The Study and Simulation of the Movement of Charged Particles in a Magnetic Field

Caspian Berggren-Noren
caspian.noren@gmail.com

under the direction of
Assist. Prof. Dhrubaditya Mitra
Nordic Institute for Theoretical Physics

Research Academy for Young Scientists
July 11, 2012

Abstract

One of the greatest mysteries in physics today is cosmic rays and their speed close to that of photons. This means that they are one of the fastest particles we know of today and often have energy levels in range from 10^7 eV up to 10^{20} eV. Today, the leading theory behind the origin of a cosmic ray's velocity is a theory presented by Fermi. The theory depicts that the repelling of a moving "magnetic mirror" in interstellar space would be the cause for the high velocity of the cosmic rays. What this study aims to do is to find out if Fermi's theory is theoretically even possible. The study will therefore focus on determining if it is possible to make a charged particle bounce when it interacts with a magnetic mirror using computer simulations and theoretical physics.

1 Introduction

The study of high energy cosmic rays is of great scientific importance. It is likely to have a huge impact on the study of the universe. Cosmic rays originate from outer space, and the study of these particle can therefore give us a deeper understanding of the composition of the universe.

Cosmic rays usually consist of subatomic particles such as protons, electrons and alpha-particles. The rays often consist of 89% hydrogen nuclei, 10% helium and 1% heavier elements [1]. There has also been proven to be a power law describing the correlation between the cosmic rays' collision with Earth and its atmosphere and the rays energy levels. This power law is $N(E)dE \propto E^{-2.66}dE$ [2]. Since cosmic rays are subatomic particles they also have a specific electric charge and are thus affected by magnetic fields. One of the traits of cosmic rays is their unusually high speed, which is often close to the speed of light. This means that the particles often contain an energy in the interval of $10^7 - 10^{20}$ eV. This is to be compared to the highest energy levels detected in modern particle accelerators: 10^{13} eV [2]. The only known particle with a velocity larger than that of the cosmic rays is the photon, and so the origin of the large velocity of cosmic rays has puzzled physicists for generations [3].

There are theories trying to explain why cosmic rays have such a large velocity. In 1949 Fermi introduced a theory to describe how charged particles would accelerate while repeatedly being reflected by so-called "magnetic mirrors". He thought that this theory could be applied to cosmic rays and explain their incredibly high velocity. Large moving magnetized clouds would act as such magnetic mirrors and the cosmic rays would be reflected by their magnetic field, keeping their initial velocity and gaining the velocity from the cloud's movement. This theory has since played an important role in the development of astrophysics [2] [4].

The problem with this theory is that it's merely a theory. It has never been proven empirically that these "magnetic mirrors" are indeed the reason for the acceleration behind the cosmic rays' velocity [5]. What this study aims to do, however, is to perform computer simulations to predict the movement of a charged particle within a magnetic field. Based on Newtonian physics, the computer simulations attempts to numerically approximate the motion of this particle. These simulations can play an important part in either confirming or disproving Fermi's theory of the cosmic rays' acceleration.

2 Method

The purpose of the computer simulations was to investigate the path of movement of a charged particle in a non-uniform magnetic field based on Newtonian physics. Using an algorithm called the Euler method such a path can be approximated using relatively simple mathematical models.

Initially an equation to describe the movement of a charged particle in a uniform magnetic field had to be established. To do that, assume there is a particle A with an electric charge q, mass m and a velocity $\vec{V_1}$. Now assume the particle acts within a magnetic field $\vec{B}$ perpendicular to $\vec{V_1}$. The particle will be affected by a Lorentz force according to

$$\vec{F} = q * (\vec{V} \times \vec{B}) \tag{1}$$

. This means that $\vec{F}$ will be perpendicular to $\vec{V_1}$ and $\vec{B}$ and will therefore be a force acting towards the magnetic guiding centre, as depicted by figure 1 [6].

