Week Four - Parameter Estimation and Fitting Data

Outline

Prelude

In one sense, all of machine learning including AI (whatever that difference is) is really about applying some function to your data and finding the "best" values for the parameters in your functions. This could be the weights in the links in the neural network (the function here is not in a standard form we know) or,as you have done before, it could be fitting a curve to data (as in the the polynomial examples below). The terminology may be different, the measures we use to define what is the "best" fit may be new, some methods (neural nets) are only just coming into their own while hardware improvements open up new options, but really this has been going on for centuries.

This week we start with curve fitting as the classic example of machine learning. In future weeks we will be looking at different types of problem but the basic principles are the same.

Before you start

This week we will be using the `iminuit` Python library to fit curves to data. This is the python interface for a fast minimiser based on a `C++` library maintained by CERN. It is comprehensive (unlike `scipy.optimize.curve_fit`) so for any scientist it is well worth the extra effort.

The `iminuit` library does not usually come installed with python. On the PC's in the Physics computing lab, you should find it is present and working. If you are on your own machine, please read the `iminuit` installation instructions. I found it worked first time on my Windows machines but we had trouble with permissions on a directory used for temporary files in the Physics lab installation.

So the first thing you should do once you have installed it is to run the next code cell which acts as a quick check that `iminuit` is installed and running on your machine.

In [1]:
```
#Check iminuit
import iminuit
print("iminuit version is "+str(iminuit.__version__))

from iminuit.cost import LeastSquares
print(LeastSquares)
```

```
iminuit version is 2.24.0
<class 'iminuit.cost.LeastSquares'>
```

Section One: Introduction to Estimation ^

In physics, it is common that we have a theory for a process which contains some parameters whose values are not known. For example Newton's law of gravity has one unknown constant G while the standard model of particle physics has over twenty. So we do some experiment to try to determine if the theory is a good fit (see goodness of fit last week) and then find the best value for the parameters in our theory. For example, in particle physics there are many experiments we can do to see if the standard model with our current best parameter values (a null model) is consistent with data and at the same time theorists have many suggestion for new physics which leads to a new model of particle physics (an alternative hypothesis) and we need to know the parameters in the alternative model. The data taken in experiments such as those at the LHC is used to look at which theory is the best fit (last week) and to make estimates of these parameters (this week).

Determining the values of these model parameters is referred to as **estimation** and is one of the most important applications of statistics in science. Last week, we saw that hypothesis tests aim to answer the question "Is the particular parameter value consistent with the data?". This

week, we instead want to answer "What value of the parameter is *most* consistent with the data?".

Such an estimate is called the **best estimate** of the parameter, and as you would expect typically corresponds to regions where the PDF is largest.

There are many ways we can estimate a parameter value from a set of data we have measured. The objective is to find some function M of the measurements X_i that gives an estimate of the value of some parameter θ,

$$\hat{\theta} = M(X_1, \cdots, X_N)$$

Specific methods of estimation specify the function M. Different parameter estimation methods are used in different situations and we will cover two major methods in this workbook.

Notation: that we denote the estimate of a parameter with a "hat" as $\hat{\theta}$ to distinguish it from the true but unknown value of that parameter θ. This is because our estimate $\hat{\theta}$ is a function of random variables so our estimate $\hat{\theta}$ is itself a random variable.

Example

Let us consider a linear relationship where $y = X$ for a given set of values for the independent variable X while we add some Gaussian noise to the values of the dependent variable y to simulate experimental noise. We want to fit this data with a straight line, with equation $y = mX$. Note we will focus on just one parameter for now. We can try several different values of the parameter m by hand and just by eye judge which line fits the data best. We will generate this data using `numpy.random`:

In [2]:
```
import numpy as np
np.random.seed(1) # Set the random seed for consistency

X = np.linspace(0,10,20) # For simplicity these X values are not chosen randomly (they could have been) but are equally spaced

y = X + np.random.default_rng().normal(loc = 0, scale = 0.5, size = len(X)) # Add Gaussian noise, mean 0, std.dev 0.5
```

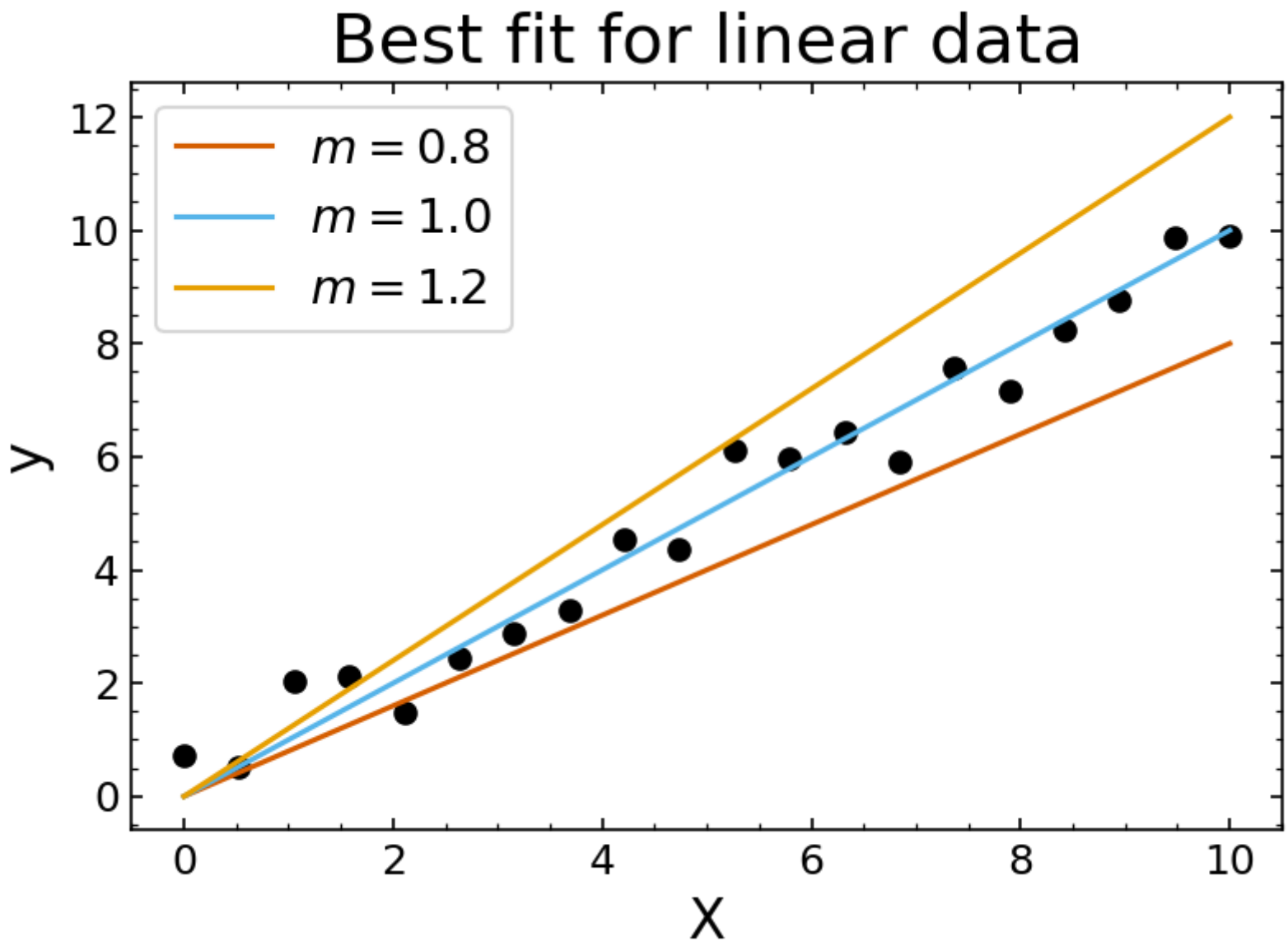
Let us consider 3 possible values of m: 0.8, 1, and 1.2. The three straight lines with these parameter values are shown against the data in the plot below. Note each measurement is a single data point and we have no information on the uncertainty in the measurement from the data so we can not add error bars to points.

In [3]:
```
import matplotlib.pyplot as plt
from matplotlib.ticker import MultipleLocator

gradients = [0.8,1,1.2] # Define gradient parameters
```

```
colors = ['#D55E00','#56B4E9','#E69F00'] # Define colours
fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.scatter(X,y,color='black')
for m,c in zip(gradients,colors):
    ax.plot(X,m*X,c=c,label='$m = {:.1f}$'.format(m))

ax.set_xlabel('X',fontsize = 16)
ax.set_ylabel('y',fontsize = 16)
ax.legend(loc='upper left',fontsize = 14)
ax.tick_params(which='both',labelsize = 12,top=True,right=True,direction='in')
ax.xaxis.set_major_locator(MultipleLocator(2))
ax.xaxis.set_minor_locator(MultipleLocator(0.5))
ax.yaxis.set_major_locator(MultipleLocator(2))
ax.yaxis.set_minor_locator(MultipleLocator(0.5))
ax.set_title('Best fit for linear data',fontsize = 20)
```

Out[3]:

```
Text(0.5, 1.0, 'Best fit for linear data')
```



Is it clear that $m = 1$ (the true underlying form) is the best choice of parameter? Try another seed to see if your answer changes.

Maybe it is clear on your computer, maybe not. For me, with random seed zero, I found slightly more circles intersected the $m = 0.8$ line than the $m = 1.0$ line while few were on the $m = 1.2$ line. Most of the issue was at low X values (EFS why?). Repeating with seed=1 the $m = 1.0$ line was the best but still very unclear at low X.

Of course, in general we can't practically test different parameter values visually to fit your data, and in science what we really want to extract is the specific value of the fit that best describes the data. This is where we need other tools to estimate our parameters.

Good estimates

Many methods of estimation exist, where some are better than others. Good estimates are:

1. Consistent: in the limit of large N, $\hat{\theta} \to \theta$ i.e. the estimate converges to the true value
2. Unbiased: $E(\hat{\theta}) = \theta$ i.e. the expectation of the estimate equals the true value
3. Efficient: $V(\hat{\theta})$ is small, i.e. the variance in the estimate is small.

The expectation value $E(\ldots)$ and the variance $V(\ldots)$ shown here come from where we imagine that same identical experiment is done by many different people. The expectation and variance is found by looking at how the results vary from person to person.

The expectation value $E(\ldots)$ and the variance $V(\ldots)$ referred to above are theoretical results in the sense that if we really had repeated the experiment N times, we would combine all the data to get better estimates of the parameters.

However you could imagine that the experiment is to measure Newton's constant of gravity G which is done by a lab full of N physicists. Each student i hands in their lab book with their estimate of G_I and does not improve this with the results obtained by other students. The HoL (Head of Lab) could do an analysis which should show the mean of all the student's results was closer to the best current estimate of G. More interestingly, by using all the student results, the HoL could estimate the standard deviation $\hat{\sigma}$ for the experiments (again, not the true value but an estimate). Now the HoL can look to see if there is any student whose result is so far off it can not be explained by statistical uncertainty, i.e. something went wrong with that experiment.

Example: finding the mean of data is an estimate

One of the most common methods of estimation that you may not have realised is formally an estimate: finding the mean of a set of measurements.

$$\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} X_i$$

In this example the function M is the sum of the measurements, divided by the number of measurements made. We have put a hat on the symbol "mu" used here for the mean. That is because if the X_i are random variables then their sum has some probability distribution which has a precise mean, denoted μ without a hat. The mean of the data is being used to estimate the true mean μ of the underlying distribution.

The method of estimating the mean that we have shown above is consistent, unbiased and efficient, its is a good estimator. However, it is obviously only applicable for finding the mean, rather than any parameter in general. This workbook will discuss two different methods of more generally estimating parameters of distributions or functions from data; namely, using the chi-squared and maximum likelihood estimators. The former is typically best applied for measurements made of variables with at least approximately Gaussian distributions, whereas the maximum likelihood method can be applied for any probability distribution.

Both of these methods involve finding parameter values to maximise or minimise some function, and are referred to as **fitting**.

Summary

In this section, we have introduced the concept of parameter estimation, including:

- Best estimates of parameters
- Properties of good estimates

In the following section, we will discuss the concept of fitting: optimising some function to find the best possible parameter values for our data.

Section Two: Fitting ^

In order to find the best estimates of parameters, one of the most common methods is fitting. To do this, we define some **cost function** (often called a **loss function** in machne learning) of our data and our proposed model (and associated parameters) and we aim to maximise or minimise this function, depending on the specific choice of function. You will see this concept again many times as we discuss machine learning in the rest of this course.

As previously mentioned, we will discuss two different methods with different cost functions; we will start with the most familiar, the chi-squared method.

The chi-squared estimation method

Recalling last week, we discussed the χ^2 value as a method for determining the goodness of fit for a given model to some experimental data, where we hypothesise the data is distributed with some Gaussian noise about the model prediction and test whether the measurements are consistent with that hypothesis.

However, we can take this concept one step further, and instead try to optimise the value of χ^2 by choosing different parameter values θ (for simplicity we assume one parameter for now). We will assume that we want the smallest value of χ^2 (but see warnings about problems when $\chi^2 \ll N_{\text{dof}}$) so

- Consider some data $\{X_i\}$ that we hypothesise is modelled by some function $f(X_i;\theta)$ with one parameter θ
- By finding a minimum of the χ^2 with respect to θ, we can choose the best estimate $\hat{\theta}$ to describe our data
- To minimise χ^2 with respect to θ, we require:

$$\frac{\partial \chi^2}{\partial \theta} = 0$$
$$\frac{\partial^2 \chi^2}{\partial \theta^2} > 0$$

where the first condition requires that we are at a stationary point of the χ^2 function with respect to θ, and the second condition requires that said stationary point is a minimum.

We can apply the same setup for a model with a set of multiple parameters $\{\theta_j\}$, where we require that

$$\frac{\partial \chi^2}{\partial \theta_j} = 0$$
$$\frac{\partial^2 \chi^2}{\partial \theta_j^2} > 0$$

for all j. For N parameters, we must solve these $2N$ equations simultaneously to find our best estimate of each parameter, where the best estimate of parameter θ_j is denoted as $\hat{\theta}_j$.

We will discuss how we can actually solve this optimisation problem in general later in this workbook.

The maximum likelihood estimation method

In some experiment, we may know that our results should be distributed according to some probability density function $f(X;\theta)$ but with some unknown parameter θ. In this scenario, we know the functional form of the PDF but not the specific parameter value. We want to do some experiment to determine the value of θ.

After we make a series of N measurements, we have some data points X_i that are samples from whatever probability distribution underlies the quantity we are measuring. We can therefore define the total probability of our measurements as the product of the probablity of each individual measurement, i.e. as

$$L(\theta;X_i) = \prod_{i=1}^{N} f(X_i;\theta),$$

where $L(\theta;X_i)$ is the **likelihood function** for our experiment. This is very similar to the likelihood parameter we saw in Bayes theorem, namely it is the probability that we make the measurements we have made for a specific value of the parameter θ. Because our data values X_i are given, they are fixed, the likelihood is only a function of the parameter θ. It is telling us how likely it is that θ is the correct parameter value given the measurements made $\{X_i\}$. As a result, we will often write the likelihood function as $L(\theta)$ to emphasise this fact.

Note: the likelihood function is *not* a PDF for θ, and is only proportional to the probability of θ being the correct value given the measurements X_i, so $\int L(\theta;X_i)d\theta \neq 1$ in general.

Example

For example, let us consider an experiment where we have a bag of coloured balls. We pull a ball out, note the colour and then return it to the bag. We do this ten times (ten "trials") and the colour of the balls is r,B,r,r,B,r,r,r,B,r where r indicates a red ball and B is a blue ball. Here X_i is the colour of the ball we pull out on the i-th attempt so $X_1 = r$ and $X_9 = B$. We want to estimate the probability p of pulling out a blue ball on any one attempt, i.e. the fraction of blue balls in the bag.

The obvious model to use is a binomial model as there are only two outcomes we are interested in: blue ball or not a blue ball. We will call our parameter p here. Our $f(X,p)$ is the probability (not a probability density) that if there are a fraction p blue balls in the bag then we get result $X = B$ with probability p and result $X = \tilde{B}$ (not B) with probability $(1-p)$. That is

$$f(X;p) = \begin{cases} p & \text{if } X = B \quad (1) \\ (1-p) & \text{if } X = \tilde{B} \quad (2) \end{cases}$$

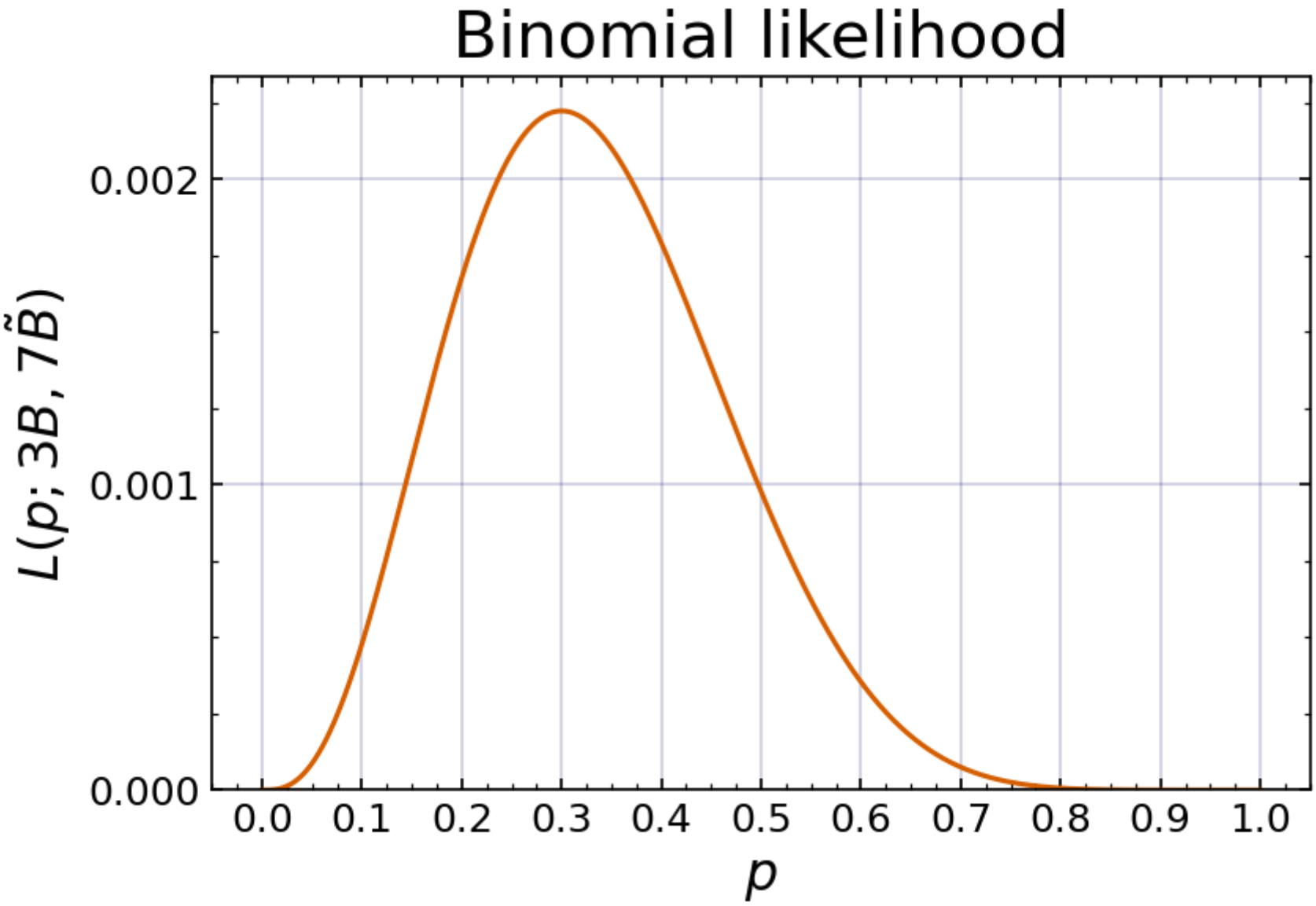
So in our case the likelihood is given as

$$\begin{aligned} L(\theta;\boldsymbol{X}) \equiv L(p;(r,B,r,r,B,r,r,r,B,r)) &= f(X_1;p)f(X_2;p)\dots f(X_{10};p) && (3)\\ &= (1-p).p.(1-p).(1-p).p.(1-p).(1-p).(1-p).p.(1-p) && (4)\\ &= p^3(1-p)^7 && (5) \end{aligned}$$

The cell below plots this likelihood, as a function of p.

