Abstract

In this project I looked at the functioning of a cyclotron by simulating the acceleration of a charged particle. In theory, the cyclotron works on the principle of EM interactions. A charged particle is placed in the presence of strong EM fields, and based on the Lorentz force, experiences some form of acceleration that causes the charge to follow what is essentially a spiral path. The simulation was carried out using two different methods of numerical analysis – Euler's method and the more precise Runge–Kutta method (Stoltz). These are two methods of approximating the solutions to ordinary differential equations. Through the simulation we see that the Runge–Kutta method is in fact more precise ("Runge–Kutta Methods"); however, neither method gives us a perfect simulation as both result in a violation of energy conservation. We will also explore other limitations of this study such as the exclusion of relativistic effects.

Introduction

The cyclotron is just one in a growing line of types of particle accelerators. For a long time, laboratories depended on linear accelerators for high energy beams to carry out high energy, particle, and nuclear physics experiments. Such accelerators were limited by their length and could only accelerate particles till the entire length of their build. This problem was combated by the invention of the circular particle accelerator.

The cyclotron works on the principle of EM interactions and uses a magnetic field to accelerate charges into circular paths without affecting their kinetic energy. The charge whose energy is increased using an oscillating electric field is also subject to a constant magnetic field so that the particle follows a circular path and can be accelerated for a greater amount of time, allowing the particle to achieve greater energies than what was possible using linear accelerators. However, for most experiments that require high energy beams of particles, generally a combination of linear and circular accelerators are used to produce beams with energies of very high orders of magnitudes.

Motivation

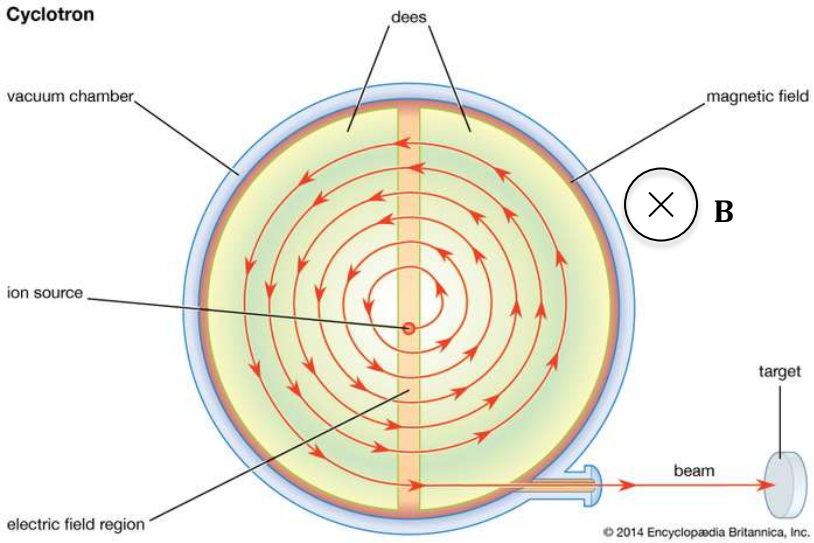
In the summer of 2017, I visited CERN in Geneva on a CMS training program. It was through this program that I decided to pursue physics in my higher education. During this, we visited the Synchrocyclotron – CERN's first particle accelerator that for around 33 years had been providing accelerated beams for their first experiments in particle and nuclear physics, since 1957. I chose to work on this topic because I only had a theoretical understanding of the working of such particle accelerators, but with my newly gained computational skills I hope to deepen my quantitative understanding of the working of such accelerators.

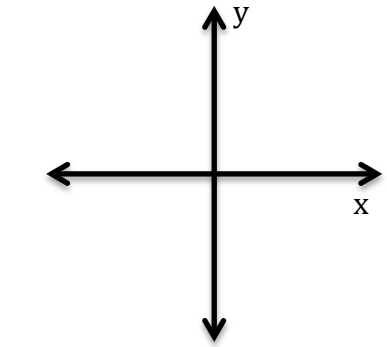
Additionally this topic provides us with a new context to analyze the effect of different kinds of EM fields on charges in motion computationally. Also conducting this simulation using different methods of numerical analyses – Euler's method and the Runge-Kutta method – helps understand the differences in these methods and helps study the effectiveness of each method. Additionally the limitations of this

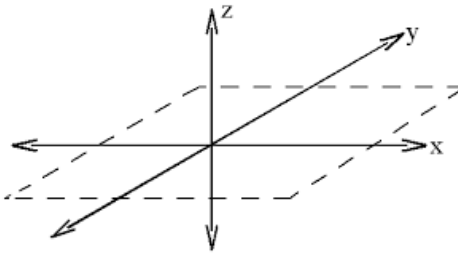
method and project provide us with opportunities to explore more precise methods of computation and complex systems.

