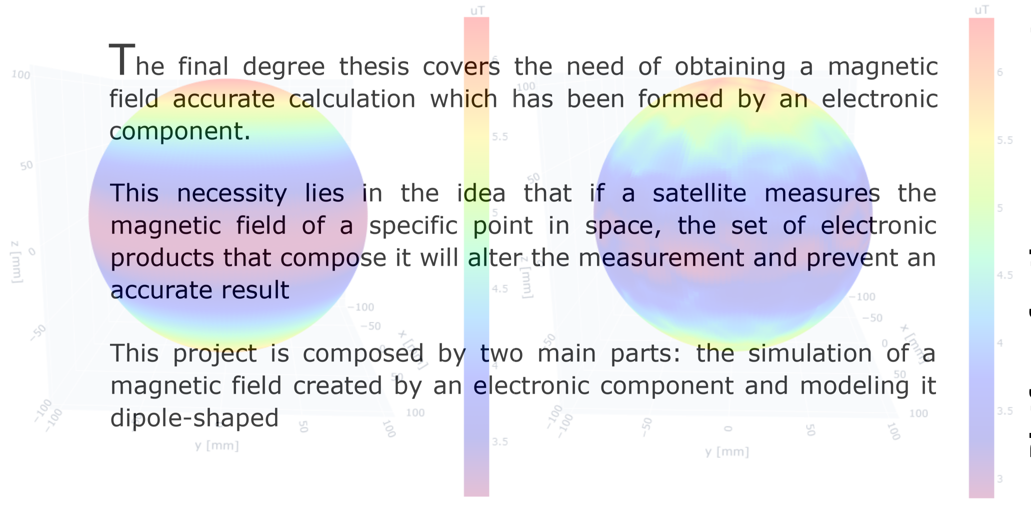


The final degree thesis covers the need of obtaining a magnetic field accurate calculation which has been formed by an electronic component.

This necessity lies in the idea that if a satellite measures the magnetic field of a specific point in space, the set of electronic products that compose it will alter the measurement and prevent an accurate result

This project is composed by two main parts: the simulation of a magnetic field created by an electronic component and modeling it dipole-shaped

Pedro Manuel Vizcaíno Delgado is a Physics student from Melilla, Spain. He was born in December 23, 1995. This work completes the Physics Degree from the University of Granada

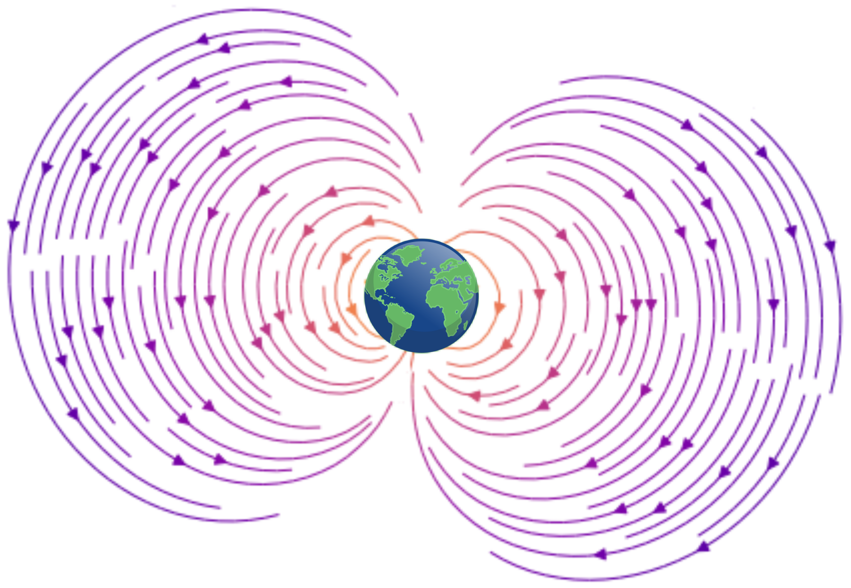
Andrés María Roldán Aranda is the academic head of the present project, and the student's tutor. He is a professor in the Departament of Electronics and Computers Technologies.

BACHELOR'S THESIS

Platform for the measuremet of an electronic product's magnetic moment

Pedro Manuel Vizcaíno Delgado

PHYSICS



Platform for the measuremet of an electronic product's magnetic moment

Pedro Manuel Vizcaíno Delgado

2019/2020

Tutor: Andrés María Roldán Aranda

“Platform for the measuremet of an electronic product’s magnetic moment”

Physics Degree

Bachelor's Thesis

"Platform for the measuremet of an electronic product's magnetic moment"

ACADEMIC COURSE: 2019/2020

Pedro Manuel Vizcaíno Delgado

Physics Degree

"Platform for the measuremet of an electronic product's magnetic moment"

AUTHOR:

Pedro Manuel Vizcaíno Delgado

SUPERVISED BY:

Andrés María Roldán Aranda

DEPARTMENT:

Electronics and Computers Technologies

Platform for the measuremet of an electronic product's magnetic moment

Pedro Manuel Vizcaíno Delgado

KEYWORDS:

Electromagnetism, Magnetostatic, Magnetic moment, Magnetic dipole, Python, PSO, Ansoft Maxwell®.

ABSTRACT:

The main goal of this final degree is to develope a computational method that would allow the simplification of a magnetic field, generated by an electronic component. To do so, once the field has been measured, the PSO method will be applied in order to turn this component into a magnetic dipole.

On the basis of the magnetic dipole theory, magnetic field equations have been developed. Therefore, simulations have been carried out, creating a Python language code, which turns the magnetic field measurements into its dipole representation.

The final objective of this project is to simplify every electronic componen of a satellite so that it takes into account the generated magnetic fields by its components when measuring a magnetic field in space. As a result, it will be possible to measure it in a more accurate way.

PALABRAS CLAVE:

Electromagnetismo, Magnetostática, Momento magnético, Dipolo magnético, Python, PSO, Ansoft Maxwell®

RESUMEN:

El propósito principal de este trabajo de fin de grado es el desarrollo de un método computacional que permita simplificar el campo magnético que genera un componente electrónico. Para ello, una vez medido el campo que genera, se aplicará el método PSO para convertir ese componente en un dipolo magnético.

Partiendo de la base teórica del dipolo magnético, se han desarrollado las ecuaciones del campo magnético que este genera. Y así, con la ayuda de simulaciones, se ha podido crear un código en lenguaje Python, el cual convierte medidas de campo magnético de un circuito complejo en su representación dipolar.

Este proyecto tiene como objetivo final poder simplificar todos y cada uno de los componentes electrónicos de un satélite para que si este mide un campo magnético en el espacio, tenga en cuenta los campos generados por sus componentes pudiendo tomar así una medida mucho mas precisa.

And Maxwell said:

$$\nabla \cdot \vec{D} = \rho$$

$$\nabla \cdot \vec{B} = 0$$

$$\nabla \times \vec{E} = -\frac{\delta \vec{B}}{\delta t}$$

$$\nabla \times \vec{H} = \vec{J} + \frac{\delta \vec{D}}{\delta t}$$

and then there was light

Acknowledgments:

Few can say that they have reach to where they are on their own, and I am not one of them. The fact that I have come this far has been thanks to the push (although not always in the right direction) of all those who are important to me.

Starting with my parents, Pedro and Isabel, who have supported me on this path, and more importantly, they have paid for it. To all my friends, especially Antonio and Javi, two other physicists, who did not refuse, most of the time, to share notes, problems and tests from other years. Finally and most importantly to my personal translator, Helena, since without her practically all this thesis would be a copy and paste of the Google translator.

Agradecimientos:

Pocos pueden decir que han llegado a donde están por si mismos, y yo no soy uno de ellos. El que haya llegado hasta aquí ha sido gracias a los empujones (aunque no siempre en la dirección correcta) de todos los que son importantes para mi.

Comenzando por mis padres, Pedro e Isabel, que me han apoyado en este camino, y mas importante, lo han pagado. A todos mis amigos en especial a Antonio y Javi, otros dos físicos, que no se negaron, la mayoría de las veces, a compartir apuntes, problemas y exámenes de otros años. Por último y mas importante a mi traductora personal, Helena, ya que sin ella prácticamente toda esta memoria sería un copia y pega del traductor de Google.

Index

List of Figures

List of Videos

Code Index

Chapter 1

Introduction

1.1 Motivation

When a satellite measures the magnetic field of a point in space, the process is not enough simply to take the measurement, since each of the electronic components of the satellite itself generates a magnetic field.

For this reason, knowing the magnetic characteristics of electronic products is necessary to be able to model their behavior and at the time of measurement, the satellite subsystems discount each of the contributions of the components, in order to obtain the measurement of the magnetic field with the highest possible precision.

In practice, the magnetic field created by the electronic product would be measured with a rotating platform, like the one in figure 1.1 owned by the European Space Agency (ESA), which has 3 magnetometers attached to it (see figure 1.2) that will measure said field in each of the three directions of space.

1.2 Project Structure

This project, divided into four chapters and two appendix, these are:

- **Chapter one**. This chapter, which is intended to be an introduction and show the general objectives and the reasons which motivate this project.

- **Chapter two**. This part addresses the theoretical framework that is necessary to understand how the project work.

- **Chapter three**. In the previous chapter we use certains approximations and in this chapter we prove that are valid.

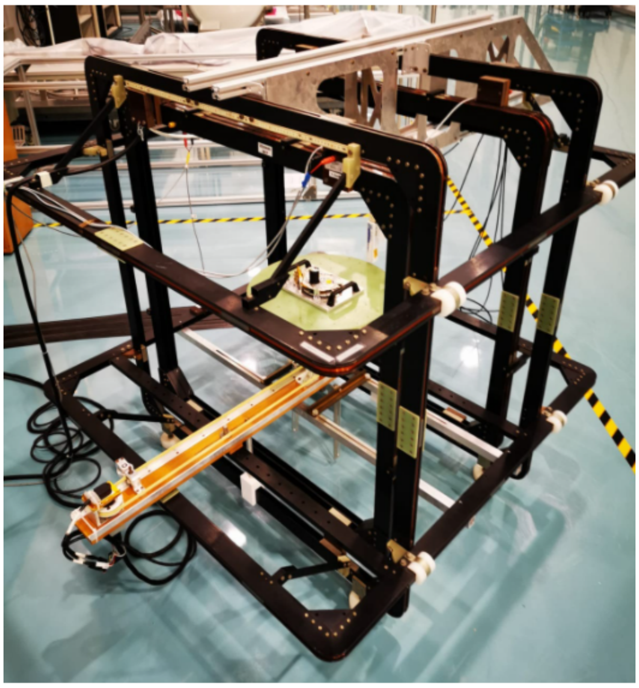


Figure 1.1 – *ESA Platform*

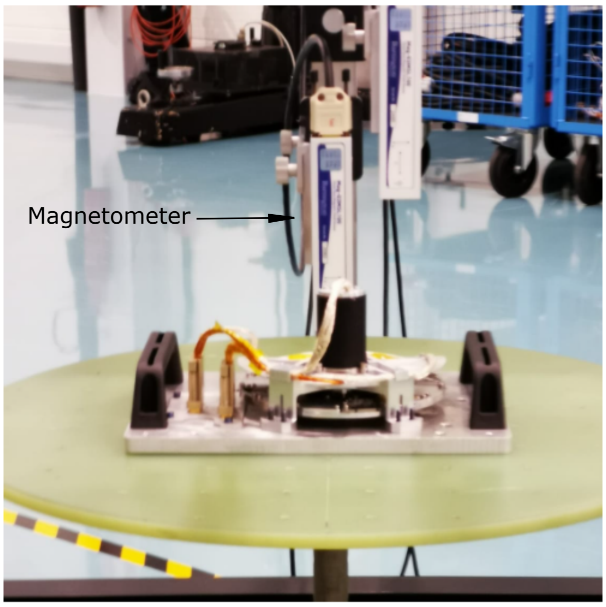


Figure 1.2 – *Platform's Magnetometes*

- **Chapter four**. The chapter four deals with the real objetive of this project. Simulate the dipole of a magnetic product from the magnetic field it generates.
- **Appendix A**. Appendix A shows how to create the simulations and export the results.
- **Appendix B**. This appendix collects the important part of the code of the Particle Swarm Optimization method used, the particle and swarm classes.
- **Appendix C**. This appendix shows the budget budget needed to replicate this project.

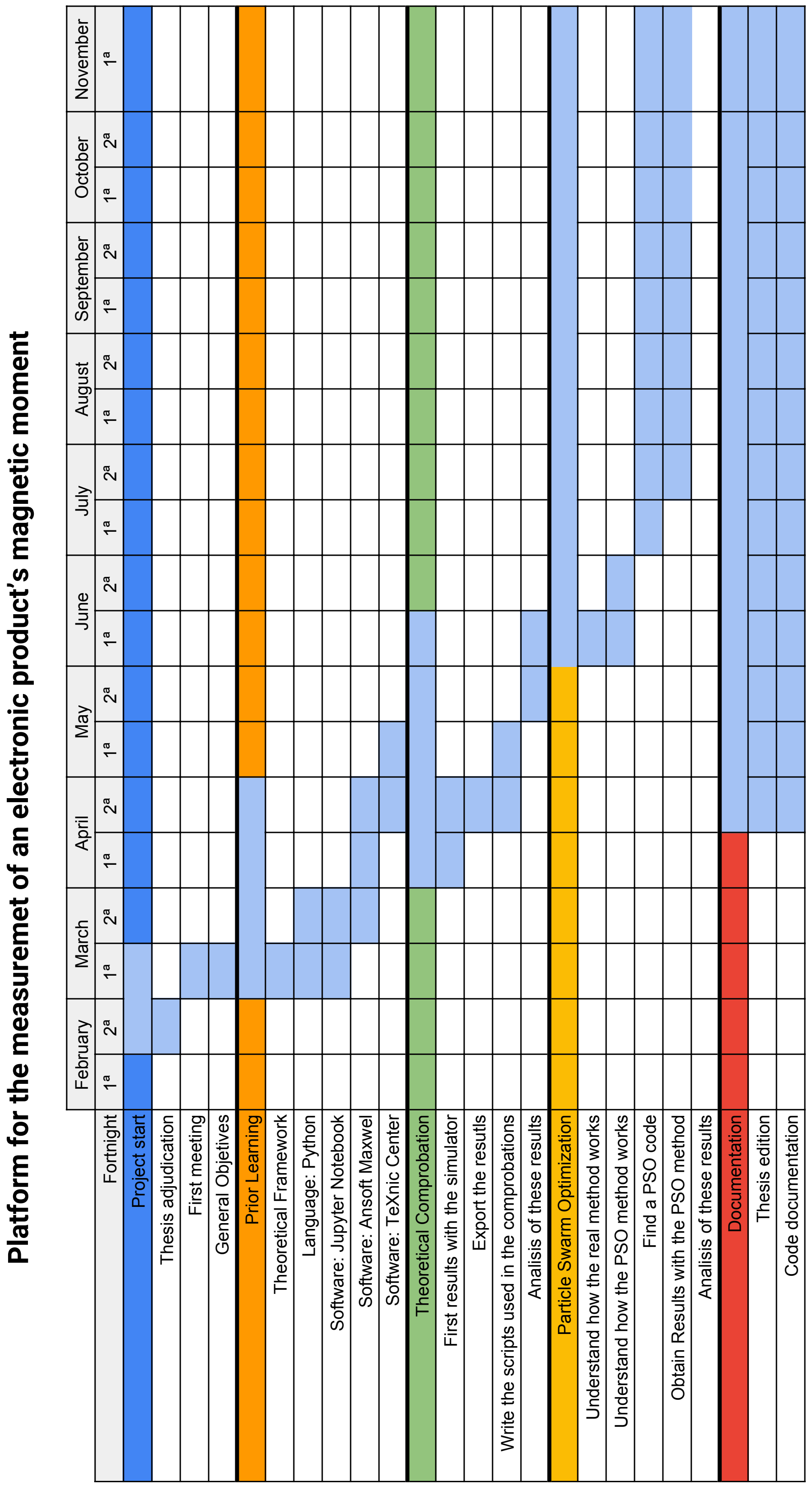


Figure 1.3 – *Gantt Chart of the Project*

Chapter 2

Theoretical framework

2.1 Magnetic Field

In this final degree project, topics related to electromagnetism are discussed, especially in the magnetostatic field. For this reason, one should start by mentioning the "Biot-Savart Law" [1] which is the fundamental equation of magnetostatics.

$$\vec{B} = \frac{\mu_0}{4\pi} \int_{V'} \frac{\vec{j}(\vec{r}) \times \vec{R}}{R^3} dV' \tag{2.1.1}$$

In this work the magnetic field $\vec{B}$ has been obtained by creating coils (closed circuits) through which a stationary current circulates.

Another important equation is the "Charge continuity equation" [1]:

$$\nabla \cdot \vec{j} + \frac{\partial \rho}{\partial t} = 0 \tag{2.1.2}$$

its says that there can only be a flow of current, $\vec{j}$, if the amount of charge ρ varies over time.

