Data Fitting with SciPy and NumPy

July 8, 2015

1 Data Fitting with SciPy and NumPy

Here we will look at two different methods to fit data to a function using Python.

1.1 SciPy and `curve_fit`

```
In [3]: %pylab inline
        params = {'backend': 'ps',
                  'text.usetex': 'true',
                  'ps.usedistiller': 'xpdf',
                  'font.size': 15,
                  'legend.fontsize': 13,
                  'figure.figsize': [7.5, 5]}
        rcParams.update(params)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Defining a linear function to generate data.
We will use the same function as argument for `curve_fit` to fit noisy data to it.

```
In [4]: def func(x, a, b):
            return a+b*x
```

```
In [5]: x = linspace(0, 10, 100)
        y = func(x, 2, 1)
```

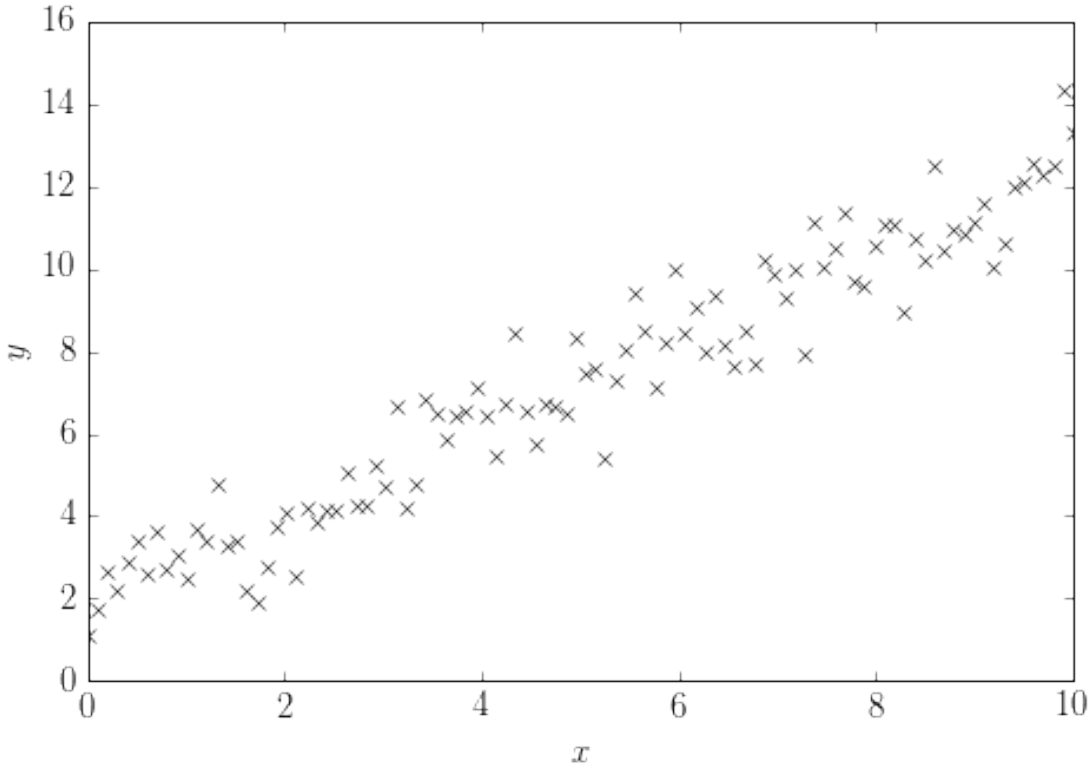
Now, we add some noise to the data using gaussian (normal) distributed random numbers.

```
In [6]: yn = y + 0.9*random.normal(size=len(x))
```

Let us plot the noisy data

```
In [7]: xlabel(r'$x$')
        ylabel(r'$y$')
        plot(x, yn, 'kx')
```

```
Out[7]: [<matplotlib.lines.Line2D at 0x10e7c3c90>]
```



Now we import the `curve_fit` function from the `scipy.optimize` package.

```
In [9]: from scipy.optimize import curve_fit
        popt, pcov = curve_fit(func, x, yn)
```

The function returns an array `popt` with the optimal parameters obtained using a non-linear least squares fit.
`pcov` is a 2d array with the estimated covariance of the parameters in popt.
It can be used to calculate the the standard deviation errors of `popt`. Type `curve_fit?` to get more information on this function.

