Numerically Modelling Rutherford Scattering

Computational Physics Prize 2022

Author:

Koza Kurumlu

Table of Contents

1. Introduction

This project aims to numerically model Rutherford scattering and investigates how the electric field of a nucleus can affect the distribution of scattered α-particles at different angles.

1.1 Dependencies and distribution

The dependencies used are NumPy, Matplotlib and Python's Random library. The code is written in Python 3.9.

1.1.1 Files

- rutherford_scatterring.py includes the complete model with a couple of complementary functions, which will be explained in chapter 4 'Usage'.

- rutherford_scattering.pdf, this document, includes all the code, all the graphs and all variations in testing. rutherford_scattering.docx was used to create this document.

1.1.2 Experimenting

The recommended way to use the code is to import rutherford.py and run the function required in another .py file. Import as shown in the code below:

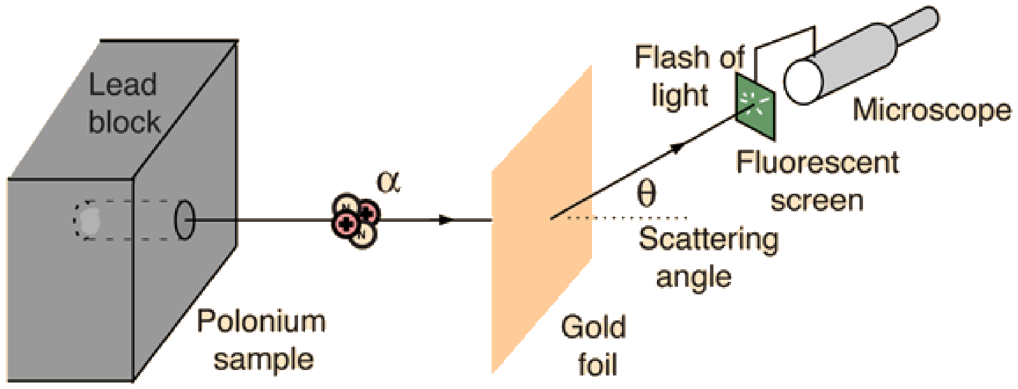
```python
from rutherford_scattering import *
```

An alternative way is to create a notebook and copy functions from the source code.

1.2 Background: Rutherford Scattering

Rutherford scattering is an experiment undertaken to investigate the atomic model. It involves the scattering of particles (in this case α-particles), due to electrical interactions from the nucleus of an atom. Then the angle at which these particles are scattered is recorded. Figure 1.A below illustrates the experiment.

Figure 1.A



Source: http://hyperphysics.phy-astr.gsu.edu/hbase/rutsca.html

1.2.1 Rutherford's conclusions

Rutherford believed that most of the α-alpha particles fired would bounce back, due to the plum-pudding atomic model. Contrary to his hypothesis, most of the alpha particles passed straight through the gold foil and were not scattered, and only some were deflected. Due to this result, Rutherford reached the following conclusions:

- Most of matter is empty.
- There is a small, concentrated, positive charge within the atom called a nucleus. The nucleus contains most of the atom's matter.
- There are electrons that orbit the nucleus. The nucleus is positive, therefore there must be a negative charge present to stabilise the atom. If the electrons were too close to the nucleus, the α-particles would not scatter.

1.2.2 Issues with the experiment

The experiment contains a few issues:

1. The gold foil has a thickness (roughly 40 atoms). Therefore, there is a possibility that the α-particles can be scattered more than once, which would affect the scattering pattern.
2. Orbiting electrons don't lose energy because of radiation, which would cause them to collapse into the nucleus.
3. The strong nuclear force is present in both the gold nucleus and the alpha particles, how much of a role does it play in the scattering pattern?

1.2.3 Addressing the issues

1. For the first problem, we could assume that the gold was not thick enough to hinder results, however, we will still test the experiment with one gold atom to eliminate that possibility.
2. The second issue is irrelevant in this project, as it was later solved by Niels Bohr with quantum physics.
3. For the final issue we can show that the electromagnetic force is the major reason for α-scattering, as we do not integrate the strong nuclear force within the code.

Addressing issue 1 & 3 will be a part of but not the whole of our investigation of the relationship between the electric field and the scattering angle.

1.3 Summary of results

Since we are computationally modelling this experiment, we can isolate specific variables and analyse their direct results. After running the model in various scenarios and qualitatively comparing it to Rutherford's scattering formula (see 5.1 'Rutherford's formula'), we reach the following results:

- The strong nuclear force has an insignificant effect on the scattering of α-particles.
- Decreasing the kinetic energy of an α-particle slightly increases the average scattering angles.
- There can be an exception to Rutherford's formula when an α-particle's energy exceeds a certain point (roughly above 25 MeV).
- Using a different metal, such as copper, does affect the scattering pattern. Decreasing the proton number, slightly decreases the scattering angle.

1.4 Assumptions

In the modelling of the experiment all the following assumptions hold:

- Both the α-particles and the gold nuclei are treated as point particles.
- The simulation is conducted with classical mechanics.
- The nucleus is sufficiently massive compared with the mass of the α-particle that the nuclear recoil is neglected.
- The experiment is conducted in a vacuum so the only factor affecting the α-particle's velocity is the electric force between it and the gold nuclei.
- α-particles are sent from y values above 0. We exclude negative y values for the start location, as the experiment is symmetrical for both $y > 0$ and $y < 0$.

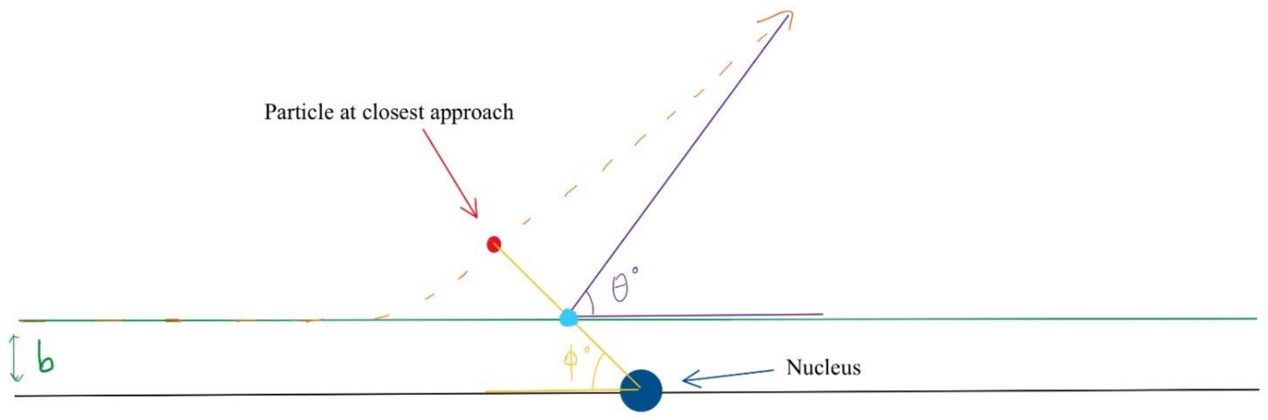
2. Mathematical details

2.1 Terminology

Below is the terminology we will use within this document, figure 2.A illustrates the scenario.

- Let b be the impact parameter: the perpendicular distance to the closest approach of the α-particle if it were undeflected.
- Let φ° be the angle from the alpha particle (at any timestep), to the gold nucleus, to the x axis, where the gold nucleus at the origin.
- Let r be the distance between the nucleus and the projectile.
- The intersection point is the point at which the line between the closest approach of the projectile and the nucleus, also known as the symmetry plane, intersects with b. It is highlighted in turquoise in figure 2.A.
- Let θ° be the scattering angle: The last detected point of the α-particle, to the intersection point, to the horizontal line $y = b$.

Figure 2.A



2.2 Calculating the scattering angle

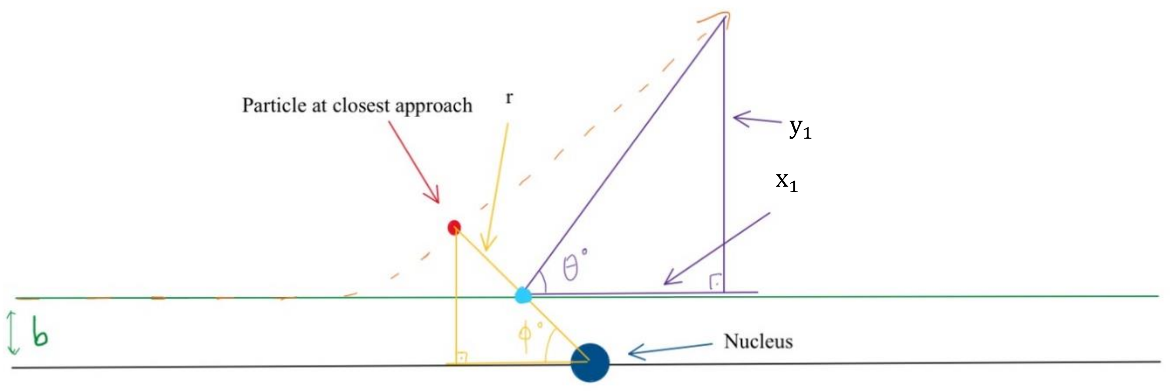
We calculate the scattering angle once the simulation has completed the trajectory of an α-particle. The scattering angle is defined above (see section 2.1 'Terminology').

Finding the intersection point

1. Find the closest approach: the trajectory is hyperbolic; therefore, the closest approach is just the point before the r value starts to increase.
2. Then we calculate the φ° of the closest approach: $\varphi° = \tan^{-1}(y/x)$, where (x, y) are the coordinates of the closest approach.

3. Then to find the intersection point we create a right-angled triangle from the intersection point to the gold nucleus (origin), to the x axis. The y coordinate of the intersection point is simply b. To find the x coordinate: $x = \frac{b}{\tan(\varphi°)}$, therefore we get the intersection coordinates (x, y).
4. We then store the coordinates of the intersection point, however if $x < 0$, then $x = -x$, where x is the component of the intersection point's coordinates. This is necessary as we use trigonometric functions later, and x must be a non-negative value.