In [4]:
```
def L(p):
    return p**3*(1 - p)**7

p = np.linspace(0,1,1000)
L_p = L(p)

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.plot(p,L_p,color='#D55E00')
ax.set_xlabel('$p$',fontsize = 16)
ax.set_ylabel(r'$L(p; 3B, 7\tilde B)$',fontsize = 16)
ax.tick_params(which='both',labelsize = 12,top=True,right=True,direction='in')
ax.xaxis.set_major_locator(MultipleLocator(0.1))
ax.xaxis.set_minor_locator(MultipleLocator(0.025))
ax.yaxis.set_major_locator(MultipleLocator(0.001))
ax.yaxis.set_minor_locator(MultipleLocator(0.00025))
ax.set_ylim(bottom = 0)
ax.set_title('Binomial likelihood',fontsize = 20)
ax.grid(color='xkcd:dark blue',alpha =0.2)

# Lazy check to show this liklihood for p is not normalised, you can do this integral exactly!
import scipy.integrate as integrate
result = integrate.quad(L, 0, 1.0)
print("Integral of L(p) between 0 and 1 = "+str(result[0]))
```

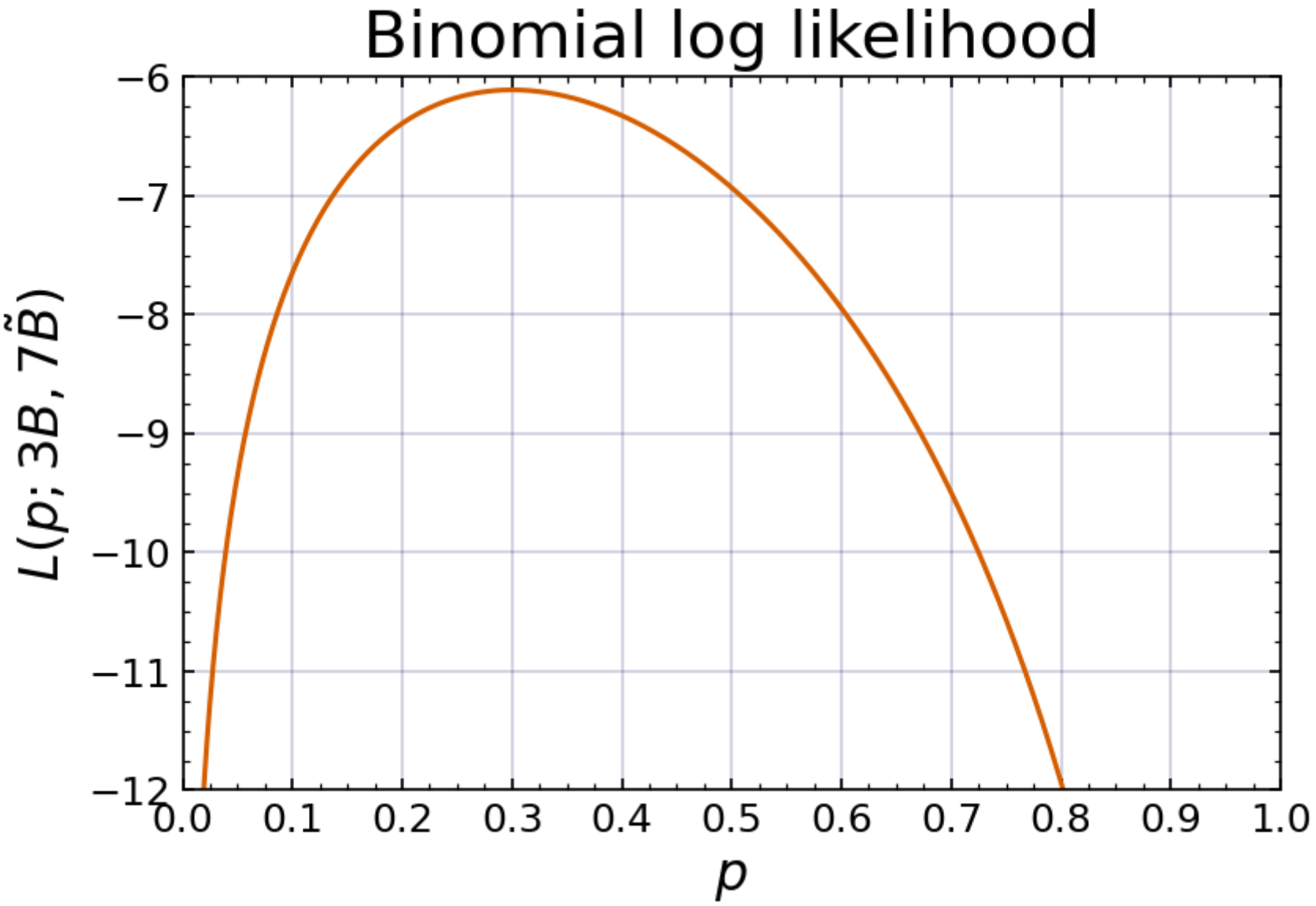
```
Integral of L(p) between 0 and 1 = 0.0007575757575757576
```



Our common sense suggests that our estimate for p, denoted $\hat{p}$, would equal the frequency of blue balls found in our experiment, that is $\hat{p} = 3/10 = 0.3$. We can now see that our more sophisticated likelihood gives exactly this results, namely that the liklihood function peaks at 0.3. The value of p that gives the maximum likelihood, which therefore maximises the probability of observing the data we have measured, gives a sensible estimate for p.

In practice, we will take logarithms of likelihood functions to produce the **log-likelihood**, because this is more numerically stable for computing values, particularly for binned measurements with many bins. Because the logarithm is a monotonically increasing function, the maximum of $L(\theta)$ is at the same value of θ as the maximum of $\ln(L(\theta))$ and so we can use $\ln(L)$ for all calculations. The code cell below plots the log-likelihood for the binomial example.

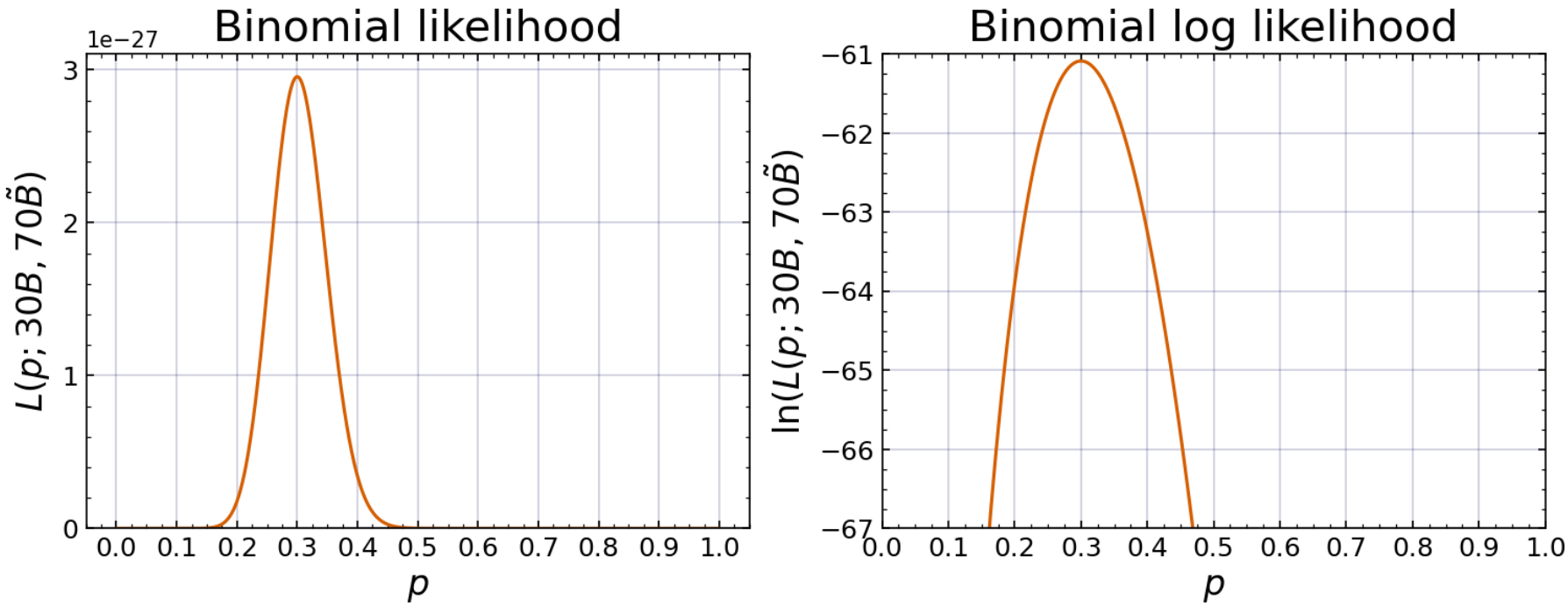
In [5]:
```
p2 = np.linspace(0.01,0.82,1000) # limit range to avoid log(0.0)
L_p2 = L(p2)

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.plot(p2,np.log(L_p2),color='#D55E00')
ax.set_xlabel('$p$',fontsize = 16)
ax.set_ylabel(r'$L(p; 3B, 7\tilde B)$',fontsize = 16)
ax.tick_params(which='both',labelsize = 12,top=True,right=True,direction='in')
ax.xaxis.set_major_locator(MultipleLocator(0.1))
ax.xaxis.set_minor_locator(MultipleLocator(0.025))
ax.yaxis.set_major_locator(MultipleLocator(1))
ax.yaxis.set_minor_locator(MultipleLocator(0.25))
ax.set_xlim(0.0,1.0)
ax.set_ylim(-12,-6)
ax.set_title('Binomial log likelihood',fontsize = 20)
ax.grid(color='xkcd:dark blue',alpha =0.2)
```



EFS: What if in our ten trials we found r,B,g,g,B,r,g,r,B,r where "g" represents a green ball pulled from the bag? Would our estimate for the probability of pulling out a blue ball change from $\hat{p} = 3/10 = 0.3$?

What happens if we make more trials of this experiment, and find we see the outcome 30 times out of 100? Again we would expect our estimate $\hat{p}$ to be equal to 0.3 (30/100), but we would expect to be more confident in that measurement. We can see this in the likelihood and log-likelihood functions in the plots below. Both of these peak at $p = 0.3$ as expected but now they are much more sharply peaked at this value. We therefore expect the width of the likelihood function to be correlated with the error on the estimated parameter value. We will discuss this in Section Three.

In [6]:
```
def L100(p):
    return p**30*(1-p)**70

fig, axs = plt.subplots(1,2,figsize = (12,4),dpi = 150)
axs[0].plot(p,L100(p),color='#D55E00')
axs[0].set_xlabel('$p$',fontsize = 16)
axs[0].set_ylabel(r'$L(p; 30B, 70\tilde{B})$',fontsize = 16)
axs[0].tick_params(which='both',labelsize = 12,top=True,right=True,direction='in')
axs[0].xaxis.set_major_locator(MultipleLocator(0.1))
axs[0].xaxis.set_minor_locator(MultipleLocator(0.025))
axs[0].yaxis.set_major_locator(MultipleLocator(1e-27))
axs[0].yaxis.set_minor_locator(MultipleLocator(2e-28))
axs[0].set_ylim(bottom = 0)
axs[0].set_title('Binomial likelihood',fontsize = 20)
axs[0].grid(color='xkcd:dark blue',alpha =0.2)

p3 = np.linspace(0.1,0.5,100) # Limit range due to overflows
axs[1].plot(p3,np.log(L100(p3)),color='#D55E00')
axs[1].set_xlabel('$p$',fontsize = 16)
axs[1].set_ylabel(r'$\ln(L(p; 30B, 70 \tilde{B})$',fontsize = 16)
axs[1].tick_params(which='both',labelsize = 12,top=True,right=True,direction='in')
axs[1].xaxis.set_major_locator(MultipleLocator(0.1))
axs[1].xaxis.set_minor_locator(MultipleLocator(0.025))
axs[1].yaxis.set_major_locator(MultipleLocator(1))
axs[1].yaxis.set_minor_locator(MultipleLocator(0.25))
axs[1].set_xlim(0.0,1.0)
axs[1].set_ylim(-67,-61)
axs[1].set_title('Binomial log likelihood',fontsize = 20)
axs[1].grid(color='xkcd:dark blue',alpha =0.2)
```



In fact, this idea can be taken as a principle, referred to as the **maximum likelihood principle**: the values of the parameters that maximise the log-likelihood function are the best estimates of those parameters. This is a very general method; we assume our measurement value is at the peak of the PDF, and we adjust the PDF parameters to match our data.

These estimates are consistent, efficient, and unbiased at least in the limit of large numbers of data points. In general, we take the same approach as for the chi-squared method: we want to maximise the log-likelihood with respect to the parameters θ_j to find the best estimate of each parameter, by requiring:

$$\frac{\partial \ln(L)}{\partial \theta_j} = 0$$
$$\frac{\partial^2 \ln(L)}{\partial \theta_j^2} < 0$$

For the binomial example above (with 3 successes and 10 trials), we find that

$$\ln(L) = \ln(120 p^3 (1-p)^7) = \ln(120) + 3\ln(p) + 7\ln(1-p)$$

and then that the derivative is given as

$$\frac{\partial \ln(L)}{\partial p} = \frac{3}{p} - \frac{7}{1-p}$$

We find the estimate $\hat{p}$ by setting this equal to 0, giving:

$$\frac{3}{\hat{p}} = \frac{7}{1-\hat{p}} \qquad \text{so} \qquad 3 - 3\hat{p} = 7\hat{p} \qquad \text{so} \qquad \hat{p} = \frac{3}{10}$$

exactly as we would expect.

In general we cannot compute parameter estimates analytically, much like with the chi-squared method.

Gaussian approximation

It is useful to consider parameter estimation for the Gaussian case explicitly for reasons that will become clear. We can write the PDF for a single measurement of a random variable X distributed according to some normal distribution $N(\mu, \sigma)$ as

$$f(X;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \qquad \text{so} \qquad \ln(f(X;\mu,\sigma)) = -\ln\left(\sigma\sqrt{2\pi}\right) - \frac{(x-\mu)^2}{2\sigma^2}$$

If we know σ, we can make an estimate of μ from one measurement X_1. The log-likelihood is given as

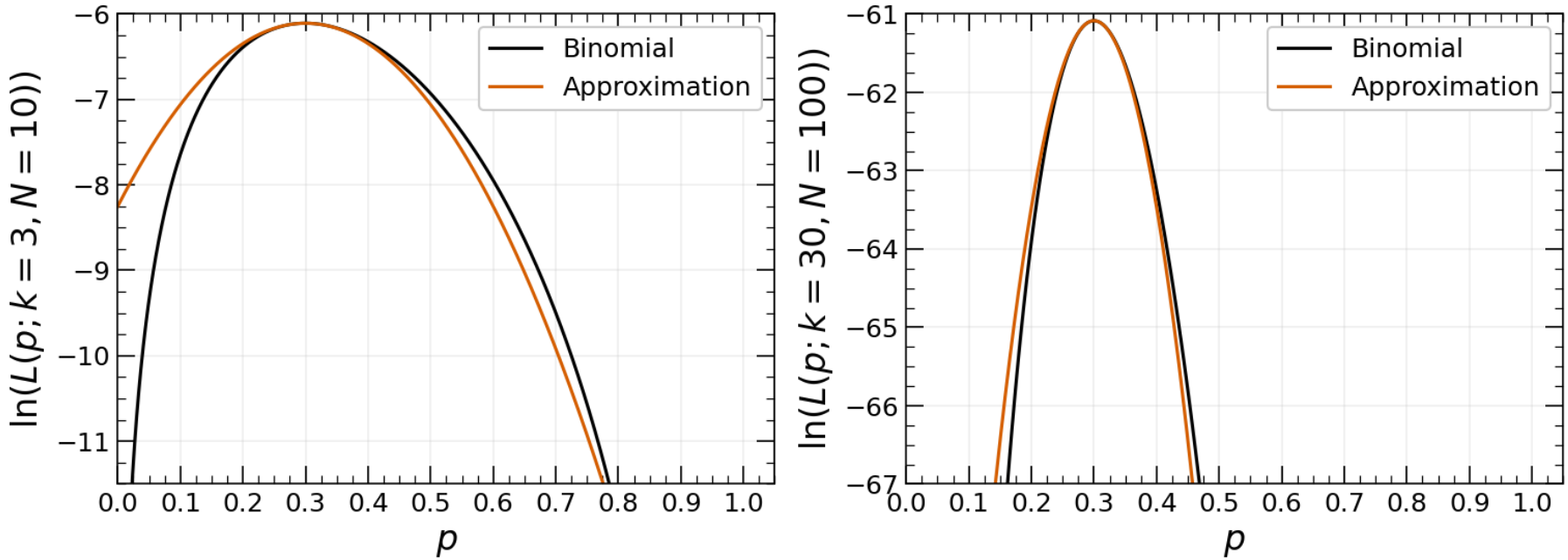
$$\ln(L(\mu)) = -\ln\left(\sigma\sqrt{2\pi}\right) - \frac{(X_i-\mu)^2}{2\sigma^2}$$

which peaks at $\hat{\mu} = X_1$ as you would expect.

