x-axis every 0.1 point.
        ax.xaxis.set_major_locator(ticker.MultipleLocator(axis_range /
10))
        # generating points on the y-axis every 0.1 point.
        ax.yaxis.set_major_locator(ticker.MultipleLocator(axis_range /
10))
        # Display grid on the graph.
        plt.grid(True)
        # Display trajectories of alpha particles.
        ax.plot(x_coordinates, y_coordinates)

    # Displaying a gold atom in the centre of the graph.
    ax.plot(0, 0, "bo")
    # X-axis scale.
    plt.xlim(-axis_range, axis_range)
    # Y-axis scale.
    plt.ylim(-axis_range, axis_range)
    plt.show()


single_nucleus("Scattering from 1 stationary atom")
```



3.4 Scattering from a 2D lattice

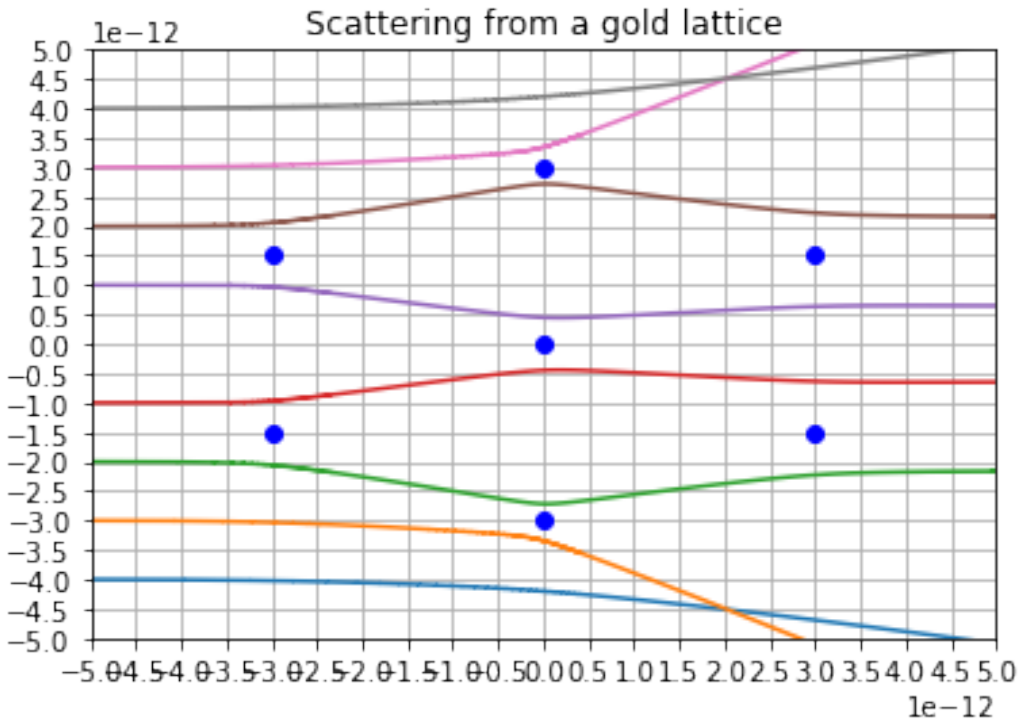
Now we will shoot alpha particles at a 2-dimensional lattice of gold atoms and plot out their trajectories.

```
# Create MatplotLib graph.
fig, ax = plt.subplots()

# Parameters and constants.
axis_range = 5e-12  # range of the axes
delta_time = 1.0e-22  # range of the axes
velocity_x0 = 1.0e7  # initial velocity of the alpha particle, along
the x-axis
velocity_y0 = 0  # initial velocity of the alpha particle, along the
y-axis
proton_charge = 1.6e-19  # electric charge of 1 proton (in coulombs)
atomic_mass_constant = 1.661e-27  # atomic mass constant
x0 = -axis_range  # initial x coordinate of alpha particles
atom_separation = 3e-12  # how far the gold atoms should be apart from
each other in the lattice
gold_atoms_coordinates = np.array([[0, 0], [0, atom_separation], [0, -
atom_separation], [atom_separation, atom_separation * 0.5],
[atom_separation, -atom_separation * 0.5], [-atom_separation,
atom_separation * 0.5], [-atom_separation, -atom_separation * 0.5]]
)
coulombs_constant = 1 / (4 * math.pi * 8.85e-12)  # Coulomb's constant
alpha_particle_mass = 4 * atomic_mass_constant  # alpha particle's
(helium nucleus) atomic mass
```

```
gold_nucleus_charge = 79 * proton_charge  # electric charge of a gold
nucleus (in coulombs)
alpha_particle_charge = 2 * proton_charge  # electric charge of an
alpha particle (in coulombs)


# Calculation of the Coulomb force acceleration.
def get_coulomb_acceleration(nucleus1_charge, nucleus2_charge, r1, r2,
mass):
    return coulombs_constant * nucleus1_charge * nucleus2_charge /
(math.sqrt(r1 ** 2 + r2 ** 2)) ** 2 / mass


def two_dimensional_lattice():
    # Alpha particle starting y positions.
    params = []
    for i in range(-4, 5):
        if i != 0:
            params.append(i * 1e-12)

    # Calculating the course of the trajectories of molecules.
    # x0 is the starting x coordinate of alpha particles, and y0 the
starting y coordinate.
    for y0 in params:
        # List of x coordinates the alpha particle has travelled
through.
        x_coordinates = [x0]
        # List of y coordinates the alpha particle has travelled
through.
        y_coordinates = [y0]
        # Calculate next x coordinate after time step (h) passes using
kinematic equation.
        # x(t+Δt)=x(t)+v(t)Δt
        x = x0 + delta_time * velocity_x0
        y = y0 + delta_time * velocity_y0

        # v(t+Δt)=v(t)+a(t)Δt
        velocity_x = velocity_x0
        velocity_y = velocity_y0
        for gold_atom_x, gold_atom_y in gold_atoms_coordinates:
            x_separation = x - gold_atom_x
            y_separation = y - gold_atom_y
            velocity_x +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, alpha_particle_mass) * x_separation /
math.sqrt(x_separation ** 2 + y_separation ** 2) * delta_time
            velocity_y +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, alpha_particle_mass) * y_separation /
math.sqrt(x_separation ** 2 + y_separation ** 2) * delta_time
```

```
        # Save previous velocities in x and y axes.
        prev_velocity_x = velocity_x0
        prev_velocity_y = velocity_y0

        # Save previous positions for x and y axes.
        prev_x = x0
        prev_y = y0

        # We record new positions and velocities as long as the alpha
particle is within range.
        while np.isfinite(y) and -axis_range < x < axis_range:
            # Add current x and y positions to the lists recording the
histories of where the alpha particle has been.
            x_coordinates.append(x)
            y_coordinates.append(y)

            constAx = 0
            constAy = 0
            for gold_atom_x, gold_atom_y in gold_atoms_coordinates:
                x_separation = x - gold_atom_x
                y_separation = y - gold_atom_y
                constAx +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, alpha_particle_mass) * x_separation /
math.sqrt(x_separation ** 2 + y_separation ** 2)
                constAy +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, alpha_particle_mass) * y_separation /
math.sqrt(x_separation ** 2 + y_separation ** 2)

            # Updating of positions on the axes
            # x(t+Δt)=x(t)+v(t)Δt
            temp = x
            x = prev_x + prev_velocity_x * delta_time
            prev_x = temp

            temp = y
            y = prev_y + prev_velocity_y * delta_time
            prev_y = temp

            # Updating the velocity of particles moving along the axis
            temp = velocity_x
            velocity_x = prev_velocity_x + constAx * delta_time
            prev_velocity_x = temp

            temp = velocity_y
            velocity_y = prev_velocity_y + constAy * delta_time
            prev_velocity_y = temp
```

```
        # Title of the graph.
        plt.title("Scattering from a gold lattice")
        # Generating points on the x-axis every 0.1 point.
        ax.xaxis.set_major_locator(ticker.MultipleLocator(axis_range /
10))
        # Generating points on the y-axis every 0.1 point.
        ax.yaxis.set_major_locator(ticker.MultipleLocator(axis_range /
10))
        # Display grid on the graph.
        plt.grid(True)
        # Display trajectories of alpha particles.
        ax.plot(x_coordinates, y_coordinates)

