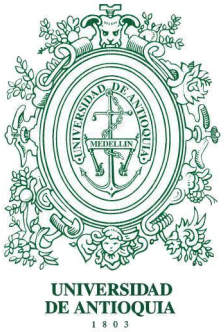
Suplemento Computacional

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

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 computaciones. La principal motivación de esto es su facilidad de implementación en comparación a otros lenguajes también de amplio 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 oficiales[1], 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
```

[1] En 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.pylab 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

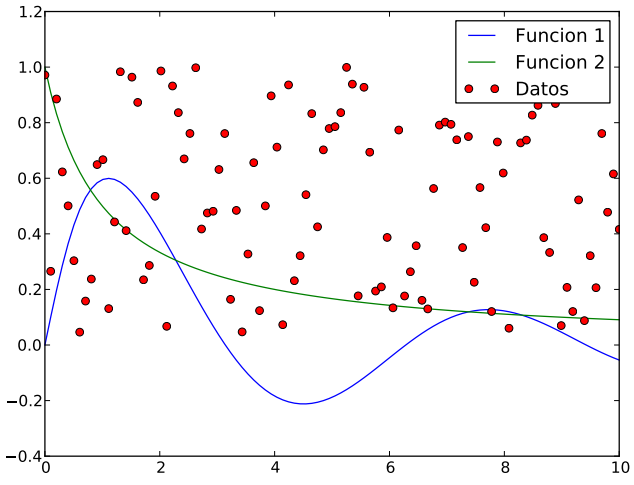


Figura 1.1: Resultado del ejemplo anterior, gráfica de dos funciones y datos aleatorios.

Para obtener el anterior script, el estudiante puede transcribirlo directamente de esta guía o puede descargarlo del repositorio oficial del curso[2] 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 $\boldsymbol{E}$, un nombre adecuado para el código puede ser `trayectory_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 _.

[2] Repositorio 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 $\boldsymbol{E}$, la ecuación de movimiento es

$$m\frac{d^2\boldsymbol{r}}{dt^2} = q\boldsymbol{E} \tag{2.1}$$

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

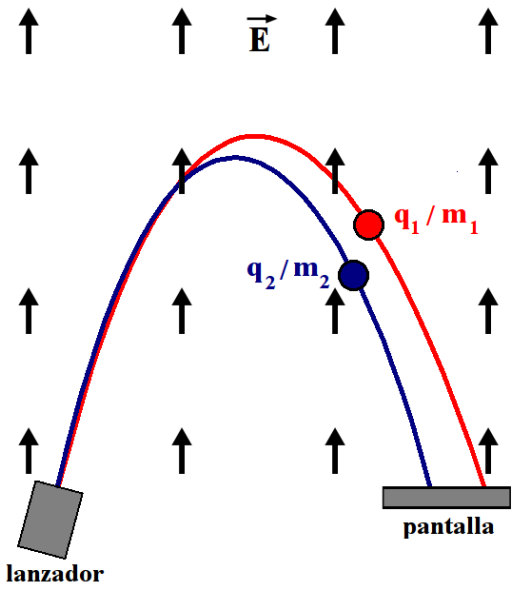


Figura 2.1: Espectrómetro de masas.

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

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

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

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

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

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

```
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 1
# Espectrometro de masas
#============================================================
from __future__ import division
import numpy as np
import matplotlib.pylab as plt

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

#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 = trayectory( X )
#Grafica de trayectoria
plt.plot( X, Y, label='particula 1' )

#PARTICULA 2
#Carga
q = -1
#Masa
m = 2
#Posicion inicial
x0 = 0
y0 = 0
#Velocidad inicial
vx0 = 1
```

