Modeling the Path of a Charged Particle in a Magnetic Mirror

Eyvind Niklasson
eyvind.niklasson@gmail.com

under the direction of
Prof. Dhrubaditya Mitra
Nordic Institution for Theoretical Physics (NORDITA)

Research Academy for Young Scientists
July 11, 2012

Abstract

Fermi's theory of the motion of charged particles in a magnetic mirror has existed for several decades, often being cited and used in further studies in the field. The theory builds on the ability of astronomical bodies in space to generate a converging magnetic field, which simultaneously increases in strength. A particle colliding with such a field will either pass through or reflect back, depeding on the angle of attack - i.e. from which direction it approaches the mirror. The theory also covers how particles can enter a loop between two such magnetic mirrors, bouncing back and forth. If these mirrors are advancing towards each other each collision slightly increasing the energy of the particle. This theorized phenomenon is also used to explain the observed exponential correlation between energy level and frequency of particle impacts on earth.

This study examines the path of a particle when colliding with a simulized, generalized, magnetic mirror. It shows the path the particle takes during the reflection or while passing through. While past research exists on this topic, it has been done on mathematically proving the reflection effect. This study provides a novel model for showing the exact path of the particle up until, and even after the actual reflection, by providing a working computer simulation model. The model is both extendable and scalable for future researchon phenomena related to reflection, as well as for more complex simulations.

1 Introduction

Cosmic rays were discovered approximately one hundred years ago. *Cosmic ray* is the counter-intuitive name given to highly charged particles which impact the earth at regular intervals. They were first discovered using electrometers by Theodor Wulf, who measured radiation at the base and the top of the Eiffel tower, but it was not until Victor Hess brought with him electrometers on an air balloon that real evidence for the extra-terrestrial origins of the particles was obtained [1].

Today it is known that these charged particles largely consist of protons (roughly 90%) with the remainder being simple atomic nuclei and electrons. Over the past century, research has shown that these particles come not only from the outer depths of space, but are even created in the sun[1]. An inverse exponential relationship has also been observed between the number of impacts per year on the surface of the Earth and the energy level of the particle (Figure 1). This relationship can even be approximately described in the power law: $N(E)dE \propto E^{-2.66}dE$ [2].

The exponent value has been experimentally determined using data collected from terrestrial particle telescopes [5].

While uncommon, particles with energy levels high above the levels artificially achievable - sometimes factors of ten or higher - also impact the surface of Earth [1]. It is still not known for certain how such particles with extremely high energy levels are created or accelerated, but the leading theory that has existed since 1949 is that of Fermi [1].

Fermi's theory builds on the existence of so-called magnetic mirrors in space. Magnetic mirror in the context of space is a term used to describe large astronomical clouds of magnetic fields of decreasing and increasing strength [3]. In such a homogenous magnetic field, if magnetic field lines decrease or increase in strength, charged particles will follow a path of a corkscrew towards a single point. This phenomenon can be described using the equation for

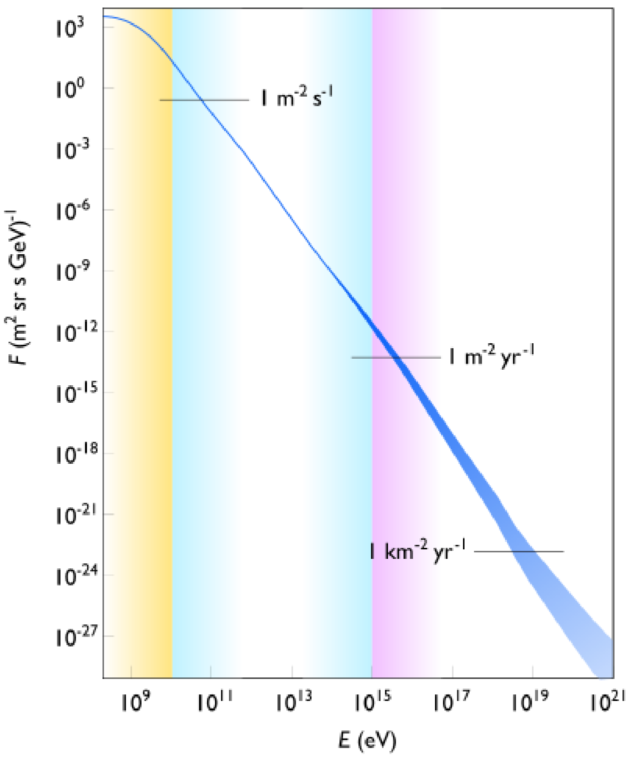

Figure 1: A graph showing the relationship between flux and energy levels in incoming particles. An exponential power-law relationship can be observed, showing how higher energy particle events are much less frequent than those of a lower energy.