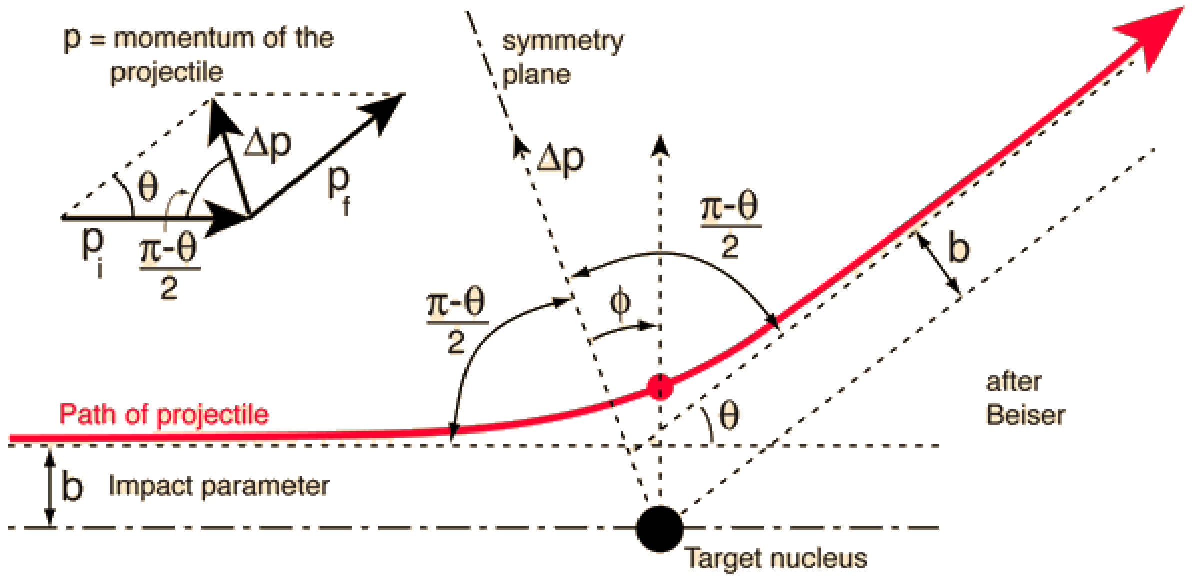
Figure 2.A



Scattering angle

Now we can find the distance between the intersection point and the final location of the α-particle: (x_1, y_1), see figure 2.A above. With this we calculate the scattering angle: $\theta° = \tan^{-1}(y_1/x_1)$. The final scattering angle is displayed in figure 2.B below.

Figure 2.B



Source: http://hyperphysics.phy-astr.gsu.edu/hbase/rutsca.html

The code for the steps described is below (note this is not the whole scattering angle function, we will see the full script in section 4 'Usage'):

```
#calculates angle psi of the closest approach of alpha particle - returns radians
psi_of_closest_approach = np.arctan(closest_y/closest_x)


#finding the point where the line between closest approach and target nuclei intercepts
with b
intersection_y = b
#np.tan() takes in rad
intersection_x = intersection_y/np.tan(psi_of_closest_approach)
intersection_location = np.array([intersection_x, intersection_y])

#since we can't divide by 0, if it bounces straight back we hardcode the scattering
angle to 0
if end_loc_y == 0:
  scattering_angle = 180
else:
  #finds difference between intersection point and final alpha location
  difference_x = end_loc_x - intersection_location[0]
  difference_y = end_loc_y - intersection_location[1]

  #calculates scattering angle
  scattering_angle = np.arctan(difference_y/difference_x)

  #converts to degrees from radians
  scattering_angle = np.rad2deg(scattering_angle)
```

2.3 Calculating change in velocity

We can calculate the force acting on the α-particle at each time step due to the electric field, using coulomb's law:

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

Where k is coulomb's constant and q_1 & q_2 are the charges of the target nucleus and α-particle in coulombs, respectively. Let Z be the atomic number of the target

$$k = 8.99 \times 10^9$$

$$q_1 = Z(1.6 \times 10^{-19})C$$

$$q_2 = 2(1.6 \times 10^{-19})\mathrm{C}$$

Once we have the force, we can derive the change in velocity as follows:

$$\vec{F} = m\vec{a}$$

$$\vec{a} = \frac{\vec{F}}{m}$$

Since we know the force, via coulombs law, and the mass of an α-particle, we can calculate the acceleration.

$$\vec{a} = \frac{\Delta\vec{v}}{\Delta t}$$

$$\therefore \Delta\vec{v} = \frac{\vec{F}}{m} \times \Delta t$$

Note that the value we get from this procedure is a vector, and we need the (x, y) components of the change in velocity, which we can achieve using trigonometry. The code for calculating change in velocity is below:

```
#calculating force between two particle in Newtons
force = alpha.coloumb_force(target.charge, distance)


#calculating the velocity acting on alpha particle due to force - this is the
hypotenuse
acting_velocity = TIME_STEP * (force / alpha.mass)



#calculating x,y components of velocity
velocity_x = acting_velocity * np.cos(angle_psi)


#cheking if alpha particle is behind or in front of target nuclei, because that
affects if acting x veloxity is '+' or '-'
if alpha.location[0] < 0:
    velocity_x = velocity_x * -1

velocity_y = acting_velocity * np.sin(angle_psi)
acting_velocity = np.array([velocity_x, velocity_y])
```

We need to pay specific attention to the signs, because if the α-particle's x component is less than 0, then the change in velocity from the electric force will be negative (this is seen in the code above).

3. Overview of model

Our simulation uses a fixed time step and is summarized very briefly below:

1. Run simulation steps until α-particle reaches x limit of 2e-14m
2. Calculate the scattering angle
3. Repeat for next α-particle

3.1 Flow

Initialisation

The α-particles are initialised after the previous alpha particle has completed its simulation and its scattering angle has been recorded. At any one time there is only one alpha particle being simulated.

Each particle is initialised from the x location -2e-14m and a random y value between 0 and a max value for the range chosen by the user. The y value randomly chosen is called the impact parameter.

Each simulation step

Then following procedure takes place:

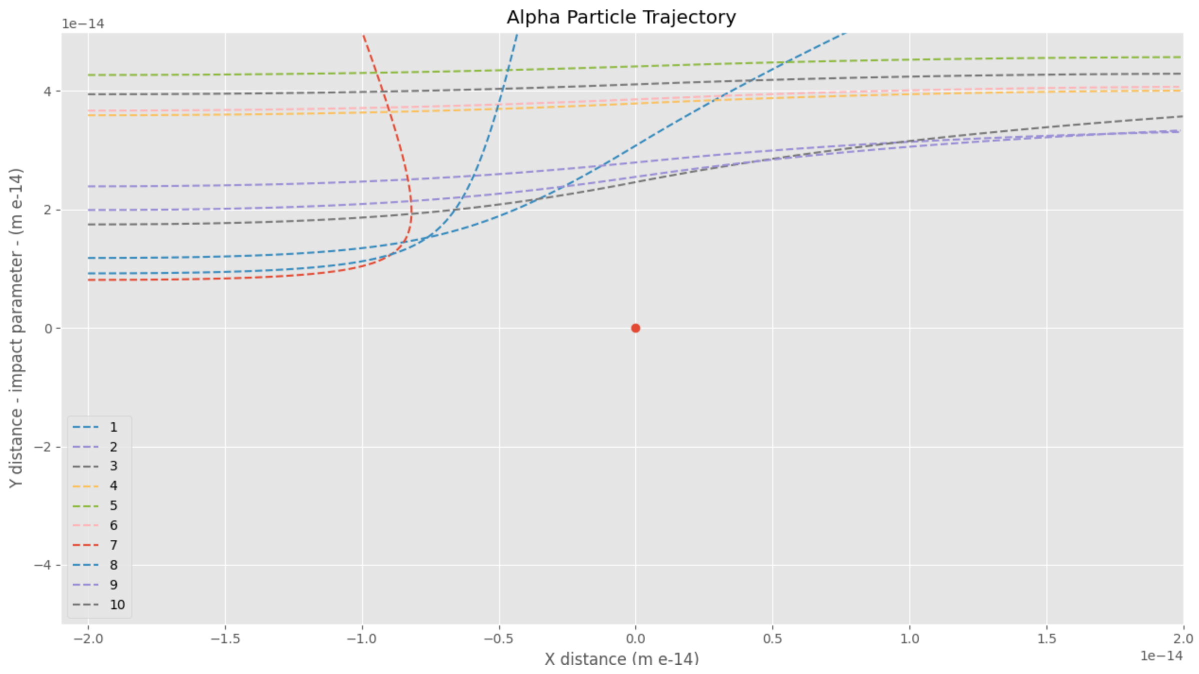
1. Calculate r distance between α-particle and gold nucleus
2. Check if r is less than shortest distance so far, if so, replace shortest distance with r
3. Calculate force acting on alpha particle with coulomb's law
4. Find the change in velocity (vector) within the timestep, from the force acting on particle
5. Find the value for $\varphi°$
6. Use trigonometry and find (x, y) components of change in velocity
7. Check if the α-particle is behind the gold nucleus, if so, the x component of the change in velocity is negative
8. Add change its velocity to the previous velocity
9. Move particle along new velocity vector for the allocated time step
10. Repeat steps 1-9 until α-particle reached the x limit, then calculate and store scattering angle as shown previously (see '2.2 Calculating scattering angle').

The code for the steps are above is in section 4 'Usage'.