```
43 vy0 = 2
44 #Valores de X a graficar
45 X = np.arange( 0, 10, 0.01 )
46 #Trayectoria
47 Y = trayectory( X )
48 #Grafica de trayectoria
49 plt.plot( X, Y, label='particula 2' )
50
51 #Limites del eje X
52 plt.xlim( (0,10) )
53 #Limites del eje Y
54 plt.ylim( (0,10) )
55 plt.legend()
56 plt.show()
```

El resultado que se obtiene es

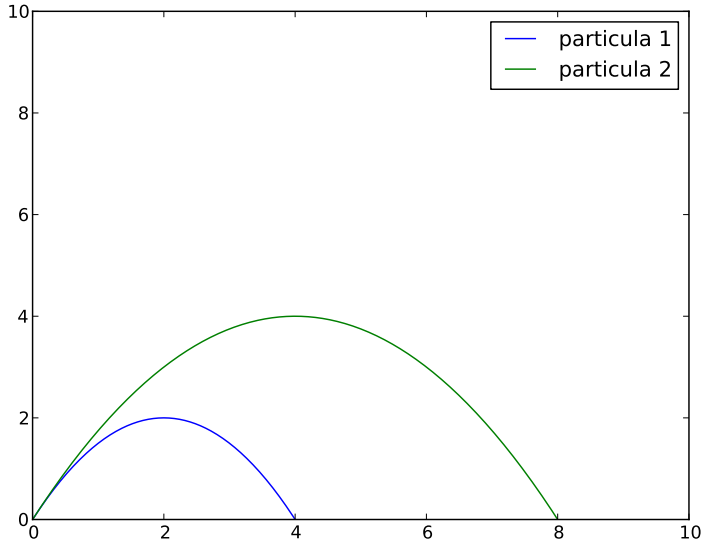


Figura 2.2: Trayectorias según la relación carga masa de las partículas.

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.pylab 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 trayectory(x):
    y = y0 + vy0/vx0*(x - x0) + 0.5*(q/m)*( (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 = trayectory( 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'`.

```
#Limites del eje X
plt.xlim( (0,10) )
#Limites 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

$$\begin{aligned} \varphi_{tot}(\boldsymbol{r}) &= \varphi_1(\boldsymbol{r}) + \varphi_2(\boldsymbol{r}) \\ &= \frac{1}{4\pi\epsilon_0}\frac{q_1}{|\boldsymbol{r}-\boldsymbol{r}_1|} + \frac{1}{4\pi\epsilon_0}\frac{q_2}{|\boldsymbol{r}-\boldsymbol{r}_2|} \end{aligned} \tag{2.5}$$

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

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

```
#!/usr/bin/env python
#============================================================
# 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.pylab as plt
9
10 #Potencial de una particula 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 particula puntual cargada
16 def Electric( r, rp, q ):
17     E = 1/(4*np.pi*eps0)*q*(r - rp)/np.linalg.norm( r - rp )
           **3
18     return E
19
20 #Permitividad del vacio
21 eps0 = 8.85418e-12
22 #Resolucion de graficas
23 Nres = 25
24 #Coordenada X
25 Xarray = np.linspace( 0, 10, Nres )
26 #Coordenada Y
27 Yarray = np.linspace( 0, 10, Nres )
28 #Construccion de la cuadricula
29 X, Y = plt.meshgrid( Xarray, Yarray )
30
31 #CONSTRUCCION DE CAMPO E Y POTENCIAL PHI, PARTICULA 1
32 #Carga electrica
33 q1 = -1
34 #Posicion particula
35 rp1 = np.array( [4,4] )
36 #Incializacion Potencial Electrico
37 phi1 = np.zeros( (Nres,Nres) )
38 #Incializacion Campo Electrico
39 E1x = np.ones( (Nres,Nres) )
40 E1y = np.ones( (Nres,Nres) )
41 #Calculo Potencial Electrico y Campo Electrico
42 for i in xrange(Nres):
43     for j in xrange(Nres):
44         r = np.array( [Xarray[i], Yarray[j]] )
45         phi1[i,j] = Phi( r, rp1, q1 )
```

```
46          E = Electric( r, rp1, q1 )
47          E1x[i,j], E1y[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 = phi1 + 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( [E1x[i,j] + E2x[i,j], \
76          E1y[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

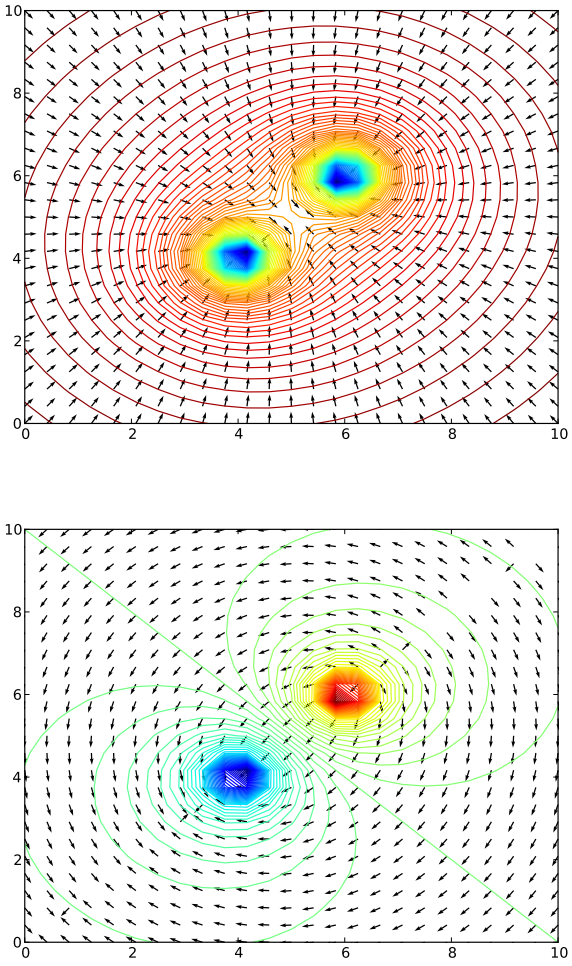


Figura 2.3: Líneas de campo y equipotenciales de dos cargas puntuales iguales. Con iguales signos (Superior) y con signos opuestos (Inferior).

A continuación se detalla cada parte del anterior script

