Extra: Fitting and the covariance matrix

Least-squares fitting and why we use the covariance matrix

In real measurements, there is always noise. We usually assume that this noise is Gaussian-distributed and that noise on points is independent. This is called **Gaussian White Noise**.

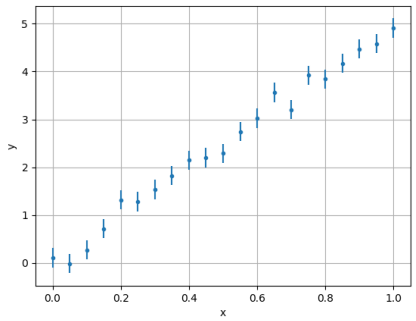
When fitting a function with free-parameters, we want to **maximize the probability of the fit parameters** being correct. i.e. what is the probability of the parameters given the data?

See Data Analysis (A Bayesian Tutorial) by Sivian and Skilling for a full description.

In general, we have a probability *density* for each free parameter.

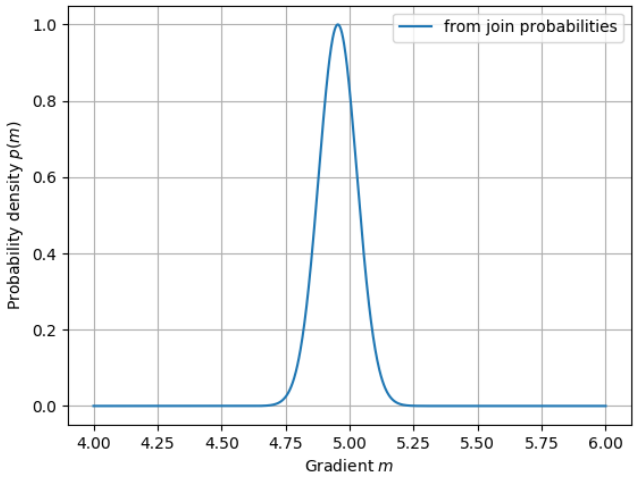
Trivial example $y = mx$:

```
import numpy as np
import matplotlib.pyplot as plt
randn = np.random.randn

def f(x,m): # equation with one free parameter
    return m*x

sigma_y = 0.2
x_data = np.linspace(0,1,21)
y_data = f(x_data,5) # simulate data with m = 5
y_data += sigma_y * randn(len(y_data)) # add modest noise to each point

plt.errorbar(x_data,y_data,yerr=0.2,fmt='.'); plt.grid()
plt.xlabel('x'); plt.ylabel('y')
plt.savefig('./figs/noisy_y_equals_mx.png'); plt.close('all')
```



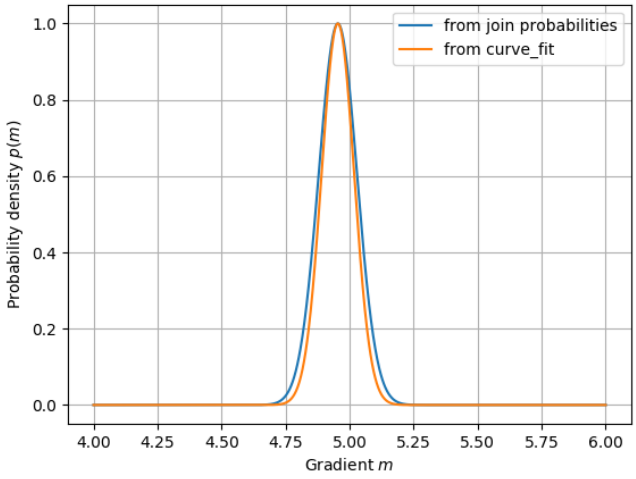
We can now find the probability density for the gradient m directly by considering the joint probability of all the points for each value of m. i.e. $p(m) \propto \Pi_i \exp(-(y_i - mx_i)^2/2\sigma_y^2)$.

```
m_axis = np.linspace(4,6,1001)
prob_m = list()
for m in m_axis:
    Delta_y = f(x_data, m) - y_data # vertical distance for each point given this value of m
    probabilities = np.exp(-Delta_y**2/(2*sigma_y**2)) # prob each point given this value of m
    prob_m.append(np.product(probabilities))           # joint prob for all pts given val of m

Normalisation = max(prob_m)
prob_m = [p/Normalisation for p in prob_m]
plt.plot(m_axis,prob_m,label='from join probabilities')
plt.xlabel(r"Gradient $m$"); plt.ylabel(r"Probability density $p(m)$")
plt.legend(); plt.grid()
plt.savefig("./figs/prob_m.png")
```