Background

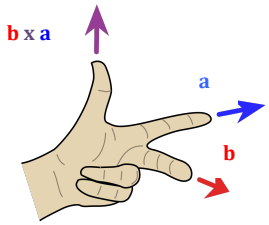
A cyclotron consists mainly of two evacuated semi-circular chambers or 'DEES'. In the middle of the cyclotron is an ion source from where the charged particles to be accelerated are released. There are two fields present in the system. One is a constant magnetic field, which is depicted by B in the figure below, and the second is the electric field, which is show as the "electric field region" in the diagram below.







When the charges are released, they are accelerated by the electric field in the direction of the arrow show. As the charge enters the magnetic field region, it experiences a Lorentz force, the direction of which is given by Fleming's left hand rule. At the instant the charge first enters the Dee, the direction of motion of charge, given by $\boldsymbol{b}$, is in the positive x direction. The direction of the magnetic field given by $\boldsymbol{a}$ is in the negative z direction. Therefore, the force on the charged particle given by $\boldsymbol{b} \times \boldsymbol{a}$ is in the positive y direction.



The charge continues to accelerate as a result of the magnetic force, till it exits the first Dee. Then, it is once again accelerated by the electric field, but this time in the negative x direction. This change is a result of the oscillating electric field. These oscillations are brought about by a high frequency oscillator. The frequency of these oscillations can be calculated based on the time period of the motion of the charge in the Dees.

Assuming circular motion, we can calculate the amount of time for which the charge is in the Dee. We can equate the magnetic force to a centripetal force for circular motion:

$$qvB = m\frac{v^2}{R}$$

Rearranging this we get:

$$R = \frac{mv}{qB}$$

The time period for one full revolution of circular motion is given by:

$$T = \frac{2\pi R}{v}$$

Substituting for R we get the expression:

$$T = \frac{2\pi m}{qB}$$

Since the charge only spends half a time period in the Dee, the amount of time the charge spends inside one D at a given radius is $\frac{T}{2}$. In this time, the electric field completes half an oscillation. Therefore the time period of these oscillations is also given by $T = \frac{2\pi m}{qB}$. Based on this, we calculate the frequency of these oscillations as:

$$f = \frac{qB}{2\pi m}\ Hz$$

and

$$\omega_c = \frac{qB}{m}\ rad\ s^{-1}$$

Where q is the charge of the particle, v is the instantaneous velocity of the particle, B is the magnetic field strength, R is the radius of any half oscillation the particle makes inside a Dee, T is the Time period of a full oscillation, f is the frequency of a full oscillation and ω_c is the angular frequency of the oscillations (Phalke).

This 'electrodynamic' nature of the electric field is necessary to maintain a circular path while increasing the energy of the particle. If these oscillations were not to exist, and the polarity of the electric field had remained the same, the charged particle would now experience negative acceleration by the electric field in the positive x direction. This would result in a decrease in the kinetic energy of the particle.

Once the particle has been accelerated by the 'switched' electric field, it now enters the second Dee with an increased velocity. Due to this increased velocity, the particle follows a circular path with a greater radius that before. This is clear from the formula of the radius of the path, as was derived earlier, given by:

$$R = \frac{mv}{qB}$$

With each switch the electron makes from one Dee to another, it is accelerated by the electric field and the resulting radius in its next half oscillation increases. In this way, the charge never follows a path of the same radius inside a Dee. Because of this, we observe the outward spiral path of the charged particle.