```
#Potencial de una particula 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 particula 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 $\boldsymbol{r} - \boldsymbol{r}_p$.

```
#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 cuadricula
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 particula
rp1 = np.array( [4,4] )
#Incializacion Potencial Electrico
phi1 = np.zeros( (Nres,Nres) )
#Incializacion Campo Electrico
E1x = np.ones( (Nres,Nres) )
```

```
E1y = 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 )
        E1x[i,j], E1y[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 `q1` y su posición `rp1`. Posteriormente se inicializan las matrices donde se van a mapear los valores de los campos, primero el potencial como `phi1 = np.zeros( (Nres,Nres) )`, 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 $\boldsymbol{E}$.

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 $\boldsymbol{r}_{ij} = x_i\boldsymbol{i} + y_j\boldsymbol{j}$, en el código `r = np.array( [Xarray[i], Yarray[j]] )`. Finalmente se calcula el potencial en este punto del espacio $\boldsymbol{r}_{ij}$ como `phi1[i,j] = Phi( r, rp1, q1 )` y cada componente del campo eléctrico `E1x[i,j]` y `E1y[i,j]`. Es importante mencionar que lo único importante para definir las líneas de campo eléctrico es la dirección local de $\boldsymbol{E}(\boldsymbol{r}_{ij})$, 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( [E1x[i,j] + E2x[i,j], \
        E1y[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 `phi_tot = phi1 + phi2`. En el caso del campo eléctrico total $\boldsymbol{E}_{tot}$ se debe usar de nuevo dos ciclos **`for`** para calcular la nueva normalización del campo

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

Obteniendo así la dirección local en $\boldsymbol{r}_{ij}$ 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 $\boldsymbol{E}_{tot}$.

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

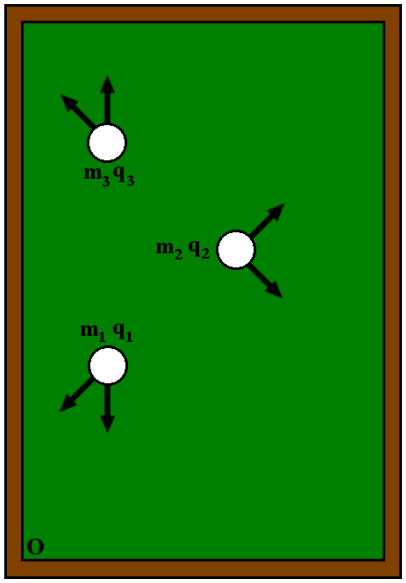


Figura 2.4: Billar electrostático con 3 bolas cargadas interactuando en dos dimensiones.

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

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 $\boldsymbol{v}_i = d\boldsymbol{r}_i/dt$, las ecuaciones de movimiento para todo el sistema pueden escribirse en coordenadas cartesianas como

$$\frac{dx_1}{dt} = v_{x,1} \tag{2.8}$$

$$\frac{dy_1}{dt} = v_{y,1} \tag{2.9}$$

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

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

$$\frac{dx_2}{dt} = v_{x,2} \tag{2.12}$$

$$\frac{dy_2}{dt} = v_{y,2} \tag{2.13}$$

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

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

$$\frac{dx_3}{dt} = v_{x,3} \tag{2.16}$$

$$\frac{dy_3}{dt} = v_{y,3} \tag{2.17}$$

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

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

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

