Cyclotron Stimulation

Christopher Ong, Cheryl Wang

June 2018

Contents

1 Introduction

Python is a high-level scripting language that is fully object-oriented, utilized for general programming purposes. It contains simple to use syntax and the code is dynamically interpreted, making it very flexible. Python provides a variety of function tools used for basic scientific research and engineering, which is supported by comprehensive, open-source, contributions from scientists for their own work. There are currently two major versions of Python available: Python 2 and Python 3. To learn which one best fits your needs, you may refer to https://wiki.python.org/moin/Python2orPython3this page.

1.1 Installing Python

You may already have the programming language since versions of Mac OS X 10.8 or later features Python 2.7 pre-installed in your system. You may quickly check by opening Terminal located in the *'Utilities'* folder in *'Applications'* , and type the following:

```
python -V
```

Note that the capitalization in V is important in this step. The command will return the version of Python installed in your system. The Apple version of Python is installed in System/Library/Frameworks/Python.framework and /usr/bin/python, respectively; **these should never be modified**. If you've never downloaded Python before, I highly encourage you to do so, as it will provide you with a framework which includes executables and libraries. You may download it https://www.python.org/downloads/here.

If everything went well, we can now check to see if some useful modules are installed. In particular, lets check if we have numpy, scipy, and matplotlib. We will open python from terminal by simply typing python in the command line, as follows:

```
python
```

You will be prompted with an introduction of the version being used along with some information. Now, to check if you have the modules, we will use the inbuilt help() function:

```
help('numpy')
```

If you have numpy installed, you will be prompted with documentation about the module, you may hit the key Q to interrupt and go back to the python interface. Do the same for all three modules. If you are missing one or more modules, you can install them by downloading the content from respitory. To exit the python interface hit Ctrl-Z and you can install numpy simply by typing

```
python -m pip install --user numpy
```

This installs packages for your local user, and does not write to the system directories. Do the same for the rest of the modules if you do not have them installed.

Now, to test the modules. The easiest and fastest way is to make a script on a text editor, before proceding, please refer to section 1.2. You can then run scripts from the Terminal window, first save the script with the .py extension in any desired folder and use Terminal to move to the location of the file you want to execute. For example, if you saved myscript.py in your Desktop in a folder named python, then to move locations, you would execute:

```
cd ~/Desktop/python
```

To run your script, execute:

```
python myscript.py
```

1.1.1 Matplolib

The matplotlib library produces publication-quality figures such as scatterplots, histograms, power spectra, power chart, etc., that allows for effortless customizable properties. There is also a procedural pylab that provides a MATLAB-like interface, giving you full control of axes properties, text annotation, line styles, and more. Let's plot a mathematical function to test matplotlib, type and execute the following:

```
from pylab import *

x = []
y = []

for i in range(100):
angle = .1 * i
sang = sin( angle )
x.append(angle)
y.append(sang)

plot( x , y , 'r' )
show()
```

We start by importing the functions in pylab and proceed to create two empty lists that will represent our x and y axis data points. The range() function generates a list of 100 numbers, that is passed on to the for loop where the integer 'i' runs from 0 to 99. Therefore, for every value in 'i' we multiply it by '.3', store the value in a temporary variable named 'angle' and added to our list 'x' by using the function 'append()'. In addition, the sine of each angle is computed and stored in a temporary variable named 'sang' and similarly added to our list 'y'. It is important to note that the for loop runs only for commands that are indented. The 'plot()' commands takes the x and y coordinates as their first and second arguments, respectively. The last statement 'show()' instructs matplotlib to display the figures.

1.1.2 Numpy

The program above is fine in the sense that it introduces a variety of functions and syntax; however, since our objective is to use python as a tool for learning physics, we need data types like arrays and matrices, such as those provided by the Numpy library. Let's plot the same sine function as above, along with some lissagous figures, but this time utilizing the power of Numpy. To test numpy, type and execute the following:

```
from numpy import *
from pylab import *

x = linspace(0.0, 2 *pi, 100)

plot(x, sin(x), 'r')
                                        # 'r' stands for red
plot(sin(x), cos(x), 'y')
                          # 'y' stands for yellow
plot(sin(2 * x, cos(x), 'b')                               # '
    b' stand for blue. The default setting.

show()
```

The first statement imports all of the functions from numpy. The 'linspace()' function generates an array of 100 equi-spaced values, starting from 0 and ending with 2π. It is also possible to specify the color of the plot by adding a third argument inside the 'plot()' function. Note that anything followed after a hashtag, '#' is ignored by the compiler, we use the hashtag to make comments inside our program. In this program we managed to plot three figures, meanwhile eliminating the need of a for loop; essentially reducing the complexity of the program, allowing us to concentrate on physics.

1.1.3 Scipy

As an extension to the scientific tools provided by numpy, the Scipy library offers a collection of mathematical algorithms and convenience functions that allows a high-level command for data manipulation and visualization that rivals systems such as MATLAB, IDL, Octave and Scilab [1]. As a brief example, we will employ the most commonly used special function: the gamma function, $\Gamma(z)$, defined via a convergent improper integral, for complex numbers with a positive real part:

$$\Gamma(z) = \int_0^\infty s^{x-1} e^{-x} dx$$

Type and execute the following, to test scipy:

```
from scipy.special import gamma
print gamma(.05)
```

1.2 Text Editor

Before you start writting code in python, you need to choose a text editor. IDLE (Integrated Development and Learning Environment) is the default text editor bundled with Python. It offers multi-window text editing with syntax highlights, smart indent and autocompletion. No download is necessary since it comes with Python.

However, if you have multiple versions of Python in your system, you often want to have control on which version you want your script to run. An easy method is to use the python interpreter from Terminal. To do that you need another text editor. The one for my personal use is BBEdit, I especially like the FTP/SFTP[1] support feature that allows you open/save/transfer scripts from remote locations and is useful in astrophysics applications. You may download it http://www.barebones.com/products/textwrangler/download.htmlhere. Other popular choices are https://www.sublimetext.com/Sublime Text and // http://aquamacs.org/Aquamacs. Nonetheless, you may choose whichever text editor you like best.

[1]Although you won't be needing it for this project

2 Introduction to Classes

This section aims to introduce Classes presented in python.
In the last guide, Python was defined as an Object Oriented Language. The study of classes gives us insight into what that actually means.

Object: An object is simply something that can be selected and manipulated. A number. A sentence. A word. They are all objects.
To be selected, it must have some data.
To be manipulated, it must have functions that define how it is to be manipulated.

Class: A class is a definition, which can be thought of as a blueprint for some *instance* of an object, consisting of two major parts:

1. Data (Atributes) that each object will have
2. The methods[2] and operators that dictate how objects will be manipulated.

Now, this is not a brand new topic. We have been working with classes and objects, without knowing about their names. **Integers, Doubles, Strings, Bool** are all examples of classes. Take the sample code:

```
mass= 1.67E-27 #mass of the particle
mString = "mass of negative hydrogen atom is:"
print mString, mass
#output: mass of negative hydrogen atom is: 1.67e-27

mString = mString[9:15]
print mString
#output: egativ
```

Here, **mass** and **mString** are object instances of the classes double and string respectively. In line 6, we are 'splitting' the string **mString** from the 9th index to 15th index using the operator **[start:end]**. It is important to note that python uses a zero-based index, in this example, the zero index would be the 'm' character. Also note that spaces count as characters.