It can easily be shown that this equation will always lead to a circular orbit lying in the plane transverse to $\vec{B}$. Since $\vec{F}$ is always directed towards the magnetic field guiding centre

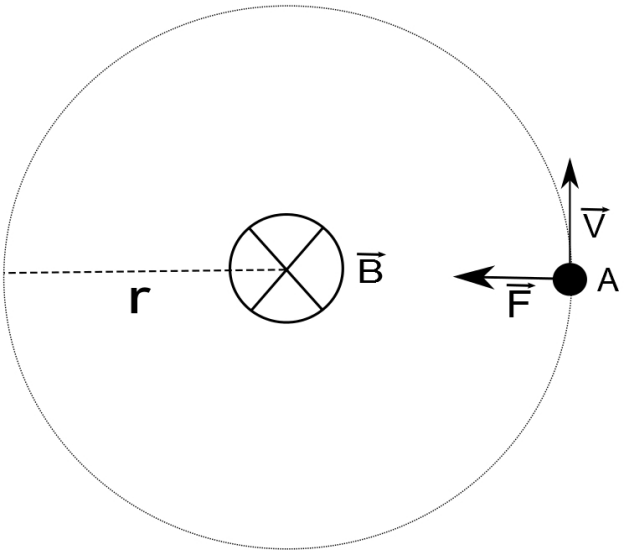

Figure 1: A charged particle in orbit around a magnetic guiding centre.

it can be described by an equation for centripetal force, $\vec{F} = \frac{mv^2}{r}$ where r refers to the radius of the orbit. With this equality r can be equal to

$$r = \frac{m * v_1}{B * q}. \tag{2}$$

If A has a component of velocity $\vec{V_2}$ parallel to $\vec{B}$, $\vec{F_2}$ will equal to zero and cause no inward force. Hence this component leads to a uniform translation of the circular trajectory with velocity $\vec{V_2}$, making the path of the particle helical as depicted in figure 2. Thus we can think of the motion of A as a result from a combination of two motions: a circular motion around the magnetic field guiding centre and a translatory motion of the guiding centre. According to the law of conservation of energy, the sum of the transverse kinetic energy and the longitudinal kinetic energy will be constant. Note that the kinetic energies are scalars and not vectors. The use of the words "transverse" and "longitudinal" are merely to describe the kinetic energy from the transverse and longitudinal motions [6].

$$E = \frac{m * V_1^2}{2} + \frac{m * V_2^2}{2} \tag{3}$$

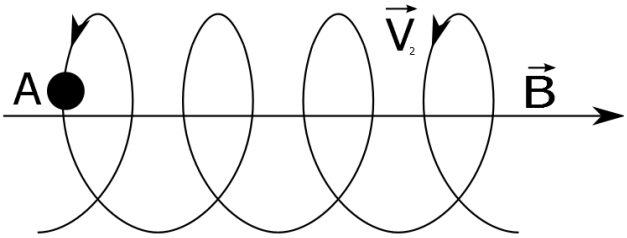

Figure 2: A figure of a charged particle in a helical motion around a magnetic guiding centre.

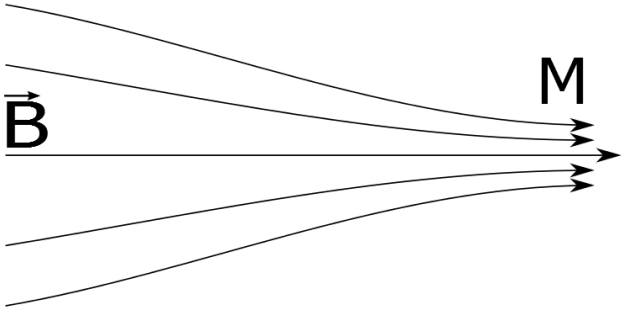

Figure 3: A non-uniform magnetic field with various magnetic field lines converging to a guiding centre.

