Laboratorio3

February 7, 2017

1 Programación para Laboratorio 3

1.1 Valores numéricos y sus operaciones

Primero guardaremos datos en variables, esto es, inaugurar un espacio de memoria que quedará rotulado con el nombre que le pongamos a la variable, y allí almacenaremos el dato que pongamos luego del "=". Esto es una asignación, no es una igualdad matemática. Dependiento del tipo de dato es el espacio que se le asigna en memoria. Para Python, es una variable con un tipo distinto.

```
In [1]: a = 5
        b = 6
        c = 7.0
        d = 9.0
        print type(a), type(c)
```

```
<type 'int'> <type 'float'>
```

Podremos hacer todo tipo de cuentas con estas variables

```
In [3]: print a+b, type(a+b)
        print a+c, type(a+c)
        print ((c+d)/c)**2
```

```
11 <type 'int'>
12.0 <type 'float'>
5.22448979592
```

1.1.1 Ejercicio 1

Calculen el resultado de

$$\frac{(2+7.9)^2}{4^{7.4-3.14*9.81}-1}$$

y guárdenlo en una variable.

Se espera que dé el resultado -98.01

1.2 Nuestro amigo Numpy

Desde siempre hemos tenido limitaciones en matemática para simplificar expresiones, especialmente con las funciones trigonométricas, exponenciales, logarítmicas... ¿Cuánto es $sin(2)$? ¿e^7? ¿$log(15)$? En el CBC aprendimos Taylor, que "aproximaba" este tipo de valores reemplazando las funciones por polinomios. Los polinomios son amigos de las computadoras, porque las potencias son productos, y luego hay que sumar. Las computadoras son buenas sumando y producteando.

¿Y por qué aproximar? En numérico uno no tiene infinitos decimales porque es limitada la memoria física, pero Python se asegurará de que tengamos suficientes decimales como para empacharse. ¿Cuántos

errores generaremos por aproximar? Seguro que algún docente de cálculo numérico podrá contestarles tal inquietud.

No reinventaremos la rueda. Alguien ya escribió un algoritmo que calcula estas operaciones usando polinomios. Nosotros simplemente la importaremos para poder utilizarla. Existe una biblioteca llamada Numpy que almacena cualquier cantidad de funciones que usaremos en el cálculo numérico.

```
In [6]: import numpy as np        #import numpy as * es otra forma de importar

        print np.sin(np.pi*3/2), np.sin(5)
        print np.log(np.e), np.log(10)
        print pow(2,4)
```

```
-1.0 -0.958924274663
1.0 2.30258509299
16
```

¿Cómo sabremos cómo se utilizan estas funciones? Habrá que buscar su correspondiente documentation. Es super fácil de googlear, pero en el caso de Numpy, podremos encontrarla en

1.3 Vectores o arrays

Un array es una lista de valores. Con ellas podremos hacer varias cuentas al mismo tiempo (Python las hace de a una, pero nosotros las escribimos todas juntas). Un array, que es un tipo que importamos de Numpy, se define de la siguiente manera

```
In [8]: vec1 = np.array([1,2,3,4])
        vec2 = np.array([5,6,7,8])
        print vec1, vec2
        print type(vec1), type(vec2)
```

```
[1 2 3 4] [5 6 7 8]
<type 'numpy.ndarray'> <type 'numpy.ndarray'>
```

Las operaciones que hagamos con ellos las hará elemento a elemento, algo bastante cómodo para nuestro trabajo de laboratorio

```
In [10]: print vec1+vec2, vec2-vec1, vec1*vec2
         print vec1**2, 2**vec1, np.log(vec2)
```

```
[ 6  8 10 12] [4 4 4 4] [ 5 12 21 32]
[ 1  4  9 16] [ 2  4  8 16] [ 1.60943791  1.79175947  1.94591015  2.07944154]
```

Otra ventaja sobre las columnas del Oriyin es que es fácil buscar algunos valores en particular

```
In [12]: print max(vec1), min(vec1), len(vec2)
         print vec1[0], vec1[1], vec1[2], (vec1[0]+vec1[1])**2 #para llamar algunos elementos particula
```

```
4 1 4
1 2 3 9
```

1.3.1 Ejercicio 2

Armen dos vectores de siete elementos, sumenlos y eleven sus resultados al cuadrado. Del vector resultante impriman promedio de esos valores.

