

Suplemento Computacional **Electricidad y Magnetismo**

Sebastian Bustamante Jaramillo

macsebas33@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Antioquia

Índice general

1. Preliminares	5
1.1. Motivación	5
1.2. Instalación de Paquetes	5
1.3. Ejemplo de Uso	8
1.4. Consejos de Programación	10
2. Electrostática	13
2.1. Demostración 1: Espectrómetro de Masas	13
2.2. Demostración 2: Líneas de Campo y Equipotenciales	18
2.3. Demostración 3: Billar Electrostático	24
2.4. Ejercicios	42
3. Magnetostática	45
3.1. Ejercicios	45
4. Electrodinámica	49
4.1. Demostración 1: Bobina fonocaptora	49
4.2. Ejercicios	56

Capítulo 1

Preliminares

1.1. Motivación

La física ha evolucionado hasta un estado actual donde la mayoría de cálculos teóricos necesarios para realizar investigación de frontera requieren de una gran componente computacional. Desde la corroboración entre teoría y experimento, la predicción y control de los resultados de un experimento hecho a posteriori y la recreación de condiciones imposibles de lograr experimentalmente, tales como simulaciones cosmológicas del universo a gran escala o complejos sistemas atómicos. Estos son sólo algunos ejemplos representativos del papel de la computación en la física moderna. Debido a esto, el principal objetivo del suplemento computacional es la introducción temprana en los cursos de física básica de herramientas computacionales que serán de utilidad a los estudiantes en este curso específico y durante el transcurso de sus carreras científicas.

1.2. Instalación de Paquetes

En la totalidad de esta guía será usado el lenguaje de programación *Python* como referente para todos las prácticas y ejercicios computacionales. La principal motivación de esto es su facilidad de implementación en comparación a otros lenguajes también de amplio uso en ciencia. Además es un lenguaje interpretado, lo que permite una depuración más sencilla por parte del estudiante, sin necesidad de usar más complicados sistemas de depuración en el caso de lenguajes compilados como C o Fortran. *Python* es un lenguaje de código abierto, lo que permite la libre distribución del paquete y evita el pago de costosas licencias de uso, además la gran mayoría de paquetes que extienden enormemente la funcionalidad de *Python* son también código abierto y de libre distribución y uso.

A pesar de que *Python* es un lenguaje multiplataforma, permitiendo correr scripts python en Linux, Windows y Mac, acá solo se indicará el método de instalación para distribuciones Linux basadas en Debian.

La última versión de *Python* de la rama 2 es 2.7.4 y de la rama 3 es la 3.3.1, debido a ligeras incompatibilidades entre ambas ramas de desarrollo, será utilizada la rama 2 en una de sus últimas versiones. En orden, para instalar *Python* en una versión Linux basta con descargarlo directamente de los repositorios oficiales¹, en el caso de una distro basada en Debian el gestor de paquetes es `apt-get`, y desde una terminal se tiene

```
\$ apt-get install python2.7
```

también puede descargarse directamente desde la página oficial del proyecto <http://python.org/>.

Una vez instalada la última versión de *Python*, es necesario instalar los siguiente paquetes para el correcto desarrollo de las aplicaciones del curso:

iPython

iPython es un shell que permite una interacción más interactiva con los scripts de python, permitiendo el resaltado de sintaxis desde consola, funciones de autocompletado y depuración de código más simple. Para su instalación basta descargarlo de los repositorios oficiales

```
\$ apt-get install ipython
```

o puede de descargarse de la página oficial <http://ipython.org/>. También puede encontrarse documentación completa y actualizada en esta página, se recomienda visitarla frecuentemente para tener las más recientes actualizaciones.

NumPy

NumPy es una librería que extiende las funciones matemáticas de *Python*, permitiendo el manejo de matrices y vectores. Es esencial para la programación científica en *Python* y puede ser instalada de los repositorios

```
\$ apt-get install python-numpy
```

¹En la mayoría de distribuciones Linux *Python* viene precargado por defecto.

La última versión estable es la 1.6.2. En la página oficial del proyecto puede encontrarse versiones actualizadas y una amplia documentación <http://www.numpy.org/>.

SciPy

SciPy es una amplia biblioteca de algoritmos matemáticos para *Python*, esta incluye herramientas que van desde funciones especiales, integración, optimización, procesamiento de señales, análisis de Fourier, etc. Al igual que los anteriores paquetes, puede ser instalada desde los repositorios oficiales

```
\$ apt-get install python-scipy
```

Una completa documentación del paquete puede ser encontrada en <http://docs.scipy.org/doc/scipy/reference/>. La última versión estable es la 0.11.0 y puede ser encontrada en la página oficial del proyecto <http://www.scipy.org/>.

Matplotlib

Matplotlib es una completa librería con rutinas para la generación de gráficos a partir de datos. Aunque en su estado actual está enfocada principalmente a gráficos 2D, permite un amplio control sobre el formato de las gráficas generadas, dando una amplia versatilidad a los usuarios. Su instalación puede realizarse a partir de los repositorios oficiales

```
\$ apt-get install python-matplotlib
```

La última versión estable es la 1.2.1. y puede encontrarse en la página oficial del proyecto <http://matplotlib.org/>. Una amplia documentación está disponible en <http://matplotlib.org/1.2.0/contents.html>.

MayaVi2

MayaVi2 es una librería para la visualización científica en python, en especial para gráficos 3D, permitiendo funciones avanzadas como renderizado, manejo de texturas, etc. Se encuentra en los repositorios oficiales

```
\$ apt-get install mayavi2
```

La versión 2 es una versión mejorada de la original, estando más orientada a la reutilización de código. Por defecto incluye una interfaz gráfica que facilita su manejo. La página oficial del proyecto es <http://mayavi.sourceforge.net/>.

Tkinter

TKinter es una librería para la gestión gráfica de aplicaciones in *Python* y viene por defecto instalada, aún así puede ser instalada de los repositorios oficiales

```
\$ apt-get install python-tk
```

La página oficial del proyecto es <http://wiki.python.org/moin/TkInter>. Para el desarrollo de entornos gráficos existen otras llamativas alternativas como PyGTK o PyQt, pero debido a la facilidad de uso y a ser la librería estándar soportada, *TKinter* será usada en este curso.

1.3. Ejemplo de Uso

En esta sección se ilustra un ejemplo sencillo que permite al estudiante identificar la manera estándar de ejecutar códigos en *Python*, además probar los paquetes instalados.

El código de ejemplo permite graficar dos funciones diferentes en una misma ventana y además un conjunto de datos aleatorios generados en el eje Y.

```
1  #!/usr/bin/env python
2  #=====
3  # EJEMPLO DE USO
4  # Grafica de funciones y datos aleatorios
5  #=====
6  import numpy as np
7  import scipy as sp
8  import matplotlib.pyplot as plt
9
10 #Funcion 1
11 def Funcion1(x):
12     f1 = np.sin(x)/( np.sqrt(1 + x**2) )
13     return f1
14
15 #Funcion 2
16 def Funcion2(x):
```



```
17     f2 = 1/(1+x)
18     return f2
19
20 #Valores de x para evaluar
21 X = np.linspace( 0, 10, 100 )
22 #Evaluacion de funcion 1
23 F1 = Funcion1(X)
24 #Evaluacion de funcion 2
25 F2 = Funcion2(X)
26
27 #Grafica funcion 1
28 plt.plot( X, F1, label='Funcion 1' )
29 #Grafica funcion 2
30 plt.plot( X, F2, label='Funcion 2' )
31
32 #Datos aleatorios eje Y
33 Yrand = sp.random.rand( 100 )
34 #Grafica datos aleatorios
35 plt.plot( X, Yrand, 'o', label='Datos' )
36
37 plt.legend()
38 plt.show()
```

El resultado obtenido es la siguiente gráfica

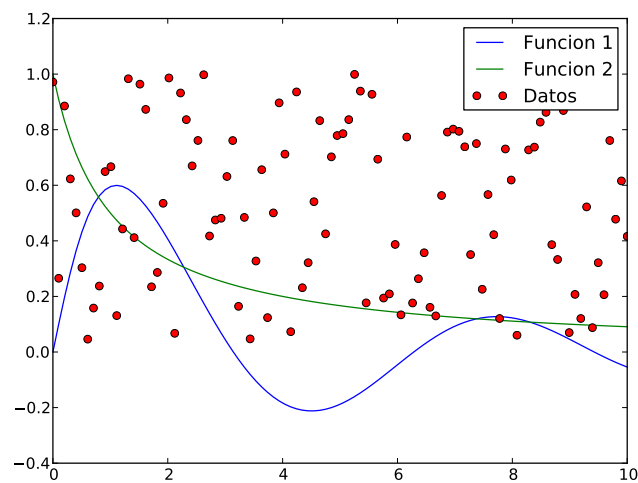


Figura 1.1: Resultado del ejemplo anterior, gráfica de dos funciones y datos aleatorios.

Para obtener el anterior script, el estudiante puede transcribirlo directamente de esta guía o puede descargarlo del repositorio oficial del curso² en el link https://github.com/sbustamante/Computacional-Campos/raw/master/codigos/usage_01.py. Una vez obtenido el archivo `usage_01.py`, abrir una terminal en la carpeta donde se ha guardado y escribir `ipython` para abrir el intérprete de *Python*

```
\$ ipython
```

Se debe obtener algo como

```
\$ ipython
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object'. ?object also works, ?? prints more.

In [1]:
```

Finalmente para ejecutar el script, usar el comando `run` seguido del nombre del código en el intérprete de código *Python*

```
In [1]: run usage_01.py
```

1.4. Consejos de Programación

En esta sección se dan algunos consejos útiles a la hora de programar. Estas buenas prácticas ayudan al programador a ser estructurado y tener un método coherente de desarrollar códigos.

- Asignar nombres a los archivos que sean coherentes con lo que realizan. Por ejemplo si está realizando una rutina que integra la trayectoria de una partícula en un campo eléctrico E , un nombre adecuado para el código puede ser `trayectoria_electric_field.py`. Evite usar nombres poco intuitivos y que tengan caracteres extraños, inclusive caracteres tildados como *í*, *ó*, *ñ*, etc. También evite el uso de espacios, si desea separar palabras, use el caracter `_`.

²Repositorio oficial en <https://github.com/sbustamante/Computacional-Campos>

- Una práctica casi obligatoria que debe realizar cualquier programador serio, consiste en comentar su código exhaustivamente. Los lenguajes de programación fueron inventados para facilitar la interacción entre el humano y la computadores, los compiladores e intérpretes de códigos *traducen* este lenguaje a lenguaje binario de máquina. Es por esta razón que cuando se crea un código, se debe pensar en que este sea comprensible, siempre recuerde que la máquina solo requiere código binario.

Comente todo lo que usted crea conveniente. Es muy común que por minimizar líneas de código o simplemente por pensar que no es necesario, muchos programadores dejan de comentar su código y semanas más tarde retoman y no entienden que han hecho. La programación en equipo es cada vez más común en ciencia, existiendo grandes proyectos donde una comunidad activa está contribuyendo, comentar el código es indispensable para este tipo de actividades.