Lorentz force $\vec{F} = q(\vec{v} \times \vec{B})$ [3]. When they reach this point, the particles are reversed and spiral back out; thereby giving such magnetic clouds the function of a mirror when charged particles hit the surface. The key difference from a conventional mirror is that no energy is lost in the collision; if the particle collided with any other astronomical body then energy would be inherently lost [3].

This lossless, reflective property of magnetic mirrors is what makes Fermi's theory feasible. When a particle reflects off a magnetic field which has a velocity in any direction, it will either gain or lose energy, depending on the direction of the cloud - i.e. if the particle hits a cloud going in the opposite direction to the particles direction, it will gain energy. With certain configurations, a particle can even become stuck between two such magnetic clouds, which are moving towards each other [3]. In such a situation, an energy increase can occur over long periods of time, gradually increasing the energy of the particle. At a certain energy level, it has been shown that particles can even penetrate such a magnetic mirror, continuing on their path until they collide with another body, such as the Earth [1]. As AM Hillas shows in his publication The Origin of Ultra-High-Energy Cosmic Rays, the power law can even be mathematically approximated using the simple model of Ta as the time spent accelerating and Tb as the average time before escape from a system of astronomical bodies capable of producing magnetic mirrors[5]. Despite the fact that the values Ta and Tb can only be estimated, they inherently describe the power law that has been experimentally observed with $E^{Ta/Tb+1}$ [5].

In other words, with the correct conditions, these assumptions can describe the range of impacts observed on earth [5].

Another theory describes the acceleration of the particles as a result of astrophysical shock waves. Astrophysical shock waves are streams of fast-moving matter emitted from astronomical events, such as supernovae [9]. In layman's terms, the particles ride these streams to accelerate to higher energy levels. While this theory is often cited and used as a basis for further research, as of today there is no explanation in the context of the theory for the aforementioned power law present in the incoming particles [5].

The study of these particles is imperative for humanity's success with long term space programs, due to the disruptive effects of the particles. Normally, the Earth's large magnetic presence, coupled with an atmosphere capable of absorbing many of the particles, shields the biosphere from charged particles. Such conditions are not present in space, and the particles pose a risk to all space-travelers as of today, and even to equipment sensitive to interference [8].

The aim of this study is to model the path of charged particles in a converging magnetic field, like that of a magnetic mirror. The results will be applicable to a wide range of further studies, such as more complex models of magnetic mirrors. The model - emphapproximator

- used is written as computer code which numerically solves complex differential equations using Euleur's method [7]. To obtain an estimation of error rates as well as proper configuration values, the approximator is also run on more conventional problems, including a swinging pendulum.

2 Method

2.1 Euler's method for approximating differential equations

Differential equations are widely used tools in modern physics, and no less in astrophysics and particle physics. More often than not, these equations need to be solved using computer approximation, as analytical integration is impossible. Approximation will always result in a certain margin of error in the result, but in many cases this margin is so small it is not relevant for the study.

Differential equations come in several forms, and can be of varying complexity. An ordinary differential equation is the term given to differential equations which only contain one function and one or more of the functions derivatives. Partial differential equations, as opposed to ordinary differential equations, contain several variables and thus add a new level of complexity in both the solving and approximation of the problem [4]. In the context of this study, differential equations and ordinary differential equations will be treated as analogous. Much like polynomial equations, differential equations can also be given grades, with the grade being the so-called level of the derivative; a differential equation involving the second derivate would be a second-grade differential equation [4]. Another important property of a differential equation problem is whether it is an initial value problem or a boundary problem. An initial value problem involves a problem where a starting point is known, and an approximation is to be made from this starting point [7]. Likewise, a boundary problem is a problem where two or more values are known, and both are used to determine remaining values between these two boundaries [7].