In a stationary current, there isn't a variation of the amount of charge neither a flow of current, this is:

$$\nabla \cdot \vec{j} = 0 \tag{2.1.3}$$

$$\frac{\partial \rho}{\partial t} = 0 \tag{2.1.4}$$

If we use the continuity equation [1] in his integral form:

$$\int_V \nabla \cdot \vec{j}\, dv = -\int_V \frac{\partial \rho}{\partial t}\, dv \tag{2.1.5}$$

and using the divergence theorem, $\int_V \nabla \cdot F\, dv = \oint_S F \cdot d\vec{s}$, the second term become:

$$\int_V \nabla \cdot \vec{j}\, dv = \oint_S \vec{j} \cdot d\vec{s} = 0 \tag{2.1.6}$$

With a stationary current only we can talk about a current that flows in a finite and close tube, a coil. The latter implies that:

$$I = \oint_{S'} \vec{j} \cdot d\vec{S'} \tag{2.1.7}$$

In addition, supposing that the coil section is very small, we will have to:

$$dV' = d\vec{S'} \cdot d\vec{l'} \tag{2.1.8}$$

With these two transformations we can rewrite equation 2.1.1 in a way that:

$$\vec{B} = \frac{\mu_0}{4\pi} \int_{V'} \vec{j}(\vec{r}) \times \frac{\vec{R}}{R^3} (d\vec{S'} \cdot d\vec{l'}) = \frac{\mu_0}{4\pi} \int_{V'} d\vec{l'} \times \frac{\vec{R}}{R^3} (\vec{j}(\vec{r}) \cdot d\vec{S'}) \tag{2.1.9}$$

Remaining [1]:

$$\vec{B} = \frac{\mu_0 I}{4\pi} \int \frac{d\vec{l'} \times \vec{R}}{R^3} \tag{2.1.10}$$

Where μ_0 is the magnetic permeability of the vacuum whose value is $\mu_0 = 4\,\pi \cdot 10^{-7}\,\mathrm{T\,mA^{-1}}$.

We would also need to define $\vec{R}$, this vector is nothing more than the vector subtraction of: $\vec{r}$ y $\vec{r'}$.

$$\vec{R} = \vec{r} - \vec{r'} \tag{2.1.11}$$

In a few words, it can be said that $\vec{r}$ is the vector that joins the origin with the point at which the magnetic field is to be measured and $\vec{r'}$ is the vector that joins the origin with the $d\vec{l'}$ that is originating that field. In the figure 2.1 these vectors have been schematized

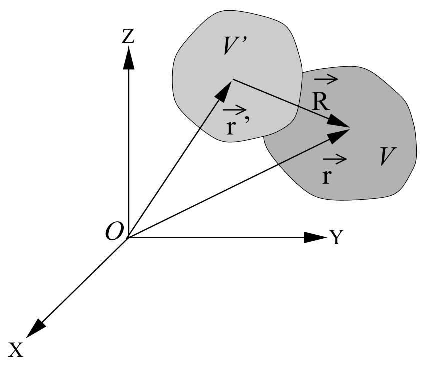


Figure 2.1 – *How to use the vectors $\vec{R}$, $\vec{r}$ y $\vec{r'}$ in order to calculate the magnetic field that V' generate in the volume V*

Applied to a coil, the vector $\vec{R}$ would be as shown in figure 2.2

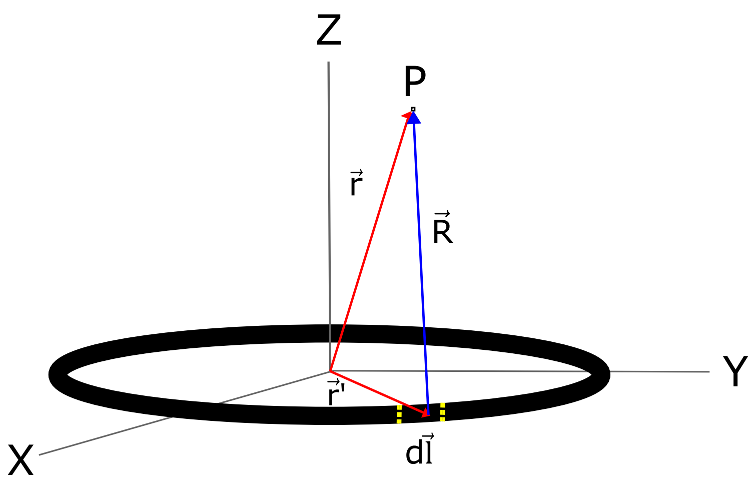


Figure 2.2 – *How to use the vectors $\vec{R}$, $\vec{r}$ y $\vec{r'}$ in order to calculate the magnetic field that $d\vec{l}$ generate in the point P*

2.2 Magnetic dipole

The central axis of this work is the use of magnetic dipoles to obtain the magnetic field generated by electronic components.

Before focusing on magnetic dipoles, it is essential to know what the potential vector of the magnetic field is.

On the basis of the equation 2.1.1 and knowing that $\frac{\vec{R}}{R^3} = -\nabla\frac{1}{R}$ we will have to:

$$\vec{B} = -\frac{\mu_0}{4\pi}\int_{V'} \vec{j} \times \nabla \frac{1}{R} dV' \tag{2.2.1}$$

According to Helmholtz's theorem [1] (Eq. 2.2.2, every vectorial field that tends to zero faster than r^{-1} when r approaches infinity, this will be determined by its scalar and vector sources.

$$\vec{F}(\vec{r}) = -\nabla f(\vec{r}) + \nabla \times \vec{g}(\vec{r}) \tag{2.2.2}$$

being $f(\vec{r})$ y $\vec{g}(\vec{r})$ a scalar potential and a vector potential, respectively

The magnetic field fulfills this condition, also according to the law of absence scalar sources for the magnetic field ($\nabla \cdot \vec{B} = 0$) it will only have vector sources and a vector potential [1].

$$\vec{B} = \nabla \times \vec{A} \tag{2.2.3}$$

Applying the following feature of the rotational to the equation 2.2.1:

$$\nabla\left(\frac{1}{R}\right) \times \vec{j} = \nabla \times \left(\frac{\vec{j}}{R}\right) - \frac{1}{R}\nabla \times \vec{j} \tag{2.2.4}$$

where the last term is canceled because it is a constant current.

Thus:

$$\vec{B} = -\frac{\mu_0}{4\pi}\int_{V'}\left(-\nabla \times \frac{\vec{j}}{R}\right) dV' = \nabla \times \left(\frac{\mu_0}{4\pi}\int_{V'}\frac{\vec{j}}{R}\, dV'\right) \tag{2.2.5}$$

Comparing this last result with the equation 2.2.3 we will have to [2]:

$$\vec{A} = \frac{\mu_0}{4\pi}\int_{V'}\frac{\vec{j}}{R}\, dV' \tag{2.2.6}$$

Once we know the vector potential of the magnetic field we can start approaching the multipolar development.

Supposing that we are going to measure the magnetic field at a point far from the current that creates it, this is $\vec{r} \gg \vec{r'}$, in section 3.3 we conclude that the optimal ratio (in terms of error) between $\vec{r}$ y $\vec{r'}$ is about 10 times ($\vec{r}\ /\ \vec{r'} \approx 10$). This last affirmation is discuss in the section 3.4

In this case, the vector that joins the origin with the measure point is roughly equal to the vector $\vec{R}$

$$\vec{R} = \vec{r} - \vec{r'} \approx \vec{r} \tag{2.2.7}$$

and its inverse can be approximated by a Taylor expansion [3] as:

$$\frac{1}{R} = \frac{1}{r} - r'\frac{\partial}{\partial r}\left(\frac{1}{r}\right) + \ldots \tag{2.2.8}$$

This development would leave us the vector potential as an infinite series of terms such that:

$$\vec{A} = \vec{A}_m + \vec{A}_d + \vec{A}_c + \ldots \tag{2.2.9}$$

These terms are known respectively as monopolar, dipolar, quadrupolar, octopolar potential, etc.

The monopolar term is zero because using the first term of the expansion $1/r$ in the equation 2.2.6, we would have [2]:

$$\vec{A}_m = \frac{\mu_0}{4\pi r}\int_{V'} \vec{j}(\vec{r'})\, dV' \tag{2.2.10}$$

At the beginning, I describe what is a stationary current and his basic property

$$\nabla \cdot \vec{j} = 0 \rightarrow \int_V \vec{j}\, dV' = 0 \tag{2.2.11}$$

Since the integral is nullified, the potential too, $A_m = 0$.

Now, focusing on the dipolar term, which is the one that interests us, we take the second term of Taylor's development:

$$-x'_j \frac{\partial}{\partial x_j}\left(\frac{1}{r}\right) = \frac{\vec{r'} \cdot \vec{r}}{r^3} \tag{2.2.12}$$

Remainging the dipolar potential as:

$$\vec{A}_d = \frac{\mu_0}{4\pi}\int_{V'} (\vec{r'} \cdot \vec{r})\vec{j}\, dV' \tag{2.2.13}$$

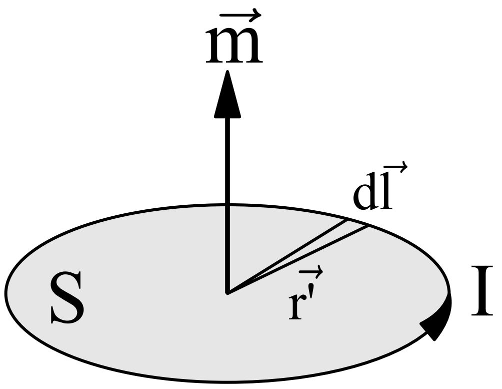


Figure 2.3 – *Since the current flows is counterclockwise the moment vector meaning is upwards.*

This last result can be transformed into [2]:

$$\vec{A}_d = \frac{\mu_0}{4\pi}\frac{\vec{m}\times\vec{r}}{r^3} = -\frac{\mu_0}{4\pi}\vec{m}\times\nabla\left(\frac{1}{r}\right) \tag{2.2.14}$$

Being $\vec{m}$ the magnetic dipolar moment, it is defined as:

$$\vec{m} = \frac{1}{2}\int_{V'}\vec{r'}\times\vec{j}\,dV' \tag{2.2.15}$$

If the same transformations are applied on the equation 2.1.10 this will remain as:

$$\vec{m} = \frac{1}{2}I\int\vec{r'}\times d\vec{l}' \tag{2.2.16}$$

And whose module:

$$m = IS \tag{2.2.17}$$

Where I is the current that flows in the coil and S is its surface. The magnetic moment has direction perpendicular to the coil's plane and meaning given by the right-hand rulewith the current flow. The figure 2.3 shows how this works

Finally, if we go back to the ratio between the magnetic field and the vector potential (Equation 2.2.3)

$$\vec{B}_d = \nabla\times\vec{A}_d = -\frac{\mu_0}{4\pi}\nabla\times\left[\vec{m}\times\nabla\left(\frac{1}{r}\right)\right] = \frac{\mu_0}{4\pi}(\vec{m}\cdot\nabla)\nabla\left(\frac{1}{r}\right) \tag{2.2.18}$$

In the same way as before we can turn $\nabla\frac{1}{r}$ into $-\frac{\vec{r}}{r^3}$ and the last result became [2]:

$$\vec{B}_d = -\frac{\mu_0}{4\pi}(\vec{m}\cdot\nabla)\frac{\vec{r}}{r^3} = -\frac{\mu_0}{4\pi}\nabla\left(\frac{\vec{m}\cdot\vec{r}}{r^3}\right) \tag{2.2.19}$$

This last result is the dipole magnetic field. All this theoretical development can be seen in greater depth in [1] and [2]

Since the gradient separates the result in each of the components, we would have:

$$\vec{B_{de_i}} = -\frac{\mu_0}{4\pi}\frac{\partial}{\partial e_i}\left(\frac{\vec{m}\cdot\vec{r}}{r^3}\right)\hat{e}_i \tag{2.2.20}$$

Where:

- $\vec{m}$ is the magnetic moment with his three cartesian component $\vec{m} = (m_x,\ m_y,\ m_z)$
- $\vec{r}$ is the vector that joins the dipole position $(x_0,\ y_0,\ z_0)$, with the measure point $(x,\ y,\ z)$.

$$\vec{r} = (x - x_0,\ y - y_0,\ z - z_0) \tag{2.2.21}$$

The zero subscripts marks the dipole position.

$$\vec{B_{de_i}} = -\frac{\mu_0}{4\pi}\frac{\partial}{\partial e_i}\left(\frac{m_x(x-x_0)+m_y(y-y_0)+m_z(z-z_0)}{[(x-x_0)^2+(y-y_0)^2+(z-z_0)^2]^{3/2}}\right)\hat{e}_i \tag{2.2.22}$$

Making the derivative will remain [4]:

$$B_x = -\frac{\mu_0}{4\pi}\left\{\frac{m_x}{[(x-x_0)^2+(y-y_0)^2+(z-z_0)^2]^{3/2}} - \frac{3(x-x_0)[m_x(x-x_0)+m_y(y-y_0)+m_z(z-z_0)]}{[(x-x_0)^2+(y-y_0)^2+(z-z_0)^2]^{5/2}}\right\} \tag{2.2.23}$$

$$B_y = -\frac{\mu_0}{4\pi}\left\{\frac{m_y}{[(x-x_0)^2+(y-y_0)^2+(z-z_0)^2]^{3/2}} - \frac{3(y-y_0)[m_x(x-x_0)+m_y(y-y_0)+m_z(z-z_0)]}{[(x-x_0)^2+(y-y_0)^2+(z-z_0)^2]^{5/2}}\right\} \tag{2.2.24}$$

$$B_z = -\frac{\mu_0}{4\pi}\left\{\frac{m_z}{[(x-x_0)^2+(y-y_0)^2+(z-z_0)^2]^{3/2}} - \frac{3(z-z_0)[m_x(x-x_0)+m_y(y-y_0)+m_z(z-z_0)]}{[(x-x_0)^2+(y-y_0)^2+(z-z_0)^2]^{5/2}}\right\} \tag{2.2.25}$$

Chapter 3

Verification of theoretical expressions

Once the theoretical development that leads to the dipole approach has been done, we will proceed to check its validity against the results that we can obtain with a simulator.

Since we cannot create a dipole in the simulator, we will need to create an object that can be induced to have a steady current. As we have seen in the previous section, this object is a coil.

With the help of the Maxwell [5] software we will create our coil and at a certain distance we will measure the magnetic field it generates, and on the other hand we will make the theoretical calculation of the field generated by a dipole at that same distance.

3.1 Obtaining the magnetic field

3.1.1 Simulation

First of all, we will obtain the results of the simulation. The main characteristics of the simulation are:

- Circular coil with a 10 mm radius, r.
- A current, I, of 100 A flows through the coil, counterclockwise.
- The sphere on which the magnetic field is measured has a 100 mm radius, R.

The process of creating the simulation and exporting the results is described in the **Appendix A**

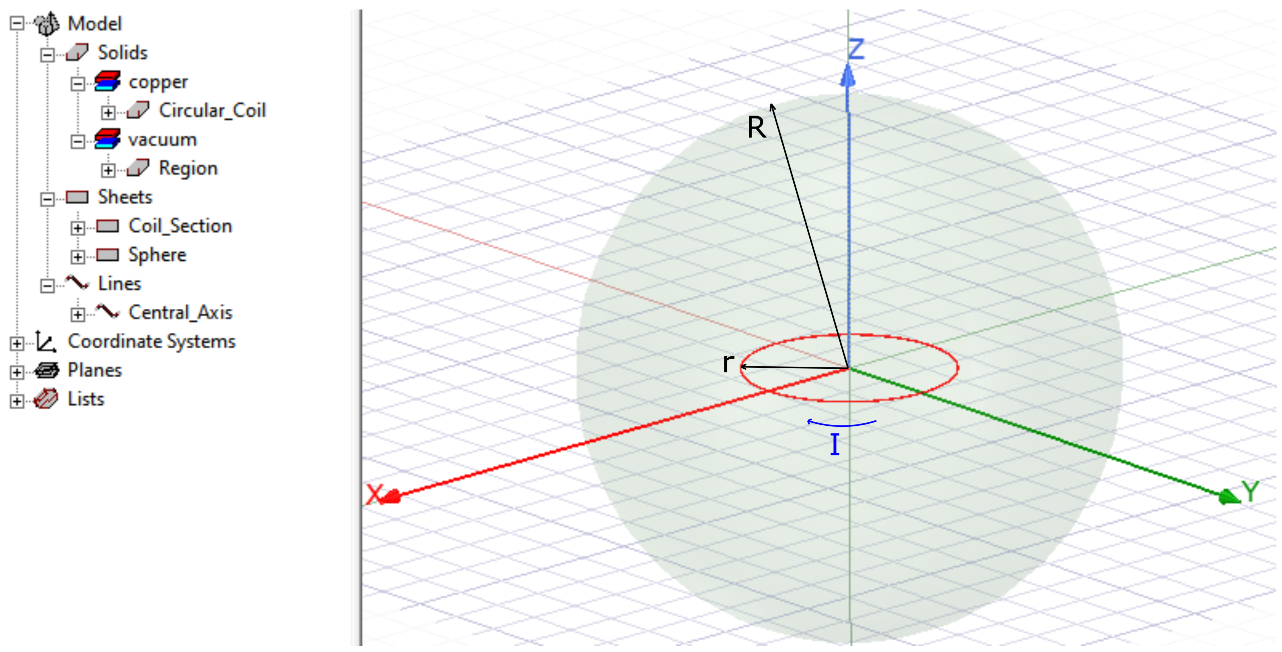


Figure 3.1 – *Simulation's elements*

Before starting it should be noted that all the codes are written in Python language, if you do not know the language I recommend reading [6] y [7].