- Use el inglés siempre que pueda, para asignar nombres a sus archivos, para comentar su código, y para datos impresos en pantalla y en archivos de datos. Tenga en cuenta que en disciplinas científicas cualquier labor que sea realizada en inglés tendrá un potencial de impacto mayor que otro si se hace en otro idioma. Su código puede ser útil después a otras personas y esto puede ayudar a que usted sea reconocido en círculos académicos.
- Comparta su código, para esto puede usar páginas diseñadas para esto tales como `github.com` o `sourceforge.net`. Estas páginas además de permitir compartir su código a terceras personas, tiene potentes herramientas de control de versiones (git, svn, etc.) las cuales le ayudan a manejar su código de forma más estructurada, permitiendo el control de las diferentes versiones de un código o un paquete completo y el desarrollo entre varias personas.
- Use software open source, este tipo de software es de libre distribución y uso. Cuando se usan herramientas que necesitan licencia se está sujeto al pago de ingentes cantidades de dinero por funcionalidades que se pueden encontrar de forma gratuita. Este tipo de herramientas pagas limitan aspectos como el poder compartir códigos y resultados, por ejemplo si usted no tiene su licencia al día puede tener problemas a la hora de reportar en un artículo de investigación sus resultados y gráficas obtenidos con estos paquetes.

Capítulo 2

Electrostática

La electrostática estudia la interacción entre cuerpos cargados eléctricamente sin tener en cuenta su movimiento. Esta simplificación permite ignorar términos dinámicos asociados a corrientes eléctricas y campos magnéticos inducidos.

En este capítulo se realizan algunas demostraciones computacionales que van desde el cálculo de trayectorias de partículas en campos eléctricos y magnéticos, la representación de las líneas de campo y superficies equipotenciales de distribuciones de carga, hasta el cálculo de capacitancia de algunos sistemas.

2.1. Demostración 1: Espectrómetro de Masas

En esta primera demostración será estudiado el espectrómetro de masas. El objetivo de este dispositivo es caracterizar partículas cargadas de acuerdo a su relación carga masa (q/m). Su funcionamiento consiste en la inmersión de las partículas en un campo magnético o eléctrico (este caso) y a partir de las trayectorias obtenidas determinar su relación carga masa (ver figura 2.1).

Tomando una partícula de masa m y carga q embebida en un campo eléctrico homogéneo y uniforme \mathbf{E} , la ecuación de movimiento es

$$m \frac{d^2 \mathbf{r}}{dt^2} = q \mathbf{E} \quad (2.1)$$

Tomando el sistema coordenado de tal forma que el campo \mathbf{E} esté en la dirección positiva de y , se obtiene

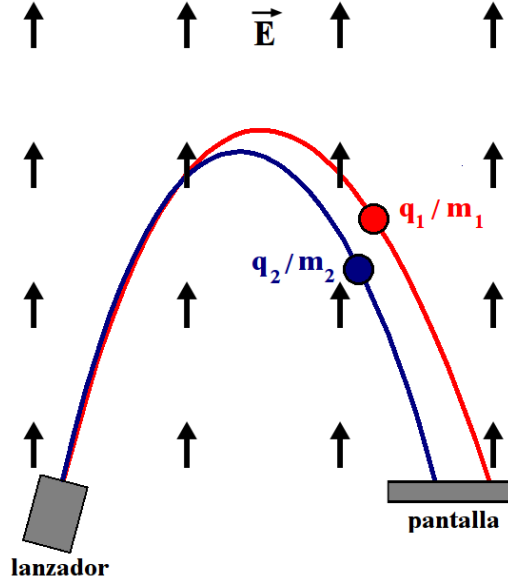


Figura 2.1: Espectrómetro de masas.

$$x(t) = x_0 + v_{x0}t \quad (2.2)$$

$$y(t) = y_0 + v_{y0}t + \frac{1}{2} \left(\frac{q}{m} \right) |\mathbf{E}| t^2 \quad (2.3)$$

donde se ha introducido la posición inicial de la partícula $\mathbf{r}(t=0) = (x_0, y_0)$ y la velocidad inicial $\mathbf{v}(t=0) = (v_{x0}, v_{y0})$.

Eliminando el tiempo de las dos ecuaciones se obtiene la siguiente trayectoria

$$y(x) = y_0 + \frac{v_{y0}}{v_{x0}}(x - x_0) + \frac{1}{2} \left(\frac{q}{m} \right) |\mathbf{E}| \left(\frac{x - x_0}{v_{x0}} \right)^2 \quad (2.4)$$

En el siguiente código de *Python* se grafica la trayectoria de dos partículas con diferente relación carga masa. La primera tiene una relación $q_1/m_1 = -1 \text{ C/1 kg}$ y la segunda $q_2/m_2 = -1 \text{ C/2 kg}$. Ambas partículas se disparan del origen y con una velocidad inicial de $\mathbf{v}_0 = (1, 2) \text{ m/s}$. El campo eléctrico tiene una intensidad de $|\mathbf{E}| = 1 \text{ N/C}$.

```
1  #!/usr/bin/env python
2  #!/usr/bin/env python
3  #=====
4  # DEMOSTRACION 1
5  # Espectrometro de masas
6  #=====
7  from __future__ import division
8  import numpy as np
9  import matplotlib.pyplot as plt
10
11  #Trayectoria
12  def trayectoria(x):
13      y = y0 + vy0/vx0*(x - x0) + 0.5*(q/m)*E*( (x-x0)/vx0 )
14          **2
15      return y
16
17  #Campo electrico
18  E = 1
19
20  #PARTICULA 1
21  #Carga
22  q = -1
23  #Masa
24  m = 1
25  #Posicion inicial
26  x0 = 0
27  y0 = 0
28  #Velocidad inicial
29  vx0 = 1
30  vy0 = 2
31  #Valores de X a graficar
32  X = np.arange( 0, 10, 0.1 )
33  #Trayectoria
34  Y = trayectoria( X )
35  #Grafica de trayectoria
36  plt.plot( X, Y, label='particula 1' )
37
38  #PARTICULA 2
39  #Carga
40  q = -1
41  #Masa
42  m = 2
```

```

42 #Posicion inicial
43 x0 = 0
44 y0 = 0
45 #Velocidad inicial
46 vx0 = 1
47 vy0 = 2
48 #Valores de X a graficar
49 X = np.arange( 0, 10, 0.1 )
50 #Trayectoria
51 Y = trayectoria( X )
52 #Grafica de trayectoria
53 plt.plot( X, Y, label='particula 2' )
54
55 #Límites del eje X
56 plt.xlim( (0,10) )
57 #Límites del eje Y
58 plt.ylim( (0,10) )
59 plt.legend()
60 plt.show()

```

El resultado que se obtiene es

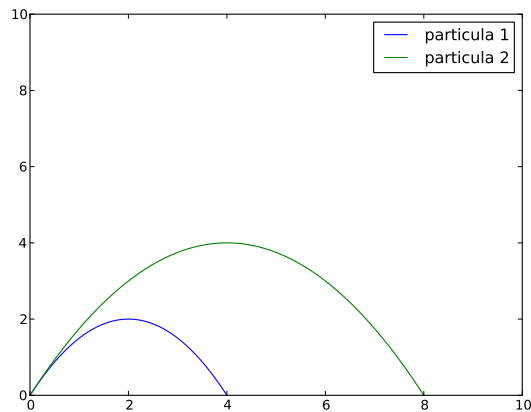


Figura 2.2: Trayectorias según la relación carga masa de las partículas.

La representación de la trayectoria de forma gráfica permite entonces comparar directamente con datos medidos en un laboratorio, por ejemplo trayectorias de partículas en cámaras de burbujas.

A continuación se explica cada componente del código anterior

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
```

En la primera línea se carga el módulo `division`, este permite a *Python* calcular fracciones de números enteros como cantidades reales. En la siguiente línea se carga la librería *NumPy* con el alias de `np` y finalmente se carga la librería *Matplotlib* con el alias de `plt`.

```
#Trayectoria
def trayectoria(x):
    y = y0 + vy0/vx0*(x - x0) + 0.5*(q/m)*E*( (x-x0)/vx0 )
        **2
    return y
```

En esta parte se define la trayectoria de la partícula en el campo eléctrico descrito.

```
#PARTICULA 1
#Carga
q = -1
#Masa
m = 1
#Posicion inicial
x0 = 0
y0 = 0
#Velocidad inicial
vx0 = 1
vy0 = 2
#Valores de X a graficar
X = np.arange( 0, 10, 0.01 )
#Trayectoria
Y = trayectoria( X )
#Grafica de trayectoria
plt.plot( X, Y, label='particula 1' )
```

Se definen las propiedades físicas y cinemáticas de la partícula 1. Su carga, su masa, su posición y velocidad inicial. Luego, usando el comando `arange` de la librería *NumPy*, se construye un arreglo de valores en el eje X que serán usados para el cálculo de la trayectoria. En este caso se toma desde 0 a 10 m con un salto de 0,01 m. Finalmente se llama la función de la trayectoria de la partícula en todos los

valores de X y se grafica, usando como etiqueta `label='particula 1'`.

```
#Límites del eje X
plt.xlim( (0,10) )
#Límites del eje Y
plt.ylim( (0,10) )
plt.legend()
plt.show()
```

Finalmente se usan las funciones de *Matplotlib* `xlim` y `ylim` para fijar los límites de la ventana de graficación. La función `legend` muestra las etiquetas de las dos trayectorias y finalmente `show` muestra en pantalla el resultado.

2.2. Demostración 2: Líneas de Campo y Equipotenciales

Un campo es una distribución espacial de alguna cantidad física, tal como el campo eléctrico y magnético. Este puede ser generado por una fuente, tal como es el caso del campo electrostático generado por una distribución de cargas, o pueden no tener una fuente asociada, como el caso de ondas electromagnéticas que viajan libre en el espacio.

En esta demostración se calculará las líneas de campo eléctrico y las líneas de equipotencial para una distribución de dos partículas puntuales. Debido al principio de superposición se tiene entonces

$$\varphi_{tot}(\mathbf{r}) = \varphi_1(\mathbf{r}) + \varphi_2(\mathbf{r}) \quad (2.5)$$

$$\begin{aligned} &= \frac{1}{4\pi\epsilon_0} \frac{q_1}{|\mathbf{r} - \mathbf{r}_1|} + \frac{1}{4\pi\epsilon_0} \frac{q_2}{|\mathbf{r} - \mathbf{r}_2|} \\ \mathbf{E}_{tot}(\mathbf{r}) &= \mathbf{E}_1(\mathbf{r}) + \mathbf{E}_2(\mathbf{r}) \quad (2.6) \\ &= \frac{1}{4\pi\epsilon_0} \frac{q_1}{|\mathbf{r} - \mathbf{r}_1|^3} (\mathbf{r} - \mathbf{r}_1) + \frac{1}{4\pi\epsilon_0} \frac{q_2}{|\mathbf{r} - \mathbf{r}_2|^3} (\mathbf{r} - \mathbf{r}_2) \end{aligned}$$

El script en *Python* para realizar la demostración es:

```
1 #!/usr/bin/env python
2 #=====
```

2.2. DEMOSTRACIÓN 2: LÍNEAS DE CAMPO Y EQUIPOTENCIALES 19

```
3 # DEMOSTRACION 2
4 # Lineas de campo y equipotenciales de cargas puntuales
5 #=====
6 from __future__ import division
7 import numpy as np
8 import matplotlib.pyplot as plt
9
10 #Potencial de una partícula puntual cargada
11 def Phi( r, rp, q ):
12     phi = 1/(4*np.pi*eps0)*q/np.linalg.norm( r - rp )
13     return phi
14
15 #Campo electrico de una partícula puntual cargada
16 def Electric( r, rp, q ):
17     E = 1/(4*np.pi*eps0)*q*(r - rp)/np.linalg.norm( r - rp )
18         **3
19     return E
20
21 #Permitividad del vacio
22 eps0 = 8.85418e-12
23 #Resolucion de graficas
24 Nres = 25
25 #Coordenada X
26 Xarray = np.linspace( 0, 10, Nres )
27 #Coordenada Y
28 Yarray = np.linspace( 0, 10, Nres )
29 #Construccion de la cuadrícula
30 X, Y = plt.meshgrid( Xarray, Yarray )
31
32 #CONSTRUCCION DE CAMPO E Y POTENCIAL PHI, PARTICULA 1
33 #Carga electrica
34 q1 = -1
35 #Posicion partícula
36 rp1 = np.array( [4,4] )
37 #Inicializacion Potencial Electrico
38 phil = np.zeros( (Nres,Nres) )
39 #Inicializacion Campo Electrico
40 Elx = np.ones( (Nres,Nres) )
41 Ely = np.ones( (Nres,Nres) )
42 #Calculo Potencial Electrico y Campo Electrico
43 for i in xrange(Nres):
44     for j in xrange(Nres):
45         r = np.array( [Xarray[i], Yarray[j]] )
```