Hint: googleen la función mean

1.4 Funciones matemáticas

Algo muy típico de dominios matemáticos es un vector que almacene valores equidistantes. Aquí aparece nuestro nuevo amigo, linspace

```
In [20]: x1 = np.linspace(-3, 3, 100) #vector equiespaciado desde -3 hasta 3, 100 puntos
         x2 = np.linspace(0,10, 20) #vector equiespaciado desde 0 hasta 10, 20 puntos
         print x1
```

```
[-3.         -2.93939394 -2.87878788 -2.81818182 -2.75757576 -2.6969697
 -2.63636364 -2.57575758 -2.51515152 -2.45454545 -2.39393939 -2.33333333
 -2.27272727 -2.21212121 -2.15151515 -2.09090909 -2.03030303 -1.96969697
 -1.90909091 -1.84848485 -1.78787879 -1.72727273 -1.66666667 -1.60606061
 -1.54545455 -1.48484848 -1.42424242 -1.36363636 -1.3030303  -1.24242424
 -1.18181818 -1.12121212 -1.06060606 -1.         -0.93939394 -0.87878788
 -0.81818182 -0.75757576 -0.6969697  -0.63636364 -0.57575758 -0.51515152
 -0.45454545 -0.39393939 -0.33333333 -0.27272727 -0.21212121 -0.15151515
 -0.09090909 -0.03030303  0.03030303  0.09090909  0.15151515  0.21212121
  0.27272727  0.33333333  0.39393939  0.45454545  0.51515152  0.57575758
  0.63636364  0.6969697   0.75757576  0.81818182  0.87878788  0.93939394
  1.          1.06060606  1.12121212  1.18181818  1.24242424  1.3030303
  1.36363636  1.42424242  1.48484848  1.54545455  1.60606061  1.66666667
  1.72727273  1.78787879  1.84848485  1.90909091  1.96969697  2.03030303
  2.09090909  2.15151515  2.21212121  2.27272727  2.33333333  2.39393939
  2.45454545  2.51515152  2.57575758  2.63636364  2.6969697   2.75757576
  2.81818182  2.87878788  2.93939394  3.        ]
```

Pensando en la imagen de una función, tenemos dos formas de generarlo. La primera, usando como ejemplo la función $g(x) = \sqrt{x}$ es la siguiente.

```
In [15]: y2 = np.sqrt(x2)
         print y2
```

```
[ 0.          0.72547625  1.02597835  1.25656172  1.4509525   1.62221421
  1.77704663  1.91942974  2.0519567   2.17642875  2.29415734  2.40613252
  2.51312345  2.61574182  2.71448357  2.80975743  2.901905    2.99121521
  3.07793506  3.16227766]
```

La otra, más recomendada, es definiendo una función lambda. Nos va a servir para después hacer los modelos de los ajustes que usemos.

```
In [21]: f = lambda x: x**2 # "f es función de x tal que f(x) = x^2
         y1 = f(x1) #Tomo todo el vector x1 y le aplico f, eso me da la imagen que guardo en y1
         print y1
```