Now assume the magnetic field to be non-uniform, in a way shown by figure 3. This would mean that the strength of the magnetic field varies along the central field line where the strongest area is at M where the field lines are as most compact. Now the vectors will have to be broken down into their coordinates to be able to proceed with the mathematical calculations. $\vec{B}$ only depends on the z-axis making its initial coordinates $(0, 0, B_0)$ and $\vec{V_1}$ has the coordinates (V_x, V_y, V_z). A's coordinates are (X, Y, Z). Since $\vec{F} = m * \vec{a}$, where $\vec{a}$ is the acceleration, a little reconfiguration in equation (1) gives the equation

$$\vec{a} = \frac{q}{m} * (\vec{V_1} \times \vec{B}). \tag{4}$$

The acceleration, however, is equal to the derivative of $\vec{V_1}$ whereas $\vec{V_1}$ is equal to the derivative of A. Therefore it is possible to calculate the coordinates for the acceleration through the coordinates of A given initial values for A, $\vec{V_1}$ and $\vec{a}$. This is a crucial step in order to approximate the path of a charged particle using computer simulations since computers can not solve second order differential equations.

We can divide the equation (4) into two parts, namely

$$\frac{\mathrm{d}\vec{P}}{\mathrm{d}t} = \vec{a} \tag{5}$$

and

$$\frac{\mathrm{d}\vec{P}}{\mathrm{d}t} = \frac{q}{m} * (\frac{\mathrm{d}A}{\mathrm{d}t} \times \vec{B}) \tag{6}$$

.

Since the magnetic field is non-uniform, equations for the changes of $\vec{B}$'s coordinates are needed. Through mathematical derivations such equations can be acquired from using the coordinates of A.

$$B_x = -\frac{1}{2} * \frac{\xi * X}{X^2 + Y^2} \tag{7}$$

$$B_y = -\frac{1}{2} * \frac{\xi * Y}{X^2 + Y^2} \tag{8}$$

$$B_z = -\xi * Z + B_0 \tag{9}$$

ξ is a constant that describes how the magnetic field lines converge.

With equations (5), (6), (7), (8) and (9) it's possible to use the Euler method to approximate the charged particle's, trajectory in a non-uniform magnetic field. The algorithm is useful for approximating ordinary differential equations when the initial values of the parameters are known. With these initial values, since part of the method includes calculating differential equations, the derivative and slope to a tangent in that inital point can be calculated. With a defined stepsize, h, the Euler method approximates curves using the step in the direction as the tangent. Since there is now new coordinates for the parameters, a new derivative can be calculated and the process iterates itself for as long as is needed. It should be noted that the longer the process goes on the larger the error rate will be, but can be compensated with a smaller h [7].

3 Results

In the simulations various velocities and strengths on the magnetic fields were tested to see if and how particles would bounce. The data showed that a combination of a strong magnetic field and a small initial velocity gave plots of a particle going in one direction and then suddenly changed to the opposite direction when it reached a certain point. In this plot the stepsize was chosen to be $h = 0.002$ and $q/m = 1$ since these constants won't affect the result of a theoretical bounce. This can be compared to another plot where only the shape of the magnetic field is interesting. Therefore the simulation let $\vec{B}_z$ depend only on Z, and a value c which can be translated into the size of the magnetic field.

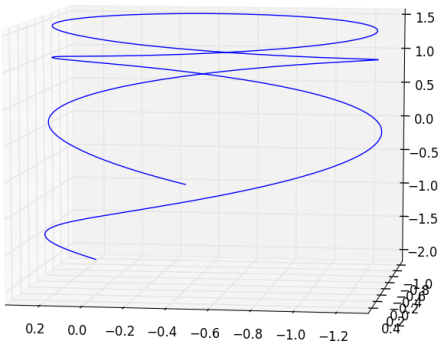

Figure 4: A plot showing the trajectory of a charged particle with a small velocity within a strong magnetic field.

