Numerical Methods for Astrophysics:

FITS

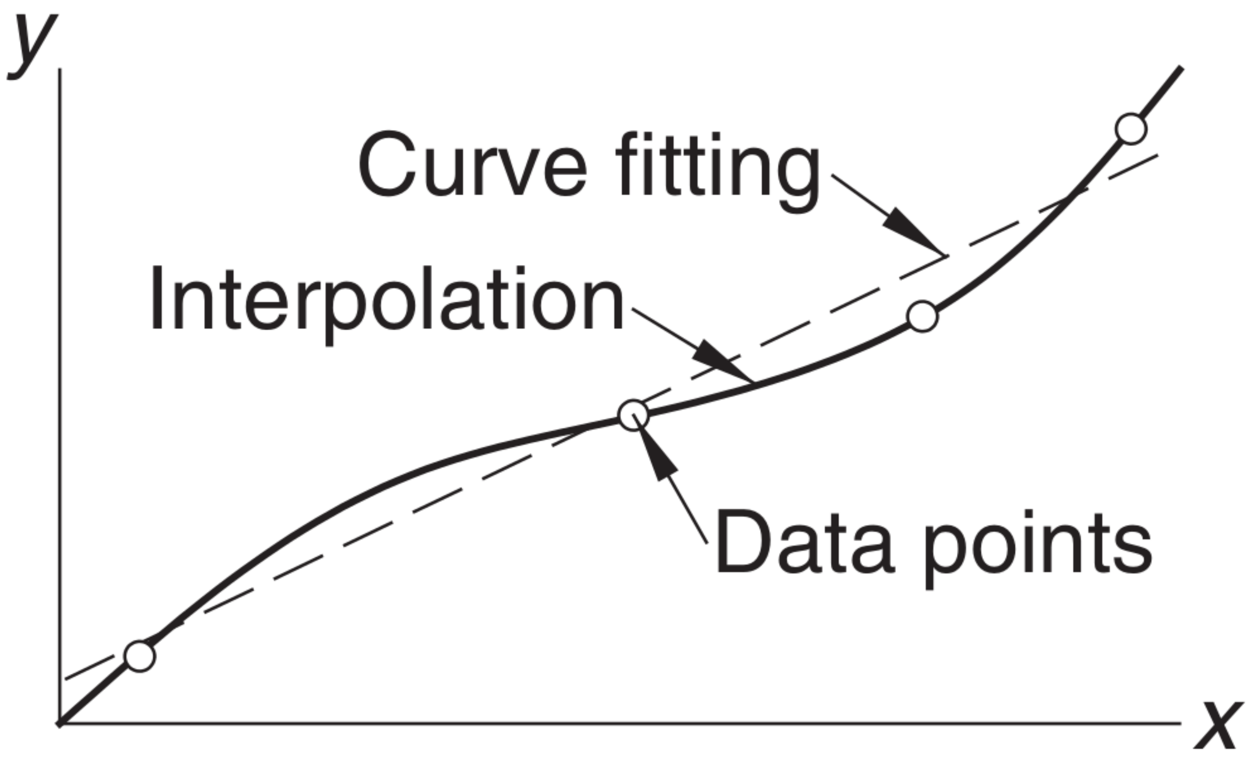
Michela Mapelli

Fits. Concept

Interpolation and curve fitting should not be confused.

Interpolation: curve passing through a discrete set of data points;
implicit assumption that all the data points are accurate.
Mostly done with **theoretical** sets of data points.

Curve fitting: smooth curve that approximates the data.
Thus, the curve does not necessarily hit the data points.
Mostly applied to data that contain scatter (noise), due to measurement errors.



Fits. Linear fit with the least square method

The simplest approach is to assume that our data points can be fit with a **straight line**.

y_i := set of measures in the points x_i with i = 0, 1, …, N – 1
(e.g. y_i = flux of astrophysical source at time x_i)

Assuming that the y_i and the x_i are connected by a linear relationship is equivalent to say that, if we could measure y_i with infinite accuracy and avoid-ing any kind of noise, the y_i would obey the following relationship:

$$y = A + B\,x$$

With A, B = constants, y and x = continuous version of y_i and x_i

Many (astro)physical quantities follow a **power law**
→ to fit data that intrinsically scale as a power law we can use the linear fit, applied to the logarithm of our data, i.e.

$$\tilde{y} = 10^A\,\tilde{x}^B \quad \text{where}\,\tilde{y} = 10^y,\ \tilde{x} = 10^x$$

Fits. Linear fit with the least square method

It is not possible to measure data without errors of measurement
→ first step is to assume the distribution of the errors of measurements

Simpler assumptions:

1. no error on the x_i, just error on the y_i

   Usually reasonable assumption – e.g. I measure the time with much more accuracy than the flux from an astrophysical source

2. measurements y_i distributed as a **Gaussian** distribution around the exact value

   $A + B\,x_i$ with a standard deviation σ_y

→ The probability of obtaining the observed value of y_i is

$$P_{A,B}(y_i) \propto \frac{1}{\sigma_y}\,\exp -\frac{(y_i - A - B\,x_i)^2}{2\,\sigma_y^2}$$

The probability of obtaining our complete set of measurements y_0 , y_1 , .., y_{N-1} is then

$$P_{A,B}(y_0, y_1, .., y_{N-1}) = P_{A,B}(y_0)\,P_{A,B}(y_1)..\,P_{A,B}(y_{N-1}) \propto \frac{1}{\sigma_y^N}\,\exp\left(-\chi^2/2\right)$$

where

$$\boxed{\chi^2 = \sum_{i=0}^{N-1} \frac{(y_i - A - B\,x_i)^2}{\sigma_y^2}}$$

Fits. Linear fit with the least square method

From $P_{A,B}(y_0, y_1, .., y_{N-1}) = P_{A,B}(y_0)\, P_{A,B}(y_1) ..\, P_{A,B}(y_{N-1}) \propto \frac{1}{\sigma_y^N} \exp\left(-\chi^2/2\right)$

$$\boxed{\chi^2 = \sum_{i=0}^{N-1} \frac{(y_i - A - B\, x_i)^2}{\sigma_y^2}}$$

we can derive the values of A and B as these for which the **probability *P*$_{A,B}$ is maximum**, i.e. these which **minimize the χ^2**.

For this reason, this method is called least **squares fitting**.