Once simulation is completed

Once we repeat the steps 1-10 for each particle, we plot either a trajectory graph for every particle, or we plot the number of particles against the angle they were scattered at, depending on which function was called by the user. Below, figure 3.B, we see the trajectory of 10 different α-particles, produced by following the steps above:

Figure 3.A



3.2 Parameters

The simulation takes in the following parameters, all the parameters have a default:

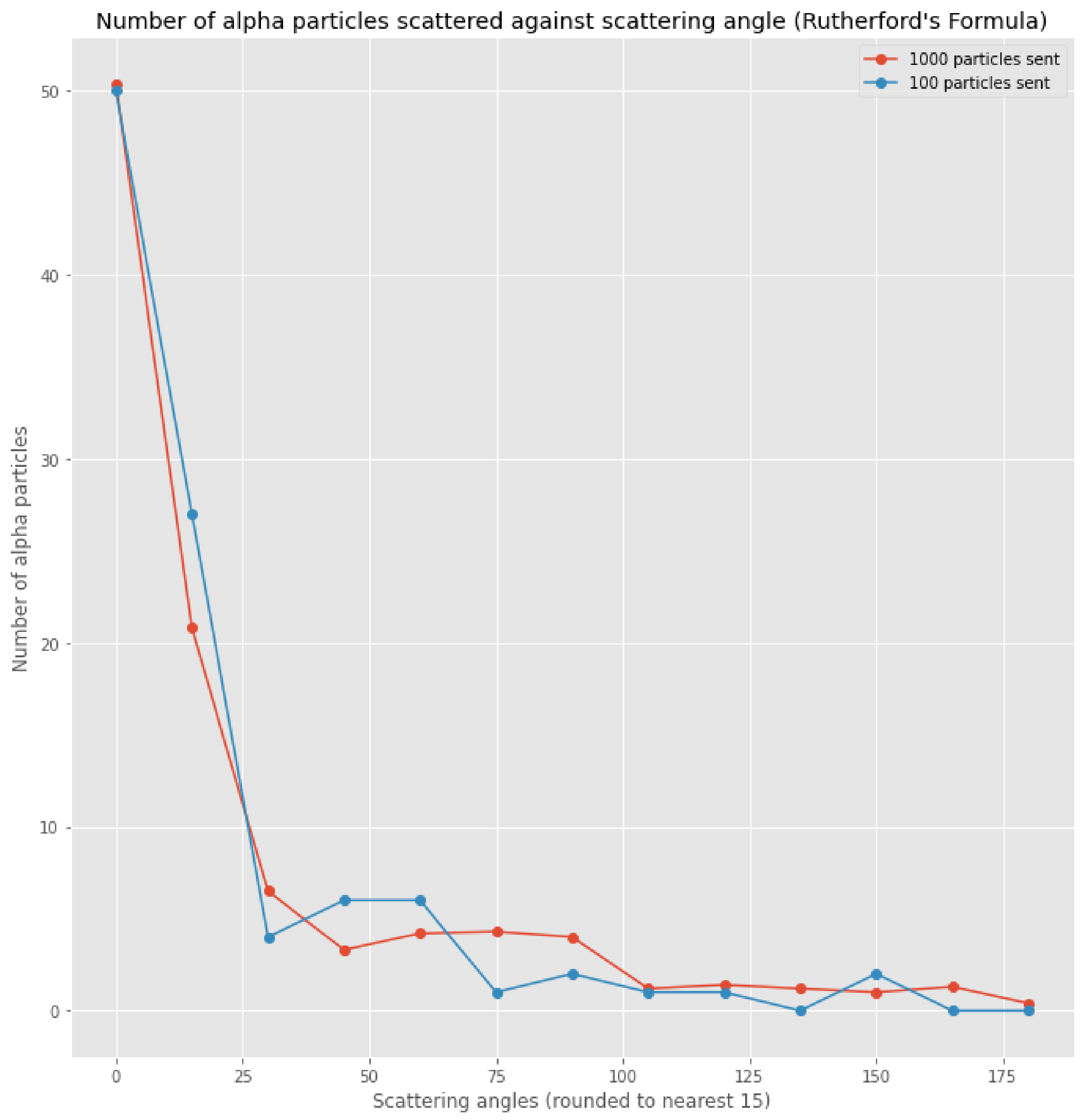
- Number of α-particles being fired: default set at 10 when plotting the trajectories, but 1000 when plotting the number of α-particles scattering against the scattering angle.
- Max b value: default set at 1e-14m when plotting trajectories, but 1e-13m when plotting the number of α-particles scattering against the scattering angle.
- Target's proton number: default set at 79 (gold).
- Kinetic energy of α-particles in MEV: default is set at 7.7 which is the energy of α-particles emitted from radium (the source Rutherford used).

3.3 Things not worth testing for

Varying the time step will change the results. The shorter the timestep the more realistic the model, however if it is too short then the computer struggles to run the simulation within a reasonable time. On other hand if we increase the time step, the displacement of the α-particle in each step increases too much that the particle is barely deflected; producing non-realistic results. The ideal time step I found is 1e-23 seconds.

Although the number of α-particles sent is a parameter, when increased, the ratio of α-particles scattered against the scattering angle stays the same. The parameter is there just to collect more data, and it is used when picking which graph to send. This is illustrated below in figure 3.B where we send 1000 particles and divide the count for each angle by 10, we also send 100 particles and plot them together. As we can see the line don't differ much.

Figure 3.B



As well as this, we only run the model with one atom rather than a lattice. This is done to make sure there is not a secondary scattering, as mentioned in section 1.2.3 ‘Addressing the issues’.

4. Usage

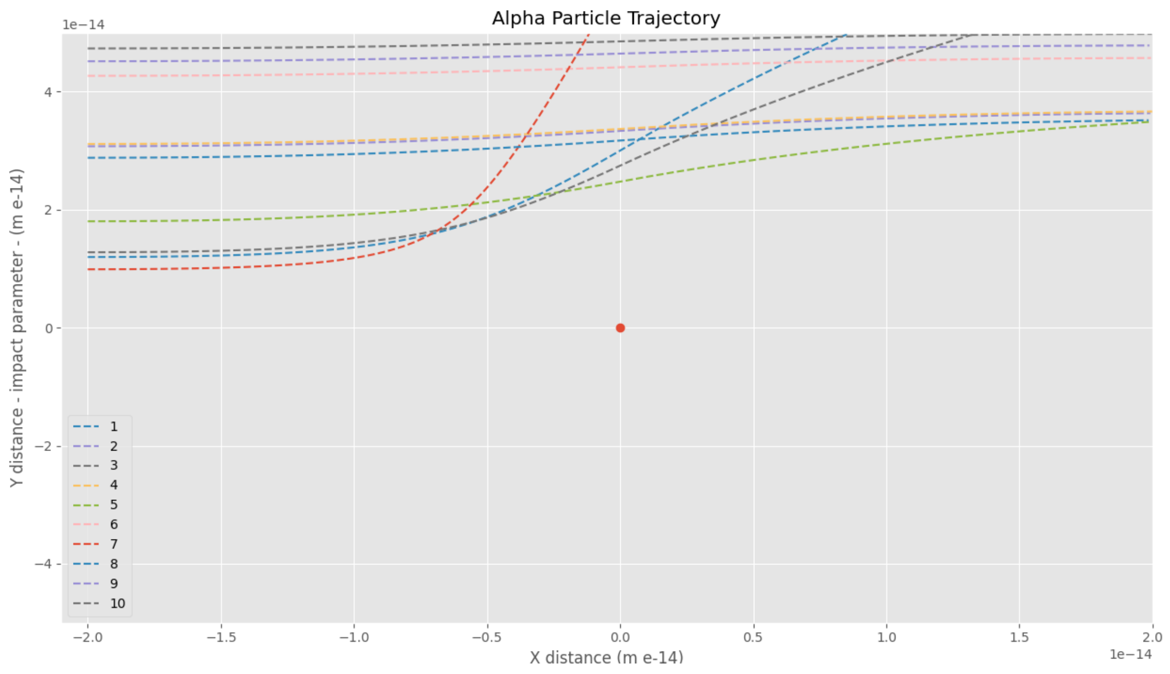
In order to use the functions provided within the code we first import it into another python file as shown below.

```
from rutherford_scattering import *
```

There are two main functions we can call which produces different graphs. The first one plots the trajectory of α-particles fired, as shown in figure 4.A. The default parameters of the function are displayed in the code below.

```
main_trajectory(number_of_alpha_particles=10,
                b_max=1e-14,
                target_proton_num=79,
                mev=7.7)
```

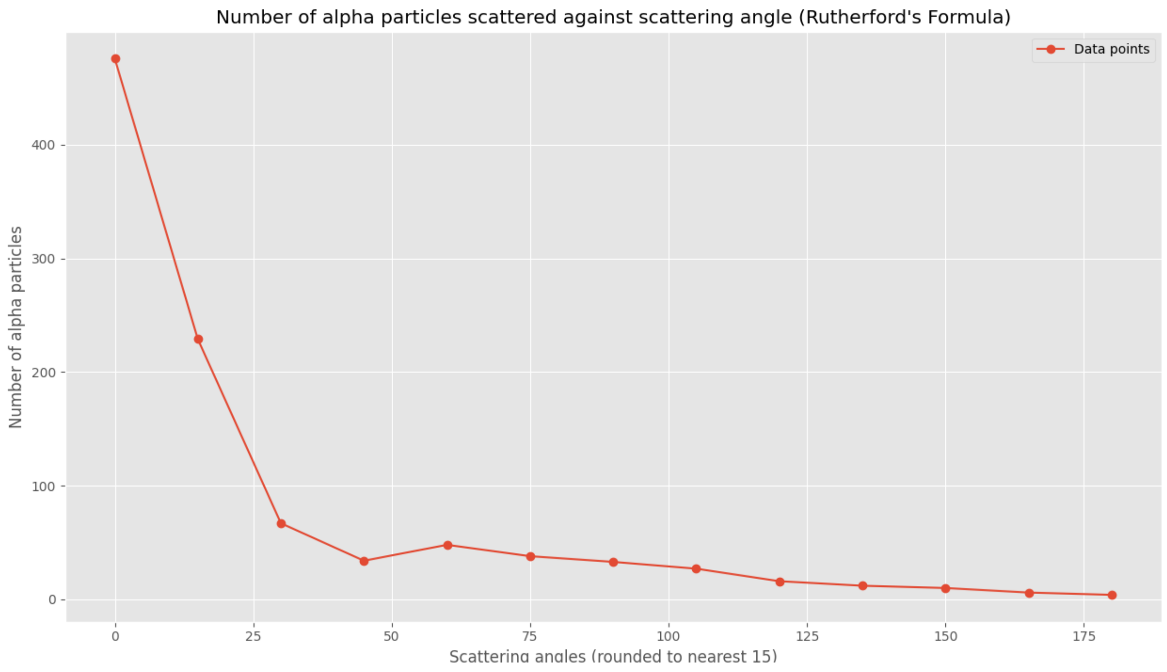
Figure 4.A



The second main function we can call will plot the number of α-particles scattered, against the angle they scattered at. Again the default parameter and the output is below (figure 4.B).

```
main_r_formula(number_of_alpha_particles=1000,
                b_max=1e-13,
                target_proton_num=79,
                mev=7.7)
```

Figure 4.B



Aside from the two main functions we have a complimentary function called compare. This function calls rutherford_r_formula twice and compares their graphs, however the parameters on each graph can be changed. The default parameters are below.

