Python para ingenieros

ECUACIONES DIFERENCIALES ORDINARIAS

¿Te acuerdas de todos esos esquemas numéricos para integrar ecuaciones diferenciales ordinarias? Es bueno saber que existen, qué peculiaridades tienen y cuando es adecuado usar cada uno, pero en este taller no queremos implementar esos esquemas: queremos hacer uso de ellos para resolver las ecuaciones. Los problemas con condiciones iniciales están por todas partes en ingeniería y son de los más divertidos de programar.

In [1]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

Para integrar EDOs vamos a usar la función `odeint` del paquete `integrate` , de la librería `scipy` que permite integrar sistemas del tipo:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}\left(\mathbf{y}, t\right)$$

con condiciones iniciales $\mathbf{y}(\mathbf{0}) = \mathbf{y_0}$.

> **¡Importante!**: La función del sistema recibe como primer argumento $\mathbf{y}$ (un array) y como segundo argumento el instante t (un escalar).

In [2]:

```
from scipy.integrate import odeint
```

Vamos a integrar primero una EDO elemental, cuya solución ya conocemos:

$$y' + y = 0$$

$$f(y, t) = \frac{dy}{dt} = -y$$

Con condición inicial $y(0) = 1$

Tiene como **solución exacta** $y(t) = e^{-t}$

In [3]:

```
def f(y, t):
    return -y
```

Condiciones iniciales:

In [4]:

```
y0 = 1
```

Vector de tiempos donde realizamos la integración:

In [5]:

```
t = np.linspace(0, 3)
```

Integramos y representamos la `solución` :

In [6]:

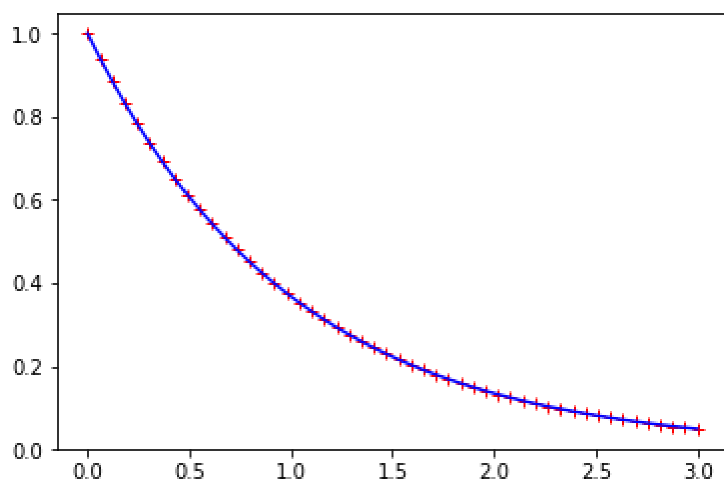
```
sol = odeint(f, y0, t)
```

In [7]:

```
# grafica la solución aproximada mediante el símbolo '+' en color rojo
plt.plot(t, sol,'+',color='r')

#grafica la solución exacta mediante línea continua en color azúl
yt=np.e**(-t)
plt.plot(t,yt,color='b')
```

Out[7]:

```
[<matplotlib.lines.Line2D at 0x1cb9a0a1780>]
```



[`scipy.integrate.odeint` (https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.integrate.odeint.html#scipy.integrate.odeint)](https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.integrate.odeint.html#scipy.integrate.odeint) es una interfaz de alto nivel para integrar ecuaciones diferenciales. Para obtener más control sobre el proceso, como especificar un solver diferente, controlar sus opciones (como precisión o el paso) se debe utilizar [`scipy.integrate.ode` (https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.integrate.ode.html#hnw93)](https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.integrate.ode.html#hnw93). Entre otros

solvers, se puede enconrar el `dopri5` , un Runge-Kutta explícito, que además cuanta con "dense output" lo que quiere decir que calcula, aunque no los devuelva por defecto, los pasos intermedios que crea necesarios en función de los parámetros del método y de los gradientes que encuentre.