One of the most widespread methods used to approximate differential equations is Euleur's method. Solving a differential equation with Euleur's method involves splitting the equation into several first-grade equations, which can then be parsed by the code. While other methods exist (such as the Runge-Kutta method), for the purposes of this study Euleur's method was chosen for its simplicity.

Euleur's method works on the principle of extrapolating from existing data - often an existing point. When solving an initial value problem, at least two pieces of what can be referred to as information about the problem need to be known - often the first derivative of the function and the initial value of the function [7]. It is also possible to solve equations where the second derivative of function is known and the initial values for the function and its first derivative are known. In the former scenario, one would start at the initial point, and then follow the derivative's path forward with a certain small interval of x - the step-size. At the next point, the derivative would be taken again, and the process repeated, until enough steps have been computed. There is always a risk for having minute changes in the function which are smaller than the step size, but aside of decreasing the step size (and thereby increasing computation time), there is no way to compensate for such irregularities [7].

2.2 Implementing Euleur's method

Implementing Euleur's method in a program language is often quite trivial, but handling the data it outputs is not as easy. Huge amounts of data are returned by the algorithm, and in an ideal implementation the size of the data should not affect the speed of the calculation. In the implementation used for the study, several extra features were also added, such as an API to add custom functions to handle the existing data. This API can be used for further research, with the main purposing being a way to find correlation between results by being able to quickly add and run code on existing, saved, data. The saving algorithm is independent of the rest of the code, and can even be used to let the program continue from a certain point if it crashes. A visualization of the code and its features can be seen in Figure 9 in Appendix A.

2.3 Determining accuracy

Determining error rates for approximations is critical for a valid result. The error rate will inherently depend on the size of the steps used in the program - generally speaking the larger the steps the larger the errors. Unfortunately, Euleur's method can also easily introduce a cumulative error, meaning in some cases longer simulations will lead to larger error margins [7]. In most implementations of Euleur's method, the step size is variable and can be changed before each simulation. To determine the error rates at different step sizes and different approximation methods, initially a problem with similar properties is chosen, albeit a problem which can be analytically solved [7]. An approximation can then be made and compared to the known analytical solution of the comparison problem to determine the error margin. For the purposes of this study, a differential equation describing a harmonic oscillating body was used, and then compared to the known solution (which in this case takes the form of a sinus curve).

2.4 Modeling path of particle

Before approximating the path of a particle the differential equations to be used needs to be determined, and these can be derived from simple rules describing magnetic field lines and their effects on particles. The effect of a magnetic field on a particle can be described using Lorentz's force; but there exist two separate magnetic forces in a in a magnetic cloud: a main magnetic force pointing in the same direction as the field lines and a lesser magnetic force implementing the convergence of the field lines as the magnetic field increases in strength.

These two expressions can be used to derive the differential equation in Figure 2 [5]. The various variables used in the derivation are defined in Figure 4. For the purposes of the study, the magnetic field was directed along the Z axis in all simulations, thereby simplifying the equation for the B_r (converging magnetic force) scalar.

$$m \cdot \frac{d\vec{v}}{dt} = \vec{A_p} = \vec{F} = q(\vec{V} \times \vec{B})$$
$$(A_p)_x = \frac{P_q}{P_m} \cdot ((V_p)_y \cdot B_z - (V_p)_z \cdot B_y)$$
$$(A_p)_y = \frac{P_q}{P_m} \cdot ((V_p)_x \cdot B_z - (V_p)_z \cdot B_x)$$
$$(A_p)_z = \frac{P_q}{P_m} \cdot ((V_p)_x \cdot B_y - (V_p)_y \cdot B_x)$$
$$\frac{d\vec{P}}{dt} = \vec{V_p}$$

Figure 2: The differential equations that was solved in the model

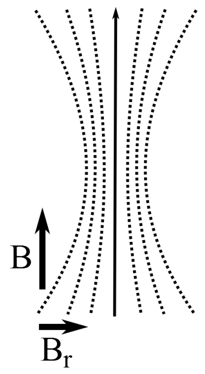

Figure 3: A visualization of the magnetic field lines, without magnitude increase of $\vec{B}$ in the center represented.