    # Displaying gold atoms.
    ax.plot(0, 0, "bo")
    ax.plot(0, atom_separation, "bo")
    ax.plot(0, -atom_separation, "bo")
    ax.plot(atom_separation, atom_separation * 0.5, "bo")
    ax.plot(atom_separation, -atom_separation * 0.5, "bo")
    ax.plot(-atom_separation, atom_separation * 0.5, "bo")
    ax.plot(-atom_separation, -atom_separation * 0.5, "bo")
    # X-axis scale
    plt.xlim(-axis_range, axis_range)
    # Y-axis scale
    plt.ylim(-axis_range, axis_range)
    plt.show()


two_dimensional_lattice()
```



3.5 Scattering from a 3D lattice

Furthermore, we will shoot alpha particles at a 3-dimensional lattice of gold atoms.

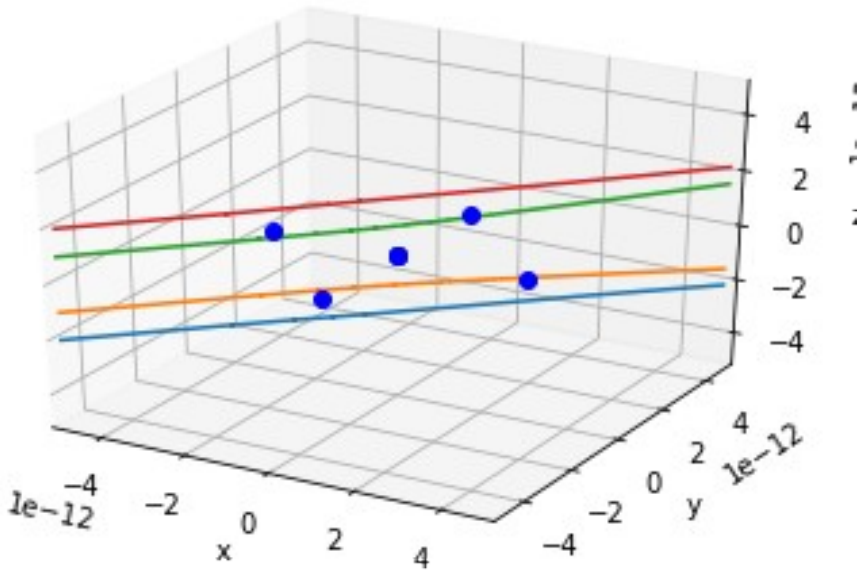
```
ax = plt.axes(projection="3d")

# Parameters and constants.
axis_range = 5e-12  # range of the axes
delta_time = 1.0e-21  # range of the axes
velocity_x0 = 1.0e7  # initial velocity of the alpha particle, along
the x-axis
velocity_y0 = 1.0e7  # initial velocity of the alpha particle, along
the y-axis
velocity_z0 = 0
proton_charge = 1.6e-19  # electric charge of 1 proton (in coulombs)
atomic_mass_constant = 1.661e-27  # atomic mass constant
x0 = -axis_range  # initial x coordinate of alpha particles
y0 = -axis_range  # initial y coordinate of alpha particles
atom_separation = 3e-12  # how far the gold atoms should be apart from
each other in the lattice
gold_atoms_coordinates = np.array([[0, 0, 0], [0, atom_separation, 0],
[0, -atom_separation, 0], [atom_separation, 0, 0], [-atom_separation,
0, 0], [0, 0, atom_separation], [0, 0, -atom_separation]])
# gold_atoms_coordinates = np.array([[0, 0, 0]])
coulombs_constant = 1 / (4 * math.pi * 8.85e-12)  # Coulomb's constant
alpha_particle_mass = 4 * atomic_mass_constant  # alpha particle's
(helium nucleus) atomic mass
```

```
gold_nucleus_charge = 79 * proton_charge  # electric charge of a gold
nucleus (in coulombs)
alpha_particle_charge = 2 * proton_charge  # electric charge of an
alpha particle (in coulombs)


# Calculation of the Coulomb force acceleration.
def get_coulomb_acceleration(nucleus1_charge, nucleus2_charge, r1, r2,
r3, mass):
    return coulombs_constant * nucleus1_charge * nucleus2_charge /
(math.sqrt(r1 ** 2 + r2 ** 2 + r3 ** 2)) ** 2 / mass


def three_dimensional_lattice():
    # Alpha particle starting y positions.
    params = []
    for i in range(-2, 3):
        if i != 0:
            params.append(i * 1e-12)


    # Calculating the course of the trajectories of molecules.
    # x0 is the starting x coordinate of alpha particles, and y0 the
starting y coordinate.
    for z0 in params:
        # List of x coordinates the alpha particle has travelled
through.
        x_coordinates = [x0]
        # List of y coordinates the alpha particle has travelled
through.
        y_coordinates = [y0]
        z_coordinates = [z0]
        # Calculate next x coordinate after time step (h) passes using
kinematic equation.
        # x(t+Δt)=x(t)+v(t)Δt
        x = x0 + delta_time * velocity_x0
        y = y0 + delta_time * velocity_y0
        z = z0 + delta_time * velocity_z0

        # v(t+Δt)=v(t)+a(t)Δt
        velocity_x = velocity_x0
        velocity_y = velocity_y0
        velocity_z = velocity_z0
        for gold_atom_x, gold_atom_y, gold_atom_z in
gold_atoms_coordinates:
            x_separation = x - gold_atom_x
            y_separation = y - gold_atom_y
            z_separation = z - gold_atom_z
            velocity_x +=
```

```
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, z_separation, alpha_particle_mass) *
x_separation / math.sqrt(x_separation ** 2 + y_separation ** 2 +
z_separation ** 2) * delta_time
            velocity_y +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, z_separation, alpha_particle_mass) *
y_separation / math.sqrt(x_separation ** 2 + y_separation ** 2 +
z_separation ** 2) * delta_time
            velocity_z +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, z_separation, alpha_particle_mass) *
z_separation / math.sqrt(x_separation ** 2 + y_separation ** 2 +
z_separation ** 2) * delta_time

        # Save previous velocities in x, y and z axes.
        prev_velocity_x = velocity_x0
        prev_velocity_y = velocity_y0
        prev_velocity_z = velocity_z0

        # Save previous positions for x and y axes.
        prev_x = x0
        prev_y = y0
        prev_z = z0

        # We record new positions and velocities as long as the alpha
particle is within range.
        while np.isfinite(z) and -axis_range < x < axis_range and -
axis_range < y < axis_range:
            # Add current x and y positions to the lists recording the
histories of where the alpha particle has been.
            x_coordinates.append(x)
            y_coordinates.append(y)
            z_coordinates.append(z)

            constAx = 0
            constAy = 0
            constAz = 0
            for gold_atom_x, gold_atom_y, gold_atom_z in
gold_atoms_coordinates:
                x_separation = x - gold_atom_x
                y_separation = y - gold_atom_y
                z_separation = z - gold_atom_z
                constAx +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, z_separation, alpha_particle_mass) *
x_separation / math.sqrt(x_separation ** 2 + y_separation ** 2 +
z_separation ** 2)
                constAy +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
```

```
x_separation, y_separation, z_separation, alpha_particle_mass) *
y_separation / math.sqrt(x_separation ** 2 + y_separation ** 2 +
z_separation ** 2)
                constAz +=
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x_separation, y_separation, z_separation, alpha_particle_mass) *
z_separation / math.sqrt(x_separation ** 2 + y_separation ** 2 +
z_separation ** 2)

            # Updating of positions on the axes
            # x(t+Δt)=x(t)+v(t)Δt
            temp = x
            x = prev_x + prev_velocity_x * delta_time
            prev_x = temp

            temp = y
            y = prev_y + prev_velocity_y * delta_time
            prev_y = temp

            temp = z
            z = prev_z + prev_velocity_z * delta_time
            prev_z = temp

            # Updating the velocity of particles moving along the axis
            temp = velocity_x
            # x(t+Δt)=x(t)+v(t)Δt
            velocity_x = prev_velocity_x + constAx * delta_time
            prev_velocity_x = temp

            temp = velocity_y
            velocity_y = prev_velocity_y + constAy * delta_time
            prev_velocity_y = temp

            temp = velocity_z
            velocity_z = prev_velocity_z + constAz * delta_time
            prev_velocity_z = temp

        # Display grid on the graph.
        plt.grid(True)
        # Display trajectories of alpha particles.
        ax.plot3D(x_coordinates, y_coordinates, z_coordinates)

    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    # Displaying gold atoms.
    ax.plot([0], [0], [0], "bo")
    ax.plot([0], [atom_separation], [0], "bo")
    ax.plot([0], [-atom_separation], [0], "bo")
```

```
    ax.plot([atom_separation], [0], [0], "bo")
    ax.plot([-atom_separation], [0], [0], "bo")
    ax.plot([0], [0], [-atom_separation], "bo")
    ax.plot([0], [0], [atom_separation], "bo")
    # X-axis scale
    plt.xlim(-axis_range, axis_range)
    # Y-axis scale
    plt.ylim(-axis_range, axis_range)
    # Z-axis scale
    ax.set_zlim(-axis_range, axis_range)
    plt.show()


three_dimensional_lattice()
```



4. Data Analaysis

4.1 Plotting the Number of Scattered Particles to Scattering Angle

This was to compare to Rutherford's results to see how accurate he was.