The reason this is such an important case to consider is due to how all functions behave near a maximum; if we take the Taylor expansion of any function near a maximum (where the first derivative is zero), then the function will approximate to a constant term plus a negative quadratic term, much like our Gaussian log-likelihood. The plot below compares the true log-likelihood and the Gaussian approximation for the binomial example before, both binomial examples given above. As a result, we say that the likelihood function for any PDF is approximately Gaussian near the peak. This will be key when we discuss how we can estimate the error on our estimated parameters.

In [7]:

```
fig, axs = plt.subplots(1,2,figsize = (12,4),dpi = 150)

# Use p2 list to avoid log(0.0)
axs[0].plot(p2,np.log(p2**3*(1-p2)**7),color='black',label='Binomial')
axs[0].plot(p,np.log(0.3**3*0.7**7) + 0.5*(-3/0.3**2 - 7/0.7**2)*(p - 0.3)**2,color='#D55E00',label='Approximation')
axs[0].set_ylim(-11.5,-6)
axs[0].set_xlim(left=0)
axs[0].set_xlabel('$p$',fontsize = 16)
axs[0].set_ylabel('$\ln(L(p; k = 3, N = 10))$',fontsize = 16)
axs[0].tick_params(which='both',labelsize = 12,top=True,right=True,direction='in')
axs[0].tick_params(which='major',size = 8)
axs[0].tick_params(which='minor',size = 4)
axs[0].xaxis.set_major_locator(MultipleLocator(0.1))
axs[0].xaxis.set_minor_locator(MultipleLocator(0.025))
axs[0].yaxis.set_major_locator(MultipleLocator(1))
axs[0].yaxis.set_minor_locator(MultipleLocator(0.25))
axs[0].legend(loc='upper right',fontsize = 12,framealpha = 1)
axs[0].grid('xkcd:dark blue',alpha = 0.2)

# Use p3 list to avoid log(0.0)
axs[1].plot(p3,np.log(p3**30*(1-p3)**70),color='black',label='Binomial')
axs[1].plot(p,np.log(0.3**30*0.7**70) + 0.5*(-30/0.3**2 - 70/0.7**2)*(p - 0.3)**2,color='#D55E00',label='Approximation')
axs[1].set_ylim(-67,-61)
axs[1].set_xlim(left=0)
axs[1].set_xlabel('$p$',fontsize = 16)
axs[1].set_ylabel('$\ln(L(p; k = 30, N = 100))$',fontsize = 16)
axs[1].tick_params(which='both',labelsize = 12,top=True,right=True,direction='in')
axs[1].tick_params(which='major',size = 8)
axs[1].tick_params(which='minor',size = 4)
axs[1].xaxis.set_major_locator(MultipleLocator(0.1))
axs[1].xaxis.set_minor_locator(MultipleLocator(0.025))
axs[1].yaxis.set_major_locator(MultipleLocator(1))
axs[1].yaxis.set_minor_locator(MultipleLocator(0.25))
axs[1].legend(loc='upper right',fontsize = 12,framealpha = 1)
axs[1].grid('xkcd:dark blue',alpha = 0.2)
```



From these plots, it can be seen that the approximation is most accurate near the peak, but the range for which it is accurate increases if we increase the number of trials (as we expect for a Gaussian approximation).

For a set of measurements X_i of Gaussian distributed random variables, each with different (unknown) mean μ_i and (known) width σ_i, we can show that maximising the log-likelihood with respect to the means μ_i is equivalent to minimising the chi-squared with respect to the same parameters. In fact, we find that

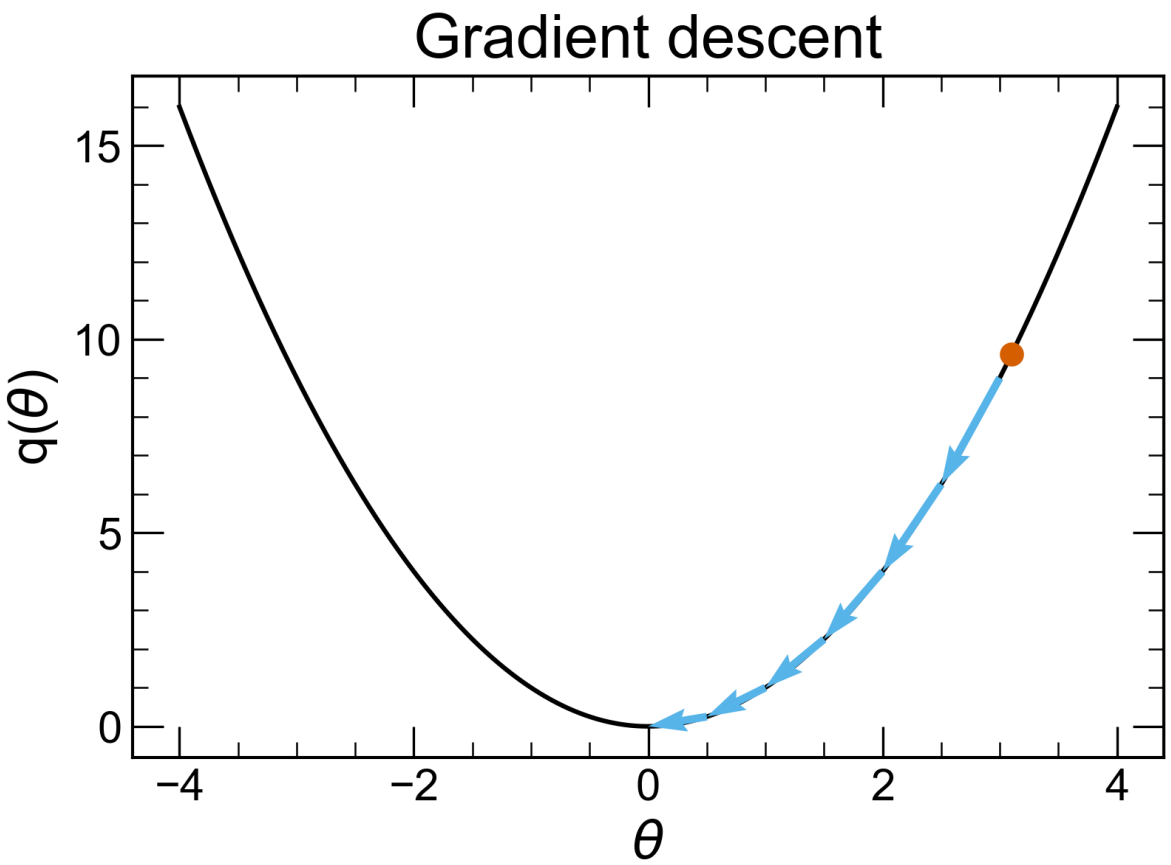
$$\chi^2 = 2C - 2\ln(L)$$

for some constant C (which is constant if the widths σ_i are known). As a result, it is common to find a minimum of $-2\ln(L)$ rather than trying to maximise the log-likelihood, as this is directly comparable to the chi-squared in the Gaussian case.

Numerical optimisation of cost functions

So far we have discussed two different cost functions that we can optimise in order to estimate our parameters: the chi-squared and the log-likelihood. However, we cannot always analytically solve to find our estimates and instead must apply numerical methods. While we won't cover this in detail here, it is important you are aware of what kinds of approaches might be used.

One of the most common algorithm used for these optimisation problems is **gradient descent**. Specifically, we use gradient descent for minimisation problems; either for minimising the χ^2, or minimising the negative log-likelihood (equivalent to maximising the log-likelihood). To do this, we make updates to the parameter estimate by changing it by some small value in the direction of the negative gradient, i.e. we step towards the point where gradient tends to 0. This is illustrated in the figure below.



Example of how the parameter value θ updates in an application of gradient descent. In this scenario, the cost function is given as X^2. With each step in the direction of the negative gradient, we move towards the minimum of the cost function

The amount we change our parameter θ by in each step is proportional to the gradient and to a parameter called the **learning rate**, which we label as α. In fact, the learning rate belongs to a type of parameter called **hyperparameters**, which can be defined as *parameters of the learning algorithms, and not of the model*. We will see this concept more when we talk about machine learning in the rest of this course.

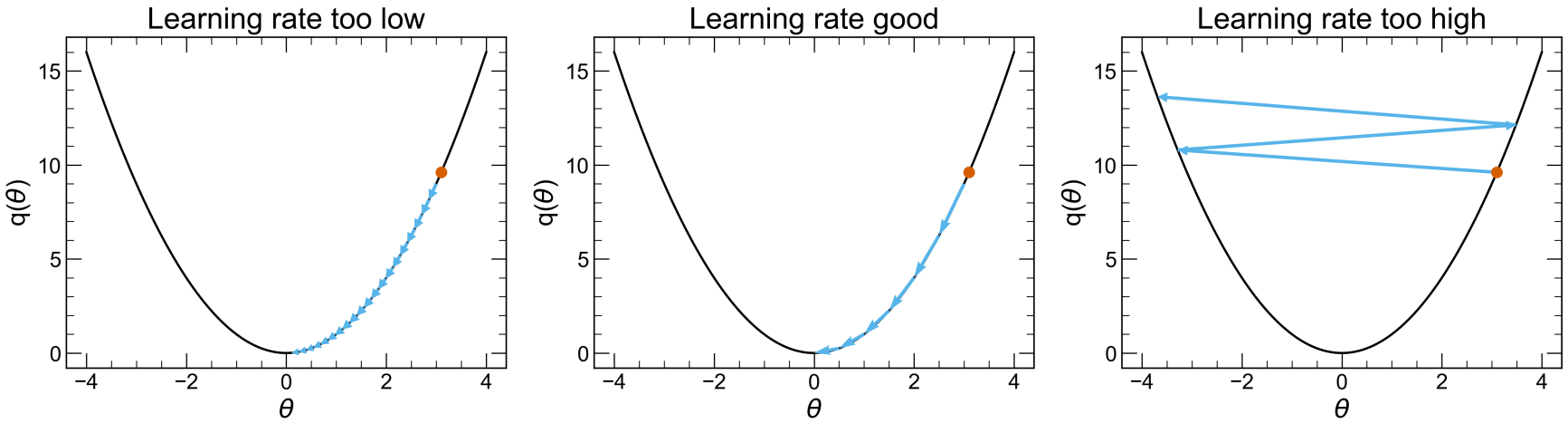
The main steps of this algorithm are as follows:

1. Define an initial parameter guess θ_{init} and set our parameter estimate $\hat{\theta}$ equal to it
2. Calculate (numerically) the gradient of the cost function q evalauted at the initial guess, $\frac{\partial q}{\partial \hat{\theta}}$
3. Update the value of $\hat{\theta}$ according to

$$\hat{\theta} = \hat{\theta} - \alpha \frac{\partial q}{\partial \hat{\theta}}$$

   where α is a parameter known as the **learning rate**.
4. Repeat steps 2 and 3 until the value of the cost function is below some satisfactory value or satisfies some convergence condition

Because the learning rate is a hyperparameter of the algorithm, it must be chosen carefully to give good performance. If the learning rate is too low, we can take a long time to find a minimum as we need to make many steps. It is also possible to get caught in a local minimum of the cost function, rather than finding the global minimum. If the learning rate is too high, we may "skip" over the minimum and instead oscillate around the minimum or diverge away from it. A good choice of learning rate will result in fast convergence to a global minimum. The figure below shows these three scenarios.



Different learning rate scenarios. For low learning rates, too many updates are required to reach the minimum, while a large learning rate causes drastic updates leading to divergent behaviour.

You will not need to code gradient descent yourself; it is very well implemented in many software packages, including the `iminuit` package we will use for fitting. You will see this library in more detail later in this workbook. The same method can be applied to models with multiple parameters, we must simultaneously optimise the cost function with respect to all of the parameters. You will also see gradient descent again in Week 9 when we discuss neural networks.

Summary

In this section you have seen how we can make best estimates of parameters from data we have measured, including:

- Chi-squared minimisation
- The maximum likelihood principle
- Gradient descent to minimise cost functions

The following section will discuss how we can determine the error on parameters we have estimated.

Section Three: Parameter Uncertainty Intervals ^

When we make measurements of anything, we need to know the uncertainty on the measurement, because this tells us how much we can trust the value we have measured. Much like we consider uncertainty on the values we measure, we need to find uncertainties for parameters that we estimate. In this section, we will discuss how we can find parameter uncertainties from our fitting methods and then how this links back to the confidence interval we discussed for hypothesis testing last week.

We will first discuss our two main estimation methods in turn.

Chi-squared parameter uncertainties

As we have previously defined, the chi-squared is a function of the parameters of our fit for measured data $\{X_i, y_i\}$, and can be written as

$$\chi^2(\theta; y_i) = \sum_{i=1}^{N} \left[\frac{y_i - f(X_i; \theta)}{\sigma_i} \right]^2$$

where $f(X_i; \theta)$ is our model prediction at the point X_i and σ_i is the uncertainty on the measurement y_i.

Consider taking a Taylor expansion around the minimum of the chi-squared which occurs at $\theta = \hat{\theta}$, assuming just one parameter for clarity. At a minimum, the linear term disappears and we are left with

$$\chi^2(\theta) \approx \chi^2(\hat{\theta}) + \left. \frac{d^2\chi^2}{d\theta^2} \right|_{\theta=\hat{\theta}} \frac{(\theta - \hat{\theta})^2}{2} + \cdots$$

We know the second derivative must be positive as the best-fit chi-squared is a minimum. We will therefore define

$$\frac{1}{\Sigma^2} = \frac{1}{2} \left. \frac{d^2\chi^2}{d\theta^2} \right|_{\theta=\hat{\theta}}$$

which allows us to write

$$\chi^2(\theta) \approx \chi^2(\hat{\theta}) + \frac{(\theta - \hat{\theta})^2}{\Sigma^2}$$

We can interpret the second term of this as a residual divided by an uncertainty, like we saw when we talked about the chi-squared goodness-of-fit test. We therefore can identify our quantity Σ as the uncertainty on the best estimate $\hat{\theta}$.

If we had a vector of multiple parameters $\boldsymbol{\theta}$, then the uncertainty Σ is actually a matrix $\boldsymbol{\Sigma}$ and is referred to as the **error matrix**. The diagonal element Σ_{ii} is are the uncertainty on the parameter θ_i, while the off-diagonal element Σ_{ij} is the covariance of the parameters θ_i and θ_j. For uncorrelated parameters θ_i and θ_j, the covariance $\Sigma_{ij} = 0$.

We define the elements of the weight matrix $\mathbf{W} = \boldsymbol{\Sigma}^{-1}$ as

$$W_{ij} = \frac{1}{2} \left. \frac{\partial^2\chi^2}{\partial\theta_i \partial\theta_j} \right|_{\theta=\hat{\theta}}$$

which is half of what is called a Hessian matrix of the function χ^2 evaluated at $\hat{\theta}$.

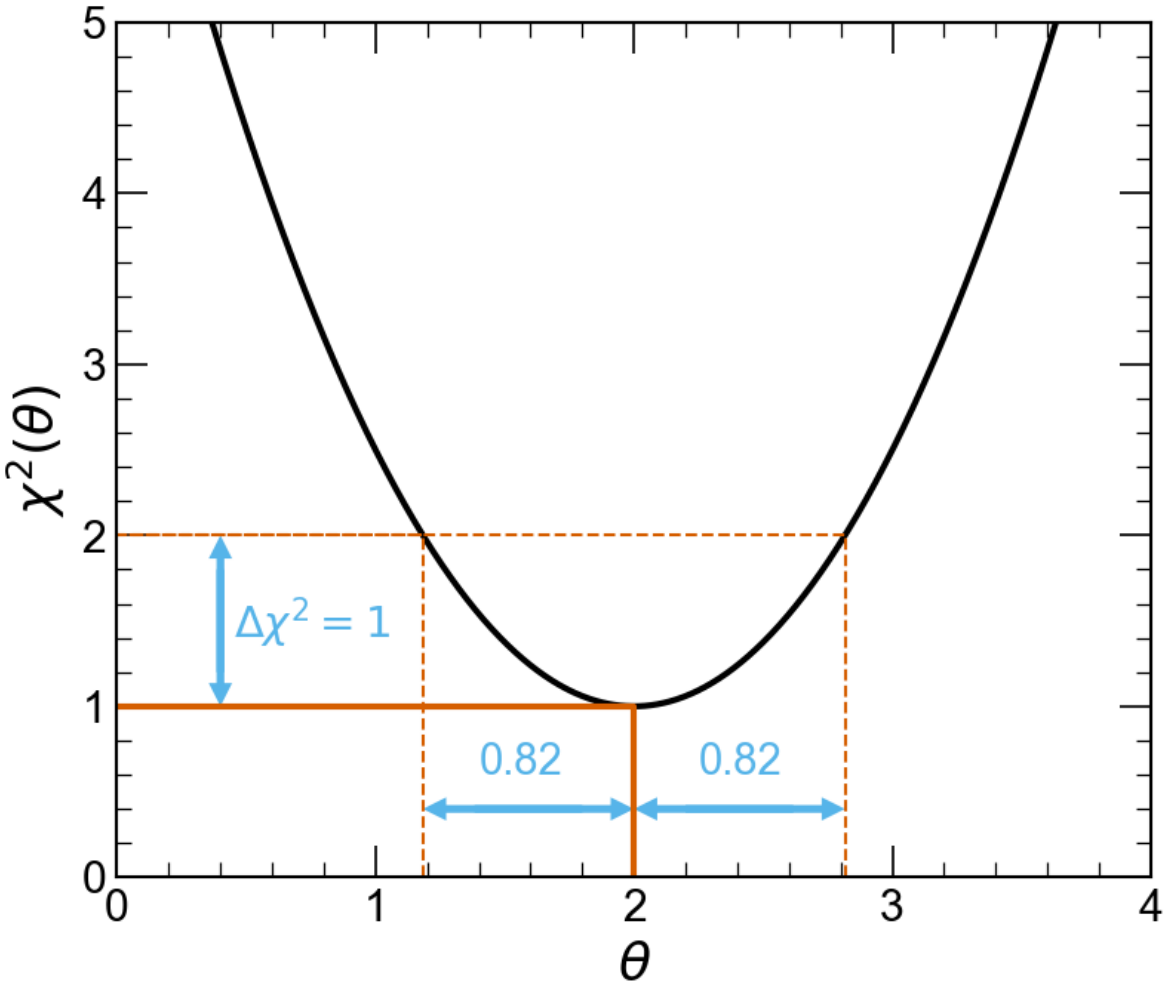
To find the covariance matrix, we must invert the weight matrix.

If the second derivative of the chi-squared is easy to calculate, the uncertainty on the best estimate can be found through this method. However, because the chi-squared method can be used for arbitrarily complex functions $f(X_i;\theta)$, we cannot always calculate the second derivative practically.