The layman's explanation of the equations can be expressed as that the particle's motion is affected by the two magnetic vectors B_r and $\vec{B}$ as well as the particle's current velocity. In turn, the magnetic vector $\vec{B}$ is determined based on the current position of the particle - in essence the when viewed from particle's reference frame, the magnetic field changes depending on the spatial coordinates of the particle. This effect occurs due to the converging effect of the magnetic field lines. The first constant, m, then decides how much this convergence affects the particle's motion. The second constant, c, is incorporated into the equations as a value which decides the size of the magnetic cloud the particle is entering - both its width and its length. A visualization of the converging magnetic field present in the simulation can be seen in Figure 3.

$$B_r = -\tfrac{1}{2} \cdot \tfrac{1}{r} \cdot \tfrac{dB_z}{dP_z}$$
$$B_z = -(P_z - c)^2 + c^2$$
$$B_x = -\tfrac{1}{2} \cdot \tfrac{mP_x \cdot (P_z - c)}{((P_x)^2 + (P_y)^2)}$$
$$B_y = -\tfrac{1}{2} \cdot \tfrac{mP_y \cdot (P_z - c)}{((P_x)^2 + (P_y)^2)}$$

Figure 4: Definition of various variables used in the model, as well as the equations for the magnetic fields present in the simulation.

$\vec{P}$ ***is a vector pointing to the position of the particle at any given time.***

$\vec{V_p}$ ***is a velocity vector describing direction and magnitude of the particle's velocity.***

$\vec{A_p}$ ***is an acceleration vector describing the magnitude and direction of the particle's acceleration.***

r ***is the particle's distance from the Z axis (specifically Z since the study employs a model where the magnetic field points in the Z axis).***

m ***is a constant which describes to what degree the*** B_r ***scalar affects the motion of the particle.***

P_q ***is the charge of the particle, set to one for the purposes of the study***

P_m ***is the mass of the particle, set to one for the purposes of the study.***

c ***is a constant describing the magnetic cloud in the simulation***

3 Results

3.1 Path of the particle

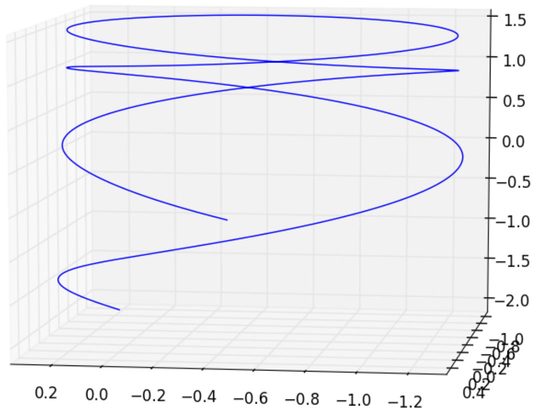

Figure 5: A visualization of a particle being reflected off of a magnetic mirror.

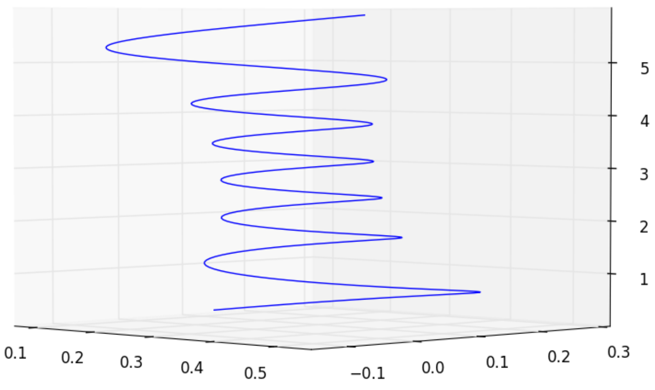

Figure 6: A visualization of a particle passing through a magnetic cloud.