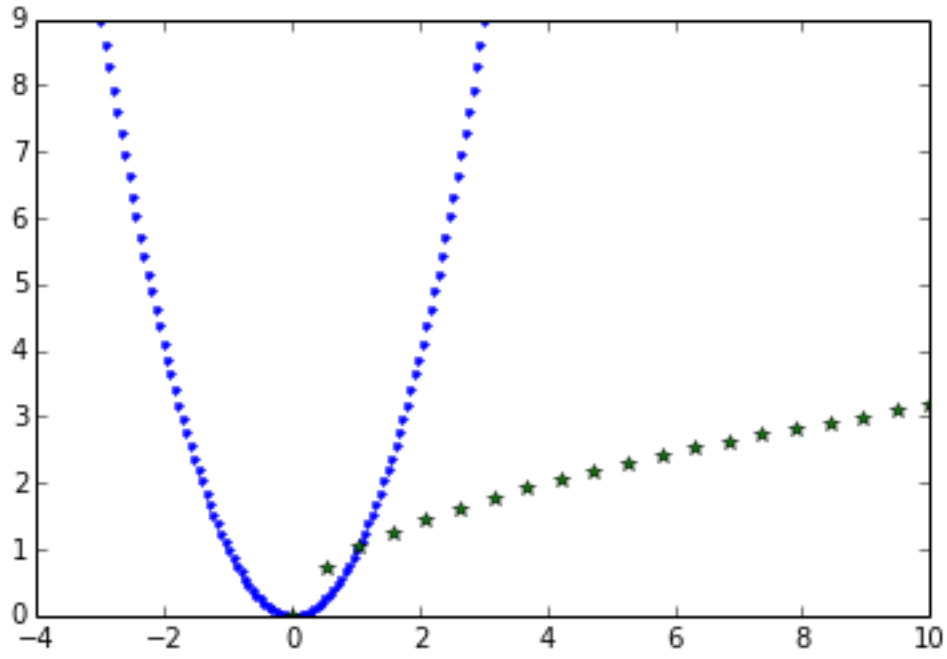
```
[  9.00000000e+00   8.64003673e+00   8.28741965e+00   7.94214876e+00
   7.60422406e+00   7.27364555e+00   6.95041322e+00   6.63452709e+00
   6.32598714e+00   6.02479339e+00   5.73094582e+00   5.44444444e+00
   5.16528926e+00   4.89348026e+00   4.62901745e+00   4.37190083e+00
   4.12213039e+00   3.87970615e+00   3.64462810e+00   3.41689624e+00
   3.19651056e+00   2.98347107e+00   2.77777778e+00   2.57943067e+00
   2.38842975e+00   2.20477502e+00   2.02846648e+00   1.85950413e+00
   1.69788797e+00   1.54361800e+00   1.39669421e+00   1.25711662e+00
   1.12488522e+00   1.00000000e+00   8.82460973e-01   7.72268136e-01
   6.69421488e-01   5.73921028e-01   4.85766758e-01   4.04958678e-01
   3.31496786e-01   2.65381084e-01   2.06611570e-01   1.55188246e-01
```

```
1.11111111e-01   7.43801653e-02   4.49954086e-02   2.29568411e-02
8.26446281e-03   9.18273646e-04   9.18273646e-04   8.26446281e-03
2.29568411e-02   4.49954086e-02   7.43801653e-02   1.11111111e-01
1.55188246e-01   2.06611570e-01   2.65381084e-01   3.31496786e-01
4.04958678e-01   4.85766758e-01   5.73921028e-01   6.69421488e-01
7.72268136e-01   8.82460973e-01   1.00000000e+00   1.12488522e+00
1.25711662e+00   1.39669421e+00   1.54361800e+00   1.69788797e+00
1.85950413e+00   2.02846648e+00   2.20477502e+00   2.38842975e+00
2.57943067e+00   2.77777778e+00   2.98347107e+00   3.19651056e+00
3.41689624e+00   3.64462810e+00   3.87970615e+00   4.12213039e+00
4.37190083e+00   4.62901745e+00   4.89348026e+00   5.16528926e+00
5.44444444e+00   5.73094582e+00   6.02479339e+00   6.32598714e+00
6.63452709e+00   6.95041322e+00   7.27364555e+00   7.60422406e+00
7.94214876e+00   8.28741965e+00   8.64003673e+00   9.00000000e+00]
```

1.5 Gráficos, todo lo que Oriyin te tacañeó

Para plotear, invitaremos a otro gran amigo, inseparable del físico que programa en Python, Matplotlib. Miren qué simple es tirar un gráfico con lo que ya tenemos.

```
In [23]: from matplotlib import pyplot as plt #Recuerden que se importa una vez y ya está
         %matplotlib inline # Esto es sólo para este cuaderno. No hace falta que lo escriban

         plt.plot(x1, y1, '.') #Ploteo x1 vs y1 con puntitos
         plt.plot(x2, y2, '*') #Ploteo x2 vs y2 con estrellitas
         plt.show()
```