Instead, we adopt a perturbation approach: what happens if we perturb the value of the parameter θ from the value that gives the minimum chi-squared? Let us specifically consider shifting the parameter from $\hat{\theta}$ to $\hat{\theta} \pm \Sigma$. From our Taylor expansion, we can see this becomes

$$\begin{aligned}\chi^2(\hat{\theta} \pm \Sigma) &\approx \chi^2(\hat{\theta}) + \frac{(\hat{\theta} \pm \Sigma - \hat{\theta})^2}{\sigma^2} \\ &\approx \chi^2(\hat{\theta}) + \frac{\Sigma^2}{\Sigma^2} \\ &\approx \chi^2(\hat{\theta}) + 1\end{aligned}$$

i.e. the value of the chi-squared changes by 1 if we change the parameter from the best estimate by $\pm\Sigma$. As a result, if we numerically evaluate the chi-squared for different values of theta, the range for which the chi-squared changes by up to 1 gives the uncertainty on $\hat{\theta}$. An example for such an approach can be seen in the figure below.



Determining uncertainty intervals from the chi-squared distribution for a parameter estimate $\hat{\theta}$. From this we determine that the best estimate is $\hat{\theta} = 2 \pm 0.82$.

Maximum likelihood parameter uncertainties

For maximum likelihood parameter uncertainties, we adopt a very similar approach to the one we have just outlined for the chi-squared. By taking a Taylor expansion around the maximum of the log-likelihood and knowing that the second derivative must be negative as we are at a maximum, we find that

$$\ln(L(\theta)) \approx \ln(L(\hat{\theta})) + \left.\frac{d^2\ln(L)}{d\theta^2}\right|_{\theta=\hat{\theta}} \frac{(\theta-\hat{\theta})^2}{2!} + \cdots$$

This time, we identify the constant Σ according to

$$\frac{1}{\Sigma^2} = -\left.\frac{d^2\ln(L)}{d\theta^2}\right|_{\theta=\hat{\theta}}$$

The negative sign means that Σ^2 is positive and thus that Σ is real. We can then write the approximation as

$$\ln(L(\theta)) \approx \ln(L(\hat{\theta})) - \frac{(\theta-\hat{\theta})^2}{2\Sigma^2} + \cdots$$

By ignoring higher order terms and taking the exponential of both sides, we find that

$$L(\theta) \approx L(\hat{\theta}) e^{-(\theta-\hat{\theta})^2/2\Sigma^2}$$

i.e. a Gaussian function in θ, with mean $\hat{\theta}$ and width Σ. We therefore identify Σ as the uncertainty on the parameter θ.

This holds for the single parameter case; if we have more than one parameter, then we must again invert the weight matrix $\mathbf{W}$ like for the chi-squared case, where the elements W_{ij} are defined as

$$W_{ij} = -\left.\frac{\partial^2 ln(L)}{\partial\theta_i \partial\theta_j}\right|_{\theta=\hat{\theta}}$$

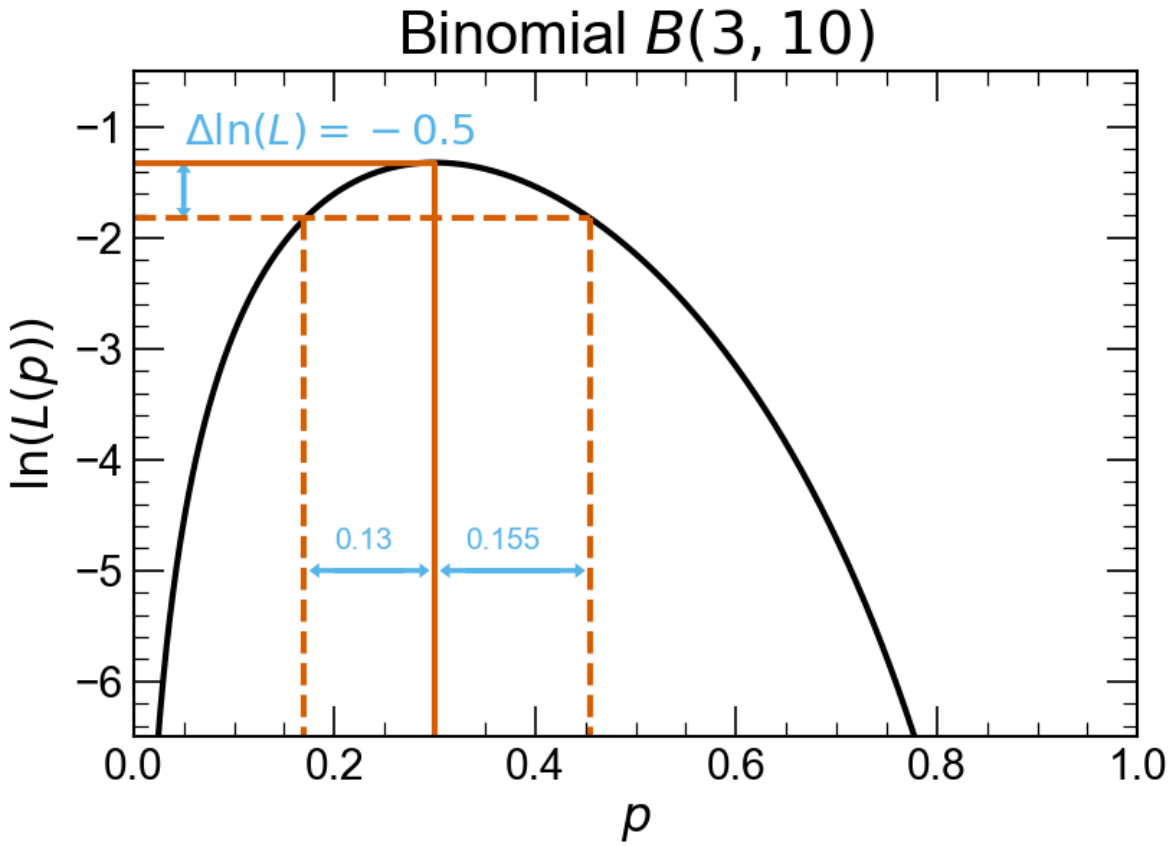
and the error matrix $\mathbf{\Sigma}$ is the inverse of the weight matrix.

What if the Gaussian approximation isn't a good approximation to our function, or it is difficult to evaluate the second derivative of the log-likelihood? We will use a similar approach to the chi-squared, namely examining the value of the log likelihood at $\theta = \hat{\theta} \pm \Sigma$ to find a range of log-likelihood values that correspond to our uncertainty.

We will start with the Gaussian case, and then claim (without proof) that this applies for non-Gaussian likelihoods as well (for the large N limit in general but as a good approximation normally).

$$\begin{aligned}\ln(L(\hat{\theta} \pm \Sigma)) &= \ln(L(\hat{\theta})) - \frac{(\hat{\theta} \pm \Sigma - \hat{\theta})^2}{2\Sigma^2} \\ &= \ln(L(\hat{\theta})) - \frac{\Sigma^2}{2\Sigma^2} \\ &= \ln(L(\hat{\theta})) - \frac{1}{2}\end{aligned}$$

i.e. the uncertainy is the range that changes the log-likelihood by -1/2, whereas for the chi-squared we look for the range that changes by +1. This range can in general be asymmetric. This is shown for a binomial example with distribution $B(3, 10)$ in the figure below.



Example of calculating estimated parameter uncertainty using log-likelihood estimation for a binomial distribution. In this case, the best estimate $\hat{p} = 0.3^{+0.155}_{-0.13}$.

We can use the same method in general rather than only for Gaussian likelihoods as a pretty good approximation (which improves as the number of measurements N increases).

Parameter uncertainties and confidence intervals

We've done a lot of talking about these errors on our estimated parameters and deriving them in analogy with the width of a Gaussian, but what do they actually mean? To understand this, we first need to understand the idea of confidence intervals on random variables.

As a basic example, consider a normally distributed random variable X distributed according to $N(\mu, \sigma)$. If we integrate the PDF between $\mu - \sigma$, and $\mu + \sigma$, we find that the probability of a measurement of X lying in this range is 68.3%. This is referred to as the 1σ confidence interval: if we make a measurement of X, we expect it to lie in this interval 68.3% of the time.

We can define ranges like these for non-Gaussian distributions, regardless of the parameters of the specific distribution. Despite the fact that the specific value of 68.3% is derived from the integral of a Gaussian, it is very common to use this with any distribution and even refer to it as the 1σ confidence interval, even if the distribution in question doesn't have σ as a parameter.

However, things can be slightly more complicated for non-Gaussian distributions because they can be asymmetric and as such there is not a unique way to define the confidence interval of a given percentage. For example for the 68.3% interval for a general PDF $f(X)$, we require that

$$\int_a^b f(X)dX = 0.683$$

where our confidence interval is bounded by $X = a$ and $X = b$. In general this only gives one constraint on two parameters, so they cannot be specified uniquely. A common way to handle this issue is to require that the probability outside of the confidence interval should be equal above and below the range, e.g. for the 68.3% interval we require that

$$\int_{-\infty}^a f(X)dX = 0.159$$
$$\int_b^{\infty} f(X)dX = 0.159$$

How does this relate to errors on estimated parameters? Unlike random variables, parameters have an exact value even if it is unkonw to us, which is why we try to estimate them from data that we measure. If we were to make a large number of experiments, we expect that

68.3% (or 95%, or whatever our quoted confidence value is) of the experiments will have the true value within the derived confidence interval. Again we can quote these in terms of numbers of σ even if our likelihood or PDF is not Gaussian.

We can also use one-sided confidence intervals, if e.g. our parameter cannot be negative like a cross section of some rare process, instead we only want to set an upper limit.

Summary

In this section we have discussed how we can estimate parameter uncertainties for our two parameter estimation methods, including:

- Estimating uncertainties through direct calculation of cost function derivatives
- Estimating uncertainties through numerical methods
- Connection of parameter uncertainties to confidence intervals

In the next section, we will discuss how we can do accurate and flexible fitting in python using `iminuit`.

Section Four: Curve Fitting Using Least Squares and `iminuit` ^

Now we have discussed the theory of how we can estimate parameters from data, we want to put it into practise. `scipy` has a series of functions for fitting data, but in general more advanced packages are preferred. The `minuit` C++ package is commonly used in High Energy Physics, and was developed at CERN. It has a python wrapper called `iminuit` that we will use for fitting in this course. This should be loaded on all physics PCs but if you are using your own machine you may need to install iminuit.

This section is intended to give an overview of some key `iminuit` syntax that you can work through to make sure you understand how to use the library. The following content is based on the `iminuit` tutorials, available on their website.

You may be familiar with routines such as `curve_fit` in `scipy.optimize`, where we apply `curve_fit` to our data and the function we are fitting to get a result. Here we will see that `iminuit` takes what is called an "object-oriented" approach. That is we create a `Minuit` object, a type of object defined by the `imunuit` package. This `Minuit` object contains everything: data, fitted function, the methods needed to generate the fit, fitted parameter values and their uncertainties.

A simple example of a fit using `iminuit`

Like all fitting, we require some data, the function (a "model") we are fitting to the data, and a cost function whose minimum indicates the parameters of the best fit.

Model and Fit Function

Our model used to create the data will also be the same function we use to fit the data. We will choose a simple line $y = m\,x + c$ with independent variable x and two fixed parameters m and c.

In [8]:
```
# Note this is both the model we use to construct the data and the function we use to fit the data.
def line(x, m, c):
    return x*m + c
```

Artificial data

We will create some artifical data starting from the function we will use to fit the data plus some random noise. An artifical data set like this is a classic way to test machine learning methods as we know what should be the best answer for our method as we created the data around this idal answer. Can the machine learning method, here curve fitting, find the "correct" answer (the "ground truth")?

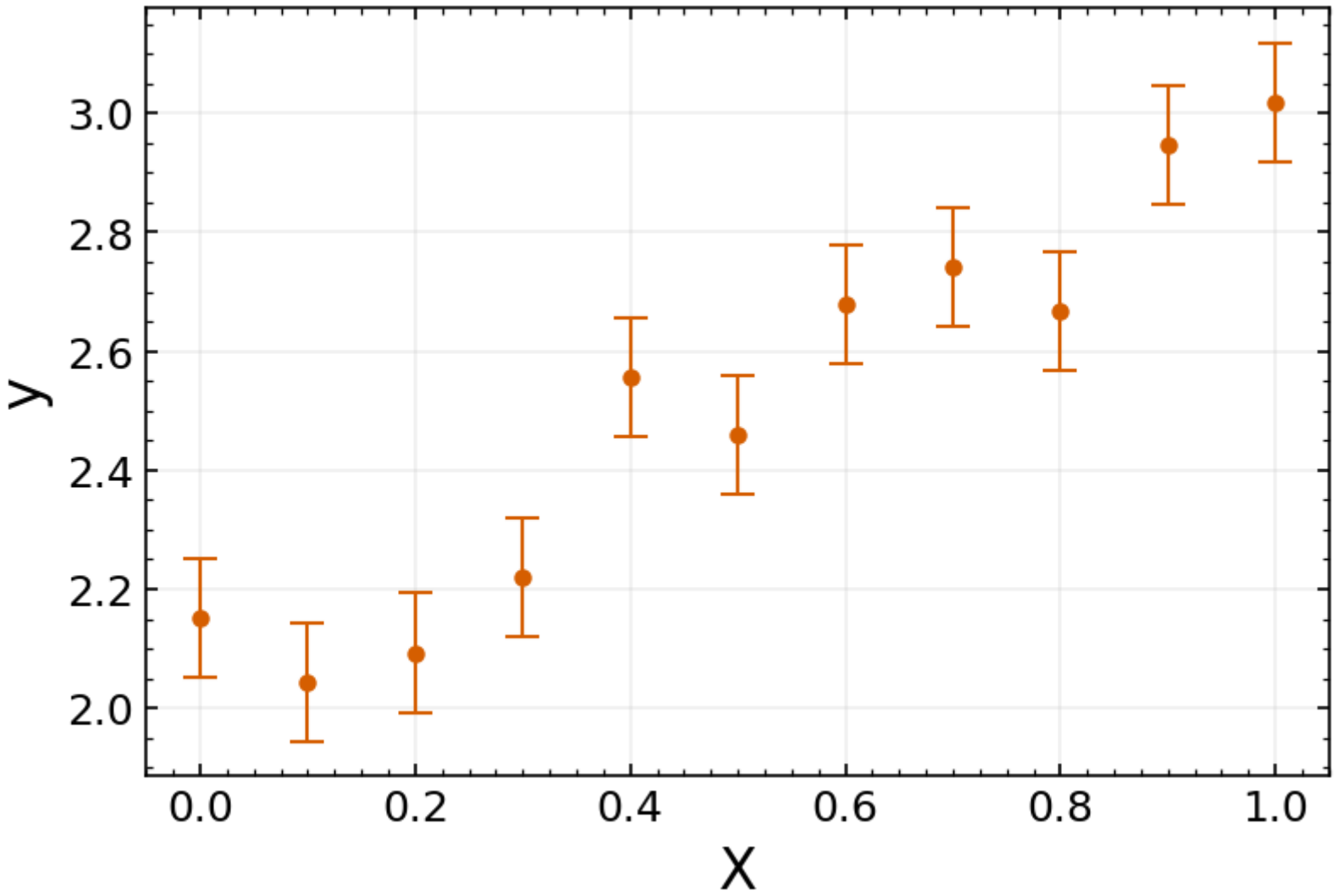
For our data:-

- We will set the parameters of this model to be $m = 1.0$ and $c = 2.0$.
- We will choose a small set of X values.
- For each X value we take our data point to be $m\,X + c$ plus a Gaussian random number chosens from a normal distribution of mean zero and standard deviation 0.1.

Note that our data points can be thought of as the mean value for y measured at each x value. Then the standard error in the mean represents the uncertainty in the y values but here this is simply the standard deviation we used when choosing a random number from a Gaussian distribution.

In [9]:
```
np.random.seed(0) # Set the random seed for consistency. Why not use 0?

# These are the ground truth values hiding in our data
```

```
m_value=1.0
c_value=2.0

n_points = 11 # 11 data points
data_x = np.linspace(0,1,n_points) # evenly spaced from 0.0 to 1.0 inclusive
data_yerr = 0.1 # the standard deviation of the noise
data_y = line(data_x, m_value, c_value) + np.random.default_rng().normal(loc = 0, scale = data_yerr, size = n_points)

# Plot to show the data
fig = plt.figure(dpi = 150,figsize = (6,4))
ax = fig.add_subplot()
ax.errorbar(data_x, data_y, yerr = data_yerr,
            color='#D55E00',fmt='.',capsize = 5,ms = 8, elinewidth = 1,capthick = 1)
ax.set_xlabel('X',fontsize = 16)
ax.set_ylabel('y',fontsize = 16)
# ax.xaxis.set_major_locator()
ax.xaxis.set_minor_locator(MultipleLocator(0.025))
ax.yaxis.set_minor_locator(MultipleLocator(0.05))
ax.tick_params(axis='both',labelsize = 12, direction='in',top = True, right = True, which='both')
ax.grid('xkcd:dark blue',alpha = 0.2)
```



Cost function

The simplest way to construct a cost function is to ue one of the many provided by `iminuit` as it has all the comon choises. Here we can use the `LeastSquares` class to generate an object which encodes the chi-squared cost function, has the data we are using and the function we want to fit, all in one object. So we need to pass the X values, y values, uncertainty on the y values, and the function we are trying to fit.

In [10]:
```
# Note: this is equivalent to our chi-squared cost function
from iminuit.cost import LeastSquares

least_squares_line = LeastSquares(data_x, data_y, data_yerr, line)
```

Note how this cost function object we defined knows what the parameters are. This cost function object looks at the function `line` given to it and assumes the second and subsequent arguments are the paramaters, while it assumes the first parameter of `line` is a vector of independent variable values X.

Minuit object

Now at last we combine data, fitted function and an initial value for the fitting parameters (needed to start the fitting method) into a single `Minuit` object - we "instantiate" a `Minuit` object. Everything we need is then in this object.

Note that choosing initial parameter values can have a big effect on the success of your algorithm in more complicated cases. If our cost function has several minima, the minimium reached after running the optimisation depends on the starting point. We might not even find the global minimum i.e. the best possible fit. Even if the cost function only has one minimum, starting parameter values in the proximity of the minimum will improve the speed of convergence.

However, for this simple case the initial values will have little effect.

In [11]:
```
# If you get an error then maybe iminuit is not installed on your machine.
from iminuit import Minuit

# Starting parameter values often important but not in this simple case
mline = Minuit(least_squares_line, m = 5.0, c = 5.0)

print(mline.init_params) # shows initial parameter values
```

	Name	Value	Hesse Err	Minos Err-	Minos Err+	Limit-	Limit+	Fixed
0	m	5.00	0.05					
1	c	5.00	0.05					

We can pass the parameters by the name *exactly as used in the fit function*, here our `line` function. These parameter names found by the `Minuit` object through "introspection".

We can also pass the parameters by positional arguments. Looking at our fit function `line` which is the `LeastSquares`, we see the first argument is m and the second is c so another way to define an equivalent Minuit object is

In [12]:
```
# This does exactly the same job as mline but isn't as clear to read.
mline1 = Minuit(least_squares_line, [5.0, 5.0])

print(mline1.init_params) # shows initial parameter values
```

	Name	Value	Hesse Err	Minos Err-	Minos Err+	Limit-	Limit+	Fixed
0	m	5.00	0.05					
1	c	5.00	0.05					

If we forget a parameter, or we write the wrong name for a parameter, `iminuit` will raise an error. The next two cells will show examples of such errors.