The simulation was run multiple times with varying values of m and c at a step-size of 0.02, until a reflection was observed. As the purpose of the study was to show that reflection can occur, the values were chosen rather than taken from existing data, although the variables that were changed were all variables which naturally occur within a large range. As can be logically deduced from the differential equations, the simulation also produced a constant energy level for the particle in question, with no relevant acceleration or deceleration.

3.2 Accuracy

A model for an oscillating body was used to check error rates with various step sizes. The model used was a differential equation describing the motion of a pendulum, depending on initial speed and position. The equation for the oscillating body was derived from Newtonian laws (without accounting for friction), which are represented and detailed in Figure 7. The differential equation to solve became thus

$\frac{d^2y}{dx^2} = -x$

$\frac{dy}{dx} = 1$

$f(0) = 0.$

The differential equation was then used to obtain the relationship between step size and error margin. This was graphed by comparing the total energy in the system to the analytical constant describing to the total energy. As is seen in Figure 8, when the step size increases exponentially, the error size likewise decreases exponentially. Actual values of error can be seen in table 1. Based on these findings, simulations on the particle were run at a step-size of 0.02.

Table 1: Error margins with various step-sizes

step-size	***largest deviation***
0.2	$0.1714 \cdot 10^-1$
0.02	$0.1703 \cdot 10^-3$
0.002	$0.1703 \cdot 10^-5$
0.0002	$0.1703 \cdot 10^-7$

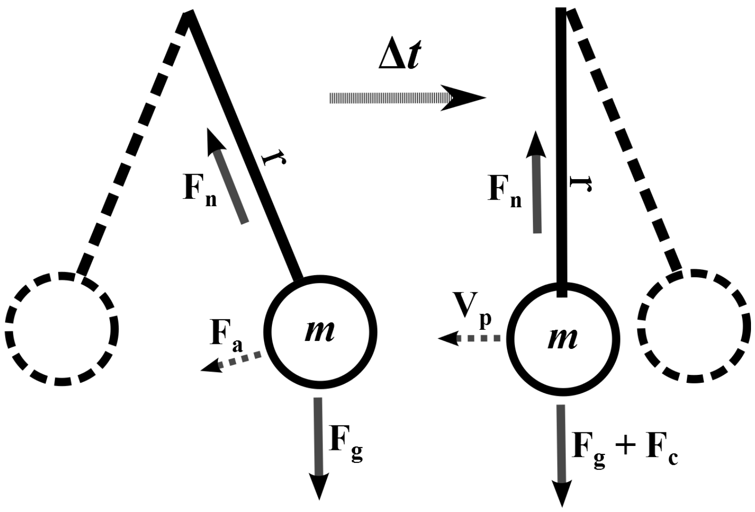

Figure 7: In the initial state (far left), the pendulum has reached its peak position, and is currently changing the direction of its velocity. The only force acting on it is F_g, since the string tension is 0 (at this point the mass at the end of the string is essentially weightless - there is no centripetal force $F_c = 0.5 \cdot \frac{v^2}{r}$ acting on it and its velocity has just balanced out F_g). In the illustration to the right the velocity Vp has peaked, and the centripetal force F_c is also at its maximum, with all the potential energy from the previous state having been converted to kinetic energy. With the absence of friction, the body continue keep oscillating until it is disturbed, and the system will contain the same amount of energy throughout the entire oscillation period - the energy will only be transferred between states.