Object Oriented: Finally, we can understand what Object Oriented programming entails. Programming in which we deal with objects, constantly creating, manipulating new objects that inherit their properties from the classes they are instances of.

2.1 Syntax

The general syntax is:

[2]Functions defined inside a class are normally called methods

```
class class_name:  # Defining a class
[statement 1]
[statement 2]
[statement 3]
[etc.]
```

However, the actual code is a little bit more intricate than this, with some conventions that are used as a general coding practices that are carried out, almost out of convention.

We will define a class **ball**, which will store the size, the mass, the elasticity and the owner of the ball.

```
class ball:
def __init__(self, sz, ms, elast, name):
self.size = sz
self.mass = ms
self.elasticity = elast
self.master = name

def volume(self):
return (4/3)*(self.size**3)

def density(self):
return (self.mass / self.volume())
```

We have three member functions. The **__init__ (self, sz, ms, elast, name)** function is a special function. It is called a constructor. It is how we initialize an instance of the class (create the object). We have two more member functions, **volume(self)** and **density(self)**. The former calculates the volume of the ball and the latter calculates the density.

Inside the function **__init__(self, sz, ms, elast, name)**, we have the member variables size, mass, elasticity and master. Each takes its respective value. For example, the string **name** is stored in the variable **master**.

When calling the functions, we don't write the parameter **self**. It is used to allow the function to refer to other member functions and member variables. Look at the following code:

```
#Now, we will create two instances of the class

b1 = ball(5, 2, 7.95, "Joe")

b2 = ball(2, 4, 5, "Josephine")

b2.master = b1.master
b2.size = 90

print b2.master, b2.size
#output: Joe 90
```

We create the first instance **b1**, with size **5**, mass **2**, elasticity **7.95** and the owner of the ball is **"Joe"**. Similarly, we initialize another instance of the ball class as **b2**. Next, we decide that the ball **2** is also owned by the master of ball **b1**, **"Joe"**. Finally, we change is size of **b2** and **print** the changes done to **2**.

2.2 Common Errors and Coding Advice

One of the most common errors during writing a class is forgetting the use of **self** keyword during the declaration and definitions of the functions. Nested functions, that is calling functions inside functions must be taken care of as well.

It should be noted that Python is an interpreted language. That is, it is read from to top to bottom, line by line. So, if instances are created before the class is defined, then code gives an error, since during its runtime, it hasn't learnt that the class **ball** has been defined yet. This should be noted for any definitions of functions and variables in general.

2.3 Examples

After the above introductions into the codes and statements to create a class by ourselves, the following are some examples. All examples are pulled and modified from the Internet.

2.3.1 Example 1: Person

In this example, the name for our Class is Person, we define the objects such as name, surname, birthdate into our data and print out what we defined in the classes at the end.

```
import datetime

class Person(object):
def __init__(self, name, surname, birthdate, address,
    telephone, email):
self.name = name
self.surname = surname
self.birthdate = birthdate

self.address = address
self.telephone = telephone
self.email = email

def age (self):
today = datetime.date.today()
age = today.year - self.birthdate.year

if today < datetime.date(today.year,  self.birthdate.month,
     self.birthdate.day):
age -= 1
```

```
return age

person = Person (
"Cheryl",
"Wang",
datetime.date(2000,8,15),
"330 De Neve Drive, SLC - L764",
"310 210 7057",
"cheryl.wang.huiyi@gmail.com"
)

print (person.name)
print (person.email)
print (person.age())
```

2.3.2 Example 2: Rocket

This example is slightly less complicated than the previous only involving the position of x and y coordinate of the rocket. If the program runs well, we will see the rocket's y value move up by 1 each time we print out a new 'Rocket Altitude'.

```
class Rocket(object):
def __init__(self):
self.x = 0
self.y = 0

def move_up(self):
#increment the y-position of the rocket
self.y += 1

#our rocket object starts to move by +1 increment
rocket01 = Rocket ()
print ("Rocket Altitude:", rocket01.y)

rocket01.move_up()
print ("Rocket Altitude:", rocket01.y)

rocket01.move_up()
print("Rocket Altitude", rocket01.y)
```

2.3.3 Example 3: Simple Coffee Maker

Hopefully this and the previous examples will provide some enlightenment into how Classes works.
In this example, functions are defined by 'def', and input into the object we defined as 'pythonBean'.

```
class coffeeMachine():
name = ""
beans = 0
water = 0

def __init__(self, name, beans, water):
```

```
self.name = name
self.beans = beans
self.water = water

def addBean (self):
self.beans = self.beans +1

def removeBean(self):
self.beans = self.beans -1

def addWater(self):
self.water = self.water +1

def removeWater(self):
self.water = self.water - 1

def printState (self):
print "Name = " + self.name
print "Beans = " + str(self.beans)
print "Water = " + str(self.water)

pythonBean = coffeeMachine("pythonBean", 83, 20)
pythonBean.printState()
print ""
pythonBean.addBean()
pythonBean.addWater()
pythonBean.printState()
```

3 Introduction to Flow Control

Flow control in programming determines how and in what order code in an program is executed. One can think of flow control as a literal flow chart, where the next step in the chart is determined by flow control methods. Flow control methods include if-else statements and for and while loops.

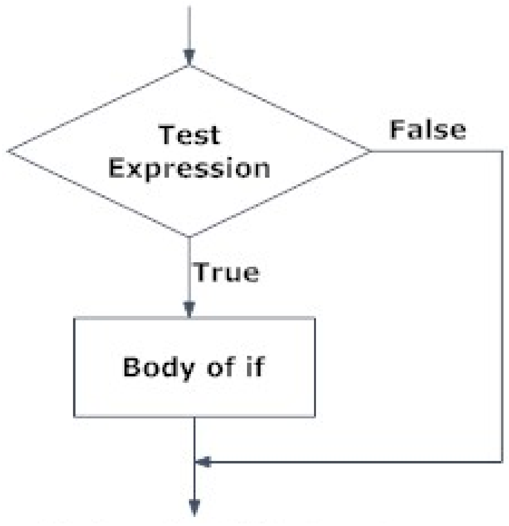


Figure 1: **Flow Chart.**

If-else statements in Python are used as logical operators. When a set of if-else statements is run, Python go down the list of statement and check whether the statements are true or false, according to Boolean values. In simple english, an if-else statement can be thought of as "If **x** condition is true, then do **y**. Else, do **z**." When thought of as in a flow chart, as displayed in figure 1, if-else statements take one step of the code and send it to the next step via different branches, branches to the next possible steps being determined by the conditions of the if-else statement. One can expand the number of conditions and outcomes of an if-else set by using the elif statement, which is shorthand for the "if else" statement. The elif statement effectively adds another if to the statement set. One can have as many elif's or as little elif's as they want in an if-else set.

Loops in Python repeat operations. In a flow chart, a loop would be a branch that leads into the same step that it came from. For loops repeat a block of code for a set number of times. In contrast, a while loop checks a conditiion before iterating, typically executing the code repeatedly until the condition becomes false. If the condition is initially false, the while loop will not run.

