Ecuaciones diferenciales ordinarias

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

Facultad de Ciencias - UNAM

25 de marzo de 2018

Un sistema de EDOs es usualmente formulado en forma estándar antes de ser resuelto numéricamente con `python`. La forma estándar es:

$$y' = f(y, t)$$

donde

$$y = [y_1(t), y_2(t), \ldots, y_n(t)]$$

y f es una función que determina las derivadas de la función $y_i(t)$.

Usando `python` para resolver las EDO

Para resolver la EDO necesitamos conocer la función f y una condición inicial, $y(0)$.

Nótese que EDOs de orden superior siempre pueden ser escritas en esta forma introduciendo nuevas variables para las derivadas intermedias.

La función `scipy.integrate.odeint`

Dentro del módulo **`integrate`** tenemos disponible la función **`odeint`**, que integra un sistema de EDO.

$$dy/dt = \text{ func}(y, t_0, \ldots)$$

donde y puede ser un vector.

La sintaxis para `odeint`

La sintaxis básica para la función es la siguiente:

```
odeint(func, y0, t, args=())
```

Los argumentos de `odeint`

Los argumentos son los siguientes:

1. **`func`** : Función que se manda llamar $(y, t0, \ldots)$ Calcula la derivada de y en t_0.

Los argumentos de `odeint`

Los argumentos son los siguientes:

1. **`func`** : Función que se manda llamar $(y, t0, \ldots)$ Calcula la derivada de y en t_0.
2. **`y0`** : arreglo. Condición inicial de y, puede ser un vector.

Los argumentos de `odeint`

Los argumentos son los siguientes:

1. **`func`** : Función que se manda llamar $(y, t0, \ldots)$ Calcula la derivada de y en t_0.
2. **`y0`** : arreglo. Condición inicial de y, puede ser un vector.
3. **`t`** : arreglo. Una secuencia de puntos temporales en los cuales se va a resolver para la variable y. La condición inicial debe de ser el primer elemento de esta secuencia.

Los argumentos de `odeint`

Los argumentos son los siguientes:

1. **`func`** : Función que se manda llamar $(y, t0, ...)$ Calcula la derivada de y en t_0.
2. **`y0`** : arreglo. Condición inicial de y, puede ser un vector.
3. **`t`** : arreglo. Una secuencia de puntos temporales en los cuales se va a resolver para la variable y. La condición inicial debe de ser el primer elemento de esta secuencia.
4. **`args`** : tupla opcional. Argumentos extra para pasar a la función.

Lo que devuelve la función **odeint**

La función **`odeint`** devuelve una serie de elementos, el principal es:

y : arreglo, `shape (len(t), len(y0))`

Que es un arreglo que contiene los valores de y para cada punto temporal t, con la condición inicial y_0 en el primer renglón.

Usando la función `odeint`

Una vez definida la función F y el arreglo y_0, podemos usar la función **`odeint`**:

$$y_t = \texttt{odeint}(F, y_0, t)$$

Nótese que contiene el mínimo de argumentos para la función.

Ejercicio 1 - Péndulo con fricción

La EDO2 para el ángulo θ de un péndulo que se desplaza bajo la acción de la gravedad y con fricción, se puede escribir como

$$\ddot{\theta} + b\,\dot{\theta} + c\,\sin\theta = 0$$

donde b y c son constantes positivas.

Solución al ejercicio

Para resolver el problema con la función **`odeint`** debemos convertir a un sistema de EDO1.

Definiendo la velocidad angular $\omega(t) = \dot{\theta}$, se obtiene el sistema

$$\begin{aligned} \dot{\theta} &= \omega \\ \dot{\omega} &= -b\,\omega - c\,\sin(\theta) \end{aligned}$$

Solución al ejercicio

Sea el vector $y = [\theta, \omega]$. Así la función que usaremos en `python` queda como

Código 1: Función a integrar

```python
def F(y, t, b, c):
    theta, omega = y
    dydt = [omega, -b * omega - c * np.sin(theta)]
    return dydt
```

Valores de las constantes

Consideramos que las constantes b y c son:

$$b = 0.25$$
$$c = 5.0$$

Condiciones inciales

Para las condiciones iniciales, supongamos que el péndulo está muy cerca de la vertical con $\theta(0) = \pi - 0.1$, y que está en reposo, por lo que $\omega(0) = 0$.

Entonces, el vector de las condiciones iniciales queda como

$$y0 = [\pi - 0.1,\ 0.0]$$

Secuencia temporal

Generamos una secuencia de 101 puntos temporales en el intervalo $0 \leq t \leq 10$, por lo que nuestro arreglo de tiempo es:

$$t = \texttt{np.linspace}(0., \ 10., \ 101)$$

Solución con `odeint`

Usamos la función **`odeint`** para la solución; el paso de los parámetros b y c a **`F`**, se hace a través de **`args`**:

Código 2: Solución con odeint

```
from scipy.integrate import odeint

sol = odeint(F, y0, t, args=(b, c))
```

Código completo I

Código 3: Función a integrar

```
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import numpy as np

def F(y, t, b, c):
    theta, omega = y
    dydt = [omega, -b * omega - c *
  np.sin(theta)]
    return dydt

b = 0.25
c = 5.0
```