```

45     phil[i,j] = Phi( r, rp1, q1 )
46     E = Electric( r, rp1, q1 )
47     Elx[i,j], Ely[i,j] = E/np.linalg.norm(E)
48
49 #CONSTRUCCION DE CAMPO E Y POTENCIAL PHI, PARTICULA 2
50 #Carga electrica
51 q2 = -1
52 #Posicion particula
53 rp2 = np.array( [6,6] )
54 #Incializacion Potencial Electrico
55 phi2 = np.zeros( (Nres,Nres) )
56 #Incializacion Campo Electrico
57 E2x = np.ones( (Nres,Nres) )
58 E2y = np.ones( (Nres,Nres) )
59 #Calculo Potencial Electrico y Campo Electrico
60 for i in xrange(Nres):
61     for j in xrange(Nres):
62         r = np.array( [Xarray[i], Yarray[j]] )
63         phi2[i,j] = Phi( r, rp2, q2 )
64         E = Electric( r, rp2, q2 )
65         E2x[i,j], E2y[i,j] = E/np.linalg.norm(E)
66
67 #CONSTRUCCION DE CAMPO E Y POTENCIAL PHI, TOTAL
68 phi_tot = phil + phi2
69 #Incializacion Campo Electrico
70 Ex_tot = np.ones( (Nres,Nres) )
71 Ey_tot = np.ones( (Nres,Nres) )
72 #Calculo Potencial Electrico y Campo Electrico Total
73 for i in xrange(Nres):
74     for j in xrange(Nres):
75         E = np.array( [Elx[i,j] + E2x[i,j], \
76             Ely[i,j] + E2y[i,j]] )
77         E = E/np.linalg.norm(E)
78         Ex_tot[i,j], Ey_tot[i,j] = E
79
80 #Grafica de equipotenciales
81 plt.contour(X, Y, phi_tot, 100)
82 #Grafica de lineas de campo
83 plt.quiver( X, Y, Ey_tot, Ex_tot)
84
85 #Limites del eje X
86 plt.xlim( (0,10) )
87 #Limites del eje Y

```

```

88 plt.ylim( (0,10) )
89 plt.legend()
90 plt.show()

```

El resultado obtenido es

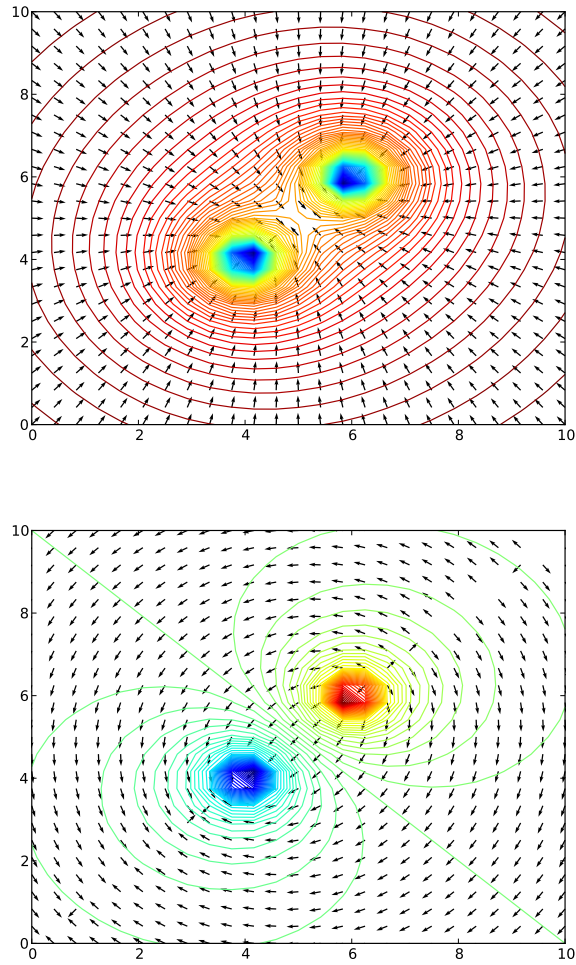


Figura 2.3: Líneas de campo y equipotenciales de dos cargas puntuales iguales. Con iguales signos (Superior) y con signos opuestos (Inferior).

A continuación se detalla cada parte del anterior script

```

#Potencial de una partícula puntual cargada
def Phi( r, rp, q ):

```

```

    phi = 1/(4*np.pi*eps0)*q/np.linalg.norm( r - rp )
    return phi

#Campo electrico de una partícula puntual cargada
def Electric( r, rp, q ):
    E = 1/(4*np.pi*eps0)*q*(r - rp)/np.linalg.norm( r - rp )
        **3
    return E

```

En estas líneas se definen la forma funcional del potencial y el campo eléctrico asociados a una partícula puntual. Los argumentos r y rp corresponden a los vectores donde se evalúan los campos y al vector posición de la partícula, respectivamente. Como tercer argumento se da la carga eléctrica q . En el computo del campo eléctrico se usa la función `norm` del paquete `linalg` de la librería *NumPy* para calcular la norma del vector $r - r_p$.

```

#Permitividad del vacio
eps0 = 8.85418e-12
#Resolucion de graficas
Nres = 25
#Coordenada X
Xarray = np.linspace( 0, 10, Nres )
#Coordenada Y
Yarray = np.linspace( 0, 10, Nres )
#Construccion de la cuadrícula
X, Y = plt.meshgrid( Xarray, Yarray )

```

Se define la permitividad del vacío en unidades SI y la resolución espacial de la malla donde serán evaluadas funciones. Finalmente usando la función `linspace` de la librería *NumPy* se construyen los arreglos asociados a cada eje espacial, de 0 a 10 metros con una resolución de `Nres` divisiones, y luego usando la función `meshgrid` de la librería *Matplotlib* se construyen las matrices de coordenadas, necesarias para la graficación de los campos.

```

#CONSTRUCCION DE CAMPO E Y POTENCIAL PHI, PARTICULA 1
#Carga electrica
q1 = -1
#Posicion partícula
rpl = np.array( [4,4] )
#Incializacion Potencial Electrico
phil = np.zeros( (Nres,Nres) )
#Incializacion Campo Electrico

```

```

Elx = np.ones( (Nres,Nres) )
Ely = np.ones( (Nres,Nres) )
#Calculo Potencial Electrico y Campo Electrico
for i in xrange(Nres):
    for j in xrange(Nres):
        r = np.array( [Xarray[i], Yarray[j]] )
        phi1[i,j] = Phi( r, rp1, q1 )
        E = Electric( r, rp1, q1 )
        Elx[i,j], Ely[i,j] = E/np.linalg.norm(E)

```

En esta parte se define y construye todas las propiedades físicas de la partícula 1. Inicialmente se define la carga q_1 y su posición rp_1 . Posteriormente se inicializan las matrices donde se van a mapear los valores de los campos, primero el potencial como $\text{phi1} = \text{np.zeros}((N_{res}, N_{res}))$, para esto se usa la función `zeros` de *NumPy* y se da como argumentos la dimensión $N_{res} \times N_{res}$ de la matriz. De igual forma, usando la función `ones` de *NumPy* se inicializan las matrices asociadas a la componente x y y del campo \mathbf{E} .

Usando los ciclos `for` de *Python* se hace un barrido de todas las matrices, tanto de las columnas como de las filas. Cada posición i, j está asociada a una coordenada $\mathbf{r}_{ij} = x_i \mathbf{i} + y_j \mathbf{j}$, en el código `r = np.array([Xarray[i], Yarray[j]])`. Finalmente se calcula el potencial en este punto del espacio \mathbf{r}_{ij} como `phi1[i,j] = Phi(r, rp1, q1)` y cada componente del campo eléctrico `Elx[i,j]` y `Ely[i,j]`. Es importante mencionar que lo único importante para definir las líneas de campo eléctrico es la dirección local de $\mathbf{E}(\mathbf{r}_{ij})$, por esta razón se realiza la normalización `E/np.linalg.norm(E)`, almacenando así el vector unitario que indica la dirección local del campo. Esto se repite para la carga 2.

```

#CONSTRUCCION DE CAMPO E Y POTENCIAL PHI, TOTAL
phi_tot = phi1 + phi2
#Incializacion Campo Electrico
Ex_tot = np.ones( (Nres,Nres) )
Ey_tot = np.ones( (Nres,Nres) )
#Calculo Potencial Electrico y Campo Electrico Total
for i in xrange(Nres):
    for j in xrange(Nres):
        E = np.array( [Elx[i,j] + E2x[i,j], \
            Ely[i,j] + E2y[i,j]] )
        E = E/np.linalg.norm(E)
        Ex_tot[i,j], Ey_tot[i,j] = E

```

Una vez calculados los campos asociados a ambas partículas, se procede a calcular

los campos totales de toda la distribución. Para esto se tiene en cuenta el principio de superposición, obteniendo el potencial total como $\text{phi_tot} = \text{phi1} + \text{phi2}$. En el caso del campo eléctrico total \mathbf{E}_{tot} se debe usar de nuevo dos ciclos **for** para calcular la nueva normalización del campo

$$\mathbf{E}_{tot}(\mathbf{r}_{ij}) = \frac{\mathbf{E}_1(\mathbf{r}_{ij}) + \mathbf{E}_2(\mathbf{r}_{ij})}{|\mathbf{E}_1(\mathbf{r}_{ij}) + \mathbf{E}_2(\mathbf{r}_{ij})|}$$

Obteniendo así la dirección local en \mathbf{r}_{ij} del campo total.