```
In [11]: print(popt)
         print(pcov)
         perr = np.sqrt(np.diag(pcov))
```

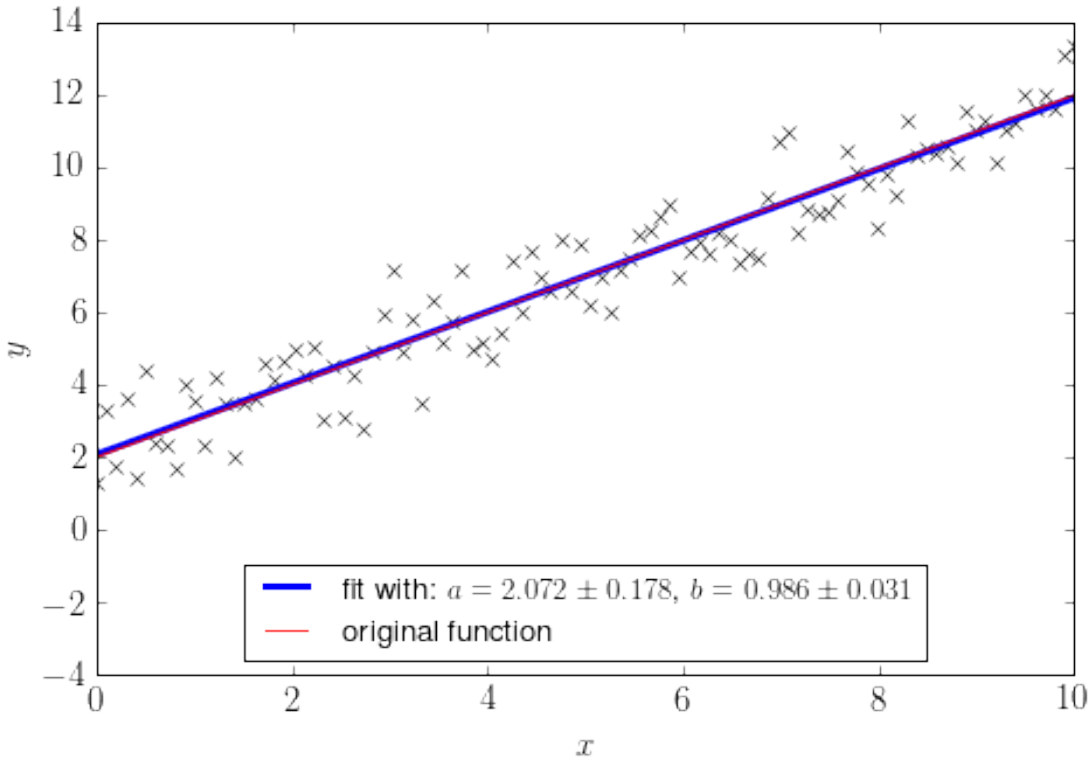
```
[ 1.85739078  1.07163101]
[[ 0.0294925  -0.00440164]
 [-0.00440164  0.00088033]]
```

Now, we will plot the noisy data again. But this time we include the original, unmodified function and our fit.

```
In [9]: xlabel(r'$x$')
        ylabel(r'$y$')
        ylim(-4, 14)
        plot(x, yn, 'kx')
        # here we create a string that contains the fit parameters and their corresponding standard err
        fit1 = r"fit with: $a={:.3f}\pm{:.3f}$, $b={:.3f}\pm{:.3f}$".format(popt[0], perr[0], popt[1], p
```

```
# we use that string as a label in the plot
plot(x, popt[0]+popt[1]*x, label=fit1, lw=3)
plot(x, y, label='original function', c='r')
legend(loc=8)
```