3.1 Syntax

Elif statements within the set are always placed between the if and the else statements. If and else may only be used once in the statement set. The else statement is optional; however if the none of the conditions provided by the if or elif statemetns are true, then no output will be provided. Each if, elif, or else statement will consist of at least two lines; one line is used for the condition and the rest of the line(s) are used in determining what happens when the condition is fulfilled. The first line for the condition statement ends in a colon. The line(s) following the statement that tell the program what to do are indented in relation to the condition statement. An important thing to note is the use of Boolean values in the if and elif statements.

For loops iterate over an item in a range. While loops iterate over a condition. Both the item range and the condition for their respective cases are specified in the line beginning the loop, which will start with "for" or "while". A colon ends the starting line, and a new indentation block below contains what will happen over each iteration.

3.1.1 Examples

```
q=3
if q==-1.6e-19:
print("The particle is an electron")
elif q==1.6e-19:
print("The particle is a proton")
elif q==0:
print("The particle is a neutron")
else:
print("The particle is unknown")
```

The above example will return the output for the else statement, as q here does not fulfill any of the conditions posed in the if and elif statements. Note the use of the double equal signs. Using a single equal sign can be thought of as setting a variable q equal to whatever value comes after the equal sign. A double equal sign instead evaluates whether what is left of the double equal sign and right of the double equal sign is true or false, which is necessary for Python to interpret the condition statements.

The following examples are modified from online examples.

```
for x in xrange(11):
print x
else: print "Stop"
```

The above example returns a sequence of numbers from 0-10. Note the use of the else statement. Else statements can be used in conjunction with for loops, and will execute their output when for loop finishes iterating over its range.

```
count = 0
while (count < 9):
print 'The count is:', count
count = count + 1
else: print "Stop"
```

```
        print "Done"
```

The above example will print the string with each number attached over each iteration. After printing for an iteration, the code adds one to the count variable, and loops over again. This will continue while the value of "count" is less than 9. Once this condition fails, the while loop will end and the program will execute the following code, thus outputting the string "Done". Note the use of the else statement. Else statements can be used in conjunction with while loops, and will execute their output when the condition fails.

3.2 Common Mistakes

One common mistake is to include two different statements in an if-else set that could both be true. Python reads each line in the set until it finds a condition that is true or reaches an else statement. If an if or elif statement is true but the elif statement after it is also true, Python will only pick out the first true statement. Sometimes this may not matter, but in cases where one has overlapping conditional statements then attention to order may be required.

When inputting the range in a for loop, x starts at 0 and goes to the value of n-1, where n is the value in the range. Thus if you want to iterate to x=10 as in the example above, you must use 11 instead of 10 in xrange. A common mistake in using while loops is to mess up the condition over which the code is iterated. It is possible to create a never-ending loop if the condition never fails.

As always, syntax errors such as omitting a colon or using a single equal sign instead of a double equal sign should be looked out for.

4 Introduction to Euler's Method

Euler's Method is a first-order numerical approximation for first order differential equation. This is a primitive version of the more sophisticated numerical method known as the classical Runge-Kutta method or RK4. Since this is a first order approximation, the local error is proportional to the square root of the step size.

We begin with a known differential equation and an initial value

$$\frac{dy}{dt} = f(t, y) \qquad y(t_0) = y_0 \tag{1}$$

We wish to know the value of $y(t_1)$ at $t_1 = t_0 + h$ where $h = \Delta t$ is called the step size. Euler says we can approximate this answer by the following equation

$$y_1 = y_0 + hf(t_0, y_0).$$

In general

$$y_{n+1} = y_n + hf(t_n, y_n). \tag{2}$$

4.1 Derivation

We simply expand the function y(t) around t_0 using the Taylor expansion

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{1}{2}h^2y''(t_0) + O(h^3) \tag{3}$$

Here we notice that

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0.$$

$$y_{n+1} = y_n + hf(t_n, y_n).$$

4.2 Exercise

Given the differential equation $\frac{dy}{dt} = t + 2y$ and initial condition $y(0) = 0$. Approximate the value of y(.5) using Eulers Method with step size h=.1

n	$y_{n+1} = y_1 + hf(t_n, y_n)$	t_n
0	$y_1 = y_0 + hf(t_0, y_0)$	t_0
1	$y_2 = y_1 + hf(t_1, y_1)$	t_1
2	$y_3 = y_2 + hf(t_2, y_2)$	t_2
3	$y_4 = y_3 + hf(t_3, y_3)$	t_3
4	$y_5 = y_4 + hf(t_4, y_4)$	t_4
5	$y_6 = y_5 + hf(t_5, y_5)$	t_5

4.3 Psudo-Code

Pesudo-Code is an informal text-based algorithm that uses structural conventions of programming in order to assist the development of programs. An example pseudo-code is shown below for programming Eulers method.

```
1. Define $f(t, y)$
2. Input initial conditions $t_0$ and $y_0$
3. Input step size, h and the number of steps, n.
4. for i from 1 to n do
   (a) $m = f(t_0, y_0)$
   (b) $y1 = y_0 + h * m$
   (c) $t_1 = t_0 + h$
   (d) Print $t_1$ and $y_1$
   (e) $t_0 = t_1$
   (f) $y_0 = y_1$
```

5. end

Define a Euler function in Python and solve the execersive above. In addition, solve the equation analytically, plot both results and calculate the error size. How can you make the answer more accurate?

4.4 Cyclotron Application

Let's recap what we have so far. We have a couple of programs to solve for period and radius given some parameters. We also have a program that outputs acceleration based on particle's position. In addition, we just learned one method to solve differential equations numerically, namely Euler's Method. This Friday, we will learn the more sophisticated classical RK4 method. However, Euler and RK4 methods are intended for first order differential equations, since acceleration is a second order differential equation, we will have to do a bit of manipulation in order to use them.

$$\dot{v} = q/m[E(x) + \dot{x} \times B(x)] \tag{4}$$

$$\dot{x} = v \tag{5}$$

What we have done is created two coupled first order differential equation. Let's use the Euler's Method to solve for position. Note that you have to solve the equations above simultaneously. Then plot the results and you have yourself your first cyclotron prototype!

A couple things to think about:

1. What's your step size function? Hint: A good answer would depend on the initial period and radius of a particle.
2. What are your independent variables for acceleration and velocity?

5 The Runge-Kutta Method

While Euler's method provides a reasonable estimation of a function for many steps over a small time, Euler's method is highly inaccurate over for larger integration bounds. The fourth order Runge-Kutta method, or RK4, is much more accurate without sacrificing much ease of computation. Like with Euler's method, we start with a differential equation and initial value

$$\frac{dy}{dt} = f(t, y) \qquad y(t_0) = y_0 \tag{6}$$

The RK4 method then takes measurements of the the function we're estimating at 4 different points, 1 at each end point of the integration step and 2 at the middle of the integration step to estimate the function we want

$$k_1 = h * f(t_n, y_n) \tag{7}$$

$$k_2 = h * f(t_n + \frac{h}{2}, y_n + \frac{k1}{2}) \tag{8}$$

$$k_3 = h * f(t_n + \frac{h}{2}, y_n + \frac{k2}{2}) \tag{9}$$

$$k_4 = h * f(t_n + h, y_n + k3) \tag{10}$$

We then weight this four estimations using the equation below to obtain our estimation

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{11}$$

The RK4 method is much more practical for integrating differential equations and making practical models of real life phenomena due to its balance of relative simplicity and accuracy. The RK4 method will be used and modified in future parts of our labwork.

