Numerico_y_datos_2016

October 12, 2016

1 Taller de Python Capítulo 2: Numérico y Laboratorio

1.1 NumPy: vectores, matrices y tablas de datos

NumPy es LA biblioteca para cálculo vectorial. Además de contener un nuevo tipo de dato que nos va a ser muy útil para representar vectores y matrices, nos provee de un arsenal de funciones de todo tipo.

Vamos a empezar por importar la bliblioteca numpy. La sintaxis típica de eso era import biblio as nombre:

```
In [3]: import numpy as np # con eso voy a poder acceder a las funciones de numpy a través de np.funció

        # en ejemplo
        print('El numero e = ', np.e)
        print('O el numero Pi = ', np.pi)

        # y podemos calcular senos y cosenos entre otras cosas
        print(np.sin(np.pi)) # casi cero! guarda con los floats!
```

```
('El numero e = ', 2.718281828459045)
('O el numero Pi = ', 3.141592653589793)
1.22464679915e-16
```

Todo eso está muy bien, pero lo importante de numpy son los arrays numéricos. Los arrays numéricos nos van a servir para representar vectores (el objeto matemático) o columnas/tablas de datos (el objeto oriyinezco o de laboratorio).

La idea es que es parecido a una lista: son muchos números juntos en la misma variable y están indexados (los puedo llamar de a uno dando la posición dentro de la variable). La gran diferencia con las listas de Python es que los arrays de numpy operan de la forma que todos queremos: 1. Si sumamos o restamos dos arrays, se suman componente a componente. 2. Si multiplicamos o dividimos dos arrays, se multiplican o dividen componente a componente.

Veamos ejemplos usando la función array para crear arrays básicos.

```
In [19]: a = np.array([1, 2, 3, 4]) # array toma como argumento un vector_like (lista, tupla, otro arra
         b = np.array([5, 6, 7, 8])

         print(a + b) # vector suma
         print(a * b) # acá multiplicó

         # en el caso de las listas esto era muy molesto
         l1 = [1, 2, 3, 4]
         l2 = [5, 6, 7, 8]

         print(l1 + l2) # sumar concatena
         # print(l1 * l2) # esto ni siquiera se puede hacer!
```

```
[ 6  8 10 12]
[ 5 12 21 32]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Y al igual que con las listas, uno puede acceder a elementos específicos de un array:

```
In [26]: print(a[0], a[1], a[2], a[3]) # son 4 elementos, los indices van del 0 al 3
         # y más o menos vale todo lo que valía con listas
         print(b[-1]) # agarro al último elemento de b
         print(b[0:3]) # desde el primero hasta el 3 (no incluido el final, nunca se incluye)
```

```
1 2 3 4
8
[5 6 7]
```

Para facilitar la vida del usuario numpy viene con un montón de rutinas de creación de arrays típicos. En particular, matrices típicas como las identidades o de todos elementos iguales a 1 o 0 y arrays con cierta cantidad de elementos entre dos números (muy útil para crear dominios para gráficos).

Veamos ejemplos de esos:

```
In [7]: # equiespaciados
        equilin = np.linspace(0, 1, 9) # 10 número equiespaciados linealmente entre 0 y 1
        print('Equiespaciado lineal:', equilin)

        arange = np.arange(0, 1, 1./10) # como el range de Python pero puede venir con un paso en coma
        print('Como el range de las listas:', arange)

        identidad = np.identity(3)
        print('Identidad de 3x3:', identidad)
        print()

        #otros para que prueben ustedes
        ceros = np.zeros((4, 4)) # todos ceros, matriz de 4x4
        unos = np.ones((2,3)) # todos unos, matriz de 2x3
        ojos = np.eye(5, k=0) # unos en la diagonal, como identidad
        ojos2 = np.eye(5, k=2) # qué pasó acá?

        print(ceros)
        print()
        print(unos)
        print()
        print(ojos)
        print()
        print(ojos2)
```

```
('Equiespaciado lineal:', array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,  0.75 ,  0.875,  1.   ]
('Como el range de las listas:', array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9]))
('Identidad de 3x3:', array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]]))
()
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]]
```

```
()
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
()
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
()
[[ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.]]
```

Y antes de seguir, algo que siempre puede ser útil: los arrays tienen ciertas propiedades como su shape (de cuánto por cuánto) y el dtype (qué tipo de cosas tiene adentro). Podemos acceder a estos datos de la siguiente manera:

```
In [36]: x = np.linspace(0, 10, 1000) # ese array tiene 1000 elementos, andar printeando es poco prácti

         print(x.dtype) # array.dtype nos dice que tipo de elementos tiene el array
         ceros = np.zeros((100, 100)) # matriz de 100x100
         print(ceros.shape) # array.shape nos dice cuántas filas y columnas tiene el array


         # prueben que pasa cuando le piden el shape a un array con una sola fila o columna como el x
```

```
float64
(100, 100)
```

1.1.1 Ejercicio 1

Creen un vector "dominio" con 20 valores entre -5 y 5, y un vector "imagen" donde la imagen sea la de la función x^2

```
In [ ]: # Realicen el ejercicio 1
```

1.1.2 Ejercicio 2

Creen una matriz de 15x15 con los números del 1 al 255.

Ayuda: siempre que puedan utilicen las funciones de creación de arrays de numpy.

Ayuda bis: la función reshape hace lo que promete (reshapear) y puede tomar tuplas como argumento. (Si, hay que googlear. ¿Nunca les pasó?)