```
#Grafica de equipotenciales
plt.contour(X, Y, phi_tot, 100)
#Grafica de lineas de campo
plt.quiver( X, Y, Ey_tot, Ex_tot)
```

Finalmente se usan las funciones `contour` y `quiver` de la librería *Matplotlib* para graficar los campos totales. La función `contour` tiene como argumentos las matrices de coordenadas X y Y , la matriz del campo total phi_tot y el número de equipotenciales a graficar. Para la función `quiver` los argumentos son las matrices de coordenadas X y Y nuevamente y las matrices de las componentes x y y del campo eléctrico total \mathbf{E}_{tot} .

2.3. Demostración 3: Billar Electrostático

En cada demostración serán introducidos gradualmente conceptos más avanzados de programación. Para este caso será usado un integrador numérico para solucionar el problema electrostático de 3 cuerpos y será representado en una animación 3D la evolución del sistema usando la librería *Mayavi2*.

El problema de tres cuerpos es un problema clásico en física y para el cual no existe solución numérica salvo casos muy particulares. Es por esta razón que el uso de métodos numéricos es completamente necesario.

Para simplificar el sistema y sin pérdida de generalidad, se asumirá una interacción en dos dimensiones y una geometría rectangular. Por este motivo se ha denominado billar electrostático, ya que este satisface la descripción del sistema (ver figura 2.4). La ecuación de movimiento para la bola i está dada por

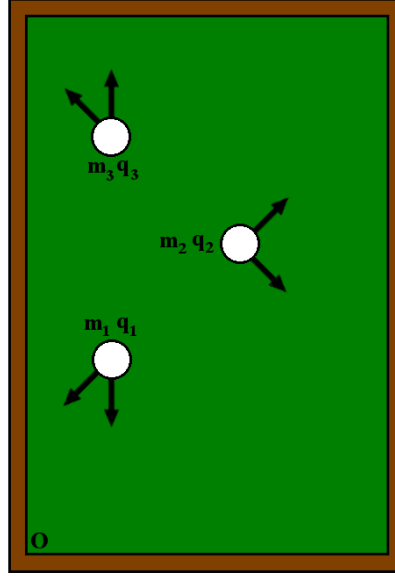


Figura 2.4: Billar electrostático con 3 bolas cargadas interactuando en dos dimensiones.

$$m_i \frac{d^2 \mathbf{r}_i}{dt^2} = \sum_{j, j \neq i}^3 \frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{|\mathbf{r}_i - \mathbf{r}_j|^3} (\mathbf{r}_i - \mathbf{r}_j) \quad (2.7)$$

donde se considera la interacción con las otras dos bolas y además se asume una distribución de carga uniforme, de tal forma que el campo eléctrico generado pueda ser reemplazado por el de una carga equivalente puntual.

Introduciendo la velocidad de la bola i como $\mathbf{v}_i = d\mathbf{r}_i/dt$, las ecuaciones de movimiento para todo el sistema pueden escribirse en coordenadas cartesianas como

$$\frac{dx_1}{dt} = v_{x,1} \quad (2.8)$$

$$\frac{dy_1}{dt} = v_{y,1} \quad (2.9)$$

$$\frac{dv_{x,1}}{dt} = \frac{q_1}{4\pi\epsilon_0 m} \left[\frac{q_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3} (x_1 - x_2) + \frac{q_3}{|\mathbf{r}_1 - \mathbf{r}_3|^3} (x_1 - x_3) \right] \quad (2.10)$$

$$\frac{dv_{y,1}}{dt} = \frac{q_1}{4\pi\epsilon_0 m} \left[\frac{q_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3} (y_1 - y_2) + \frac{q_3}{|\mathbf{r}_1 - \mathbf{r}_3|^3} (y_1 - y_3) \right] \quad (2.11)$$

$$\frac{dx_2}{dt} = v_{x,2} \quad (2.12)$$

$$\frac{dy_2}{dt} = v_{y,2} \quad (2.13)$$

$$\frac{dv_{x,2}}{dt} = \frac{q_2}{4\pi\epsilon_0 m} \left[\frac{q_1}{|\mathbf{r}_2 - \mathbf{r}_1|^3} (x_2 - x_1) + \frac{q_3}{|\mathbf{r}_2 - \mathbf{r}_3|^3} (x_2 - x_3) \right] \quad (2.14)$$

$$\frac{dv_{y,2}}{dt} = \frac{q_2}{4\pi\epsilon_0 m} \left[\frac{q_1}{|\mathbf{r}_2 - \mathbf{r}_1|^3} (y_2 - y_1) + \frac{q_3}{|\mathbf{r}_2 - \mathbf{r}_3|^3} (y_2 - y_3) \right] \quad (2.15)$$

$$\frac{dx_3}{dt} = v_{x,3} \quad (2.16)$$

$$\frac{dy_3}{dt} = v_{y,3} \quad (2.17)$$

$$\frac{dv_{x,3}}{dt} = \frac{q_3}{4\pi\epsilon_0 m} \left[\frac{q_1}{|\mathbf{r}_3 - \mathbf{r}_1|^3} (x_3 - x_1) + \frac{q_2}{|\mathbf{r}_3 - \mathbf{r}_2|^3} (x_3 - x_2) \right] \quad (2.18)$$

$$\frac{dv_{y,3}}{dt} = \frac{q_3}{4\pi\epsilon_0 m} \left[\frac{q_1}{|\mathbf{r}_3 - \mathbf{r}_1|^3} (y_3 - y_1) + \frac{q_2}{|\mathbf{r}_3 - \mathbf{r}_2|^3} (y_3 - y_2) \right] \quad (2.19)$$

El siguiente script de *Python* realiza la integración numérica del sistema y grafica las trayectorias de cada bola.

```

1  #!/usr/bin/env python
2  #=====
3  # DEMOSTRACION 3: Parte 1
4  # Solucion numerica de problema de 3 cuerpos
5  # electrostaticos
6  #=====
7  from __future__ import division
8  import numpy as np
9  import matplotlib.pyplot as plt
10 import scipy.integrate as integ
11 from RungeKutta4 import rk4_step
12
13 #Ecuaciones de movimiento
14 def dF(Y, t):
15     #Posicion X particula 1
16     x1 = Y[0]
17     #Posicion Y particula 1
18     y1 = Y[1]
19     #Velocidad X particula 1
20     vx1 = Y[2]
21     #Velocidad Y particula 1
22     vy1 = Y[3]
23

```

```

24     #Posicion X partícula 2
25     x2 = Y[4]
26     #Posicion Y partícula 2
27     y2 = Y[5]
28     #Velocidad X partícula 2
29     vx2 = Y[6]
30     #Velocidad Y partícula 2
31     vy2 = Y[7]
32
33     #Posicion X partícula 3
34     x3 = Y[8]
35     #Posicion Y partícula 3
36     y3 = Y[9]
37     #Velocidad X partícula 3
38     vx3 = Y[10]
39     #Velocidad Y partícula 3
40     vy3 = Y[11]
41
42     #Modulo distancia entre partícula 1 y 2
43     r12 = np.linalg.norm( [x1-x2, y1-y2] )
44     #Modulo distancia entre partícula 1 y 3
45     r13 = np.linalg.norm( [x1-x3, y1-y3] )
46     #Modulo distancia entre partícula 2 y 3
47     r23 = np.linalg.norm( [x2-x3, y2-y3] )
48
49     #Derivada dx/dt partícula 1
50     dx1 = vx1
51     #Derivada dy/dt partícula 1
52     dy1 = vy1
53     #Derivada d vx/dt partícula 1
54     dvx1 = q1/(4*np.pi*eps0*m1)*( q2/r12**3*(x1-x2) + \
55     q3/r13**3*(x1-x3) )
56     #Derivada d vy/dt partícula 1
57     dvy1 = q1/(4*np.pi*eps0*m1)*( q2/r12**3*(y1-y2) + \
58     q3/r13**3*(y1-y3) )
59
60     #Derivada dx/dt partícula 2
61     dx2 = vx2
62     #Derivada dy/dt partícula 2
63     dy2 = vy2
64     #Derivada d vx/dt partícula 2
65     dvx2 = q2/(4*np.pi*eps0*m2)*( q1/r12**3*(x2-x1) + \
66     q3/r23**3*(x2-x3) )

```

```

67     #Derivada d vy/dt particula 2
68     dvy2 = q2/(4*np.pi*eps0*m2)*( q1/r12**3*(y2-y1) + \
69     q3/r23**3*(y2-y3) )
70
71     #Derivada dx/dt particula 3
72     dx3 = vx3
73     #Derivada dy/dt particula 3
74     dy3 = vy3
75     #Derivada d vx/dt particula 3
76     dvx3 = q3/(4*np.pi*eps0*m3)*( q1/r13**3*(x3-x1) + \
77     q2/r23**3*(x3-x2) )
78     #Derivada d vy/dt particula 3
79     dvy3 = q3/(4*np.pi*eps0*m3)*( q1/r13**3*(y3-y1) + \
80     q2/r23**3*(y3-y2) )
81
82     #Derivadas
83     return np.array([ dx1, dy1, dvx1, dvy1, \
84     dx2, dy2, dvx2, dvy2, \
85     dx3, dy3, dvx3, dvy3 ])
86
87
88     #CONSTANTES
89     #Permitividad del vacio
90     eps0 = 8.85418e-12
91     #Ancho de la mesa
92     ancho = 1.2
93     #Largo de la mesa
94     largo = 2.4
95
96     #CONDICIONES BOLA 1
97     #masa
98     m1 = 0.1
99     #carga
100    q1 = 5e-5
101    #radio
102    r1 = 0.05
103    #posicion inicial
104    x10 = 0.1
105    y10 = 0.1
106    #velocidad inicial
107    vx10 = 5.0
108    vy10 = 5.0
109

```

```
110 #CONDICIONES BOLA 2
111 #masa
112 m2 = 0.1
113 #carga
114 q2 = 5e-5
115 #radio
116 r2 = 0.05
117 #posicion inicial
118 x20 = 0.3
119 y20 = 0.1
120 #velocidad inicial
121 vx20 = -5.0
122 vy20 = 5.0
123
124 #CONDICIONES BOLA 3
125 #masa
126 m3 = 0.1
127 #carga
128 q3 = 5e-5
129 #radio
130 r3 = 0.05
131 #posicion inicial
132 x30 = 0.2
133 y30 = 0.2
134 #velocidad inicial
135 vx30 = -1.0
136 vy30 = 5.0
137
138 #INTEGRACION DEL SISTEMA
139 #tiempo maximo a integrar
140 t_max = 10
141 #salto del tiempo
142 t_step = 0.001
143 #condiciones iniciales
144 cond_ini = [ x10, y10, vx10, vy10, \
145 x20, y20, vx20, vy20, \
146 x30, y30, vx30, vy30]
147 #tiempo de evaluacion
148 tiempo = np.arange( 0, t_max, t_step )
149 #integracion del sistema
150 solucion = []
151 Y = cond_ini
152 for t in tiempo:
```

```
153     Y = rk4_step( dF, Y, t, t_step )
154     #Condiciones de colision con la mesa
155     if Y[0] < r1 or Y[0] >= ancho-r1:
156         Y[2] = -Y[2]
157     if Y[4] < r2 or Y[4] >= ancho-r2:
158         Y[6] = -Y[6]
159     if Y[8] < r3 or Y[8] >= ancho-r3:
160         Y[10] = -Y[10]
161
162     if Y[1] < r1 or Y[1] >= largo-r1:
163         Y[3] = -Y[3]
164     if Y[5] < r2 or Y[5] >= largo-r2:
165         Y[7] = -Y[7]
166     if Y[9] < r3 or Y[9] >= largo-r3:
167         Y[11] = -Y[11]
168
169     solucion.append( Y )
170
171     #resultado de integracion
172     x1_t, y1_t, vx1_t, vy1_t, \
173     x2_t, y2_t, vx2_t, vy2_t, \
174     x3_t, y3_t, vx3_t, vy3_t = \
175     np.transpose( solucion )
176
177     #Guardando archivo de datos
178     np.savetxt( 'trayectorias.txt', np.transpose([tiempo,\
179     x1_t, y1_t, x2_t, y2_t, x3_t, y3_t]) )
180
181     #Grafica de trayectorias
182     plt.plot( x1_t, y1_t, label='particula 1' )
183     plt.plot( x2_t, y2_t, label='particula 2')
184     plt.plot( x3_t, y3_t, label='particula 3')
185
186     #Formato de grafica
187     plt.xlim( (0,ancho) )
188     plt.ylim( (0,largo) )
189     plt.grid()
190     plt.legend()
191     plt.show()
```

Se obtiene la siguiente figura con las trayectorias de cada bola

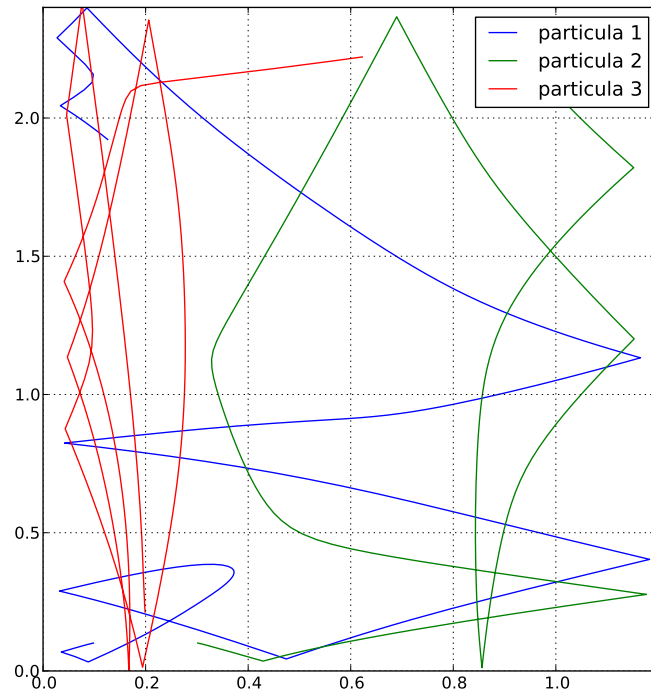


Figura 2.5: Trayectoria calculada numéricamente de las tres bolas del problema.

y un archivo de texto `trayectorias.txt` con las soluciones del sistema en el formato `[tiempo, x1, y1, x2, y2, x3, y3]`, donde x_i , y_i denotan la posición x y y de la bola i .

A continuación se explica cada parte del código

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
from RungeKutta4 import rk4_step
```