```
delta_time = 1.0e-21  # time step for kinematic equations


scattering_angles = []


def data_analysis():
```

```
    params = []
    # The range of alpha particles y coordinates to fire at nucleus.
Values can be changed.
    for i in np.arange(-300, 300, 0.01):
        if i != 0:
            params.append(i*1e-14)

    # Calculating the course of the trajectories of molecules.
    # x0 is the starting x coordinate of alpha particles, and y0 the
starting y coordinate.
    for y0 in params:
        # List of x coordinates the alpha particle has travelled
through.
        x_coordinates = [x0]
        # List of y coordinates the alpha particle has travelled
through.
        y_coordinates = [y0]
        # Calculate next x coordinate after time step (h) passes using
kinematic equation.
        # x(t+Δt)=x(t)+v(t)Δt
        x = x0 + delta_time * velocity_x0
        y = y0 + delta_time * velocity_y0

        # v_x = initial velocity + acceleration * interval
        velocity_x = velocity_x0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * x / math.sqrt(x ** 2 + y ** 2) *
delta_time
        velocity_y = velocity_y0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * y / math.sqrt(x ** 2 + y ** 2) *
delta_time

        # Save previous velocities in x and y axes.
        prev_velocity_x = velocity_x0
        prev_velocity_y = velocity_y0

        # Save previous positions for x and y axes.
        prev_x = x0
        prev_y = y0

        # We record new positions and velocities as long as the alpha
particle is within range.
        while np.isfinite(y) and -axis_range < x < axis_range:
            # Add current x and y positions to the lists recording the
histories of where the alpha particle has been.
            x_coordinates.append(x)
            y_coordinates.append(y)

            constAx = get_coulomb_acceleration(gold_nucleus_charge,
```

```
alpha_particle_charge, x, y, alpha_particle_mass) * x / math.sqrt(x **
2 + y ** 2)
            constAy = get_coulomb_acceleration(gold_nucleus_charge,
alpha_particle_charge, x, y, alpha_particle_mass) * y / math.sqrt(x **
2 + y ** 2)

            # Update position values.
            # x(t+Δt)=x(t)+v(t)Δt
            temp = x
            x = prev_x + prev_velocity_x * delta_time
            prev_x = temp

            temp = y
            y = prev_y + prev_velocity_y * delta_time
            prev_y = temp

            # Updating the velocity of particles moving along the
axis.
            temp = velocity_x
            velocity_x = prev_velocity_x + constAx * delta_time
            prev_velocity_x = temp

            temp = velocity_y
            velocity_y = prev_velocity_y + constAy * delta_time
            prev_velocity_y = temp

        opposite = y_coordinates[-1] - y0
        adjacent = x_coordinates[-1]
        if x_coordinates[-1] < 0:
            angle = 180 - abs(math.degrees(math.atan(opposite /
adjacent)))
        else:
            angle = abs(math.degrees(math.atan(opposite / adjacent)))
        scattering_angles.append(angle)


data_analysis()
```

The above code is intended to measure the number alpha particles as a function of their scattering angle. The scattering angle was measured using basic trigonometry:

$$\theta° = \arctan\left(\frac{opposite}{adjacent}\right)$$

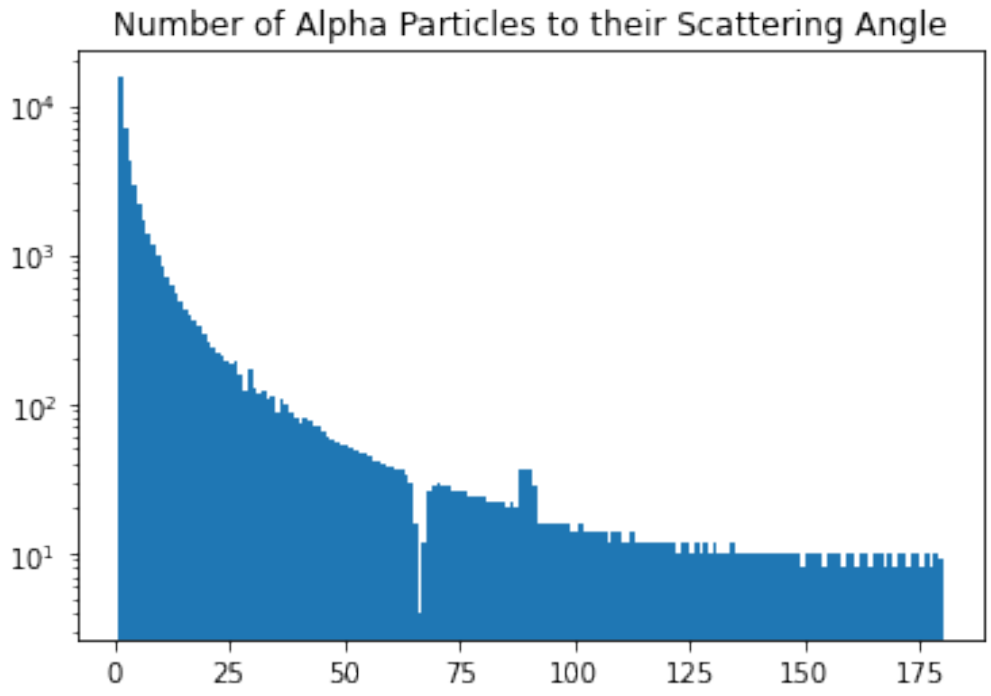
Where the opposite is calculated as the final y coordinate minus the initial position of the alpha particle on the y axis. The adjacent is calculated as the final iteration of the movements of the alpha particle on the x axis.

Equally, if the alpha particles final x coordinate is less than 0, i.e, it has returned from where it was emitted, it's θ is calculated as:

$$\theta° = 180 - \arctan\left(\frac{opposite}{adjacent}\right)$$

```
sorted_scattering_angle = sorted(scattering_angles, reverse=True)

x_axis_angles = []
for i in np.arange(1,181,1):
    x_axis_angles.append(i)

plt.hist(sorted_scattering_angle, bins=x_axis_angles)
plt.yscale("log")