In addition, the codes are created in Jupyter Notebook, an open-source tool that allows both compiling codes in Python and writing texts or equations in Latex. The main characteristics is the separation of codes in cells, that's why I present the code by "pieces".

First you have to import the libraries we need. In this process, shown in the code 3.1 have been used.

```python
# Libraries Used
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objects as go
import math
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from plotly.subplots import make_subplots
```

Code 3.1 – *Libraries used for the calculation and visualization of magnetic fields*

The most important library is plotly the one that allows to obtain the 3D plots from the magnetic field. This is a very powerful library with which we can obtain all kind of graphics. See [9] to learn more about this.

Once the results have been exported from the simulator, the magnetic field will be obtained, based on the code 3.2.

```python
# Read the file that contains the exported data
Maxwell_Export = "./DATA/B10mm_I100A_E100mm.txt"

# The file contains the simulated points in spheric coordenates
# and the Cartesian components of the magnetic field in this order
# R [m], theta [rad], phi [rad], Bx [T], By [T], Bz [T]
R = np.loadtxt(Maxwell_Export, usecols=[0])
tetha = np.loadtxt(Maxwell_Export, usecols=[1])
phi = np.loadtxt(Maxwell_Export, usecols=[2])
Bx = np.loadtxt(Maxwell_Export, usecols=[3])
By = np.loadtxt(Maxwell_Export, usecols=[4])
```

```
Bz = np.loadtxt(Maxwell_Export, usecols=[5])

# Magnetic field module in uT
B_sim = np.sqrt(Bx**2 + By**2 + Bz**2)*10**6
```

Code 3.2 – *Simulation magnetic field calculation*

3.1.2 Dipole Approximation

Thereupon, a magnetic field has been estimated by means of dipole approximation. Equations from 5 to 5 have been used for this purpose. However, it is relevant to take two factors into account which depend on the simulation:

- The coil is centred in the origin and therefore, the dipole coordinates will be: $(x_0 = y_0 = z_0 = 0)$
- The coil is positioned on the XY plane, and, as a consequence, it will only have a magnetic moment m which component will be Z, whose value is given according to the equation 2.2.17, by $m_z = I\pi r^2 = 31.42 \text{ mAm}^2$.

Rewriting them:

$$B_x = \frac{\mu_0}{4\pi}\frac{3xm_z z}{(x^2+y^2+z^2)^{5/2}} \tag{3.1.1}$$

$$B_y = \frac{\mu_0}{4\pi}\frac{3ym_z z}{(x^2+y^2+z^2)^{5/2}} \tag{3.1.2}$$

$$B_z = -\frac{\mu_0}{4\pi}\left\{\frac{m_z}{(x^2+y^2+z^2)^{3/2}} - \frac{3zm_z z]}{(x^2+y^2+z^2)^{5/2}}\right\} \tag{3.1.3}$$

In this equations, x, y and z are exact to the ones which were used in order to carry out the simulation. Therefore, we have to modify the spherical coordinates of the file into cartesian coordinates just as in code 3.3. The operation of spherical coordinates is schematised in figure 3.2.

```
# Calculation of x, y, z with their
# relationship in spherical coordinates
# For both methods X, Y, Z are the same
X = R*np.sin(tetha)*np.cos(phi)
Y = R*np.sin(tetha)*np.sin(phi)
Z = R*np.cos(tetha)
```

Code 3.3 – *Transform spherical coordinates to cartesian coordinates*

Now we can proceed with the magnetic field calculation with the dipole approximation using the equations from 3.1.1 to 3.1.3, following code 3.4:

```
# Calculation of the magnetic field
# using the dipole approximation
muo = 4*math.pi*10**(-7) # [T*m/A]
cte = muo/(4*math.pi)
I = 100 # [A]
```

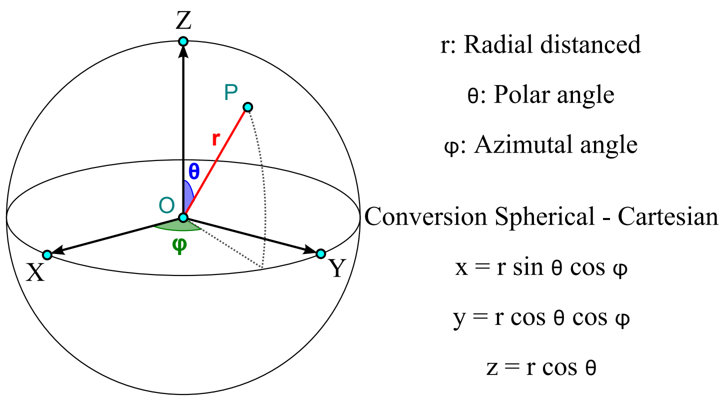


Figure 3.2 – *Spherical coordinates and its conversion to cartesian coordinates*

```
r = 0.01 # [m]

# Magnetic moment components [A*m^2]
mx = 0
my = 0
mz = I*math.pi*r**2

# Dipole position [m]
xo = 0
yo = 0
zo = 0

# R
R = (x**2+y**2+z**2)**(0.5)

# m \cdot R
mR = mx*(X-xo) + my*(Y-yo) + mz*(Z-zo)

Bx_dip = cte*((3*(X-xo)*mR)/R**5) #[T]
By_dip = cte*((3*(Y-yo)*mR)/R**5) #[T]
Bz_dip = -cte*(mz/R**3-(3*(Z-zo)*mR)/R**5) #[T]

# Magnetic field module in uT
B_dip = np.sqrt(Bx_dip**2 + By_dip**2 + Bz_dip**2)*10**6
```

Code 3.4 – *Dipole approximation magnetic field calculation*

3.2 Visualization of the results obtained

Once the magnetic fields have been calculated following the two methods, we can visualize them in 3D using codes 3.5 and 3.6, resulting in figure 3.3

```
# Simulation 3D-plot
Plot_B_sim = go.Figure(data=[go.Scatter3d(
    x=X*1000, y=Y*1000, z=Z*1000, #[mm]
    mode='markers',
    marker=dict(
        size=5,
        color=B_sim,
        colorscale='Rainbow',
        showscale=True,
        opacity=1
    ))])

Plot_B_sim.update_layout(
    title_text="uT",
    margin=dict(
        l=0, r=0, b=0, t=0
    ),

    scene = dict(
        xaxis = dict(
            title = 'x [cm]'),
        yaxis = dict(
            title = 'y [cm]'),
        zaxis = dict(
            title = 'z [cm]'),
        camera = dict(
            eye = dict(
                x = 2,
                y = 0.15,
                z = 1.3
            )
        )
    ),

)
Plot_B_sim.show()

#Save the plot as svg
Plot_B_sim.write_image("IMAGES/Plot_B_Sim.svg")
```

Code 3.5 – *Simulation magnetic field 3D-plot*

```
# Dipole aproximation 3D-plot
Plot_B_dip = go.Figure(data=[go.Scatter3d(
    x=X*1000, y=Y*1000, z=Z*1000, #[mm]
    mode='markers',
    marker=dict(
        size=5,
        color=B_dip,
        colorscale='Rainbow',
        showscale=True,
        opacity=1
    ))])

Plot_B_dip.update_layout(
    title_text="uT",
    margin=dict(
        l=0, r=0, b=0, t=0
    ),

    scene = dict(
        xaxis = dict(
            title = 'x [cm]'),
        yaxis = dict(
            title = 'y [cm]'),
        zaxis = dict(
            title = 'z [cm]'),
        camera = dict(
            eye = dict(
                x = 2,
                y = 0.15,
                z = 1.3
            )
        )
    ),

)
Plot_B_dip.show()

#Save the plot as svg
Plot_B_dip.write_image("IMAGES/Plot_B_dip.svg")
```

Code 3.6 – *Dipole aproximation magnetic field 3D-plot*

3.3 Field variation and error between methods

Finally, with the formula of the relative error

$$E_{\%} = \frac{|B_{sim} - B_{dip}|}{B_{dip}} * 100 \tag{3.3.1}$$

We can study how the results differ between both methods.

Code 3.7 builds a graph that shows how both magnetic fields vary with respect to the position on the z axis and the error that exists between both results (Figure 3.4)

```
# Calculation of the error between the simulation and the dipole approximation
Err_sim_dip = abs(B_sim-B_dip)*100/B_dip

```

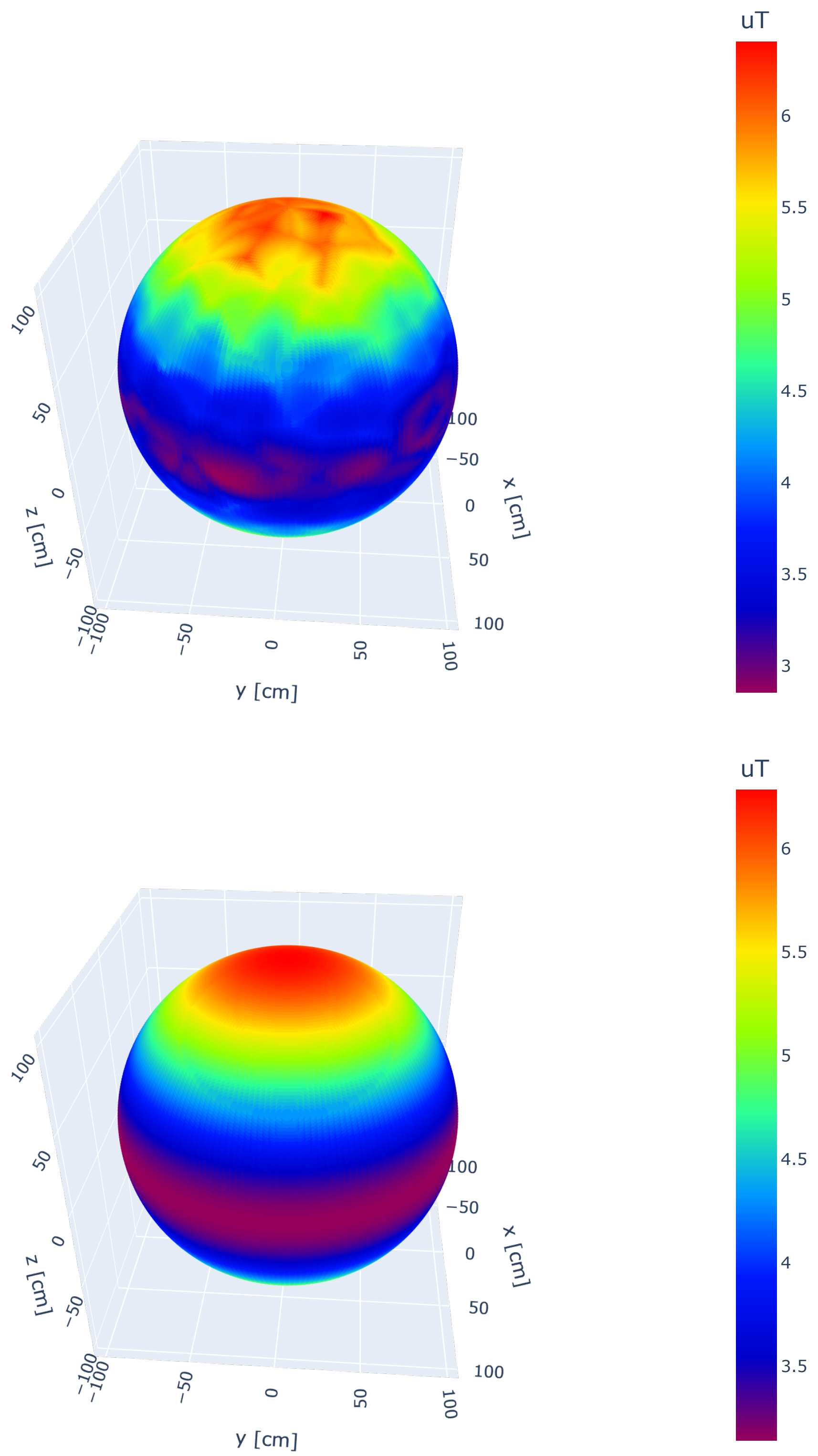


Figure 3.3 – *Three-dimensional plots showing the magnetic field modulus in uT obtained with the Maxwell software (top) and calculated using the equations of the dipole approximation (bottom)*

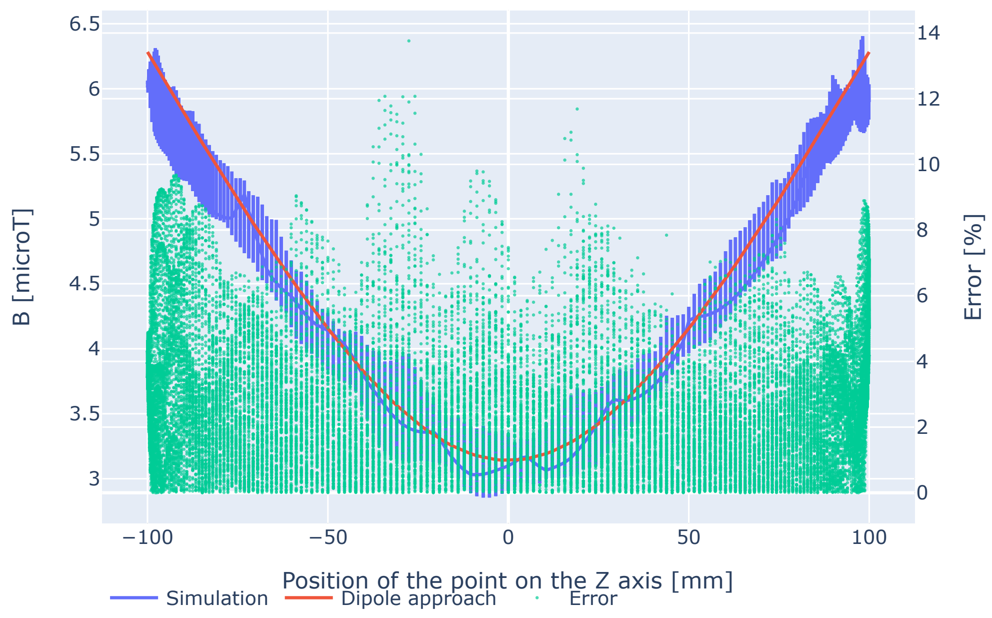


Figure 3.4 – *Variation of magnetic field at a distance of 100 mm as a function of the position on the Z axis (lines), and the error (points) between both methods used to obtain them.*

```
# Plot creation
# Set the secondary Y axis for the error values
PlotError = make_subplots(specs=[[{"secondary_y": True}]])

PlotError.add_trace(go.Scatter(
                    x=Z*1000, y=B_sim, #mm, uT
                    mode='lines',
                    name='Simulation'),
                    secondary_y=False)

PlotError.add_trace(go.Scatter(
                    x=Z*1000, y=B_dip, #mm, uT
                    mode='lines',
                    name='Dipole approach'),
                    secondary_y=False)

PlotError.add_trace(go.Scatter(
                    x=Z*1000, y=Err_sim_dip, #mm, %
                    mode='markers',
                    name='Error',
                    marker=dict(
                        size=2,
                        opacity=0.7)),
                    secondary_y=True)

PlotError.update_layout(
    legend=dict(orientation="h")
)
PlotError.update_xaxes(title_text="Position of the point on the Z axis [mm]")
PlotError.update_yaxes(title_text="B [microT]", secondary_y=False)
PlotError.update_yaxes(title_text="Error [%]", secondary_y=True)

PlotError.show()

#Save the plot as svg
PlotError.write_image("IMAGES/PlotError.svg")
```

Code 3.7 – *Creation of the plot that shows variation of the magnetic field with z and error between methods*

3.4 Connection between coil radius and measurement distance

In addition to those shown, during this part numerous simulations have been carried out to find the optimal ratio between the radius of the simulated coil and the distance over which the magnetic field is measured.

The dipole approximation is valid if the distance to the measurement point is greater than the coil radius (See section 2.2), therefore the ratio between distance and radius that has been tested are: double, five times more, ten times more and twenty times more.