En estas líneas iniciales se cargan las librerías estándares ya conocidas. En la última línea `from RungeKutta4 import rk4_step` se importa la función `rk4_step` que corresponde al integrador numérico para solucionar las ecuaciones

de movimiento. Este se puede encontrar en el archivo `RungeKutta4.py`. Note que a pesar de no ser una librería, *Python* permite tratarlo como tal, e importar funciones específicas desde un archivo externo.

```
#Ecuaciones de movimiento
def dF(Y, t):
    #Posicion X partícula 1
    x1 = Y[0]
    #Posicion Y partícula 1
    y1 = Y[1]
    #Velocidad X partícula 1
    vx1 = Y[2]
    #Velocidad Y partícula 1
    vy1 = Y[3]

    #Posicion X partícula 2
    x2 = Y[4]
    #Posicion Y partícula 2
    y2 = Y[5]
    #Velocidad X partícula 2
    vx2 = Y[6]
    #Velocidad Y partícula 2
    vy2 = Y[7]

    #Posicion X partícula 3
    x3 = Y[8]
    #Posicion Y partícula 3
    y3 = Y[9]
    #Velocidad X partícula 3
    vx3 = Y[10]
    #Velocidad Y partícula 3
    vy3 = Y[11]

    #Modulo distancia entre partícula 1 y 2
    r12 = np.linalg.norm( [x1-x2, y1-y2] )
    #Modulo distancia entre partícula 1 y 3
    r13 = np.linalg.norm( [x1-x3, y1-y3] )
    #Modulo distancia entre partícula 2 y 3
    r23 = np.linalg.norm( [x2-x3, y2-y3] )

    #Derivada dx/dt partícula 1
    dx1 = vx1
```



```

#Derivada dy/dt partícula 1
dy1 = vy1
#Derivada d vx/dt partícula 1
dvx1 = q1/(4*np.pi*eps0*m1)*( q2/r12**3*(x1-x2) + \
q3/r13**3*(x1-x3) )
#Derivada d vy/dt partícula 1
dvy1 = q1/(4*np.pi*eps0*m1)*( q2/r12**3*(y1-y2) + \
q3/r13**3*(y1-y3) )

#Derivada dx/dt partícula 2
dx2 = vx2
#Derivada dy/dt partícula 2
dy2 = vy2
#Derivada d vx/dt partícula 2
dvx2 = q2/(4*np.pi*eps0*m2)*( q1/r12**3*(x2-x1) + \
q3/r23**3*(x2-x3) )
#Derivada d vy/dt partícula 2
dvy2 = q2/(4*np.pi*eps0*m2)*( q1/r12**3*(y2-y1) + \
q3/r23**3*(y2-y3) )

#Derivada dx/dt partícula 3
dx3 = vx3
#Derivada dy/dt partícula 3
dy3 = vy3
#Derivada d vx/dt partícula 3
dvx3 = q3/(4*np.pi*eps0*m3)*( q1/r13**3*(x3-x1) + \
q2/r23**3*(x3-x2) )
#Derivada d vy/dt partícula 3
dvy3 = q3/(4*np.pi*eps0*m3)*( q1/r13**3*(y3-y1) + \
q2/r23**3*(y3-y2) )

#Derivadas
return np.array([ dx1, dy1, dvx1, dvy1, \
dx2, dy2, dvx2, dvy2, \
dx3, dy3, dvx3, dvy3 ])

```

En esta parte se define la función dinámica del sistema, la cual contiene todas las derivadas asociadas a las ecuaciones de movimiento 2.8 - 2.19. El arreglo Y contiene todas las variables del sistema en el orden x_i, y_i, v_{xi}, v_{yi} para cada bola. En las primeras líneas se extrae entonces cada valor para las bolas, por ejemplo para la bola 1 se tiene $x1 = Y[0]$, $y1 = Y[1]$, $vx1 = Y[2]$ y $vy1 = Y[3]$ y de igual forma para las otras dos.

Luego se calcula el módulo de distancia entre los vectores de posición de cada bola, por ejemplo entre las bolas 1 y 2 se tiene $r_{12} = \text{np.linalg.norm}([x_1 - x_2, y_1 - y_2])$, de igual forma para las bolas 1 y 3 y las bolas 2 y 3..

Finalmente se calculan las derivadas de las variables de cada bola acorde a las ecuaciones de movimiento y se retorna un arreglo con todas estas en el mismo orden en que están en el arreglo inicial Y. `np.array([dx1, dy1, dvx1, dvy1, dx2, dy2, dvx2, dvy2, dx3, dy3, dvx3, dvy3])`. La función `array` de la librería *NumPy* se usa con el fin de convertir una lista en un objeto matemático tipo vector.

```
#CONSTANTES
#Permitividad del vacio
eps0 = 8.85418e-12
#Ancho de la mesa
ancho = 1.2
#Largo de la mesa
largo = 2.4
```

Se definen las constantes del sistema, la permitividad del vacío ϵ_0 , y las dimensiones de la mesa, todo en unidades SI.

```
#CONDICIONES BOLA 1
#masa
m1 = 0.1
#carga
q1 = 5e-5
#radio
r1 = 0.05
#posicion inicial
x10 = 0.1
y10 = 0.1
#velocidad inicial
vx10 = 5.0
vy10 = 5.0
```

Las condiciones físicas de la bola 1 son definidas en estas líneas. Se dan valores a la masa, la carga, el radio de la bola, su posición inicial y su velocidad inicial, todo en unidades SI. Esto se repite para las otras dos bolas.

```
#INTEGRACION DEL SISTEMA
```

```
#tiempo maximo a integrar
t_max = 10
#salto del tiempo
t_step = 0.001
#condiciones iniciales
cond_ini = [ x10, y10, vx10, vy10, \
x20, y20, vx20, vy20, \
x30, y30, vx30, vy30]
#tiempo de evaluacion
tiempo = np.arange( 0, t_max, t_step )
```

Se procede con la integración de la ecuaciones del sistema. Primero se define el tiempo máximo en el cual se desea evolucionar las bolas, en este caso 10 s. Luego se define el paso de integración, que corresponde al valor del intervalo de tiempo en que se van almacenando los valores x y y de las bolas. Se construye un arreglo con las condiciones iniciales de posición y velocidad definidas para cada bola `cond_ini`. Finalmente se construye el arreglo con todos los tiempos en los que se desea calcular las soluciones `tiempo = np.arange(0, t_max, t_step)`

```
#integracion del sistema
solucion = []
Y = cond_ini
for t in tiempo:
    Y = rk4_step( dF, Y, t, t_step )
    #Condiciones de colision con la mesa
    if Y[0] < r1 or Y[0] >= ancho-r1:
        Y[2] = -Y[2]
    if Y[4] < r2 or Y[4] >= ancho-r2:
        Y[6] = -Y[6]
    if Y[8] < r3 or Y[8] >= ancho-r3:
        Y[10] = -Y[10]

    if Y[1] < r1 or Y[1] >= largo-r1:
        Y[3] = -Y[3]
    if Y[5] < r2 or Y[5] >= largo-r2:
        Y[7] = -Y[7]
    if Y[9] < r3 or Y[9] >= largo-r3:
        Y[11] = -Y[11]

    solucion.append( Y )
```

```
#resultado de integracion
x1_t, y1_t, vx1_t, vy1_t, \
x2_t, y2_t, vx2_t, vy2_t, \
x3_t, y3_t, vx3_t, vy3_t = \
np.transpose( solucion )
```

Se crea un arreglo vacío `solucion = []` donde se almacenan las soluciones de la trayectorias en cada tiempo. Luego se define el arreglo `Y` con las condiciones iniciales `Y = cond_ini` para posteriormente comenzar una iteración en el tiempo, calculando en cada ciclo las nuevas soluciones. Se usa entonces la función `rk4_step` del archivo `RungeKutta4.py` para calcular la solución del siguiente tiempo `t+t_step`, `Y = rk4_step(dF, Y, t, t_step)`. Los argumentos de esta función son entonces, el nombre de la función con las ecuaciones de movimiento `dF`, las soluciones en el tiempo anterior `Y`, el tiempo actual `t` y el salto de tiempo de integración `t_step`. Luego se extrae del arreglo (matriz) `solucion` cada una de las soluciones en el tiempo, para esto se usa la función `transpose` de *NumPy* en orden para tomar los datos acorde a las columnas y no a las filas.

```
#Guardando archivo de datos
np.savetxt( 'trayectorias.txt', np.transpose([tiempo,\
x1_t, y1_t, x2_t, y2_t, x3_t, y3_t]) )

#Grafica de trayectorias
plt.plot( x1_t, y1_t, label='particula 1' )
plt.plot( x2_t, y2_t, label='particula 2' )
plt.plot( x3_t, y3_t, label='particula 3' )

#Formato de grafica
plt.xlim( (0,ancho) )
plt.ylim( (0,largo) )
plt.grid()
plt.legend()
plt.show()
```

En esta última parte se guarda en un archivo externo `trayectorias.txt` las trayectorias calculadas en el formato `tiempo, x1_t, y1_t, x2_t, y2_t, x3_t, y3_t`. Para esto se usa la función `saveetxt` de la librería *NumPy*, esta tiene como argumentos el nombre del archivo externo a guardar y los datos calculados. Estos se dan en un arreglo (matriz) que se transpone de nuevo para obtener los datos acorde a las columnas y no a las filas. En lo siguiente se grafica la trayectoria de cada bola usando la función `plot` de *Matplotlib*. Finalmente se da formato a la

ventana de graficación y se muestra en pantalla.