plt.title("Number of Alpha Particles to their Scattering Angle")
```

```
Text(0.5, 1.0, 'Number of Alpha Particles to their Scattering Angle')
```



If we refer to Figure 4.1a below, we see a striking comparison; the graphs gradients are nearly identical. Though our results have brief anomalies for 65 $\theta°$ and 90 $\theta°$, the only major difference is the scale, they extend to 10^7 in comparison to our 10^4, owing to our lack of iterations.

The results therefore show that a computer has been able to correctly model their experiment.

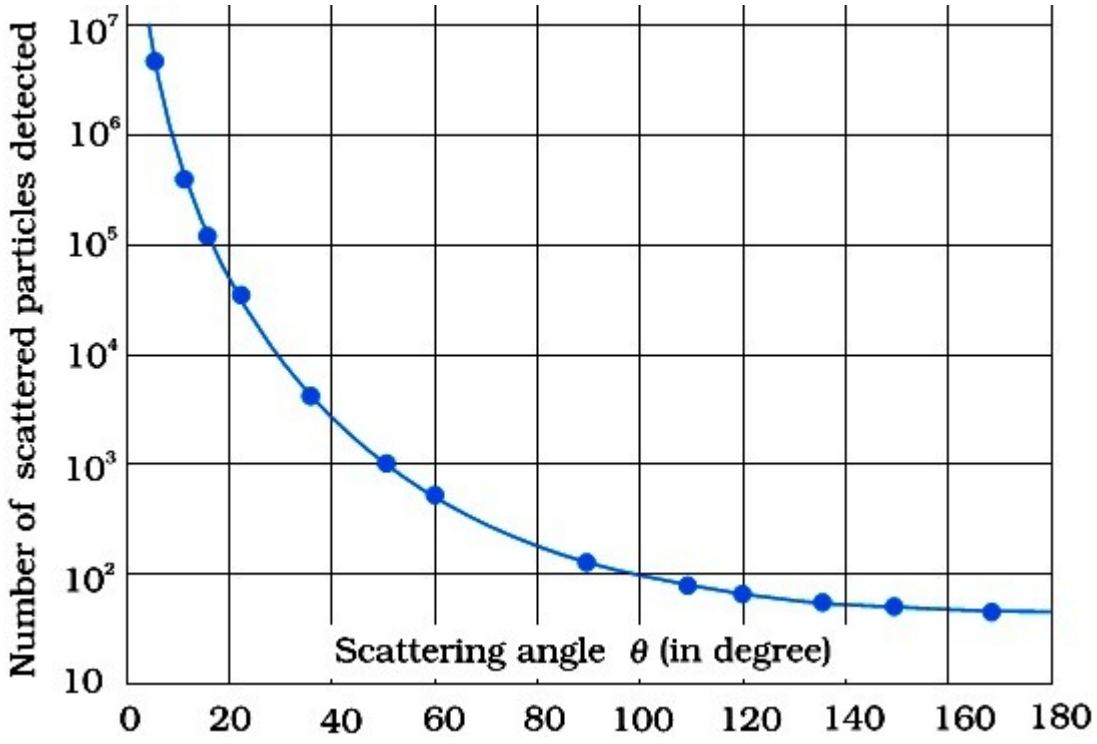


Figure 4.1a, The Geiger-Marsden graph for Number of alpha particles against scattering angle

4.2 Changing speed of alpha particles

We are measuring this in order to determine how well the model works at various speeds. We predict the slower velocities to produce lower

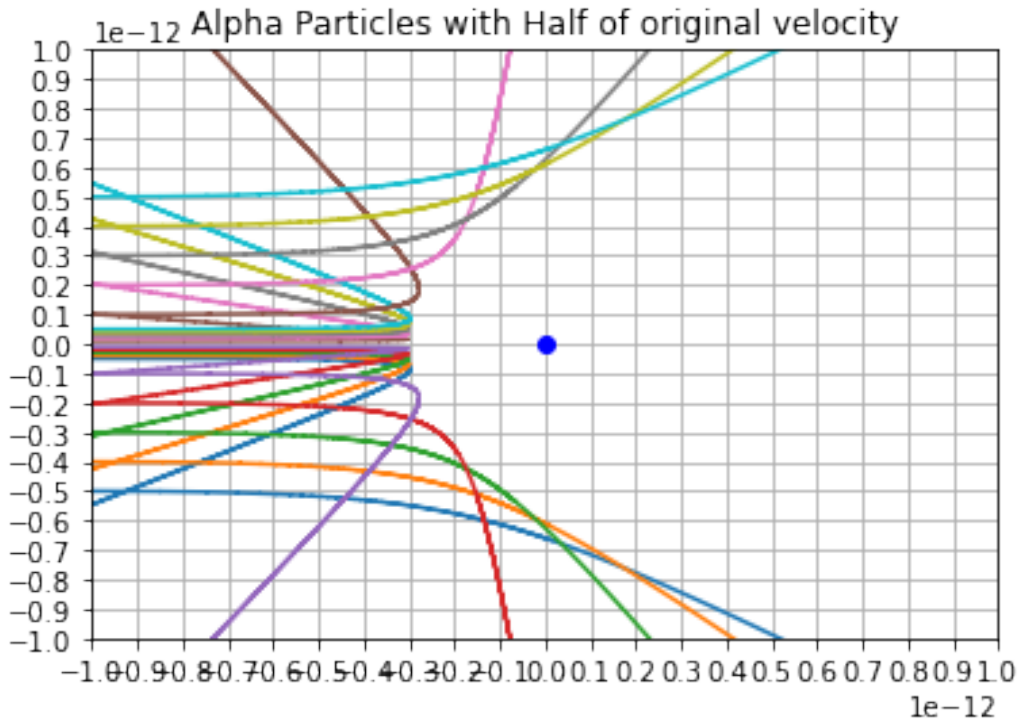
4.2.1 Half of the original speed

```
velocity_x0 = 0.5e7
axis_range = 1e-12
# Create MatplotLib graph.
fig, ax = plt.subplots()

# Parameters and constants.
axis_range = 1e-12  # range of the axes
delta_time = 1.0e-24  # time step for kinematic equations

velocity_y0 = 0  # initial velocity of the alpha particle, along the
y-axis
proton_charge = 1.6e-19  # electric charge of 1 proton (in coulombs)
atomic_mass_constant = 1.661e-27  # atomic mass constant
x0 = -axis_range  # initial x coordinate of alpha particles
coulombs_constant = 1 / (4 * math.pi * 8.85e-12)  # Coulomb's constant
alpha_particle_mass = 4 * atomic_mass_constant  # alpha particle's
(helium nucleus) atomic mass
gold_nucleus_charge = 79 * proton_charge  # electric charge of a gold
nucleus (in coulombs)
```

```
alpha_particle_charge = 2 * proton_charge  # electric charge of an
alpha particle (in coulombs)


# Calculation of the Coulomb force acceleration.
def get_coulomb_acceleration(nucleus1_charge, nucleus2_charge, r1, r2,
mass):
    return coulombs_constant * nucleus1_charge * nucleus2_charge /
(math.sqrt(r1 ** 2 + r2 ** 2)) ** 2 / mass


def single_nucleus(plot_title):
    params = []
    # The range of alpha particles y coordinates to fire at nucleus.
Values can be changed.
    for i in range(-5, 6, 1):
        if i != 0:
            params.append(i*1e-14)
    for i in range(-50, 51, 10):
        if i != 0:
            params.append(i*1e-14)

    # Calculating the course of the trajectories of molecules.
    # x0 is the starting x coordinate of alpha particles, and y0 the
starting y coordinate.
    for y0 in params:
        # List of x coordinates the alpha particle has travelled
through.
        x_coordinates = [x0]
        # List of y coordinates the alpha particle has travelled
through.
        y_coordinates = [y0]
        # Calculate next x coordinate after time step (h) passes using
kinematic equation.
        # x(t+Δt)=x(t)+v(t)Δt
        x = x0 + delta_time * velocity_x0
        y = y0 + delta_time * velocity_y0

        # v_x = initial velocity + acceleration * interval
        velocity_x = velocity_x0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * x / math.sqrt(x ** 2 + y ** 2) *
delta_time
        velocity_y = velocity_y0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * y / math.sqrt(x ** 2 + y ** 2) *
delta_time

        # Save previous velocities in x and y axes.
```

```
        prev_velocity_x = velocity_x0
        prev_velocity_y = velocity_y0

        # Save previous positions for x and y axes.
        prev_x = x0
        prev_y = y0

        # We record new positions and velocities as long as the alpha
particle is within range.
        while np.isfinite(y) and -axis_range < x < axis_range:
            # Add current x and y positions to the lists recording the
histories of where the alpha particle has been.
            x_coordinates.append(x)
            y_coordinates.append(y)

            constAx = get_coulomb_acceleration(gold_nucleus_charge,
alpha_particle_charge, x, y, alpha_particle_mass) * x / math.sqrt(x **
2 + y ** 2)
            constAy = get_coulomb_acceleration(gold_nucleus_charge,
alpha_particle_charge, x, y, alpha_particle_mass) * y / math.sqrt(x **
2 + y ** 2)

            # Update position values.
            # x(t+Δt)=x(t)+v(t)Δt
            temp = x
            x = prev_x + prev_velocity_x * delta_time
            prev_x = temp

            temp = y
            y = prev_y + prev_velocity_y * delta_time
            prev_y = temp

            # Updating the velocity of particles moving along the
axis.
            temp = velocity_x
            velocity_x = prev_velocity_x + constAx * delta_time
            prev_velocity_x = temp

            temp = velocity_y
            velocity_y = prev_velocity_y + constAy * delta_time
            prev_velocity_y = temp