Código completo II

```

y0 =  [np.pi - 0.1, 0.0]

t = np.linspace(0., 10., 101)

sol = odeint(F, y0, t, args=(b, c))

plt.figure(1)
plt.plot(t, sol[:,0], 'b', label='
   theta(t)')
plt.plot(t, sol[:,1], 'g', label='
   omega(t)')
plt.xlabel('t')
```

Código completo III

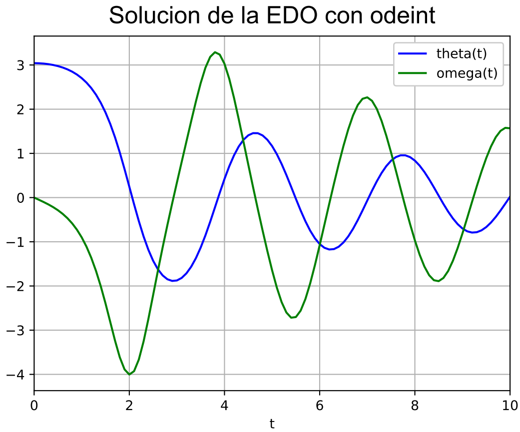
```
plt.xlim([0, 10])
plt.title('Solucion de la EDO con odeint')
plt.legend(loc='best')
plt.grid()
plt.show()

plt.figure(2)
plt.plot(sol[:,0], sol[:,1])
plt.axhline(y = 0, ls='dashed', lw = 0.7, color = 'k')
plt.axvline(x = 0, ls='dashed', lw = 0.7, color = 'k')
```

Código completo IV

```
plt.title('Diagrama fase del pendulo')
plt.show()
```

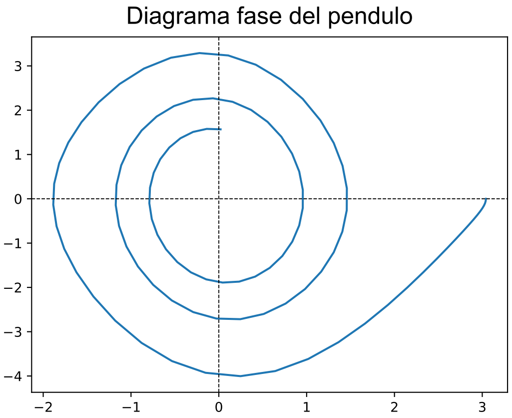
Gráficas de la solución

La primera gráfica representa la posición y la velocidad angular del péndulo.



Gráfica del espacio fase

La segunda gráfica representa el espacio fase, y como podemos ver, hay un atractor debido a la fricción en el péndulo.



Ejercicio 2 - Oscilador amortiguado

La ecuación de movimiento para el oscilador amortiguado es:

$$\frac{d^2x}{dt^2} + 2\,\zeta\,\omega_0 \frac{dx}{dt} + \omega_0^2\,x = 0$$

donde x es la posición del oscilador, ω_0 la frecuencia, y ζ es el factor de amortiguamiento.

Re-escribiendo la EDO

Para escribir esta EDO2 en la forma estándar, introducimos $p = dx/dt$

$$\begin{aligned} \frac{dp}{dt} &= -2\,\zeta\,\omega_0\,p - \omega^2\,x \\ \frac{dx}{dt} &= p \end{aligned}$$

Usando argumentos

Veremos con este ejemplo, la versatilidad de pasar argumentos extras a la función, que representan diferentes valores del factor de amortiguamiento.

Usando argumentos

De tal manera que en una sola ejecución del código, podemos realizar el pase de valores, de otra manera, tendríamos que realizar una ejecución del código y modificar a mano el valor del factor de amortiguamiento.

Usando argumentos

Como consecuencia de los argumentos extra, necesitamos pasar un argumento clave `args` a la función `odeint`.

$\zeta = 0.0, 0.2, 1.0, 5.0$

Código para resolver el problema I

Código 4: Código completo

```
from scipy.integrate import odeint
from numpy import zeros, array, linspace
import matplotlib.pyplot as plt

def F(y, t, zeta, w0):
    F = zeros((2), dtype='float64')
    F[0] = y[1]
    F[1] = -2 * zeta * w0 * y[1] - w0**2 * y[0]
    return F

```

Código para resolver el problema II

```
y0 =array([1.0, 0.0])

t = linspace(0, 10, 1000)
w0 = 2 * pi * 1.0


y1 = odeint(F, y0, t, args=(0.0, w0))
y2 = odeint(F, y0, t, args=(0.2, w0))
y3 = odeint(F, y0, t, args=(1.0, w0))
```

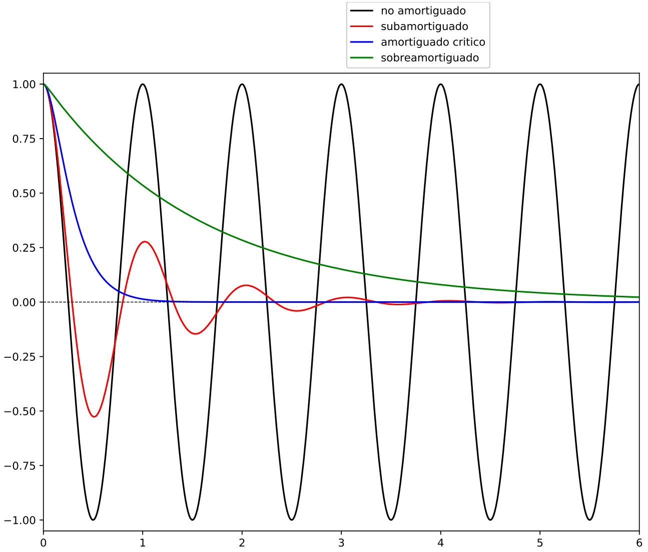
Código para resolver el problema III

```python
y4 = odeint(F, y0, t, args=(5.0, w0))

plt.axis([0, 10, -1.05, 1.05])
plt.plot(t, y1[:,0], ’k’, label="no amortiguado")
plt.plot(t, y2[:,0], ’r’, label="subamortiguado")
plt.plot(t, y3[:,0], ’b’, label="amortiguado critico")
plt.plot(t, y4[:,0], ’g’, label="sobreamortiguado")
plt.legend()
```