In practice, we differentiate χ^2 wrt A and B and set the derivatives to zero:

$$\frac{\partial \chi^2}{\partial A} = -\frac{2}{\sigma_y^2} \sum_{i=0}^{N-1} (y_i - A - B\, x_i) = 0$$

$$\frac{\partial \chi^2}{\partial B} = -\frac{2}{\sigma_y^2} \sum_{i=0}^{N-1} x_i\, (y_i - A - B\, x_i) = 0$$

which can be rewritten as $A\, N + B \sum x_i = \sum y_i$

$$A \sum x_i + B \sum x_i^2 = \sum x_i\, y_i$$

where we have defined $\sum \equiv \sum_{i=0}^{N-1}$

Fits. Linear fit with the least square method

After some math

$$A = \frac{\sum x_i^2 \sum y_i - \sum x_i \sum x_i\, y_i}{\Delta}$$

$$B = \frac{N \sum x_i\, y_i - \sum x_i \sum y_i}{\Delta}$$

where we have defined

$$\Delta \equiv N \sum x_i^2 - \left(\sum x_i\right)^2$$

Numerically, the above eqs for A and B are subject to **large rounding errors** because the two **terms of the subtraction** in the numerators and in the denominators are **similar numbers**

→ in your scripts, better to use the equivalent equations

$$\boxed{\begin{aligned} B &= \frac{\sum y_i\,(x_i - \langle x\rangle)}{\sum x_i\,(x_i - \langle x\rangle)} \\ A &= \langle y\rangle - \langle x\rangle\, B \end{aligned}}$$

where

$$\langle x\rangle = \frac{\sum x_i}{N}$$

$$\langle y\rangle = \frac{\sum y_i}{N}$$

are the mean values of the xi and yi data

which reduce rounding errors (see Chapter 4)

Fits. Linear fit with the least square method

Since we assumed that the measurements on y_i are normally distributed about the true value $A + B\,x_i$,
the deviations ($y_i - A - B\,x_i$) are also normally distributed,
with central value = 0 and standard deviation σ_y.
Hence:

$$\sigma_y = \sqrt{\frac{1}{(N-1)} \sum_{i=0}^{N-1} (y_i - A - B\,x_i)^2}$$

And we can derive the erros on A and B by error propagation:

$$\sigma_A = \sigma_y \sqrt{\frac{\sum x_i^2}{\Delta}}$$

$$\sigma_B = \sigma_y \sqrt{\frac{N}{\Delta}}$$

The larger the uncertainties on A and B,
the worse our assumptions that x_i and y_i are connected by a linear relationship

Fits. Linear fit with the least square method

Since we assumed that the measurements on y_i are normally distributed about the true value $A + B\,x_i$,
the deviations $(y_i - A - B\,x_i)$ are also normally distributed,
with central value = 0 and standard deviation σ_y.
Hence:

$$\sigma_y = \sqrt{\frac{1}{(N-1)} \sum_{i=0}^{N-1} (y_i - A - B\,x_i)^2}$$

And we can derive the erros on A and B by error propagation:

$$\sigma_A = \sigma_y \sqrt{\frac{\sum x_i^2}{\Delta}}$$

$$\sigma_B = \sigma_y \sqrt{\frac{N}{\Delta}}$$

The larger the uncertainties on A and B,
the worse our assumptions that x_i and y_i are connected by a linear relationship

Fits. Exercise on the least square method

$$B = \frac{\sum y_i \, (x_i - \langle x \rangle)}{\sum x_i \, (x_i - \langle x \rangle)}$$

EXERCISE:

$$A = \langle y \rangle - \langle x \rangle \, B$$

Write a script to calculate the least square fit using the formulas 277 for A and B and the formulas 279 and 280 for the uncertainties. Construct your "fake" set of data as follows

$$\sigma_y = \sqrt{\frac{1}{(N-1)} \sum_{i=0}^{N-1} (y_i - A - B\,x_i)^2}$$

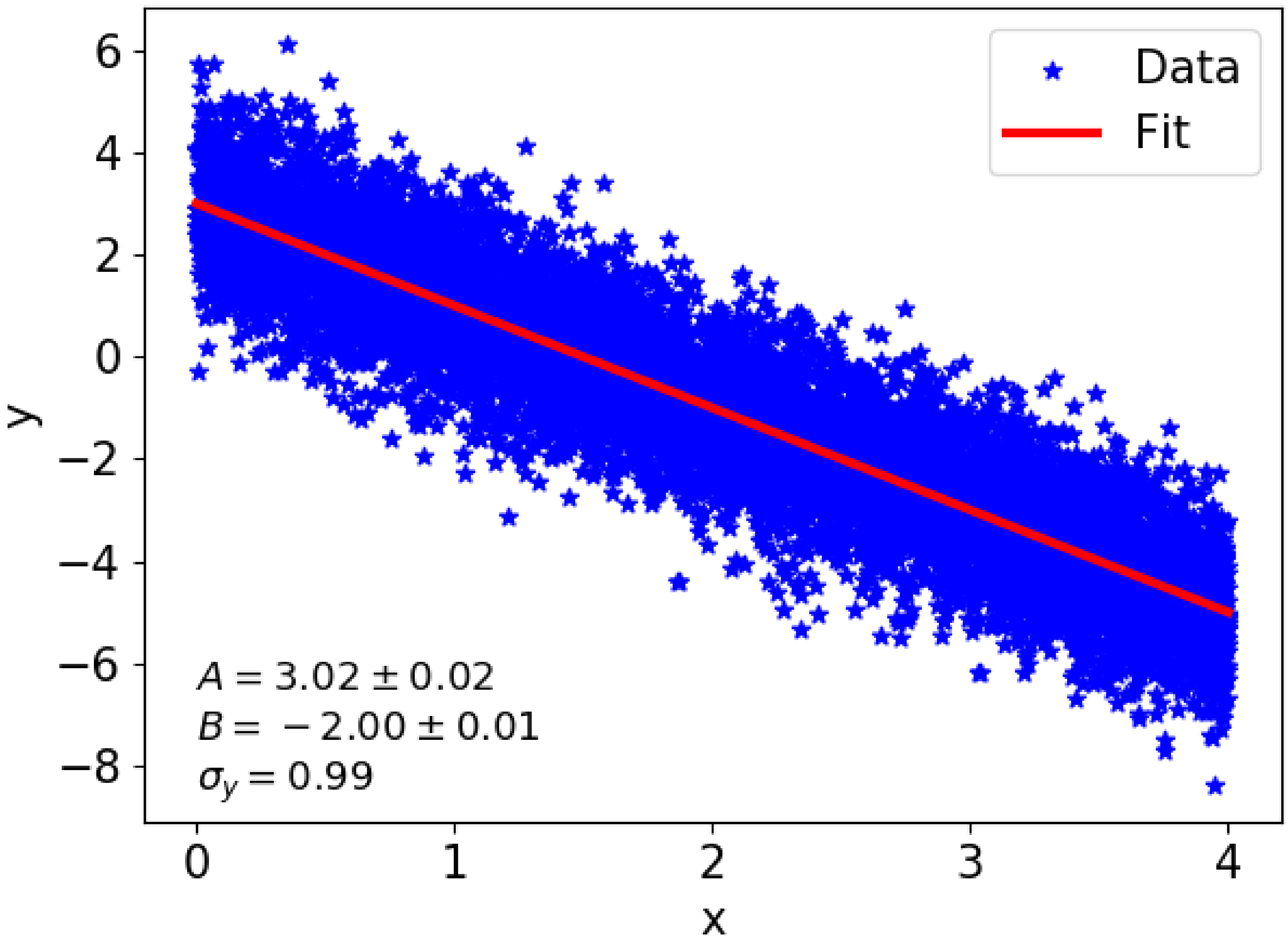
$$\sigma_A = \sigma_y \sqrt{\frac{\sum x_i^2}{\Delta}}$$