        # Title of the graph.
        plt.title(plot_title)
        # Generating points on the x-axis every 0.1 point.
        ax.xaxis.set_major_locator(ticker.MultipleLocator(axis_range /
10))
        # generating points on the y-axis every 0.1 point.
        ax.yaxis.set_major_locator(ticker.MultipleLocator(axis_range /
```

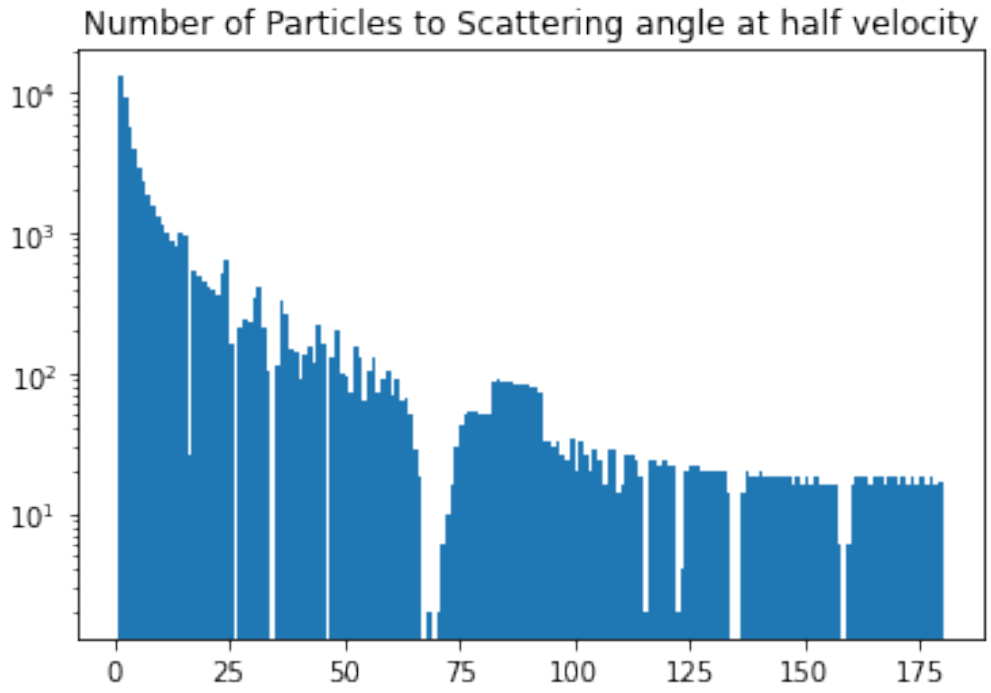
```
10))
        # Display grid on the graph.
        plt.grid(True)
        # Display trajectories of alpha particles.
        ax.plot(x_coordinates, y_coordinates)

    # Displaying a gold atom in the centre of the graph.
    ax.plot(0, 0, "bo")
    # X-axis scale.
    plt.xlim(-axis_range, axis_range)
    # Y-axis scale.
    plt.ylim(-axis_range, axis_range)
    plt.show()



single_nucleus("Alpha Particles with Half of original velocity")
```




```
scattering_angles = []
delta_time = 1.0e-20
data_analysis()

sorted_scattering_angle = sorted(scattering_angles, reverse=True)

x_axis_angles = []
for i in np.arange(1,181):
    x_axis_angles.append(i)
```

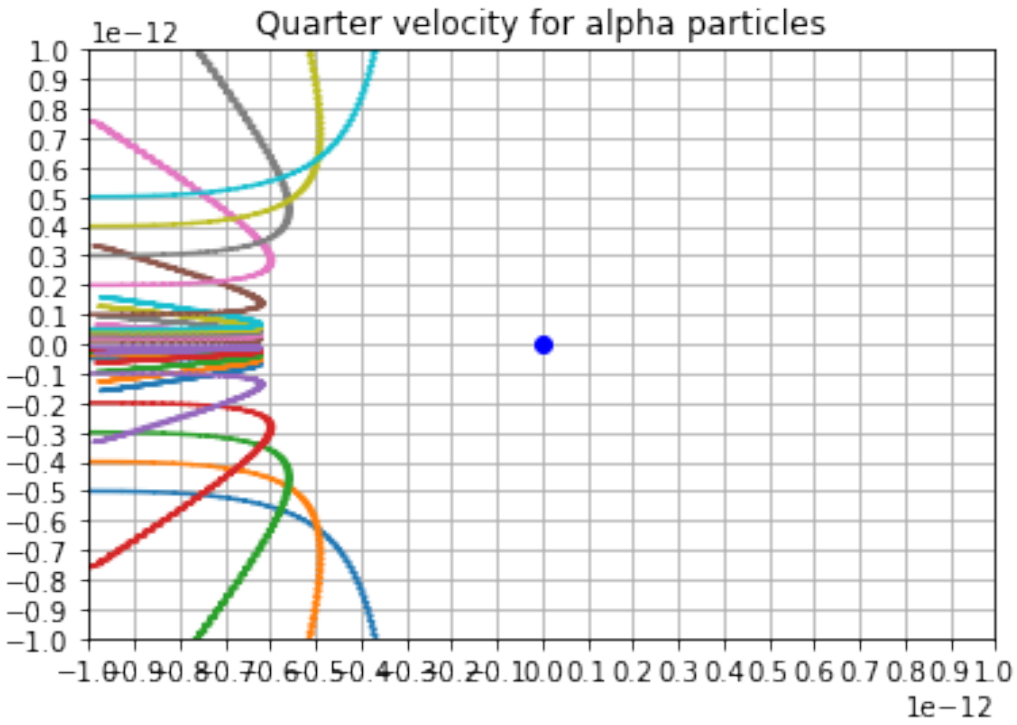
```
plt.hist(sorted_scattering_angle, bins=x_axis_angles)
plt.yscale("log")
plt.title("Number of Particles to Scattering angle at half velocity")
```

```
Text(0.5, 1.0, 'Number of Particles to Scattering angle at half
velocity')
```



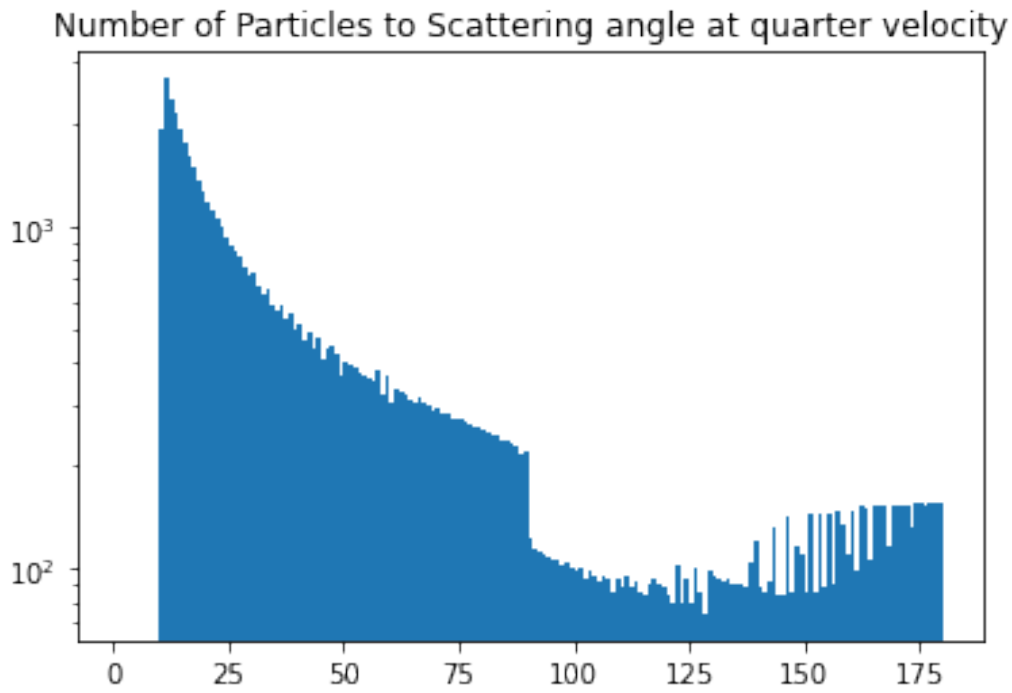
4.2.2 Quarter of the original speed

```
velocity_x0 = 0.25e7
fig, ax = plt.subplots()
single_nucleus("Quarter velocity for alpha particles")
```




```
scattering_angles = []
data_analysis()

sorted_scattering_angle = sorted(scattering_angles, reverse=True)

x_axis_angles = []
for i in np.arange(1,181):
    x_axis_angles.append(i)

plt.hist(sorted_scattering_angle, bins=x_axis_angles)
plt.yscale("log")
plt.title("Number of Particles to Scattering angle at quarter
velocity")
```