Código para resolver el problema IV

```
plt.show()
```

Resultado

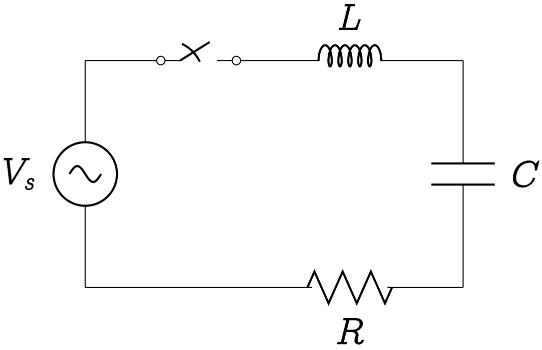


no amortiguado
subamortiguado
amortiguado critico
sobreamortiguado
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6

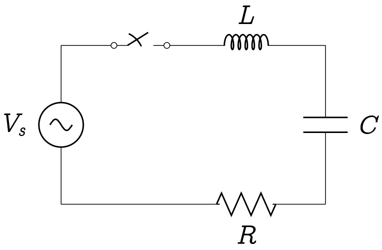
Ejercicio 3 - Circuito RLC

La corriente eléctrica de un circuito RLC en serie, satisface la ecuación

$$L\frac{di}{dt}+Ri+\frac{1}{C}\int_0^t i(t')dt'+\frac{1}{C}q(0)=E(t), \quad t>0 \quad (1)$$

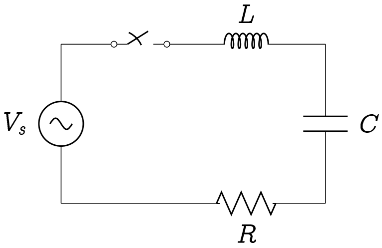


Circuito RLC



Cuando el circuito se cierra en el instante $t = 0$, se tiene que $i = i(t)$ es la corriente, R es la resistencia, L, C, E vienen dadas por: $L = 200\,\text{H}$, $C = 0.001\,\text{F}$, $E(t) = 1\,\text{V}$ para $t > 0$.

Circuito RLC



Las condiciones iniciales son $q(0) = 0$ (carga inicial del condensador), $i(0) = 0$.

Resolver el problema

Calcular la corriente para $0 \leq t \leq 5$ segundos y el factor de amortiguamiento y la frecuencia de oscilación del circuito RLC para los siguientes valores de R:

1. $R = 0\,\Omega$
2. $R = 50\,\Omega$
3. $R = 100\,\Omega$
4. $R = 300\,\Omega$

Solución

Si definimos

$$q(t) = \int_0^{t'} i(t')\,dt' \tag{2}$$

derivando la expresión anterior

$$\frac{d}{dt}q(t) = i(t), \qquad q(0) = 0 \tag{3}$$

Solución

Sustituimos en la ecuación inicial, para re-escribir

$$\frac{d}{dt} i(t) = -\frac{R}{L} i(t) - \frac{1}{L\,C} q(t) + \frac{1}{L\,C} q(0) + \frac{E(t)}{L}, \; i(0) = 0 \quad (4)$$

La ecuación (1) se transformó en un sistema de dos EDO de primer orden: las ecuaciones (3) y (4).

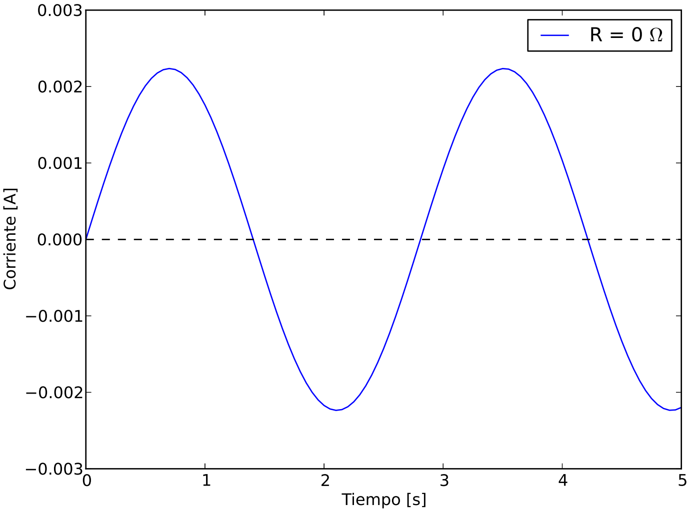
Código 5: Código para el circuito

```
from numpy import zeros, array, linspace
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def F(y, t, R, L) :
    C = 0.001
    E = 1.0

    F = zeros(2)
    F[0] = y[1]
    F[1] = -(R/L) * y[1] - (1.0/(L * C)) * y[0] + E/L
    return F
```