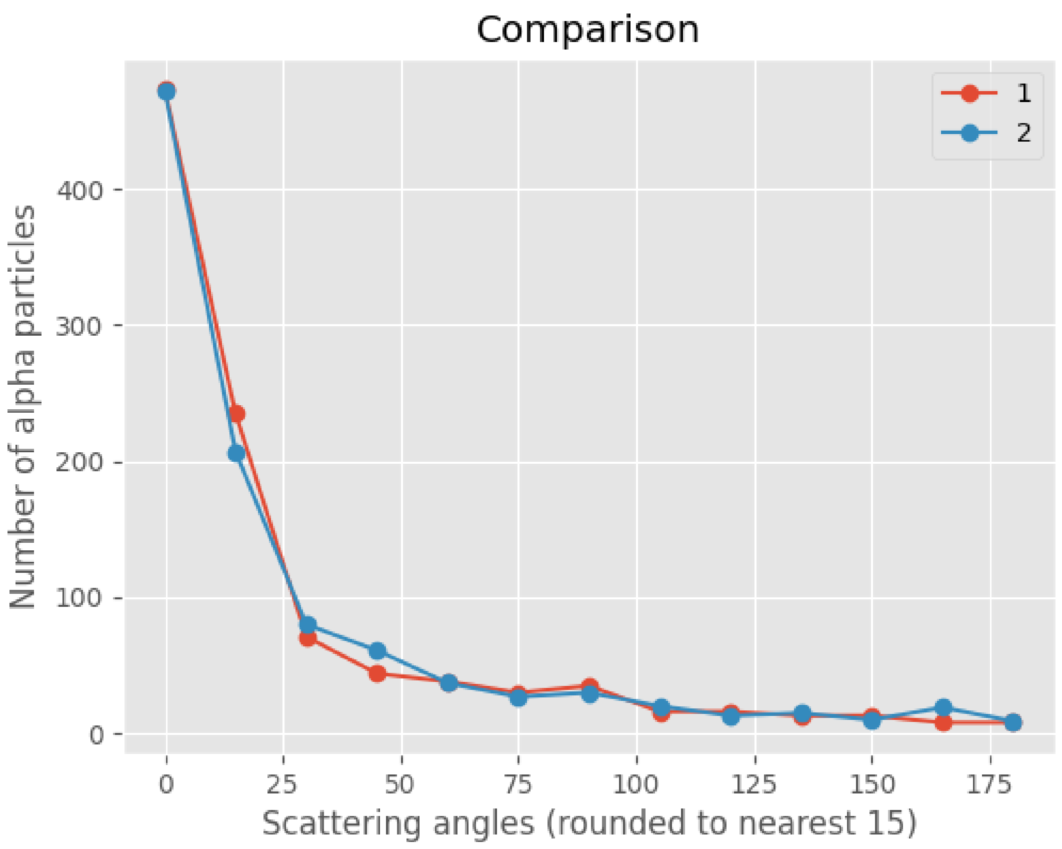
```
compare(number_of_alpha_particles=1000,
        b_max=1e-13,
        target_proton_num=79,
        mev=7.7,
        label = '1',

        number_of_alpha_particles1=1000,
        b_max1=1e-13,
        target_proton_num1=79,
        mev1=7.7,
        label1 ='2',

        title='Comparison')
```

The output is below (figure 4.C)

Figure 4.C

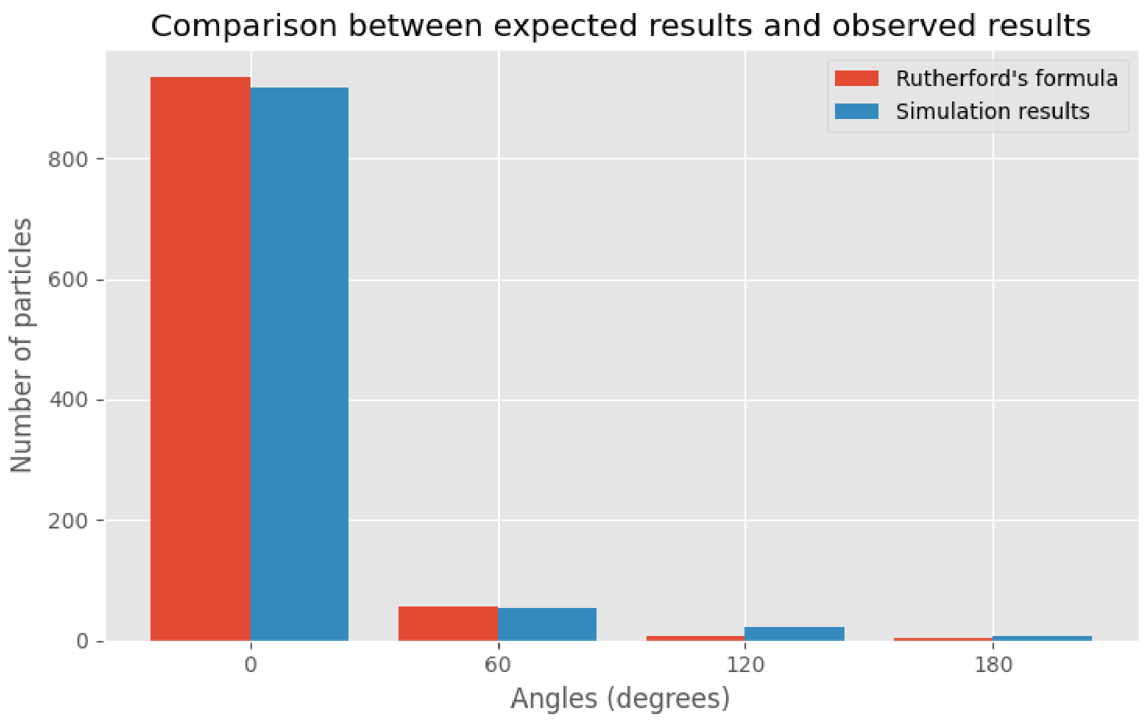


Our final complementary function is called test. This compares the expected results from Rutherford's formula and plots them against the output of the model. The default parameters are below:

```
test(number_of_alpha_particles=1000,
    target_proton_num=79,
    mev=7.7)
```

The results are below (figure 4.D):

Figure 4.D


Comparison between expected results and observed results
Rutherford's formula
Simulation results
Number of particles
0
200
400
600
800
0
60
120
180
Angles (degrees)

4.1 Source code and notes

First we import our dependencies.

```
import numpy as np
from numpy import pi
from matplotlib import pyplot as plt
import random
```

Then we define our particle class which will hold both the α-particle and the target nucleus. The class has a couple helper function for the α-particle.

```
class Particle():

    def __init__(self, mass, location, charge, velocity):
        self.mass = mass
        self.location = location
        self.charge = charge
        self.velocity = velocity

    #calculates distance from alpha particle to target nuclei (origin)
    def calculate_distance(self):
        return np.linalg.norm(self.location)

    def coloumb_force(self, target_particle_charge, r):

        #coulombs constant
        k = 8.99e9

        #calculating force in Newtons
        force = self.charge*target_particle_charge*k/r**2
        return force
```

Now we define our constants and initiate the target nucleus.

```
#time step in seconds
TIME_STEP = 1e-23

#b = 13e-15 # initialized later

#mass in kg
alpha_particle_mass = 6.6446573357e-27

#charge in coulombs
alpha_particle_charge = 2 * 1.6e-19

#no need
target_particle_mass = None
#197 * 1.6605390666e-27

target_particle_charge = 79 * 1.6e-19

#target nuclei at origin
target_location = np.array([0,0])
```

```
#it's at rest
target_velocity = np.array([0,0])

#intilializing the target
target = Particle(target_particle_mass, target_location, target_particle_charge, target_velocity)
```

This is the main simulation loop. We call this function to model the trajectory of an α-particle.

```
def simulation_loop(alpha):

  #stores total history of alpha particles movement
  history = []

  #stores closest distance reached by alpha particle to target nuclei
  closest_distance = 1

  #stores the x,y coordinates of that closest point
  closest_location = np.array([])
```

Here we set an x limit for the simulation to run within.

```
  while (alpha.location[0] >= -2e-14 and alpha.location[0] < 2e-14):

      #adds location of alpha particle to history
      history.append(alpha.location)

      distance = alpha.calculate_distance()
      if distance < closest_distance:
          closest_distance = distance
          closest_location = alpha.location



      #calculating psialpha particle - target nuclei - X axis
      #adding a '+' to x coordinate because it is negative and it must be positive to calculate angle
      angle_psi = np.arctan(alpha.location[1]/-alpha.location[0])


      #calculating force between two particle in Newtons
      force = alpha.coloumb_force(target.charge, distance)


      #calculating the velocity acting on alpha particle due to force - this is the hypoenuse
      acting_velocity = TIME_STEP * (force / alpha.mass)



      #calculating x,y components of velocity
      velocity_x = acting_velocity * np.cos(angle_psi)


      #cheking if alpha particle is behind or infront of target nuclei, because that affects if acting x
      veloxity is '+' or '-'
      if alpha.location[0] < 0:
          velocity_x = velocity_x * -1

      velocity_y = acting_velocity * np.sin(angle_psi)
```

```
        acting_velocity = np.array([velocity_x, velocity_y])

        #vector addition with acting velocity and initial velocity
        alpha.velocity = alpha.velocity + acting_velocity

        #calculating the movement of the alpha particle within the timestep for x,y components
        movement_x = alpha.velocity[0] * TIME_STEP
        movement_y = alpha.velocity[1] * TIME_STEP
        movement = np.array([movement_x, movement_y])

        #moving the alpha particle
        alpha.location = alpha.location + movement

        #moving the velocity the same amount so relative origin stays same for next step
        alpha.velocity = alpha.velocity + movement

    #print(closest_location, alpha.location, history)
    return history, closest_location
```