```
Text(0.5, 1.0, 'Number of Particles to Scattering angle at quarter
velocity')
```



4.2.3 Speed of light

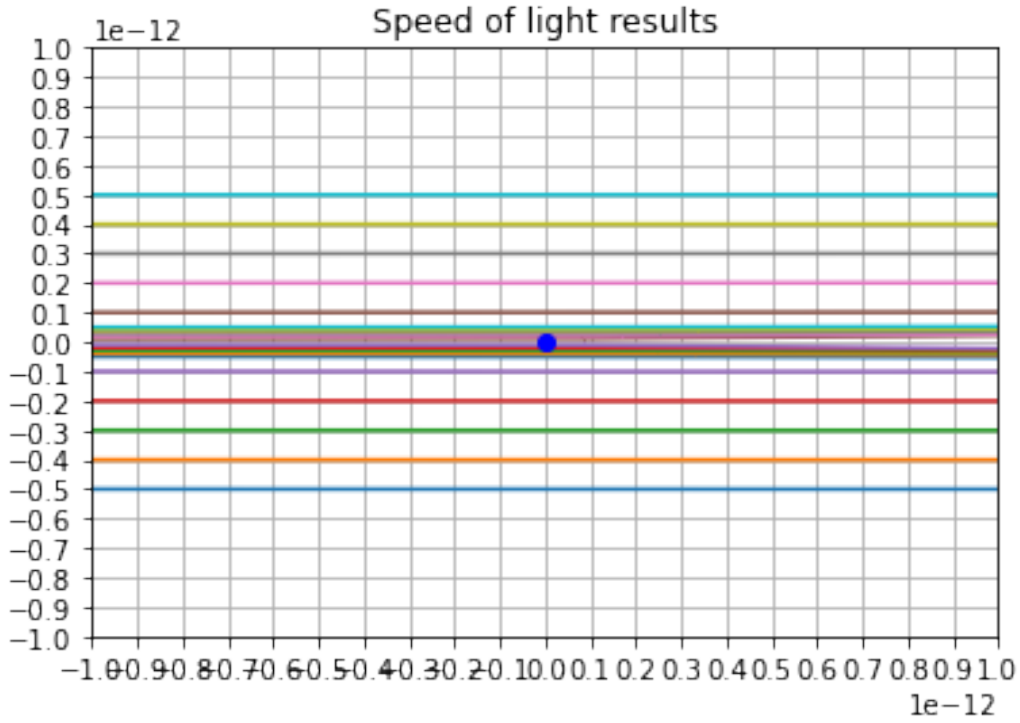
This serves as a proof to check that the scattering angle is negligble as the time for their interaction is so minute that the impact on the alpha particles trajectory should be non-existent.

```
# Create MatplotLib graph.

velocity_x0 = 299792458
# Create MatplotLib graph.
fig, ax = plt.subplots()

# Parameters and constants.
axis_range = 1e-12  # range of the axes
delta_time = 1.0e-24  # time step for kinematic equations

velocity_y0 = 0  # initial velocity of the alpha particle, along the
y-axis
proton_charge = 1.6e-19  # electric charge of 1 proton (in coulombs)
atomic_mass_constant = 1.661e-27  # atomic mass constant
x0 = -axis_range  # initial x coordinate of alpha particles
coulombs_constant = 1 / (4 * math.pi * 8.85e-12)  # Coulomb's constant
alpha_particle_mass = 4 * atomic_mass_constant  # alpha particle's
(helium nucleus) atomic mass
gold_nucleus_charge = 79 * proton_charge  # electric charge of a gold
nucleus (in coulombs)
alpha_particle_charge = 2 * proton_charge  # electric charge of an
alpha particle (in coulombs)
```

```
# Calculation of the Coulomb force acceleration.
def get_coulomb_acceleration(nucleus1_charge, nucleus2_charge, r1, r2,
mass):
    return coulombs_constant * nucleus1_charge * nucleus2_charge /
(math.sqrt(r1 ** 2 + r2 ** 2)) ** 2 / mass


def single_nucleus(plot_title):
    params = []
    # The range of alpha particles y coordinates to fire at nucleus.
Values can be changed.
    for i in range(-5, 6, 1):
        if i != 0:
            params.append(i*1e-14)
    for i in range(-50, 51, 10):
        if i != 0:
            params.append(i*1e-14)

    # Calculating the course of the trajectories of molecules.
    # x0 is the starting x coordinate of alpha particles, and y0 the
starting y coordinate.
    for y0 in params:
        # List of x coordinates the alpha particle has travelled
through.
        x_coordinates = [x0]
        # List of y coordinates the alpha particle has travelled
through.
        y_coordinates = [y0]
        # Calculate next x coordinate after time step (h) passes using
kinematic equation.
        # x(t+Δt)=x(t)+v(t)Δt
        x = x0 + delta_time * velocity_x0
        y = y0 + delta_time * velocity_y0

        # v_x = initial velocity + acceleration * interval
        velocity_x = velocity_x0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * x / math.sqrt(x ** 2 + y ** 2) *
delta_time
        velocity_y = velocity_y0 +
get_coulomb_acceleration(gold_nucleus_charge, alpha_particle_charge,
x, y, alpha_particle_mass) * y / math.sqrt(x ** 2 + y ** 2) *
delta_time

        # Save previous velocities in x and y axes.
        prev_velocity_x = velocity_x0
        prev_velocity_y = velocity_y0
```

```
        # Save previous positions for x and y axes.
        prev_x = x0
        prev_y = y0

        # We record new positions and velocities as long as the alpha
particle is within range.
        while np.isfinite(y) and -axis_range < x < axis_range:
            # Add current x and y positions to the lists recording the
histories of where the alpha particle has been.
            x_coordinates.append(x)
            y_coordinates.append(y)

            constAx = get_coulomb_acceleration(gold_nucleus_charge,
alpha_particle_charge, x, y, alpha_particle_mass) * x / math.sqrt(x **
2 + y ** 2)
            constAy = get_coulomb_acceleration(gold_nucleus_charge,
alpha_particle_charge, x, y, alpha_particle_mass) * y / math.sqrt(x **
2 + y ** 2)

            # Update position values.
            # x(t+Δt)=x(t)+v(t)Δt
            temp = x
            x = prev_x + prev_velocity_x * delta_time
            prev_x = temp

            temp = y
            y = prev_y + prev_velocity_y * delta_time
            prev_y = temp

            # Updating the velocity of particles moving along the
axis.
            temp = velocity_x
            velocity_x = prev_velocity_x + constAx * delta_time
            prev_velocity_x = temp

            temp = velocity_y
            velocity_y = prev_velocity_y + constAy * delta_time
            prev_velocity_y = temp

        # Title of the graph.
        plt.title(plot_title)
        # Generating points on the x-axis every 0.1 point.
        ax.xaxis.set_major_locator(ticker.MultipleLocator(axis_range /
10))
        # generating points on the y-axis every 0.1 point.
        ax.yaxis.set_major_locator(ticker.MultipleLocator(axis_range /
10))
        # Display grid on the graph.
```

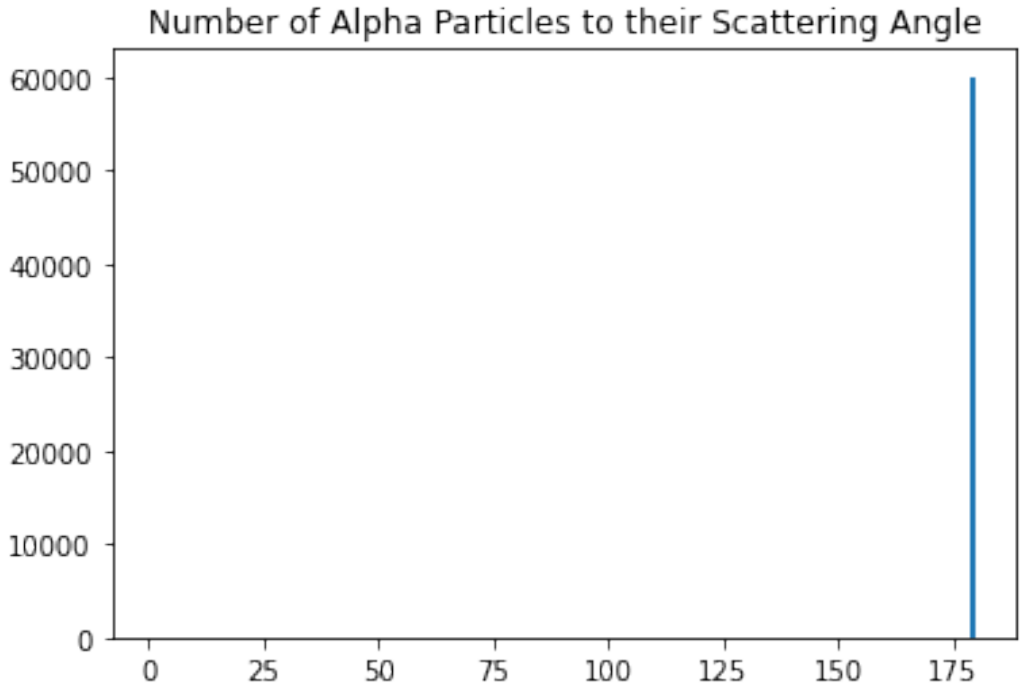
```
        plt.grid(True)
        # Display trajectories of alpha particles.
        ax.plot(x_coordinates, y_coordinates)