This is what `curve_fit` achieves in a much faster way, assuming all the noise is Gaussian.

```
from scipy.optimize import curve_fit

opt,cov = curve_fit(f, x_data, y_data, p0=4)

m_best = opt[0]
m_sigma = np.sqrt(cov[0,0])

plt.plot(m_axis, np.exp(-(m_axis-m_best)**2/(2*m_sigma**2)),label='from curve_fit')
plt.legend()
plt.savefig("./figs/curve_fit_m.png"); plt.close('all')
```



(Maximizing this probability, since the propbability for each point depends on $(y_i - f(x_i))^2$, is the same as minimizing the sum of the squares. This is where "least squares fitting" comes from. If noise is not Gaussian, or points are not independent, then proceed with caution!)

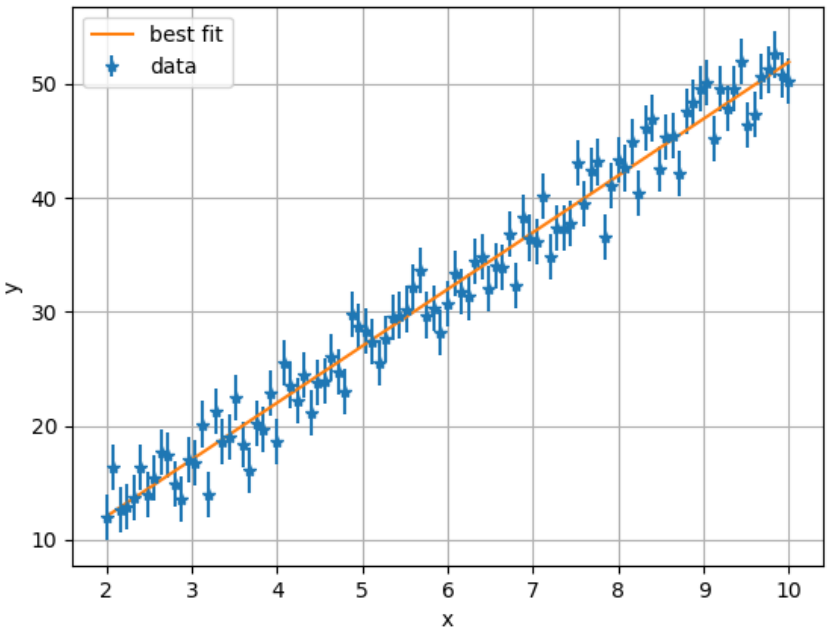
This **generalises to multiple free-parameters**. Instead of a 1D Gaussian, we get a multi-variate Gaussian.

The properties of this multi-variate Gaussian are capture in the covariance matrix, which is a generalisation of the width of a Gaussian. The diagonals are the (squares of) the width in each dimension; the off-diagonals capture correlations.

Covariance matrix and a 2D example

Simulate noisy data based on a linear curve with an offset ($y = mx + c$) and fit the best-fit with `curve_fit`:

```
def f(x,m,c):
    return m*x+c

actual = (5,2) # actual values of m and c
sigmay = 2     # noise per point for this simulation
N = 101        # number of data points to simulate
xmin,xmax = 2,10 # region over which to work

x_data = np.linspace(xmin,xmax,N)
y_data = f(x_data,*actual) + randn(N)*sigmay # simulate noisy data

from scipy.optimize import curve_fit
guess = (3,2)
best,covariance = curve_fit(f,x_data,y_data,p0=guess)


x_axis = np.linspace(xmin,xmax,1001)
y_best = f(x_axis,*best)
plt.errorbar(x_data,y_data,yerr=sigmay,fmt='*',label='data')
plt.plot(x_axis,y_best,label='best fit')
plt.xlabel('x'); plt.ylabel('y'); plt.legend(); plt.grid()
plt.savefig("./figs/noisy_y_equals_mx_plus_c.png"); plt.close('all')
```