In [13]:
```
# error, our least_squares_line uses the line function
#        which takes two parameters

# UNCOMMENT THIS NEXT LINE TO SEE ERROR
#mline2 = Minuit(least_squares_line)
```

In [14]:
```
# error as no h parameter in the line function
#       inside the least_squares_line

# UNCOMMENT THIS NEXT LINE TO SEE ERROR
#mline3 = Minuit(least_squares_line, h = 10, c = 5)
```

The fit

After instantiating our `Minuit` object, the `mline` in the previous cell, we will use two different methods:

- `migrad` (the name of a particular minimisation method) to find the local minimum of the cost function. There are a few other pre-defined minimizers, but we will just use the MIGRAD algorithm.
- `hesse` (named after the Hessian matrix) to compute the uncertainty on our parameter.

In a Jupyter notebook, we get a nice rendering of the results of the fitting.

In [15]:
```
# Do the fitting

# minimise the least_squares cost function, i.e. find the best fit
mline.migrad()

mline.hesse() # calculate parameter uncertainties
```

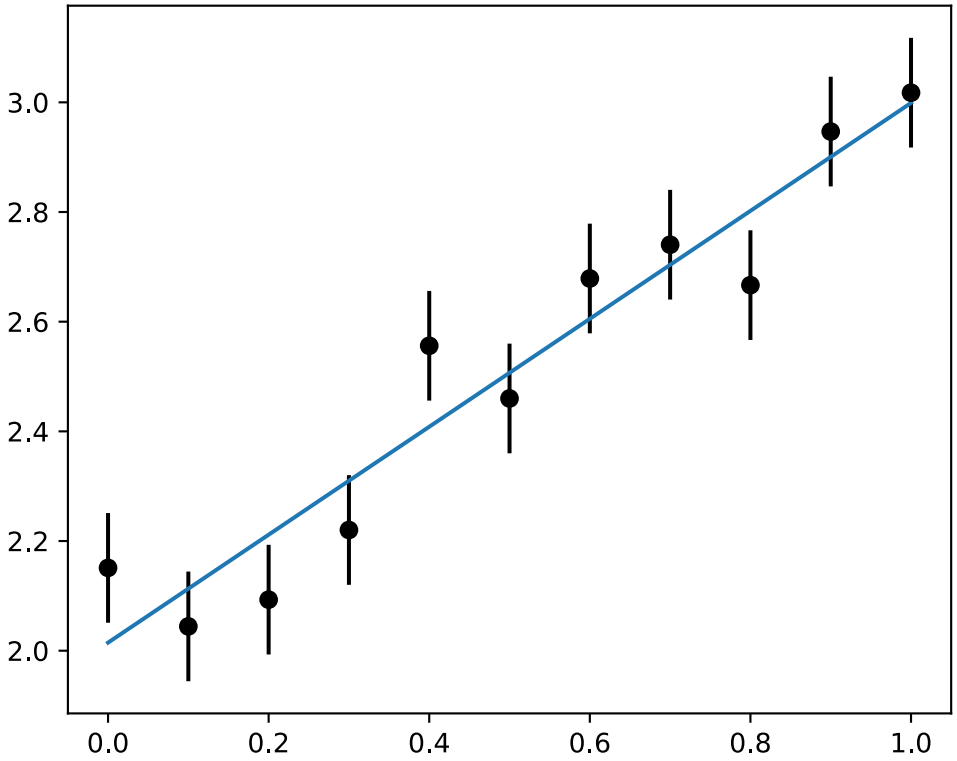
Out[15]:

Migrad

FCN = 9.7 (χ^2/ndof = 1.1)	Nfcn = 44
EDM = 2.97e-21 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	0.98	0.10					
1	c	2.01	0.06					

	m	c
m	0.00909	-0.0045 **(-0.845)**
c	-0.0045 **(-0.845)**	0.00318



OPTIONAL EFS: You might notice the reduced chi-square is quoted in the output as χ^2/ndof. What are the degrees of freedom ndof in this case? As a bonus, can you find out how to get this from the iminuit object, our `mline`?

Hurray! You should see the line is a good fit, not too perfect, not too bad. The m and c values found are close to, within error, the values we used to create the data (the ground truth values).

Inspecting current parameter values

The `Minuit` object contains the current parameter values, errors, and some other information about each parameter. We can display the current values using `Minuit.params`:

In [16]:
```
display(mline.params)
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	0.98	0.10					
1	c	2.01	0.06					

If we just run `Minuit.params`, it returns a tuple-likecontainer of `Param` objects which are data structures representing properties of the parameters being fitted.

For instance

- `mline.params` is the tuple-like container of Param data objects which contain information about the fitted parameters. `display(mline.params)` shows the values nicely.
- This means `mline.params[p]` gives information about the p-th parameter.

We can use the `repr` function to get a detailed representation of a given parameter, `print(repr(mline.params[p]))`.

Each parameter has several fields including:

- `number`: parameter index.
- `name`: parameter name.
- `value`: value of the parameter at the minimum.
- `error`: uncertainty estimate for the parameter value.

For instance, `mline.params[0].name` gives the name of the first parameter

In [17]:
```
for param in mline.params:
    print(repr(param))
```

```
Param(number=0, name='m', value=0.9841982321520254, error=0.09534625684881527, merror=None, is_const=False, is_fixed=Fa
lse, lower_limit=None, upper_limit=None)
Param(number=1, name='c', value=2.0146896387717783, error=0.05640760923195247, merror=None, is_const=False, is_fixed=Fa
lse, lower_limit=None, upper_limit=None)
```

If we want properties of an individual parameter we can access them as follows

In [18]:
```
for param in mline.params:
    print('{} = {:.2f} +/- {:.2f}'.format(param.name, param.value, param.error))
```

```
m = 0.98 +/- 0.10
c = 2.01 +/- 0.06
```

In [19]:
```
print('{} = {:.2f} +/- {:.2f}'.format(mline.params[0].name, mline.params[0].value, mline.params[0].error))
```

```
m = 0.98 +/- 0.10
```

The initial parameter guesses can still be viewed using `Minuit.init_params` e.g. `mline.init_params[0]` is the information on initial values of the `m` parameter.

In [20]:
```
for param in mline.init_params:
    print(repr(param))
```

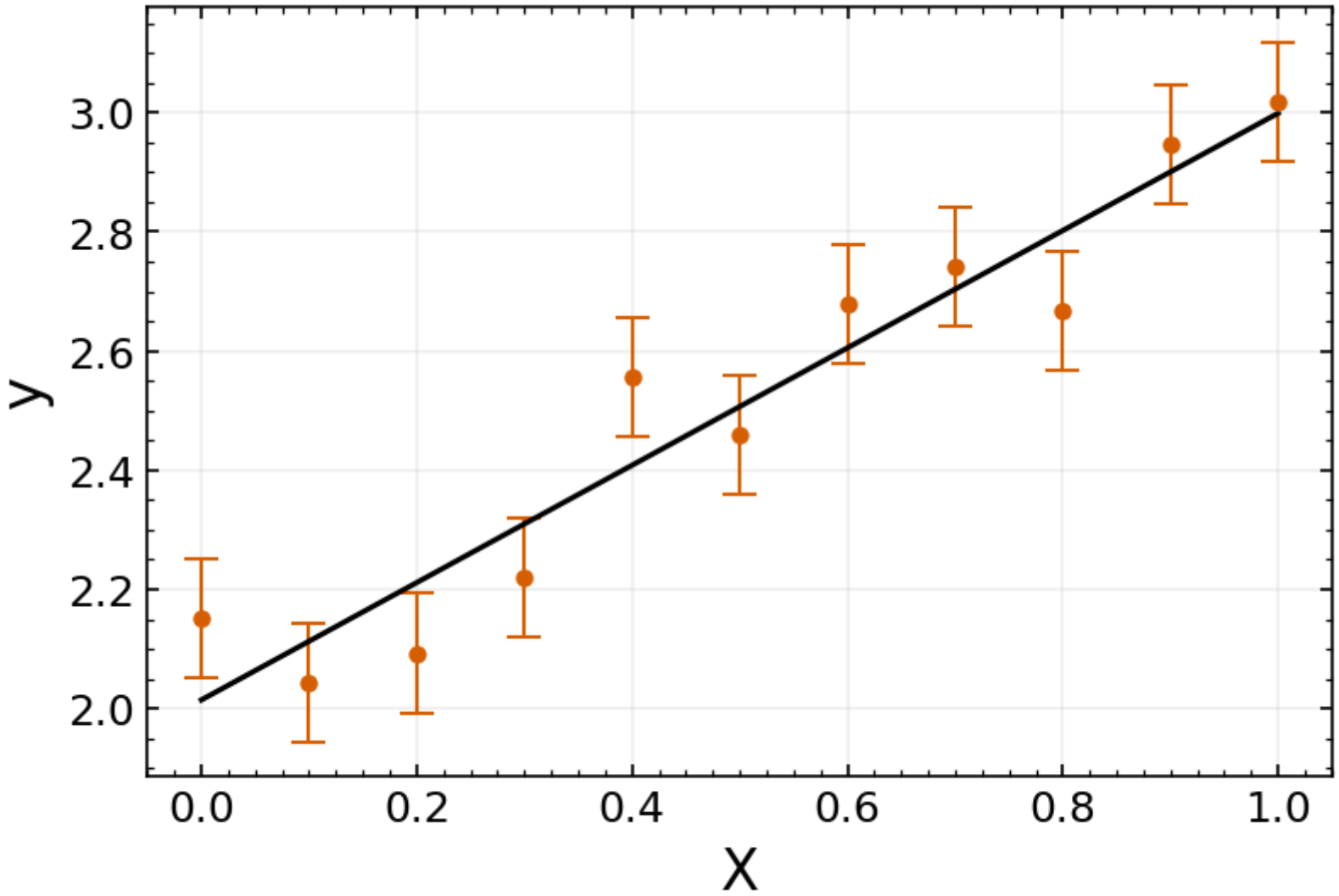
```
Param(number=0, name='m', value=5.0, error=0.05, merror=None, is_const=False, is_fixed=False, lower_limit=None, upper_l
imit=None)
Param(number=1, name='c', value=5.0, error=0.05, merror=None, is_const=False, is_fixed=False, lower_limit=None, upper_l
imit=None)
```

Visualise the fit

And that's all you need to do for a simple fit using `iminuit`. You should always visualise the fit. This isn't precise statistics but you need to use your common sense and just look to make sure your fitting is working. the automatic version above will do fine for development but here is a fancier version needed for presentation.

In [21]:
```
fig = plt.figure(dpi = 150,figsize = (6,4))
ax = fig.add_subplot()
ax.errorbar(data_x, data_y, yerr = data_yerr,
            color='#D55E00',fmt='.',capsize = 5,ms = 8, elinewidth = 1,capthick = 1)
ax.plot(np.linspace(0,1,100),line(np.linspace(0,1,100),*mline.values),color='black')

x_min = min(data_x)
x_max = max(data_y)
fit_x_values = np.linspace(x_min,x_max,100)
fit_y_values = line(fit_x_values, *mline.values)

ax.set_xlabel('X',fontsize = 16)
ax.set_ylabel('y',fontsize = 16)
# ax.xaxis.set_major_locator()
ax.xaxis.set_minor_locator(MultipleLocator(0.025))
ax.yaxis.set_minor_locator(MultipleLocator(0.05))
```

```
ax.tick_params(axis='both',labelsize = 12, direction='in',top = True, right = True, which='both')
ax.grid('xkcd:dark blue',alpha = 0.2)
```



Now we have seen how we can do a simple fit, we will talk about each part in more detail.

Using `iminuit` with an arbitrary number of parameters

We can also use iminuit with functions that accept an arbitrary number of parameters which are defined through data structures such as numpy arrays e.g. n-th order polynomials. This has pros and cons.

Pros

- Easy to change number of fitted parameters
- Sometimes simpler function body that's easier to read
- Technically this is more efficient, but this is hardly going to be noticable

Cons

- `iminuit` cannot (automatically) figure out names for each parameter
- user (you) might get confused over the name of the parameter linked to each entry in the array of parameter values

In [22]:
```
""" Functions used by iminuit's LeastSquares take the argument x first
    (could be an array of x_0, x_1, ... if in more than one dimension)
    followed by any number of parameters.

    And then notice that numpy's polyval,
    https://numpy.org/doc/stable/reference/generated/numpy.polyval.html
    which gives us a polynomial in x, puts the arguments the other way round!
    That is polyval wants parameter array first, x array second.
    N.B. The polynomial with N=len(par) has parameter of highest power of x from the first par[0] entry:-
    par[0]*x**(N-1) + par[1]*x**(N-2) + ... + par[N-2]*x + par[N-1]

"""

def line_np(x, par):
    """
    Input
    -----
    x = a 1D array of values
    par a tuple (list etc) of parameter values

    Return
    ------
    An array of polynomial values equal to
    par[0]*x**(N-1) + par[1]*x**(N-2) + ... + par[N-2]*x + par[N-1]

    """
```

```
    return np.polyval(par, x)  # for len(par) == 2, this is a line

least_squares_np = LeastSquares(data_x, data_y, data_yerr, line_np)

"""
    ISSUE.
    The version above works as long as you don't visualise the result in iminuit which the
    migrad method does in Jupyter notebooks. The solution used below is always to assign
    a variable to the Minuit object when using migrad. See Appendix for alternative solution.
"""
```

Out[22]:
```
"\n    ISSUE.\n    The version above works as long as you don't visualise the result in iminuit which the \n    migrad method does in Jupyter notebooks. The solution used below is always to assign \n    a variable to the Minuit object when using migrad. See Appendix for alternative solution.\n"
```

Calling `line_np` with more or fewer arguments is easy. For N arguments, a polynomial of order $(N-1)$ is used to predict the behavior of the data.

The built-in cost functions of `iminuit`, such as the `LeastSquares` used here, support models like our `line_np` with an unspecified number of arguments. However, for this to work properly, we need to pass the starting values in the form of a single sequence of numbers when constructing the `Minuit` object. That way `iminuit` can work out how many parameters it is fitting.

For example:-

In [23]:
```
display(Minuit(least_squares_np, (5.5, 6.0)))
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	x0	5.50	0.06					
1	x1	6.00	0.06					

Here we have used a tuple `(5.5,6.0)` for initialisation of the parameters, but any sequence (including lists and numpy arrays) would work. `iminuit` uses the length of the sequence to detect how many parameters the model has, so two here. By default, the *parameters* are named automatically x_0 to x_N - yes confusing as these are parameter names and nothing to do with the "x" values of our data.

We can override this naming convention with the keyword argument `name`, passing a sequence of parameter names. Of course, this sequence must be of the same length as the sequence of starting points.

In [24]:
```
# This means that the parameter we call "a" starts with value 5.5, etc
display (Minuit(least_squares_np, (5.5, 6.0), name=("a", "b")))
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	a	5.50	0.06					
1	b	6.00	0.06					

In [25]:
```
# As a quick check lets use the straight line version of the polynomial fit
m_line_obj_np = Minuit(least_squares_np, [5.5, 6.0])
display(m_line_obj_np)

""" These lines fail at line 1878 in the visualize method in iminuit\cost.py

m_line_obj_np.migrad()
m_line_obj_np.hesse() # calculate parameter uncertainties
display(m_line_obj_np)

As this looks to be routine for Jupyter notebook visualisation,
supressing the graphical output foes work suggesting
the main minimisation effort is working.

BUG?

"""

# minimise the least_squares cost function, i.e. find the best fit
# BUT SUPRESS THE GRAPHICAL OUTPUT!!!
mmm = m_line_obj_np.migrad()

hhh = m_line_obj_np.hesse() # calculate parameter uncertainties

# Now check it works and see if the answer is exactly as the line fit above.
for param in mline.params:
    print('{} = {:.2f} +/- {:.2f}'.format(param.name, param.value, param.error))

# Note that for Exercise 1 you need to get the values of the parameters into one list
fit_param_values = [ v for v in m_line_obj_np.values]
print(fit_param_values)
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	x0	5.50	0.06					
1	x1	6.00	0.06					

```
m = 0.98 +/- 0.10
c = 2.01 +/- 0.06
[0.9841982321512632, 2.014689638771314]
```

Since our `least_squares_np` works for parameter arrays of any length, we can now define a different `Minuit` object which will fit a cubic polynomial by specifying four initial values:

In [26]:
```
m_cubic_obj = Minuit(least_squares_np, (5.5, 4.1, 6.9, 0.4))
display (m_cubic_obj)
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	x0	5.50	0.06					
1	x1	4.10	0.04					
2	x2	6.90	0.07					
3	x3	0.400	0.004					

In [27]:
```
# minimise the least_squares cost function, i.e. find the best fit
# BUT SUPRESS THE GRAPHICAL OUTPUT!!!
mmm3 = m_cubic_obj.migrad()

hhh3 = m_cubic_obj.hesse() # calculate parameter uncertainties

# It works and the answer is comparable with the linear fit.
# noting that x3 is the constant (c) and x2 is the coefficient of x^1 (m)
for param in m_cubic_obj.params:
    print('{} = {:.2f} +/- {:.2f}'.format(param.name, param.value, param.error))

# Note that for Exercise 1 you need to get the values of the parameters into one list
fit_param_values = [ v for v in m_cubic_obj.values]
print(fit_param_values)

# Again useful for exercise 1
display(m_cubic_obj.fmin)
print( f" chi^2/dof = {m_cubic_obj.fval:.1f} / {m_cubic_obj.ndof:.0f} = {m_cubic_obj.fmin.reduced_chi2:.1f}")
```

```
x0 = -1.11 +/- 1.27
x1 = 1.81 +/- 1.94
x2 = 0.20 +/- 0.81
x3 = 2.08 +/- 0.09
[-1.1107605602036925, 1.8105744613543533, 0.20440957066574855, 2.076342062096706]
```

Migrad	
FCN = 8.759 (χ^2/ndof = 1.3)	Nfcn = 117
EDM = 9.48e-18 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

```
chi^2/dof = 8.8 / 7 = 1.3
```

We can try different orders of polynomial for fitting our data in this way, just by providing different numbers of starting parameters. Of course, getting the wrong order of polynomial for our data can lead to under or over fitting.

Exercise 1

Fit the data set in the following code cell with different orders of polynomial. Describe qualitatively how the fit changes, and how well the fit describes the data as you increase the order of the polynomial. When does the fit show signs of overfitting? Remember the following points:

- Define your cost function
- Define your Minuit object
- Run the optimisation
- Plot the data and your fit

Use the `line_np` function we have previously defined as your model. You can change the order of polynomial by passing more parameters as `pars` to this function. For instance if you call `line_np(X, (5, 5))` you will be testing a linear model, but if you call `line_np(X, (5, 5, 5, 5))` you will be testing a fit to a cubic, a polynomial of order 3.

In [28]:
```
x_values = [-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
y_measured = [343.39452514, 251.31914561, 186.7806368, 121.30027965, 45.23336652,
              23.49470302, 18.46581766, 6.58329486, 1.98522328, -6.74799454,
              -7.55489379, -29.55544088, -31.32898172, -36.27525348, -25.23860169,
              -1.09156819, 52.32898397, 225.38126087, 460.67437131]

y_err = [141.312, 86.205, 49.632, 26.877, 13.872,
         7.197, 4.08, 2.397, 0.672, 1.923, 5.568,
         9.795, 13.488, 14.883, 11.568, 0.483,
         22.08, 60.477, 119.712]
```

In [29]:
```
# Define your cost function

""" You don't need to repeat everything from above
but as a reminder we need:-

from iminuit.cost import LeastSquares
def line_np(x, par):
    return np.polyval(par, x)

Also remember that polyvar with N=len(par) parameters represents
    par[0]*x**(N-1) + par[1]*x**(N-2) + ... + par[N-2]*x + par[N-1]

"""

least_squares_ex1 = None
```

In [30]:
```
# Define your Minuit object

""" Initial Values.
You could play around by hand to find a good starting point.
For these polynomials it isn't too important but for
complicated functions and data a good choice is important
and it can be a real pain finding it.
So I usually make some simple estimates and use those as a starting point.
"""
# Trying to fit straight line, y = m x + c
m_est = None
c_est = None

# Minuit object for example 1 with two parameters (straight line)
mobj_ex1_p2 = None
```

In [31]:
```
# Run the minimisation

# *** I HAD TO SUPRESS THE OUTPUT HERE BY SETTING EQUAL TO A VARIBLE CALLED "_"
#     the single underscore character, often used as a dummy variable in python
# e.g.
# _ = (stuff to run the minimisation)
```

In [32]:
```
# Plot data and fit
```

End of Exercise 1.

Advanced parameter methods

Setting limits on parameters

When we are fitting for parameters, we may want to set limits on possible values as our parameter may be limited to a certain range either mathematically or physically. For example, if the function includes $\sqrt{a}$, then a must be non-negative, or if our parameter is an energy then we know it must be positive.

We can set limits on our parameters using `Minuit.limits` and slicing by the name of the parameter we want to set limits on:

In [33]:
```
# Create a fresh version of our Minuit object for a simple linear fit
mline = Minuit(least_squares_line, m = 3, c = 3)

mline.limits['m'] = (0,None) # set limit on m to be minimum of 0, no maximum
mline.limits['c'] = (-float("infinity"), 10) # set maximum value of c as 10, no minimum
mline.limits['m'] = (0,10) # set limit on m to be minium of 0, maximum of 10
mline.limits['m'] = (None, None)
```

We can also set multiple limits at once by passing a sequence of limits:

In [34]:
```
mline.limits = [(0, None), (0,10)]
display(mline.params)
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	3.00	0.03			0		
1	c	3.00	0.03			0	10	

We can see the limits for each parameter in this listing.