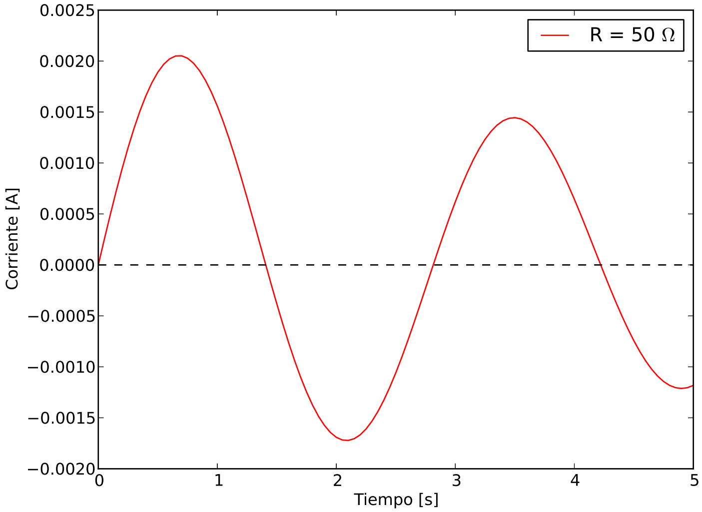
```
t = linspace(0.0, 5.0)
y0 = array([0., 0.])
h = 0.01

L = 200.0
R = [0., 50, 100, 500]

for r in R:
    sol = odeint(F, y0, t, args=(r, L))
    plt.plot(t,sol[:,1], label='R = ' + str(r) + ' $\Omega$')

plt.axhline(y=0, lw=0.75, ls='dashed', color='k')
```

```
plt.legend(loc='best')
plt.title('Solucion de la EDO')
plt.xlim([0,5])
plt.show()
```