$$\sigma_B = \sigma_y \sqrt{\frac{N}{\Delta}}$$

```
def data_set():
    x=np.logspace(0,4,num=10000) #x data
    b=-2.0
    a=3.0
    y=10.**a * x**b #intrinsic relation y=10**a * x**b
    x=np.log10(x)
    y=np.log10(y)

    sigma=1.
    y=rnd.normal(y,sigma) #add gaussian noise
    return x,y
```

Use your script to fit the fake data. Your results should look like Figure 65.

Fits. Exercise on the least square method



Fits. Linear fit by weighting of data

The previous application of the least squares method does not include the possibility that the uncertainty is different for each single datum.

The most common case is that the **uncertainty does depend on the data.**

In this case, we might want to assign a **weight** to each single data point:

$$\chi^2 = \sum_{i=0}^{N-1} W_i \, (y_i - A - B\, x_i)^2$$

where $\mathbf{W_i}$ **: = weight of the i-th data point. Usual choice**: $\boxed{W_i \equiv 1/\sigma_{y_i}^2}$

We proceed as before by minimizing χ^2

$$\frac{\partial \chi^2}{\partial A} = -2 \sum_{i=0}^{N-1} W_i \, (y_i - A - B\, x_i) = 0$$

$$\frac{\partial \chi^2}{\partial B} = -2 \sum_{i=0}^{N-1} W_i \, x_i \, (y_i - A - B\, x_i) = 0$$

which yields $A = \dfrac{\sum W_i \, x_i^2 \, \sum W_i \, y_i - \sum W_i \, x_i \, \sum W_i \, x_i \, y_i}{\Delta}$

$$B = \frac{\sum W_i \, \sum W_i \, x_i \, y_i - \sum W_i \, x_i \, \sum W_i \, y_i}{\Delta}$$

where $\Delta \equiv \sum W_i \, \sum W_i \, x_i^2 - (\sum W_i \, x_i)^2$

Fits. Linear fit by weighting of data

As we said for the non-weighted linear fit, the previous equations for A and B suffer from large rounding errors.

Better to rewrite them as

$$\boxed{\begin{gathered} B = \frac{\sum W_i \, y_i \, (x_i - \langle x \rangle_w)}{\sum W_i \, x_i \, (x_i - \langle x \rangle_w)} \\[2em] A = \langle y \rangle_w - B \, \langle x \rangle_w \end{gathered}}$$

where

$$\langle x \rangle_w = \frac{\sum W_i \, x_i}{\sum W_i}$$

$$\langle y \rangle_w = \frac{\sum W_i \, y_i}{\sum W_i}$$

are the weighted averages of the xi and yi data

Fits. Exercise on linear fit by weighting of data

EXERCISE:

Redo the previous exercise with weights on the errors. Considering the following set up:

```
x=np.logspace(0.,4,num=10000) #x data
b=-2.0
a=3.0
y=10.**a * x**b #intrinsic relation y=10**a * x**b
x=np.log10(x)
y=np.log10(y)

sigma=np.zeros(len(x),float)
for i in range(len(y)):
    sigma[i]=10.*rnd.random()
    y[i]=rnd.normal(y[i],sigma[i])
w=1./sigma**2
return x,y,w
```

You will find that the result is as accurate as before (or even better) with a much larger apparent scatter of the data. Expected errors on A and B of the order of $\mathcal{O}(10^{-3} - 10^{-2})$.

Fits. General approach to least – square fitting

So far, we have seen only the application of the least-square fit method to a straight line. Let us now generalize to a **general function f (x)**

– Best general practice to follow:
first assume that the data can be fitted with a straight line.

If the fit we obtain with the straight line is poor,
then let us start considering some more complex functional form.

– Or special case:
if we have some **theoretical understanding** of
what the distribution of data should be,

it is smart to start fitting the data with the **function suggested by theory**.

If we have to consider a more complex function than a straight line,
it is important that this function has a **simple form** (for example a polynomial)

to **AVOID OVERFITTING** the data, i.e. to avoid that we start modeling
not only the underlying distribution of the data but also the noise

Fits. General approach to least – square fitting

Define a general function as $f(x) = f(x; a_0, a_1, ..., a_m)$

where x is the variable and a_0 , a_1 , ..., a_m are the *m* + 1 parameters.

If we want to fit this function to our data (x_i, y_i) with i = 0, 1, ..., *N* − 1, it is important that **(*m* + 1) < *N*** better if (*m* + 1) << *N*.

The least-square fit is the one which minimizes the function

$$S(a_0, a_1, .., a_m) = \sum_{i=0}^{N-1} [y_i - f(x_i)]^2$$

with respect to each a_k :

$$\frac{\partial S}{\partial a_k} = 0, \quad \forall k = 0, 1, ..., m$$

which is the generalization of

$$\frac{\partial \chi^2}{\partial A} = -2 \sum_{i=0}^{N-1} W_i \, (y_i - A - B\, x_i) = 0$$

$$\frac{\partial \chi^2}{\partial B} = -2 \sum_{i=0}^{N-1} W_i \, x_i \, (y_i - A - B\, x_i) = 0$$

to a number *m* + 1 of parameters

Fits. General approach to least – square fitting

The spread of the data about the fitting curve is quantified by the standard deviation, defined as

$$\sigma_y = \sqrt{\frac{S}{N - m}}$$

where *N* − *m* is the **number of degrees of freedom** of the fit.