Based on the same code that creates the figure 3.4 (Code 3.7), we can create some graphs that show us the errors of the other reactions.

Comparing the results of figure 3.5 with those obtained in figure 3.4 it can be seen that the ratio that generates the least error between the simulation and the dipole approximation is that of a measurement distance ten times greater than the radius.

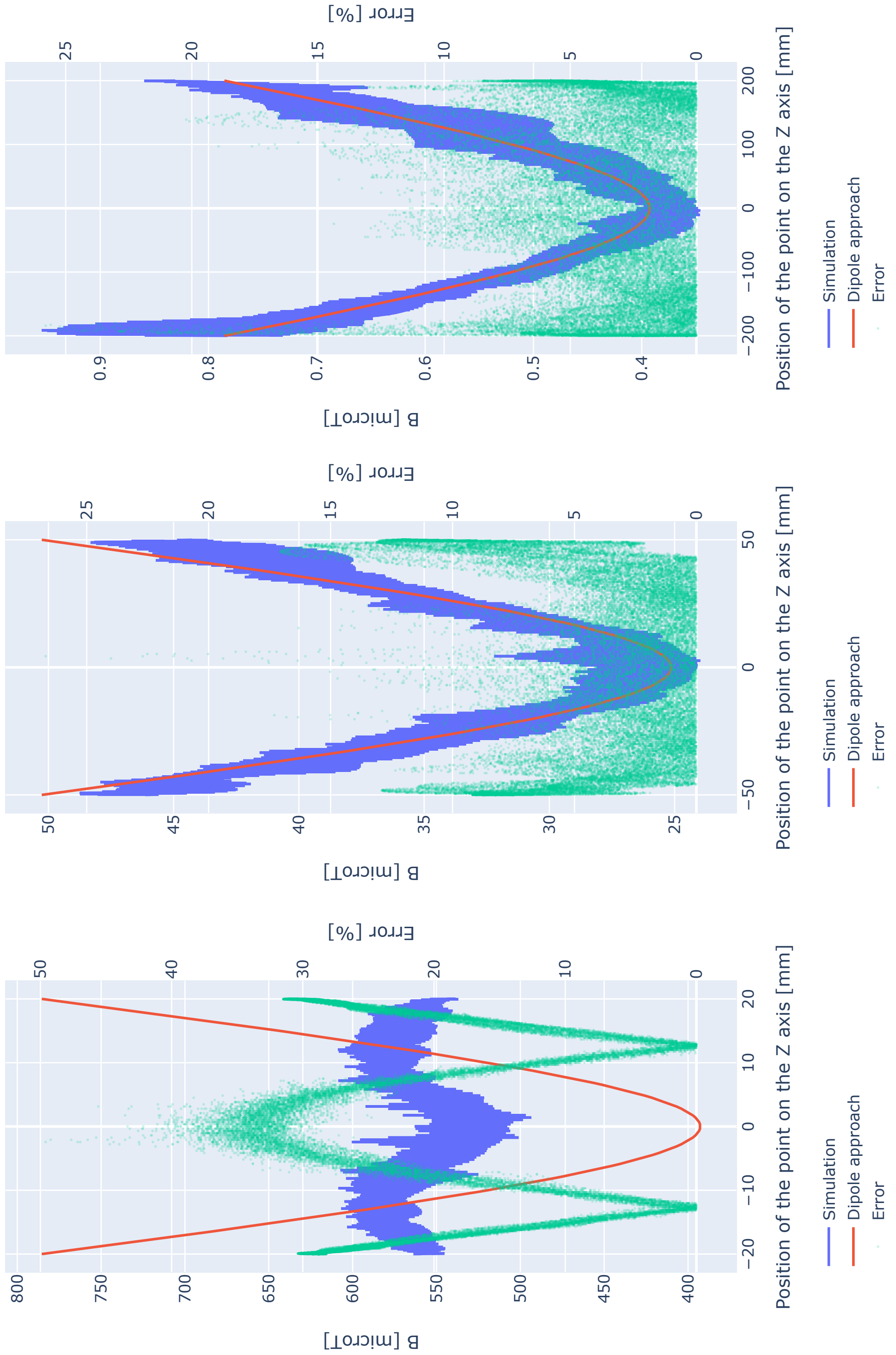


Figure 3.5 – *Magnetic field variation depending on the Z axis position, concerning the ratio between distance and radius x2 (20 mm), x5 (50 mm), x20 (200 mm) respectively.*

Chapter 4

Particle Swarm Optimization

Once the validity of the equations which link the magnetic moment with the magnetic field has been checked (Eq. 5 a 5), we can proceed dealing with the main topic of this thesis which is, the magnetic field estimation produced by dipoles.

Before focusing on the steps to follow in order to obtain the approximatoin, it is necessary to talk about the computing method used in the last step: the Particle Swarm Optimization method (PSO).

The PSO method is a heuristic optimization method used in obtaining global extremes. As its name indicates, it is inspired by a swarm where the movement of the individual is determined by the behavior of the collective.

4.1 Algorithm

In general terms this method consists of:

1. Creation of the swarm composed of n particles whose position vector is random. This vector not only collects values of the position of the particle but also contains the initial values of the quantities to be optimized. In our case, this vector has 6 component which are the initial position (x_0, y_0, z_0) and the component of the magnetic dipole moment (m_x, m_y, m_z).

2. Evaluate each particle in the function to optimize, this function is call fitness function.

3. Update the speed and position vector according to the equations described below (Eq. 4.1.1 and 4.1.2).

4. Check if the given stop criterion is met, and if not go back to step 2.

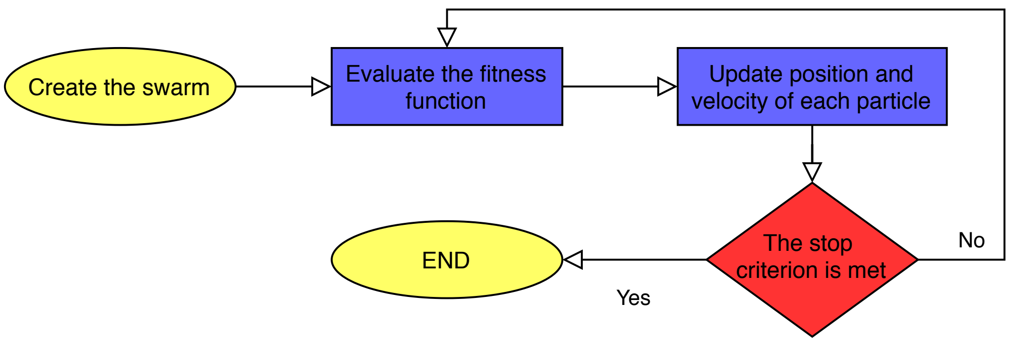

Create the swarm
Evaluate the fitness function
Update position and velocity of each particle
The stop criterion is met
No
Yes
END

Figure 4.1 – *Particle swarm optimization flowchart*

The speed and position of each of the swarm particles are updated according to the following equations:

$$V_i(t+1) = wV_i(t) + c_1\phi_1(P_{best_i} - x_i(t)) + c_2\phi_2(P_{best_S} - x_i(t)) \tag{4.1.1}$$

$$x_i(t+1) = x_i(t) + V_i(t+1) \tag{4.1.2}$$

- w is the inertial weight, determine how the particle maintains its original course.
- $c_1 \in [0,1]$ is the cognitive rate that determines how much the particle is influenced by the memory of its best location.
- $c_2 \in [0,1]$ is the social rate that decides how much the particle is influenced by the rest of the swarm.
- P_{best_i} is the best location found by the single particle.
- P_{best_S} is the best location globally found by the swarm.
- ϕ_1 and ϕ_2 are two random functions equally distributed in the range (0, 1)

The image 4.1 is a flowchart that show how the PSO algorithm works

4.2 PSO application

In order to obtain the above-mentioned estimation, it is required to follow the next steps:

1. Dipole creation by means of Maxwell Software, as shown in **Appendix A**.
2. Magnetic field assessment, generated by the dipoled at the desired distance.

3. Swarm generation, as described in the **Appendix B**.

4. Particle assessment with the Fitness Function, F_1.

$$F_1 = \frac{\sqrt{\sum_{i=1}^{M} \left[\left(B_{Sx}(i) - B_{Px}(i)\right)^2 + \left(B_{Sy}(i) - B_{Py}(i)\right)^2 + \left(B_{Sz}(i) - B_{Pz}(i)\right)^2 \right]}}{\sqrt{\sum_{i=1}^{M} \left(B_{Sx}^2(i) + \tilde{B}_{Sy}^2(i) + \tilde{B}_{Sz}^2(i) \right)}} \tag{4.2.1}$$

B_{Si} would be the magnetic field obtained with the simulation and B_{Pi} would be the magnetic field assessed for each of the particles randomly generated.

4.2.1 Example

Based on the event described in [10], where a dipole is obtained with following characteristics:

$$\begin{aligned} (x_d, y_d, z_d) &= (0,\ 0,\ 0)\ \text{cm} \\ (m_x, m_y, m_z) &= (150,\ -200,\ 180)\ \text{mAm}^2 \end{aligned} \tag{4.2.2}$$

In order to create this kind of dipole with the simulator, we simply need to apply the superposition principle. Therefore, we would need to create three coils centred on the origin, each of them being on a different plane, with an only magnetic moment component though.

It is relevant to obtain the magnetic field at a distance of 30 cm from the origin. As shown in the previous part of the thesis, using the results obtained from figures 3.4 and 3.5, the optimal ratio between the measured distance and radius must be ten times bigger. Therefore, we will first create a 3 cm radius coil on the XY plane and this will give us the dipole moment on the Z axis, and according to the equation 2.2.17 we will obtain the density that will traverse the coils.

$$I = \frac{m}{S} = \frac{m}{r^2\pi} = 200/\pi \text{ A}$$

We will reapply the same equation in order to calculate the radius from the other two coils:

- The $m_x = 150$ mAm^2 one in the YZ plane, with a 2.7 cm radius.
- The $m_y = -200$ mAm^2 one in the XZ plane, with a 3.1 cm radius.
- The $m_z = 180$ mAm^2 one in the XY plane, with a 3 cm radius.

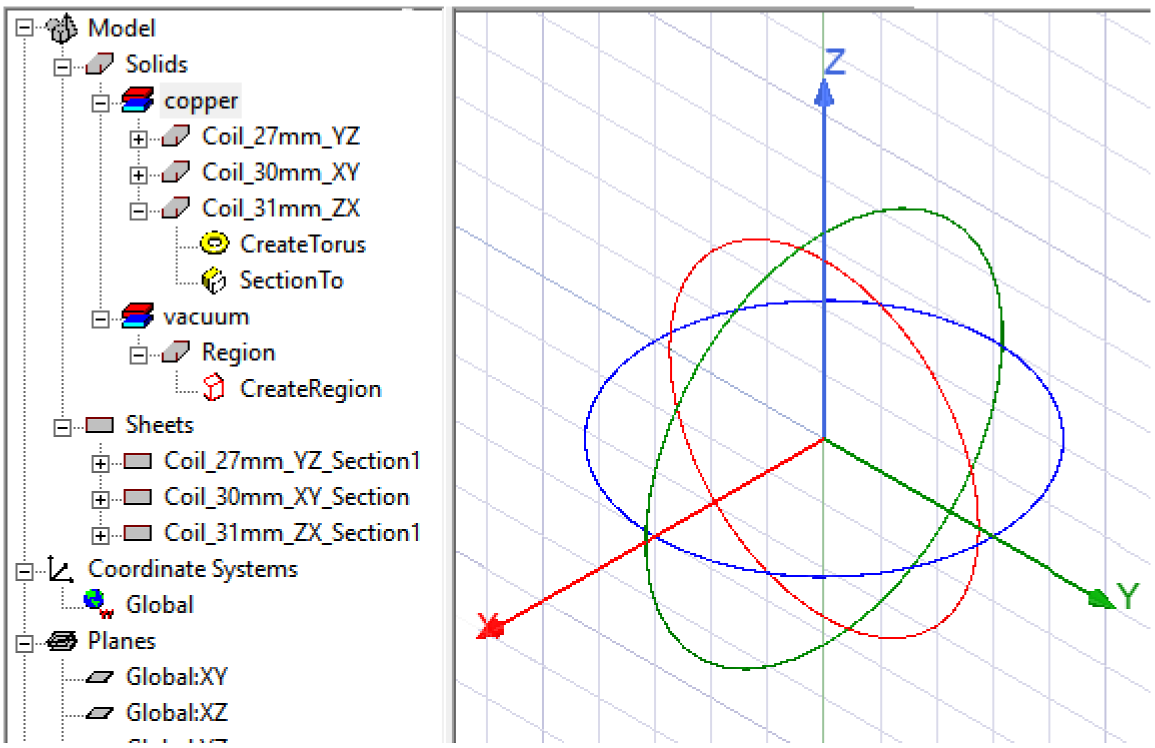


Figure 4.2 – *Layout of the three coils to generate the magnetic dipole given in the equation 4.2.2. The color of each coil indicates the component to which it contributes dipole moment.*

The direction of the current is counterclockwise for those with positive m_i and clockwise for negative.

The image 4.2 shows how the coils are arranged in the simulator.

In reality, this arrangement would create the magnetic coupling between the coils, but to simplify the procedure this effect hasn't been considerated.

Once the simulation is complete, the magnetic field is exported on a sphere of radius 30 cm. The generated magnetic field is shown in figure 4.4.

For practical purposes it is more useful to export a single circumference, since the computation time is drastically reduced. In the figure 4.3 the values obtained with the simulation for the polar angle, θ, of 10° (Remember the figure 3.2) have been represented.

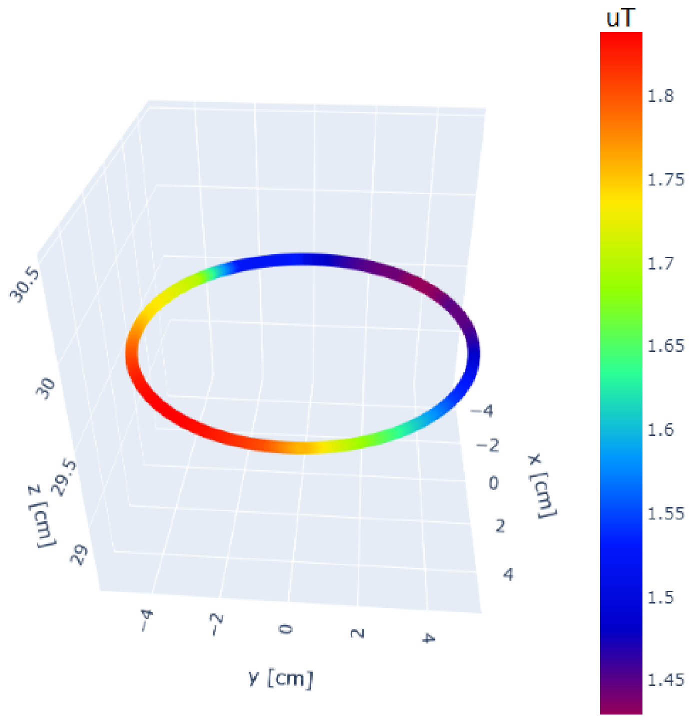
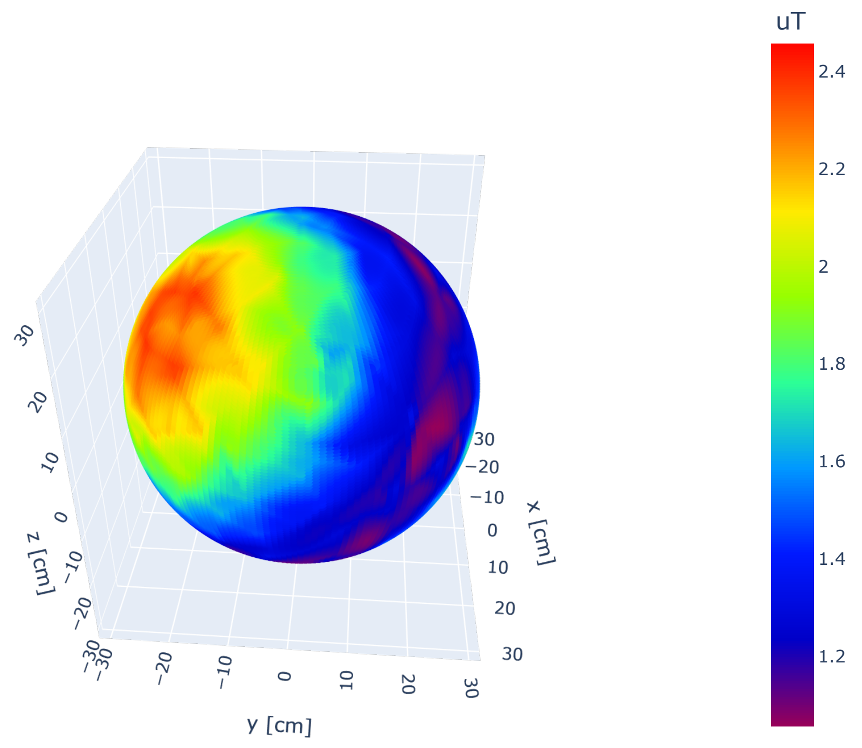


Figure 4.3 – *To reduce the computation time we use only the values of* $\theta = 10$ *and* $\phi = [0, 360]$

Thanks to that it's possible to represent in the figure 4.5 the values of the components of the magnetic field with respect to the azimuth angle, $\phi = [0^{\circ}, 360^{\circ}]$, both simulated value and theoretical value.