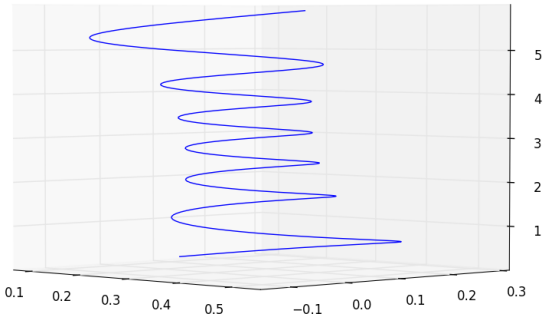

Figure 5: A plot showing the shape of a non-uniform magnetic field.

4 Discussion

The path of the particle was an expected trajectory. Mathematically, you can quite easily draw a conclusion that the particle would bounce if it had a low velocity and a strong magnetic field using the following reasoning: since the orbital radius depends inversely on the strength of the magnetic field, as shown in (2), the radius will become smaller as $\vec{B}$ gets stronger. The transverse velocity $\vec{V}_1$ will then increase as the radius becomes smaller. As E has to be constant, the longitudinal kinetic energy would have to decrease. If the magnetic field gets too strong and A has a small kinetic energy E the particle is going to be reflected by the magnetic field since the particle does not have sufficient enough energy to be able to pass through the critical point M as shown in figure 3. This does, however, mean that Fermi's theory of acceleration has a solid ground of confirmation in the theoretical aspect. What this study has shown is that not only can these magnetic mirrors be visualized by plots, as shown in figure 5 but the actual path of the charged particle can be projected and approximated. This method can therefore be applied and developed in the future to approximate a more exact path and give us a deeper understanding of the reason behind the extreme velocities of the cosmic rays.

One of the main problems with this study is that it has only used the Euler method to approximate the curves. The Euler method is notorious for being quite inexact, especially if the calculations continue for large numbers of iterations. Since this study only used a small numbers of iterations meaning that the error will not be so large. One of the main focus of future researches should consequently develop more exact simulations of the type this study has performed, preferably using the Runge-Kutta method.

Future research should also introduce new scenarios such as moving magnetic mirrors placed randomly in a three-dimensional space with several charged particles in the simulation. This

could lead to confirming that Fermi's theory can explain the power law relation between particles and their energy levels mentioned earlier in this article.

5 Acknowledgments

I would to thank *Dhrubaditya Mitra*, my incredible mentor, who had the patience to stay with me for two weeks and explain even the simplest vector operations, for the help and mentorship. I would also like to thank *NORDITA*, for the several liters of coffee and milk, and for the stylish and functional office we could use during our stay. None of this would have been possible without *Rays**, a huge thank you to all the Rays leaders for the incredible amount of time and effort put down to give me the opportunity to write this article. I would even like to thank *Teknikföretagen*, whom without I wouldnt have been able to complete this study. Last, but not least, I would like to thank *Europaskolan*, for the wonderful living quarters and facilities.

References

[1] NASA. The OWL Mission of the most energetic;.

[2] a D Erlykin, a W Wolfendale. Cosmic rays: the centenary of their discovery. Europhysics News. 2012 Apr;43(2):26–28. Available from: http://www.europhysicsnews.org/10.1051/epn/2012205.

[3] Hillas A. The origin of ultra-high-energy cosmic rays. Annual Review of Astronomy and Astrophysics. 1984;Available from: http://adsabs.harvard.edu/full/1984ARA\&A..22..425H.

[4] Fermi E. On the Origin of Cosmic Rays. 1949;105(1943):4–9.

[5] Press WH, Teukolsky SA, Vellerling WT, Flannery BP. Numerical Recipies; 1993.

[6] Choudhuri AR. The Physics of Fluids and Plasmas; 1998.

[7] Gonze D, Gillespie DT. Numerical methods to solve Ordinary Differential Equations. 2009;.