Fixing and releasing parameters

We can also fix parameters during the optimisation, so they are not optimised and instead kept at a fixed value. For example, we may have a guess for a parameter value and want to find the optimal values for other parameters for that fixed value.

You might also have a complex function with many parameters, where a few parameters cause large variations in the function but the rest cause much smaller variations. By first fixing the less important parameters, you can more quickly optimise the most important parameters, and then release the fixed parameters to complete the optimisation.

We can fix an individual parameter using `Minuit.fixed[<name>] = True`. For example, we can fix m in our straight line example:

In [35]:
```
# Create a fresh version of our Minuit object for a simple linear fit
mline = Minuit(least_squares_line, m = 3, c = 3)

mline.fixed['m'] = True
display(mline.params)
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	3.00	0.03					yes
1	c	3.00	0.03					

If we now run `migrad`, only c with be varied:

In [36]:
```
mline.migrad()
```

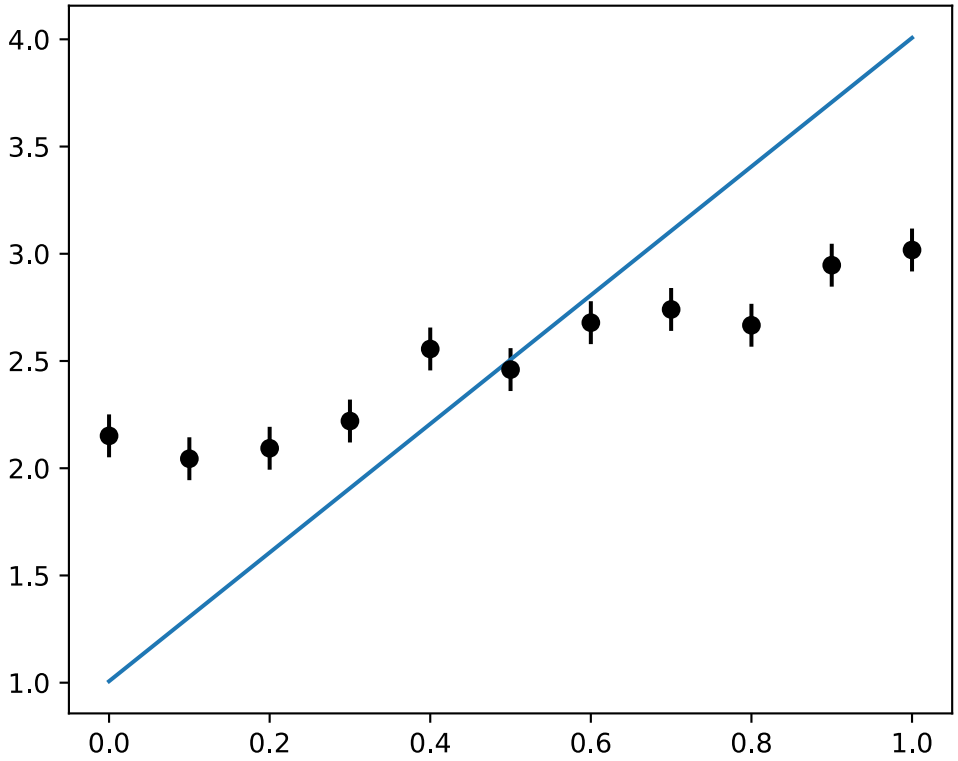
Out[36]:

Migrad

FCN = 456.7 (χ^2/ndof = 45.7)	Nfcn = 11
EDM = 2.55e-16 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	3.00	0.03					yes
1	c	1.007	0.030					

	m	c
m	0	0
c	0	0.000909



We can now release m and instead fix c, and run the optimisation again:

In [37]:
```
mline.fixed['m'] = False
mline.fixed['c'] = True
mline.migrad()
```

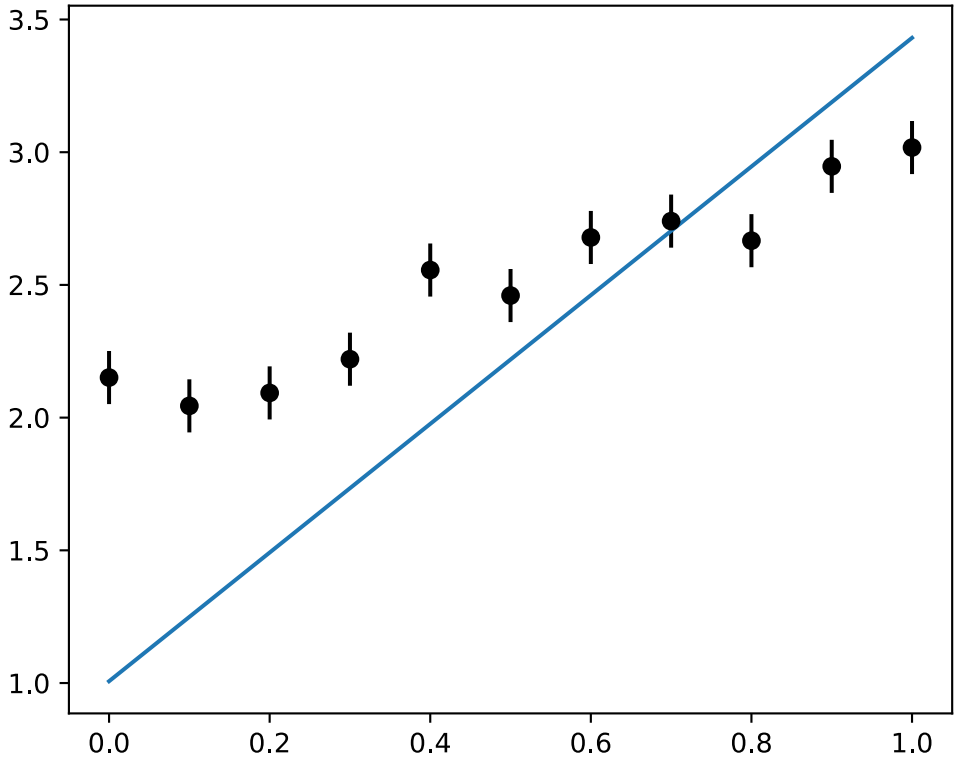
Out[37]:

Migrad

FCN = 329 (χ^2/ndof = 32.9)	Nfcn = 24
EDM = 2.03e-16 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	2.42	0.05					
1	c	1.007	0.030					yes

	m	c
m	0.0026	0.0000
c	0.0000	0



We could keep iterating like this, but it is only really necessary for more complex cases. For now we will release both parameters and rerun the minimisation.

In [38]:
```
mline.fixed = False # Can release all parameters at once this way
mline.migrad()
```

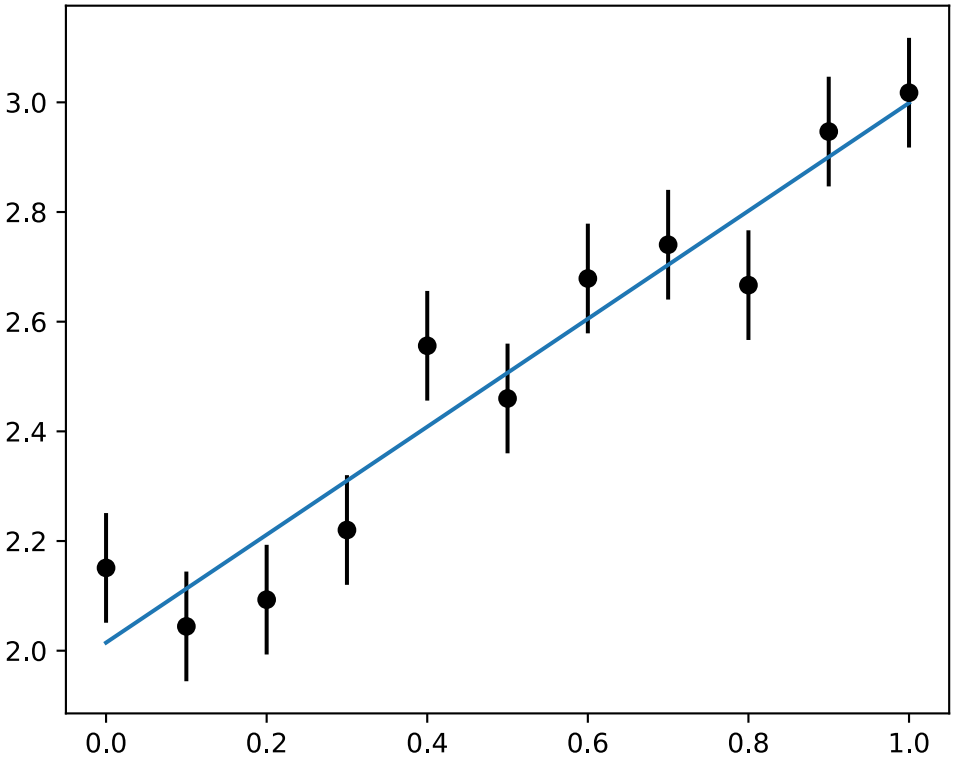
Out[38]:

Migrad

FCN = 9.7 (χ^2/ndof = 1.1)	Nfcn = 51
EDM = 1.11e-14 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	0.98	0.10					
1	c	2.01	0.06					

	m	c
m	0.00909	-0.0045 **(-0.845)**
c	-0.0045 **(-0.845)**	0.00318



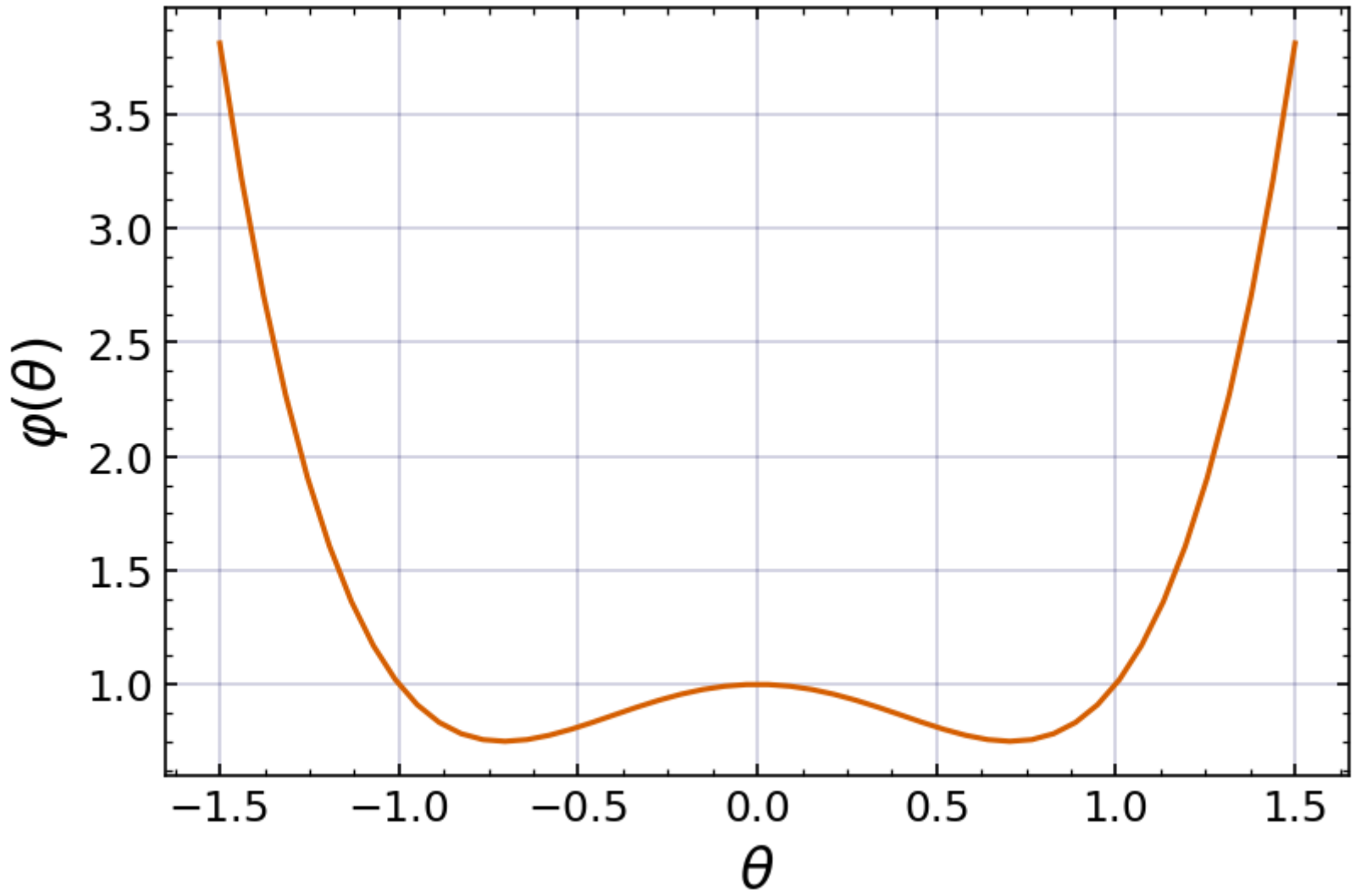
Varying starting points for minimisation

It can also be useful to be able to manually change the values of some fixed parameters and fit the others, or to redo a fit from a different starting point. For example, if the cost function has several minima, changing the starting parameter values can be used to help find the other minimum.

We define a custom cost function below to show this. We will denote our cost function as $\varphi(\theta)$.

In [39]:
```
# ! Note how python supports Unicode characters in variables names so
# !    you can use non-ascii characters such ϑ or 𝜍 in variables names.
# !    However, ascii characters work in anything and may be more
# !    readable to a wider audience.

def cost_function_with_two_minima(θ):
    return θ**4 - θ**2 + 1
```

```
# We set a parameter called errordef here; we will discuss this later
cost_function_with_two_minima.errordef = Minuit.LEAST_SQUARES

x = np.linspace(-1.5,1.5)
fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.plot(x, cost_function_with_two_minima(x),color='#D55E00')
ax.set_xlabel(r'$\theta$',fontsize = 16)
ax.set_ylabel(r'$\varphi(\theta)$',fontsize = 16)
ax.tick_params(direction='in',which='both',top=True,right=True,labelsize = 12)
ax.xaxis.set_minor_locator(MultipleLocator(0.125))
ax.yaxis.set_minor_locator(MultipleLocator(0.125))
ax.grid(color='xkcd:dark blue',alpha = 0.2)
```



In [40]:
```
# Starting at ϑ = -0.1 should give us the left minimum
m2minima = Minuit(cost_function_with_two_minima, θ=-0.1)
m2minima.migrad()
print("Starting value θ = -0.1, minimum at θ = {:.2f}".format(m2minima.values['θ']))

# Now changing the starting value to 0.1 gives the right minimum
m2minima.values['θ'] = 0.1
```

```
m2minima.migrad()
print("Starting value θ = +0.1, minimum at θ = {:.2f}".format(m2minima.values['θ']))
```