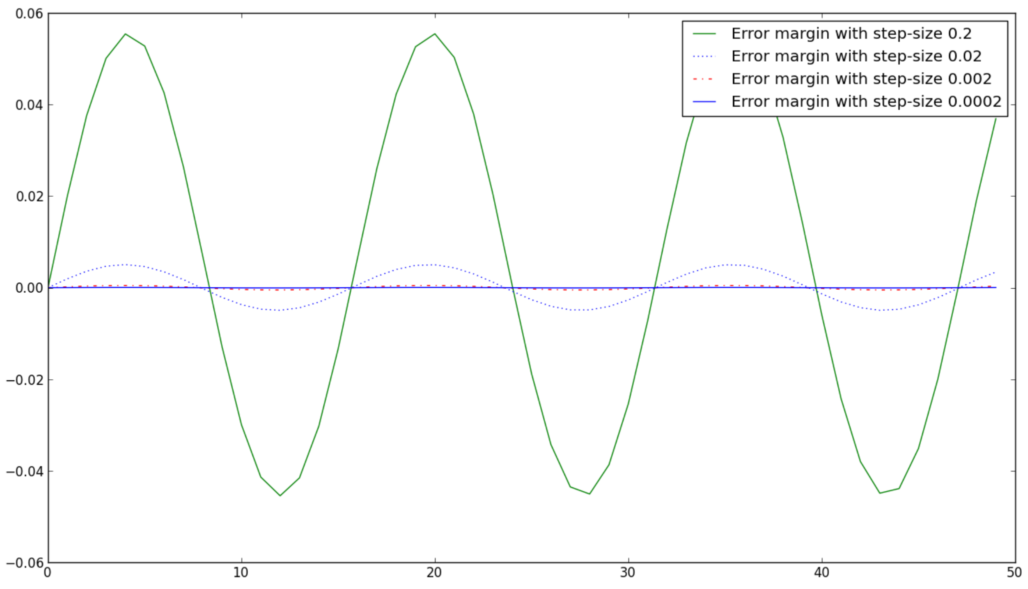

Figure 8: Graph of deviation in error in simulation with different step-sizes.

4 Discussion

4.1 Euleur's method

The oscillating body approximation used to determine the error rate for the step size was chosen simply because the analytical solution to the problem resembled a sinus curve. This is relevant because motion of a particle in the various component directions when corkscrewing will, logically, also resemble some form of a sinus wave. In spite of this similarity, there is still no proof or evidence that the margin of error in the final simulation correlates to the error margin observed in the simulation of the oscillating body - in essence - an assumption was made. Despite these unchecked margins of error, the study has still fulfilled its purpose - to model the reflection from a magnetic mirror. Since the data is currently not being used in any further extrapolation, the accuracy is not of key importance.

4.2 Path of the particle

Fermi's method is often criticized for being based off very ad-hoc assumptions, despite it being the most accepted theory. [2] While there already exists mathematical proof, or at the least mathematical evidence for his theory, visualization of this evidence is applicable in many fields. This study takes the first steps towards modeling the actual reflection of a particle, which, while already having been shown mathematically, still provides a solid ground for further research, as well as a deeper, visual, insight into the actual process of reflection. Further research is also facilitated by the model's extreme extendibility.

4.3 Future studies

The existing model is easily extendable to introduce further variable conditions, and even new scenarios. This is why the next obvious step is to implement motion in the magnetic clouds, as well as an algorithm for placing these clouds pseudo-randomly in a virtualized implementation of space. Multiple particles could then be positioned with varying speeds and charges, and a longer simulation could be run to determine which particles eventually increase in charge. The reflection process may even be simplified to ease on processor time (computation time). The results from this simulation can then be easily correlated to the currently terrestrially observed power law relationship between particles and their energy levels. If these results are sufficiently close, they could be used as further evidence for Fermi's theory [2].

Another possible study based on this model is a study investigating the motion and energy of a particle in a changing magnetic field. Logical deduction leads to the conclusion that the energy level of the particle should either increase or decrease, since a certain force is being excerpted on it as the magnetic field is changing. Such *chaotic magnetic fields* exist naturally in many astronomical events, such as supernovae [6].

5 Acknowledgements

I would to thank *Dhrubaditya Mitra*, my incredible mentor, who had the patience to stay with me for two weeks and explain the 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 wouldn't 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] a.D. Erlykin and a.W. Wolfendale. Cosmic rays: the centenary of their discovery. *Europhysics News*, 43(2):26–28, April 2012.

[2] Arnab Rai Choudhuri. *The Physics of Fluids and Plasmas: an introduction for Astrophysicists*. The Press Syndicate of the University of Cambridge, 1998.

[3] Enrico Fermi. On the Origin of Cosmic Rays. 105(1943):4–9, 1949.

[4] Didier Gonze and Daniel T Gillespie. Numerical methods to solve Ordinary Differential Equations. 2009.

[5] AM Hillas. The origin of ultra-high-energy cosmic rays. *Annual Review of Astronomy and Astrophysics*, 1984.