    # Displaying a gold atom in the centre of the graph.
    ax.plot(0, 0, "bo")
    # X-axis scale.
    plt.xlim(-axis_range, axis_range)
    # Y-axis scale.
    plt.ylim(-axis_range, axis_range)



single_nucleus("Speed of light results")
```




```
velocity_x0 = 299792458  # initial velocity of the alpha particle,
along the x-axis
delta_time = 1.0e-20
scattering_angles = []

data_analysis()

sorted_scattering_angle = sorted(scattering_angles, reverse=False)

numbers = []
for i in np.arange(1, 181):
```

```
  numbers.append(i)

plt.yscale("log")

plt.hist(sorted_scattering_angle, bins=numbers)
plt.title("Number of Alpha Particles to their Scattering Angle")
```

```
Text(0.5, 1.0, 'Number of Alpha Particles to their Scattering Angle')
```



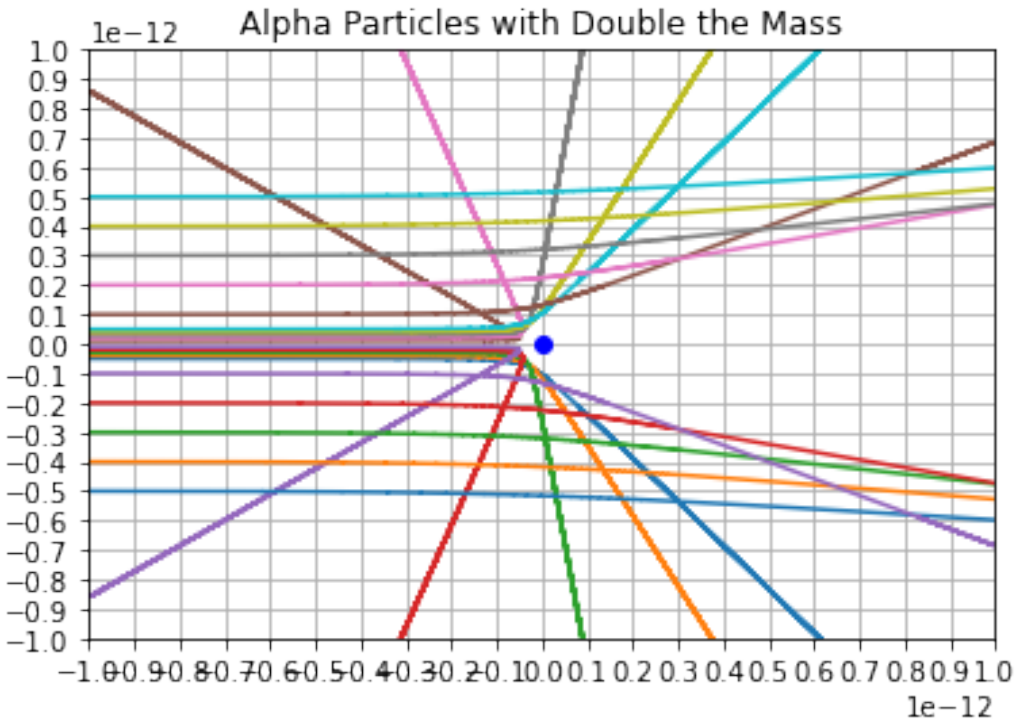

```
sorted_scattering_angle[-1]
```

```
4.299961369126907
```

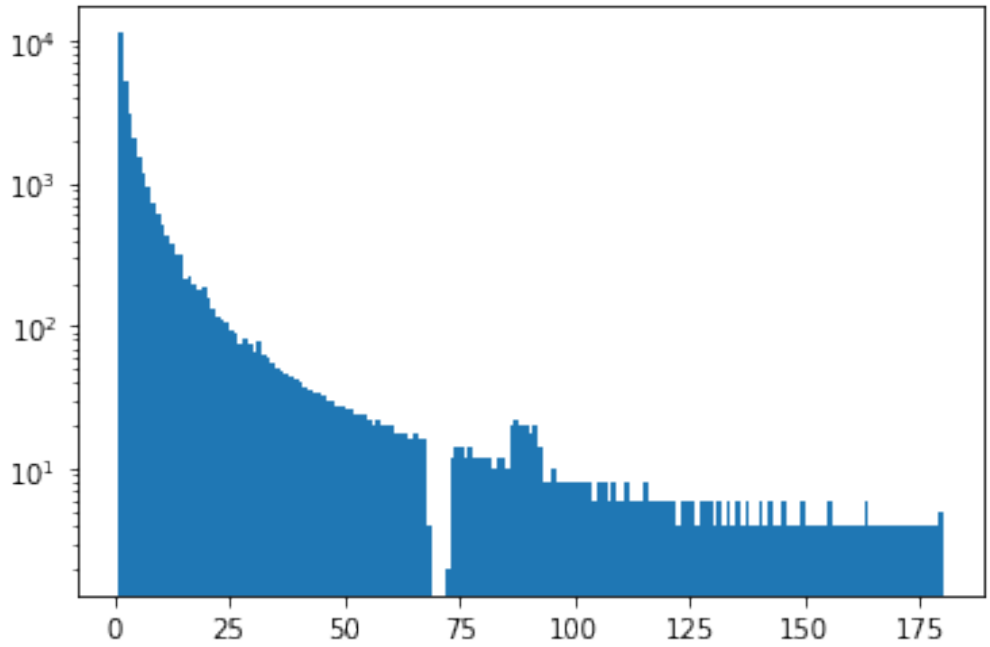
4.3 Altered alpha particle mass

4.3.1 Two times the original mass

```
velocity_x0 = 1.0e7
delta_time = 1e-23
alpha_particle_mass = 8 * atomic_mass_constant
fig, ax = plt.subplots()
def get_coulomb_acceleration(nucleus1_charge, nucleus2_charge, r1, r2,
mass):
    return coulombs_constant * nucleus1_charge * nucleus2_charge /
(math.sqrt(r1 ** 2 + r2 ** 2)) ** 2 / mass
single_nucleus("Alpha Particles with Double the Mass")
```

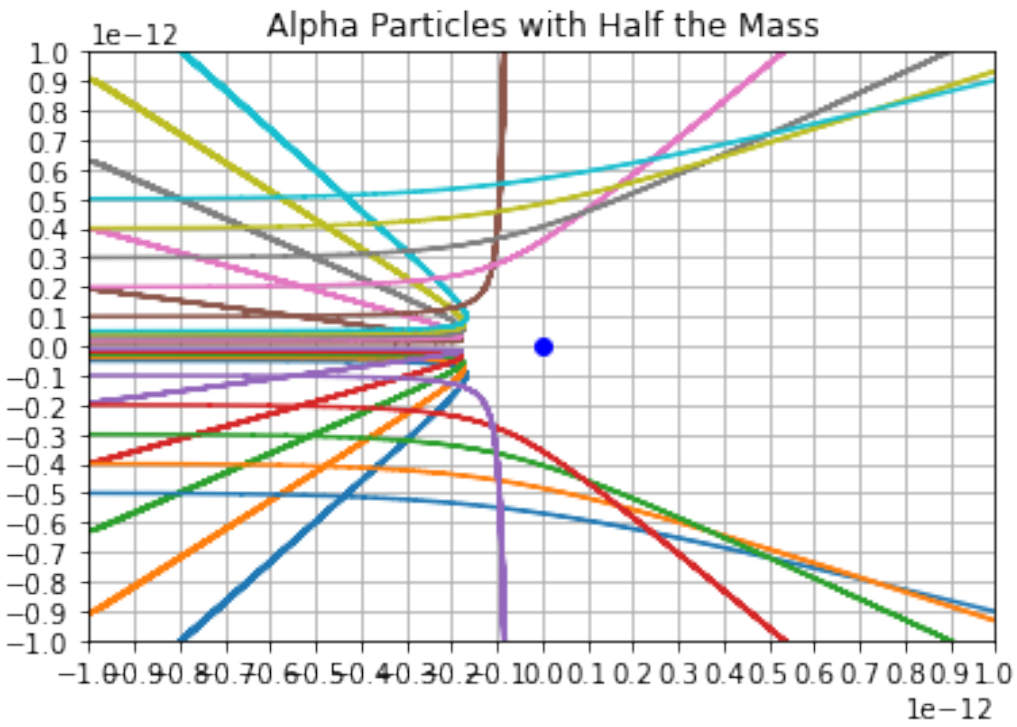