EDOs de orden superior

Ya que para utilizar las funciones de SciPy, tenemos que formular el problema como un una función que devuleva el valor de la derivada (forma normal). En caso de tener una EDO de orden mayor que uno, tendremos que acordarnos ahora de cómo reducir las ecuaciones de orden. De nuevo, vamos a probar con un ejemplo académico:

$$y + y'' = 0$$

$$\mathbf{y} \leftarrow \begin{pmatrix} y \\ y' \end{pmatrix}$$

$$\mathbf{f}(\mathbf{y}) = \frac{d\mathbf{y}}{dt} = \begin{pmatrix} y \\ y' \end{pmatrix}' = \begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} y' \\ -y \end{pmatrix}$$

Con condición inicial $y(0) = 1$ y $y'(0) = 0$

In [8]:

```
# Definición de la función
def f(y, t):
    return np.array([y[1], -y[0]])
```

In [9]:

```
# Tiempo y condición inicial
t = np.linspace(0, 10)
y0 = np.array([1.0, 0.0])
```

In [10]:

```
# Solución
sol = odeint(f, y0, t)
sol
```

Out[10]:

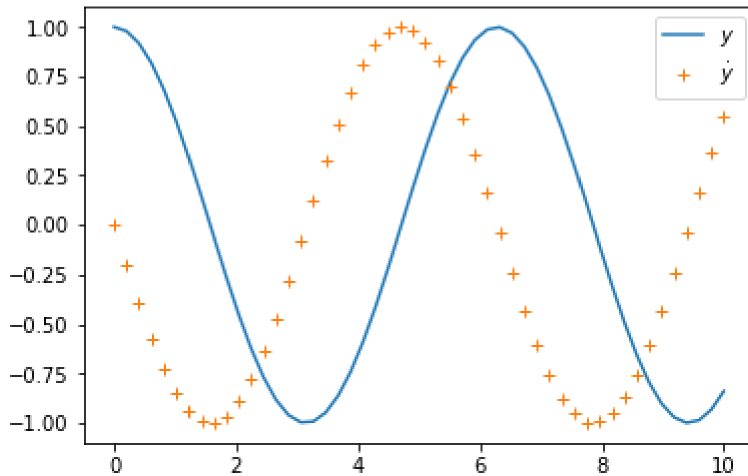
```
array([[ 1.        ,  0.        ],
       [ 0.97924752, -0.20266792],
       [ 0.91785142, -0.39692413],
       [ 0.81835993, -0.57470602],
       [ 0.68490246, -0.72863476],
       [ 0.52301813, -0.85232155],
       [ 0.33942597, -0.94063276],
       [ 0.14174595, -0.98990306],
       [-0.06181722, -0.99808747],
       [-0.26281468, -0.96484631],
       [-0.45290402, -0.89155926],
       [-0.6241956 , -0.78126807],
       [-0.76957997, -0.6385504 ],
       [-0.88302297, -0.46932972],
       [-0.95981615, -0.28062952],
       [-0.99677219, -0.0802818 ],
       [-0.99235725,  0.12339801],
       [-0.94675456,  0.32195619],
       [-0.86185686,  0.5071516 ],
       [-0.74118783,  0.6712977 ],
       [-0.58975583,  0.80758162],
       [-0.41384604,  0.9103469 ],
       [-0.22075958,  0.97532827],
       [-0.01851051,  0.99982868],
       [ 0.18450685,  0.98283125],
       [ 0.37986625,  0.92504144],
       [ 0.55945933,  0.82885784],
       [ 0.71583207,  0.69827252],
       [ 0.84249423,  0.53870543],
       [ 0.93418871,  0.35677939],
       [ 0.98710972,  0.16004524],
       [ 0.99906079, -0.04333159],
       [ 0.96954589, -0.24490994],
       [ 0.89979002, -0.43632331],
       [ 0.79268841, -0.60962711],
       [ 0.65268629, -0.75762836],
       [ 0.48559446, -0.87418429],
       [ 0.29834805, -0.95445723],
       [ 0.09871872, -0.99511547],
       [-0.10500794, -0.99447148],
       [-0.30437624, -0.952552  ],
       [-0.49111143, -0.87109689],
       [-0.65746306, -0.75348693],
       [-0.79652672, -0.60460354],
       [-0.90253057, -0.43062609],
       [-0.97107493, -0.23877553],
       [-0.99931487, -0.0370146 ],
       [-0.98607829,  0.16628263],
       [-0.93191456,  0.36267831],
       [-0.83907176,  0.54402103]])
```