```
Starting value θ = -0.1, minimum at θ = -0.71
Starting value θ = +0.1, minimum at θ = 0.71
```

Investigating the fit status

As seen before, calling `Minuit.migrad()` runs the actual minimization with the Migrad algorithm. Migrad essentially tries a Newton-step and if that does not produce a smaller function value, it tries a line search along the direction of the gradient. So far so ordinary. The clever bits in Migrad are how various pathological cases are handled.

Let's look again at the output of `Minuit.migrad()`.

In [41]:
```
# Again set up a fresh Minuit object using our original simple linear fit example
mline = Minuit(least_squares_line, m=5, c=5)
mline.migrad()
```

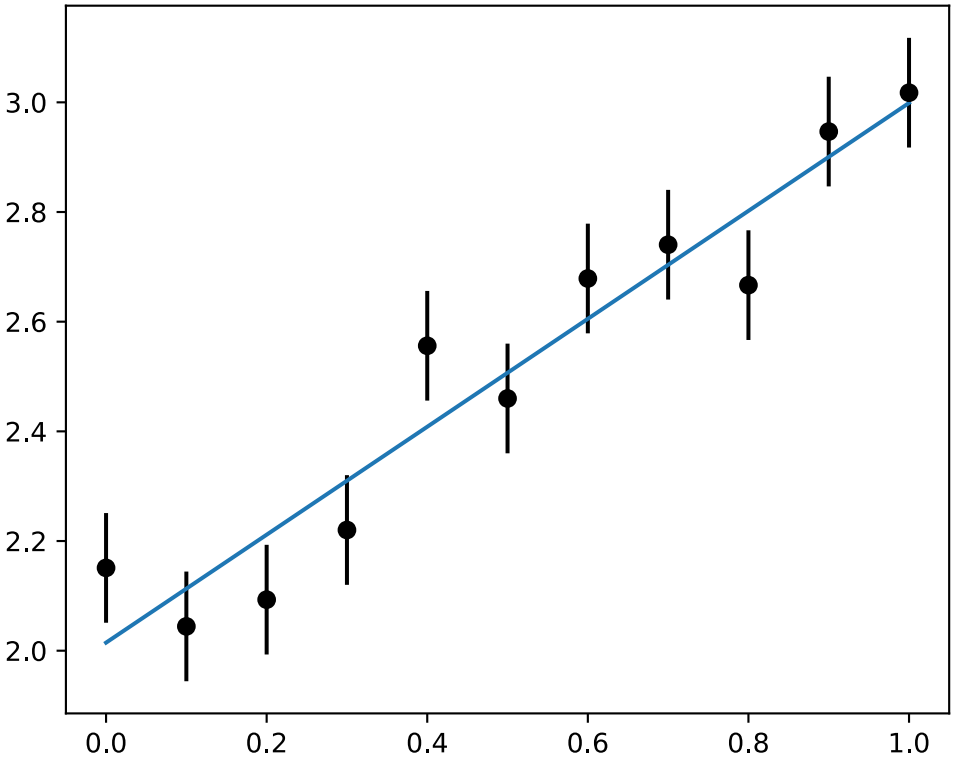
Out[41]:

Migrad

FCN = 9.7 (χ^2/ndof = 1.1)	Nfcn = 34
EDM = 9.86e-23 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	0.98	0.10					
1	c	2.01	0.06					

	m	c
m	0.00909	-0.0045 **(-0.845)**
c	-0.0045 **(-0.845)**	0.00318



The `Minuit.migrad()` method returns the Minuit instance so that one can chain method calls. The instance also prints the latest state of the minimization in a nice way as we can see in the output in teh previous cell. The informationis in three blocks.

First block of `Minuit.migrad()` information

The *first block* in this `Minuit.migrad()` output is showing information about the function minimum. This is good for a quick check:

All blocks should be green, other colours mean:

- Purple means something bad.
- Yellow is a warning, telling you that you need to be careful.

Let's see how it looks when the function is bad:

In [42]:
```
m_bad = Minuit(lambda x: 0, x=1)  # a constant function has no minimum
m_bad.errordef = 1  # avoid the errordef warning
m_bad.migrad()
```

Out[42]:

Migrad

FCN = 0	Nfcn = 80
EDM = 0 (Goal: 0.0002)	
INVALID Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse FAILED	Covariance NOT pos. def.

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	x	1	0					

Coming back to our line fit example, the info about the function minimum can be directly accessed with `Minuit.fmin`:

In [43]:
```
mline.fmin
```

Out[43]:

Migrad

FCN = 9.7 (χ^2/ndof = 1.1)	Nfcn = 34
EDM = 9.86e-23 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

In [44]:
```
# print(repr(...)) to see a detailed representation of the data object
print(repr(mline.fmin))
```

```
<FMin algorithm='Migrad' edm=9.858911035813475e-23 edm_goal=0.0002 errordef=1.0 fval=9.699698346688667 has_accurate_
covar=True has_covariance=True has_made_posdef_covar=False has_parameters_at_limit=False has_posdef_covar=True has_r
eached_call_limit=False has_valid_parameters=True hesse_failed=False is_above_max_edm=False is_valid=True nfcn=34 ng
rad=0 reduced_chi2=1.0777442607431853 time=0.0>
```

The most important one here is `is_valid`. If this is false, the fit does not converge and the result is useless. Since this value is so often looked at, a shortcut is provided with `Minuit.valid`.

If the fit fails, there is usually a numerical or logical issue.

Either:

- The fit function is not analytical everywhere in the parameter space, e.g. it has a discrete step;

or

- The fit function does not have a local minimum, e.g. the minimum may be at infinity or the extremum may be a saddle point or maximum.

Indicators for this are `is_above_max_edm=True`, `hesse_failed=True`, `has_posdef_covar=False`, or `has_made_posdef_covar=True`.

Possible problems are:

- Migrad reached the call limit before the convergence so that `has_reached_call_limit=True`. The used number of function calls is `nfcn`, and the call limit can be changed with the keyword argument `ncall` in the method `Minuit.migrad()`. Note that `nfcn` can be slightly larger than `ncall`, because Migrad internally only checks this condition after a full iteration, in which several function calls can happen.
- Migrad detects convergence by a small `edm` value, the estimated distance to minimum. This is the difference between the current minimum value of the minimized function and the prediction based on the current local quadratic approximation of the function, which is something that Migrad computes as part of its algorithm. If the fit does not converge, `is_above_max_edm` is true.

If you are interested in parameter uncertainties, you should make sure that:

- `has_covariance`, `has_accurate_covar`, and `has_posdef_covar` are `True`; and
- `has_made_posdef_covar` and `hesse_failed` are `False`.

We are working with least squares as our loss function, the function we minimise to find the best values of the parameters. With this loss function, each entry is weighted by the errors we provided so if these errors are the standard deviation of each

measurement then we are really minimising chi square (see week 3). As a result we can get this goodness-of-fit measure out of the `Minuit` object. For instance the value of the loss function, the number of degrees of freedom and the reduced chi square are obtained using `.fval` `.fval` `.fval` respectively.

For example the reduced chi square for the simple line fit is about what we would hope for a good (never perfect) fit.

In [45]:
```
# Have to come across the f strings in python?  A recent addition but you can live without it too.
print( f"Simple line fit has chi^2/dof = {mline.fval:.2f} / {mline.ndof:.0f} = {mline.fmin.reduced_chi2:.2f}")
```

```
Simple line fit has chi^2/dof = 9.70 / 9 = 1.08
```

Second block of `Minuit.migrad()` information

The *second block* that is printed with `Minuit.migrad()` after information about the fit minimum is the parameter list, which can also be directly accessed with `Minuit.params` .

A quick reminder about the information held on parameters. Suppose we have a Minuit object `m` . Then `m.params` is a tuple-like container of Param data objects so `m.params[p]` gives information about the p-th parameter. Important fields for each of these parameters are:

- `number` : parameter index.
- `name` : parameter name.
- `value` : value of the parameter at the minimum.
- `error` : uncertainty estimate for the parameter value.

For instance, `m.params[0].name` gives the name of the first parameter

In [46]:
```
display(mline.params)

for ppp in mline.params:
    print("Parameter index {:d} called {:s} has value {:.2f} +/- {:.2f}".format(ppp.number, ppp.name, ppp.value, ppp

print("Name of parameter 0 is "+mline.params[0].name)
```

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	m	0.98	0.10					
1	c	2.01	0.06					

```
Parameter index 0 called m has value 0.98 +/- 0.10
Parameter index 1 called c has value 2.01 +/- 0.06
Name of parameter 0 is m
```

The accuracy of the uncertainty estimate depends on two factors: the correct mathematical modeling of the fitting problem and the appropriate usage of the `errordef` value in Minuit. By "correct mathematical modelling of the fitting problem", we mean have we chosen an appropriate model to describe our data; we can check how well our model with its fitted parameters describe the data by calculating the reduced chi-squared statistic, as we discussed in Week 3.

Third block of `Minuit.migrad()` information

The *last block* that gets printed by running `Minuit.migrad()` shows the covariance matrix, which can also be directly accessed with `Minuit.covariance`. This is useful to check for large correlations which are usually a sign of trouble.

In [47]:
```
mline.covariance
```

Out[47]:

	m	c
m	0.00909	-0.0045 **(-0.845)**
c	-0.0045 **(-0.845)**	0.00318

Exercise 2

Now that we have introduced how to investigate the fit status, go back to your polynomial fits from before. As you increase the order of the polynomial how does the goodness of fit change? Can you now tell by some goodness of fit measure when the fit shows signs of overfitting?

In [48]:
```
# Your code here
```

End of exercise 2

Parameter uncertainties, covariances and confidence intervals

You will have seen when we did our simple example, we called `Minuit.hesse` after running the minimisation, to calculate the error accurately. In fact, `iminuit` has two different methods for calculating the uncertainty on parameter estimates, referred to as Hesse and Minos. These two algorithms sound fancy, but they just correspond to the two different methods we saw earlier:

- Hesse numerically calculates the second derivative of the cost function with respect to the parameters to form a matrix then inverts it to find the error matrix (called the Hesse matrix)
- Minos examines the cost function near the minimum to find where the cost function changes by `errordef`, a value specific to the cost function. This can give asymmetric errors.

The two main cost functions we use are the chi-squared and the negative log-likelihood (so it is still a minimisation problem). The `errordef` parameter is equal to 1 for the chi-squared, and 0.5 for the negative log-likelihood, as we would expect from our discussion of fitting earlier.

We can easily visualise the covariance matrix:

In [49]:
```
mline.covariance
```

Out[49]:

	m	c
m	0.00909	-0.0045 **(-0.845)**
c	-0.0045 **(-0.845)**	0.00318

After we have calculated the covariance matrix by running `Minuit.hesse`, we can also find the correlation matrix:

In [50]:
```
mline.covariance.correlation()
```

Out[50]:

	m	c
m	1	-0.8
c	-0.8	1

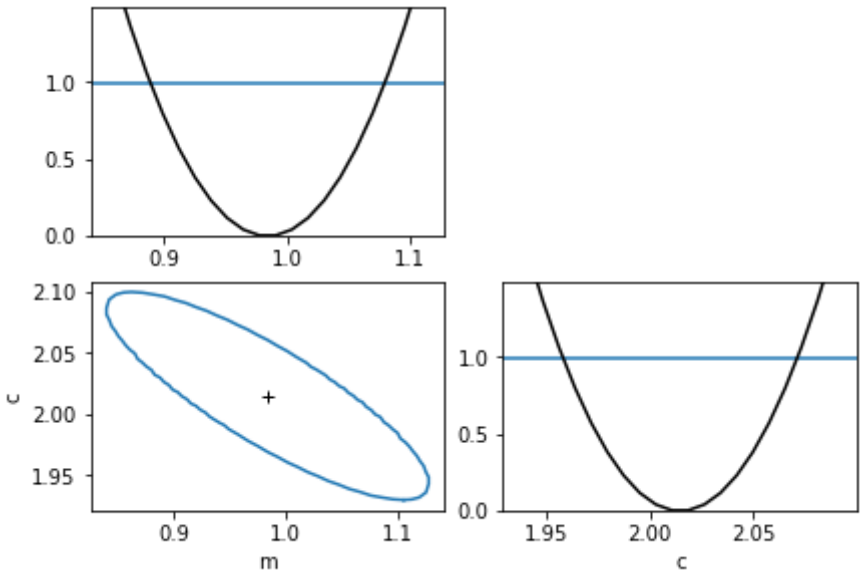
Correlation is in generally not necessarily bad, but if you can redefine the parameters of the fit function it is generally good to choose parameters which are not strongly correlated.

In general, it is best to use `hesse` by default to calculate parameter uncertainties and we will aim to stick with this for the remainder of the course.

It is also possible to draw confidence regions easily using `iminuit`; for example, we can draw a matrix of contours for the 1σ confidence interval for all parameters:

In [51]:
```
mline.draw_mnmatrix()
```

Out[51]:
```
(<Figure size 432x288 with 4 Axes>,
 array([[<AxesSubplot:>, <AxesSubplot:>],
        [<AxesSubplot:xlabel='m', ylabel='c'>, <AxesSubplot:xlabel='c'>]],
       dtype=object))
```

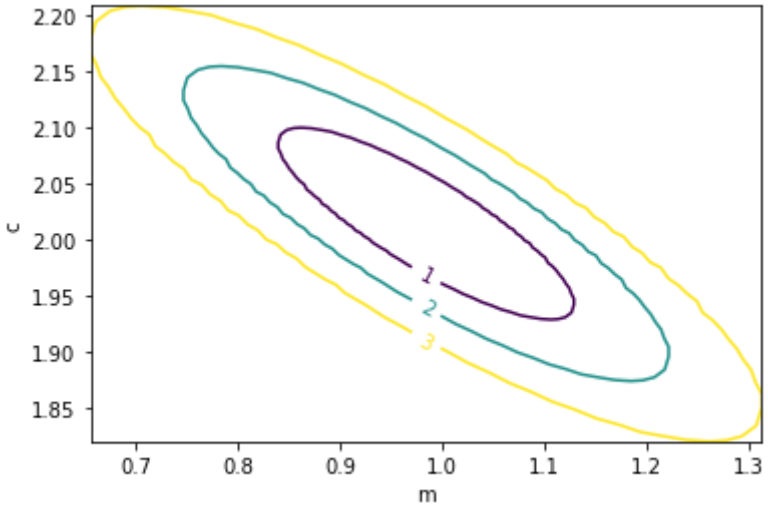


The top-left to bottom-right diagonal plots show the 1D profiles around the minimum of each parameter, while the other plots show the 2D contour in the two parameters.

We can also draw countours at specified confidence levels, such as the levels corresponding to 1, 2 and 3 σ in the Gaussian case, using `Minuit.draw_mncontour`:

In [52]:
```
# CARE. mline.contour is closely related but different. Be sure you understand what these are doing.