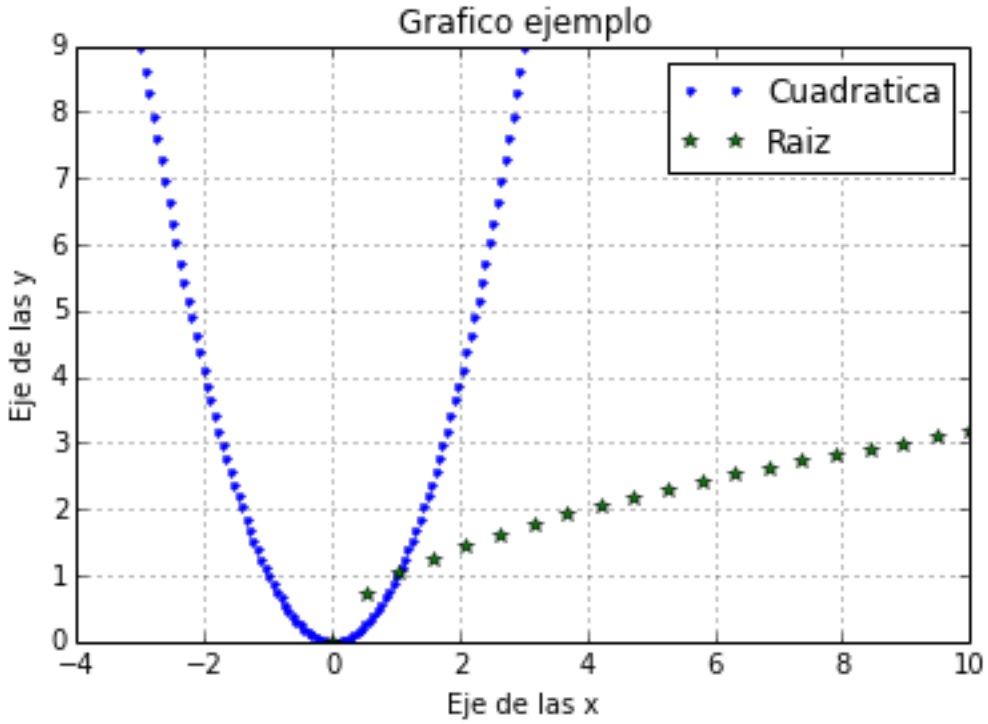
1.5.1 Ejercicio 3

Definan las funciones $f(x) = ln(x)$ y $g(x) = \frac{1}{x}$ con el mismo dominio de valores de 0 a 10, grafíquen ambos gráficos y estimen el resultado de la ecuación trascendental

$$ln(x) = \frac{1}{x}$$

Ahora, el chetaje. Veamos detalles que te dejen el gráfico pipí cucú. Grilla, título, nombres a los ejes y a las curvas.

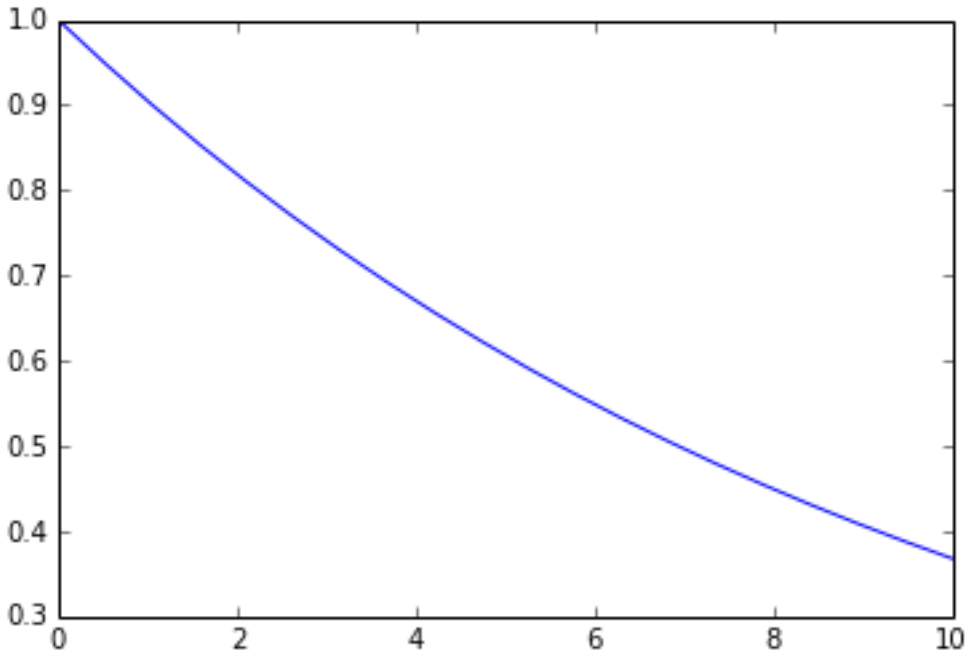
```
In [25]: plt.plot(x1, y1, '.', label = 'Cuadratica') #Noten que ahora le pongo un nombre a las curvas
         plt.plot(x2, y2, '*', label = 'Raiz')

         plt.grid(True)
         plt.title('Grafico ejemplo')
         plt.xlabel('Eje de las x')
         plt.ylabel('Eje de las y')
         plt.legend(loc = 'best')
         plt.show()
```