Here we calculate our scattering angle.

```
def calculating_scattering_angle(closest_approach,alpha, b):

    #we split the closest approach into x and y components
    closest_y = closest_approach[1]

    #if the x component is negative we make it positive
    if closest_approach[0] < 0:
      closest_x = -1 * closest_approach[0]
    else:
      closest_x = closest_approach[0]

    #we split the final location of the alpha particle into x and y components
    end_loc_y = alpha.location[1]

    #if the x component is negative we make it positive
    if alpha.location[0] < 0:
      end_loc_x = -1 * alpha.location[0]
    else:
      end_loc_x = alpha.location[0]



    #calculates angle psi of the closest approach of alpha particle - returns radians
    psi_of_closest_approach = np.arctan(closest_y/closest_x)


    #finding the point where the line between closest approach and target nuclei intercepts with b
    intersection_y = b
    #np.tan() takes in rad
    intersection_x = intersection_y/np.tan(psi_of_closest_approach)
    intersection_location = np.array([intersection_x, intersection_y])

    #since we can't divide by 0, if it bounces straight back we hardcode the scattering angle to 0
    if end_loc_y == 0:
      scattering_angle = 180
    else:
```

```
    #finds difference between intersection point and final alpha location
    difference_x = end_loc_x - intersection_location[0]
    difference_y = end_loc_y - intersection_location[1]

    #calculates scattering angle
    scattering_angle = np.arctan(difference_y/difference_x)

    #converts to degrees from radians
     scattering_angle = np.rad2deg(scattering_angle)


    #if the alpha particle bounces back it returns 180 - scattering angle
    if alpha.location[0] < closest_approach[0]:
      scattering_angle = 180 - scattering_angle

  #for case when end_loc x is negative as well as closest approach x but end_lox x is bigger
  if scattering_angle < 0:
    scattering_angle = scattering_angle *-1

  return scattering_angle
```

This splits our trajectory history into x and y values, so that it will be easier to plot.

```
def plot_prepare_trajectory(history):
    x_loc = []
    y_loc = []

    for i in history:
      x_loc.append(i[0])
      y_loc.append(i[1])

    return x_loc, y_loc
```

This function simulates one α-particle and calculates the scattering angle.

```
def simulater(alpha, b):
  history, closest_approach = simulation_loop(alpha)
  scattering_angle = calculating_scattering_angle(closest_approach, alpha, b)
  return scattering_angle, history, closest_approach

#converts MeV to Joules
def mev_to_vel(mev, mass):
  joule = mev * 1.6022E-13
  velocity = np.sqrt(joule/(0.5*mass))
  return velocity

def myround(x, base=15):
    return base * round(x/base)
```

Plots the trajectory of an α-particle.

```
def main_trajectory(number_of_alpha_particles=10, b_max=1e-14, target_proton_num=79, mev=7.7):

  start_velocity = mev_to_vel(mev, alpha_particle_mass)

  target.charge = target_proton_num * 1.6e-19
  plt.figure(figsize=(10,10))
  plt.style.use('ggplot')

  #plotting target nucleus
  plt.plot(0,0, marker = 'o')

  for i in range(number_of_alpha_particles):

    #selecting y distance (impact parameter) range to randomly initialize particles from
    b = np.random.uniform(0,5)*(b_max)

    #start location in meters
    alpha_start_location = np.array([-2e-14,b])


    alpha_start_velocity = np.array([start_velocity- alpha_start_location[0], b])

    alpha = Particle(alpha_particle_mass, alpha_start_location, alpha_particle_charge, alpha_start_velocity)

    #first alpha particle
    scattering_angle, history, closest_approach = simulater(alpha, b)

    x_loc, y_loc = plot_prepare_trajectory(history)


    plt.plot(x_loc, y_loc, '--',label=f'{i+1}')



  plt.legend()

  plt.title('Alpha Particle Trajectory')
  plt.ylabel('Y distance - impact parameter - (m e-14)')
  plt.xlabel('X distance (m e-14)')

  plt.tight_layout()

  ax = plt.gca()
  ax.set_ylim([-5e-14,5e-14])
  ax.set_xlim([-2.1e-14, 2e-14])

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

Plots number of α-particles against scattering angle.

```
def main_r_formula(number_of_alpha_particles=1000,  b_max=1e-13, target_proton_num=79, mev=7.7, label = 'Data
points'):
  global alpha_particle_mass

  #calcualtes the start x velocity of alpha particle
  start_velocity = mev_to_vel(mev, alpha_particle_mass)

  #calculates the charge of the target in joules
  target.charge = target_proton_num * 1.6e-19
  plt.figure(figsize=(10,10))
  plt.style.use('ggplot')

  #counts for the scattering angle rounded to nearest 15 degrees
  scattering_angle_count = {0: 0, 15: 0, 30: 0, 45: 0, 60: 0, 75: 0, 90: 0, 105: 0, 120: 0, 135: 0, 150: 0, 165:
0, 180: 0}

  #counts for the scattering angle rounded to nearest 60 degrees - this is used later for quantitative analysis
  scattering_angle_count_grouped = {0: 0,60: 0,120: 0, 180: 0}


  for i in range(number_of_alpha_particles):
    b = np.random.uniform(0,1)*(b_max)
    #start location in meters
    alpha_start_location = np.array([-2e-14,b])

    #velocity at the start of the experiment
    alpha_start_velocity = np.array([start_velocity - alpha_start_location[0], b])

    #initializing the particle
    alpha = Particle(alpha_particle_mass, alpha_start_location, alpha_particle_charge, alpha_start_velocity)

    #first alpha particle
    scattering_angle, history, closest_approach = simulater(alpha, b) # passing the first particle

    #rounding to nearest 15 degrees
    rounded_scattering_angle = myround(scattering_angle)

    higher_rounded_scattering_angle = myround(scattering_angle, base=60)

    #adding to count
    scattering_angle_count[rounded_scattering_angle] = scattering_angle_count[rounded_scattering_angle] + 1

    #adding to count
    scattering_angle_count_grouped[higher_rounded_scattering_angle] =
scattering_angle_count_grouped[higher_rounded_scattering_angle] + 1

    #shows the user how many particles have been sent
    print(f'{i+1} particles fired')


  #gets values from scattering angle dictionary
  angles = list(scattering_angle_count.keys())
  count = list(scattering_angle_count.values())

  #gets count from dictionary rounded to nearest 60 degrees
  rounded_count = list(scattering_angle_count_grouped.values())

  #plots number of particles against scattering angle
  plt.plot(angles, count, marker = 'o', label=label)

  plt.title("Number of alpha particles scattered against scattering angle (Rutherford's Formula)")
  plt.ylabel('Number of alpha particles')
  plt.xlabel('Scattering angles (rounded to nearest 15)')

  plt.legend()
  plt.tight_layout()

  plt.grid(True)
  plt.show()

  return rounded_count
```

This function quantitatively tests the scattering counts produced by our model with the results taken from Rutherford's formula.

```
def test(number_of_alpha_particles=1000, target_proton_num=79, mev=7.7):
  #atoms per unit volume gold
  n = 1

  #thichness of target (we only have one nucleus) in meters
  L = 1.2e-15 * 5.8

  #electron charge
  e = -1.60217663e-19

  #coulombs constant
  k = 8.99e9

  #atomic number of target
  Z = target_proton_num

  #kinetic energy of particle in mev
  KE = mev

  #total fired
  Ni = number_of_alpha_particles

  #distance to detector
  r = 2e-14

  scattering_counts = []

  #goes through angles 0-180 in steps of 60
  for i in range(0,240,60):

    #converts the degrees into radians
    scattering_angle = np.deg2rad(i)

    #since the formula won't work with 0, and the angle never is truely 0
    if i==0:
      N = (Ni * n * L * Z**2 * k**2 * e**4) / (4 * r**2 * KE**2 * np.sin(0.5)/2)**4)

    #we plug in the values here: all SI units
    else:
      N = (Ni * n * L * Z**2 * k**2 * e**4) / (4 * r**2 * KE**2 * np.sin(scattering_angle/2)**4)

    scattering_counts.append([i, N])
```

Here we take the total of the counts taken from Rutherford's formula as we need to normalize them so that the total is the same as Ni. We take a factor and divide each result by it.

```
  total = 0

  #adds solutions to total
  for i in scattering_counts:
   total+=i[1]

  #creates a factor so that the total = Ni
  factor = total/Ni

  #uses factor and we get new results
  for i in scattering_counts:
     i[1] = i[1]/factor

    #rounded to nearest whole number
     i[1] = round(i[1])

  counts = []
  #we add the factored counts to the counts array
  for i in scattering_counts:
    counts.append(i[1])

  angles = [0,60,120,180]

  simulation_count = main_r_formula(number_of_alpha_particles=1000,  b_max=3.1e-13, target_proton_num=79,
mev=7.7, label = 'Simulation results')

  #width of bar
  w = 0.4

  #this part is done so we can have two bars side by side.
  bar1 = np.arange(len(angles))
  bar2 = [i+w for i in bar1]
  plt.bar(bar1, counts, width = w, label = "Rutherford's formula",align="center")
  plt.bar(bar2, simulation_count, width = w, label = "Simulation results",align="center")
  plt.xticks(bar1 + w/2, angles)

  plt.legend()
  plt.xlabel('Angles (degrees)')
  plt.ylabel('Number of particles')

  print(f'Expected: {counts}')
  print(f'Observed: {simulation_count}')

  plt.title('Comparison between expected results and observed results')
  plt.legend()
  plt.tight_layout()
  plt.grid(True)
  plt.show()

#compares rutherford's formula for alpha particles with different energies
def compare(number_of_alpha_particles=1000,  b_max=1e-13, target_proton_num=79, mev=7.7, label = '1',
number_of_alpha_particles1=1000,  b_max1=1e-13, target_proton_num1=79, mev1=7.7, label1 ='2',
title='Comparison'):
  angles = [0,15,30,45,60,75,90,105,120,135,150,165,180]
```