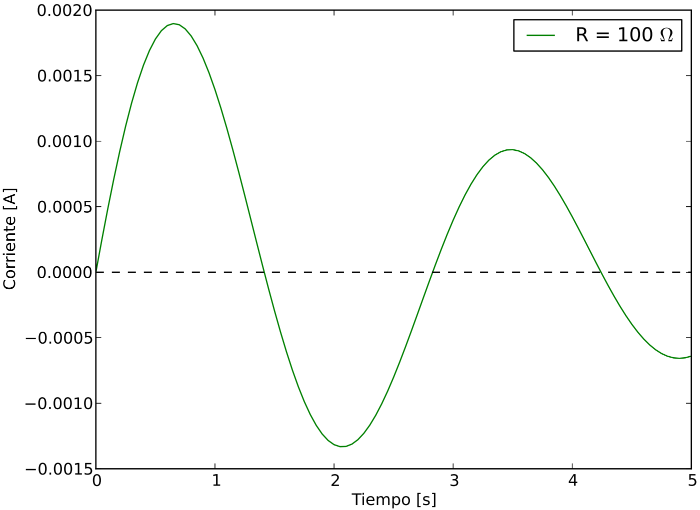
Solución gráfica con $R = 0\,\Omega$



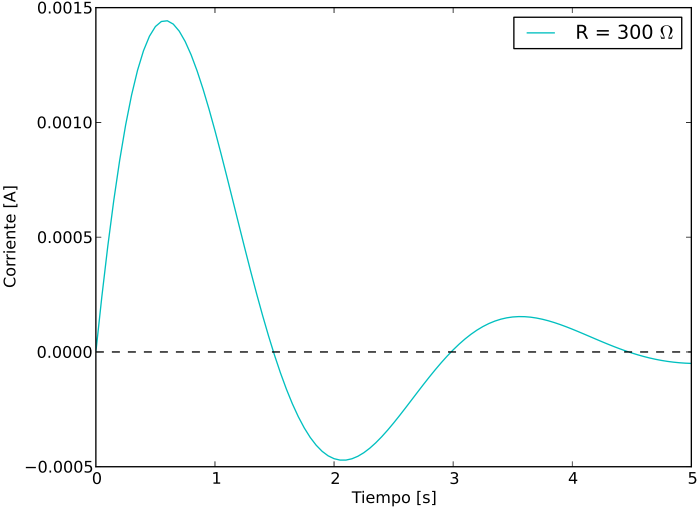
Solución gráfica con $R = 50\ \Omega$



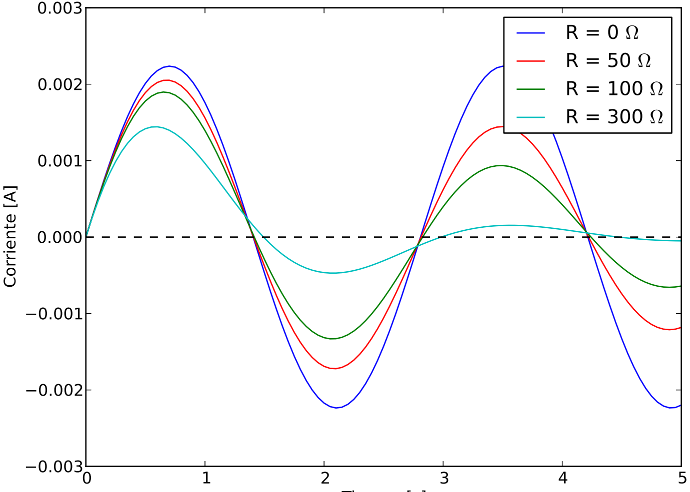
Solución gráfica con $R = 100\ \Omega$



Solución gráfica con $R = 300\ \Omega$

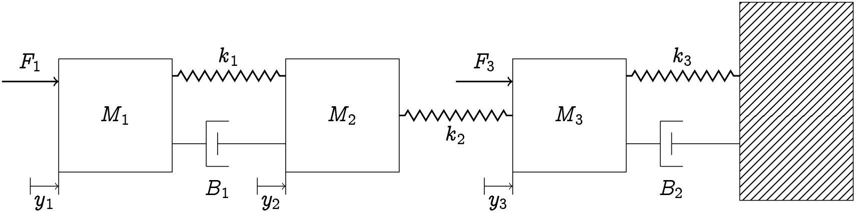


Solución gráfica con valores de R superpuestos



Ejercicio 4 - Masas acopladas

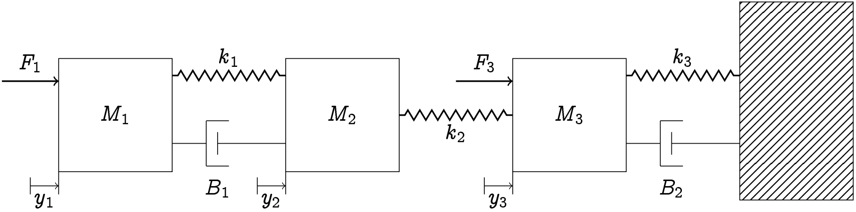
En la figura se muestra un sistema de tres masas acopladas mediante resortes y amortiguadores.



Ejercicio 4 - Masas acopladas

Los desplazamientos de estas tres masas satisfacen las ecuaciones dadas por:

$$\begin{aligned} M_1\, y''1 + B_1\, y_1' + K_1\, y_1 - B_1\, y_2' - K_2\, y_2 &= F_1(t) \\ -B_1\, y_1' - K_1\, y_1 + M_2\, y_2'' + B_1\, y_2' + (K_1 + K_2)\, y_2 - K_2\, y_3 &= 0 \\ -K_2\, y_2 + M_3\, y_3'' + B_2\, y_3' + (K_2 + K_3)\, y_3 &= F_3(t) \end{aligned}$$



Condiciones iniciales

Las constantes y condiciones iniciales son

$K_1 = K_2 = K_3 = 1$ (constantes de los resortes, kgm/s^2)
$M_1 = M_2 = M_3 = 1$ (masa, kg)
$F_1(t) = 1, F_3(t) = 0$ (fuerza, N)
$B_1 = B_2 = 0.1$ (coeficientes de amortiguamiento, kg

$y_1(0) = y_1'(0) = y_2(0) = y_2'(0) = y_3(0) = y_3'(0) = 0$
(condiciones iniciales)

Problema a resolver

Resuelve el sistema para determinar la posición y_i de cada masa en $0 \leq t \leq 60$ segundos, elabora una gráfica con las tres trayectorias.

Antes de proponer un código

Necesitamos manejar el sistema de 3 EDO2, de tal manera que podamos representar un sistema de 6 EDO1, entonces consideremos el siguiente:

Hint: Definiendo

$$y_4 = y_1', \qquad y_5 = y_2', \qquad y_6 = y_3'$$

Antes de proponer un código

Así, la ecuación inicial se escribe como un conjunto de seis EDO de primer orden, de la siguiente manera:

$$
\begin{aligned}
y_1' &= y_4 \\
y_2' &= y_5 \\
y_3' &= y_6 \\
y_4' &= \left[-B_1\, y_4 - K_1\, y_1 + B_1\, y_5 + K_2\, y_2 + F_1 \right] / M_1 \\
y_5' &= \left[B_1\, y_4 + K_1\, y_1 - B_1\, y_5 - \left(K_1 + K_2 \right)\, y_2 + K_2\, y_3 \right] / M_2 \\
y_6' &= \left[K_2\, y_2 - B_2\, y_6 - \left(K_2 + K_3 \right)\, y_3 + F_3 \right] / M_3
\end{aligned}
$$

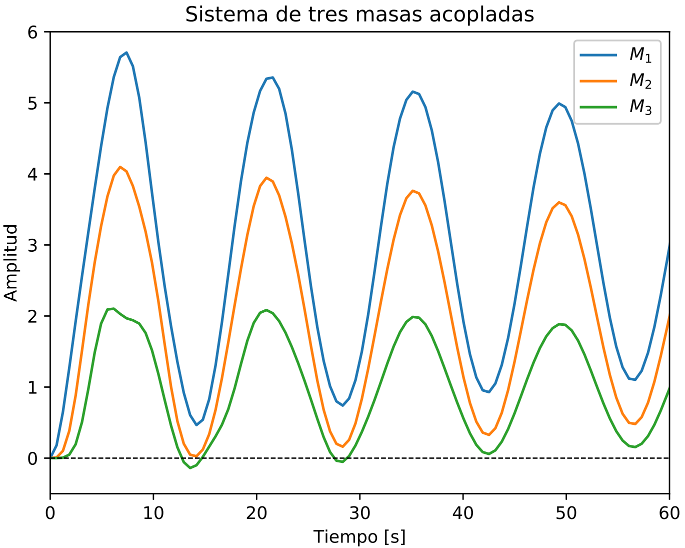
Código 6: Código para el sistema de masas

```
def F(y, t) :
    F = zeros(6)
    F[0] = y[3]
    F[1] = y[4]
    F[2] = y[5]
    F[3] = (-0.1 * y[3]- y[0] + 0.1
  * y[4] + y[1] + 1.)
    F[4] = (0.1 * y[3] + y[0] - 0.1
  * y[4] - 2. * y[1] + y[2])
    F[5] = (y[1] - 0.1 * y[5] - 2. *
  y[2])
    return F

t = linspace(0, 61, 100)
```

```
y0 = array([0., 0., 0., 0., 0., 0.])

sol = odeint(F, y0, t)

plt.plot(t, sol[:,0], label = "$M{1}$")
plt.plot(t, sol[:,1], label = "$M{2}$")
plt.plot(t, sol[:,2], label = "$M{3}$")
plt.legend(loc='upper right')
plt.xlabel("Tiempo [s]")
plt.ylabel("Amplitud")
plt.axis([0, 60, -0.5, 6])
plt.axhline(y=0, lw=0.75, ls='dashed', color='k')
```

```
plt.title("Sistema de tres masas")
plt.show()
```