1.6 Gráficos en laboratorio

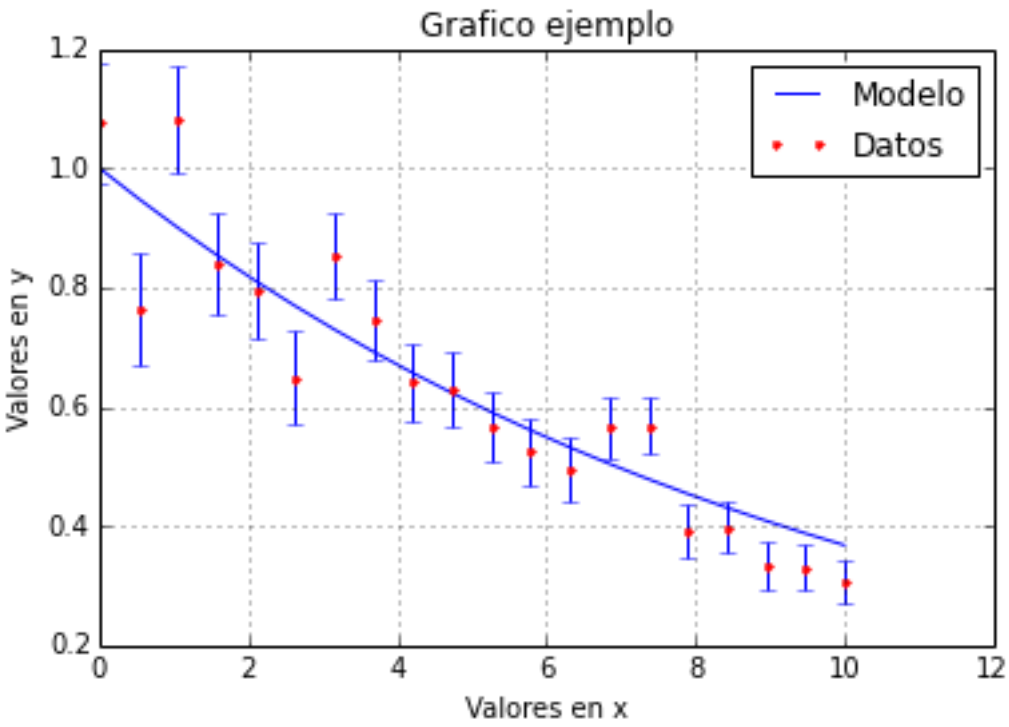
Para darle la última patadita al Oriyin, veamos como vamos a usar esto en el laboratorio. Supongamos un experimento donde el modelo que usamos es una exponencial del estilo $f(x) = Ae^{x/T}$

```
In [31]: f = lambda x, A, T: A*np.exp(-x/T)
         x = np.linspace(0, 10, 20)
         A = 1  #Estos serían los parámetros "correctos". Ahora los voy a usar para fabricarme los dato
         T = 10 #nada en este ejemplo. En el laboratorio estos los estiman, pero nunca los saben. Son l

         y = f(x,A,T) # Me creo una imagen con el dominio que me definí y con los parámetros que definí
         plt.plot(x,y)
         plt.show()
```



Este es el modelo. Yo del laboratorio mido datos concretos, que vienen con una distribución de probabilidad caracterizados por un valor medio y una desviación estandar. Acá me los voy a fabricar con una función random pero en la vida real los puntos que vamos a graficar ahora ustedes los miden con instrumentos.

```
In [32]: # Creamos un ruido para inventarnos datos
         ruido =  (2*(np.random.random(len(y)))-1)*0.2*y #Esto significa que quiero sumarles a cada pun
         y_data = y + ruido                               #un 20% de su valor. No le presten atención.

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

         #Y ahora ploteamos

         plt.plot(x, y, 'b-', label = 'Modelo')
         plt.plot(x,y_data, 'r.', label = 'Datos')
         plt.errorbar(x, y_data, error_y, linestyle = 'None') #Para los que se preguntaban cómo se pone
         #plt.errorbar(x, y_data, error_y , error_x)

         # Chiches 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()
```