En la segunda parte de esta demostración se usa el archivo de texto guardado en el primer script para representar en una animación 3D el sistema físico. Para esto se usa la librería *MayaVi2* y el script es el siguiente.

```
1  #!/usr/bin/env python
2  #=====
3  # DEMOSTRACION 3: Parte 2
4  # Solucion numerica de problema de 3 cuerpos
5  # electrostaticos. Animacion 3D
6  #=====
7  import numpy as np
8  import enthought.tvtk.tools.visual as visual
9
10 #Cargando datos de las bolas
11 tiempo, x1_t, y1_t, x2_t, y2_t, x3_t, y3_t = \
12 np.transpose( np.loadtxt('trayectorias.txt') )
13
14 #CONSTANTES
15 #Ancho de la mesa
16 ancho = 1.2
17 #Largo de la mesa
18 largo = 2.4
19 #Grosor de los muros del billar
20 grosor = 0.1
21 #Radio bola 1
22 r1 = 0.05
23 #Radio bola 2
24 r2 = 0.05
25 #Radio bola 3
26 r3 = 0.05
27
28 #Creando bola 1
29 bola1 = visual.sphere( radius=r1, color=(1.0, 1.0, 1.0) )
30 bola1.pos = [ 0., 0., 0. ]
31 bola1.t = 0
32 bola1.dt = 1
33
34 #Creando bola 2
35 bola2 = visual.sphere( radius=r2, color=(1.0, 1.0, 1.0) )
```

```
36 bola2.pos = [ 0., 0., 0. ]
37 bola2.t = 0
38 bola2.dt = 1
39
40 #Creando bola 1
41 bola3 = visual.sphere( radius=r3, color=(1.0, 1.0, 1.0) )
42 bola3.pos = [ 0., 0., 0. ]
43 bola3.t = 0
44 bola3.dt = 1
45
46 #Creando mesa
47 mesa = visual.box( pos=(ancho/2., largo/2., -grosor/2.), \
48 size=(ancho, largo, grosor), color=(0.0, 0.3, 0.0) )
49
50 muro_l = visual.box( pos=(-grosor/2., largo/2., 0.0), \
51 size=(grosor, largo + 2*grosor, grosor), \
52 color=(0.6, 0.3, 0.0) )
53 muro_r = visual.box( pos=(ancho+grosor/2., largo/2., 0.0), \
54 size=(grosor, largo + 2*grosor, grosor), \
55 color=(0.6, 0.3, 0.0) )
56 muro_d = visual.box( pos=(ancho/2., -grosor/2., 0.0), \
57 size=(ancho + 2*grosor, grosor, grosor), \
58 color=(0.6, 0.3, 0.0) )
59 muro_u = visual.box( pos=(ancho/2., largo+grosor/2., 0.0), \
60 size=(ancho + 2*grosor, grosor, grosor), \
61 color=(0.6, 0.3, 0.0) )
62
63
64 #ITERACION DEL SISTEMA
65 def anim():
66     #Evolucion de la bola 1
67     bola1.t = bola1.t + bola1.dt
68     i = bola1.t
69     bola1.pos = visual.vector( x1_t[i], y1_t[i], r1 )
70
71     #Evolucion de la bola 2
72     bola2.t = bola2.t + bola2.dt
73     i = bola2.t
74     bola2.pos = visual.vector( x2_t[i], y2_t[i], r2 )
75
76     #Evolucion de la bola 3
77     bola3.t = bola3.t + bola3.dt
78     i = bola3.t
```

```
79     bola3.pos = visual.vector( x3_t[i], y3_t[i], r3 )
80
81 a = visual.iterate(10, anim)
82 visual.show()
```

El resultado obtenido está ilustrado en la siguiente figura

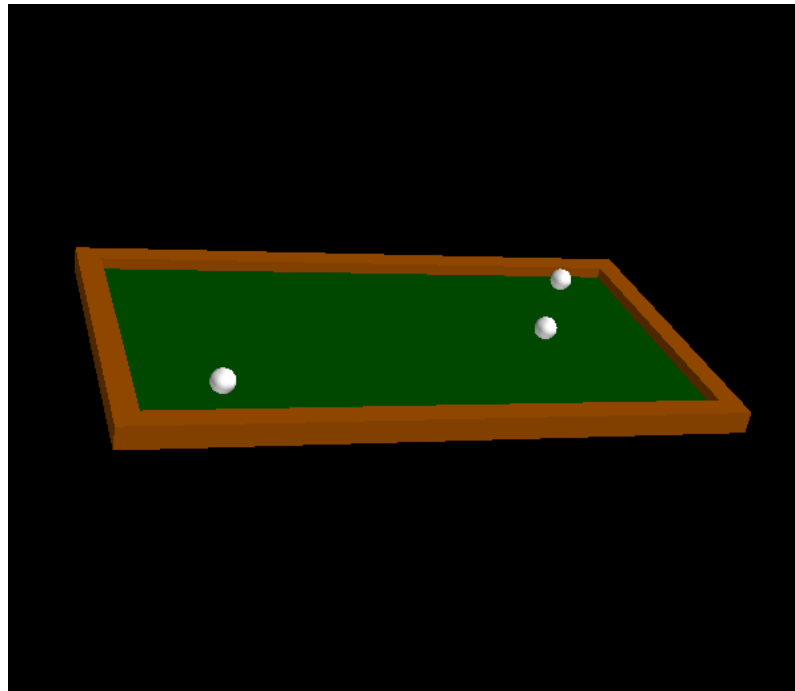


Figura 2.6: Animación 3D del sistema de billar electrostático usando *MayaVi2* .

A continuación se describe el código anterior

```
import numpy as np
import enthought.tvtk.tools.visual as visual
```

En estas primeras líneas se carga las diferentes librerías necesarias para esta segunda parte de la demostración. Primero la librería *NumPy* ya conocida y luego se carga el paquete *visual* de la librería *MayaVi2* . **Nota:** para algunas versiones de *MayaVi2* la forma correcta de cargar esta librería es `import tvtk.tools.visual as visual`.

```
#Cargando datos de las bolas
tiempo, x1_t, y1_t, x2_t, y2_t, x3_t, y3_t = \
```

```
np.transpose( np.loadtxt('trayectorias.txt') )
```

Usando la función `loadtxt` de la librería *NumPy* se cargan los datos guardados en el primer script. Como único argumento se tiene el nombre del archivo. De nuevo se usa la función `transpose` para cargar los datos acorde a las columnas y no a las filas.

```
#CONSTANTES
#Ancho de la mesa
ancho = 1.2
#Largo de la mesa
largo = 2.4
#Grosor de los muros del billar
grosor = 0.1
#Radio bola 1
r1 = 0.05
#Radio bola 2
r2 = 0.05
#Radio bola 3
r3 = 0.05
```

En esta parte se definen nuevamente los parámetros físicos del sistema asociados longitudes.

```
#Creando bola 1
bola1 = visual.sphere( radius=r1, color=(1.0, 1.0, 1.0) )
bola1.pos = [ 0., 0., 0. ]
bola1.t = 0
bola1.dt = 1
```

Usando la función `sphere` del módulo `visual` de *MayaVi2* se construye la bola 1 del sistema. Como argumentos de la función `sphere` se da su radio `radius` y el color como `color`. El formato de color está en código RGB donde un color es un arreglo de tres números que toman valores entre 0 y 1. La primera componente determina la cantidad de rojo, con 0 nulo y 1 máximo, de igual forma la segunda componente para el verde y la tercera para el azul. Como ejemplos, el rojo equivale a $(1, 0, 0)$, mientras que el negro $(0, 0, 0)$, el blanco $(1, 1, 1)$, el violeta $(1, 0, 1)$, etc. Luego se da una posición inicial de la bola, la cual no es definitiva y es cambiada cuando se cargue la trayectoria correspondiente. Finalmente se da el tiempo inicial y el salto de tiempo `dt`. Este salto de tiempo no corresponde al salto de tiempo físico del script anterior, en vez de esto, este debe ser un entero (1) debido

a que corresponde a la posición en los arreglos de las trayectorias entre tiempo y tiempo. Esto se repite para las otras 2 bolas.

```
#Creando mesa
mesa = visual.box( pos=(ancho/2., largo/2., -grosor/2.), \
size=(ancho, largo, grosor), color=(0.0, 0.3, 0.0) )
```

En estas líneas se define el objeto asociado a la mesa de billar. Para esto se usa la función `box` de `visual` de la siguiente forma `mesa = visual.box(pos=(ancho /2., largo/2., -grosor/2.), size=(ancho, largo, grosor), color=(0.0, 0.3, 0.0))`. Como primer argumento se da la posición del centro geométrico de la caja, el segundo argumento es un arreglo con el grosor, el alto y el ancho de la caja, en el mismo orden. Finalmente se da el color en formato RGB, con `(0.0, 0.3, 0.0)` equivalente al color verde oscuro. De igual forma se crean los límites de la mesa.

```
#ITERACION DEL SISTEMA
def anim():
    #Evolucion de la bola 1
    bola1.t = bola1.t + bola1.dt
    i = bola1.t
    bola1.pos = visual.vector( x1_t[i], y1_t[i], r1 )
```

Se define la función `anim` encargada de trazar la trayectoria de cada bola. Inicialmente se aumenta el tiempo asociado a la bola al siguiente valor `t+dt`. Luego se define el índice `i` en los arreglos de datos asociado a los datos del tiempo actual. Finalmente se actualiza la posición de la bola 1 con el atributo `pos` del objeto `bola1` de tal forma que `bola1.pos = visual.vector(x1_t[i], y1_t[i], r1)`. Se debe tener en cuenta que la posición en el eje `z` debe ser el radio de la bola `r1` y no 0, esto para que la bola no intercepte la mesa sino que esté sobre ella.

```
a = visual.iterate(10, anim)
visual.show()
```

Finalmente se ejecuta la función `visual.iterate`, esta presenta la animación del sistema de forma gráfica. Como primer argumento se da el tiempo en milisegundos que se quiere entre cada iteración del sistema, así por ejemplo un valor bajo como 10 implica un tiempo de 10 ms entre frame y frame, produciendo una animación más fluida, mientras que un valor más alto como 50 produce una animación en cámara lenta. El segundo argumento es el nombre de la función donde está la evolución de los péndulos, en este caso `anim`.

2.4. Ejercicios

Ejercicio 1 (★)

Determinación de relación carga masa

Considere de nuevo el espectrómetro de masas de la demostración 4.1. Usando un estroboscopio se midió la posición de una partícula cargada en un campo eléctrico $E = 1 \text{ N/C} \mathbf{j}$ y se obtuvieron los siguientes datos <https://raw.githubusercontent.com/sbustamante/Computacional-Campos/master/codigos/datos.txt>.

Teniendo en cuenta que la partícula se lanza desde el origen y que la rapidez de lanzamiento inicial en la dirección x fue de $v_{x0} = 1 \text{ m/s}$, realice un programa que realice los siguiente numerales:

- Calcule, usando la ecuación de trayectoria 2.4 y un ajuste por χ^2 de los datos, la relación carga masa de la partícula.
- Grafique en una misma ventana la trayectoria correspondiente a la relación carga masa que encontró, y los datos de la trayectoria (grafique los datos con puntos y no con una línea continua).

Ejercicio 2 (★)

Trayectoria en campo no homogéneo

Considere una partícula de masa m y carga q embebida en un campo eléctrico con la siguiente forma funcional.

$$\mathbf{E}(x, y) = A(x^2 + y^2)\mathbf{i} + B \sin(x^2 - y^2)\mathbf{j}$$

Haga un código que utilice la rutina de integración `RungeKutta4.py` de la demostración 2.3 para integrar la trayectoria de esta partícula en el campo \mathbf{E} . Puede guiarse de la demostración 2.3, además puede usar valores numéricos de las condiciones iniciales, constantes A y B y masa m y carga q de su elección. Finalmente, usando la función `quiver` de *Matplotlib*, grafique las líneas de campo¹. Grafique también en la misma ventana la trayectoria calculada de la partícula cargada.