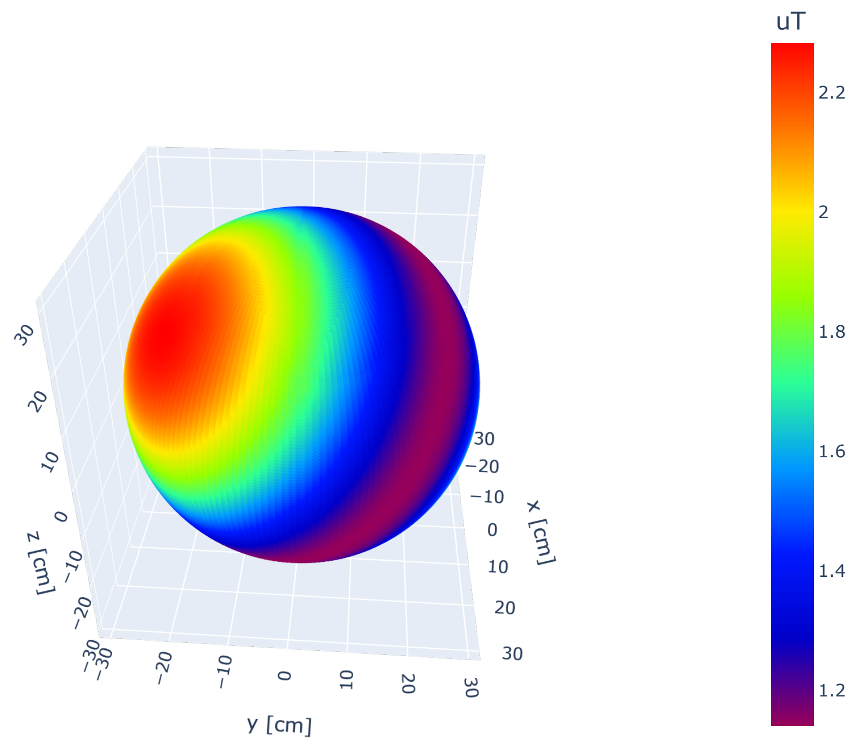


Figure 4.4 – *Three-dimensional plots showing the modulus of the magnetic field in uT generated by the three coils at a distance of 30 cm from the origin of coordinates, obtained with the Maxwell software (top) and calculated using the equations of the dipole approximation (bottom)*

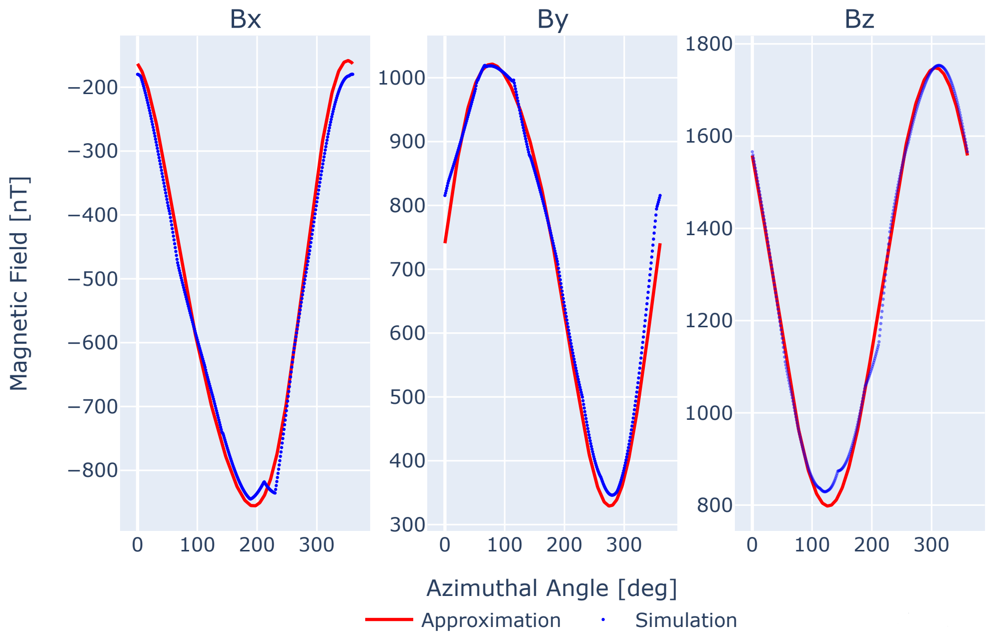


Figure 4.5 – *Behavior of the components of the magnetic field as a function of the azimuth angle. The theoretical result according to the dipolar approximation is shown in red (solid line) and the results obtained with the simulator are shown in blue (dots).*

Also in the figure 4.6 the error between these results.

Once we have obtained our simulated magnetic field, we have to create our swarm and evaluate it according to the Fitness Function, equation 4.2.1.

For this example a swarm of 32 particles has been used which were limited in the position of $[-15, 15]$ cm in each direction and for the magnetic moment of $[-800, 800]$ mAm^2 in each component.

In the **Appendix B** is the PSO code used, you just have to add the following codes at the end:

```
#Fitness Funtion definition
def F1(x_0, y_0, z_0, mx, my, mz):  # 6 component position vector
    cte = -10**(-7) # Dipole approximation calculation
    R = ((X-x_0)**2 + (Y-y_0)**2 + (Z-z_0)**2)**(0.5)
    mR = mx*(X-x_0) + my*(Y-y_0) + mz*(Z-z_0)
    Bx_PSO = cte*(mx/R**3-(3*(X-x_0)*mR)/R**5) #[T]
    By_PSO = cte*(my/R**3-(3*(Y-y_0)*mR)/R**5) #[T]
    Bz_PSO = cte*(mz/R**3-(3*(Z-z_0)*mR)/R**5) #[T]
    l = len(Bx) - 1
    num = 0
    den = 0
    for i in range (0,l):
        num = num + (Bx[i]-Bx_PSO[i])**2 + (By[i]-By_PSO[i])**2 + (Bz[i]-Bz_PSO[i])**2
        den = den + Bx[i]**2 + By[i]**2 + Bz[i]**2
        F1 = (num)**0.5/(den)**0.5
    return(F1)
```

Code 4.1 – *Fitness Funtion definition*

```
swarm = Enjambre(
                n_particulas = 32, # particles number
                n_variables  = 6, # component number
                limites_inf  = [-15, -15, -15, -800, -800, -800], # limits
```

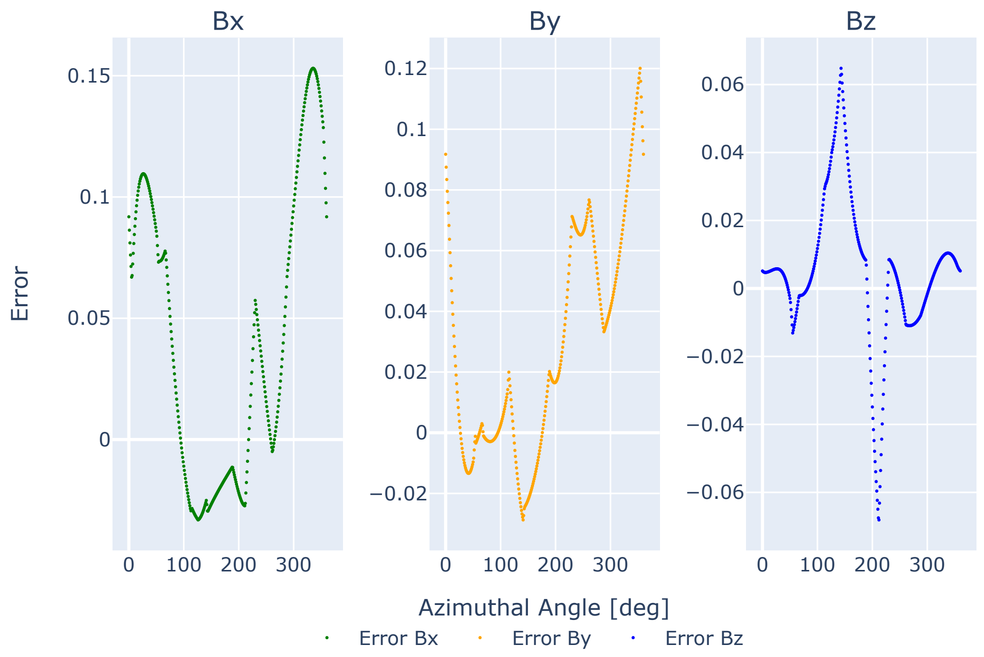


Figure 4.6 – *Simulator magnetic field component error and dipole approximation*

```
                limites_sup   = [15, 15, 15, 800, 800, 800], # [cm,,, mAm^2,,]
                verbose       = False
            )
```

Code 4.2 – *Creation of the swarm with its properties*

```

enjambre.optimizar(
    funcion_objetivo  = F1,
    optimizacion      = "minimizar",
    n_iteraciones     = 1000,
    inercia           = 1, # w
    reduc_inercia     = False, # w is not reduced
    inercia_max       = 1,
    inercia_min       = 1,
    peso_cognitivo    = 0.5, #c_1
    peso_social       = 0.5, #c_2
    parada_temprana   = False, # If you want the optimization to
    # stop earlier than expected
    rondas_parada     = 5, # Stop if X iterations occur with
    # variation less than the tolerance
    tolerancia_parada = 10**-5, # tolerance
    verbose           = False
)
```

Code 4.3 – *Call to the optimization function*

Once the program is finished, the result is that the dipole has the following characteristics:

$$\begin{aligned}(x_d, y_d, z_d) &= (1.9,\ 0.3,\ -5.2)\ \text{cm} \\ (m_x, m_y, m_z) &= (184,\ -375,\ 315)\ \text{mAm}^2\end{aligned} \tag{4.2.3}$$

We can also obtain a graph that shows the convergence of the solution (Figure 4.7) by adding this code:

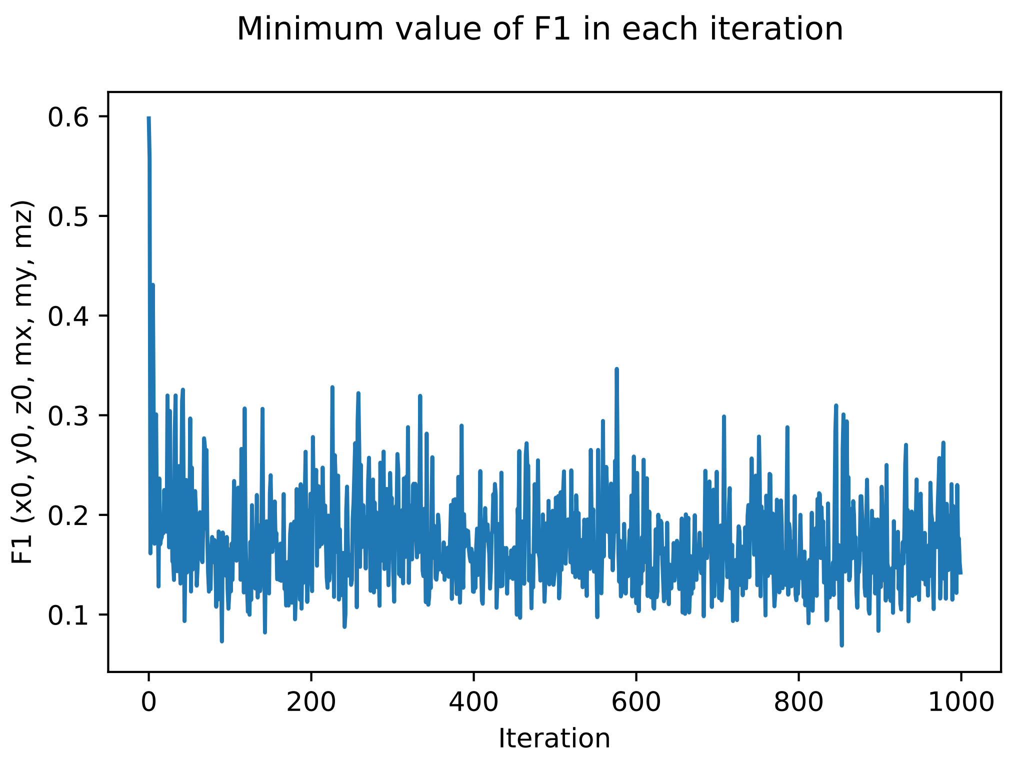


Figure 4.7 – *Solution convergence plot*

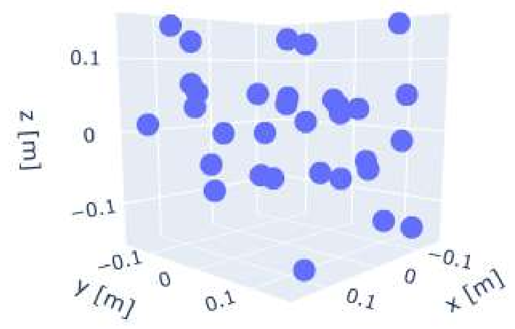
```python
#Solution Convergence Plot
fig = plt.figure(figsize=(6,4))
plt.xlabel('Iteration')
plt.ylabel('F1 (x_0, y_0, z_0, mx, my, mz)')
fig.suptitle('Minimum value of F1 in each iteration')
enjambre.resultados_df['mejor_valor_enjambre'].plot()
```

Code 4.4 – *Script to create a solution convergence plot*

And finally, you can create an animation that shows the movement of the particles in each iteration (Video 4.1):

```python
# Animated graphic particle evolution representation

# The position of the particles is
# extracted in each iteration of the swarm
import plotly.express as px

def extraer_posicion(particula):
    pos = particula.posicion
    return(pos)

lista_df_temp = []

for i in np.arange(len(enjambre.historico_particulas)):
    poss = list(map(extraer_posicion,
        enjambre.historico_particulas[i]))
    df_temp = pd.DataFrame({"iteracion": i, "pos": poss})
    lista_df_temp.append(df_temp)

df_poss = pd.concat(lista_df_temp)

df_poss[['x_0','y_0', 'z_0', 'mx', 'my', 'mz']] = pd.DataFrame(df_poss["pos"]
.values.tolist(),
                                            index= df_poss.index)

s = 0.16 # Plot size
fig = px.scatter_3d(
    df_poss,
    x       = "x_0",
    y       = "y_0",
    z        = "z_0",
    labels={
                     "x_0": "x [m]",
                     "y_0": "y [m]",
                     "z_0": "z [m]"
    },
    range_x = [-s, s],
    range_y = [-s, s],
    range_z = [-s, s],
    animation_frame = "iteracion"
)

fig.update_xaxes(fixedrange=True)
```

Code 4.5 – *Animated position plot*



Video 4.1 – *Animation that show how the PSO method works (double click for view and another double click for fullscreen)*

Chapter 5

Conclusion

If we recapitulate, in this thesis we have started from the basic law of magnetostatics, the Biot-Savart law (Eq. 2.1.1), and we have developed it to obtain the equations that allow obtaining the magnetic field produced by a magnetic dipole (Eq. -).

We have also learned to use the Maxwell simulator, to model coils that act as magnetic dipoles and to be able to compare with the results obtained with the theoretical expressions, and as we have seen in section 3.4 the best results were obtained at a distance 10 times greater than the radius of the coils. In this same area, it has also been learned what to do in the case of having a moment with 2 of its 3 non-zero components (Eq. 4.2.2) and how to create a simulation that models a valid dipole.

Finally, in Chapter 4 we have the swarm method (PSO) that we now know a little better. We have started from its base and thanks to the codes in section 4.2.1 and **Appendix B** we can find the magnetic dipole that will simplify an electronic product, from the measurements that we take of its magnetic field.

It is true that comparing the values of equations 4.2.2 and 4.2.3, it is observed that the results for the position of the dipole obtained are close to what was sought, but in the case of the values corresponding to the magnetic moment, m_i, they are not good at all.

The fact that the results do not correspond to the sought values is due to the fact that the code that calculates the magnetic fields of the particles created during the PSO method is still in the debugging phase. Later, when it is possible to test the experimental measurements obtained, the process can be completed and thus be able to execute the algorithm correctly and obtain more reliable measurements.

Appendix A

Coil simulation with Ansoft Maxwell [11] software

In sections 3.1 and 4.2 we use the software Ansoft Maxwell to create simulations, which will give us results that we will have to export to work with them. These simulations are intended to recreate the behavior of a coil in which an electric current is induced, generating a magnetic field.