```
In [8]: # Realicen el ejercicio 2
```

1.2 Un poco de algebra lineal con NumPy

NumPy trae muchas funciones para resolver problemas típicos de algebra lineal usando a los arrays como vectores. El que nos interesa en general es el de autovalores y autovectores y el sistema de ecuaciones lineales, pero empecemos con un ejemplo más fácil:

```
In [15]: # cargamos el módulo de algebra lineal
         from numpy import linalg
         v = np.array([1, 1, 1])
         w = np.array([2, 2, 2])
         z = np.array([1, 0, 1])

         norma =linalg.norm(v) # la norma 2 / módulo del vector v
         print(norma)
         print(np.sqrt(3)) # calculado a mano
         print(norma == np.sqrt(3)) # y numpy sabe que son lo mismo
```

```
1.73205080757
1.73205080757
True
```

Ahora si, usemos los vectores que creamos recién para crear una matriz y digamosle a NumPy que calcule los autovectoresy autovalores de esa matriz:

```
In [46]: matriz = np.array([v, w, z], dtype=np.float64)
         #eig devuelve una tupla de arrays con los autovalores en un array 1D y los autovec en un array
         eigens = linalg.eig(matriz)
         print('Los autovalores:', eigens[0])
         print()
         print('Los autovectores:', eigens[1])

         #se terminó el problema
```

```
Los autovalores: [  3.41421356e+00  -1.23150120e-16   5.85786438e-01]

Los autovectores: [[  4.39732612e-01   7.07106781e-01  -3.03890631e-01]
 [  8.79465224e-01  -5.45825614e-17  -6.07781262e-01]
 [  1.82143212e-01  -7.07106781e-01   7.33656883e-01]]
```

Y para un sistema de ecuaciones del tipo $Ax = b$:

```
In [30]: mat = np.array([[1, 2, 5], [2, 5, 8], [4, 0, 8]], dtype=np.float64)
         b = np.array([1, 2, 3])
         x = linalg.solve(mat, b) #resuelve el sistema A*x = b
         print x

         #se terminó el problema
```

```
[ 0.67857143  0.07142857  0.03571429]
```

Por supuesto, también se puede hacer producto matriz con vector, y... oh si, se pueden calcular inversas.

```
In [33]: print np.dot(mat,x)
         print linalg.inv(mat)
```

```
[ 1.  2.  3.]
[[-1.42857143  0.57142857  0.32142857]
 [-0.57142857  0.42857143 -0.07142857]
 [ 0.71428571 -0.28571429 -0.03571429]]
```

1.2.1 Ejercicio 3

Parecen incrédulos. Fabriquen entonces la matriz

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{1}$$

cuyos autovalores son $\lambda_1 = 1$ y $\lambda_2 = -1$, hallen sus autovalores y autovectores (a mano y con Python) y calculen Ax con

$$x = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{2}$$

```
In [34]: # Realicen el ejercicio 3
```

1.2.2 Ejercicio 4

Inventen una matriz de 5x5 (con el método que quieran y ¡que no sea la identidad!) y averigüen si es invertible. Si están prestando atención, saben que para resolver este problema les conviene ver la documentación (Shift+Tab) de numpy.linalg.

Ayuda: ¿es necesario calcular la inversa? ¿Se acuerdan algo de álgebra lineal? ¿Qué importancia tienen los autovalores?

```
In [45]: # Realicen el ejercicio 4
```

1.3 Gráficos, datos y ajustes

Hacer gráficos es lo primero que aprendimos a hacer en Origin, así que es lo primero que vamos a aprender para reemplazarlo. Van a ver que no es nada complicado.

Primero, debemos importar las bibliotecas necesarias para graficar, numpy por si no la teníamos, y de la biblioteca matplotlib (que tiene infinitas funciones y posibilidades), solamente pyplot de donde sacaremos las funciones que necesitaremos para graficar.