6 Application to Earth's Magnetic Field

6.1 Definition

Declination: The angle between the direction of the compass and the true direction of the north.
Inclination: The vertical angle between the north and the direction the compass points when it is suspended in free fall.
Magnetic poles: Positions on Earth's Surface for where the horizontal component of the magnetic field is 0.
Magnetic Equator: An undulating curve near the Earth's geographical equator where the vertical component of the magnetic field is 0.
Cosmic Rays: High energy particles coming from cosmic. Solar particles count as one of them.
Before modeling the Earth as a magnetic dipole, it is best to define a few terminologies about magnetic dipole, which is the first order approximation of the Earth's magnetic field.

1. **Magnetic Dipole:** A pair of South and North magnetic poles kept closed together or viewed from a large distance. More formally, it is the limit of either a closed loop of electric current or a pair of poles as the dimensions of the source are reduced to zero while keeping the magnetic moment constant.

2. **Magnetic Dipole Moment:** A quantity that represents the magnetic field strength and orientation of a magnet or other objects that produces a magnetic field.

3. **Magnetic Vector Potential ($\vec{A}$):** It is a vector field that serves as the potential for magnetic field. The curl of the magnetic field potential is the magnetic field.

$$\vec{B} = \nabla \times \vec{A}$$

6.2 Magnetic field as Dipole

We are able to write the Earth's magnetic pole by modeling it according to a dipole - a first order approximation.

$$\mathbf{B(r)} = (B_r, B_\theta, B_\phi) \tag{12}$$

When we align the Earth's magnetic field along the z-axis:

$$\begin{aligned} V(\mathbf{r}) &= \frac{1}{4\pi r^3} m \cdot r \\ &= \frac{mr\cos(\theta)}{4\pi r^3} \\ &= \frac{m\cos(\theta)}{4\pi r^2} \end{aligned} \tag{13}$$

We can derive three equations separately for B_r, B_θ, and B_ϕ, stated as the following. We fine $\mathbf{m}$ as the dipole moment, and μ_o as the vacuum permeability.

$$\begin{aligned} B_r(r,\theta,\phi) &= \frac{-2\mu_o mcos(\theta)}{4\pi r^3} \\ B_\theta(r,\theta,\phi) &= \frac{-\mu_o msin(\theta)}{4\pi r^3} \\ B_\phi(r,\theta,\phi) &= 0 \end{aligned} \tag{14}$$

Thus the total field, which is just the vector sum of the three B functions.

$$B(r,\theta,\phi) = \sqrt{B_r^2 + B_\theta^2 + B_\phi^2} = \frac{\mu_o m}{4\pi r^3}\sqrt{1+3cos^2(\theta)} \tag{15}$$

At the **North Pole**

$$\begin{aligned} B_r(r,\theta,\phi) &= \frac{-\mu_o mcos(\theta)}{2\pi r^3} \\ B_\theta(r,\theta,\phi) &= 0 \end{aligned} \tag{16}$$

At the **Magnetic Equator**

$$\begin{aligned} B_r(r,\theta,\phi) &= 0 \\ B_\theta(r,\theta,\phi) &= \frac{-\mu_o msin(\theta)}{4\pi r^3} \end{aligned} \tag{17}$$

Let's define

$$k_o = \frac{\mu_o m}{4\pi}$$

and we know that at the Earth's Dipole $k_o = 8 * 10^{15}\ Tm^3$
At the dipole, $\theta = \pm 90°$, and we return the value of B as $B = \frac{2k_o}{R_E^3} \approx 60\mu T$
At the dipole, $\theta = \pm 0°$, and we return the value of B as $B = \frac{k_o}{R_E^3} \approx 30\mu T$

Conceptually, Earth's magnetic field does not lie directly on the north and south pole and is not necessarily antipodal. The best fit dipole currently is 11.5° from the geographical north pole.

6.3 Modeling the Earth's Magnetic Field - 2D

In this section, We strive to plot the magnetic field lines from the Earth first to see how might the magnetic field lines behave when particles from the Sun enters the field.
From the equations we are given above, we acquire:

$$\begin{aligned} B(r,\theta) &= \frac{-2\mu_o mcos(\theta)}{4\pi r^3} + \frac{-\mu_o mcos(\theta)}{4\pi r^3} \\ B(r,\theta) &= \frac{-\mu_o m}{4\pi r^3}(2cos(\theta) + sin(\theta)) \end{aligned} \tag{18}$$

Here, the m stands for the Earth's magnetic dipole moment, which from L03 we know the best value should be $m = 7.94 * 10^{22} Am^2$. The value of is $\mu_o = 4\pi * 10^{-7} Hm^{-1}$.
We define

$$M_E = \frac{-\mu_o m}{4\pi} = -7.94 * 10^{15} AHm \tag{19}$$

which, satisfactorily, is the highly as k_0 we acquire from other source we found. Thus we acquire the equation:

$$B(r,\theta) = \frac{-7.94 * 10^{15}}{r^3}(2cos(\theta) + sin(\theta)) \tag{20}$$

which gives us a beautiful plot of the shape of a pea, tilted at 11.5° to the left.

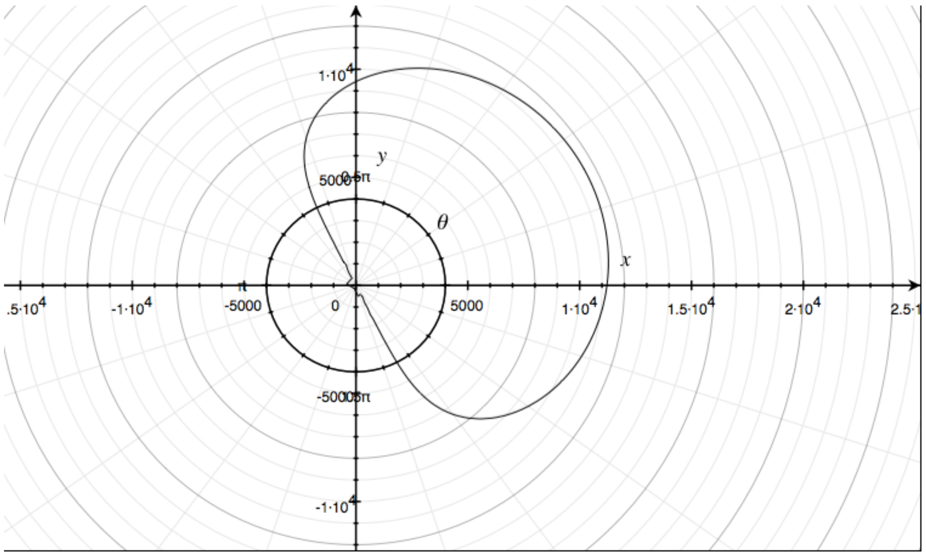


Figure 2: Polar Coordinate Graph of the Earth's Magnetic Field

However, if we want to plot it in Python, we will need to change the coordinates into Cartesian. For which is calculated as the following:

$$\begin{aligned}(x^2+y^2)^{\frac{5}{2}} &= \frac{-M_E(y+2x)}{2} \\ (x^2+y^2)^{\frac{5}{2}} &= \frac{-7.94*10^{15}(y+2x)}{2}\end{aligned} \tag{21}$$

which gives us almost the same graph (the origin of the Polar graph of Figure 1 is slight twisted).