```
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 3: Parte 1
# Solucion numerica de problema de 3 cuerpos
# electrostaticos
#============================================================
from __future__ import division
import numpy as np
import matplotlib.pylab as plt
import scipy.integrate as integ
from RungeKutta4 import rk4_step

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

```

```
24      #Posicion X particula 2
25      x2 = Y[4]
26      #Posicion Y particula 2
27      y2 = Y[5]
28      #Velocidad X particula 2
29      vx2 = Y[6]
30      #Velocidad Y particula 2
31      vy2 = Y[7]
32
33      #Posicion X particula 3
34      x3 = Y[8]
35      #Posicion Y particula 3
36      y3 = Y[9]
37      #Velocidad X particula 3
38      vx3 = Y[10]
39      #Velocidad Y particula 3
40      vy3 = Y[11]
41
42      #Modulo distancia entre particula 1 y 2
43      r12 = np.linalg.norm( [x1-x2, y1-y2] )
44      #Modulo distancia entre particula 1 y 3
45      r13 = np.linalg.norm( [x1-x3, y1-y3] )
46      #Modulo distancia entre particula 2 y 3
47      r23 = np.linalg.norm( [x2-x3, y2-y3] )
48
49      #Derivada dx/dt particula 1
50      dx1 = vx1
51      #Derivada dy/dt particula 1
52      dy1 = vy1
53      #Derivada d vx/dt particula 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 particula 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 particula 2
61      dx2 = vx2
62      #Derivada dy/dt particula 2
63      dy2 = vy2
64      #Derivada d vx/dt particula 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

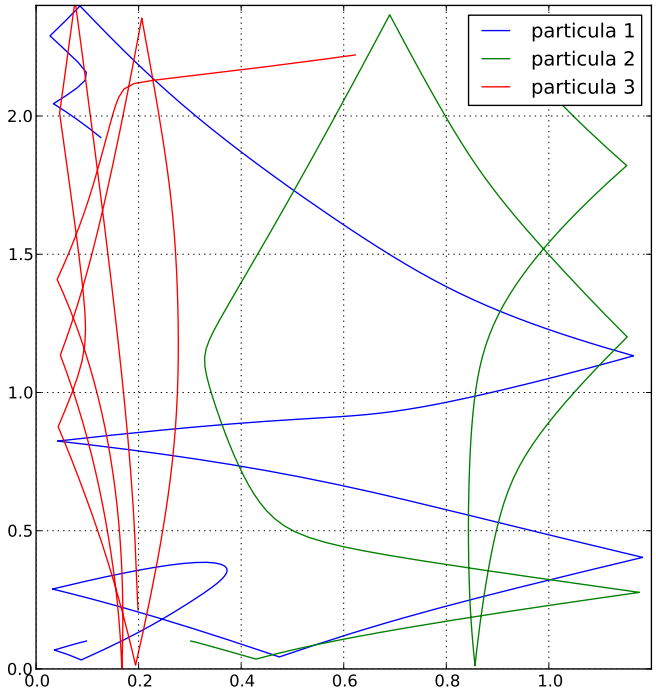


Figura 2.5: Trayectoria calculada numéricamente de las tres bolas del problema.

y un archivo de texto `trayectorias.txt` con las soluciones del sistema en el formato `[tiempo, x1, y1, x2, y2, x3, y3]`, donde `xi, yi` denontan 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.pylab 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 particula 1
    x1 = Y[0]
    #Posicion Y particula 1
    y1 = Y[1]
    #Velocidad X particula 1
    vx1 = Y[2]
    #Velocidad Y particula 1
    vy1 = Y[3]