In order to launch the simulation, we will create four sections by using the software: a toroid, a region, a line –which will work as the coil axis- and, a spherical surface.

We will produce the coil by means of a 98 mm inner radius and a 100 mm external radius toroid. Once the coil is created, we proceed selecting and right-clicking the toroid, "Assign Material. . . " and we assign copper to it. As we see in the figure A.1

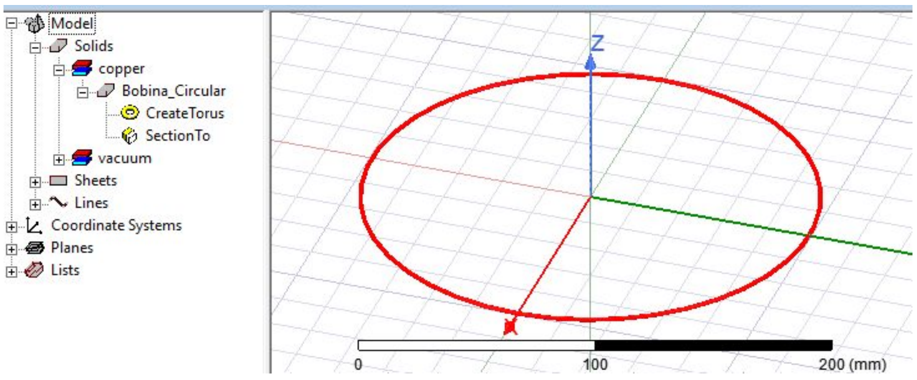
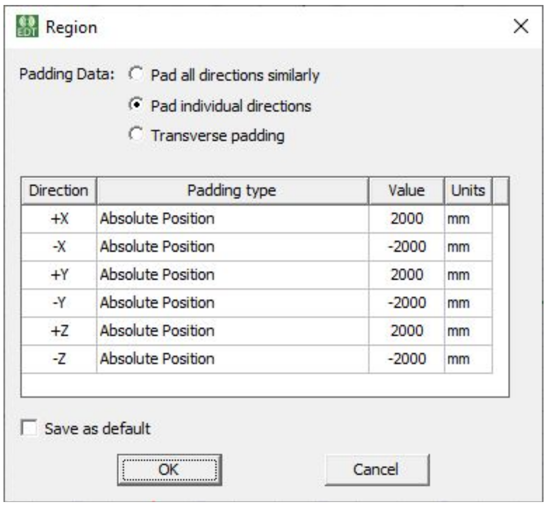


Figure A.1 – *100 mm radius coil*

We will now create the intregation region. In order to do so, we will select "Pad Individual Directions" in the pop-up window and for each coordinate we will select "Absolute Position" in +X = 2000 mm, -X = -2000 mm, +Y = 2000 mm, -Y = -2000 mm,



Padding Data: Pad all directions similarly / [✓ Pad individual directions] / Transverse padding

Direction	Padding type	Value	Units
+X	Absolute Position	2000	mm
-X	Absolute Position	-2000	mm
+Y	Absolute Position	2000	mm
-Y	Absolute Position	-2000	mm
+Z	Absolute Position	2000	mm
-Z	Absolute Position	-2000	mm

Figure A.2 – *Region parameters*

+ Z = 2000 mm, -Z = -2000 mm. (See figure A.2)

We will next create a line from (0,0,-1000) to (0,0,+1000) that will work as the sprial's central axis.

Lastly, we will draw a solid sphere that we will later transform into a spherical shell. We need to create a centered sphere with a 400 mm radius. It must stay within the integration region where Maxwell equations are solved.

We will now proceed to prepare the sphere to empty it. We will click MOUSE and RMS (right-click) and select FACES mode

We right-click on the sphere again and select Edit→Surface→Detach Faces. A new category "Unclassified" should have been created in the "Project Manager" section. We can proceed to delete the former.

The Project Manager section should remain as in Figure A.3

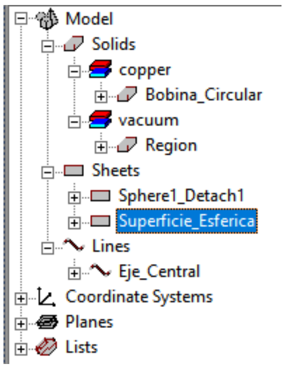


Figure A.3 – *Current Project Manager*

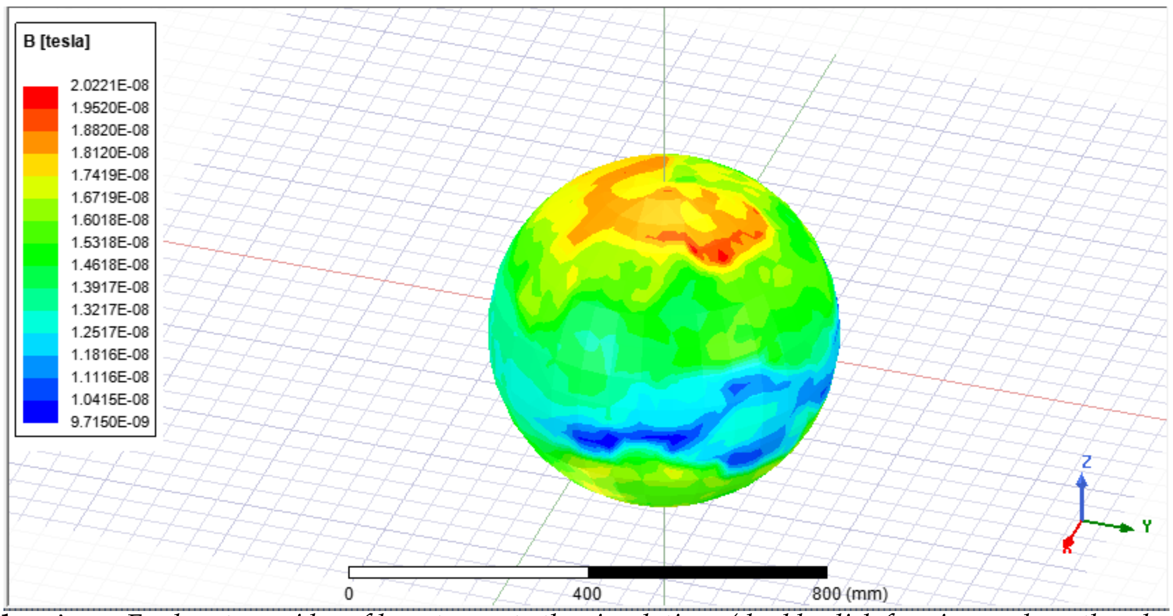
As for current source features, we will reselect the toroid and click

Modeler → Surface → Section → YZ. We will now have two different sections of the toroid when only needing one. In order to delete the spare one: Modeler → Boolean → Separate Bodies, and we will be able to delete it.

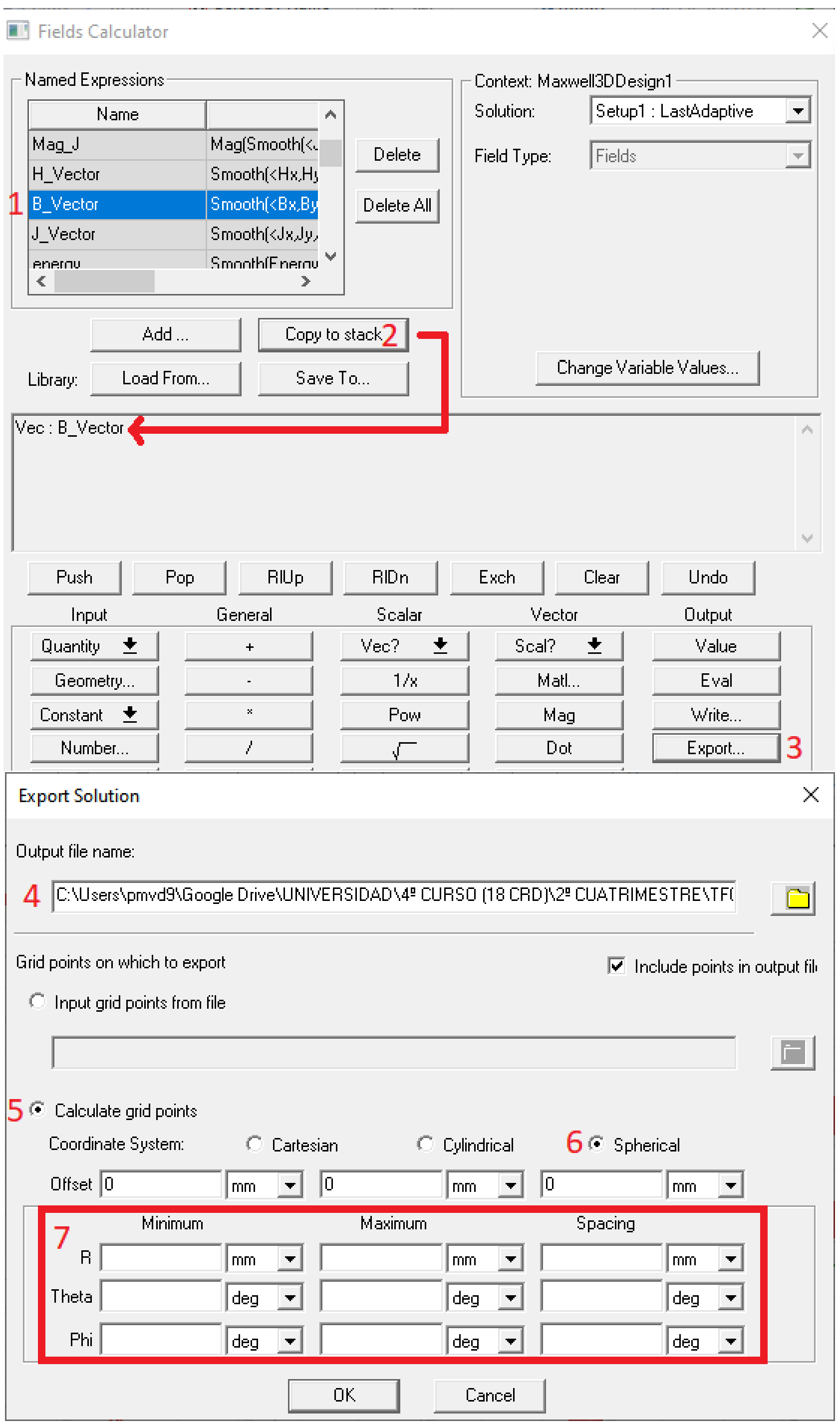
After having deleted one of the sections from the toroid, we will continue assigning a 100 A current to the section: we will right-click and Assign Excitation → Current

We can now begin with the simulation by clicking on Maxwell 3D → Analyze All. Once the simulation is over we will proceed to export the simulation results – Maxwell 3D → Fields → Calculator- by following the steps in the figure A.4

Finally, if you need visual help to create the simulations, video A.1 will help you.



Video A.1 – *Explanatory video of how to create the simulations (double click for view and another double click for fullscreen)*



Name	
Mag_J	Mag(Smooth(<.
H_Vector	Smooth(<Hx,Hy
B_Vector	Smooth(<Bx,By
J_Vector	Smooth(<Jx,Jy,
energy	Smooth(Energy

Context: Maxwell3DDesign1
Solution: Setup1 : LastAdaptive
Field Type: Fields

Vec : B_Vector

Output file name: C:\Users\pmvd9\Google Drive\UNIVERSIDAD\4º CURSO (18 CRD)\2º CUATRIMESTRE\TF(

[✓] Include points in output fil

Input grid points from file

[✓ Calculate grid points]

Coordinate System: Cartesian Cylindrical [✓ Spherical]

Offset	0 mm	0 mm	0 mm

	Minimum	Maximum	Spacing
R	mm	mm	mm
Theta	deg	deg	deg
Phi	deg	deg	deg

Figure A.4 – *Steps to export the simulation's data*

Appendix B

Particle Swarm Optimization Code [12]

The PSO code use in this proyect was created by Joaquin Amat Rodrigo [12], he created two class: the particle class and the swarm class.

For more information, in Spanish, about his code and examples go to `https://github.com/JoaquinAmatRodrigo/optimizacion_PSO_python/blob/master/PSO_python.ipynb`

B.1 Particle Class

This class represent a particle, with a random position and zero speed. When you create a particle you need to set 4 parameters:

- Number of variables (int). Set the number of variables that defines the position of a particle.
- Lower limit (list). Set the lower limits of a variable. You need to set a limit for every variable
- Upper limit (list). Set the upper limits of a variable. You need to set a limit for every variable
- Verbose (bool). Shows the information about the particle