In the linear case, we have two parameters A and B (*m* + 1 = 2),
so the number of degrees of freedom is always N − 1.

Note that if ***N* = *m*** we have **interpolation**, not curve fitting.

In this case both *S* and *N* − *m* are equal to zero, so σ_y is indeterminate.

Fits. Polynomial least – square fitting

Often the fitting function f (x) is a linear combination of functions:

$$f(x) = a_0\, f_0(x) + a_1\, f_1(x) + \ldots + a_m\, f_m(x)$$

so that eqs. $\frac{\partial S}{\partial a_k} = 0, \quad \forall k = 0, 1, ..., m$ are linear

If f(x) is a **polynomial function**, then

$$f_0(x) = 1, \quad f_1(x) = x, \quad , f_2(x) = x^2, \quad ..., \quad f_m(x) = x^m$$

If the degree of the polynomial is *m*, we have

$$f(x) = \sum_{j=0}^{m} a_j\, x^j$$

where the basis functions are

$$f_j(x) = a_j\, x^j \quad (j = 0, 1, 2, ..., m)$$

The χ^2 minimization becomes

$$\frac{\partial S}{\partial a_k} = -2 \left[\sum_{i=0}^{N-1} \left(y_i - \sum_{j=0}^{m} a_j\, x_i^j \right) x_i^k \right] = 0 \quad (k = 0, 1, ..., m)$$

Fits. Polynomial least – square fitting

Let's rewrite the last eq. of previous slide:

$$\frac{\partial S}{\partial a_k} = -2\left[\sum_{i=0}^{N-1}\left(y_i - \sum_{j=0}^{m} a_j\, x_i^j\right)\, x_i^k\right] = 0 \quad (k = 0, 1, ..., m)$$

Separating the terms that depend on j from those who do not,
and reshuffling a bit, we get:

$$\sum_{j=0}^{m}\left(\sum_{i=0}^{N-1} x_i^j\, x_i^k\right)\, a_j = \sum_{i=0}^{N-1} y_i\, x_i^k$$

Since the a_j are our unknowns and we have a linear combination of the terms in a_j, the above equation can be rewritten as a **system of linear equations** :

$$\boxed{A_{jk}\, a_j = b_k}$$

where

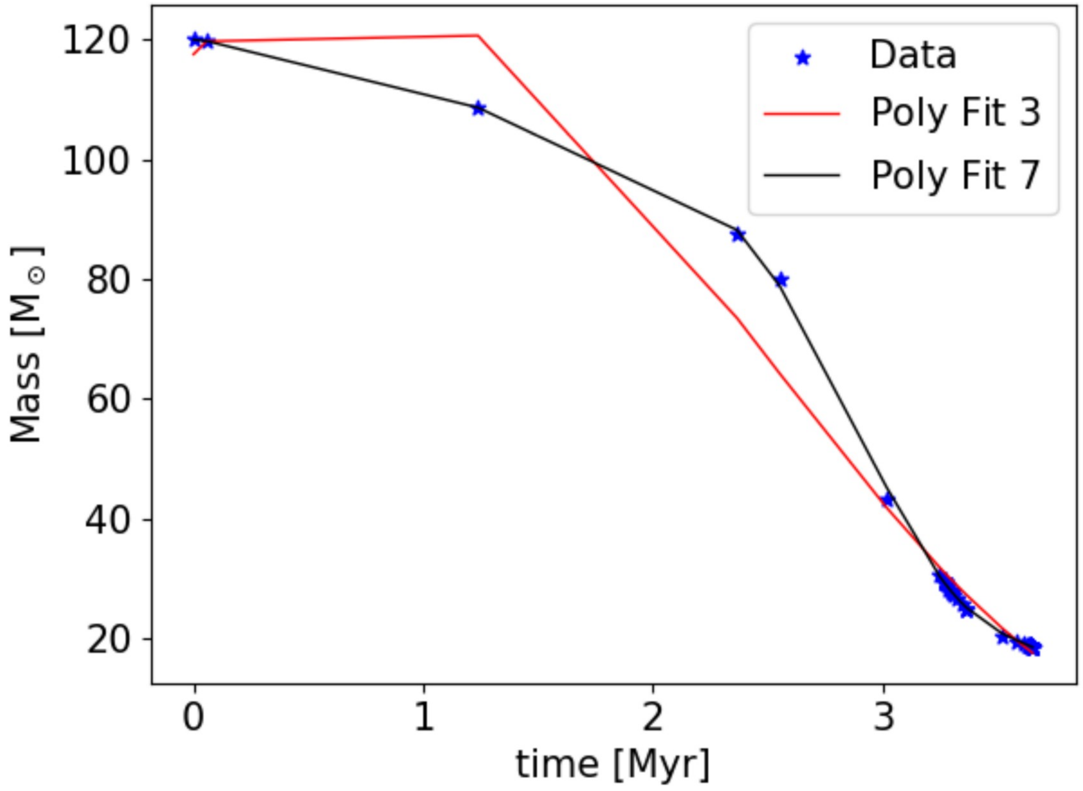
$$\boxed{\begin{aligned} A_{jk} &\equiv \sum_{i=0}^{N-1} x_i^{j+k} \\ b_k &\equiv \sum_{i=0}^{N-1} x_i^k\, y_i. \end{aligned}}$$

These eqs can be implemented into a script to perform a **polynomial fit of degree *m*.**

Fits. Exercise on polynomial least – square fitting

EXERCISE:

Write a script to perform a polynomial fit using equations 293. For the solution of the system of linear equations, use the method you prefer (I used LU decomposition).
Apply this method to fit the data of the file evol_120msun_scattered.dat (the stellar track of a 120 $M_\odot$ star) with a polynomial of the 3rd order. Repeat for a polynomial of 7th order. You should obtain something similar to Figure 66.



Fits. Python functions for fits: scipy.optimize.least_squares( )

There are several possible functions in python that do least-square fitting,
EVEN TOO MANY.

We will see **scipy.optimize.least_squares( )** and **scipy.optimize.curve_fit( )**

scipy.optimize.least_squares : solves a general least-squares problem
given the residuals r(x)

I must start providing a **functional form r(x) of the residuals**.