```
In [36]: import numpy as np
         from matplotlib import pyplot as plt
         # muestra los gráficos en el mismo notebook
         %matplotlib inline
```

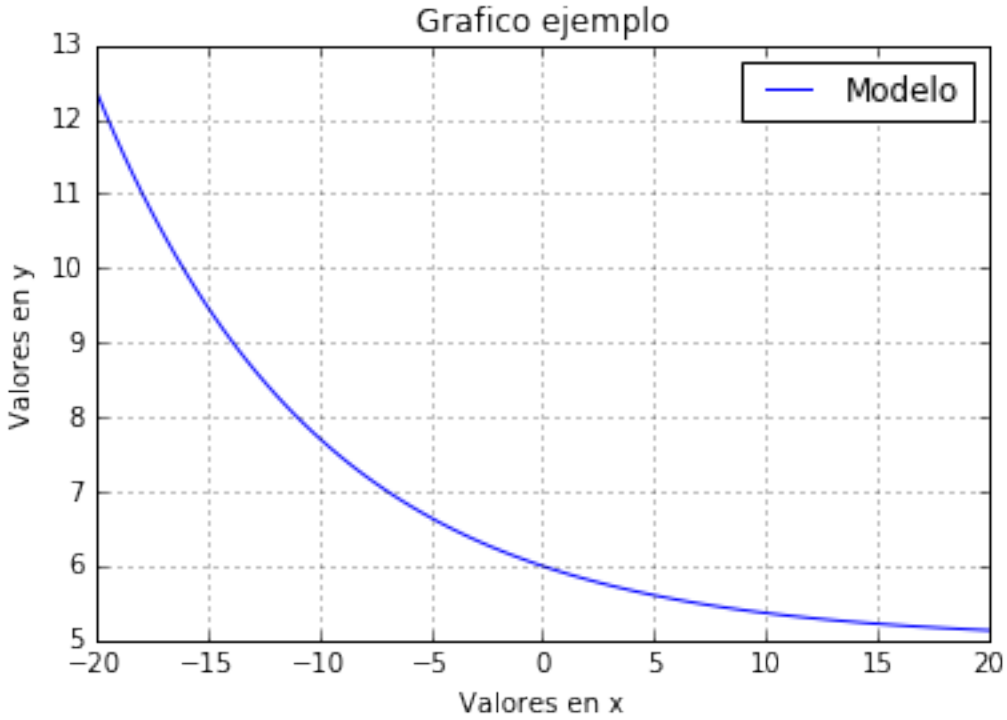
Luego, debemos definir lo que queremos ver ploteado. Definamos un dominio y una función que nos dé una imagen. Por ejemplo, la función

$$f(x) = Ae^{-x/\tau} + C$$

```
In [2]: # Definimos la función
        f = lambda x, A,T,C: A*np.exp(-x/T)+C

        # Definimos los parámetros
        A = 1
        T = 10
        C = 5

        # Definimos dominio e imagen
        x = np.linspace(-20, 20, 100)
        y = f(x,A,T,C)
```

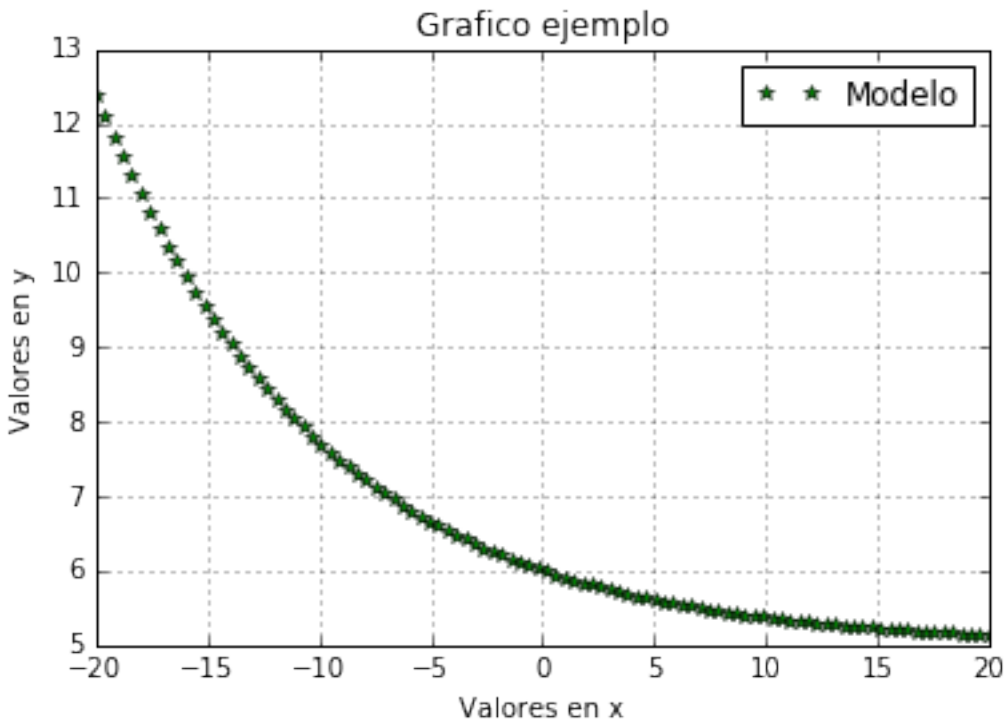
Ahora que tenemos dos vectores del mismo tamaño, uso las función plot para graficar, más otras funciones como grid, title, xlabel, ylabel, legend que le darán formato presentable a nuestro gráfico.