A Appendix

```
class problem():


        def __init__(self):
                Configuration values for the actual differential equation
                self.particle_charge = 1
                self.particle_mass = 1


        def v_prim(self, direction, vector_v, vector_b):
                if(direction == 'x'):
                        return (self.particle_charge/self.particle_mass)*(vector_v['y']*
                if(direction == 'y'):
                       print "acceleration in y", (vector_v['x']*vector_b['z'] + vector_
                        return (self.particle_charge/self.particle_mass)*(vector_v['z']*
                if(direction == 'z'):
                        print "acceleration in z", self.particle_charge/self.particle_ma
                        print "accel_part1", vector_v['x']*vector_b['y']
                        print "accel_part2", vector_b['x']
                        return (self.particle_charge/self.particle_mass)*(vector_v['x']*


        def particle_pos(self, current_pos, v_prim, step_size):
                print "old", current_pos
                print v_prim
                for component in current_pos:
                        current_pos[component] = current_pos[component] + (v_prim[compon
```

```
                print "updated", current_pos
                return current_pos

class solver():

        def __init__(self, step_size, target_x, problem):

                Be nice and put the vectors right in here:
                self.vector_pos = {'x' : 1, 'y' : -1, 'z' : 0.001}
                self.vector_v = {'x' : (1-(0.5*0.5)), 'y' : (1-(0.5*0.5)), 'z' : 0.2}
                self.vector_v_next = {'x' : None, 'y' : None, 'z' : None}
                self.vector_b = {'x' : 0, 'y' : 0, 'z' : 15}

                #Config stuffs
                self.step_size = step_size
                self.target_x = target_x
                self.handles = {}
                self.handles_read = {}
                self.problem = problem
                self.debug = False

        def load_handle_write(self,file_name):
                if not file_name in self.handles:
                        if(os.path.exists(file_name)):
                                os.remove(file_name)
                        #Open file/files for saving
```

```
                self.handles[file_name] = open(file_name, 'wb')
        return self.handles[file_name]

def load_handle_read(self,file_name):
        if not file_name in self.handles_read:
                #Open file/files for saving
                self.handles_read[file_name] = open(file_name, 'rb')
        return self.handles_read[file_name]

def load(self, file_name, debug):
        if(debug and self.debug):
                print "Loading values from:", file_name
        values = self.load_handle_read(file_name + '.txt').readlines()
        self.load_handle_read(file_name + '.txt').seek(0)
        for i, value in enumerate(values):
                #print float(value.rstrip('\n'))
                values[i] = float(value.rstrip('\n'))
        return values

def save(self, file_name, value, debug):

        if(debug and self.debug):
                print "Saving to file [\"" + debug + "\"]:", value
        self.load_handle_write(file_name + '.txt').write(str(value) + "\n")

def euler(self):
```

```
#No of steps to take!
no_of_steps = int(self.target_x/self.step_size + 2) # +1 for range, +1 f

print "\nRunning " + str(no_of_steps) + " no. of steps!"

#       Set m
m = 15
c_value = 9

for i in range(no_of_steps):

        if i < 1:

                #First step doesn't need to be calculated
                pass

        if i > 0:

                #       Update magnetic field lines to create "convergen
                self.vector_b['z'] = -((self.vector_pos['z'] - c_value)*
                if self.vector_b['z']<0:
                        self.vector_b['z'] = 0
                        self.vector_b['x'] = 0
                        self.vector_b['y'] = 0
                else:
```

```
        self.vector_b['x'] = -(0.5) * (m * self.vector_p
        self.vector_b['y'] = -(0.5) * (m * self.vector_p


#       For each component in the speed, update it using
for component in self.vector_v:

        #       Calculate v value from it's derivative w
        acceleration = self.problem.v_prim(component, se
        self.vector_v_next[component] = self.vector_v[co

#       Update particle position based on velocity and p
self.vector_pos = self.problem.particle_pos(self.vector_

#       Save all known data for future plotting, this co
#               spent writing data to disk.
self.save('part_speed_x', self.vector_v['x'], 'speed_x')
self.save('part_speed_y', self.vector_v['y'], 'speed_y')
self.save('part_speed_z', self.vector_v['z'], 'speed_z')