¹Recuerde que para graficar líneas de campo debe normalizar el campo tal como $\mathbf{E}/|\mathbf{E}|$. Para más información, remítase a la demostración 2.2.

BONUS (★)

Este ejercicio puede ser considerado de dos estrellas si se realiza lo siguiente.

Al final del código anterior, guarde en un archivo externo `.txt` los datos de la trayectoria integrada con el formato `[tiempo, coordenada x, coordenada y]`. Luego haga otro código que, usando la librería *MayaVi2*, realice una animación 3D de la trayectoria, cargando previamente los datos almacenados (use la función `loadtxt` de la librería *NumPy*). *Hint*: En la ventana de animación, es conveniente crear una mesa con el fin de tener un punto de referencia para notar el movimiento de la partícula.

Ejercicio 3 (★)**Billar electrostático con campo eléctrico**

Considere la demostración 2.3. Modifique ambos scripts para incluir la acción de un campo eléctrico uniforme y homogéneo sobre las bolas cargadas, además agregue otra bola al problema. La dirección y magnitud del campo eléctrico, y las condiciones iniciales de la cuarta bola son de libre elección.

Ejercicio 4 (★)**Líneas de potencial de una barra finita**

Considere una barra finita de carga Q y longitud L , de tal forma que la densidad lineal de carga está dada por $\sigma = Q/L$. Un método computacional muy popular para solucionar problemas de distribuciones continuas consiste en reducir el problema a un conjunto de cargas puntuales con una distribución *equivalente*.

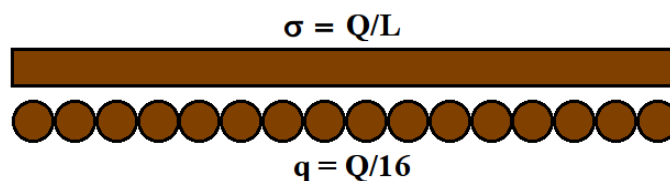


Figura 2.7: Potencial de una barra finita cargada.

Realice un código que calcule las líneas de equipotencial de una distribución de 16 partículas cargadas con carga $q = Q/16$ separadas una distancia $l = L/16$ entre ellas. Para esto puede guiarse de la demostración 2.2. ¿La estructura del potencial es similar a lo que espera del potencial de la barra? Tome los valores numéricos de Q y L como desee.

Capítulo 3

Magnetostática

3.1. Ejercicios

Ejercicio 1 (★)

Trayectoria en campo no homogéneo

Considere una partícula de masa m y carga q embebida en un campo magnético con la siguiente forma funcional.

$$\mathbf{B}(x, y) = A(x^2 + y^2)\mathbf{k}$$

Haga un código que utilice la rutina de integración `RungeKutta4.py` de la demostración 2.3 para integrar la trayectoria de esta partícula en el campo \mathbf{B} . Puede guiarse de la demostración 2.3, además puede usar valores numéricos de las condiciones iniciales, constante A , masa m y carga q de su elección. Finalmente grafique la trayectoria calculada de la partícula cargada y realice una animación 3D de la partícula usando la librería *MayaVi2*.

Hint: En la ventana de animación, es conveniente crear una mesa con el fin de tener un punto de referencia para notar el movimiento de la partícula.

Ejercicio 2 (★)

Billar electrostático con campo magnético

Considere la demostración 2.3. Modifique ambos scripts para incluir la acción de un campo magnético uniforme y homogéneo que entra hacia el tablero. La magnitud del campo magnético es de libre elección.

Ejercicio 3 (★)**Campo magnético de 4 corrientes**

Considere cuatro alambres por los cuales circula una corriente I_0 , tal como se muestra en la siguiente figura. Usando la función `quiver` de *Matplotlib*, graficar las líneas de campo magnético en un plano perpendicular a las corrientes. Escoja los valores que desee para las cantidades del problema.

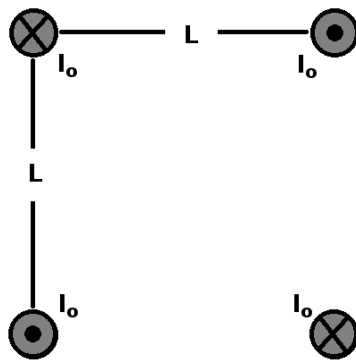


Figura 3.1: Cuatro corrientes de igual magnitud.

Ejercicio 4 (★★)**Partícula en un campo dipolar**

Considere una partícula cargada en un campo magnético dipolar, integre numéricamente usando la rutina de integración `RungeKutta4.py` la trayectoria de la partícula. Realice los siguientes numerales

- Grafique usando la función `quiver` de *Matplotlib* las líneas de campo magnético.
- Encuentre una condición inicial (velocidad y posición) para la cual la partícula interactúe apreciablemente con el campo magnético. Una interacción interesante puede ser el confinamiento de la trayectoria en zonas cercanas a los polos del dipolo (Este fenómeno explica las auroras boreales). Guarde la trayectoria obtenida en un archivo de texto con el formato $[x, y, z, vx, vy, vz]$.
- Realice una animación 3D usando *Mayavi2* donde ilustre el movimiento de la partícula cargada.

Hint: En la ventana de animación, es conveniente crear un objeto con el fin de tener un punto de referencia para notar el movimiento de la partícula. Este objeto puede ser una esfera en el centro del campo dipolar.

Escoja los valores que desee para las cantidades físicas del problema.

Capítulo 4

Electrodinámica

La electrodinámica aborda la evolución temporal de sistemas donde interactúan campos magnéticos y eléctricos y cargas en movimiento. En este caso se tratan temas como la ley de inducción de Faraday y posibles aplicaciones, circuitos RL, LC, corriente alterna, etc.

4.1. Demostración 1: Bobina fonocaptora

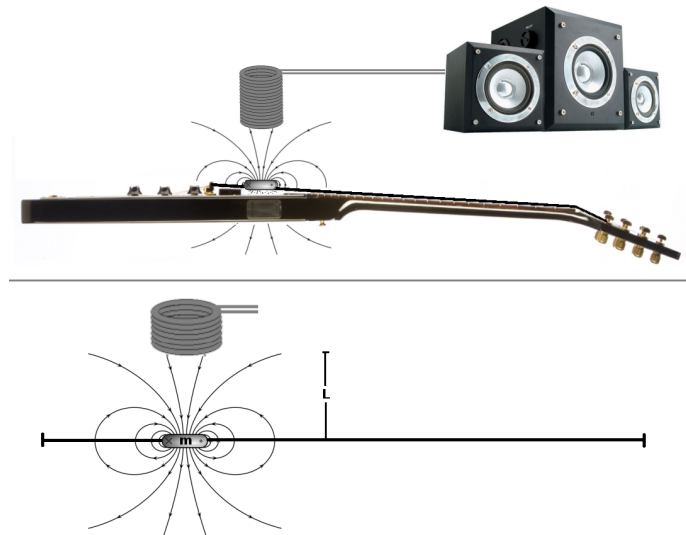


Figura 4.1: Bobina fonocaptora.

Una interesante aplicación del fenómeno de inducción es la bobina fonocaptora. Este dispositivo permite la amplificación del sonido en una guitarra eléctrica a partir de la corriente de inducción generada en una bobina. Un montaje ilustrativo del experimento puede verse en la figura 4.1.

El funcionamiento de este dispositivo se explica a continuación. La cuerda de la guitarra tiene un material que puede ser magnetizado por un campo externo permanente, creando así un momento dipolar inducido y por tanto un campo magnético dipolar de la forma

$$\mathbf{B}(\mathbf{r}) = \frac{\mu_0}{4\pi} \left(\frac{3\mathbf{r}(\mathbf{m} \cdot \mathbf{r})}{r^5} - \frac{\mathbf{m}}{r^3} \right) \quad (4.1)$$

Por simplicidad, y teniendo en cuenta que la mayor contribución al flujo magnético a través de la bobina es debida a la componente vertical del campo magnético, se puede reescribir el campo dipolar como

$$\mathbf{B}(\mathbf{r}) \approx \frac{\mu_0 m}{2\pi r^3} \mathbf{u}_j \quad (4.2)$$

donde el vector unitario \mathbf{u}_j se ha definido perpendicular a la cuerda en reposo y apuntando hacia la bobina.

Para la cuerda, se asumirá que tiene extremos fijos, de tal forma que la solución de onda tiene la forma

$$y(x, t) = y_0(t) \sin \left[2\pi \left(f_1 t - \frac{x}{\lambda_1} \right) \right] \quad (4.3)$$

donde $y_0(t) = y_i e^{-t/\tau}$, con y_i la amplitud inicial y τ el tiempo de vida media de la oscilación, f_1 y λ_1 la frecuencia y longitud de onda fundamental, t el tiempo y x la coordenada horizontal definida sobre la cuerda.

La frecuencia y longitud de onda fundamental en una cuerda con extremos fijos está dada por

$$f_1 = \frac{1}{2D} v \quad \lambda_1 = \frac{2D}{n} \quad v = \sqrt{\frac{T}{\mu}} \quad (4.4)$$

donde v es la velocidad de la onda en la cuerda, D la longitud, T la tensión y μ la densidad lineal.

Puesto que solo interesa evaluar la oscilación de la cuerda donde está ubicado el material magnetizado, se puede reducir la expresión 4.3 a

$$y_m(t) = y(x_m, t) = y_0(t) \sin[2\pi f_1 t + \delta] \quad (4.5)$$

donde y_m es la amplitud de la onda evaluada en la posición del material magnetizado x_m , y sin pérdida de generalidad, la fase obtenida δ se puede hacer nula $\delta = 0$.

El campo magnético medido en la ubicación de la espira debido al material magnetizado sobre la cuerda es

$$\mathbf{B}(\mathbf{r}) \approx \frac{\mu_0}{2\pi} \frac{m}{[L - y_m(t)]^3} \mathbf{u}_j \quad (4.6)$$

Calculando el potencial inducido en la bobina fonocaptora, se obtiene

$$V_{fem} = N \frac{d\phi}{dt} = NA \frac{dB}{dt} \quad (4.7)$$

$$= NA \frac{3\mu_0}{2\pi} \frac{m}{[L - y_m(t)]^4} \frac{dy_m(t)}{dt} \quad (4.8)$$

Usando la expresión para la amplitud de la oscilación en la cuerda se llega a

$$\frac{V_{fem}}{V_0} = \left[1 - \frac{y_0}{L} \sin(2\pi f_1 t)\right]^{-4} \frac{y_0}{L} e^{-t/\tau} \left[2\pi f_1 \cos(2\pi f_1 t) - \frac{1}{\tau} \sin(2\pi f_1 t)\right] \quad (4.9)$$

donde se ha definido el potencial de referencia V_0 como

$$V_0 = \frac{3\mu_0 N A m}{2\pi L^3}$$

A continuación se ilustra el código para el cálculo del problema

```

1  #!/usr/bin/env python
2  #=====
3  # DEMOSTRACION 1:
4  # Bobina fonocaptora
5  #=====
6  import numpy as np
7  import matplotlib.pyplot as plt
8  import AudioLib as ad
9
10 #Funcion de oscilacion original
11 def y(t):
12     return y0*np.exp(-t/tau)*np.sin( 2*np.pi*frequ1*t )
13