```
In [3]: # Ploteamos
        plt.plot(x, y, 'b-', label = 'Modelo')

        # Detalles del gráfico
        plt.grid(True) # Para que quede en hoja cuadriculada
        plt.title('Grafico ejemplo')
        plt.xlabel('Valores en x')
        plt.ylabel('Valores en y')
        plt.legend(loc = 'best')

        plt.show() # si no usaron %matplotlib inline, esto abre una ventanita con el gráfico
```



Notemos que dentro de la función plot pusimos como parámetros b- que significa que el color que queremos para la curva sea azul, y que el trazo sea una línea continua. Esto es customizable, pueden probar letras de otros colores (g, r, y, k) o bien otros trazos.

```
In [4]: # Ploteamos
        plt.plot(x, y, 'g*', label = 'Modelo')

        # Detalles del gráfico
        plt.grid(True) # Para que quede en hoja cuadriculada
        plt.title('Grafico ejemplo')
        plt.xlabel('Valores en x')
        plt.ylabel('Valores en y')
        plt.legend(loc = 'best')

        plt.show()
```



Imagino que se les ocurren infinitas cosas para customizar el gráfico. Lugares para buscar esos settings pueden ser el github de la FIFA, o también en el documentation de matplotlib

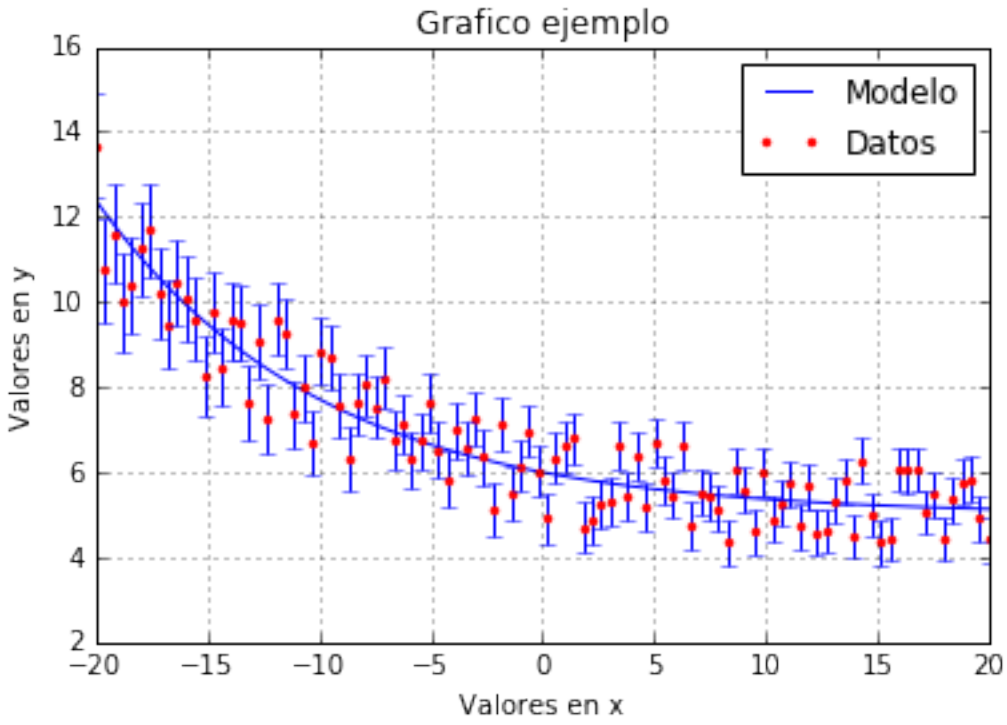
Vayamos ahora a un caso experimental. Supongamos que tenemos algunos datos medidos de alguna manera. Si los datos se refieren a este modelo, tendremos puntos de esta curva corridos aleatoriamente dentro de σ^2 (la varianza o desvío estandar). Para simular esta situación, tomaremos los datos del modelo y le sumaremos algún "ruido" estadístico. Usaremos la función de numpy, random.random para producir un vector de largo igual al vector y con números distribuidos entre el 0 y el 1, al multiplicarlo por dos, se distribuirán entre 0 y 2, y al restarle 1 estarán entre -1 y 1. Ahora lo multiplicamos por el 20% del valor del vector y solamente para que sea un ejemplo. Y propondremos una incerteza del 10% por ahora en y aunque podría incluirse en x también.

```
In [5]: # Creamos un ruido y lo agregamos a los datos
        ruido =  (2*(np.random.random(len(y)))-1)*0.2*y
        y_data = y + ruido