With each switch the electron makes from one Dee to another, its kinetic energy increases on account of being accelerated by the oscillating electric field. This continues for a specified amount of time till the particle acquires the desired energy. Once the particle has enough energy it exits the cyclotron through a gate, as is shown in the figure above.

Model and Set Up

Initially we set up necessary functions to run the simulation.

```
def calc_force_on_charge_by_efield(electric_field, q = 1.67E-19) :
    force = q*electric_field
    return force

def calc_force_on_charge_by_bfield(magnetic_field, charge_velocity, q = 1.67E-19) :
    force = q*np.cross(charge_velocity, magnetic_field)
    return force
```

The above lines of code set up a function to calculate the force on the charged particle (proton, $q = 1.67 \times 10^{-19}\ Q$). The force due to the electric field is: $F = qE$. The force due to the magnetic field is: $F = qvB$. Over here E, v, and B are treated as vectors and hence numpy arrays with three elements. We use these functions next to define a new function: acceleration due to the EB field.

```
def calc_acceleration_of_charge_in_ebfields(electric_field, magnetic_field, charge_position, charge_velocity, q, m) :
    ''' Calculate the acceleration of a charge in both electric and magnetic fields '''

    if charge_position[0] > 0 or charge_position[0] <= -gap:
        a = q*(np.cross(charge_velocity,magnetic_field))/m
        a = calc_force_on_charge_by_bfield(magnetic_field, charge_velocity, q)/m
    else:
        a = q*electric_field
        a = calc_force_on_charge_by_efield(electric_field,q)/m
        if charge_position[1] > 0:
            a = -a
    return a
```

It is in this function (Marcrowo) that we take steps to model the effect of the electric and magnetic fields on the charged particle.

```
if charge_position[0] > 0 or charge_position[0] <= -gap:
```

The above line of code creates a condition in which the charged particle is inside either of the two Dees. '*Gap*' is treated as the distance between the two Dees in the simulation. It is a constant in the code and can be changed to give varied results. When the x-position of the particle is positive (inside the right hand side Dee) or lesser than '*-gap'* (inside the left hand side Dee), it is accelerated only by the magnetic field. This acceleration is calculated using the previous function '*calc_force_on_charge_by_bfield*', which is divided by the mass of the particle that is later defined in the code to be $m = 1.67 \times 10^{-27}$kg.

```
gap = 0.1
```

For the condition where the charged particle is inside the 'gap' or in between the Dees, the charged particle only experiences the effect of the electric force. This is also calculated using the previous function '*calc_force_on_charge_by_efield*' which is also divided by m to give acceleration.

A simplification of this model is that, the electric field has not been modeled as an oscillating field with the equations given before. However, the model does account for the switch in polarity of the electric field in the '*calc_acceleration_of_charge_in_ebfields'* function.

```
if charge_position[1] > 0:
    a = -a
```

The above lines of code lie in the '*else'* part of the condition (particle is in the gap). When the y-position of the particle is positive – this is when the particle should be accelerated in the negative x direction – the acceleration is set to be negative of its value to account for the switch in polarity of the electric field. In this way we can work with constant EB fields.

The next step is to set our electric and magnetic fields.

```
def constant_electric_x_field(charge_position) :
    ''' This creates a field as $\vec{E} = [5000000, 0, 0]$'''
    return np.array([5000000.0,0.0,0.0])

def constant_magnetic_z_field(charge_position) :
    ''' This creates a field as $\vec{B} = [0, 0, -2]$'''
    return np.array([0,0,-2])
```

The values for these fields were set through troubleshooting. As mentioned earlier, these fields are set as arrays with three elements.

Next, we set up two more functions, which will help us in computing the trajectory of the particle.

```
def get_updated_value(current_value, rate_of_change, dt) :
    updated_value = current_value+rate_of_change*dt
    return updated_value
```

This first function is nothing but an ordinary differential equation to get updated values of a certain parameter.

```
def get_rates_of_change_euler(particle_position, particle_velocity, electric_field_function,
                              magnetic_field_function, q, m, dt) :

    electric_field_at_position = electric_field_function(particle_position)
    magnetic_field_at_position = magnetic_field_function(particle_position)

    particle_acceleration = calc_acceleration_of_charge_in_ebfields(electric_field_at_position,
                                                                    magnetic_field_at_position,
                                                                    particle_position, particle_velocity, q, m)
    return particle_velocity, particle_acceleration
```

The above function uses Euler's method, as described earlier, to get the values of the particle's velocity and acceleration, which will be substituted as the rates of change in the '*get_updated_value*' function to calculate the particle's updated position and velocity.