```
Out[9]: <matplotlib.legend.Legend at 0x10c2c9cd0>
```



The `curve_fit` function works with arbitrary functions. Basically every function you define can be plugged into `curve_fit`.
So let's look a another example.
We define a Gaussian like this: $g(x) = a \exp\left(\frac{-(x-b)^2}{2c^2}\right)$

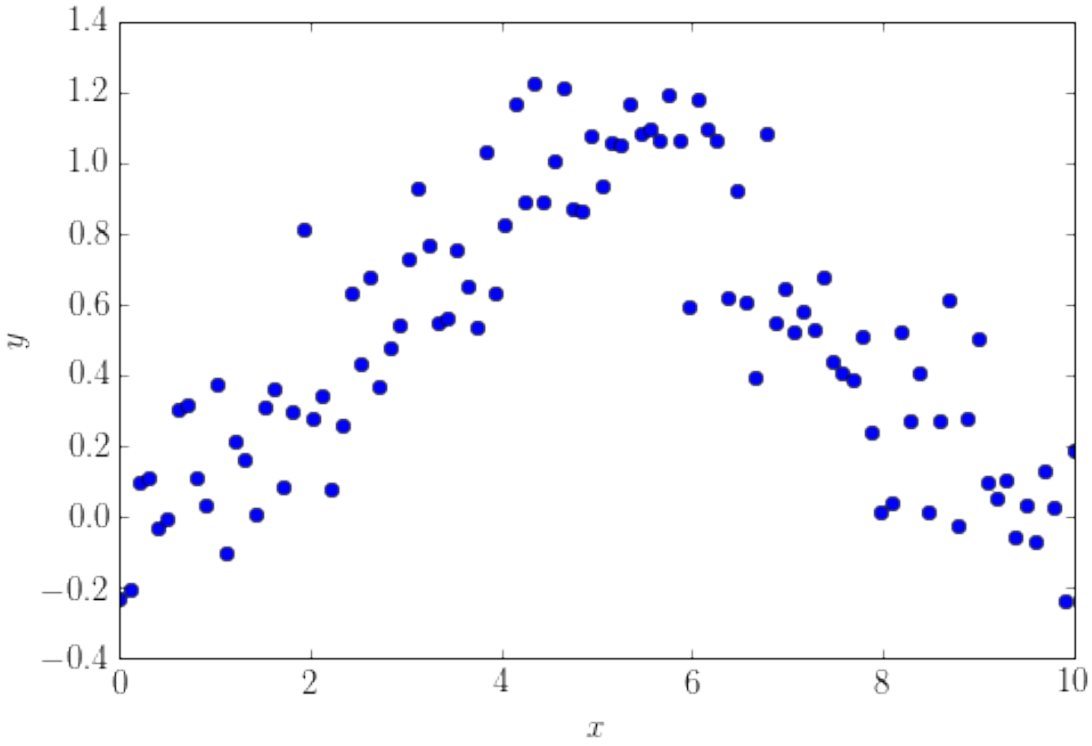
```
In [12]: def gauss(x, a, b, c):
             return a*exp(-(x-b)**2/(2*c**2))
```

```
In [14]: # Generate data and and random noise to it
         x2 = linspace(0, 10, 100)
         y2 = gauss(x2, 1, 5, 2)
         yn2 = y2 + 0.2*random.normal(size=len(x2))
```

Plot the noisy data