```
one = main_r_formula(number_of_alpha_particles,  b_max, target_proton_num, mev)

two = main_r_formula(number_of_alpha_particles1,  b_max1, target_proton_num1, mev1)

plt.plot(angles, one, marker = 'o', label = label)
plt.plot(angles, two, marker = 'o', label = label1)
plt.title(title)
plt.ylabel('Number of alpha particles')
plt.xlabel('Scattering angles (rounded to nearest 15)')


plt.legend()
plt.show()
```

5. Data analysis

In this chapter we will compare the data collected by changing the parameters of the simulation. This chapter will allow us to understand more clearly how the electric field of a nucleus affects the α-particle's trajectory and hence scattering angle. We will be exploring how we reached the conclusion in section 1.3 'Summary of results'.

We note in this chapter we only consider one gold nucleus; we will analyse the 2d lattice next chapter (6. More data).

5.1 Rutherford's Formula

Rutherford's formula is the analytical relationship which provides the number of α-particles scattered at a certain angle. In Figure 5.A we can see the formula and its parameters. In figure 5.B we see the graph of the formula.

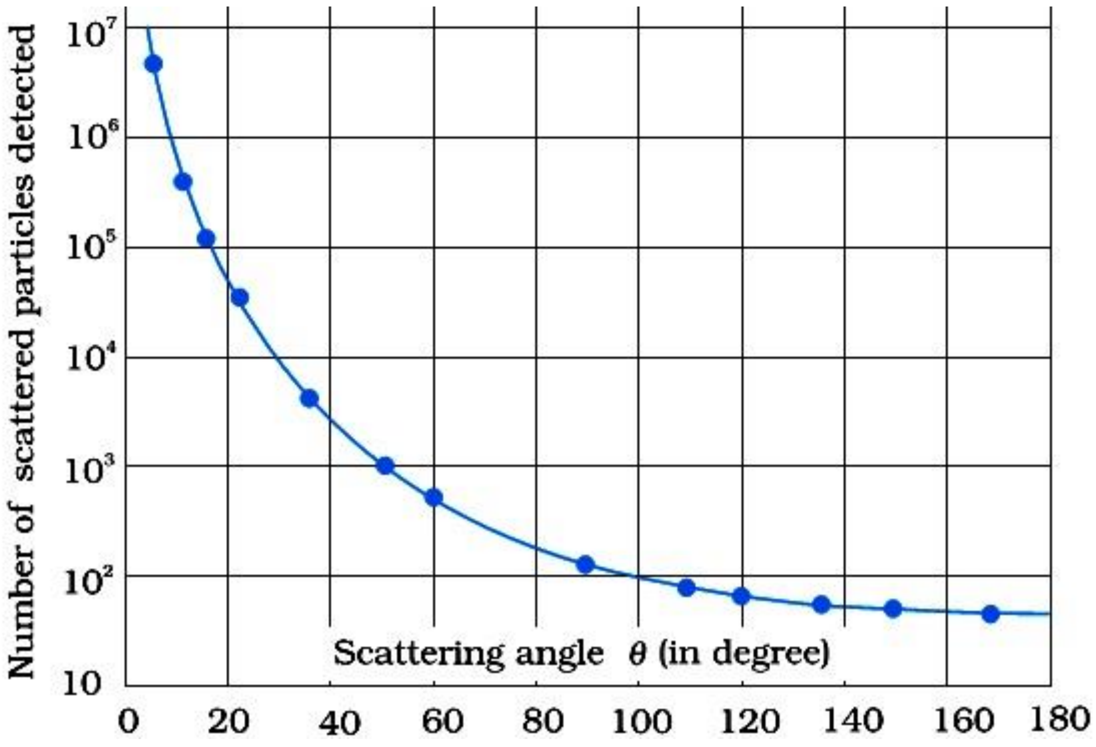
Figure 5.A

$$N(\theta) = \frac{N_i n L Z^2 k^2 e^4}{4r^2 KE^2 \sin^4(\theta / 2)}$$

$N_i =$	*number of incident alpha particles*
$n =$	*atoms per unit volume in target*
$L =$	*thickness of target*
$Z =$	*atomic number of target*
$e =$	*electron charge*
$k =$	*Coulomb's constant*
$r =$	*target-to-detector distance*
$KE =$	*kinetic energy of alpha*
$\theta =$	*scattering angle*

Source: http://hyperphysics.phy-astr.gsu.edu/hbase/rutsca.html

Figure 5.B



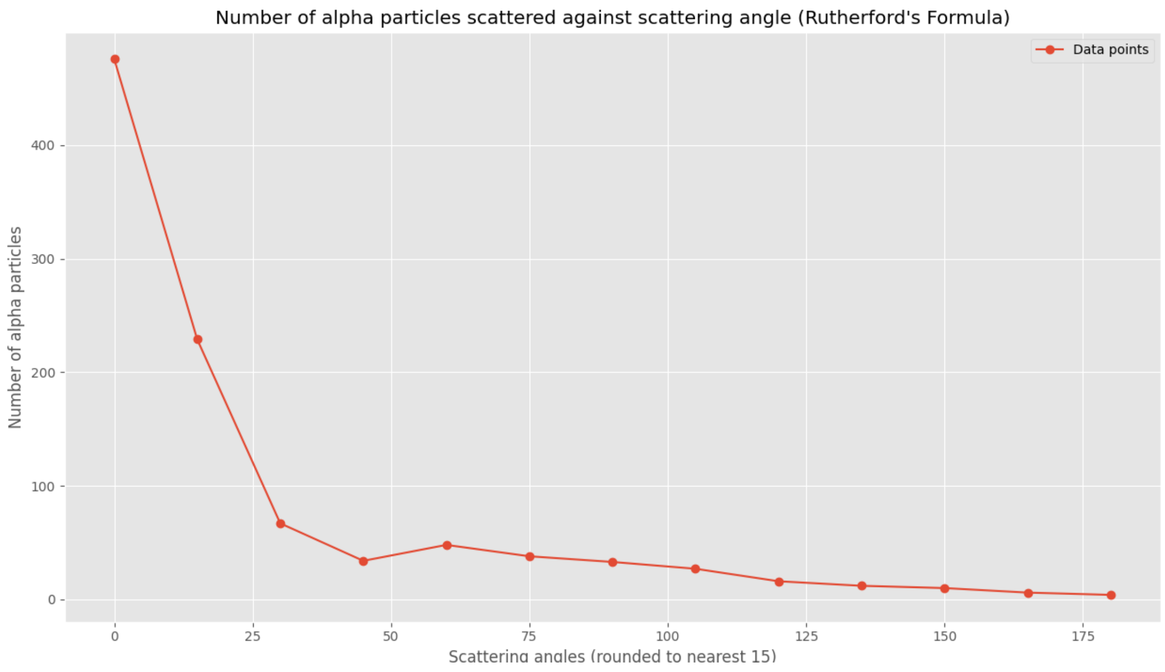
Source: https://physicsopenlab.org/2017/04/11/the-rutherford-geiger-marsden-experiment/

The model we created produces a similar graph with the default parameters (listed in 3.2 'Parameters), which we can see below in figure 5.C. This graph is sufficiently similar to Rutherford's formula quantitatively, which means we can answer a couple of the queries we outlined in section 1.2.3 'Addressing the issues'.

Firstly, as mentioned previously, we only include the electrostatic force (no strong nuclear force), and yet we produce a very similar graph. This displays that the strong nuclear force has an extremely little effect on the scattering of α-particles.

Secondly, we can confirm our assumption that the thickness of the gold foil has a near irrelevant effect on the scattering pattern. Even though our graph uses one atom, and Rutherford used a 3d lattice the graphs are alike. In chapter 6 we run our own simulation with a 2d lattice and test the effects of that, and see if it is any different to the graph we currently have.

Figure 5.C

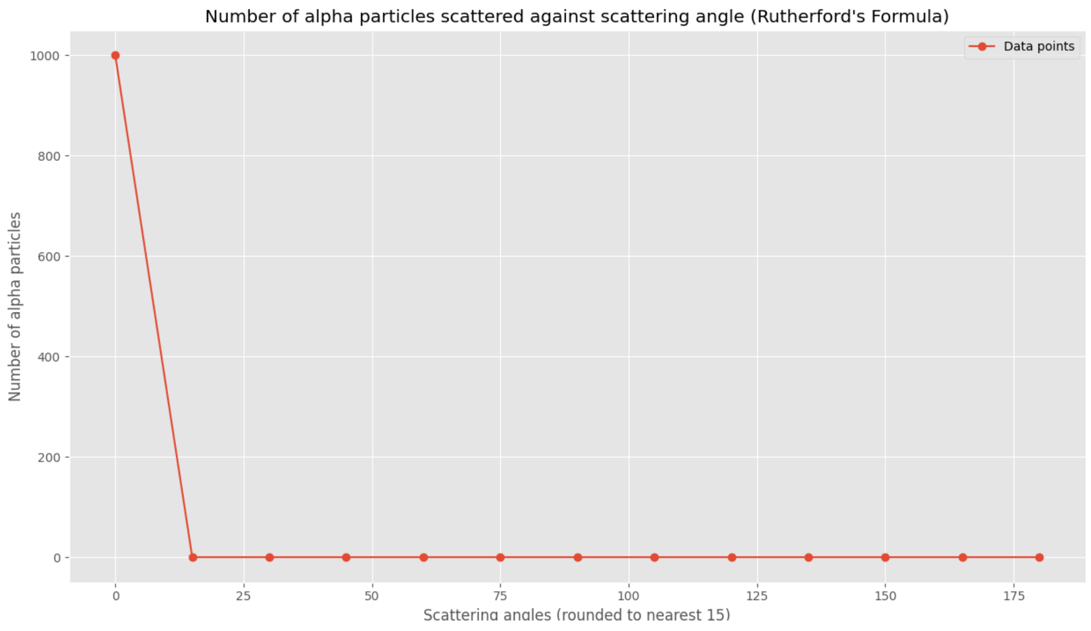


5.2 Changing the max impact parameter

To be able to conduct a fair experiment with only one atom we have to select our max impact parameter carefully. In the actual experiment Rutherford used a 1mm slit to focus a beam of α-particles towards the gold foil. Below (figure 5.D) we see the graph produced when we use a slit of 1e-10m let alone 1mm (which is 1e-3m). All of the α-particles are not deflected.

Figure 5. D

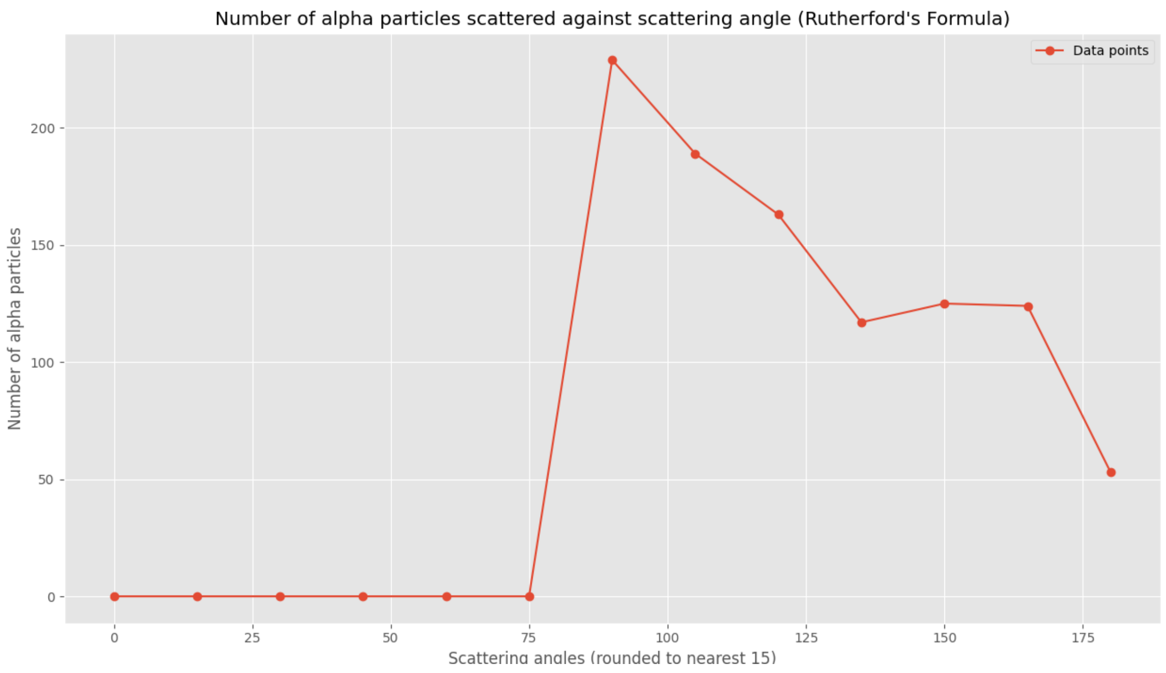
```
#All default parameters except b_max
main_r_formula(b_max=1e-10)
```



The problem here is that Rutherford had a lattice of gold, not just a single atom. The atomic radius of a gold is 1.46e-10. However we can't use this as the max b because we send far less α-particles in the simulation compared to the real experiment. This leads to higher scattering angles not occurring. But if we decrease the max value for b too much then we don't see any low scattering angles, as seen below in figure 5.E.

Figure 5.E