        # Proponemos un error
        error_y = 0.1*y
        #error_x = 0.05*x
```

Lo que resulta en el siguiente gráfico

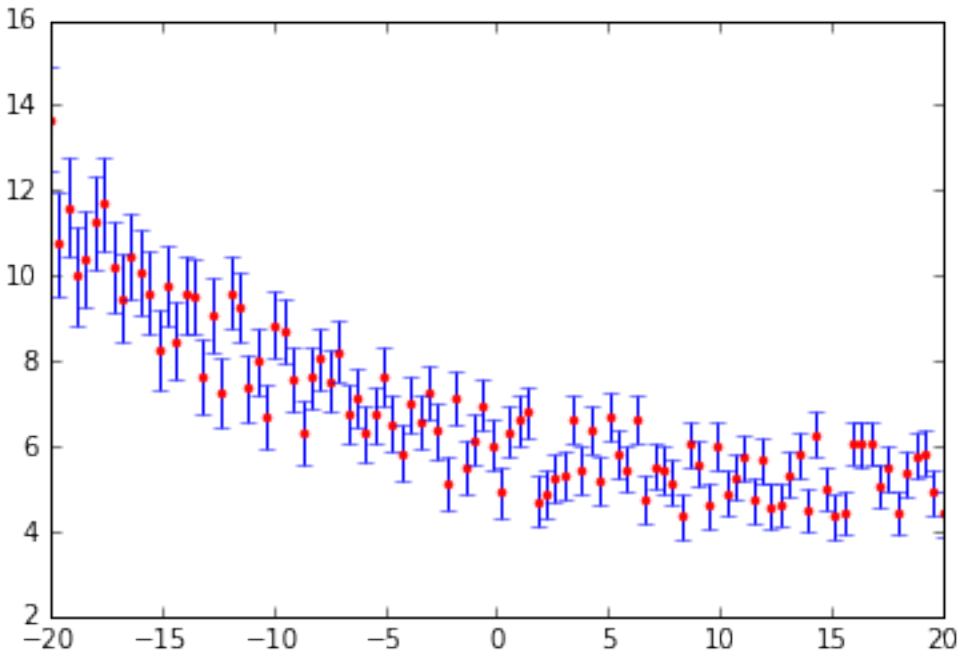
```
In [6]: plt.plot(x, y, 'b-', label = 'Modelo')
        plt.plot(x,y_data, 'r.', label = 'Datos')
        plt.errorbar(x, y_data, error_y, linestyle = 'None')
        #plt.errorbar(x, y_data, error_y , error_x)

        # Detalles del gráfico
        plt.grid(True) # Para que quede en hoja cuadriculada
```

```
plt.title('Grafico ejemplo')
plt.xlabel('Valores en x')
plt.ylabel('Valores en y')
plt.legend(loc = 'best')

plt.show()
```



Consideramos valiosos los datos obtenidos. Veamos como desde numpy pueden guardarse. Con otra función veamos también cómo se cargan datos, en este caso los mismos, pero en cualquier otro si los datos vienen de un instrumental o de cualquier fuente (ver documentación de estas funciones para poder usarlas en cualquier caso de presentación de datos, por ejemplo con formato cvs o cualquier otro)

In [7]:
```
# Guardamos y cargamos, a modo de ejemplo

np.savetxt('Datos_taller.txt', [x,y_data, error_y], delimiter = '\t')
Data = np.loadtxt('Datos_taller.txt', delimiter = '\t')

plt.plot(Data[0],Data[1], 'r.') # Veamos que son los mismos datos
plt.errorbar(Data[0], Data[1], Data[2], linestyle = 'None')
plt.show()
```



Obtenidos los datos, queremos el ajuste. aPara eso importamos la biblioteca con la función que usaremos, que aplica cuadrados mínimos para obtener los coeficientes.