```
In [15]: xlabel(r'$x$')
         ylabel(r'$y$')
         plot(x2, yn2, 'bo')
```

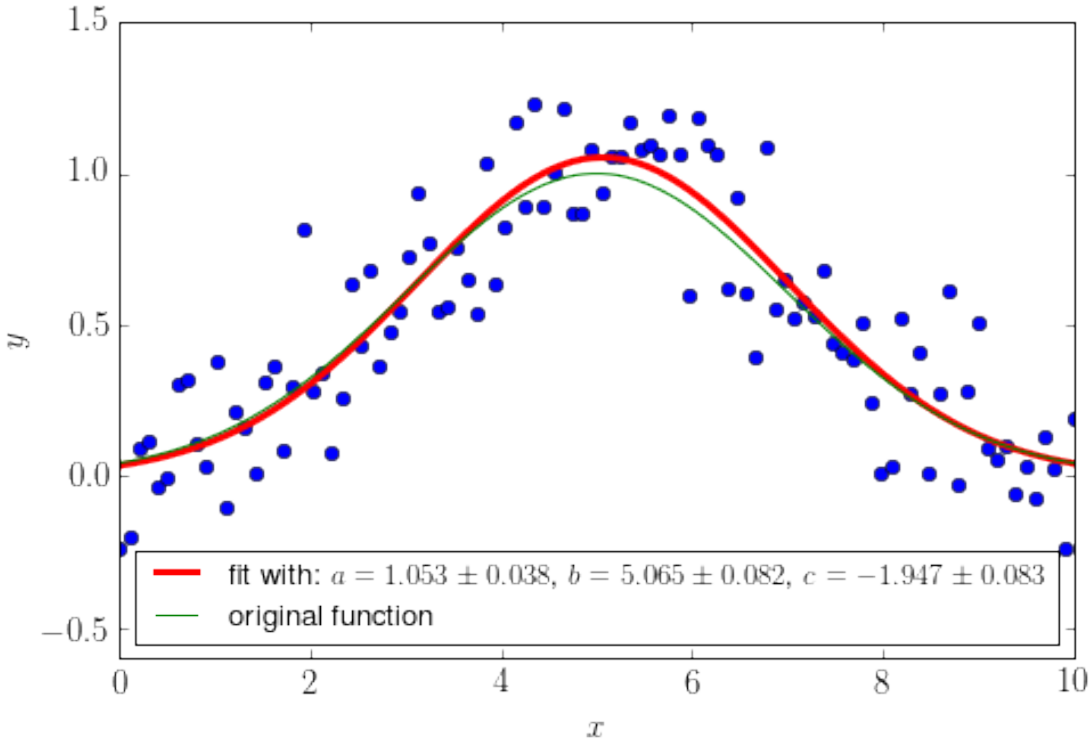
```
Out[15]: [<matplotlib.lines.Line2D at 0x10f704090>]
```



In [16]:
```
# Calling curve_fit with the Gaussian as target function
popt2, pcov2 = curve_fit(gauss, x2, yn2)
perr2 = np.sqrt(np.diag(pcov2))
```

In [17]:
```
ylim(-0.6, 1.5)
xlabel(r'$x$')
ylabel(r'$y$')
plot(x2, yn2, 'bo')
fit2 = r"fit with: $a={:.3f}\pm{:.3f}$, $b={:.3f}\pm{:.3f}$, $c={:.3f}\pm{:.3f}$".format(popt2
plot(x2, gauss(x2, popt2[0], popt2[1], popt2[2]), c='r', label=fit2, lw=3)
plot(x2, y2, c='g', label='original function')
legend(loc=8)
```

Out[17]:
```
<matplotlib.legend.Legend at 0x10f8d4c90>
```



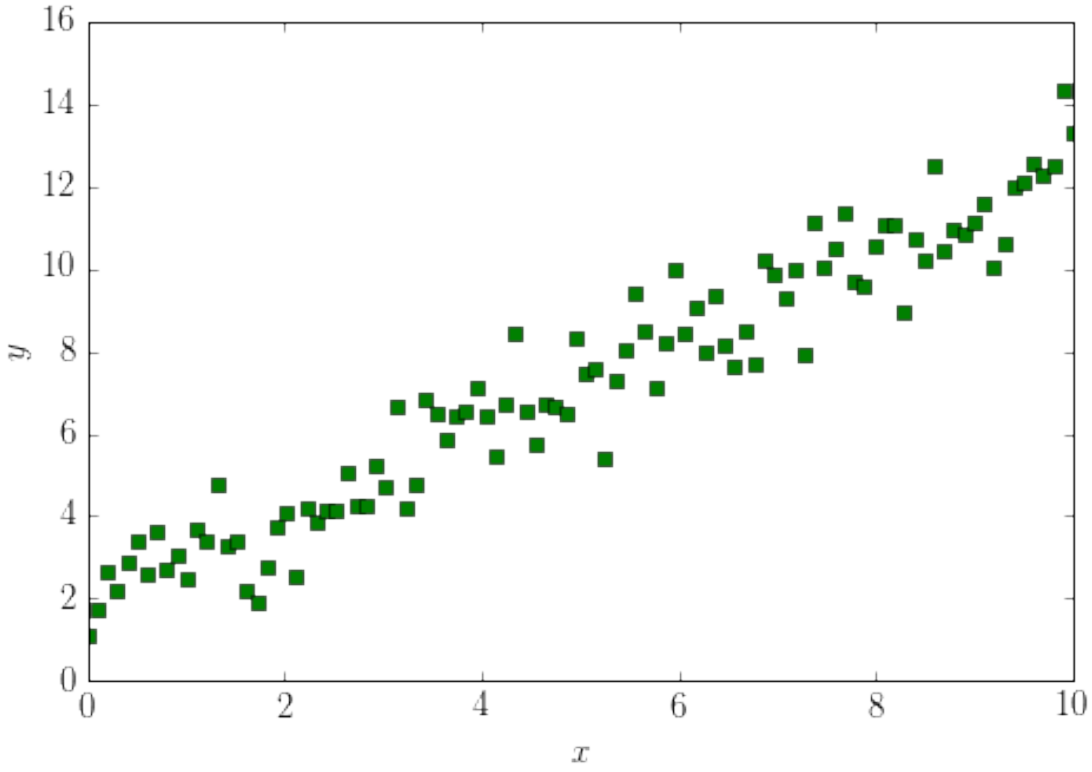
1.2 NumPy and `polyfit`

Without the need to define our function first we can use NumPy to fit data to polynomials of arbitrary degrees.

Let us replot our noisy data from the beginning. But this time with green squares!!