```
#All default parameters except b_max
main_r_formula(b_max=1e-14)
```



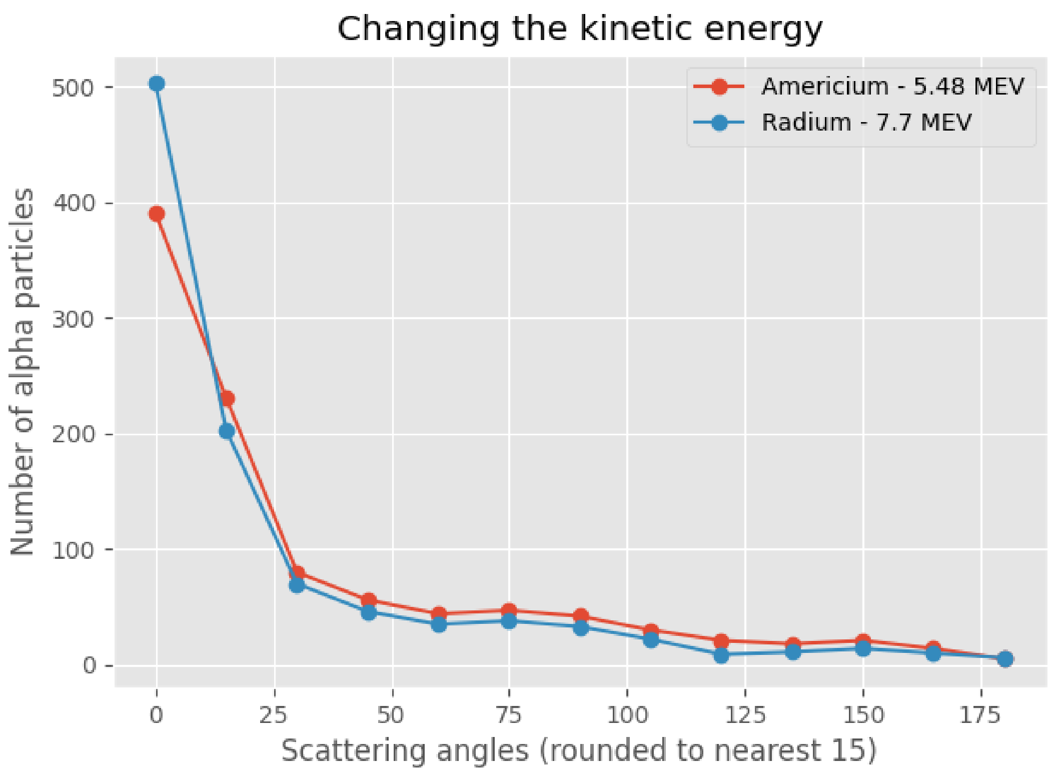
A suitable max b value I found was 1e-13m. This produces a graph which we can qualitatively compare to Rutherford's formula. Figure 5.C is the graph we get with 1e-13m as the max b and figure 5.B is Rutherford's formula.

5.3 Changing the particle's kinetic energy

As we see in Rutherford's formula (figure 5.A), the kinetic energy of an α-particle being fired does affect the scattering pattern. Since we are not analytically analysing the effects of this, let's quantitively view the change in the graph below (figure 5.G). We have chosen two different radioactive sources which can both be used for the experiment (radium was originally used by Rutherford). Radium emits particles with 7.7 MEV, and americium 5.48 MEV.

Figure 5.G

```
#Comparing scattering pattern at different kinetic energies
compare(mev=5.48, label='Americium - 5.48 MEV', mev1=7.7, label1='Radium - 7.7 MEV',
title='Changing the kinetic energy')
```



As we can see from figure 5.G the α-particles emitted from americium has less particles scattering at angles near zero, compared to radium, and more particles scattered at higher angles. This is also consistent throughout different values for the kinetic energy.

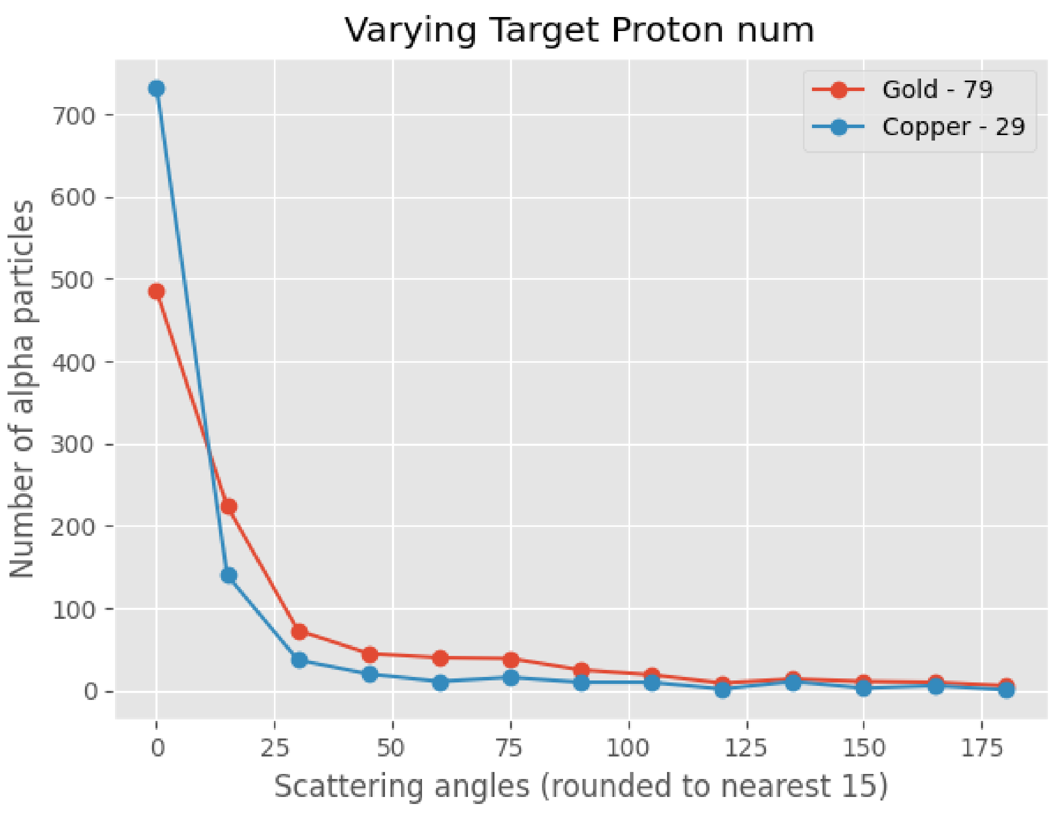
5.4 Changing the target's proton number

In the actual experiment Geiger and Marsden used gold foil as the target metal due to its unparalleled malleability. However, we don't have any such limitation. Therefore we will test the effect of varying the proton number of the target.

Before viewing the graph, we can logically assume that the scattering angle will decrease as we decrease the atomic number of the target, as the coulombic force decreases. Below in figure 5.K see the difference between using gold (79 protons) and copper (29 protons).