The complete code for the particle class is shown in the code B.1

```
################################################################################
#                              CLASE PARTÍCULA                                 #
################################################################################

class Particula:
    """
    Esta clase representa nueva partícula con una posición inicial definida por
    una combinación de valores numéricos aleatorios y velocidad de 0. El rango
    de posibles valores para cada variable (posición) puede estar acotado. Al
    crear una nueva partícula, solo se dispone de información sobre su posición
    inicial y velocidad, el resto de atributos están vacíos.

    Parameters
    ----------
    n_variables : `int`
        número de variables que definen la posición de la partícula.

    limites_inf : `list` or `numpy.ndarray`, optional
        límite inferior de cada variable. Si solo se quiere predefinir límites
        de alguna variable, emplear ``None``. Los ``None`` serán remplazados
        por el valor (-10**3). (default is ``None``)

    limites_sup : `list` or `numpy.ndarray`, optional
        límite superior de cada variable. Si solo se quiere predefinir límites
        de alguna variable, emplear ``None``. Los ``None`` serán remplazados
        por el valor (+10**3). (default is ``None``)

    verbose : `bool`, optional
        mostrar información de la partícula creada. (default is ``False``)

    Attributes
    ----------
    n_variables : `int`
        número de variables que definen la posición de la partícula.

    limites_inf : `list` or `numpy.ndarray`
        límite inferior de cada variable. Si solo se quiere predefinir límites
        de alguna variable, emplear ``None``. Los ``None`` serán remplazados por
        el valor (-10**3).

    limites_sup : `list` or `numpy.ndarray`
        límite superior de cada variable. Si solo se quiere predefinir límites
        de alguna variable, emplear ``None``. Los``None`` serán remplazados por
        el valor (+10**3).

    mejor_valor : `numpy.ndarray`
        mejor valor que ha tenido la partícula hasta el momento.

    mejor_posicion : `numpy.ndarray`
        posición en la que la partícula ha tenido el mejor valor hasta el momento.

    valor : `float`
        valor actual de la partícula. Resultado de evaluar la función objetivo
        con la posición actual.

    velocidad : `numpy.ndarray`
        array con la velocidad actual de la partícula.

    posicion : `numpy.ndarray`
        posición actual de la partícula.

    Raises
    ------
    raise Exception
        si `limites_inf` es distinto de None y su longitud no coincide con
        `n_variables`.

    raise Exception
        si `limites_sup` es distinto de None y su longitud no coincide con
        `n_variables`.

    Examples
    --------
    Ejemplo creación partícula.

    >>> part = Particula(
                    n_variables = 3,
                    limites_inf = [-1,2,0],
                    limites_sup = [4,10,20],
                    verbose     = True
                    )

    """

    def __init__(self, n_variables, limites_inf=None, limites_sup=None,
                 verbose=False):

        # Número de variables de la partícula
        self.n_variables = n_variables
        # Límite inferior de cada variable
        self.limites_inf = limites_inf
        # Límite superior de cada variable
        self.limites_sup = limites_sup
        # Posición de la partícula
        self.posicion = np.repeat(None, n_variables)
```

```python
        # Velocidad de la parícula
        self.velocidad = np.repeat(None, n_variables)
        # Valor de la partícula
        self.valor = np.repeat(None, 1)
        # Mejor valor que ha tenido la partícula hasta el momento
        self.mejor_valor = None
        # Mejor posición en la que ha estado la partícula hasta el momento
        self.mejor_posicion = None

        # CONVERSIONES DE TIPO INICIALES
        # ----------------------------------------------------------------------
        # Si limites_inf o limites_sup no son un array numpy, se convierten en
        # ello.
        if self.limites_inf is not None \
        and not isinstance(self.limites_inf,np.ndarray):
            self.limites_inf = np.array(self.limites_inf)

        if self.limites_sup is not None \
        and not isinstance(self.limites_sup,np.ndarray):
            self.limites_sup = np.array(self.limites_sup)

        # COMPROBACIONES INICIALES: EXCEPTIONS Y WARNINGS
        # ----------------------------------------------------------------------
        if self.limites_inf is not None \
        and len(self.limites_inf) != self.n_variables:
            raise Exception(
                "limites_inf debe tener un valor por cada variable. " +
                "Si para alguna variable no se quiere límite, emplear None. " +
                "Ejemplo: limites_inf = [10, None, 5]"
                )
        elif self.limites_sup is not None \
        and len(self.limites_sup) != self.n_variables:
            raise Exception(
                "limites_sup debe tener un valor por cada variable. " +
                "Si para alguna variable no se quiere límite, emplear None. " +
                "Ejemplo: limites_sup = [10, None, 5]"
                )
        elif (self.limites_inf is None) or (self.limites_sup is None):
            warnings.warn(
                "Es altamente recomendable indicar los límites dentro de los " +
                "cuales debe buscarse la solución de cada variable. " +
                "Por defecto se emplea [-10^3, 10^3]."
                )
        elif any(np.concatenate((self.limites_inf, self.limites_sup)) == None):
            warnings.warn(
                "Los límites empleados por defecto cuando no se han definido " +
                "son: [-10^3, 10^3]."
            )

        # COMPROBACIONES INICIALES: ACCIONES
        # ----------------------------------------------------------------------

        # Si no se especifica limites_inf, el valor mínimo que pueden tomar las
        # variables es -10^3.
        if self.limites_inf is None:
            self.limites_inf = np.repeat(-10**3, self.n_variables)

        # Si no se especifica limites_sup, el valor máximo que pueden tomar las
        # variables es 10^3.
        if self.limites_sup is None:
             self.limites_sup = np.repeat(+10**3, self.n_variables)

        # Si los límites no son nulos, se reemplazan aquellas posiciones None por
        # el valor por defecto -10^3 y 10^3.
        if self.limites_inf is not None:
            self.limites_inf[self.limites_inf == None] = -10**3

        if self.limites_sup is not None:
            self.limites_sup[self.limites_sup == None] = +10**3

        # BUCLE PARA ASIGNAR UN VALOR A CADA UNA DE LAS VARIABLES QUE DEFINEN LA
        # POSICIÓN
        # ----------------------------------------------------------------------
        for i in np.arange(self.n_variables):
        # Para cada posición, se genera un valor aleatorio dentro del rango
        # permitido para esa variable.
            self.posicion[i] = random.uniform(
                                    self.limites_inf[i],
                                    self.limites_sup[i]
                                )

        # LA VELOCIDAD INICIAL DE LA PARTÍCULA ES 0
        # ----------------------------------------------------------------------
        self.velocidad = np.repeat(0, self.n_variables)

        # INFORMACIÓN DEL PROCESO (VERBOSE)
        # ----------------------------------------------------------------------
        if verbose:
            print("Nueva partícula creada")
            print("----------------------")
            print("Posición: " + str(self.posicion))
            print("Límites inferiores de cada variable: " \
                  + str(self.limites_inf))
            print("Límites superiores de cada variable: " \
                  + str(self.limites_sup))
            print("Velocidad: " + str(self.velocidad))
            print("")

    def __repr__(self):
        """
        Información que se muestra cuando se imprime un objeto partícula.

        """

        texto = "Partícula" \
                + "\n" \
                + "---------" \
                + "\n" \
                + "Posición: " + str(self.posicion) \
                + "\n" \
                + "Velocidad: " + str(self.velocidad) \
                + "\n" \
                + "Mejor posicion: " + str(self.mejor_posicion) \
                + "\n" \
                + "Mejor valor: " + str(self.mejor_valor) \
                + "\n" \
                + "Límites inferiores de cada variable: " \
                + str(self.limites_inf) \
                + "\n" \
                + "Límites superiores de cada variable: " \
                + str(self.limites_sup) \
                + "\n"

        return(texto)

    def evaluar_particula(self, funcion_objetivo, optimizacion, verbose = False):
        """
        Este método evalúa una partícula calculando el valor que toma la función
        objetivo en la posición en la que se encuentra. Además, compara si la
        nueva posición es mejor que las anteriores. Modifica los atributos
        valor, mejor_valor y mejor_posicion de la partícula.

        Parameters
        ----------
        funcion_objetivo : `function`
            función que se quiere optimizar.

        optimizacion : {'maximizar', 'minimizar'}
            dependiendo de esto, el mejor valor histórico de la partícula será
            el mayor o el menor valor que ha tenido hasta el momento.

        verbose : `bool`, optional
            mostrar información del proceso por pantalla. (default is ``False``)

        Raises
        ------
        raise Exception
            si el argumento `optimizacion` es distinto de 'maximizar' o
            'minimizar'.

        Examples
        --------
        Ejemplo evaluar partícula con una función objetivo.

        >>> part = Particula(
                n_variables = 3,
                limites_inf = [-1,2,0],
                limites_sup = [4,10,20],
                verbose     = True
                )

        >>> def funcion_objetivo(x_0, x_1, x_2):
                f= x_0**2 + x_1**2 + x_2**2
                return(f)

        >>> part.evaluar_particula(
                funcion_objetivo = funcion_objetivo,
                optimizacion     = "maximizar",
                verbose          = True
                )

        """

        # COMPROBACIONES INICIALES: EXCEPTIONS Y WARNINGS
        # ----------------------------------------------------------------------
        if not optimizacion in ["maximizar", "minimizar"]:
            raise Exception(
                "El argumento optimizacion debe ser: 'maximizar' o 'minimizar'"
                )

        # EVALUACIÓN DE LA FUNCIÓN OBJETIVO EN LA POSICIÓN ACTUAL
        # ----------------------------------------------------------------------
        self.valor = funcion_objetivo(*self.posicion)

        # MEJOR VALOR Y POSICIÓN
        # ----------------------------------------------------------------------
        # Se compara el valor actual con el mejor valor histórico. La comparación
        # es distinta dependiendo de si se desea maximizar o minimizar.
        # Si no existe ningún valor histórico, se almacena el actual. Si ya
        # existe algún valor histórico se compara con el actual y, de ser mejor
        # este último, se sobrescribe.

        if self.mejor_valor is None:
            self.mejor_valor    = np.copy(self.valor)
            self.mejor_posicion = np.copy(self.posicion)
        else:
            if optimizacion == "minimizar":
                if self.valor < self.mejor_valor:
                    self.mejor_valor    = np.copy(self.valor)
                    self.mejor_posicion = np.copy(self.posicion)
            else:
                if self.valor > self.mejor_valor:
                    self.mejor_valor    = np.copy(self.valor)
                    self.mejor_posicion = np.copy(self.posicion)

        # INFORMACIÓN DEL PROCESO (VERBOSE)
        # ----------------------------------------------------------------------
        if verbose:
            print("La partícula ha sido evaluada")
            print("-----------------------------")
            print("Valor actual: " + str(self.valor))
            print("")

    def mover_particula(self, mejor_p_enjambre, inercia=0.8, peso_cognitivo=2,
                        peso_social=2, verbose=False):
        """
        Este método ejecuta el movimiento de una partícula, lo que implica
        actualizar su velocidad y posición. No se permite que la partícula
        salga de la zona de búsqueda acotada por los límites.

        Parameters
        ----------
        mejor_p_enjambre : `np.narray`
            mejor posición de todo el enjambre.

        inercia : `float`, optional
            coeficiente de inercia. (default is 0.8)

        peso_cognitivo : `float`, optional
            coeficiente cognitivo. (default is 2)

        peso_social : `float`, optional
            coeficiente social. (default is 2)

        verbose : `bool`, optional
            mostrar información del proceso por pantalla. (default is ``False``)

        Examples
        --------
        Ejemplo mover partícula.

        >>> part = Particula(
                n_variables = 3,
                limites_inf = [-1,2,0],
                limites_sup = [4,10,20],
                verbose     = True
                )

        >>> def funcion_objetivo(x_0, x_1, x_2):
                f= x_0**2 + x_1**2 + x_2**2
                return(f)

        >>> part.evaluar_particula(
                funcion_objetivo = funcion_objetivo,
                optimizacion     = "maximizar",
                verbose          = True
                )

        >>> part.mover_particula(
                mejor_p_enjambre = np.array([-1000,-1000,+1000]),
                inercia          = 0.8,
                peso_cognitivo   = 2,
                peso_social      = 2,
                verbose          = True
                )

        """

        # ACTUALIZACIÓN DE LA VELOCIDAD
        # ----------------------------------------------------------------------
        componente_velocidad = inercia * self.velocidad
        r1 = np.random.uniform(low=0.0, high=1.0, size = len(self.velocidad))
        r2 = np.random.uniform(low=0.0, high=1.0, size = len(self.velocidad))
        componente_cognitivo = peso_cognitivo * r1 * (self.mejor_posicion \
                                                      - self.posicion)
        componente_social = peso_social * r2 * (mejor_p_enjambre \
                                                - self.posicion)
        nueva_velocidad = componente_velocidad + componente_cognitivo \
                          + componente_social
        self.velocidad = np.copy(nueva_velocidad)

        # ACTUALIZACIÓN DE LA POSICIÓN
        # ----------------------------------------------------------------------
        self.posicion = self.posicion + self.velocidad

        # COMPROBAR LÍMITES
        # ----------------------------------------------------------------------
        # Se comprueba si algún valor de la nueva posición supera los límites
        # impuestos. En tal caso, se sobrescribe con el valor del límite
        # correspondiente y se reinicia a 0 la velocidad de la partícula en esa
        # componente.
        for i in np.arange(len(self.posicion)):
            if self.posicion[i] < self.limites_inf[i]:
                self.posicion[i] = self.limites_inf[i]
                self.velocidad[i] = 0

            if self.posicion[i] > self.limites_sup[i]:
                self.posicion[i] = self.limites_sup[i]
                self.velocidad[i] = 0

        # INFORMACIÓN DEL PROCESO (VERBOSE)
        # ----------------------------------------------------------------------
        if verbose:
            print("La partícula se ha desplazado")
            print("----------------------------")
            print("Nueva posición: " + str(self.posicion))
            print("")
```

Code B.1 – *Particle class complete code*

B.2 Swarm Class

This class represent a n particles swarm. When you create a particle you need to set 5 parameters:

- Number of particles (int). Set the number of particles that your swarm has.
- Number of variables (int). Set the number of variables that defines the position of a particle.
- Lower limit (list). Set the lower limits of a variable. You need to set a limit for every variable
- Upper limit (list). Set the upper limits of a variable. You need to set a limit for every variable
- Verbose (bool). Shows the information about the particle