The Runge-Kutta method to obtain the values of the particle's velocity and acceleration is also defined.

```
def get_rates_of_change_runge_kutta(particle_position, particle_velocity, electric_field_function,
                                    magnetic_field_function, q, m, dt) :


    current_velocity, current_acceleration = get_rates_of_change_euler(particle_position, particle_velocity,
                                                                       electric_field_function,
                                                                       magnetic_field_function, q, m, dt)

    mid_particle_velocity = current_velocity + current_acceleration * dt/2
    mid_particle_position = particle_position + mid_particle_velocity * dt/2

    # Calculate the acceleration due to the electric and magnetic fields at the mid_particle_position
    runge_kutta_velocity, runge_kutta_acceleration = get_rates_of_change_euler(mid_particle_position,
                                                                               mid_particle_velocity,
                                                                               electric_field_function,
                                                                               magnetic_field_function, q, m, dt)
    return runge_kutta_velocity, runge_kutta_acceleration
```

Then the values with which the simulation will run are initialized.

```
c = 3.0E08
initial_position = np.array([0,0,0])
initial_velocity = np.array([0.05*c,0,0])
gap = 0.1

timesteps = np.arange(0, 4.0E-7, 1.6E-10)
timesteps.shape

position_evolution = np.zeros((timesteps.size,3))
position_evolution.shape
```

The initial position is set at the point (0,0,0) and the initial velocity is given the value 0.05*c in the x direction. These are also set up as arrays. The time for the simulation and time steps was calculated using the time period formula for one oscillation (and was calculated based on the number of oscillations desired), and the steps were calculated based on the number of appropriate points over which the integration would be carried out.

```
def create_place_holder_array(timesteps) :
    ''' Returns a placeholder array '''

    return np.zeros((timesteps.size,3))
```

This next line of code shown above creates a function that we will use to create an empty array for the particle's position and velocity, which will be populated by the updated values.

Next we create the function to calculate the updated values of the particle's position and velocity.

```
def calc_trajectory_in_fields(electric_field_function, magnetic_field_function, get_rates_of_change_function,
                              timesteps, particle_position, particle_velocity, q = 1.67E-19, m = 1.67E-27) :

    # Create place holder arrays
    position_evolution = create_place_holder_array(timesteps)
    velocity_evolution = create_place_holder_array(timesteps)

    dt = timesteps[1]
    for inum, timestep in enumerate(timesteps) :
        # Populate
        position_evolution[inum,:] = particle_position
        velocity_evolution[inum,:] = particle_velocity

        # Calculate velocity and acceleration due to e- and b-fields to update particle_position and particle_velocity
        position_rate_of_change, velocity_rate_of_change = \
            get_rates_of_change_function(particle_position, particle_velocity, electric_field_function,
                                         magnetic_field_function, q, m, dt)

        # Update
        particle_position = get_updated_value(particle_position, position_rate_of_change, dt)
        particle_velocity = get_updated_value(particle_velocity, velocity_rate_of_change, dt)

    return position_evolution, velocity_evolution
```

This final function makes use of all the functions defined earlier to calculate the updated values of the particle's position and velocity. The '*create_placeholder_array*' function is used to create an array of zeros, which will be populated with the updated values of position and velocity.

Using the '*get_rates_of_change_function*' we calculate the rate of change of the particle's position and velocity (the velocity and acceleration). When running the '*calc_trajectory_in_fields*' function we can pick between inputting '*get_rates_of_change_euler*' or '*get_rates_of_change_runge_kutta*' to decide which method of analysis we wish to compute with.

With the newly calculated rates of change, we use the '*get_updated_value*', defined earlier in the code, to calculate the new values of the particle's position and velocity. These values are saved in '*position_evolution*' and '*velocity_evolution*'.