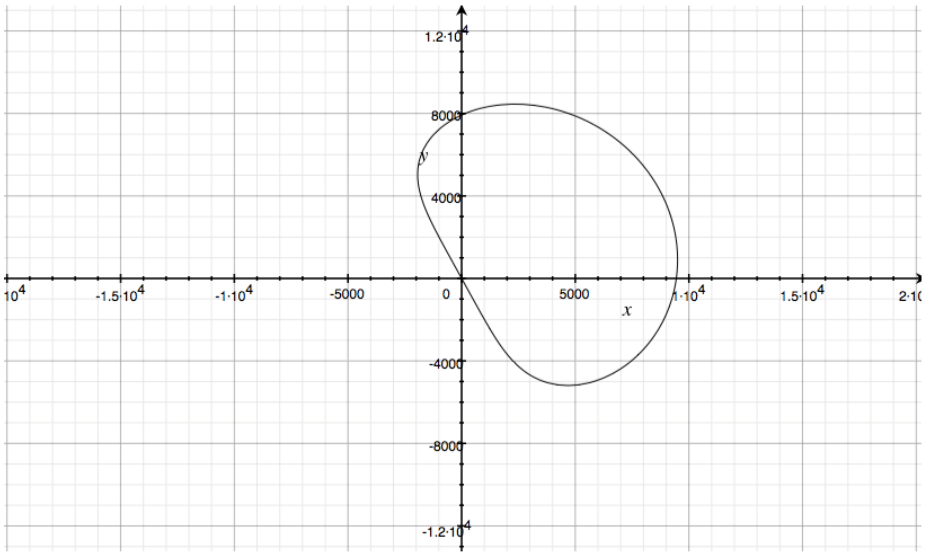


Figure 3: Cartesian Coordinate Graph of the Earth's Magnetic Field

If we add a negative sign to the graph, we acquire the other half of the pea. The next step will be to plot multiple lines of the magnetic field, as the Earth does. However, the equations should yield multiple lines according to r instead of just one! Something must be wrong. We need to draw the contour of B.

We complete this using the help of Wolfram Mathematica, which we can plot using the command Contour.

```
Y[r_, f_] = -7.94*10^15 (2 Cos[f] + Sin[f])/r^3
ContourPlot[Y[Sqrt[x^2 + y^2], ArcTan[x, y]], {x, -8*10^6,
    8*10^6}, {y, -8*10^6, 8 *10^6}]
```

which generates the following graph:

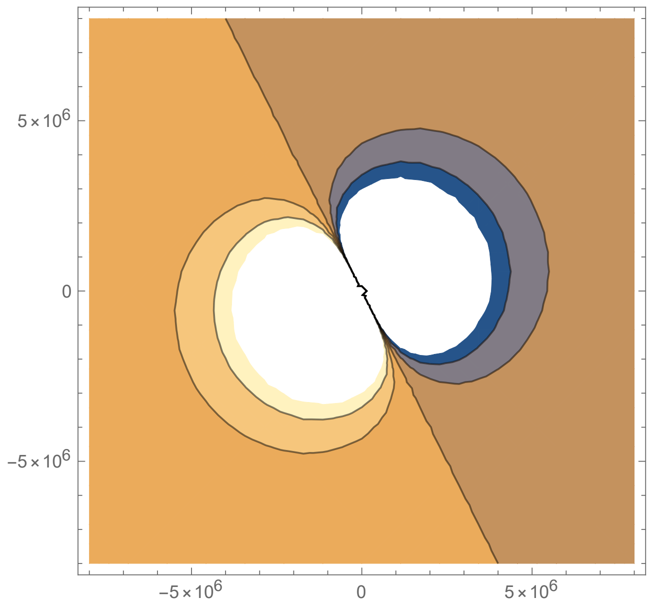


Figure 4: Earth's Magnetic Field Line contour Map using Wolfram Mathematica

I will try to use python to plot it later and include the code for it.

6.4 Modeling the Earth's Magnetic Field - 3D

6.4.1 Mathematica

We therefore try to plot the 2D version of Magnetic field into 3D version by also using Mathematica. By using the formula

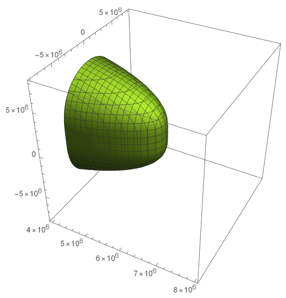
$$B(r,\theta,\phi) = \frac{\mu_o m}{4\pi r^3}\sqrt{1+3cos^2(\theta)} \tag{22}$$

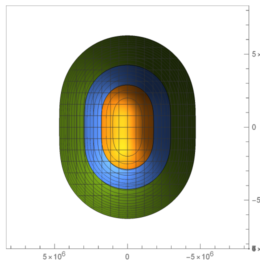
The code should be the following:

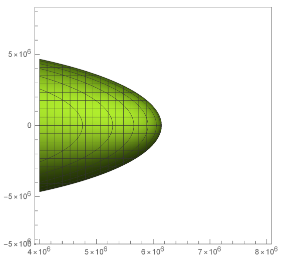
```
Y[r_, f_, p_] = −7.94*10^15 (Sqrt[1 + 3 (Cos[f])^2])/r^3
ContourPlot3D[
Y[Sqrt[x^2 + y^2 + z^2], ArcCos[z/ Sqrt[x^2 + y^2 + z^2]],
ArcTan [y/ x]], {x, 5*10^6, 8*10^6}, {y, −8*10^6,
8*10^6}, {z, −8*10^6, 8*10^6}]
```

However, I realize that if we put the range of x, y, z, all as $\{x, -8*10^6, 8*10^6\}, \{y, -8*10^6, 8*10^6\}, \{z, -8*10^6, 8*10^6\}$. However, this will results in the command of $\frac{1}{0}$, and the command will not run.

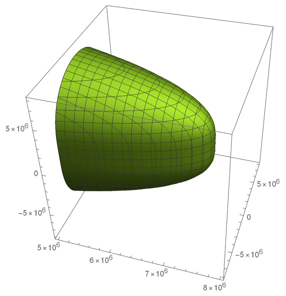
Thus, I try to set the value of x axis into $\{x, 5*10^6, 8*10^6\}$ and $\{x, 4*10^6, 8*10^6\}$. Which gives us the following graphs.

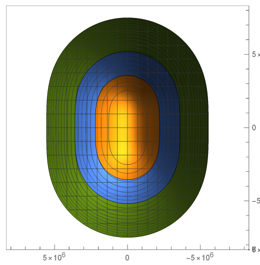

(a) Ranging from $4.0*10^6$

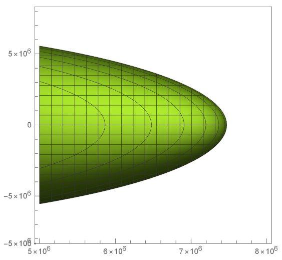

(b) Side view of $4.0*10^6$


(c) Top view of $4.0*10^6$

We now try to plot the graph for the range $\{x, 5*10^6, 8*10^6\}$.