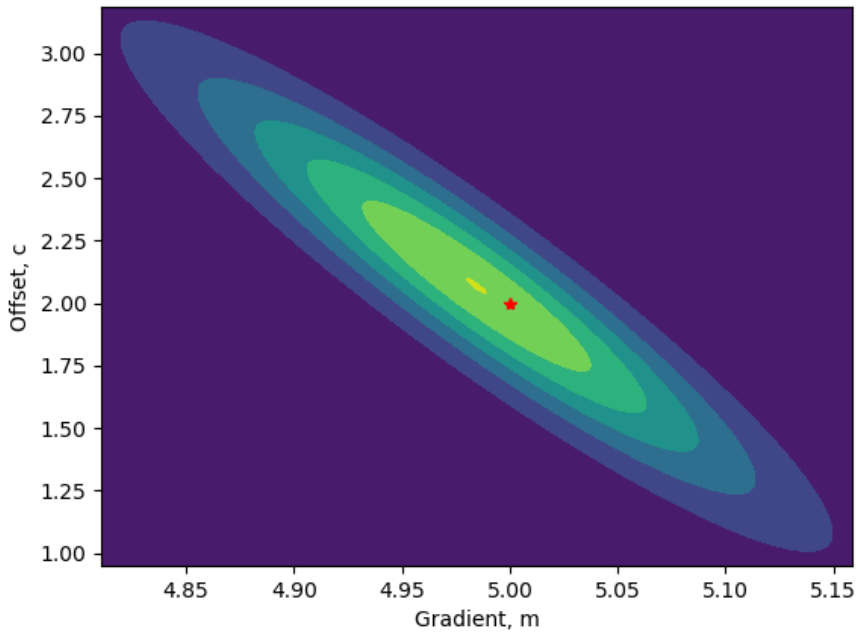


Here's the covariance found by `curve_fit`:

```
covariance

[[ 0.00757097 -0.04542585]
 [-0.04542585  0.31374118]]
```

Plot the 2D Gaussian represented by the covariance matrix from `curve_fit`:

```
def G(X,X0,Sigma):
    """Multi-variate Gaussian; see wikipedia.org/wiki/Multivariate_normal_distribution"""
    from numpy import exp, sqrt, pi
    from numpy.linalg import det, inv
    k = len(X)
    DX = X - X0
    return exp(-0.5*DX.transpose().dot(inv(Sigma).dot(DX)))/sqrt((2*pi)**k*det(Sigma))

sigma = np.sqrt(covariance.diagonal())
ms = np.linspace(best[0]-2*sigma[0],best[0]+2*sigma[0],300)
cs = np.linspace(best[1]-2*sigma[1],best[1]+2*sigma[1],300)

A = np.zeros((len(ms),len(cs)))
for nm,m in enumerate(ms):
    for nc,c in enumerate(cs):
        X = np.array([m,c])
        A[nm,nc] = G(X,best,covariance)

plt.contourf(ms,cs,A.transpose())
plt.plot(actual[0],actual[1],'r*') # mark the true value (known only because we simulated it)
plt.xlabel('Gradient, m'); plt.ylabel('Offset, c')
plt.savefig("./figs/gaussian_covariancy_y_equals_mx_plus_c.png"); plt.close('all')
```



In this example, it is very unlikely that m is over-estimated and c is simultaneously also over-estimated. If m is over-estimated, it's more likel that c is **under**-estimated. This sort interdependence of fit parameter estimation is not captured by σ_m and σ_c independently.

The covariance matrix has the structure:

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_m^2 & \sigma_{mc} \\ \sigma_{mc} & \sigma_c^2 \end{pmatrix}.$$

The square roots of the diagonals are the uncertainties, and the off-diagonals represent the correlations. In this case, σ_{mc} is negative (anti-correlated).