```
scattering_angles = []
delta_time = 1.0e-21  # time step for kinematic equations

data_analysis()

sorted_scattering_angle = sorted(scattering_angles, reverse=True)

x_axis_angles = []
for i in np.arange(1,181):
    x_axis_angles.append(i)

plt.hist(sorted_scattering_angle, bins=x_axis_angles)
plt.yscale("log")
```



4.3.2 Half of the original mass

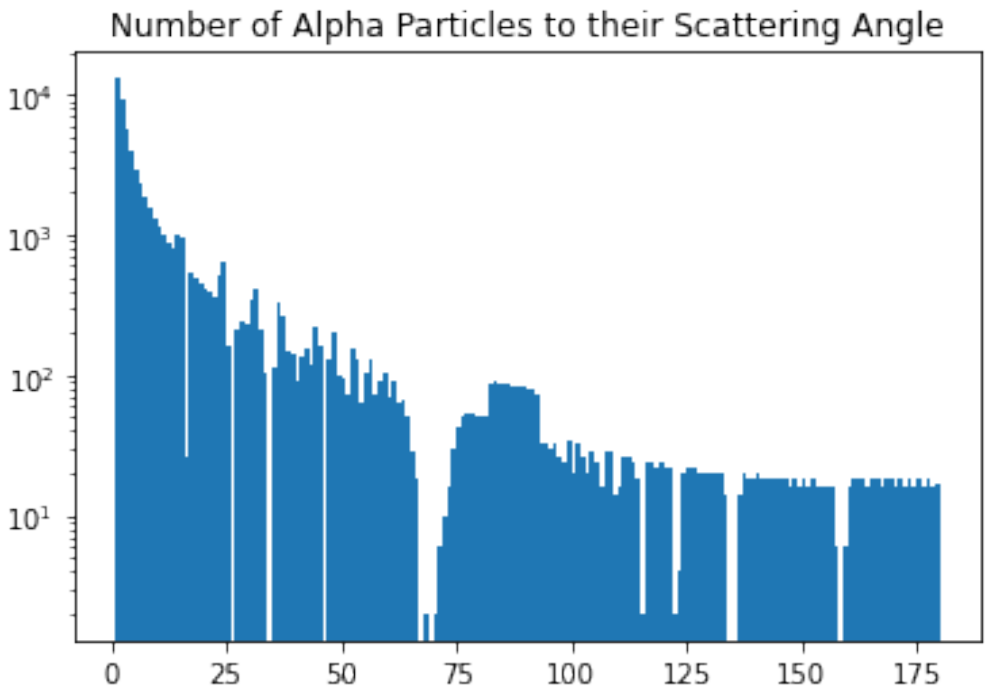
```
velocity_x0 = 1.0e7
delta_time = 1e-22
alpha_particle_mass = 2 * atomic_mass_constant
fig, ax = plt.subplots()
def get_coulomb_acceleration(nucleus1_charge, nucleus2_charge, r1, r2,
mass):
    return coulombs_constant * nucleus1_charge * nucleus2_charge /
(math.sqrt(r1 ** 2 + r2 ** 2)) ** 2 / mass
single_nucleus("Alpha Particles with Half the Mass")
```



With half the mass, the distance of closest approach is further from the nuceleus than before. This can be seen by the deflection angle plot below.

```
delta_time = 1.0e-20
scattering_angles = []

data_analysis()

sorted_scattering_angle = sorted(scattering_angles, reverse=False)

numbers = []
for i in np.arange(1, 181):
  numbers.append(i)

plt.hist(sorted_scattering_angle, bins=numbers)
plt.yscale("log")
plt.title("Number of Alpha Particles to their Scattering Angle")
```

```
Text(0.5, 1.0, 'Number of Alpha Particles to their Scattering Angle')
```



It can be seen that the general trend has shiften more of the particles to the higher scattering angle.

5. Conclusion

5.1 Limitations of the model

While we have managed to build several successful models in simulating Rutherford's scattering, both in 2D and 3D, we have faced limitations in computing power and time. The variable Δt - the time step - dictates how accurate the model is; as it gets smaller, the gap between iterations decreases, and hence the positions and velocities of alpha particles are calculated more often, resulting in better outputs. However, by decreasing Δt, the code executing time increases exponentially. For instance, when Δt is decreased from 10^{-23} to 10^{-24}, the value is decreased by 10 times, thus the executing time also increases by about 10 times.

Throughout this project, we ran all the code on the Google Colab CPU-only VM (Virtual Machine): dual-core Intel Xeon at 2.30GHz, 12GB RAM, 25GB disk space, with no GPU. Therefore, we often could not run code with as low a Δt as we would like to have, since we did not want to wait for more than 2 hours for the code to run. Having a high Δt would result in jaggedness and inaccuracies on the graph.

5.2 Further development

To run code more quickly and efficiently, we could utilise better computers with GPU acceleration. One option is to rent a VM on a cloud computing service, such as AWS or Google Cloud Platform. Cloud VMs offer computers with more CPU cores, higher CPU clock rate, higher RAM, and GPU acceleration. Cloud computing services charge by the hour; depending on the configuration, CPU-only machines cost less than $1/hour, while GPU-machines cost slightly over $1/hour. This is a very affordable option, especially if we only need to run the code once.

Moreover, to achieve the most accurate results in the Data Analysis section, we need to fire as many alpha particles from as wide a range as possible. However, as explained in the previous paragraph, there are limitations in computational power and time; with the help of a Cloud VM, we would be able to fire more alpha particles to achieve accurate plots. Furthermore, to organise the data, we could use an SQL database for easy and quick access to the data produced by the models.

References

Rutherford Scattering. (n.d.). History of Rutherford Gold Foil Experiment. Retrieved 1 May 2022, from http://hyperphysics.phy-astr.gsu.edu/hbase/Nuclear/rutsca2.html

LXXIX. The scattering of Î± and Î2 particles by matter and the structure of the atom. (2009, April 21). Taylor & Francis. Retrieved 1 May 2022, from https://www.tandfonline.com/doi/abs/10.1080/14786440508637080

Geiger & Marsden. (1913, January 1). Geiger-Marsden Experiment. Geiger-Marsden. Retrieved 29 April 2022, from http://web.ihep.su/dbserv/compas/src/geiger13/eng.pdf

Rutherford and the nucleus - Models of the atom - AQA - GCSE Physics (Single Science) Revision - AQA. (n.d.). BBC Bitesize. Retrieved 25 April 2022, from https://www.bbc.co.uk/bitesize/guides/zxkxfcw/revision/2

Cone, M. C. (n.d.). Basic Syntax | Markdown Guide. Markdown Guide. Retrieved 26 April 2022, from https://www.markdownguide.org/basic-syntax/

Gibbs, K. G. (n.d.). Distance of closest approach of an alpha particle to a gold nucleus. School Physics. Retrieved 3 April 2022, from https://www.schoolphysics.co.uk/age16-19/Atomic%20physics/Atomic%20structure%20and%20ions/text/Alpha_particle_scattering/index.html

Rutherford Scattering. (n.d.-b). Phys3002. Retrieved 2 April 2022, from http://www.personal.soton.ac.uk/ab1u06/teaching/phys3002/course/02_rutherford.pdf

Wikipedia contributors. (n.d.). Geiger–Marsden experiments. Wikipedia. Retrieved 28 April 2022, from https://en.wikipedia.org/wiki/Geiger%E2%80%93Marsden_experiments

Wikipedia contributors. (n.d.-a). Ernest Rutherford. Wikipedia. Retrieved 28 April 2022, from https://en.wikipedia.org/wiki/Ernest_Rutherford

The Matplotlib development team. (n.d.). Matplotlib documentation — Matplotlib 3.5.2 documentation. Matplotlib. Retrieved 27 April 2022, from https://matplotlib.org/stable/index.html

NumPy documentation — NumPy v1.22 Manual. (n.d.). NumPy. Retrieved 30 April 2022, from https://numpy.org/doc/stable/