(a) Ranging from $5.0*10^6$


(b) Side view of $5.0*10^6$


(c) Top view of $5.0*10^6$

I'm not entirely sure about why it is not displaying lines like magnetic field. However, from the side view we can clearly see the layers of the contour map for which we can model that the magnetic field of the Earth decreases as it goes outwards.

6.4.2 Python, Canopy and Mayavi

Chris, on the other hand had tried to plot the magnetic field lines by using python. Here, a special visualization module called MayaVi was used to help visualize the field lines. MayaVi was obtained through the python installer Canopy, which provides python and a host of other packages useful to the sciences, including numpy and scipy.

Here, due to time limitation, we will be presenting the final code which gives us the graph we desired.

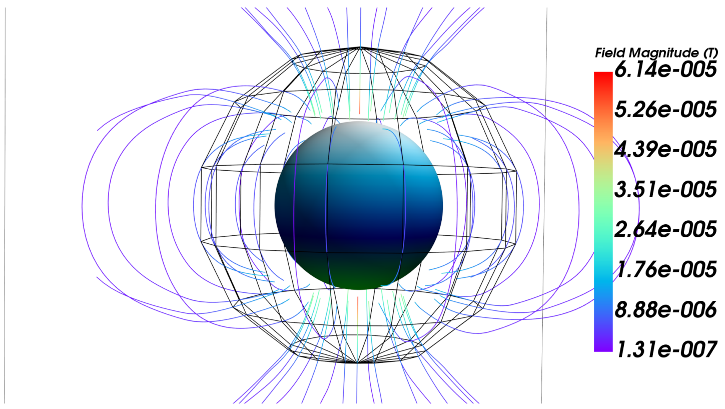
The equations used to graph the equations are the following:

$$\begin{aligned} B_x &= \frac{3xz}{(\sqrt{x^2+y^2+z^2})^5} \\ B_y &= \frac{3yz}{(\sqrt{x^2+y^2+z^2})^5} \\ B_z &= \frac{2z^2-x^2-y^2}{(\sqrt{x^2+y^2+z^2})^5} \end{aligned} \tag{23}$$

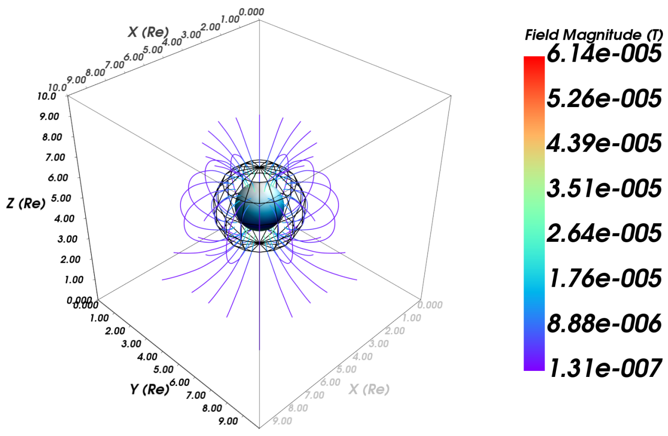
We now add in the constant for Earth's magnetic field which are $B_0 = 3.07 * 10^{-5}T$, representing the measured field strength at the magnetic equator.

R_e is the Radius of the Earth, which is $6.3781 * (10^6)m$. In this model, we take $R_e = 1$, which makes everything scaled to the magnitude of Earth's radius. The magnitude for the field is $\sqrt{B_x^2 + B_y^2 + B_z^2}$

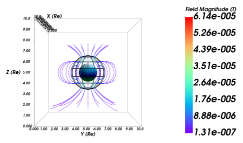
Which yields the following graphs:



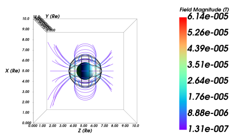
(a) Close up view



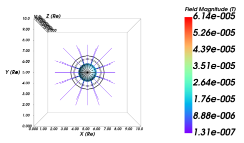
(b) Far side view



(a) X-axis Head On



(b) Y-axis Head On



(c) Z-axis Head On

As can be seen in the figures above, the Earth's magnetic field is represented by the various lines flowing in and out of the Earth's poles. Their magnitudes are colored according to the strength of the fields. Where the magnetic field is strongest, at the poles, the field lines are colored red. As the lines extend outward, they move down the color spectrum towards purple. The solid bluish-green sphere represents the Earth, with a graphical radius of R_e. The black mesh object merely determines how far out the field lines will be extended.