Solución al problema



Sistema Lotka-Volterra

Las ecuaciones de Lotka-Volterra, también son conocidas como ecuaciones de depredador-presa.

Está descrito por un sistema de 2 ecuaciones diferenciales no lineales de primer orden.

Sistema Lotka-Volterra

Se utiliza frecuentemente para describir la dinámica de sistemas biológicos donde interaccionan dos especies: un depredador y una de sus presas.

Sistema de ecuaciones acopladas

El sistema evoluciona de acuerdo al par de ecuaciones:

$$\frac{du}{dt} = a\,u - b\,u\,v$$
$$\frac{dv}{dt} = -c\,v + d\,b\,u\,v$$

donde:

1. u es el número de presas (ej. Conejos)

Sistema de ecuaciones acopladas

El sistema evoluciona de acuerdo al par de ecuaciones:

$$\frac{du}{dt} = a\,u - b\,u\,v$$

$$\frac{dv}{dt} = -c\,v + d\,b\,u\,v$$

donde:

1. u es el número de presas (ej. Conejos)
2. v es el número de depredadores (ej. Zorros)

Sistema de ecuaciones acopladas

$$\frac{du}{dt} = a\,u - b\,u\,v$$
$$\frac{dv}{dt} = -c\,v + d\,b\,u\,v$$

donde:

3. a es la tasa natural de crecimiento de conejos, sin que haya zorros.

Sistema de ecuaciones acopladas

$$\frac{du}{dt} = a\,u - b\,u\,v$$
$$\frac{dv}{dt} = -c\,v + d\,b\,u\,v$$

donde:

5. a es la tasa natural de crecimiento de conejos, sin que haya zorros.
6. b es la tasa natural de la muerte de conejos, debido a la depredación.

Sistema de ecuaciones acopladas

$$\frac{du}{dt} = a\,u - b\,u\,v$$
$$\frac{dv}{dt} = -c\,v + d\,b\,u\,v$$

donde:

7. c es la tasa natural de la muerte del zorro, cuando no hay conejos.

Sistema de ecuaciones acopladas

$$\begin{aligned} \frac{du}{dt} &= a\,u - b\,u\,v \\ \frac{dv}{dt} &= -c\,v + d\,b\,u\,v \end{aligned}$$

donde:

7. c es la tasa natural de la muerte del zorro, cuando no hay conejos.
8. d es el factor que describe el número de conejos capturados.

Poblaciones iniciales

Vamos a utilizar $X = [u, v]$ para describir el estado de las poblaciones.

Definiendo las ecuaciones

Código 7: Código inicial

```
from numpy import *
import matplotlib.pyplot as plt

a = 1.
b = 0.1
c = 1.5
d = 0.75

def dXdt(X, t = 0):
    return array([ a * X[0] - b * X[0] * X[1], -c * X[1] + d * b * X[0] * X[1] ])
```

Población en equilibrio

Antes de usar **`odeint`** para integrar el sistema, veremos de cerca la posición de equilibrio.

El equilibrio ocurre cuando la tasa de crecimiento es igual a 0, lo que nos da dos puntos fijos:

Código 8: Equilibrio en las poblaciones

```
Xf0 = array([0., 0.])
Xf1 = array([ c/(d * b), a/b])
all(dXdt(Xf0) == zeros(2) ) and all(
   dXdt(Xf1) == zeros(2))
```

Variable temporal y cond. iniciales

Para usar la función `odeint`, hay que definir el parámetro de tiempo t, así como las condiciones inciales de la población: 10 conejos y 5 zorros.