For example, if I think may **data are distributed according to a Gaussian function** (e.g. I am looking at an emission line in a spectrum), the residuals should be estimated as

$$f(x_i) = a_2 \exp\left[-(x_i - a_0)^2/(2. * a_1^2)\right]$$

$$r(x_i) = f(x_i) - y_i$$

where **$f(x_i)$ is a Gaussian function** evaluated in x_i

y_i is the value of the measure at x_i,

a_0 : = mean, a_1 : = standard deviation, a_2: = normalization

Note: here we are not talking of probability distribution functions,
so a_2 can be whatever normalization

Fits. Python functions for fits: scipy.optimize.least_squares( )

Let’s rewrite the last few lines

$$f(x_i) = a_2 \, \exp\left[-(x_i - a_0)^2/(2. * a_1^2)\right]$$

$$r(x_i) = f(x_i) - y_i$$

where f(x_i) is a Gaussian function evaluated in x_i

y_i is the value of the measure at x_i,

a_0 : = mean, a_1 : = standard deviation, a_2: = normalization

In python formalism this becomes:

```
from scipy import optimize as opt

def fun_res(a,x,y):
   gauss_res=a[2]*np.exp( – (x – a[0])**2/2./a[1]**2) – y
   return gauss_res
```

array of parameters

array of data

```
lsq=opt.least_squares(fun_res, a, args=(x,y), xtol=1e-07, loss='cauchy')
```

Note that I must not specify the arguments of the function fun_res when called by scipy.optimize.least_squares( )

Fits. Python functions for fits: scipy.optimize.least_squares( )

Function scipy.optimize.least_squares( )
has several additional options like

xtol = 1e-07	the tolerance (minimum requested accuracy)
loss = ‘cauchy’	the loss function

g(z), which is used to ``modulate'' the cost function F(x)
we want to minimize:

$$F(x) = 0.5 \text{ sum}(g(r(x)^2))$$

For example, if I choose ‘loss=cauchy',

$$g(r(x)^2) = \ln\left[1 + r(x)^2\right]$$

which weakens the influence of outliers in the data (bad measurements)
but might slow down the optimization (or even make it fail).

Fits. Python functions for fits: scipy.optimize.least_squares( )

The **OUTPUT** of scipy.optimize.least_squares() is a quite **complicated structure of data,** defined in scipy.

Here below, I summarize the most important outputs and how you can get them:

lsq.x

is the **array of the a_j after the optimization**.

In the case of the Gaussian fit, a_0, a_1 and a_2 will contain the best fit values of the mean, the standard deviation and the normalization.

lsq.success

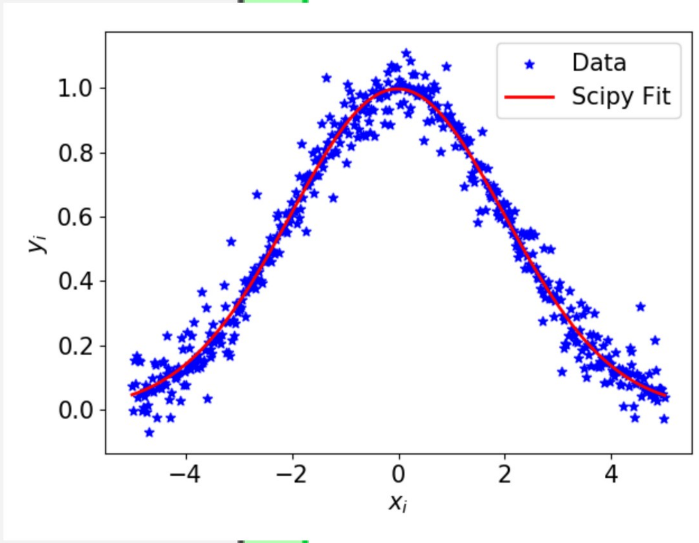
The answer can be only True or False.
False means that the fit failed, True that it succeeded.

Fits. Python functions for fits: scipy.optimize.least_squares( )

EXERCISE:

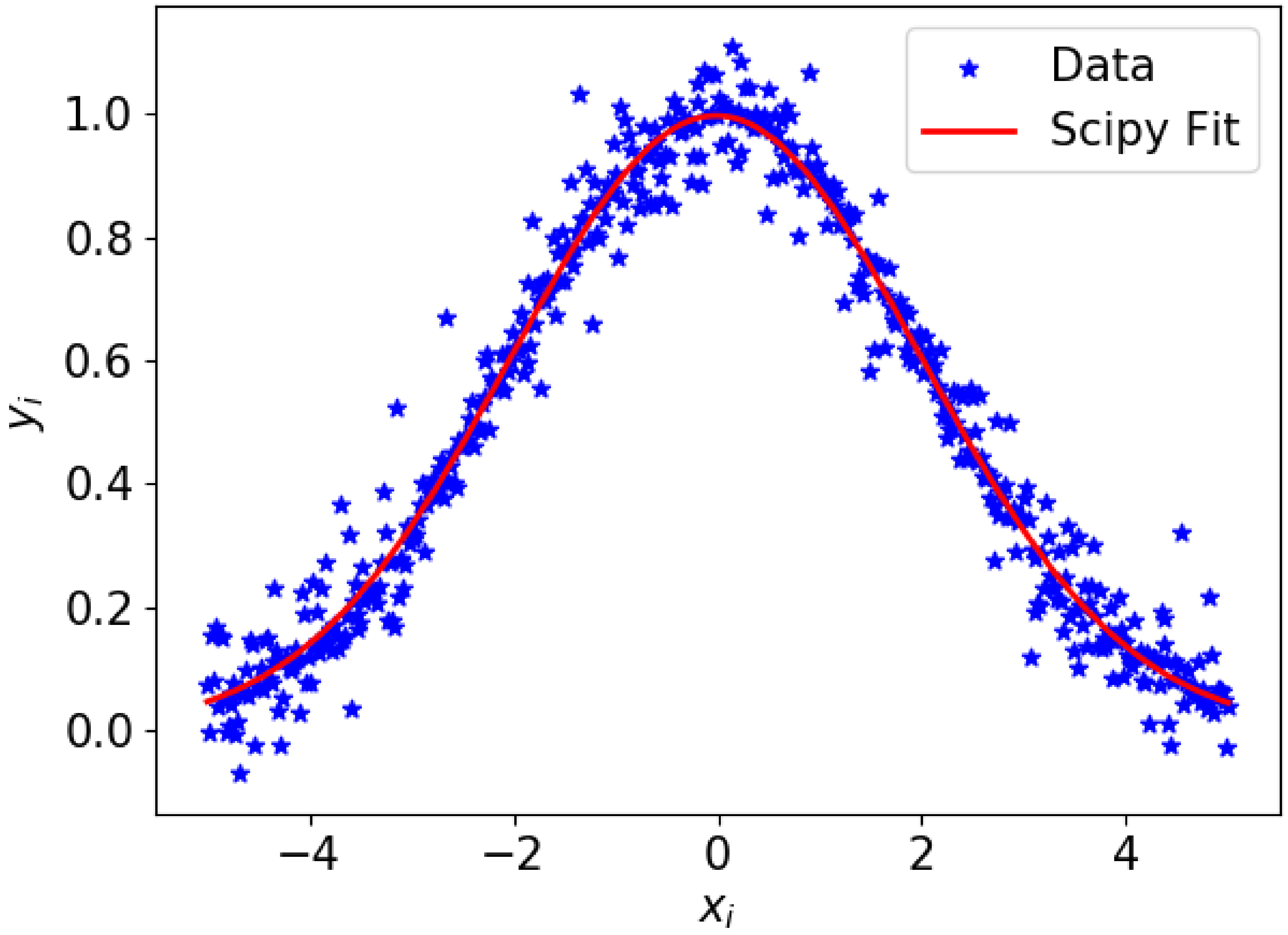
Write a script to perform a least-square fit using **scipy.optimize.least_squares**. Apply it to this set of mock data

```
def data_set():
    a=0.0
    s=2.0
    h=1.0
    x=np.linspace(-5.,5.,num=500) #x data
    y=h*np.exp(-(x-a)**2/(2.*s**2))


    sigma=np.zeros(len(x),float)
    for i in range(len(y)):
        sigma[i]=0.1*rnd.random()
        y[i]=rnd.normal(y[i],sigma[i]) #add gaussian noise

    return x,y
```