The complete code is the following:

```
# -*- coding: utf-8 -*-
import numpy as np
from scipy import special

#I've made a number of improvements on the code so far. The
     most obvious is that the field lines are 3d now, and
    more closely resemble what you would see out of a
#representation of a dipole. Other fixes include changing
    the way x, y, and z were defined. mgrid is now used
    instead of ogrid. While I'm not entirely sure how
#this solved the problem of the vector magnitudes being
    incorrect, it does anyways. With our current settings,
    we can see that the field at the north pole (assuming
#the geographical north pole and the magnetic north pole
    are the same) is equal to 6.14E-5 T, which is exactly
    what we would expect given our equations. This makes me
#much more confident in using this graphical representation
     as a model. The field lines cut off a bit unevenly,
    but overall it seems fine. And most importantly the
    field
#magnitudes make sense

#Fair warning, I had trouble getting mlab and mayavi
    components to install correctly, so I used 3rd party
    software called Enthought Canopy that included the
    whole thing

#Here are the relevant equations for the magnitude of the
    magnetic field of a dipole in cartesian coordinates

#Bx =((np.sqrt(x**2+y**2+z**2))**-5)*3*x*z
#By = ((np.sqrt(x**2+y**2+z**2))**-5)*3*y*z
#Bz = ((np.sqrt(x**2+y**2+z**2))**-5)*(2*(z**2)-(x**2)-(y
    **2))

#Magnitude=np.sqrt((Bx**2)+(By**2)+(Bz**2))
```

```
#An additional source corroborates this https://ccmc.gsfc.
    nasa.gov/RoR_WWW/presentations/Dipole.pdf
#This additional source expresses the terms differently,
    specifically using M instead of B0 and Re, but
#should be equivalent

#Now adjusting for the Earth's dipole specifically
#Bx =-((np.sqrt(x**2+y**2+z**2))**-5)*B0*(Re**3)*3*x*z
#By = -((np.sqrt(x**2+y**2+z**2))**-5)*B0*(Re**3)*3*y*z
#Bz = -((np.sqrt(x**2+y**2+z**2))**-5)*B0*(Re**3)*(2*(z**2)
    -(x**2)-(y**2))

#Where B0 is the measured field strength at the magnetic
    equator, B0=3.07*(10**-5)T
#And where Re is the radius of the Earth, Re=6.3781*(10**6)
    m
#With these adjustments made, the expression for magnitude
    is still the same: np.sqrt((Bx**2)+(By**2)+(Bz**2))


#Below is what I've managed so far with creating a
    graphical model, I'll upload pictures of the model as
    well


#### Calculate the field
    ##########################################################

a = -4
b = 4
ds = 1

x, y, z = np.mgrid[a:b:ds, a:b:ds, a:b:ds];
#a and b seem to not change anything in the magnitude, only
     the position of our field in the plot
#ds=1 is great
#ds=.1 multiplies the field mag by 1000
#ds=10 puts everything to something along the lines of 6.68
    E-8, which from our largest magnitude at ds=1, seems to
     indicate that multiplying ds by 10 divided the
#field mag by 1000
#Looking at our equation for the magnetic field, I believe
    this makes some amount of sense, at least for the field
     at the north pole
#The magnetic field at the north pole is equal to 6.14E-5.
    but with ds=0.1, it is set to 6.14E-2...
#If one looks at the magnetic field Bz, (which is the only
    component at the north pole), we see that the our
    singular distance component, z, is to the order of -3
#This I believe, is the connection between ds and the
    changing of the vector magnitudes... thus we ought to
    be able to safely set ds=1, as that is the "true"
    setting for our
#values... everything is scaled to 1
#ds also changes the position of our field in the plot, but
     I'm not too worried about that since we're setting it
    equal to 1 anyways
```

```
B0=3.07*(10**-5)
Re=6.3781*(10**6)


# These are the equations for the x,y,z components of the
    magnetic field
#Important bit: We'll take Re=1 for now, which will thus
    set our scale of plot to multiples of Re
Bx =-((np.sqrt(x**2+y**2+z**2))**-5)*B0*3*x*z
By = -((np.sqrt(x**2+y**2+z**2))**-5)*B0*3*y*z
Bz = -((np.sqrt(x**2+y**2+z**2))**-5)*B0*(2*(z**2)-(x**2)-(
    y**2))
#Norming these components to get the magnetic field
    magnitude
np.sqrt((Bx**2)+(By**2)+(Bz**2))


# Free memory early
del x, y, z

#### Visualize the field
    ####################################################
from mayavi import mlab
fig = mlab.figure(1, size=(400, 400), bgcolor=(1, 1, 1),
    fgcolor=(0, 0, 0))
#The values for the are some optional settings for the
    figure, like the initial size of the window, and the
    color

field = mlab.pipeline.vector_field(Bx, By, Bz)


# Unfortunately, the above call makes a copy of the arrays,
     so we delete
# this copy to free memory.
del Bx, By, Bz

# Create a sphere
r = 1 #Radius is 1, or 1 Re, as noted before
pi = np.pi
cos = np.cos
sin = np.sin
phi, theta = np.mgrid[0:pi:180j, 0:2 * pi:180j]

x = r * sin(phi) * cos(theta)
y = r * sin(phi) * sin(theta)
z = r * cos(phi)

mlab.figure(1, bgcolor=(1, 1, 1), fgcolor=(0, 0, 0), size
    =(400, 400))
mlab.clf()

#Placing and centering the Earth
mlab.mesh(x+5, y+5, z+5, colormap='ocean')

#The entire above section starting from the "Create a
```

```
    sphere" comment is making a representation of the earth


magnitude = mlab.pipeline.extract_vector_norm(field)


field_lines = mlab.pipeline.streamline(magnitude, seedtype
    ='sphere',
seed_scale=1,
integration_direction='both',
seed_resolution=10,
colormap='rainbow',
#vmin=0, vmax=1
)
#Seedtype is important, we set it to sphere so that we can
    see the lines flowing outward and back in through the
    sphere
#Integration direction must be kept to both, setting it to
    "forward" or "backward" will only give half the field
#I commented out vmin and vmax, as it rescaled the field
    line strength to a max of 1 and a min of 0
#Increasing Seed resolution gives larger numbers of field
    lines, but it's better to override them imo using the
    streamline->seed tab in the in viewer tool

field_lines.seed.widget.center = [5, 5, 5]    #This places
    our seed widget, else it would be in some other (
    nonoptimal spot, though we can move it within the
    viewer)
field_lines.seed.widget.radius = 1 #Initial size of seed
    widget


#Adjusting various integration options here, so that our
    lines are relatively smooth
field_lines.stream_tracer.initial_integration_step = 0.1
field_lines.stream_tracer.integration_step_unit = 1
field_lines.stream_tracer.maximum_number_of_steps = 25000



#Plotting axes and using an additional outline to help
    visualize the axes
mlab.axes(field, extent=(0,10, 0,10, 0,10), nb_labels=11)
mlab.outline(field, extent=(0,10, 0,10, 0,10))

#I believe propagation here affects how far the lines can
    extend, while I'm not sure about the previous statement
    , I am sure that you don't want to set this to 0 or
    something like that
field_lines.stream_tracer.maximum_propagation = 100.

#Note: When in the mayavi viewer, click on the upper left
    button that says "View the Mayavi pipeline"; head to
    colors and legends for the streamline tab, and click on
     show legend
#This will give a legend for the colorcoding of the vector
```

```
    field strength
#Can also adjust how axes look
#Adjust number of labels, font size, etc.
#Also checking the streamline section and going to the
    StreamTracer tab gives integration options which can
    smooth the field lines
#The size of the seed widget can also be set, which will
    extend or contract the field lines...

#So the viewer is kinda screwy, you have to finangle it a
    little...

###Settings adjusted for the pictures###
#The radius of the seed sphere = 1.88801633371
#Phi and Theta resolution has been set to 8 and 16
    respectively


mlab.show()
```

6.5 Solar wind

When solar particles enter Earth's magnetic field, they are highly energized. Thus, we can view that particles are accelerated through sun as if it were passing through an electric or magnetic field. A spectra of particles of energy around 10^{20} is found.

The types of particles which are emitted from the solar flame varies but are mostly three types: electrons, protons and alpha particles (which includes two protons and two neutrons). In the following model, we will be most likely using electron as the first example we modeled

The velocity from the solar wind of the particle is approximately 400,000 m/s. While the position of which the solar wind enters the Earth magnetic field can be of our choice in a 3-D model (possibly any point 1 Earth diameter away).

6.6 Modeling Particle Trajectory With the RK4

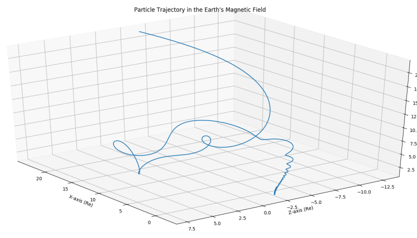
The RK4 method that we introduced previously will be essential to modeling the trajectory of a particle in the Earth's magnetic field. We will need to use RK4 twice simultaneously, one for integrating the acceleration and one for integrating the velocity. Given an initial position and velocity, we can therefore use RK4 over acceleration using our equation for the magnetic force, which depends on the velocity of the particle and the field strength, which in turn depends on the position. The code for the RK4 used to model this is provided below

```
def rk4(particle, iterations, desired_value):
#initialize arrays to be appended by rk4 results
RK4_pos = []
RK4_vel = []


n = 100
```