In [11]:

```
# Plot
plt.plot(t, sol[:, 0], label='$y$')
plt.plot(t, sol[:, 1], '+', label='$\dot{y}$')
plt.legend()
```

Out[11]:

```
<matplotlib.legend.Legend at 0x1cb9a1b5518>
```



Para ampliar

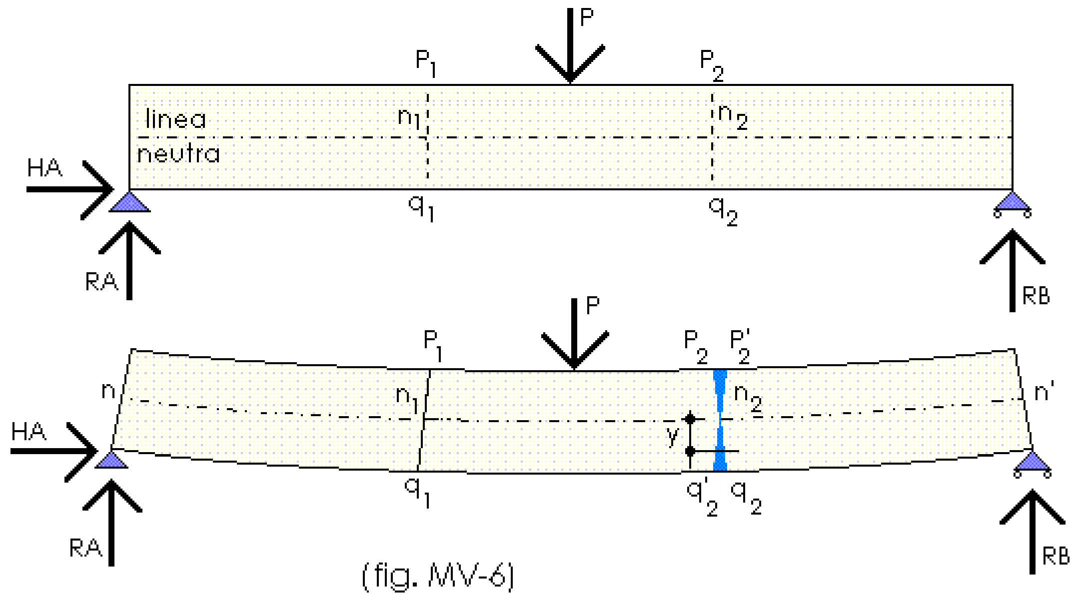
En el blog de www.pybonacci.org puedes encontrar algunos ejemplos interesante de Python científico. Entre otros, algunos relacionados con la integración de ecuaciones diferenciales:

- Salto de Felix Baumgartner desde la estratosfera: https://pybonacci.es/2012/10/15/el-salto-de-felix-baumgartner-en-python/ (https://pybonacci.es/2012/10/15/el-salto-de-felix-baumgartner-en-python/)
$$m\frac{d^2y}{dt^2} = -mg + D$$
- Modelo presa depredador, ecuaciones de Lotka-Volterra: https://pybonacci.es/2015/01/05/ecuaciones-de-lotka-volterra-modelo-presa-depredador/ (https://pybonacci.es/2015/01/05/ecuaciones-de-lotka-volterra-modelo-presa-depredador/)
$$\frac{dx}{dt} = \alpha x - \beta xy$$
$$\frac{dy}{dt} = -\gamma y + \delta yx$$