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

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

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

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

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

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

    #Derivada dx/dt particula 3
    dx3 = vx3
    #Derivada dy/dt particula 3
    dy3 = vy3
    #Derivada d vx/dt particula 3
    dvx3 = q3/(4*np.pi*eps0*m3)*( q1/r13**3*(x3-x1) + \
    q2/r23**3*(x3-x2) )
    #Derivada d vy/dt particula 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 `r12 = np.linalg.norm( [x1-x2, y1-y2] )`, 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 = []` sonde 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 en 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 `savetxt` 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.

```
#!/usr/bin/env python
#============================================================
# DEMOSTRACION 3: Parte 2
# Solucion numerica de problema de 3 cuerpos
# electrostaticos. Animacion 3D
#============================================================
import numpy as np
import enthought.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') )

#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

#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

#Creando bola 2
bola2 = visual.sphere( radius=r2, color=(1.0, 1.0, 1.0) )
```

```
bola2.pos = [ 0., 0., 0. ]
bola2.t = 0
bola2.dt = 1

#Creando bola 1
bola3 = visual.sphere( radius=r3, color=(1.0, 1.0, 1.0) )
bola3.pos = [ 0., 0., 0. ]
bola3.t = 0
bola3.dt = 1

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

muro_l = visual.box( pos=(-grosor/2., largo/2., 0.0), \
size=(grosor, largo + 2*grosor, grosor), \
color=(0.6, 0.3, 0.0) )
muro_r = visual.box( pos=(ancho+grosor/2., largo/2., 0.0), \
size=(grosor, largo + 2*grosor, grosor), \
color=(0.6, 0.3, 0.0) )
muro_d = visual.box( pos=(ancho/2., -grosor/2., 0.0), \
size=(ancho + 2*grosor, grosor, grosor), \
color=(0.6, 0.3, 0.0) )
muro_u = visual.box( pos=(ancho/2., largo+grosor/2., 0.0), \
size=(ancho + 2*grosor, grosor, grosor), \
color=(0.6, 0.3, 0.0) )


#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 )

    #Evolucion de la bola 2
    bola2.t = bola2.t + bola2.dt
    i = bola2.t
    bola2.pos = visual.vector( x2_t[i], y2_t[i], r2 )

    #Evolucion de la bola 3
    bola3.t = bola3.t + bola3.dt
    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

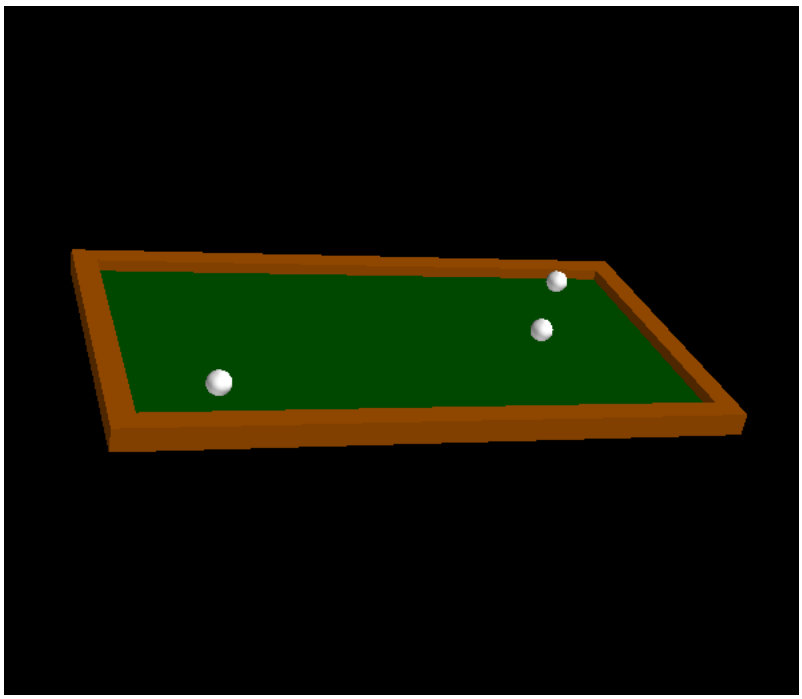


Figura 2.6: Animación 3D del sistema de billar electrostático usando *MayaVi2* .

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`.

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.