```
In [20]: xlabel(r'$x$')
         ylabel(r'$y$')
         plot(x, yn, 'gs')
```

```
Out[20]: [<matplotlib.lines.Line2D at 0x10fd097d0>]
```



With NumPy's `polyfit` we can estimate the coefficients of a polynomial.

```
In [21]: coeff = polyfit(x, yn, 1)

In [22]: print(coeff)

[ 1.07163101  1.85739078]
```

If we want to get error estimates for the fit parameters, we have to set the keyword argument `cov` to `True`.

```
In [23]: popt3, pcov3 = polyfit(x, yn, 1, cov=True)

In [24]: print(popt3)
         print(pcov3)

[ 1.07163101  1.85739078]
[[ 0.00089867 -0.00449334]
 [-0.00449334  0.03010692]]
```

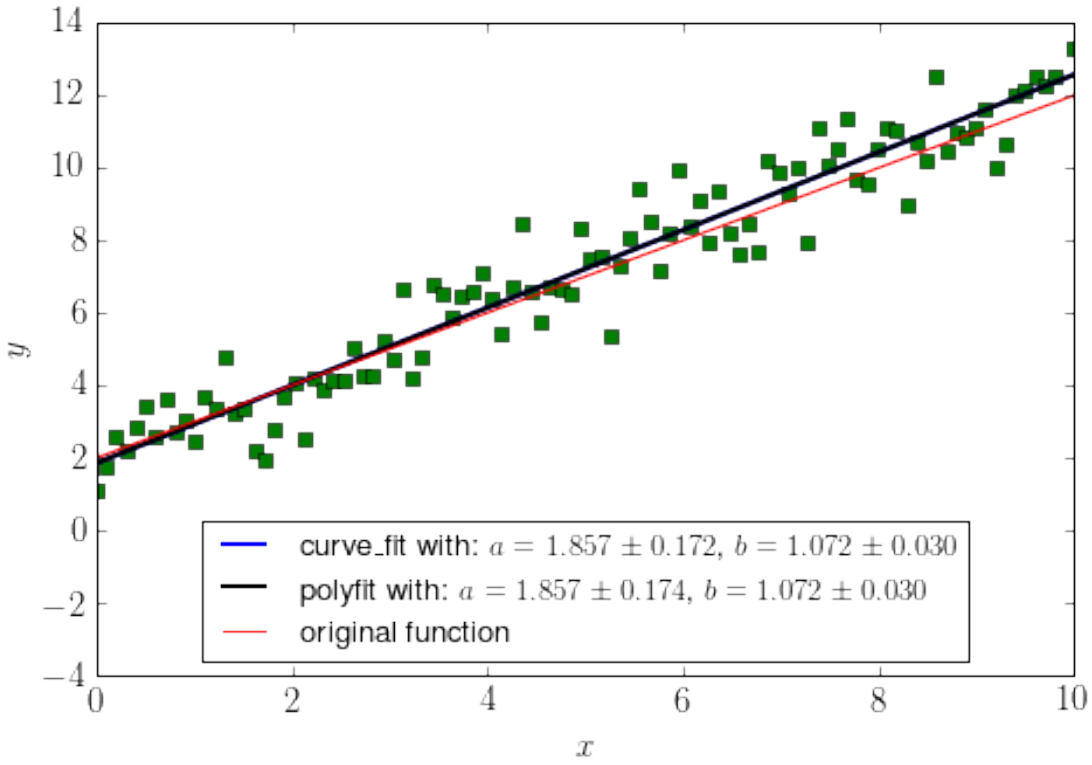
The diagonal of the returned matrix again contains the variances for the fit parameters. We can calculate the standard errors just as above.

```
In [25]: perr3 = np.sqrt(np.diag(pcov3))

In [26]: xlabel(r'$x$')
         ylabel(r'$y$')
         ylim(-4, 14)
         fit1 = r"curve\_fit with: $a={:.3f}\pm{:.3f}$, $b={:.3f}\pm{:.3f}$".format(popt[0], perr[0], p
```

```
fit3 = r"polyfit with: $a={:.3f}\pm{:.3f}$, $b={:.3f}\pm{:.3f}$".format(popt3[1], perr3[1], po
plot(x, yn, 'gs')
plot(x, popt[0]+popt[1]*x, label=fit1, lw=2, c='b')
plot(x, popt3[1]+popt3[0]*x, label=fit3, lw=2, c='k')
plot(x, y, label='original function', c='r')
legend(loc=8)
```

Out[26]: <matplotlib.legend.Legend at 0x10fe7f6d0>



Now let us look at the math behind `polyfit` and the linear least squares method!