Of course, these mock data correspond to a Gaussian with mean, sigma and height equal to 0, 2 and 1, respectively, and with some Gaussian noise. Plot the results. They should look as Figure 67

Fits. Python functions for fits: scipy.optimize.least_squares( )



Fits. Python functions for fits: scipy.optimize.curve_fit( )

scipy.optimize.curve_fit : solves a general least-squares problem to fit a function, f(x), to the data.

The main (but not the only) difference with respect to scipy.optimize.least_squares is that scipy.optimize.curve_fit **requires the function in input, not the residuals**.

For example, if I think may data are distributed according to a Gaussian function, scipy.optimize.curve_fit( ) can be used as

```
from scipy import optimize as opt

def func(x,a,s,h):
    gauss=h*np.exp( – (x – a)**2/2./s**2)
    return gauss

popt,pcov=opt.curve_fit(func ,x ,y ,p0=(a ,s ,h ))
```

$$\begin{tikzcd}
\text{Fitting function} & \text{array of x data} & \text{array of y data} & \text{parameters} \\
\text{func} \arrow[u] & x \arrow[u] & y \arrow[u] & (a ,s ,h) \arrow[u]
\end{tikzcd}$$

Additional possible input parameters:
sigma: the array of uncertainties on the data y
bounds: the lower and upper bounds on parameters (default is no bounds)
methods: the method used by the function to do the optimization.

Fits. Python functions for fits: scipy.optimize.curve_fit( )

The OUTPUT of scipy.optimize.least_squares is composed of two arrays:

popt: contains the optimal values for the **best-fitting parameters**
i.e. these for which the sum of the squared residuals of f(x, *popt) – y is minimized.

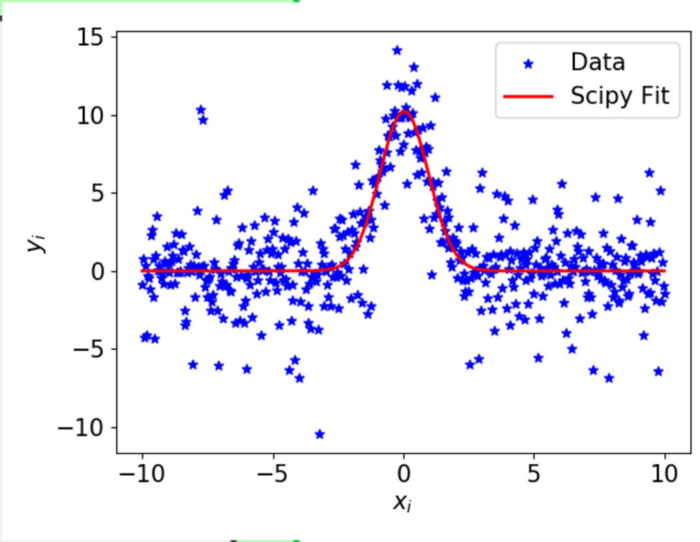
pcov: is the estimated **covariance of popt**.
The diagonals provide the variance of the parameter estimate.
To compute **one – standard deviation errors on the parameters** use
perr = np.sqrt(np.diag(pcov))

Fits. Python functions for fits: scipy.optimize.curve_fit( )

EXERCISE:

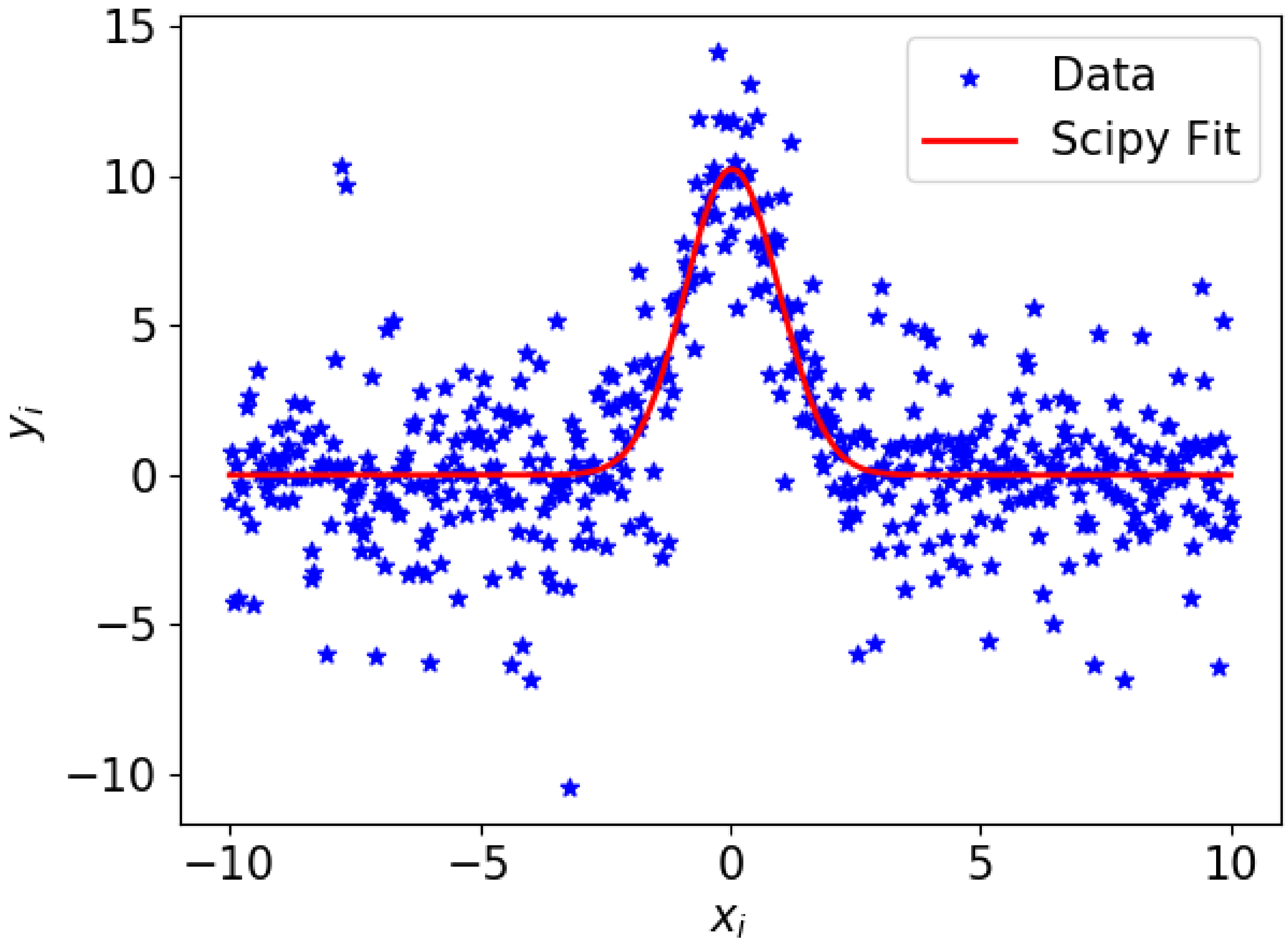
Write a script to perform a least-square fit using **scipy.optimize.curve_fit**. Apply it to this set of mock data

```
def data_set():
    a=0.
    s=1.
    h=10.

    x=np.linspace(-10.,10.,num=500) #x data
    y=h*np.exp(-(x-a)**2/2./s**2)

    sigma=np.zeros(len(x),float)
    for i in range(len(y)):
        sigma[i]=4.*rnd.random()
        y[i]=rnd.normal(y[i],sigma[i]) #add gaussian noise

    return x,y
```



Of course, these mock data correspond to a Gaussian with mean, sigma and height equal to 0, 2 and 1, respectively, and with some Gaussian noise. Plot the results. They should look like Figure 68

Fits. Python functions for fits: scipy.optimize.curve_fit( )



Fits. Python functions for fits: scipy.optimize.curve_fit( )

optimize.curve_fit( ) can be also used to do fits in multi-dimensions.

For example, let us see now the example of a **Gaussian in two dimensions**.

Let's write down a **Gaussian in two dimensions** oriented along x and y axis:

$$f(x,y) = h\ \exp\left\{-\left[\frac{(x-x_0)^2}{2\,\sigma_x^2} + \frac{(y-y_0)^2}{2\,\sigma_y^2}\right]\right\}$$

here h is the height, x0 and σx are the mean and the standard along x,
y0 and σy are the mean and the standard deviation along y

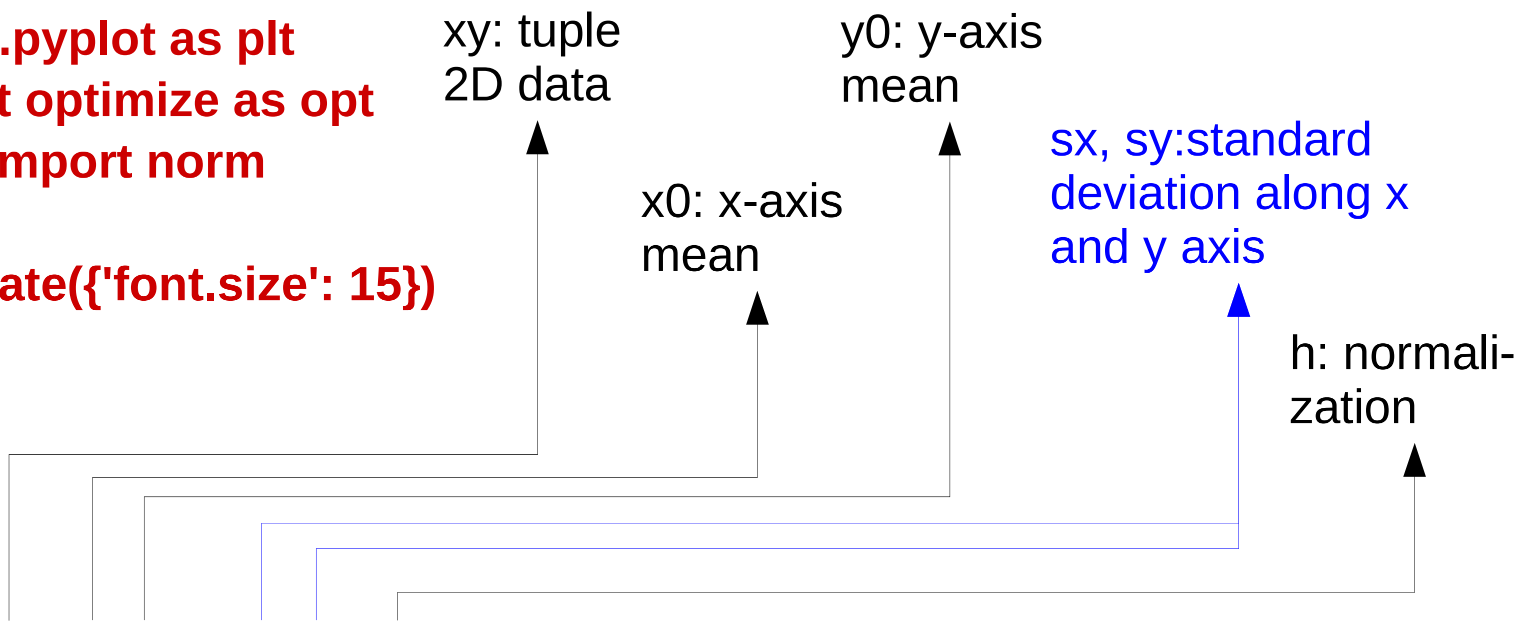
In python

```
def twoD_gauss(xy, x0, y0, sx, sy, h):
    z=h*np.exp( – ((xy[0] – x0)**2/(2.*sx*sx) + (xy[1] – y0)**2/(2.*sy*sy)))
    return z.ravel()
```