```
In [13]: from scipy.optimize import curve_fit
```

El algoritmo de cuadrados mínimos necesita la función con la que queremos ajustar, que ya definimos en un renglón como función lambda (¡EN TU CARA ORIGIN!), dominio, los datos, un vector con los valores iniciales de los parámetros desde donde debe comenzar a iterar (un buen valor inicial haría que el tiempo que tarde en converger la solución sea menor, nosotros usaremos los que conocemos y propusimos) y una desviación estandar que hayamos considerado (en nuestro caso el vector error_y.

La función nos devolverá 2 cosas. Primero, los parámetros optimizados por este algoritmo, ordenados como los pusimos en la función lambda cuando la definimos, que lo guardamos en el vector popt. Por otro lado nos dará la matriz de covarianza, que tiene en su diagonal los σ^2 de cada parámetro, y fuera de ella nos dará el valor de covarianzas entre los parámetros (qué tanto se modifica un parámetro si variáramos un poco otro parámetro). Lo guardaremos en la matriz pcov aunque, para un análisis básico, usaremos sólo las varianzas y las guardaremos en un vector llamado sigmas.

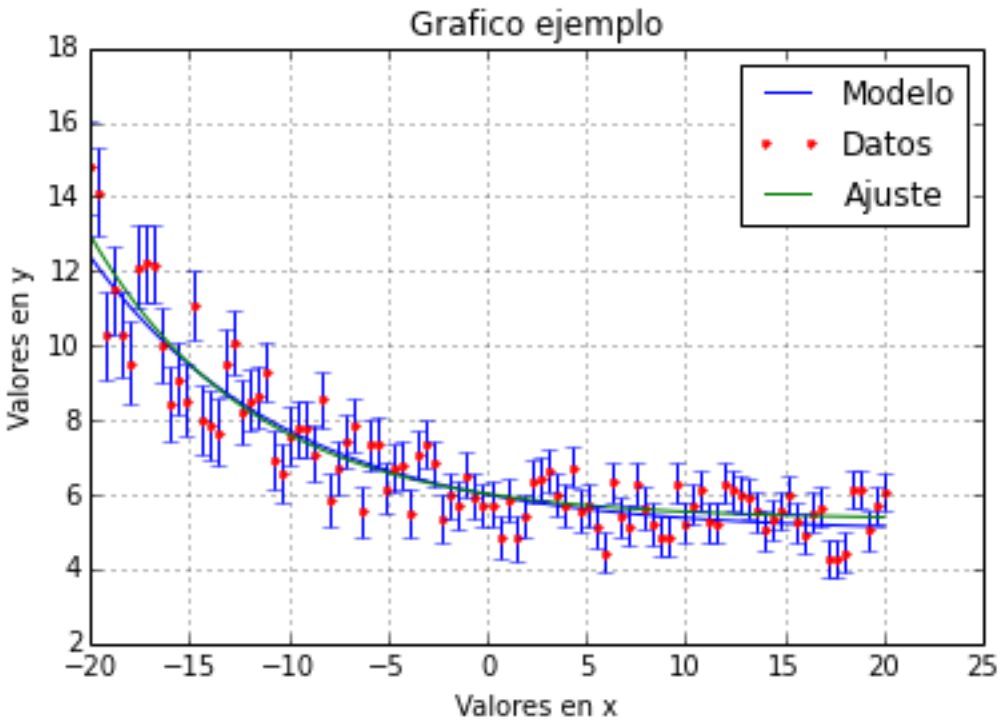
```
In [31]: # Ajustamos
         popt, pcov = curve_fit(f, x, y_data, sigma = error_y)
         sigmas = [pcov[0,0],pcov[1,1], pcov[2,2]]
         print popt, sigmas
```

```
[ 0.68794298  8.31174041  5.33001347] [0.025277527651388246, 0.82622616804029547, 0.018262671020511122]
```

Listo, ahora chequeamos con un gráfico que haya ajustado

```
In [32]: plt.plot(x, y, 'b-', label = 'Modelo')
         plt.plot(x,y_data, 'r.', label = 'Datos')
         plt.plot(x,f(x, popt[0], popt[1],popt[2]), 'g-', label = 'Ajuste') # Hacemos el gráfico en otr
                                                                            #la función evaluada en los
         plt.errorbar(x, y_data, error_y, linestyle = 'None')
```

```
# Detalles del gráfico
plt.grid(True)
plt.title('Grafico ejemplo')
plt.xlabel('Valores en x')
plt.ylabel('Valores en y')
plt.legend(loc = 'best')

plt.show()
```



1.3.1 Ejercicio 5

1. Hagan un ajuste sobre la función $f(x) = Acos(\omega x)$ con $A = 2$ y $\omega = 3$ para 40 valores en $Dom = [-\pi, \pi]$ con valores que varían el 15% del valor dado por el modelo, y compare los parámetros obtenidos con los dados.
2. **Bonus track**: Se puede escribir en LaTeX sobre los gráficos. Averiguen qué biblioteca hace falta importar y presenten los parámetros ajustados en el título.