```

```

14 #Funcion de potencial inducido por bobina fonocaptora
15 def Vfem(t):
16     return (1-y0/L*np.sin( 2*np.pi*freq1*t ))**-4*y0/L*\
17     np.exp( -t/tau )*( 2*np.pi*freq1*\
18     np.cos( 2*np.pi*freq1*t ) - \
19     1/tau*np.sin( 2*np.pi*freq1*t ) )
20
21
22 #CONSTANTES
23 #Frecuencia fundamental de la cuerda [Hz]
24 freq1 = 440.
25 #Amplitud maxima de oscilacion de la cuerda [m]
26 y0 = 0.002
27 #Longitud de la cuerda [m]
28 L = 0.7
29 #Tiempo de vida medio de una oscilacion [s]
30 tau = 1.
31
32 #Tiempo maximo [s]
33 tmax = 5
34 #Intervalos
35 dt = 1/44100.
36 #Arreglo de tiempo
37 tiempo = np.arange( 0, tmax, dt )
38
39 #Potencial
40 V = Vfem( tiempo )
41 #Oscilacion original
42 y_cuerda = y( tiempo )
43
44 #Grafica
45 plt.plot( tiempo, V, linewidth = 0.5 )
46 plt.grid()
47 plt.title( "Potencial inducido por una bobina fonocaptora")
48 plt.xlabel( "t [s]" )
49 plt.ylabel( "Potential [V_0]" )
50 plt.show()
51
52 #Audio producido por la bobina
53 nota_bobina = ad.audio()
54 nota_bobina.load( V*ad.Amplitude/max(V) )
55 nota_bobina.play()
56

```

```
57 #Audio original de la cuerda
58 nota_cuerda = ad.audio()
59 nota_cuerda.load( y_cuerda*ad.Amplitude/max(y_cuerda) )
60 nota_cuerda.play()
```

El resultado que se obtiene es

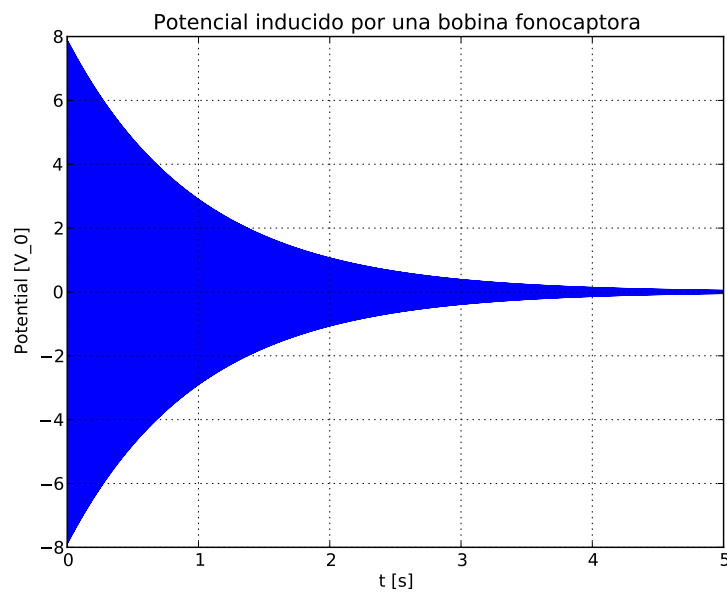


Figura 4.2: Potencial de inducción producido por una bobina fonocaptora.

El potencial obtenido es una función oscilante con la misma frecuencia de la cuerda original, lo que permite entonces digitalizar el sonido de la guitarra. Ambos sonidos son recreados por el código anterior, primero el sonido producido por un parlante conectado a la bocina y el segundo, el sonido producido por la guitarra originalmente.

A continuación se explica cada parte del código

```
import numpy as np
import matplotlib.pyplot as plt
import AudioLib as ad
```

Se importan las librerías estándar más la librería `AudioPy.py` con el alias de `ad`. Esta librería se anexa en la sección de códigos de la parte del curso y permite la

manipulación de audio desde *Python* . En orden para correr el código de la demostración apropiadamente, es necesario instalar el paquete `pyaudio`

```
\$ sudo apt-get install python-pyaudio
```

```
#Funcion de oscilacion original
def y(t):
    return y0*np.exp(-t/tau)*np.sin( 2*np.pi*frequ1*t )

#Funcion de potencial inducido por bobina fonocaptora
def Vfem(t):
    return (1-y0/L*np.sin( 2*np.pi*frequ1*t ))**-4*y0/L*\
    np.exp( -t/tau )*( 2*np.pi*frequ1*\
    np.cos( 2*np.pi*frequ1*t ) - \
    1/tau*np.sin( 2*np.pi*frequ1*t ) )
```

Definición de las funciones usadas en el código. La primera, $y(t)$, es la función de oscilación de la guitarra (expresión 4.5) y la segunda, $V_{fem}(t)$, es la función de potencial inducido para la bobina.

```
#CONSTANTES
#Frecuencia fundamental de la cuerda [Hz]
frequ1 = 440.
#Amplitud maxima de oscilacion de la cuerda [m]
y0 = 0.002
#Longitud de la cuerda [m]
L = 0.7
#Tiempo de vida medio de una oscilacion [s]
tau = 1.

#Tiempo maximo [s]
tmax = 5
#Intervalos
dt = 1/44100.
#Arreglo de tiempo
tiempo = np.arange( 0, tmax, dt )

#Potencial
V = Vfem( tiempo )
#Oscilacion original
y_cuerda = y( tiempo )
```

En esta parte se definen las variables físicas del problema, incluyendo la frecuencia de oscilación de la guitarra `freq1`, la amplitud de la cuerda `y0`, la longitud `L` y el tiempo de vida medio de la oscilación `tau`. Luego se definen los parámetros relacionados con el tiempo de muestreo de la señal del potencial, esto es, tiempo máximo `tmax` en segundos, paso de tiempo `dt`¹, y el arreglo de tiempo `tiempo`, el cual se crea usando la función `arange` de la librería *NumPy*. Finalmente se calcula el array con la información del potencial inducido en el tiempo como `V = Vfem (tiempo)` y la oscilación de la cuerda originalmente `y_cuerda = y(tiempo)`.

```
#Grafica
plt.plot( tiempo, V, linewidth = 0.5 )
plt.grid()
plt.title( "Potencial inducido por una bobina fonocaptora" )
plt.xlabel( "t [s]" )
plt.ylabel( "Potential [V_0]" )
plt.show()
```

Se grafica el potencial inducido vs el tiempo usando la librería *Matplotlib*.

```
#Audio producido por la bobina
nota_bobina = ad.audio()
nota_bobina.load( V*ad.Amplitude/max(V) )
nota_bobina.play()

#Audio original de la cuerda
nota_cuerda = ad.audio()
nota_cuerda.load( y_cuerda*ad.Amplitude/max(y_cuerda) )
nota_cuerda.play()
```

Finalmente se crean los audios correspondientes a la guitarra y a la bobina. Para esto se usa la clase `audio` de la librería *AudioLib* (`ad`). Primero se crea el objeto `nota_bobina`, luego, usando el método `nota_bobina.load`, se carga la señal para este audio. Para esto se normaliza la señal con el máximo valor que toma la amplitud del potencial `max(V)`², y luego se multiplica por la amplitud `ad.Amplitude` que corresponde al máximo volumen que puede reproducir un computador, este valor es estándar y está almacenado en la librería *AudioLib*. Ahora, usando el método `nota_bobina.play()` se reproduce el audio.

¹Este valor es estándar para la generación de señales de audio y no debe cambiarse.

²A pesar de que esta señal es un potencial, se puede modelar un parlante como una resistencia, de tal forma que la señal sonora es proporcional al potencial.

Se realiza el mismo procedimiento para la señal de la guitarra.

4.2. Ejercicios

Ejercicio 1 (★)

Bobina fonocaptora con armónicos

Para simular de forma realista el sonido de una guitarra se deben considerar los armónicos de orden superior, esto es, cuando se reproduce una cierta nota en una guitarra, y en general en cualquier instrumento musical, no solo se reproduce esa nota específica sino un conjunto de notas con frecuencias que son múltiplos de la frecuencia fundamental, estas notas se denominan armónicos.

Considere el mismo ejercicio de la demostración 1 pero incluya los primeros 4 armónicos. Para esto sume a la amplitud de la guitarra 4 notas sinusoidales con frecuencias múltiplos de la frecuencia fundamental, es decir

$$\omega_n = n \times \omega_1, \quad n = 1, 2, 3, 4$$

con ω_1 la frecuencia escogida (En la demostración 1 se ha tomado 440 Hz). Escoja las amplitudes relativas de cada armónico como desee, pero manteniendo siempre la frecuencia fundamental con mayor amplitud.

Al igual que la demostración 1, grafique el potencial inducido y reproduzca los audios de la bobina y de la guitarra.

Ejercicio 2 (★)

Gravímetro magnético

Una interesante aplicación del fenómeno de inducción magnética es el gravímetro magnético. Su funcionamiento se basa en la caída de un cuerpo magnetizado con un campo dipolar y al pasar por cada espira se registra un pico en la corriente del amperímetro debido a inducción magnética, tal como se muestra en la siguiente figura.

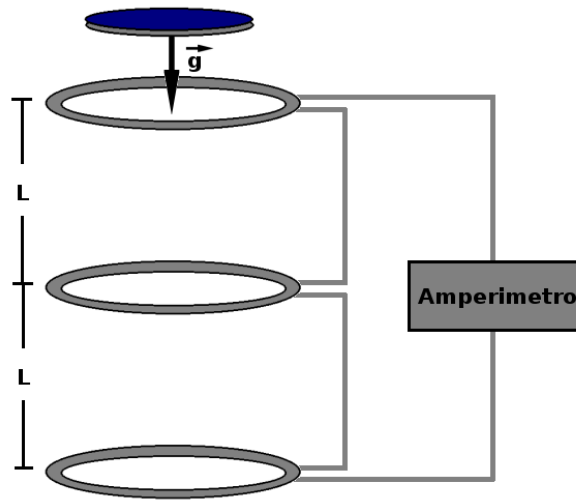


Figura 4.3: Gravímetro magnético.

Basado en esto, realice un código que simule la señal de la corriente de inducción medida en función del tiempo a medida que el cuerpo cae.

Hint: para el campo magnético del cuerpo, asuma un campo con la aproximación de la ecuación 4.2, donde la dirección u_j se toma en dirección vertical hacia arriba.

Ejercicio 3 (★)

Inducción por deslizamiento

Considere la situación de la siguiente figura

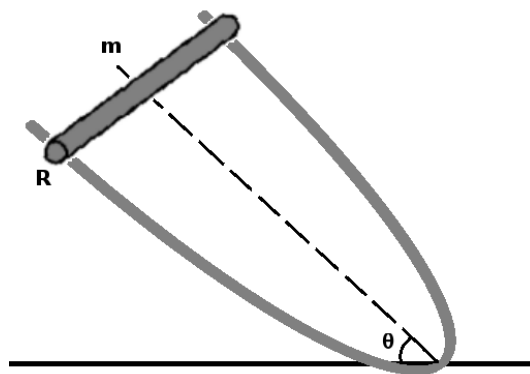


Figura 4.4: Inducción por deslizamiento.

Se tiene un cilindro macizo conductor de radio R y masa m que rueda sin deslizar desde una altura h sobre una base de forma parabólica también conductora y que está inclinada un ángulo θ respecto a la horizontal.

Realice un código que grafique el potencial inducido en función del tiempo mientras el cilindro rueda. *Hint:* asigne los valores numéricos y físicos que desee, además de los parámetros geométricos de la parábola.

Bibliografía

- [1] Purcell E. M. Electricity and Magnetism, Berkeley Physics Course Vol. 2. Mc Graw Hill. 1965.
- [2] Alonso & Finn. Física, Campos y Ondas Vol. 2. Addison-Wesley. 1998.
- [3] Sears, Zemanski, Young & Freedman, Física Universitaria Vol. 2. Pearson Addison-Wesley, 11 ed, 2004.