[6] Martin Lemoine. Transport of Cosmic Rays in Chaotic Magnetic Fields. 2008.

[7] William H. Press, Saul A. Teukolsky, William T. Vellerling, and Brain P. Flannery. *Numerical Recipies*. 1993.

[8] E.S. Simopoulos. The Natural Radiation Environment VII: VIIth Int. Symp. On the NRE. 7:894.

[9] Frank M Rieger Valent and Bosch-ramon Peter Duffy. Fermi acceleration in astrophysical jets. 1949.

A Appendix A

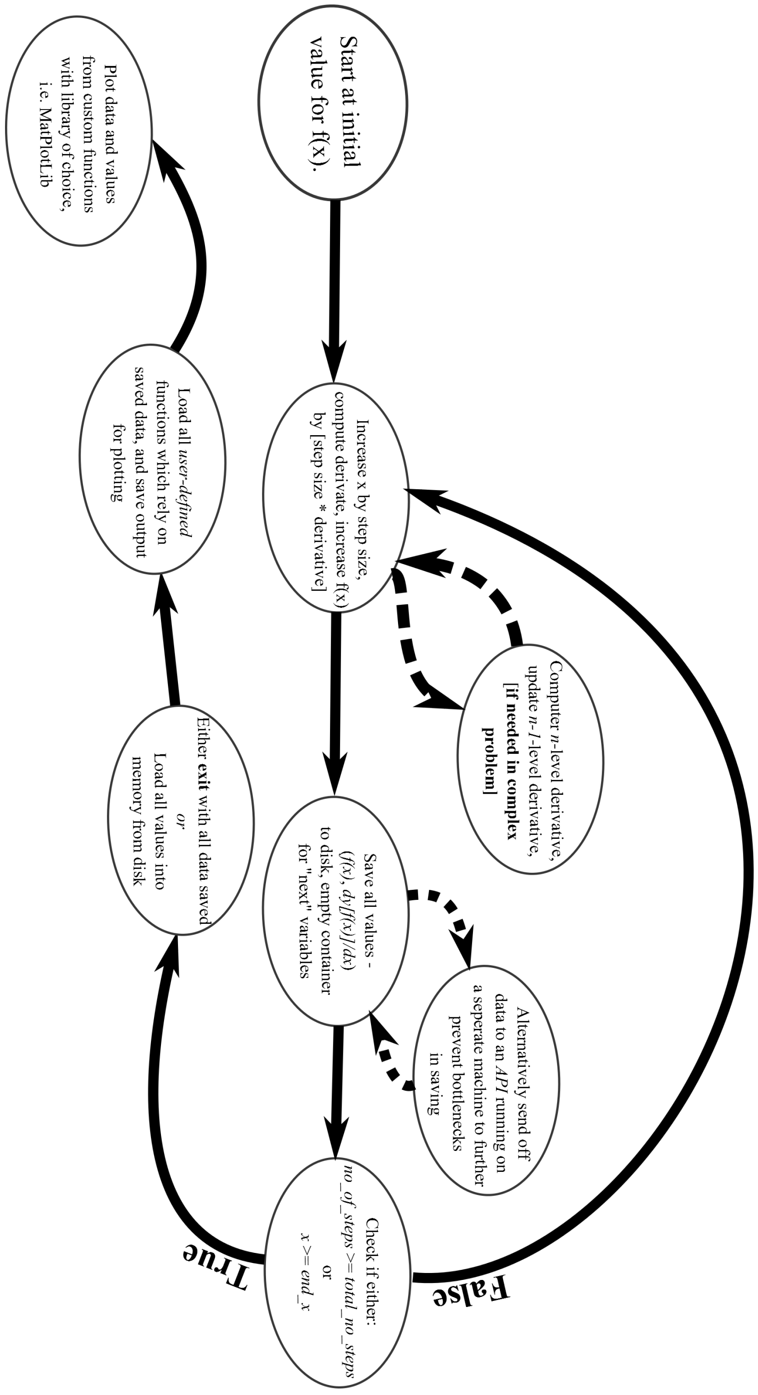
Start at initial value for f(x).
Increase x by step size, compute derivate, increase f(x) by [step size * derivative]
Computer n-level derivative, update n-1-level derivative, [if needed in complex problem]
Save all values - (f(x), dy[f(x)]/dx) to disk, empty container for "next" variables
Alternatively send off data to an API running on a seperate machine to further prevent bottlenecks in saving
Check if either: no_of_steps >= total_no_steps or x >= end_x
True
False
Either exit with all data saved or Load all values into memory from disk
Load all user-defined functions which rely on saved data, and save output for plotting
Plot data and values from custom functions with library of choice, i.e. MatPlotLib