The next step is to calculate the new values for the particle's position and velocity through the function we just defined, using both the Euler method and the Runge-Kutta method. This is shown in the lines of code below.

```
trajectory_evolution, velocity_evolution = calc_trajectory_in_fields(constant_electric_x_field,
                                                                     constant_magnetic_z_field,
                                                                     get_rates_of_change_euler,
                                                                     timesteps, initial_position, initial_velocity)
```

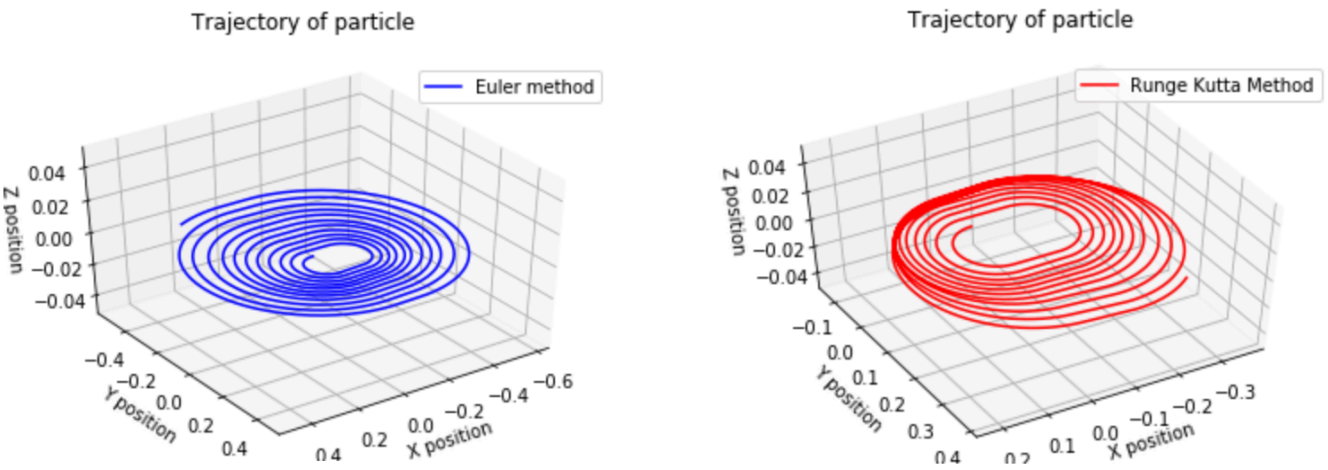
Euler's method

```
trajectory_evolution_2, velocity_evolution_2 = calc_trajectory_in_fields(constant_electric_x_field,
                                                                         constant_magnetic_x_field,
                                                                         get_rates_of_change_runge_kutta,
                                                                         timesteps, initial_position, initial_velocity)
```

Runge-Kutta method

Results and Conclusion

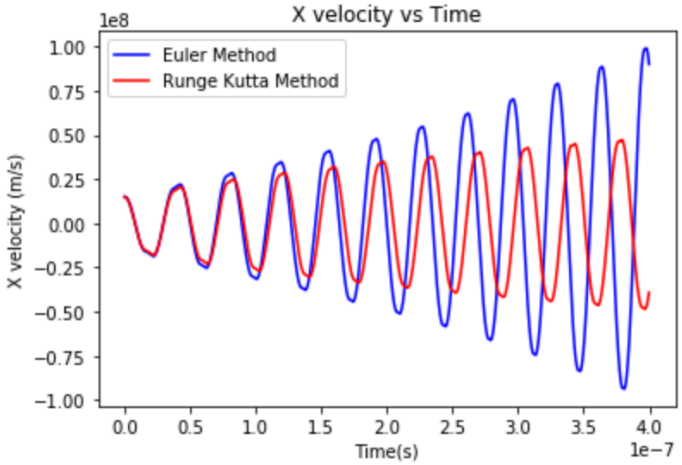
Shown below are the two trajectories of the particles, calculated using Euler's method and the Runge-Kutta method.