```
In [46]: # Realicen el ejercicio 5
```

1.4 Histogramas

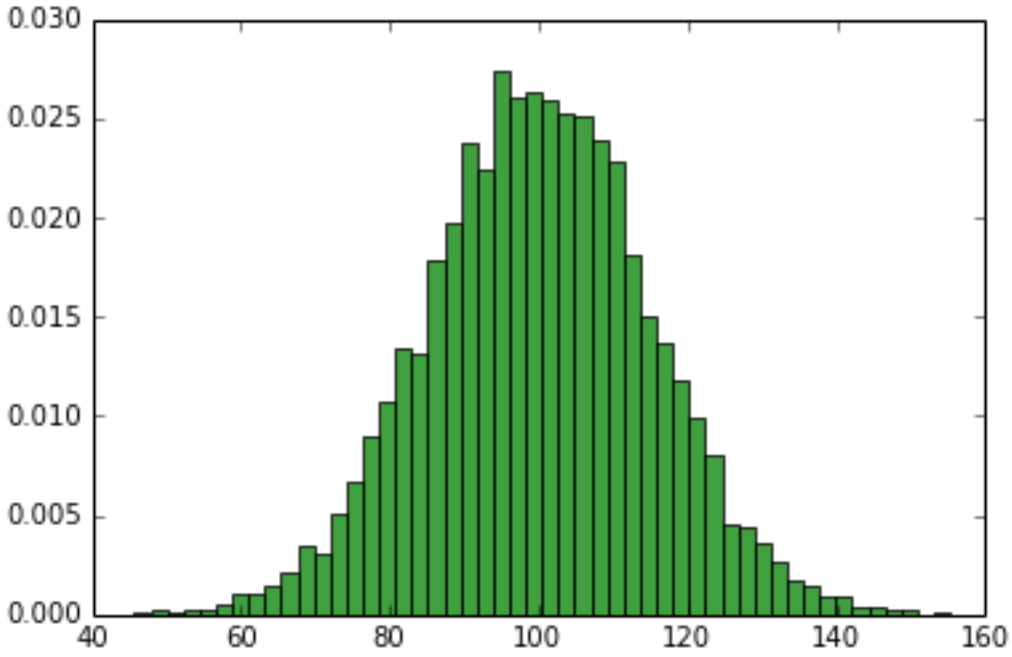
Una cosita más que nos va a ser útil a la hora de dejar el Oriyin sin instalar es poder hacer histogramas. Con pyplot eso lo podemos obtener de la función hist.

Recordemos que en un histograma dividimos una serie de datos en rangos y contamos cuántos de nuestros datos caen en cada rango. A esos rangos se los llama bins.

hist toma como argumentos un array de números, en cuántos bins queremos dividir a nuestro eje x y algunas otras opciones de color como constante de normalización y color de las barras.

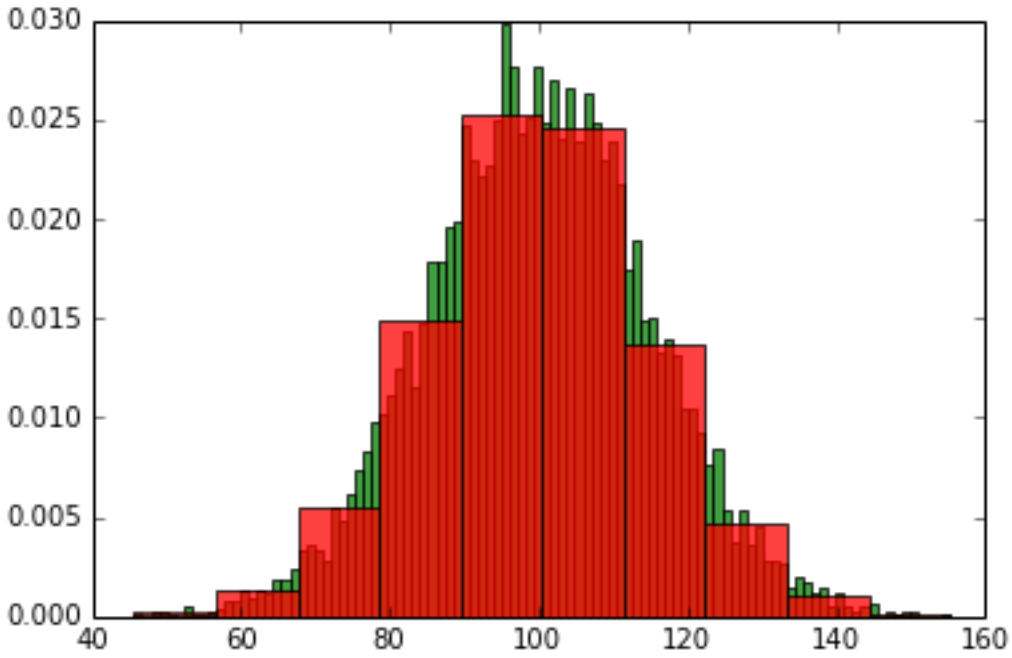
Hagamos un histograma simple de un set gaussiano. Para eso, creemos datos alrededor de algún valor medio usando randn de NumPy:

```
In [40]: mu, sigma = 100, 15 # mu es mi valor medio, sigma la desviación
         x = mu + sigma*np.random.randn(10000) # le sumo ruido gaussiano a mu

         n, bins, patches = plt.hist(x, bins=50, normed=1, facecolor='green', alpha=0.75)
         # en la variable n se encuentran los datos del histograma
         # bins es un vector con los bordes de los rangos de datos
         # patches no nos interesa en general
```



Y ya que estamos, para mostrar cómo afecta la elección de bins, graficamos dos histogramas uno arriba del otro.