```
#Changing target's atomic number
compare(target_proton_num=79, label='Gold - 79', target_proton_num1=29, label1='Copper -
29', title='Varying Target Proton num')
```

Figure 5.K



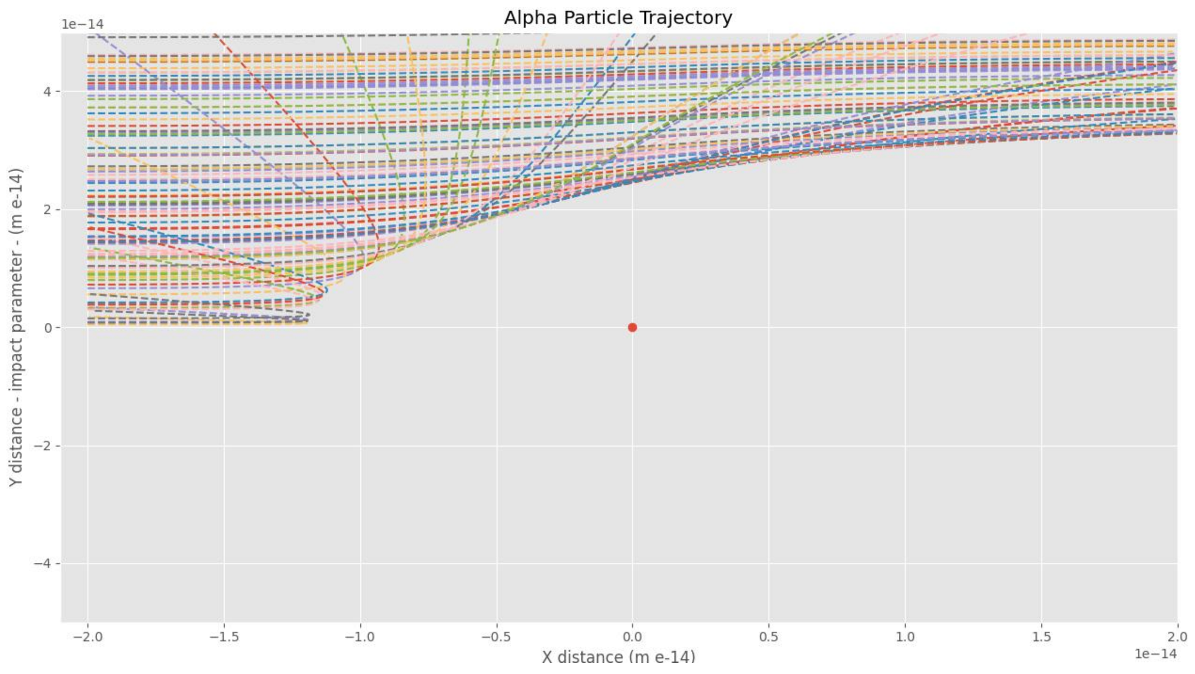
Our assumptions looks right, as the majority of the α-particles fired at copper atom scatter at near zero degrees. Whereas the gold's electric field, being more strong, deflects the particles at higher angles.

5.5 Closest approach relationship

We notice another relationship when we plot the trajectory of 1000 α-particles, as seen in figure 5.L. It shows a distinct curved line that no α-particles travel through; a line of all the closest approaches.

```
main_trajectory(number_of_alpha_particles=1000, max_b=1e-13)
```

Figure 5.L



When the α-particle is fired at b = 0, we can calculate the closest approach of the trajectory easily, as shown below. Where E is the kinetic energy the particle is fired with, and D is the closest approach.

$$E = k\frac{q_1 q_2}{D^2}$$

$$D = \sqrt{k\frac{q_1 q_2}{E}}$$

This calculates the point at which all the kinetic energy is converted to potential energy by the repulsive coulombic force, and the particle comes to rest. However, as I mentioned this relationship is only applicable when there is a head on collision. When $b > 0$, the following relationship is applicable:

$$\tan(\theta/2) = D/2b$$

This relationship's derivation won't be covered but can be seen in this source.[1]

[1] http://www.personal.soton.ac.uk/ab1u06/teaching/phys3002/course/02_rutherford.pdf

6. Quantitative analysis

In the previous chapter we qualitatively compared our results to Rutherford's formula, and also observed the results of varying parameters. However, in this chapter we will quantitatively compare our experimental results to the expected results. For convenience Rutherford's formula is provided again below in figure 6.A.

Figure 6.A

$$N(\theta) = \frac{N_i n L Z^2 k^2 e^4}{4r^2 KE^2 \sin^4(\theta / 2)}$$

$N_i =$ *number of incident alpha particles*

$n =$ *atoms per unit volume in target*

$L =$ *thickness of target*

$Z =$ *atomic number of target*

$e =$ *electron charge*

$k =$ *Coulomb's constant*

$r =$ *target-to-detector distance*

$KE =$ *kinetic energy of alpha*

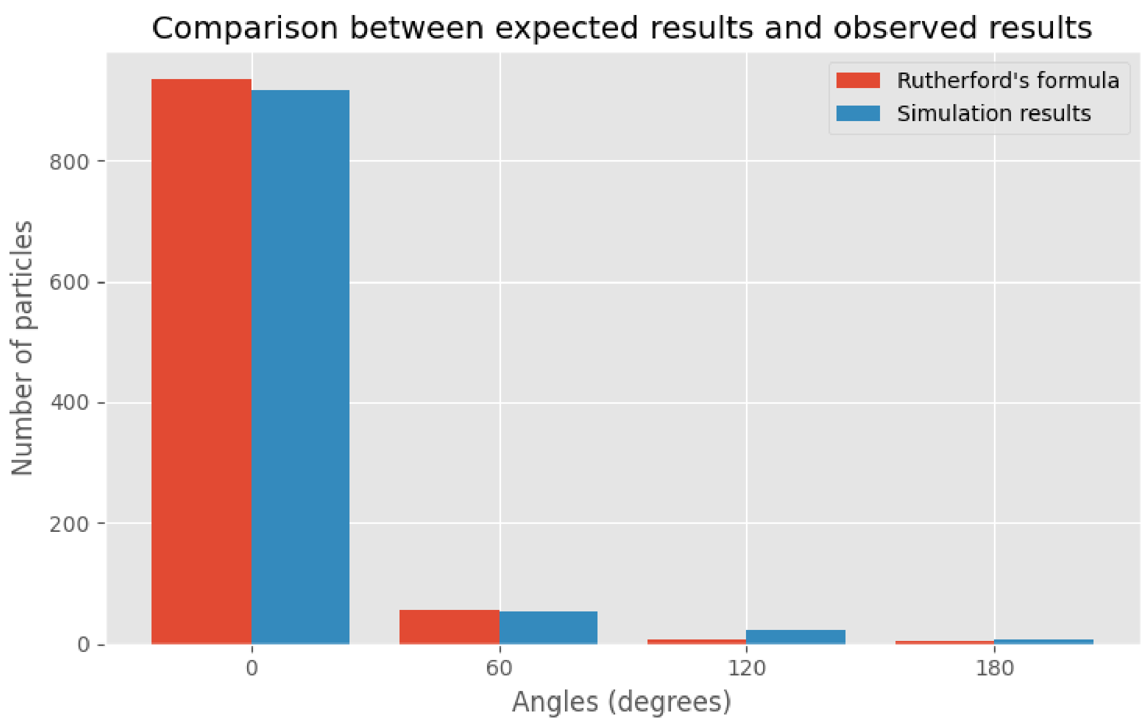
$\theta =$ *scattering angle*

Source: http://hyperphysics.phy-astr.gsu.edu/hbase/rutsca.html

The default values to the variables defined above can be seen in section 4 'Usage'. We use Rutherford's formula and plug in different scattering angles, this produces the expected scattering count for different angles. Then we compare these counts with the counts our model produces. The results are seen below in figure 6.B.

An important thing to note is that these counts are rounded to the nearest 60, therefore each bar represents a range. One difference to our previous Rutherford scattering graph is that our ranges are bigger and we round to the nearest 60 degrees rather than 15. We do this when comparing with expected results as after 120 degrees the count trails off, this is not useful when quantitatively analysing results. As well as this after 120 degrees, we start to see 0s in our expected count, and the Chi-square test can't handle that.

Figure 6.B



6.1 Chi-square goodness of fit test applied

We can utilize the Chi-square goodness of fit test in this scenario to test if our observed results, although they look similar, are following Rutherford's formula. I will not go through the details of the test, but they can be seen here[2]. The results are below:

Row	Category	Observed	Expected #	Expected %
1	0	936	934	93.307%
2	60	50	56	5.594%
3	120	10	6	0.599%
4	180	5	5	0.500%

The Chi-square equals 3.314, with three degrees of freedom. The two-tailed P value equals 0.3457. This suggests that the difference between observed and expected data is not statistically significant. However, there is a difference. Ideally our P value would be slightly higher, and the next steps to achieve that goal can be seen in section 7.2 'Further development'.

[2] https://www.jmp.com/en_sg/statistics-knowledge-portal/chi-square-test/chi-square-goodness-of-fit-test.html

7. Conclusion

Within this project we made use of not being restrained by the physical word. We tested several parameters in our model, while addressing the issues with the actual experiment Geiger and Marsden conducted. After analysing the data in chapter 5, we reach the following conclusions which are listed in section 1.3 and are below:

- The strong nuclear force has an insignificant effect on the scattering of α-particles.
- Decreasing the kinetic energy of an α-particle slightly increases the average scattering angles.
- There can be an exception to Rutherford's formula when an α-particle's energy exceeds a certain point (roughly above 25 MeV).
- Using a different metal, such as copper, does affect the scattering pattern. Decreasing the proton number, slightly decreases the scattering angle.

These conclusions give us a clearer insight into how the electric field of a nucleus affects the scattering of α-particles.

7.1 Limitations of the model

- Our model can't compute the trajectory of α-particles as quick as real life, therefore we struggle to collect the same amount of data as Geiger and Marsden within a reasonable time frame
- Just by the nature of selecting a time step we create an inaccuracy as the model isn't continuous. The shorter we make the timestep, we can increase the accuracy. However, we calibrated our timestep such that there was only slight deviation observed (see chapter 6 'Quantitative analysis')

7.2 Further development

- It would be interesting to run the model with a 2d lattice to observe the affect on the scattering angle. After this we could extend it to a 3d lattice.
- Adding a radius nuclei would better imitate reality, rather than just treating them as point particles.
- Including electrons would also create a more realistic model, as they can slightly affect the scattering pattern. However, our focus is on how the nucleus interacts with α-particles.
- Modelling the strong nuclear force could concretely display the effect is has on the scattering pattern.

References

Schaeffer, B. (2016) Anomalous Rutherford Scattering Solved Magnetically. *World Journal of Nuclear Science and Technology*, **6**, 96-102. doi: 10.4236/wjnst.2016.62010. Retrieved 25th March, 2022, from https://www.scirp.org/journal/paperinformation.aspx?paperid=65604

Nave, Hyper Physics, Rutherford Scattering Retrieved 28th March, 2022, from http://hyperphysics.phy-astr.gsu.edu/hbase/rutsca.html

Advanced Lab, Rutherford Scattering Experiment. Retrieved 1st April, 2022, from https://advancedlab.physics.gatech.edu/labs/rutherford/rutherford-2.html

Physics Open Lab, The Rutherford-Geiger-Marsden Experiment. Retrieved 3rd April, 2022, from https://physicsopenlab.org/2017/04/11/the-rutherford-geiger-marsden-experiment/

Study Smarter, Rutherford Scattering. Retrieved 5th April, 2022, from https://www.studysmarter.de/en/explanations/physics/nuclear-physics/rutherford-scattering/