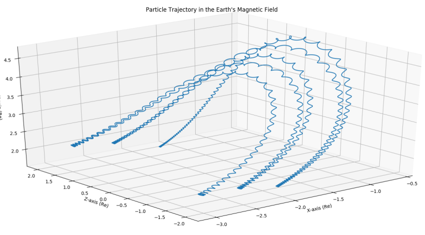
```
h = particle.period() / n
charge_to_mass_ratio = particle.charge / particle.mass

ini_pos = np.array(particle.position)
ini_vel = np.array(particle.velocity)

for i in range(iterations + 10):
i += 1
p_i = ini_pos
v_i = ini_vel

k1 = h * (1.60E-19/1.67E-27)*np.cross(v_i,[((np.sqrt(p_i
    [0]**2+p_i[1]**2+p_i[2]**2))**-5)*B0*3*p_i[0]*p_i[2],
    -((np.sqrt(p_i[0]**2+p_i[1]**2+p_i[2]**2))**-5)*B0*3*
    p_i[1]*p_i[2], -((np.sqrt(p_i[0]**2+p_i[1]**2+p_i
    [2]**2))**-5)*B0*(2*(p_i[2]**2)-(p_i[0]**2)-(p_i[1]**2)
    )])
l1 = h * v_i


k2 = h * (1.60E-19/1.67E-27)*np.cross(v_i+k1/2.0,[((np.sqrt
    ((np.array(p_i)+np.array(l1)/2.0)[0]**2+(p_i+l1/2.0)
    [1]**2+(np.array(p_i)+np.array(l1)/2.0)[2]**2))**-5)*B0
    *3*(np.array(p_i)+np.array(l1)/2.0)[0]*(np.array(p_i)+
    np.array(l1)/2.0)[2], -((np.sqrt((np.array(p_i)+np.
    array(l1)/2.0)[0]**2+(np.array(p_i)+np.array(l1)/2.0)
    [1]**2+(np.array(p_i)+np.array(l1)/2.0)[2]**2))**-5)*B0
    *3*(np.array(p_i)+np.array(l1)/2.0)[1]*(np.array(p_i)+
    np.array(l1)/2.0)[2], -((np.sqrt((np.array(p_i)+np.
    array(l1)/2.0)[0]**2+(np.array(p_i)+np.array(l1)/2.0)
    [1]**2+(np.array(p_i)+np.array(l1)/2.0)[2]**2))**-5)*B0
    *(2*((np.array(p_i)+np.array(l1)/2.0)[2]**2)-((np.array
    (p_i)+np.array(l1)/2.0)[0]**2)-((np.array(p_i)+np.array
    (l1)/2.0)[1]**2))])
l2 = h * (v_i + (k1 * 0.5))


k3 = h * (1.60E-19/1.67E-27)*np.cross(v_i+k2/2.0,[((np.sqrt
    ((np.array(p_i)+np.array(l2)/2.0)[0]**2+(np.array(p_i)+
    np.array(l2)/2.0)[1]**2+(np.array(p_i)+np.array(l2)
    /2.0)[2]**2))**-5)*B0*3*(np.array(p_i)+np.array(l2)
    /2.0)[0]*(np.array(p_i)+np.array(l2)/2.0)[2], -((np.
    sqrt((np.array(p_i)+np.array(l2)/2.0)[0]**2+(np.array(
    p_i)+np.array(l2)/2.0)[1]**2+(np.array(p_i)+np.array(l2
    )/2.0)[2]**2))**-5)*B0*3*(np.array(p_i)+np.array(l2)
    /2.0)[1]*(np.array(p_i)+np.array(l2)/2.0)[2], -((np.
    sqrt((np.array(p_i)+np.array(l2)/2.0)[0]**2+(np.array(
    p_i)+np.array(l2)/2.0)[1]**2+(np.array(p_i)+np.array(l2
    )/2.0)[2]**2))**-5)*B0*(2*((np.array(p_i)+np.array(l2)
    /2.0)[2]**2)-((np.array(p_i)+np.array(l2)/2.0)[0]**2)
    -((np.array(p_i)+np.array(l2)/2.0)[1]**2))])
l3 = h * (v_i + (k2 * 0.5))

k4 = h * (1.60E-19/1.67E-27)*np.cross(v_i+k3,[((np.sqrt((np
    .array(p_i)+np.array(l3))[0]**2+(np.array(p_i)+np.array
    (l3))[1]**2+(np.array(p_i)+np.array(l3))[2]**2))**-5)*
    B0*3*(np.array(p_i)+np.array(l3))[0]*(np.array(p_i)+np.
    array(l3))[2], -((np.sqrt((np.array(p_i)+np.array(l3))
    [0]**2+(np.array(p_i)+np.array(l3))[1]**2+(np.array(p_i
    )+np.array(l3))[2]**2))**-5)*B0*3*(np.array(p_i)+np.
    array(l3))[1]*(np.array(p_i)+np.array(l3))[2], -((np.
    sqrt((np.array(p_i)+np.array(l3))[0]**2+(np.array(p_i)+
    np.array(l3))[1]**2+(np.array(p_i)+np.array(l3))[2]**2)
    )**-5)*B0*(2*((np.array(p_i)+np.array(l3))[2]**2)-((np.
    array(p_i)+np.array(l3))[0]**2)-((np.array(p_i)+np.
    array(l3))[1]**2))])
l4 = h * (v_i + k3)


ini_vel = ini_vel + (k1 + 2.0 * (k2 + k3) + k4) / 6.0
ini_pos = ini_pos + (l1 + 2.0 * (l2 + l3) + l4) / 6.0
vel_mag = np.linalg.norm(ini_vel)
RK4_vel.append(vel_mag)
RK4_pos.append(ini_pos)

#Controls which array function returns, resets jump count
if desired_value == 'velocity':
value = RK4_vel
elif desired_value == 'position':
value = RK4_pos

return value
```

6.7 Plotting the Particle Trajectories

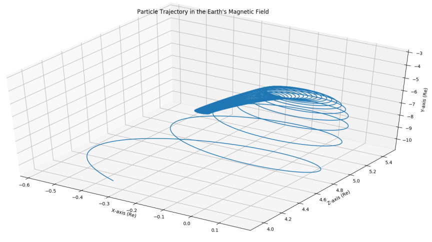
With our RK4 method we can plot the trajectory of the particle to attain a visual understanding of how the particle moves. Below are the plots for trajectories with different starting conditions.



(a) Starting position and velocity respectively:(-1.0, 2.0, -3.0), (2.0, -2.0, 2.0)



(b) Starting position and velocity respectively:(-3.0, 2.0, -2.0), (0.0, -1.0, 2.0)



(c) Starting position and velocity respectively:(0.0, -3.0, 4.0),(-1.0, 0.0, 0.0)

7 Reference

1. "Classes¶." Selection Control Statements - Object-Oriented Programming in Python 1 Documentation, python-textbok.readthedocs.io/en/1.0/Classes.html.

2. Matthes, Eric. "Classes¶." Introduction to Python: An Open Resource for Students and Teachers, introtopython.org/classes.html.

3. *https://pythonspot.com/objects-and-classes/*

4. M. Kaan. "Trajectories of Charged Particles Trapped in Earth's Magnetic Field." American Journal of Physics, vol. 80, no. 5, 2012, pp. 420–428., doi:10.1119/1.3684537.

5. http://helios.fmi.fi/ juusolal/geomagnetism/Lectures/Chapter3dipole.pdf

6. https://ccmc.gsfc.nasa.gov/RoRWWW/presentations/Dipole.pdf

7. https://seismo.berkeley.edu/ rallen/eps122/lectures/L05.pdf

8. Andersson, Ottilia. "Simulation of the Trajectory of a Charged Particle in Two Superpositioned Magnetic Fields." Research Academy for Young Scientists, 10 July 2014.