Fits. Python functions for fits: scipy.optimize.curve_fit( )

In python:

```
#examples/fit/fit_2D_scipy.py
#fit a 2D gaussian with scipy.optimize.curve_fit
import numpy as np
import numpy.random as rnd
import matplotlib.pyplot as plt
from scipy import optimize as opt
from scipy.stats import norm

plt.rcParams.update({'font.size': 15})
```



```
def twoD_gauss(xy, x0, y0, sx, sy, h):
    z=h*np.exp( – ((xy[0] – x0)**2/(2.*sx*sx) + (xy[1] – y0)**2/(2.*sy*sy)))
    return z.ravel()
```

z is a matrix, while curve_fit only takes 1D arrays as argument.
Hence, we must first convert z to a 1D array thanks to **numpy.ravel()**

Fits. Python functions for fits: scipy.optimize.curve_fit( )

```
def data_set():
    x=np.linspace(-5.,5.,num=500) #x data
    y=np.linspace(-5.,5.,num=500) #x data
    x,y=np.meshgrid(x,y)
    z=np.zeros([len(x),len(y)],float)
    x0=0.0
    y0=0.0
    sx=1.0
    sy=0.7
    h=1.0
    xy=(x,y)
    z=twoD_gauss(xy,x0,y0,sx,sy,h)
    z=z + 1e-1*np.random.normal(size=z.shape)
    return x,y,z

def sci_fit(x,y,x0,y0,sx,sy,h):
    xy=(x,y)
    popt,pcov=opt.curve_fit(twoD_gauss ,xy ,z ,p0=(x0,y0,sx,sy,h))
    return popt,pcov
```

Input mock data distributed according to a 2D Gaussian

2D Gaussian function

xy data

z data

Fit parameters:
x0, y0: means along x and y axis

sx, sy: std along x and y axis

h: normalization

Fits. Python functions for fits: scipy.optimize.curve_fit( )

```
#main
x,y,z=data_set() #data points
x0=0. #Ansatz for parameters x0, y0, sx, sy and h
y0=1.
sx=2.
sy=2.
h=2.

popt,pcov=sci_fit(x,y,x0,y0,sx,sy,h)

perr = np.sqrt(np.diag(pcov))

print(popt[0],perr[0],popt[1],perr[1],popt[2],perr[2],popt[3],perr[3],popt[4],perr[4])
#x0+err,y0+err,sx+err,sy+err,h+err

xy=(x,y)
data_fitted = twoD_gauss(xy, *popt) #fitted data
```

Call scipy.optimize.curve_fit()

Print best – fit parameters and their uncertainties

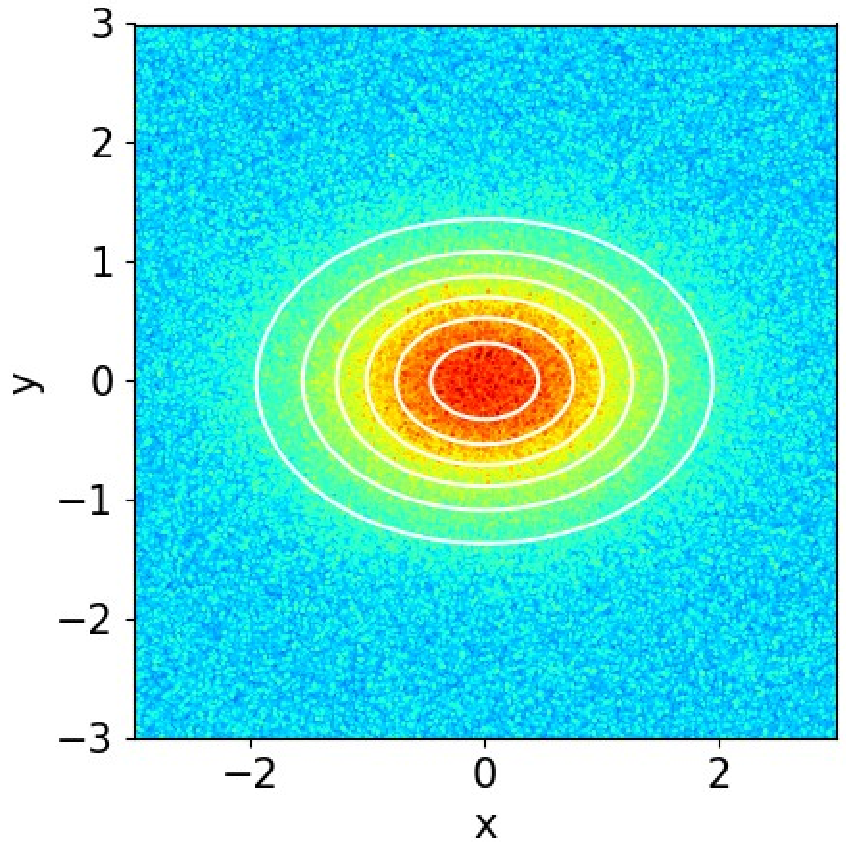
Print values of z fitted

***popt** : The asterisk here has the meaning of calling a **pointer to the array popt**, which contains the best fitted parameters. This is a pointer **like in C and C++**. This **does not exist in python, just in scipy** and is not commonly used

Fits. Python functions for fits: scipy.optimize.curve_fit( )

```
#plot results
figsize=plt.figaspect(1.0)
fig, ax =  plt.subplots(1, 1, figsize=figsize)
ax.set_xlim(-3,3)
ax.set_ylim(-3,3)
ax.contourf(x,y,z.reshape(len(x),len(y)),100,cmap=plt.cm.jet)
ax.contour(x, y, data_fitted.reshape(len(x),len(y)), 8, colors='w')
ax.set_xlabel("x")
ax.set_ylabel("y")

plt.tight_layout()
plt.show()
```

We need to transform z to a matrix again with function reshape()



3
2
1
0
−1
−2
−3
y
−2
0
2
x