Ecuación de una viga: Modelo de Euler-Bernoulli

Problema tomado de https://mat.caminos.upm.es/wiki/Ecuaci%C3%B3n_de_vigas:_Modelo_de_Euler-Bernoulli_%2813A%29 (https://mat.caminos.upm.es/wiki/Ecuaci%C3%B3n_de_vigas:_Modelo_de_Euler-Bernoulli_%2813A%29), por María Aguilera, Paula Martínez, Miguel Sánchez, Laura García, Isabel Roselló y Sarah Boufounas

En esta sección vamos a estudiar el comportamiento de una viga de 10 metros de longitud y sección rectangular al someterla a la acción de diferentes esfuerzos.



En este caso, supondremos que el desplazamiento de la viga $y(x)$ y el momento flector $M(x)$ cumplen la ecuación de la elástica:

$$y'' = \frac{M(x)}{EI(x)}$$

Viga biapoyada sometida a la acción de momentos flectores

Datos:

- $E = 5 \cdot 10^4$
- $a = 0.6, b = 0.3$
- $M(x) = \frac{L}{2} - |x - \frac{L}{2}|$

Nos acordamos de la expresión para el momento de inercia:

$$I = \frac{1}{12} ba^3$$

(Luego veremos un caso en el que las dimensiones no son constantes a lo largo de la longitud)

En este caso las condiciones de contorno son:

$$y(0) = 0$$
$$y(L) = 0$$

Como **ya no tenemos un problema con condiciones iniciales sino un problema de contorno** hay que escoger un método apropiado, así que nosotros vamos a usar el **método del disparo** (*shooting method*).

Como no tenemos todas las condiciones iniciales, el método del disparo consiste en:

1. **Probar una solución inicial**.
2. Resolver el problema y comprobar si el otro extremo coincide con la condición de contorno dada.
3. Iterar hasta converger. Para nuestro caso se trata de hallar el valor apropiado k para $\dot{y}(0) = k$ tal que $y(L) = 0$

¡Vamos allá!

En primer lugar, probamos a resolver el problema con estas **condiciones iniciales**:

$$y(0) = 0$$
$$y'(0) = 5 \cdot 10^{-3}$$

Para comprobar que la solución no coincide con lo que buscamos.

In [12]:

```
# preserve
def f(y, x):
    return np.array([y[1], M(x) / (E * I(x))])
```

In [13]:

```
# preserve
# Datos
L = 10   # Longitud de la viga
E = 5e4  # módulo de Young
a = 0.6  # alto sección
b = 0.3   # ancho sección

# Dominio
x = np.linspace(0, L, num=50)

# Inercia
def I(x):
    return b * a**3 / 12

# Momento flector
def M(x):
    return L/2 - np.abs(x - L/2)
```

In [14]:

```
# Probamos solución con la condición inicial
yp0 = 5e-3  # Prueba
y0 = np.array([0, yp0])

sol = odeint(f, y0, x)
sol
```

Out[14]:

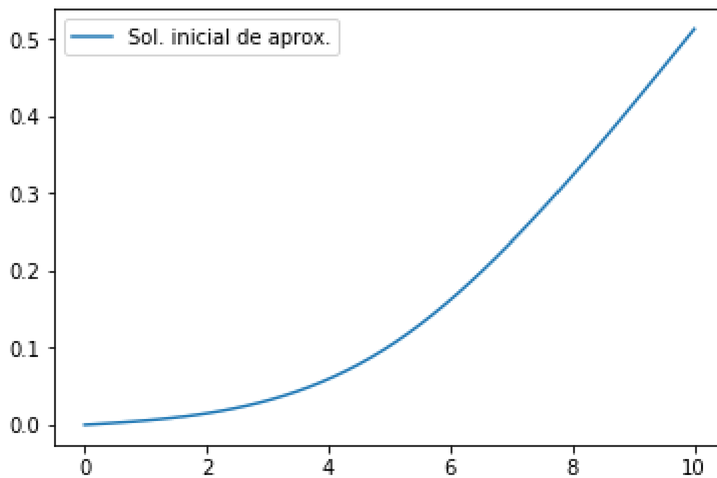
```
array([[0.        , 0.005     ],
       [0.00102567, 0.00507713],
       [0.0020828 , 0.00530851],
       [0.0032029 , 0.00569416],
       [0.00441744, 0.00623405],
       [0.00575791, 0.00692821],
       [0.00725578, 0.00777662],
       [0.00894253, 0.00877929],
       [0.01084965, 0.00993621],
       [0.01300862, 0.0112474 ],
       [0.01545092, 0.01271284],
       [0.01820803, 0.01433253],
       [0.02131143, 0.01610648],
       [0.0247926 , 0.01803469],
       [0.02868302, 0.02011716],
       [0.03301418, 0.02235388],
       [0.03781755, 0.02474486],
       [0.04312461, 0.0272901 ],
       [0.04896686, 0.02998959],
       [0.05537576, 0.03284334],
       [0.06238279, 0.03585134],
       [0.07001945, 0.03901361],
       [0.07831721, 0.04233012],
       [0.08730755, 0.0458009 ],
       [0.09702195, 0.04942593],
       [0.10749059, 0.05316666],
       [0.11871345, 0.05679169],
       [0.13066037, 0.06026247],
       [0.14329988, 0.06357899],
       [0.15660049, 0.06674125],
       [0.17053071, 0.06974925],
       [0.18505908, 0.072603  ],
       [0.2001541 , 0.0753025 ],
       [0.2157843 , 0.07784773],
       [0.23191819, 0.08023871],
       [0.2485243 , 0.08247543],
       [0.26557114, 0.0845579 ],
       [0.28302723, 0.08648611],
       [0.3008611 , 0.08826006],
       [0.31904125, 0.08987976],
       [0.33753622, 0.09134519],
       [0.35631451, 0.09265638],
       [0.37534465, 0.0938133 ],
       [0.39459516, 0.09481597],
       [0.41403456, 0.09566438],
       [0.43363136, 0.09635854],
       [0.45335408, 0.09689844],
       [0.47317124, 0.09728408],
       [0.49305137, 0.09751546],
       [0.51296298, 0.09759259]])
```

In [15]:

```
# La representamos
plt.plot(t, sol[:, 0], label='Sol. inicial de aprox.')
plt.legend()
sol[-1, :]
```

Out[15]:

```
array([0.51296298, 0.09759259])
```



Pero debe ser $y(10) = 0$

Ahora iteramos sobre el desplazamiento en L para converger a una solución buena. Para ello, vamos a crear una función:

- **Entrada:** y'(0)
- **Salida:** y(L)

In [16]:

```
def viga_biap(yp0):
    y0 = np.array([0, yp0])
    sol = odeint(f, y0, x)
    return sol[-1, 0]

viga_biap(5e-3)
```

Out[16]:

```
0.5129629801243984
```

Y ahora la resolvemos utilizando el método de la bisección o alguna función de `scipy.optimize` .

In [17]:

```
# importamos newton
from scipy.optimize import newton
```

In [18]:

```
# Lo utilizamos sobre la funcion anterior para hallar la solución
newton(viga_biap, 5e-2)
```

Out[18]:

```
-0.04629628736757714
```

¡Y ya lo tenemos!

In [19]:

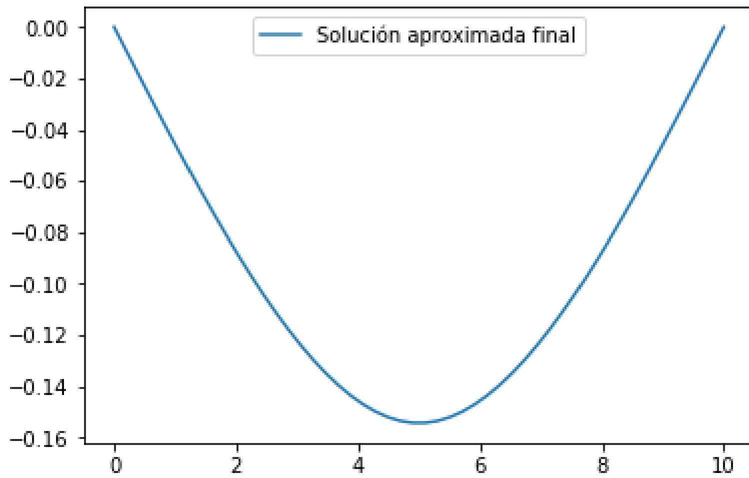
```
# Pintamos la solución final

# Cálculo de condición inicial para cumplir la condición
# de contorno
yp0 = newton(viga_biap, 5e-2)

# Condición inicial
y0 = np.array([0, yp0])

# Solución
sol = odeint(f, y0, x)
print(sol)
# Plot
plt.plot(t, sol[:, 0], label='Solución aproximada final')
plt.legend()
```

```
[[ 0.00000000e+00 -4.62962874e-02]
 [-9.44296136e-03 -4.62191590e-02]
 [-1.88544555e-02 -4.59877739e-02]
 [-2.82029877e-02 -4.56021322e-02]
 [-3.74570770e-02 -4.50622337e-02]
 [-4.65852425e-02 -4.43680784e-02]
 [-5.55560031e-02 -4.35196665e-02]
 [-6.43378780e-02 -4.25169979e-02]
 [-7.28993861e-02 -4.13600725e-02]
 [-8.12090466e-02 -4.00488904e-02]
 [-8.92353784e-02 -3.85834517e-02]
 [-9.69469006e-02 -3.69637562e-02]
 [-1.04312132e-01 -3.51898040e-02]
 [-1.11299592e-01 -3.32615950e-02]
 [-1.17877800e-01 -3.11791294e-02]
 [-1.24015274e-01 -2.89424070e-02]
 [-1.29680533e-01 -2.65514280e-02]
 [-1.34842098e-01 -2.40061922e-02]
 [-1.39468486e-01 -2.13066997e-02]
 [-1.43528216e-01 -1.84529505e-02]
 [-1.46989808e-01 -1.54449446e-02]
 [-1.49821781e-01 -1.22826819e-02]
 [-1.51992654e-01 -8.96616257e-03]
 [-1.53470946e-01 -5.49538651e-03]
 [-1.54225176e-01 -1.87035373e-03]
 [-1.54225168e-01  1.87034936e-03]
 [-1.53470939e-01  5.49538214e-03]
 [-1.51992648e-01  8.96615820e-03]
 [-1.49821776e-01  1.22826776e-02]
 [-1.46989804e-01  1.54449402e-02]
 [-1.43528213e-01  1.84529461e-02]
 [-1.39468483e-01  2.13066953e-02]
 [-1.34842096e-01  2.40061878e-02]
 [-1.29680533e-01  2.65514236e-02]
 [-1.24015274e-01  2.89424027e-02]
 [-1.17877801e-01  3.11791250e-02]
 [-1.11299594e-01  3.32615907e-02]
 [-1.04312135e-01  3.51897996e-02]
 [-9.69469045e-02  3.69637518e-02]
 [-8.92353832e-02  3.85834473e-02]
 [-8.12090523e-02  4.00488861e-02]
 [-7.28993927e-02  4.13600681e-02]
 [-6.43378855e-02  4.25169935e-02]
```

```
 [-5.55560115e-02  4.35196621e-02]
 [-4.65852517e-02  4.43680741e-02]
 [-3.74570871e-02  4.50622293e-02]
 [-2.82029987e-02  4.56021278e-02]
 [-1.88544674e-02  4.59877696e-02]
 [-9.44297419e-03  4.62191546e-02]
 [ 2.60534649e-12  4.62962830e-02]]
```

Out[19]:

```
<matplotlib.legend.Legend at 0x1cb9a1e2748>
```



Juan Luis Cano, Mabel Delgado, Alejandro Sáez, Jesús Espinola