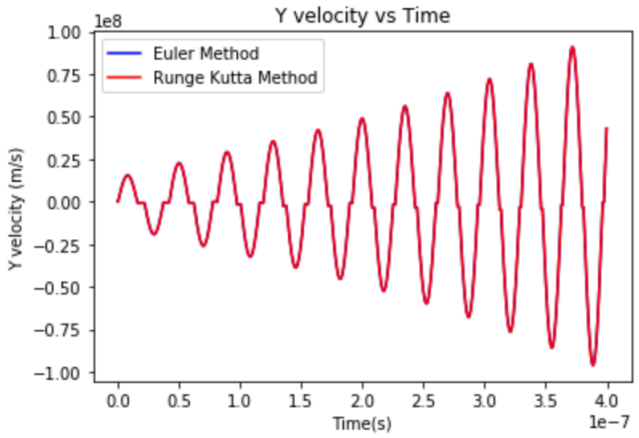
As expected, there is no displacement of the particle along the z direction. The magnetic field is oriented along the negative z direction and hence cannot apply any effective force on the particle in that direction. Also, the electric field acts along the x direction and only applies a force on the particle in that direction.

Both trajectories follow the outward spiral path. However, as is evident, the paths are not perfectly outward spiraling. In both plots the particle either spirals inward as for positive y-values in the first plot and negative y-values in the second plot or outwards as in negative x-values for both plots. This is a result of the limitations of the methods of numerical analysis used. Due to the degree of imprecision in the methods (four force evaluations per step) (Stoltz), the particle essentially does not conserve energy perfectly in the simulation. This is seen as the imperfections in the trajectory plot of the particle.

The parameter '*gap*' is also evident in the trajectories, seen as the straight line in the path. This is more evident in the Runge-Kutta plot. This is possibly due to the difference in precision of the two methods. The magnitude of the electric field is of the order 10^6. Since the Runge-Kutta method uses a time step that is half the value of that used in Euler's method, it has more precise measurements for the position and velocity values. However, due to the high order of magnitude of the electric field, it is possible that the effect of the electric force on the particle's velocity in the value of half a delta time = 0.5 * 1.6E-10 is very high. This can be further analyzed by seeing a plot of the x-velocity versus time.



The difference in the x-velocities of the particle calculated using the two methods increases with time. This is probably a result of the phenomenon explained above. Analyzing the y-velocity versus time plot for the particle also gives insight into this.



Since the magnitude of the magnetic field is defined to be only 2T, the effect of its force on the velocity of the particle in half a delta time is not as high relative to the effect of the electric field. This is why the plots for the y-velocities calculated using the two methods overlap. This result gives insight into the nature of different numerical approximation methods. When dealing with high orders of magnitude, numerical approximation methods such as Euler's method and even the Runge-Kutta method may yield results with non-trivial errors. The horizontal lines at y-velocity = 0 in the plot above represent the time during which the particle is being accelerated from one Dee to another.

A limitation of this simulation is that it does not take into account relativistic effects (Phalke). At such high energies, the particle experiences changes in mass, which lead to complications in the calculation of the particle's trajectory. In real world use of cyclotrons, such effects need to be accounted for in order to get the desired results. Additionally, this model makes use of a simplification of the electric field. The electric field is modeled to be constant and not oscillating. The effects of the oscillatory field are however accounted for in the definition of the acceleration function.

Having said that, this simulation does provide insight into the effectiveness for different computational methods of numerical analysis in various contexts. It is also a simple, yet effective model of a cyclotron and provides a good explanation of the working of such a particle accelerator.

Works Cited

Phalke, Vivek. “Cyclotron (Theory).” *YouTube,* uploaded by Impetus Gurukul PHYSICS Vivek Phalke, 10 November 2016, https://www.youtube.com/watch?v=98-ruTyK8uE.

Stoltz, P. H, et al. “Efficiency of a Boris-like Integration Scheme with Spatial Stepping.” *Physical Review Special Topics - Accelerators and Beams*, vol. 5, no. 9, American Physical Society, 9/2002, p. 094001, doi:10.1103/PhysRevSTAB.5.094001.

Marcrowo. “cyclotron.”*GitHub,* 21 July 2014, https://github.com/marcrowo/cyclotron/blob/master/cyclotron.py.

“Runge–Kutta Methods.” *Wikipedia*, Wikimedia Foundation, 14 Apr. 2020, en.wikipedia.org/wiki/Runge–Kutta_methods.