Variable temporal y cond. iniciales

Código 9: Código para las condiciones iniciales

```
t = linspace(0, 15,  1000)

X0 = array([10, 5])

X = integrate.odeint(dXdt, X0, t)
```

Graficando la solución

Una vez obtenido el código para la solución del problema, ahora nos corresponde graficar el conjunto de datos obtenido:

Graficando la solución I

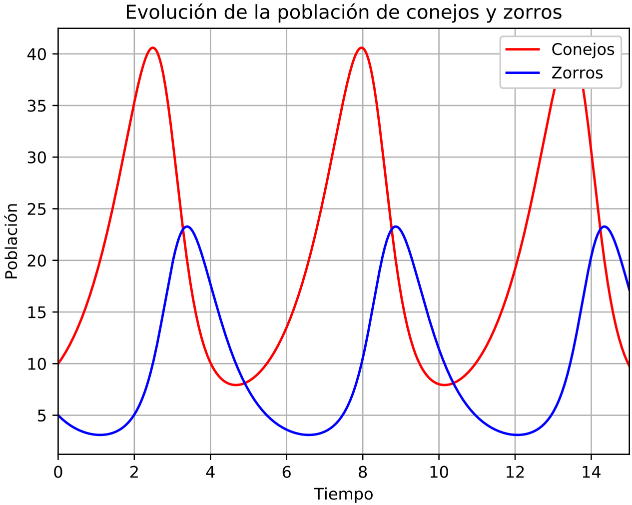
Código 10: Código para graficar

```
conejos, zorros = X.T

f1 = plt.figure()
plt.plot(t, conejos, 'r-', label='Conejos')
plt.plot(t, zorros  , 'b-', label='Zorros')
plt.grid()
plt.legend(loc='best')
plt.xlabel('tiempo')
plt.ylabel('poblacion')
```

Graficando la solución II

```
plt.title('Evolucion de la poblacion de conejos y zorros')
plt.show()
```

Resultado gráfico



Primer resultado

La gráfica anterior nos da la información sobre el número tanto de conejos como de zorros durante el intervalo de tiempo estudiado, es decir, tenemos una especie de “censo”.

Para ver la dinámica de las poblaciones propiamente, ahora representamos el espacio fase del sistema, por lo que tenemos que hacer algunos ajustes en el código que usamos anteriormente.

Condiciones de equilibrio

Consideremos las condiciones de equilibrio, es decir, donde la tasa de crecimiento es cero:

Código 11: Condiciones de equilibrio

```
Xf0 = array([0. , 0.])
Xf1 = array([ c/(d * b), a/b])
```

Elementos adicionales para la gráfica

Dibujaremos el espacio fase con algunos elementos visuales con el fin de decoración nada más.

Código 12: Obteniendo los colores

```
values  = linspace(0.3, 0.9, 5)

vcolors = plt.cm.autumn
   r(linspace(0.3, 1.,
   len(values)))f2 = plt.figure()
```

El módulo `color map`

El módulo **`cm`** proporciona un conjunto de mapas de colores predeterminados, así como las funciones necesarias para crear nuevos mapas de color.

El módulo `color map`

Existen varios mapas ya definidos: **`autumn`**, **`bone`**, **`cool`**, **`copper`**, **`flag`**, **`gray`**, **`hot`**, **`hsv`**, **`jet`**, **`pink`**, **`prism`**, **`spring`**, **`summer`**, **`winter`**, **`spectral`**.

Curvas de nivel

Se van a dibujar ahora las trayectorias para diferentes condiciones iniciales (número de conejos y zorros)

Código 13: Graficando las curvas de nivel

```
for v, col in zip(values, vcolors):
    X0 = v * Xf1

    X = integrate.odeint( dXdt, X0, t)

    plt.plot( X[:,0], X[:,1], lw=3.5 * v, color=col, label='X0=(%.f, %.f)' % ( X0[0], X0[1]) )
```

La función `zip`

La función **`zip`** sirve para reorganizar las listas en `python`.

Como parámetros admite un conjunto de listas.

La función `zip`

Lo que realmente hace es tomar el elemento i-ésimo elemento de cada lista y los une en una tupla, después une todas las tuplas en una lista.

En cada gráfica se modificará el grosor de la línea y el color que se le asocia.

Definición de una malla

Se define una malla sobre nuestro espacio de solución:

Código 14: Creando una malla

```
ymax = plt.ylim(ymin = 0)[1]
xmax = plt.xlim(xmin = 0)[1]

nbpoints    = 20

x = linspace(0, xmax, nbpoints)
y = linspace(0, ymax, nbpoints)

X1, Y1  = meshgrid(x, y)
```

¿Qué hace `meshgrid`?

La función **`meshgrid`** genera un arreglo n-dimensional para evaluaciones vectoriales de campos n-dimensionales ya sea escalares o vectoriales, a partir de arreglos unidimensionales $x_1, x_2, \ldots, x_n$.

Calculando la magnitud del vector

Código 15: Magnitud del vector y su dirección

```
DX1, DY1 = dXdt([X1, Y1])

M = (hypot(DX1, DY1))

M[ M == 0] = 1.

DX1 /= M
DY1 /= M
```

Calculando la magnitud del vector

1. Con `X1` y `Y1` se crea una malla.

Calculando la magnitud del vector

1. Con `X1` y `Y1` se crea una malla.
2. Con `DX1` y `DY1` se calcula el crecimiento de las poblaciones en la malla.

Calculando la magnitud del vector

1. Con `X1` y `Y1` se crea una malla.
2. Con `DX1` y `DY1` se calcula el crecimiento de las poblaciones en la malla.
3. Con la variable `M` se calcula la norma de la tasa de crecimiento, usando la función **`hypot`**.

Calculando la magnitud del vector

1. Con `X1` y `Y1` se crea una malla.
2. Con `DX1` y `DY1` se calcula el crecimiento de las poblaciones en la malla.
3. Con la variable `M` se calcula la norma de la tasa de crecimiento, usando la función **`hypot`**.
4. La expresión `M[M==0]  =1.` evita que tengamos una división entre cero.