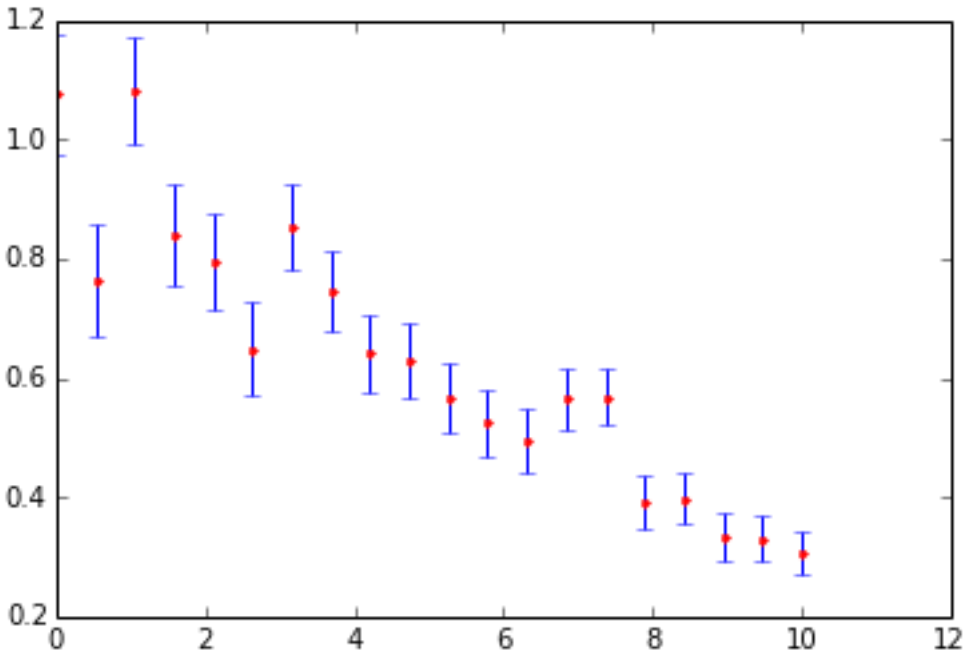


Bien. Ya vemos entonces como los puntos medidos se "parecen" a la ""realidad"". Supongamos que llegaste hasta acá, terminó el labo, te tenés que ir. ¿Y los datos? ¡Hay que guardarlos! Nos vamos a armar una matriz (algo que todavía no hicimos) y la vamos a guardar en un archivo .txt de la siguiente manera.

```
In [33]: np.savetxt('Datos_labo.txt', [x,y_data, error_y], delimiter = '\t')
```

Ahora, supongamos que los datos que midieron los guardaron en un Excel o en el mismo Oriyin (¿por qué alguien haría eso?). ¿Cómo los uso? Y acá hay un poco de maña, porque depende de cómo lo hallas guardado. Lo estandar sería que lo guarden en un texto plano de extensión .csv (comma separated values). Dependiendo de eso es cómo los cargan. Lo mejor que pueden hacer es leer la biblioteca de la función que vamos a usar. Pero sería algo así (coherente con cómo guardé mis datos fabricados)

```
In [35]: Data = np.loadtxt('Datos_labo.txt', delimiter = '\t') #Invento una variable que me guarda todo

         #Data es un vector de vectores, una matriz. La forma de llamar a una fila es con corchetes enu
         #las posiciones dentro del vector.

         plt.plot(Data[0], Data[1], 'r.')
         plt.errorbar(Data[0], Data[1], Data[2], linestyle = 'None')
         plt.show()
```



Ahora sí, llegó el momento que todos estaban esperando. Con ustedes. ¡El ajuste por cuadrados mínimos!

Para el ajuste, utilizaremos una función de otra biblioteca, pero como sólo necesitamos una función, importaremos sólo esa función.

```
In [38]: from scipy.optimize import curve_fit
```

Ahora sí, para ajustar le daremos a esta función un modelo (f), los puntos (x, y_{data}) y las incertezas ($error_y$). El resultado lo guardaremos de esta manera, para que estemos todos de acuerdo (hay otras maneras pero para empezar lo hacemos así).

```
In [42]: # Ajustamos
         popt, pcov = curve_fit(f, x, y_data, sigma = error_y)
         sigmas = [pcov[0,0],pcov[1,1]]
         print popt, sigmas
```

```
[ 1.04724101  8.77327431] [0.0029364824916215866, 0.50094471435290355]
```

La función curve_fit nos devolverá los parámetros optimizados (popt) y la matriz de covarianza (pcov) que tiene mucha información, entre otros, los σ^2 de los parámetros ajustados. Por eso los separamos de pcov en la variable sigmas.

Ahora chequeamos con un gráfico que haya ajustado, y ya está.

```
In [43]: plt.plot(x, y, 'b-', label = 'Modelo')
         plt.plot(x,y_data, 'r.', label = 'Datos')
         plt.plot(x,f(x, popt[0], popt[1]), 'g-', label = 'Ajuste') # Hacemos el gráfico en otro color
                                                                    #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()
```