```
In [41]: n, bins, patches = plt.hist(x, bins=100, normed=1, facecolor='green', alpha=0.75)
         n, bins, patches = plt.hist(x, bins=10, normed=1, facecolor='red', alpha=0.75)
```



1.4.1 Ejercicio 6

La función randn que usamos nos brinda números aleatorios distribuidos de manera gaussiana (distribución normal). 1. Haga el mismo ejercicio de recién pero con números aleatorios distribuidos de manera uniforme 2. Lo mismo pero con números con una distribución de χ^2

1.5 Resolución de un sistema de ecuaciones diferenciales ordinarias (ODE)

Vamos con un caso simple y conocido por la mayoría: el infaltable problema del péndulo. Arrancamos desde donde sabemos todos.

$$\frac{d^2\theta}{dt^2} + \omega^2\theta = 0$$

con $\omega^2 = \frac{g}{l}$

Para integrar numéricamente, proponemos utilizar la función odeint de la biblioteca scipy.integrate, y esa función utiliza el método de Euler para la solución de ODE's, (inserte conocimientos de Cálculo Numérico aquí, sí, dale, cursala antes de recibirte) por lo que, para un problema con derivadas de segundo orden debemos armar un sistema de ecuaciones de primer orden.

Sea $\phi = \dot{\theta} \rightarrow \dot{\phi} = \ddot{\theta}$

Nos queda entonces

$$\dot{\theta} = \phi \tag{3}$$
$$\dot{\phi} = -\omega^2\theta \tag{4}$$

o bien

$$\begin{pmatrix} \dot{\theta} \\ \dot{\phi} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\omega^2 & 0 \end{pmatrix} \begin{pmatrix} \theta \\ \phi \end{pmatrix} \tag{5}$$

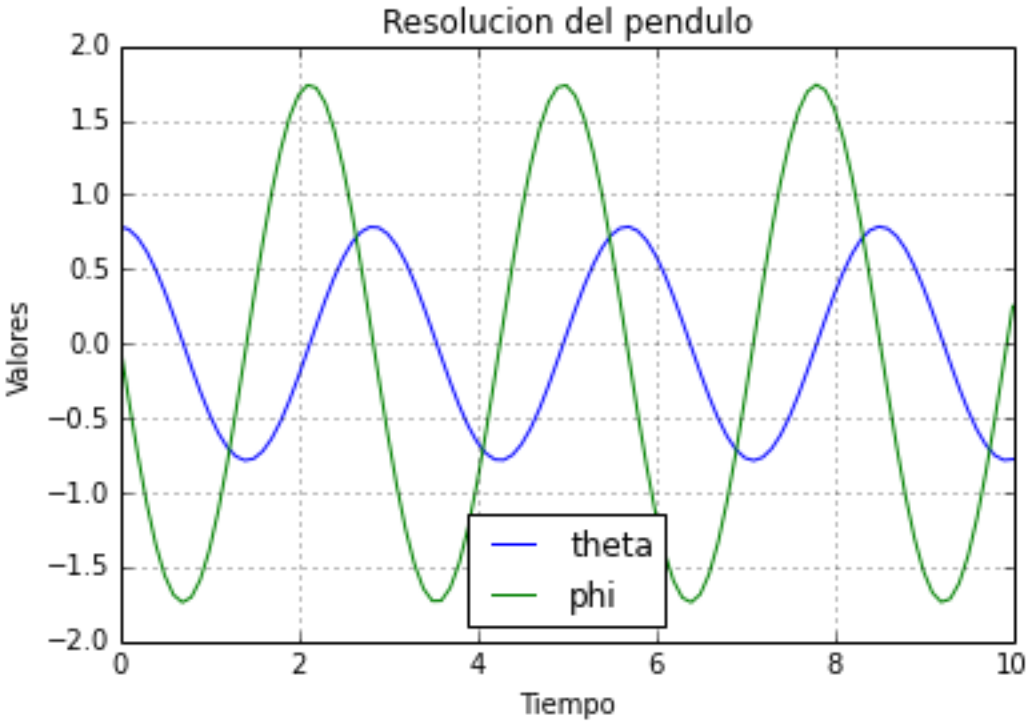
o también

$$\dot{\vec{X}} = A\vec{X}$$

Escencialmente, nuestro odeint va a intentar resolver ese sistema para distintos t mientras le hayamos dado un valor inicial de donde comenzar. Nada demasiado extraño. Así que lo que la función va a necesitar es una función def que tome como argumento $\vec{X}$ y opere para obtener $\dot{\vec{X}}$, el valor inicial y los tiempos donde se requiere resuelto, más parámetros como g y l en este caso.

```
In [43]: from scipy.integrate import odeint
```

```
In [44]: def ecdif(X,t,g,l):
             theta, phi = X
             omega2 = g/l
             return [phi, -omega2 * theta]

         g = 9.8
         l = 2
         X0 = [np.pi/4,0] # Inicio a 45 grados con velocidad = 0
         t = np.linspace(0, 10, 101)
         solucion = odeint(ecdif, X0, t, args = (g,l))

         plt.plot(t,solucion[:,0], label = 'theta')
         plt.plot(t,solucion[:,1], label = 'phi')
         plt.title('Resolucion del pendulo')
         plt.xlabel('Tiempo')
         plt.ylabel('Valores')
         plt.grid(True)
         plt.legend(loc = 'best')
         plt.show()
```



Altísimamente recomendado el resuelto del oscilador con el forzante completo del péndulo sin aproximación y comparado con la solución analítica (sí, esa que sin aproximación no buscó nadie). También recomendamos, a cuento de esto, un pasito más en resolución de ODE's que es una simulación con la biblioteca ipywidgets o tantas otras posibles.

1.5.1 Ejercicio 7

1. Resuelvan el oscilador armónico amortiguado con el parámetro γ
2. Vuelvan a todas esas guías que nadie terminó, busquen los ejercicios con asterisco, métanlos en Python y aprecien el poder del cálculo numérico.

```
In [ ]:
```

```
In [ ]:
```