Calculando la magnitud del vector

1. Con `X1` y `Y1` se crea una malla.
2. Con `DX1` y `DY1` se calcula el crecimiento de las poblaciones en la malla.
3. Con la variable `M` se calcula la norma de la tasa de crecimiento, usando la función **`hypot`**.
4. La expresión `M[M==0] =1.` evita que tengamos una división entre cero.
5. Con la operación `DX1/M` y `DY1/M` se normaliza cada vector.

Dibujando las direcciones del vector I

Se dibujan las direcciones usando **`quiver`**

Código 16: Dibujando las direcciones del vector resultante

```
plt.title('Trayectorias y campo de direccion')

Q = plt.quiver(X1, Y1, DX1, DY1, M, pivot='mid', cmap=plt.cm.jet)

plt.xlabel('Numero de conejos')
plt.ylabel('Numero de zorros')
plt.legend()
plt.grid()
```

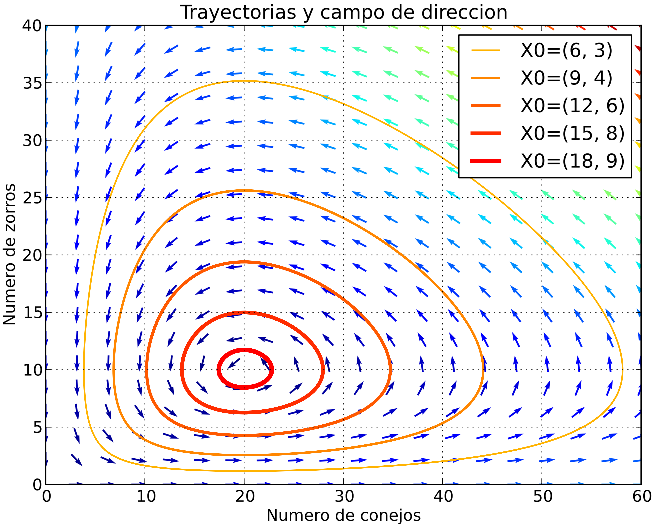
Dibujando las direcciones del vector II

```
plt.xlim(0, xmax)
plt.ylim(0, ymax)
plt.show()
```

La función `quiver`

La función **`quiver`** genera el mapa vectorial, requiere de cinco argumentos: las posiciones $X1$,$Y1$ de inicio, el valor de las componentes del vector $DX1$, $DY1$ y el color asociado, el argumento `pivot` indica en qué parte de la malla se va a colocar el vector.

Resultado gráfico



Ejercicio para resolver

El modelo de Lorenz se usa para estudiar la formación de torbellinos en la atmósfera, aunque abordó el problema de manera general, estableció las bases para el estudio de sistemas dinámicos.

Ejercicio para resolver

El conjunto de ecuaciones está dado por

$$
\begin{aligned}
\frac{dy_1}{dt} &= a(y_2 - y_1) \\
\frac{dy_2}{dt} &= (b - y_3)\, y_1 - y_2 \\
\frac{dy_3}{dt} &= y_1\, y_2 - c\, y_3
\end{aligned}
$$

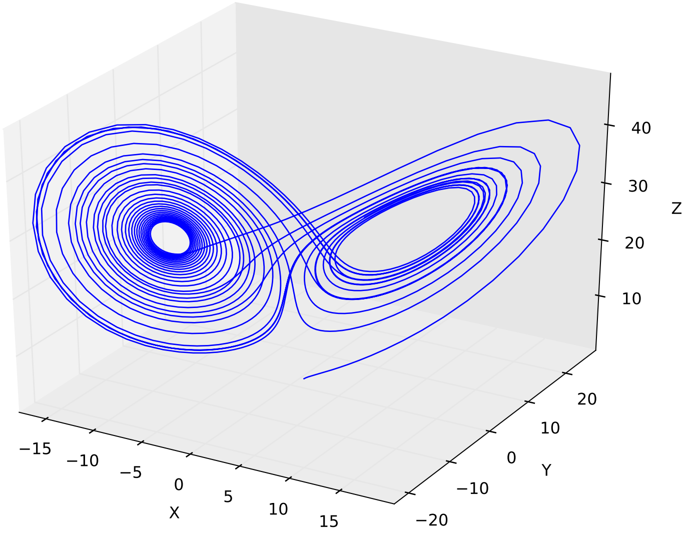
en el modelo a, b y c son parámetros positivos.

Ejercicio para resolver

Resuelve este modelo numéricamente y grafica la solución.

Utiliza los siguientes valores $a = 10$, $b = 28$ y $c = 8/3$. Interpreta la solución.

Resultado gráfico



Para generar una gráfica 3D

Para graficar una función de tres variables en **`matplotlib`**, debemos de utilizar una combinación de dos librerías:

```
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
```

Usando `plot3D`

Hay un importante cambio cuando graficamos tres variables con **`matplotlib`**, hay adecuar el espacio de trabajo mediante la siguiente referencia:

```
fig = plt.figure()

ax = p3.Axes3D(fig)
```

Con `plt` definimos el espacio común de graficación, pero con `ax`, ahora y contamos con la manera de usar la graficación de tres variables.

La función plot3D

La función **plot3D** ocupa los argumentos de la misma manera que **plot**, por lo que debemos de usar la sintaxis:

```
ax.plot3D(x, y, z)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

fig.add_axes(ax)
plt.show()
```