Figure 9: A visualization of the program flow used in the implementation of Euleur's method for this study.

```
#  Euler's method solver for differential equations
#  <eyvind.niklasson@gmail.com> / http://eyvindniklasson.se
#  libraries_needed:
#  matplotlib
#  numpy
#  scipy

from math import *
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import os

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']*vector_b['z'] - vector_v['z']*vector_b['y'])
        if(direction == 'y'):
            #print "acceleration in y", (vector_v['x']*vector_b['z'] + vector_v['z']*vector_b['x'])
            return (self.particle_charge/self.particle_mass)*(vector_v['z']*vector_b['x'] - vector_v['x']*vector_b['z'])
        if(direction == 'z'):
            print "acceleration in z", self.particle_charge/self.particle_mass*(vector_v['x']*vector_b['y'] + vector_v['y']*vector_b['x'])
            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']*vector_b['y'] - vector_v['y']*vector_b['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[component] * step_size)
        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 for getting the division right :)

        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 "convergence" effect. This should eventually let the particle "flip" direction as well.
                self.vector_b['z'] = -((self.vector_pos['z'] - c_value)*(self.vector_pos['z'] - c_value)) + c_value*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_pos['x'])*(((self.vector_pos['z'] - c_value)*(self.vector_pos['z'] - c_value))/
(c_value*c_value))/(self.vector_pos['x']*self.vector_pos['x'] + self.vector_pos['y']*self.vector_pos['y'])
                    self.vector_b['y'] = -(0.5) * (m * self.vector_pos['y'])*(((self.vector_pos['z'] - c_value)*(self.vector_pos['z'] - c_value))/
(c_value*c_value))/(self.vector_pos['x']*self.vector_pos['x'] + self.vector_pos['y']*self.vector_pos['y'])


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

                    #  Calculate v value from it's derivative with good old euler :)
                    acceleration = self.problem.v_prim(component, self.vector_v, self.vector_b) * self.step_size
                    self.vector_v_next[component] = self.vector_v[component] + acceleration

                #  Update particle position based on velocity and previous position.
                self.vector_pos = self.problem.particle_pos(self.vector_pos, self.vector_v, self.step_size)

                #  Save all known data for future plotting, this could be rewritten with a "listener"/"server" system to minimize time
                #    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'], 'magnetic_z')
                self.save('magnetic_vector_bx', self.vector_b['x'], 'magnetic_x')
                self.save('magnetic_vector_by', self.vector_b['y'], 'magnetic_y')

                #  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 variables!
    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], values[i]

                #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 variables!
    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 iteration
            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 value')

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

#Solve this using two euler approximations at the same time. Predict first derivative then second using the first.
#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 -> name of custom function,
#    arg3 -> the "x" value you want to iterate with, can be any value, arg(3+n) -> any other variables you
#    want to use in your custom function

#Config
step_size = 0.001                                               #Accuracy - "stepsize" - adjust for more/less precise plots, experiment to trade-off time/cpu-hours

#Run it
problem = problem()                                              #Define our problem

#Run it for several stepsizes

print "\nDoing stepsize:", float(0.2)
solver_instance = solver(step_size, 20, problem)                         #Config our solver
solver_instance.euler()
solver_instance.plot('time', 'part_speed_z', 'magnetic_vector_by', 'magnetic_vector_bx')                              #Plot the speed versus the z pos
solver_instance.plot('part_pos_z', 'magnetic_vector_bz')
solver_instance.plot_3d('part_pos_x', 'part_pos_y', 'part_pos_z')          #Plot our beloved points in the 3d graph

#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()
```