The complete code for the swarm class is shown in the code B.2

```
################################################################################
#                          CLASE ENJAMBRE (SWARM)                              #
################################################################################

class Enjambre:
    """
    Esta clase crea un enjambre de n partículas.

    Parameters
    ----------
    n_particulas :`int`
        número de partículas del enjambre.

    n_variables : `int`
        número de variables que definen la posición de las partícula.

    limites_inf : `list` or `numpy.ndarray`
        límite inferior de cada variable. Si solo se quiere predefinir límites
        de alguna variable, emplear ``None``. Los ``None`` serán remplazados por
        el valor (-10**3).

    limites_sup : `list` or `numpy.ndarray`
        límite superior de cada variable. Si solo se quiere predefinir límites
        de alguna variable, emplear ``None``. Los``None`` serán remplazados por
        el valor (+10**3).

    verbose : `bool`, optional
        mostrar información del proceso por pantalla. (default is ``False``)

    Attributes
    ----------
    partículas : `list`
        lista con todas las partículas del enjambre.

    n_particulas :`int`
        número de partículas del enjambre.

    n_variables : `int`
        número de variables que definen la posición de las partícula.

    limites_inf : `list` or `numpy.ndarray`
        límite inferior de cada variable.

    limites_sup : `list` or `numpy.ndarray`
        límite superior de cada variable.

    mejor_particula : `object particula`
        la mejor partícula del enjambre en estado actual.

    mejor_valor : `floar`
        el mejor valor del enjambre en su estado actual.

    historico_particulas : `list`
        lista con el estado de las partículas en cada una de las iteraciones que
        ha tenido el enjambre.

    historico_mejor_posicion : `list`
        lista con la mejor posición en cada una de las iteraciones que ha tenido
        el enjambre.

    historico_mejor_valor : `list`
        lista con el mejor valor en cada una de las iteraciones que ha tenido el
        enjambre.

    diferencia_abs : `list`
        diferencia absoluta entre el mejor valor de iteraciones consecutivas.

    resultados_df : `pandas.core.frame.DataFrame`
        dataframe con la información del mejor valor y posición encontrado en
        cada iteración, así como la mejora respecto a la iteración anterior.

    valor_optimo : `float`
        mejor valor encontrado en todas las iteraciones.

    posicion_optima : `numpy.narray`
        posición donde se ha encontrado el valor_optimo.

    optimizado : `bool`
        si el enjambre ha sido optimizado.

    iter_optimizacion : `int`
        número de iteraciones de optimizacion.

    Examples
    --------
    Ejemplo crear enjambre

    >>> enjambre = Enjambre(
                n_particulas = 5,
                n_variables  = 3,
                limites_inf  = [-5,-5,-5],
                limites_sup  = [5,5,5],
                verbose      = True
            )

    """

    def __init__(self, n_particulas, n_variables, limites_inf = None,
                 limites_sup = None, verbose = False):

        # Número de partículas del enjambre
        self.n_particulas = n_particulas
        # Número de variables de cada partícula
        self.n_variables = n_variables
        # Límite inferior de cada variable
        self.limites_inf = limites_inf
        # Límite superior de cada variable
        self.limites_sup = limites_sup
        # Lista de las partículas del enjambre
        self.particulas = []
        # Etiqueta para saber si el enjambre ha sido optimizado
        self.optimizado = False
        # Número de iteraciones de optimización llevadas a cabo
        self.iter_optimizacion = None
        # Mejor partícula del enjambre
        self.mejor_particula = None
        # Mejor valor del enjambre
        self.mejor_valor = None
        # Posición del mejor valor del enjambre.
        self.mejor_posicion = None
        # Estado de todas las partículas del enjambre en cada iteración.
        self.historico_particulas = []
        # Mejor posición en cada iteración.
        self.historico_mejor_posicion = []
        # Mejor valor en cada iteración.
        self.historico_mejor_valor = []
        # Diferencia absoluta entre el mejor valor de iteraciones consecutivas.
        self.diferencia_abs = []
        # data.frame con la información del mejor valor y posición encontrado en
        # cada iteración, así como la mejora respecto a la iteración anterior.
        self.resultados_df = None
        # Mejor valor de todas las iteraciones
        self.valor_optimo = None
        # Mejor posición de todas las iteraciones
        self.posicion_optima = None

        # CONVERSIONES DE TIPO INICIALES
        # ----------------------------------------------------------------------
        # Si limites_inf o limites_sup no son un array numpy, se convierten en
        # ello.
        if self.limites_inf is not None \
        and not isinstance(self.limites_inf,np.ndarray):
            self.limites_inf = np.array(self.limites_inf)

        if self.limites_sup is not None \
        and not isinstance(self.limites_sup,np.ndarray):
            self.limites_sup = np.array(self.limites_sup)

        # SE CREAN LAS PARTÍCULAS DEL ENJAMBRE Y SE ALMACENAN
        # ----------------------------------------------------------------------
        for i in np.arange(n_particulas):
            particula_i = Particula(
                                n_variables = self.n_variables,
                                limites_inf = self.limites_inf,
                                limites_sup = self.limites_sup,
                                verbose     = verbose
                              )
            self.particulas.append(particula_i)

        # INFORMACIÓN DEL PROCESO (VERBOSE)
        # ----------------------------------------------------------------------
        if verbose:
            print("---------------")
            print("Enjambre creado")
            print("---------------")
            print("Número de partículas: " + str(self.n_particulas))
            print("Límites inferiores de cada variable: "
                  + str(self.limites_inf))
            print("Límites superiores de cada variable: " \
                  + str(self.limites_sup))
            print("")

    def __repr__(self):
        """
        Información que se muestra cuando se imprime un objeto enjambre.

        """

        texto = "============================" \
                + "\n" \
                + "         Enjambre" \
                + "\n" \
                + "============================" \
                + "\n" \
                + "Número de partículas: " + str(self.n_particulas) \
                + "\n" \
                + "Límites inferiores de cada variable: " + str(self.limites_inf) \
                + "\n" \
                + "Límites superiores de cada variable: " + str(self.limites_sup) \
                + "\n" \
                + "Optimizado: " + str(self.optimizado) \
                + "\n" \
                + "Iteraciones optimización: " + str(self.iter_optimizacion) \
                + "\n" \
                + "\n" \
                + "Información mejor partícula:" \
                + "\n" \
                + "----------------------------" \
                + "\n" \
                + "Mejor posición actual: " + str(self.mejor_posicion) \
                + "\n" \
                + "Mejor valor actual: " + str(self.mejor_valor) \
                + "\n" \
                + "\n" \
                + "Resultados tras optimizar:" \
                + "\n" \
                + "----------------------------" \
                + "\n" \
                + "Posición óptima: " + str(self.posicion_optima) \
                + "\n" \
                + "Valor óptimo: " + str(self.valor_optimo)

        return(texto)

    def mostrar_particulas(self, n=None):
        """
        Este método muestra la información de cada una de las n primeras
        partículas del enjambre.

        Parameters
        ----------

        n : `int`
            número de particulas que se muestran. Si no se indica el valor
            (por defecto ``None``), se muestran todas. Si el valor es mayor
            que `self.n_particulas` se muestran todas.

        Examples
        --------
        >>> enjambre = Enjambre(
               n_particulas = 5,
               n_variables  = 3,
               limites_inf  = [-5,-5,-5],
               limites_sup  = [5,5,5],
               verbose      = True
            )

        >>> enjambre.mostrar_particulas(n = 1)

        """

        if n is None:
            n = self.n_particulas
        elif n > self.n_particulas:
            n = self.n_particulas

        for i in np.arange(n):
            print(self.particulas[i])
        return(None)

    def evaluar_enjambre(self, funcion_objetivo, optimizacion, verbose = False):
        """
        Este método evalúa todas las partículas del enjambre, actualiza sus
        valores e identifica la mejor partícula.

        Parameters
        ----------
        funcion_objetivo : `function`
            función que se quiere optimizar.

        optimizacion : {maximizar o minimizar}
            Dependiendo de esto, el mejor valor histórico de la partícula será
            el mayor o el menor valorque ha tenido hasta el momento.

        verbose : `bool`, optional
            mostrar información del proceso por pantalla. (default is ``False``)

        Examples
        --------
        Ejemplo evaluar enjambre

        >>> enjambre = Enjambre(
               n_particulas = 5,
               n_variables  = 3,
               limites_inf  = [-5,-5,-5],
                limites_sup   = [5,5,5],
                verbose       = True
            )

        >>> def funcion_objetivo(x_0, x_1, x_2):
                f= x_0**2 + x_1**2 + x_2**2
                return(f)

        >>> enjambre.evaluar_enjambre(
                funcion_objetivo = funcion_objetivo,
                optimizacion     = "minimizar",
                verbose          = True
                )

        """
        # SE EVALÚA CADA PARTÍCULA DEL ENJAMBRE
        # ----------------------------------------------------------------------
        for i in np.arange(self.n_particulas):
            self.particulas[i].evaluar_particula(
                funcion_objetivo = funcion_objetivo,
                optimizacion     = optimizacion,
                verbose          = verbose
                )

        # MEJOR PARTÍCULA DEL ENJAMBRE
        # ----------------------------------------------------------------------
        # Se identifica la mejor partícula de todo el enjambre. Si se está
        # maximizando, la mejor partícula es aquella con mayor valor.
        # Lo contrario si se está minimizando.

        # Se selecciona inicialmente como mejor partícula la primera.
        self.mejor_particula =  copy.deepcopy(self.particulas[0])
        # Se comparan todas las partículas del enjambre.
        for i in np.arange(self.n_particulas):
            if optimizacion == "minimizar":
                if self.particulas[i].valor < self.mejor_particula.valor:
                    self.mejor_particula = copy.deepcopy(self.particulas[i])
            else:
                if self.particulas[i].valor > self.mejor_particula.valor:
                    self.mejor_particula = copy.deepcopy(self.particulas[i])

        # Se extrae la posición y valor de la mejor partícula y se almacenan
        # como mejor valor y posición del enjambre.
        self.mejor_valor    = self.mejor_particula.valor
        self.mejor_posicion = self.mejor_particula.posicion

        # INFORMACIÓN DEL PROCESO (VERBOSE)
        # ----------------------------------------------------------------------
        if verbose:
            print("-----------------")
            print("Enjambre evaluado")
            print("-----------------")
            print("Mejor posición encontrada : " + str(self.mejor_posicion))
            print("Mejor valor encontrado : " + str(self.mejor_valor))
            print("")

    def mover_enjambre(self, inercia, peso_cognitivo, peso_social,
                       verbose = False):
        """
        Este método mueve todas las partículas del enjambre.

        Parameters
        ----------
        optimizacion : {'maximizar', 'minimizar'}
            si se desea maximizar o minimizar la función.

        inercia : `float` or `int`
            coeficiente de inercia.

        peso_cognitivo : `float` or `int`
            coeficiente cognitivo.

        peso_social : `float` or `int`
            coeficiente social.

        verbose : `bool`, optional
            mostrar información del proceso por pantalla. (default is ``False``)

        """

        # Se actualiza la posición de cada una de las partículas que forman el
        # enjambre.
        for i in np.arange(self.n_particulas):
            self.particulas[i].mover_particula(
                mejor_p_enjambre = self.mejor_posicion,
                inercia          = inercia,
                peso_cognitivo   = peso_cognitivo,
                peso_social      = peso_social,
                verbose          = verbose
            )

        # Información del proceso (VERBOSE)
        # ----------------------------------------------------------------------
        if verbose:
            print("----------------------------------------------------------" \
                  "------------")
            print("La posición de todas las partículas del enjambre ha sido " \
                  "actualizada.")
            print("----------------------------------------------------------" \
            "------------")
            print("")


    def optimizar(self, funcion_objetivo, optimizacion, n_iteraciones = 50,
                  inercia = 0.8, reduc_inercia = True, inercia_max = 0.9,
                  inercia_min = 0.4, peso_cognitivo = 2, peso_social = 2,
                  parada_temprana = False, rondas_parada = None,
                  tolerancia_parada  = None, verbose = False):
        """
        Este método realiza el proceso de optimización de un enjambre.

        Parameters
        ----------
        funcion_objetivo : `function`
            función que se quiere optimizar.

        optimizacion : {'maximizar' o 'minimizar'}
            si se desea maximizar o minimizar la función.

        m_iteraciones : `int` , optional
            numero de iteraciones de optimización. (default is ``50``)

        inercia : `float` or `int`, optional
            coeficiente de inercia. (default is ``0.8``)

        peso_cognitivo : `float` or `int`, optional
            coeficiente cognitivo. (default is ``2``)

        peso_social : `float` or `int`, optional
            coeficiente social. (default is ``2``)

        reduc_inercia: `bool`, optional
           activar la reducción del coeficiente de inercia. En tal caso, el
           argumento `inercia` es ignorado. (default is ``True``)

        inercia_max : `float` or `int`, optional
            valor inicial del coeficiente de inercia si se activa `reduc_inercia`.
            (default is ``0.9``)

        inercia_min : `float` or `int`, optional
            valor minimo del coeficiente de inercia si se activa `reduc_min`.
            (default is ``0.4``)

        parada_temprana : `bool`, optional
            si durante las últimas `rondas_parada` iteraciones la diferencia
            absoluta entre mejores partículas no es superior al valor de
            `tolerancia_parada`, se detiene el algoritmo y no se crean nuevas
            iteraciones. (default is ``False``)

        rondas_parada : `int`, optional
            número de iteraciones consecutivas sin mejora mínima para que se
            active la parada temprana. (default is ``None``)

        tolerancia_parada : `float` or `int`, optional
            valor mínimo que debe tener la diferencia de iteraciones consecutivas
            para considerar que hay cambio. (default is ``None``)

         verbose : `bool`, optional
            mostrar información del proceso por pantalla. (default is ``False``)

        Raises
        ------
        raise Exception
            si se indica `parada_temprana = True` y los argumentos `rondas_parada`
            o `tolerancia_parada` son ``None``.

        raise Exception
            si se indica `reduc_inercia = True` y los argumentos `inercia_max`
            o `inercia_min` son ``None``.

        Examples
        --------
        Ejemplo optimización

        >>> def funcion_objetivo(x_0, x_1):
                # Para la región acotada entre  10 <=x_0<=0 y  6 .5<=x_1<=0 la
                # funcion tiene multiples minimos locales y un unico minimo
                # global en f( 3 .1302468, 1 .5821422)=  106 .7645367.
                f = np.sin(x_1)*np.exp(1-np.cos(x_0))**2 \
                    + np.cos(x_0)*np.exp(1-np.sin(x_1))**2 \
                    + (x_0-x_1)**2
                return(f)

        >>> enjambre = Enjambre(
                        n_particulas = 50,
                        n_variables  = 2,
                        limites_inf  = [-10, -6.5],
                        limites_sup  = [0, 0],
                        verbose      = False
                        )

        >>> enjambre.optimizar(
                funcion_objetivo  = funcion_objetivo,
                optimizacion      = "minimizar",
                n_iteraciones     = 250,
                inercia           = 0.8,
                reduc_inercia     = True,
                inercia_max       = 0.9,
                inercia_min       = 0.4,
                peso_cognitivo    = 1,
                peso_social       = 2,
                parada_temprana   = True,
                rondas_parada     = 5,
                tolerancia_parada = 10**-3,
                verbose           = False
            )

        """

        # COMPROBACIONES INICIALES: EXCEPTIONS Y WARNINGS
        # ----------------------------------------------------------------------
        # Si se activa la parada temprana, hay que especificar los argumentos
        # rondas_parada y tolerancia_parada.
        if parada_temprana \
        and (rondas_parada is None or tolerancia_parada is None):
            raise Exception(
                "Para activar la parada temprana es necesario indicar un " \
                + " valor de rondas_parada y de tolerancia_parada."
                )

        # Si se activa la reducción de inercia, hay que especificar los argumentos
        # inercia_max y inercia_min.
        if reduc_inercia \
        and (inercia_max is None or inercia_min is None):
            raise Exception(
            "Para activar la reducción de inercia es necesario indicar un " \
            + "valor de inercia_max y de inercia_min."
            )

        # ITERACIONES
        # ----------------------------------------------------------------------
        start = time.time()

        for i in np.arange(n_iteraciones):
            if verbose:
                print("-------------")
                print("Iteracion: " + str(i))
                print("-------------")

            # EVALUAR PARTÍCULAS DEL ENJAMBRE
            # ------------------------------------------------------------------
            self.evaluar_enjambre(
                funcion_objetivo = funcion_objetivo,
                optimizacion     = optimizacion,
                verbose          = verbose
                )

            # SE ALMACENA LA INFORMACIÓN DE LA ITERACIÓN EN LOS HISTÓRICOS
            # ------------------------------------------------------------------
            self.historico_particulas.append(copy.deepcopy(self.particulas))
            self.historico_mejor_posicion.append(copy.deepcopy(self.mejor_posicion))
            self.historico_mejor_valor.append(copy.deepcopy(self.mejor_valor))

            # SE CALCULA LA DIFERENCIA ABSOLUTA RESPECTO A LA ITERACIÓN ANTERIOR
            # ------------------------------------------------------------------
            # La diferencia solo puede calcularse a partir de la segunda
            # iteración.
            if i == 0:
                self.diferencia_abs.append(None)
            else:
                diferencia = abs(self.historico_mejor_valor[i] \
                                 - self.historico_mejor_valor[i-1])
                self.diferencia_abs.append(diferencia)

            # CRITERIO DE PARADA
            # ------------------------------------------------------------------
            # Si durante las últimas n iteraciones, la diferencia absoluta entre
```

```python
            # mejores partículas no es superior al valor de tolerancia_parada,
            # se detiene el algoritmo y no se crean nuevas iteraciones.
            if parada_temprana and i > rondas_parada:
                ultimos_n = np.array(self.diferencia_abs[-(rondas_parada): ])
                if all(ultimos_n < tolerancia_parada):
                    print("Algoritmo detenido en la iteracion "
                          + str(i) \
                          + " por falta cambio absoluto mínimo de " \
                          + str(tolerancia_parada) \
                          + " durante " \
                          + str(rondas_parada) \
                          + " iteraciones consecutivas.")
                    break

            # MOVER PARTÍCULAS DEL ENJAMBRE
            # ------------------------------------------------------------------
            # Si se ha activado la reducción de inercia, se recalcula su valor
            # para la iteración actual.
            if reduc_inercia:
                inercia = ((inercia_max - inercia_min) \
                          * (n_iteraciones-i)/n_iteraciones) \
                          + inercia_min

            self.mover_enjambre(
               inercia         = inercia,
               peso_cognitivo = peso_cognitivo,
               peso_social    = peso_social,
               verbose        = False
            )

        end = time.time()
        self.optimizado = True
        self.iter_optimizacion = i

        # IDENTIFICACIÓN DEL MEJOR PARTÍCULA DE TODO EL PROCESO
        # ----------------------------------------------------------------------
        if optimizacion == "minimizar":
            indice_valor_optimo=np.argmin(np.array(self.historico_mejor_valor))
        else:
            indice_valor_optimo=np.argmax(np.array(self.historico_mejor_valor))

        self.valor_optimo    = self.historico_mejor_valor[indice_valor_optimo]
        self.posicion_optima = self.historico_mejor_posicion[indice_valor_optimo]

        # CREACIÓN DE UN DATAFRAME CON LOS RESULTADOS
        # ----------------------------------------------------------------------
        self.resultados_df = pd.DataFrame(
            {
            "mejor_valor_enjambre"   : self.historico_mejor_valor,
            "mejor_posicion_enjambre": self.historico_mejor_posicion,
            "diferencia_abs"         : self.diferencia_abs
            }
        )
        self.resultados_df["iteracion"] = self.resultados_df.index

        print("-------------------------------------------")
        print("Optimización finalizada " \
              + datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
        print("-------------------------------------------")
        print("Duración optimización: " + str(end - start))
        print("Número de iteraciones: " + str(self.iter_optimizacion))
        print("Posición óptima: " + str(self.posicion_optima))
        print("Valor óptimo: " + str(self.valor_optimo))
        print("")
```

Code B.2 – *Swarm class complete code*

Appendix C

Project Budget

C.1 Hardware and Software

As far as hardware and software are concerned, a computer is needed for this project, the used one costs around **450 €**, and a license of the Ansoft Mawell simulator, for students you can get a free version [13]. In Hardware and Software is necesary to spend **450 €**

C.2 Emplacement

The project lasted 8 months with a monthly rent of **200 €** and an approximate monthly electricity cost of **25 €**. The emplacement total cost is **1800 €**

C.3 Human Resources Cost

The development of this Bachelor's Thesis has required two people. The first one is a junior physicist (10 €/h), as a full-time worker during eight months. Secondly, as Project Supervisor a senior engineer (50 €/h), computing 5 hours per week. Then, Human Resources amounts to **20800 €**, as detailed in table C.1.

Post	Weekly Hours	Total Hours	Cost (€)
Junior Physicist	40	1280	12800
Senior Engineer	5	160	8000
		TOTAL	**20800 €**

Table C.1 – *Human Resources Cost*

References

[1] Olmedo, B. G. (2006). Campo eléctrico y campo magnético. *Fundamentos de Electromagnetismo* (pp. 5-26). Granada: Dpto. de Electromagnetismo y Física de la Materia. Universidad de Granada.

[2] Olmedo, B. G. (2006). Campos Multipolares estáticos. *Fundamentos de Electromagnetismo* (pp. 121-141). Granada: Dpto. de Electromagnetismo y Física de la Materia. Universidad de Granada.

[3] González, F. J. P. Derivadas. *Cálculo diferencial e integral de funciones de una variable*. (pp. 200-322). Granada: Dpto. de Análisis Matemático. Universidad de Granada. Recover from: `http://www.ugr.es/~fjperez/textos/calculo_diferencial_integral_func_una_var.pdf`

[4] K. Yamazaki and T. Kawamoto, *Simple estimation of equivalent magnetic dipole moment to characterize ELF magnetic fields generated by electric appliances incorporating harmonics*, in IEEE Transactions on Electromagnetic Compatibility, vol. 43, no. 2, pp. 240-245, May 2001, doi: 10.1109/15.925547

[5] *Ansys Maxwell*. (2019). Windows. Pittsburgh: Ansoft Corporation.

[6] Matthes, E. (2015). *Python crash course: a hands-on, project-based introduction to programming*. No Starch Press.

[7] Ramalho, L. (2015). *Fluent python: Clear, concise, and effective programming*. " O'Reilly Media, Inc.".

[8] Jupyter. *Jupyter*. Recover from: `https://jupyter.org/`

[9] Plotly Python Open Source Graphing Library. *Plotly*. Recover from: `https://plotly.com/python/`

[10] E. Carrubba, A. Junge, F. Marliani and A. Monorchio, *Particle Swarm Optimization for Multiple Dipole Modeling of Space Equipment*, in IEEE Transactions on Magnetics, vol. 50, no. 12, pp. 1-10, Dec. 2014, Art no. 7028010, doi: 10.1109/TMAG.2014.2334277.

[11] Ansoft Corporation. (2006). *Ansoft Maxwell 3D Field Simulator v11 User's Guide*. Pittsburgh, USA: Ansoft Corporation. Recover from: `http://ansoft-maxwell.narod.ru/en/CompleteMaxwell3D_V11.pdf`

[12] Joaquin Amat Rodrigo. (2019). *Optimización con enjambre de partículas (Particle Swarm Optimization)*. Recover from: `https://github.com/JoaquinAmatRodrigo/optimizacion_PSO_python/blob/master/PSO_python.ipynb`

[13] Ansoft Corporation. (2006). *Ansys Free Student Software Downloads*. Pittsburgh, USA: Ansoft Corporation. Recover from: `https://www.ansys.com/academic/free-student-products`