self.save('part_pos_x', self.vector_pos['x'], 'pos_x')
self.save('part_pos_y', self.vector_pos['y'], 'pos_y')
self.save('part_pos_z', self.vector_pos['z'], 'pos_z')
self.save('time', i, 'time')
self.save('magnetic_vector_bz', self.vector_b['z'], 'mag
self.save('magnetic_vector_bx', self.vector_b['x'], 'mag
```

```
                        self.save('magnetic_vector_by', self.vector_b['y'], 'mag

                        #       Update old speed to new speed.
                        self.vector_v = self.vector_v_next

        for i, handle in self.handles.items():
                handle.close()

def plot_show(self):
        #Let's show the graph as well
        plt.show()

def plot_3d(*args):
        #Initialize a new graph
        figure = plt.figure()
        figure_3d = Axes3D(figure)
        #Need at least x and y values to begin!
        if(len(args) < 4):
                print "Error - please provide at least three variables to plot!"
                sys.exit()

        #Manually define self since we are running with a variable no of variabl
        self = args[0]
        #x_val = self.load(args[1], True)

        for i, arg in enumerate(args):
```

```
                if i > 0:
                        if (i-1) % 3 == 0:
                                valuex = self.load(args[i], True)
                                valuey = self.load(args[i+1], True)
                                valuez = self.load(args[i+2], True)

                                #Debug info
                                #if(self.debug):
                                #       for i, x in enumerate(x_val):
                                #               print "Plotting: ", x_val[i], va

                                #Plot each value-set
                                figure_3d.plot(valuex, valuey, valuez)

def plot(*args):
        #Initialize a new graph
        plt.figure()

        #Need at least x and y values to begin!
        if(len(args) < 3):
                print "Error - please provide at least two variables to plot!"
                sys.exit()

        #Manually define self since we are running with a variable no of variabl
        self = args[0]
        x_val = self.load(args[1], True)
```

```
        for i, arg in enumerate(args):
                if i > 1:
                        values = self.load(arg, True)

                        #Debug info
                        if(self.debug):
                                for i, x in enumerate(x_val):
                                        print "Plotting: ", x_val[i], values[i]

                        #Plot each value-set
                        plt.plot(x_val, values, '.-')

def custom_vars(*args):
        self = args[0]

        if(len(args) < 4):
                sys.exit('Error')

        # We are given variable names, load them from the files
        pass_args = []

        for arg in args[3:]:
                pass_args.append(self.load(arg, True))

        for var_no, var in enumerate(self.load(args[3], True)):
```

```
                        # We only want to give one value per "variable" on in each itera
                        temp_pass_args = []

                        for arg_no, arg in enumerate(pass_args):
                                temp_pass_args.append(pass_args[arg_no][var_no])

                        # Run the custom function
                        self.save(args[1], args[2](*temp_pass_args), 'Adding custom valu

                for i, handle in self.handles.items():
                        handle.close()

#Solve this using two euler approximations at the same time. Predict first derivative th
#Initial values - x = 0, x' = 1, x'' = 1

# Usage of custom functions:
# 1) Define custom function with variables you want from previous data
# 2) Run custom_vars(), with arg1 -> name of variable you want to save it to, arg2 -> na
#               arg3 -> the "x" value you want to iterate with, can be any value, arg(3+
#               want to use in your custom function

#Config
step_size = 0.001

#Run it
```

```
problem = problem()


#Run it for several stepsizes


print "\nDoing stepsize:", float(0.2)
solver_instance = solver(step_size, 20, problem)
solver_instance.euler()
solver_instance.plot('time', 'part_speed_z', 'magnetic_vector_by', 'magnetic_vector_bx')
solver_instance.plot('part_pos_z', 'magnetic_vector_bz')
solver_instance.plot_3d('part_pos_x', 'part_pos_y', 'part_pos_z')               #Plot ou


#Close all open handles after we've shown the graphs!
try:
        solver_instance.plot_show()
except:
        print "Quitting!"
        for i, handle in solver_instance.handles.items():
                handle.close()
        for i, handle in solver_instance.handles_read.items():
                handle.close()
```