mline.draw_mncontour("m","c",cl=(1,2,3))
```

Out[52]:
```
<matplotlib.contour.ContourSet at 0x1eb0fa16950>
```

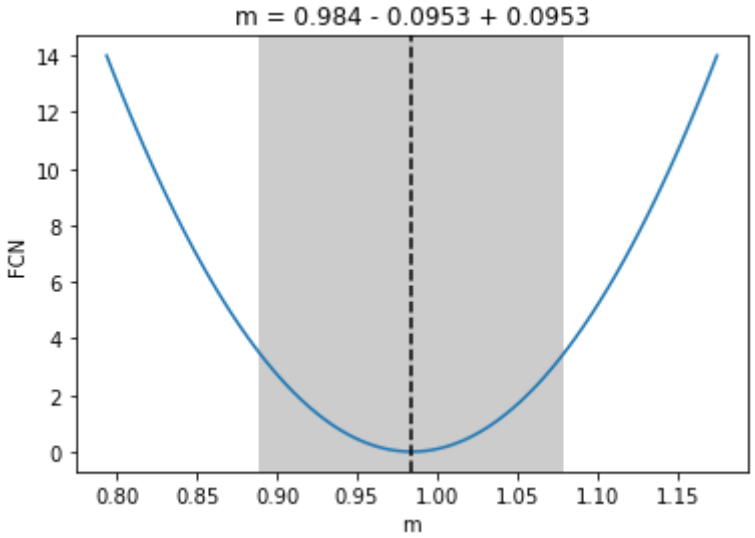


We can draw the 1D profiles using `Minuit.draw_profile`:

In [53]:
```
mline.draw_profile("m")
```

Out[53]:
```
(array([0.79350571, 0.79735809, 0.80121046, 0.80506284, 0.80891521,
        0.81276759, 0.81661996, 0.82047233, 0.82432471, 0.82817708,
        0.83202946, 0.83588183, 0.8397342 , 0.84358658, 0.84743895,
        0.85129133, 0.8551437 , 0.85899607, 0.86284845, 0.86670082,
        0.8705532 , 0.87440557, 0.87825794, 0.88211032, 0.88596269,
        0.88981507, 0.89366744, 0.89751982, 0.90137219, 0.90522456,
        0.90907694, 0.91292931, 0.91678169, 0.92063406, 0.92448643,
        0.92833881, 0.93219118, 0.93604356, 0.93989593, 0.9437483 ,
        0.94760068, 0.95145305, 0.95530543, 0.9591578 , 0.96301017,
        0.96686255, 0.97071492, 0.9745673 , 0.97841967, 0.98227205,
        0.98612442, 0.98997679, 0.99382917, 0.99768154, 1.00153392,
        1.00538629, 1.00923866, 1.01309104, 1.01694341, 1.02079579,
        1.02464816, 1.02850053, 1.03235291, 1.03620528, 1.04005766,
        1.04391003, 1.0477624 , 1.05161478, 1.05546715, 1.05931953,
        1.0631719 , 1.06702428, 1.07087665, 1.07472902, 1.0785814 ,
        1.08243377, 1.08628615, 1.09013852, 1.09399089, 1.09784327,
        1.10169564, 1.10554802, 1.10940039, 1.11325276, 1.11710514,
        1.12095751, 1.12480989, 1.12866226, 1.13251463, 1.13636701,
        1.14021938, 1.14407176, 1.14792413, 1.15177651, 1.15562888,
        1.15948125, 1.16333363, 1.167186  , 1.17103838, 1.17489075]),
 array([1.39985715e+01, 1.34386287e+01, 1.28901132e+01, 1.23530252e+01,
        1.18273645e+01, 1.13131313e+01, 1.08103254e+01, 1.03189470e+01,
        9.83899599e+00, 9.37047238e+00, 8.91337616e+00, 8.46770735e+00,
        8.03346595e+00, 7.61065195e+00, 7.19926536e+00, 6.79930617e+00,
        6.41077439e+00, 6.03367002e+00, 5.66799305e+00, 5.31374348e+00,
        4.97092132e+00, 4.63952657e+00, 4.31955922e+00, 4.01101927e+00,
        3.71390673e+00, 3.42822160e+00, 3.15396387e+00, 2.89113355e+00,
        2.63973063e+00, 2.39975512e+00, 2.17120701e+00, 1.95408631e+00,
        1.74839302e+00, 1.55412713e+00, 1.37128864e+00, 1.19987756e+00,
        1.03989389e+00, 8.91337616e-01, 7.54208752e-01, 6.28507293e-01,
        5.14233240e-01, 4.11386592e-01, 3.19967349e-01, 2.39975512e-01,
        1.71411080e-01, 1.14274053e-01, 6.85644320e-02, 3.42822160e-02,
        1.14274053e-02, 0.00000000e+00, 1.14752652e-12, 1.14274053e-02,
        3.42822160e-02, 6.85644320e-02, 1.14274053e-01, 1.71411080e-01,
        2.39975512e-01, 3.19967349e-01, 4.11386592e-01, 5.14233240e-01,
        6.28507293e-01, 7.54208752e-01, 8.91337616e-01, 1.03989389e+00,
        1.19987756e+00, 1.37128864e+00, 1.55412713e+00, 1.74839302e+00,
        1.95408631e+00, 2.17120701e+00, 2.39975512e+00, 2.63973063e+00,
        2.89113355e+00, 3.15396387e+00, 3.42822160e+00, 3.71390673e+00,
        4.01101927e+00, 4.31955922e+00, 4.63952657e+00, 4.97092132e+00,
        5.31374348e+00, 5.66799305e+00, 6.03367002e+00, 6.41077439e+00,
        6.79930617e+00, 7.19926536e+00, 7.61065195e+00, 8.03346595e+00,
        8.46770735e+00, 8.91337616e+00, 9.37047238e+00, 9.83899599e+00,
```

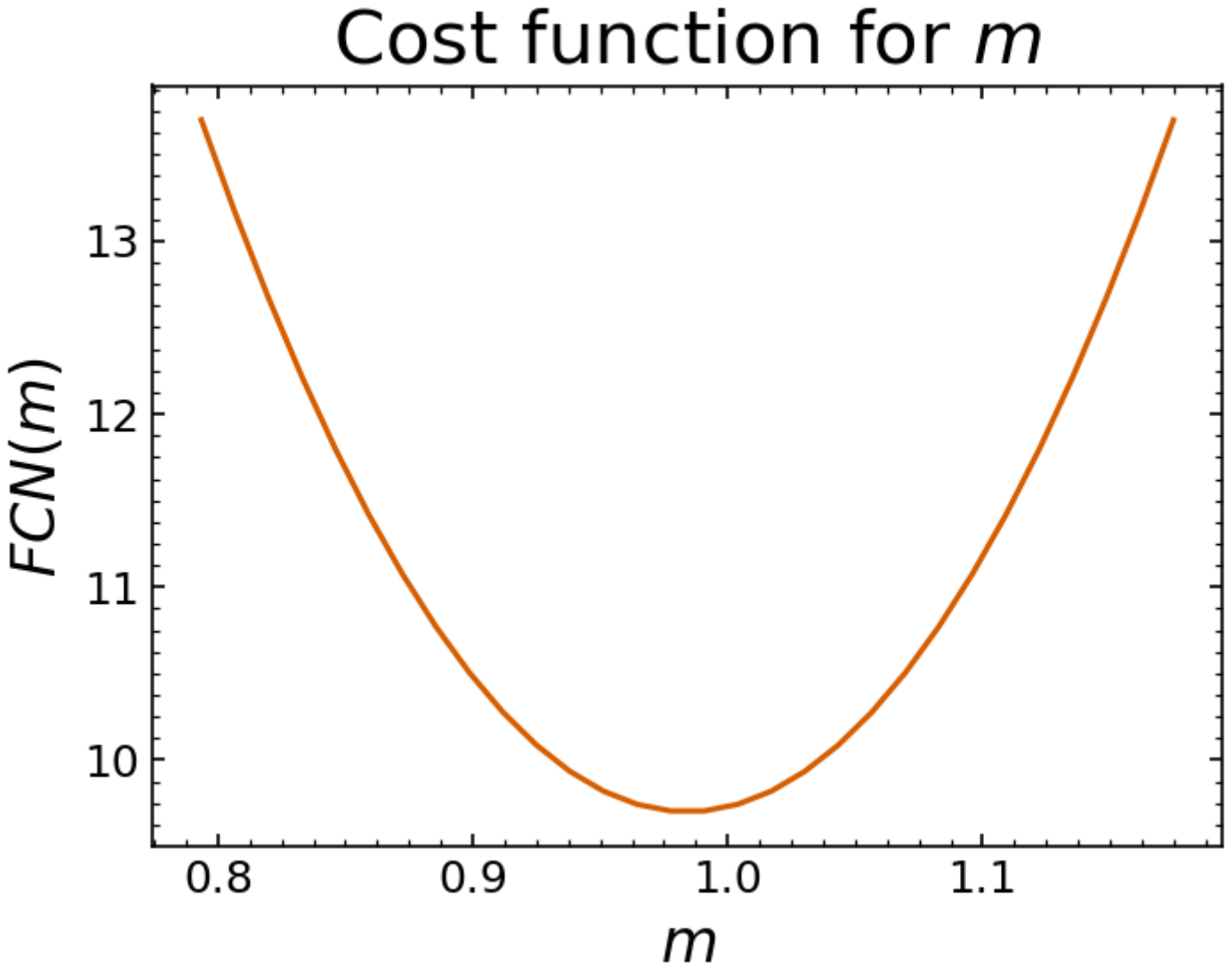
```
1.03189470e+01, 1.08103254e+01, 1.13131313e+01, 1.18273645e+01,
1.23530252e+01, 1.28901132e+01, 1.34386287e+01, 1.39985715e+01]))
```

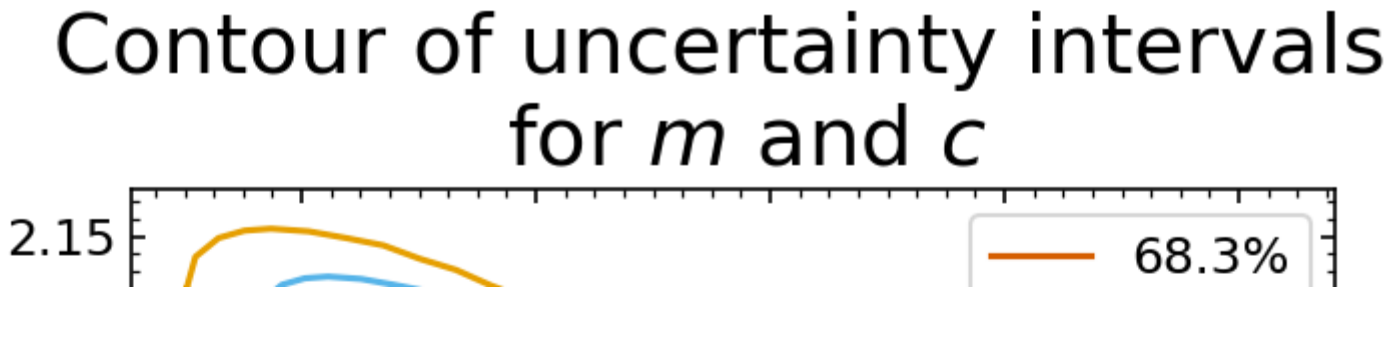


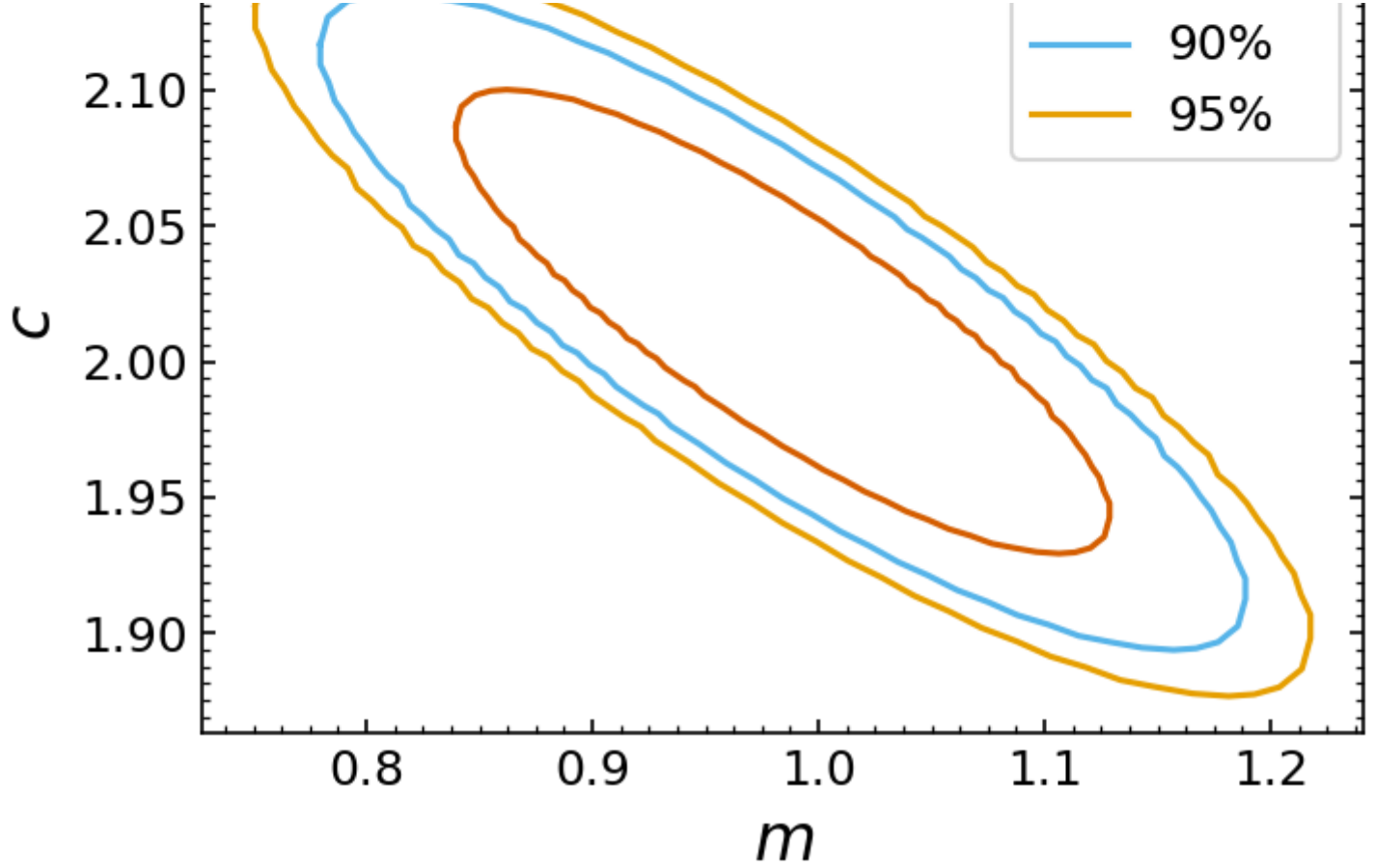
If you want to get the profiles or contours to plot yourself (to make it easier to improve formatting etc), you can use `Minuit.mnprofile` and `Minuit.mncontour` respectively:

In [54]:
```
# Colours are from Okabe_Ito list, see appendix week 1.
m_h, Lm_h, _ = mline.mnprofile("m")

pts_1 = mline.mncontour("m","c",cl=0.683, size = 100)
pts_2 = mline.mncontour("m","c",cl=0.9,size = 100)
pts_3 = mline.mncontour("m","c",cl=0.95,size = 100)
fig, ax = plt.subplots(2,1,figsize = (5,8),dpi = 150)
ax[0].plot(m_h,Lm_h,color='#D55E00')
ax[0].set_xlabel('$m$', fontsize = 16)
ax[0].set_ylabel(r'$FCN(m)$',fontsize = 16)
ax[0].tick_params(direction='in',top=True,right=True,which='both',labelsize = 12)
ax[0].xaxis.set_minor_locator(MultipleLocator(0.0125))
ax[0].yaxis.set_minor_locator(MultipleLocator(0.125))
ax[0].set_title('Cost function for $m$',fontsize = 20)

ax[1].plot(pts_1.T[0],pts_1.T[1],color='#D55E00',label = '68.3%')
ax[1].plot(pts_2.T[0],pts_2.T[1],color='#56B4E9',label = '90%')
ax[1].plot(pts_3.T[0],pts_3.T[1],color='#E69F00',label = '95%')
ax[1].set_xlabel('$m$',fontsize = 16)
ax[1].set_ylabel('$c$',fontsize = 16)
```

```
ax[1].tick_params(which='both',direction='in',top = True, right = True,labelsize = 12)
ax[1].xaxis.set_minor_locator(MultipleLocator(0.0125))
ax[1].yaxis.set_minor_locator(MultipleLocator(0.025/4))
ax[1].legend(loc='upper right',fontsize = 12)
ax[1].set_title('Contour of uncertainty intervals \n for $m$ and $c$',fontsize = 20)

fig.tight_layout()
```


Cost function for m
$FCN(m)$
13
12
11
10
0.8
0.9
1.0
1.1
m


Contour of uncertainty intervals for m and c
2.15
68.3%



Section 5: Curve Fitting and Binning, Maximum-Likelihood and `iminuit` ^

So far, we have not used binned data. All our data was set of (X, y) points and we compared that to a function $f(X)$. The comparison has always used the least squares, the chi square statistics, as a way to assess how good the fit was. In this section, we will consider the effect of binning data, something which you might want to do whatever method you use when fitting data.

While considering this aspect we will also work consider an alternative approach to the least square method, the popular maximum likelihood method.

So far, we have looked at simple fits using `LeastSquares`, which is the `iminuit` implementation of the chi-squared parameter estimation method. In practice, many applications use maximum-likelihood estimation, which is also implemented in several ways in `iminuit`.

We will discuss both unbinned and binned maximum-likelihood fitting using `iminuit`. These differ as follows:

- Unbinned fits apply directly to the raw data sample. The easiest to use, but can become slow when the sample size is large.
- Binned fits are applied to binned data, which means you must appropriately bin your sample. The binning has to be fine enough to retain all the essential information. These are in general much faster than unbinned fits.

We will discuss how we can do both of these using `iminuit` in turn. We will demonstrate both methods using the same dataset which is a standard example from high-energy physics, the fit of a peak over a smooth background. We will assume a constant background.

Counts and Binning

The previous section worked with raw data in the form of a set of (X_i, y_i) measurements where i is the index of each measurement. We may also have an estimate of the errors in y_i. A typical example would be the data on our pendulum where we are given data as $(L_i, T_i, \Delta T_i)$ so $X = L_i$ is the independent variable (we set the length of the pendulum in the experiment), $T_i = y_i$ is the dependent variable (we find out what this given L_i) and ΔT_i is our estimate of the error in our measurement. (Aside: so far we have ignored the fact that there will also be an error in our measurement of length too but that question is for another day or see how I sidestepped this in one of the exercises in week 3. Lets just assume in this course that the errors in X_i are always negligible.)

Independent events

Suppose though, that our data is in the form of events. Formally $X_i = i$, each event is simply the next independent event, and all we do is measure some property y of each event. Our data is then simply the list of y_i values as the order in which they came tells us nothing.

This is very common in wider science. For instance in medical studies we might have a series of independent subjects in our medical trial, and we measure some property of each subject, e.g. the height.

In physics, the classic example is particle physics. At CERN, each collision is captured in the detectors independenly from every other collision. The physicists will look for, say, the energy of a particular particle emerging from the collision (yes, there are lots of particles emerging so there is a lot of work to get to the simple data we will use in the exercises here). The data then is simply a list of the energies of particles.

Our predictions come in the form of a probability distribution function $f(X;\theta)$ that for a given theory where θ represent a set of parameters in our theory we want to find e.g. the mass of predicted new particles. In particle physics a lot of difficult calculations (see QFT, Unification and APP courses in fourth year physics or all the courses in the QFFF MSc), tell us the probability $f(X;\theta)dX$ of finding our particle between energies X and $X+dX$.

Now we could just use the KS test to decide if the cdf $F(X;\theta)$ of our theory is a good fit to the data. We construct a CDF of the data by counting the fraction of events $F_{\text{data}}(X;\theta)$ that have energy less than X. In this context, we would vary the parameters θ in our theory (e.g. masses of predicted particles) until we found the best fit (minimum D for the KS test).

Definition of Binned Data

A popular way to deal with such data of events is to gather up the data up into **bins**. That is we work with the number of events $n(b)$ in bin b where the index b runs from 1 to the number of bins. Bin b is defined to contain all the events whose energy y_i lies between X_{b-1} and X_b. Note the change of notation is because we decide where the bins are, the X_b are independent variables set up and chosen by us. The universe decides what is the energy of each particle measured will be, the y_i in the original data are dependent variables. Also note that the number of these **bin edges** X_b is one more than the number of bins.

Now we compare the pairs $(b,n(b))$ to the prediction $N(F(X_b;\theta)-F(X_{b-1};\theta))$ where N is the total number of events we have, so $N=\sum_b n(b)$. (EFS: why do we need the cdf here?) So we have now reduced the problem to onw of a series of (X,y) measurements as we had in the previous section, where the bins (or some property of each bin such as the index or mean of the bin edge values) play the role of our independent variable X and the counts $n(b)$ play the role of the dependent variables y.

Pros and Cons of Binning Data

A general rule in statistics is to try and do a little manipulation of the data as possible. Try and work with the raw data to avoid introducing additional problems. For instance, with bins a big question that can have a significant effect on results is what bins

to choose. By definition, binning reduces the number of data points you have so you have lost information on your system. You do not always know if that was important information you lost or mostly irrelevant noise.

So unbinned fits are ideal when the data sample isn't too large or very high dimensional. If your statistical tools are appropriate, finding a genuine signal in the data is exactly what they should be able to do.

The downside of unbinned data is that these can be very inefficient for large samples and can even become numerically unstable. So binning out of necessity is an acceptable reason.

Approximating the predicted count in a bin

Finally note that using the cdf to find the predicted value from a theory for each bin can be time consuming. So many people use the following *approximation* often without comment (but you will all remember that this is an approximation for that time when it makes a difference).

- Let p_b be the predicted value for the probability of finding an event in bin b.
- Then p_b is approximately equal to the the width of bin b, $(X_b - X_{b-1})$, multiplied by the typical probability density of that bin.
- Assuming bins are not too wide so the PDF does vary much across the bin, we can use the probability density in the centre of the bin at $\bar{X}_b = (X_b + X_{b-1})/2$ as a good measure of the probability density in the bin, that is $f(\bar{X}_b; \theta)$.
- If we have N events in total then the approximate prediction for the number of events in the bin will be Np_b.

To summarise, we often use (as I do in the exercises at the end of the notebook) the following formula

$$n(b) \approx (X_b - X_{b-1})\, f(\bar{X}_b; \theta)\,, \quad \text{where} \quad \bar{X}_b = (X_b + X_{b-1})/2\,.$$

EFS: Optional. This is simple to derive using the Taylor expansion. The derivation is given as an appendix here too.

Choosing your bins

There is no perfect way to choose bins, so no simple set of rules I can give you. There are, however, many ways to work with badly binned data. So all I can give is a few suggestions and we will look at one example in the exercises later.

The main rule is to avoid bins that have counts that are too low. They will be noisy and so tell us little. In fact their noise may hinder the numerical process to find the good fit that is there for the data.

In particular if you assume that the counts in each bin follow Poisson statistics rather than Binomial statistics (a common simplification), then try not to have less than a count of 5 events in any one bin. This is not an exact number (10 might be better) but the message is that the Poisson approximation is poor for bins with low numbers as the variation is large. So again it is a low count/ high noise issue.

Bins with zero counts are a particular problem. People often ignore them but having nothing in a bin is non-trivial information ("The dog did nothing in the night-time.", "That was the curious incident," remarked Sherlock Holmes. The Hound of the Baskervilles). Including them (empty bins, not quotes from Shelock Holmes) without any changes to your code may cause errors as most codes are designed for large or at least positive numbers. Including them is possible but may require a lot of work and pehaps it does not help you much (see discussion of low count bins again).

So an obvious route forward is to choose fewer bins with larger widths. Clearly, if the bins are too large (everything in one bin) you learn nothing. You may miss a small signal which gets hidden under the large numbers of wide bins. So there is some unquantifiable sweet spot between too many bins and too few bins.

Other tricks might include dropping some bins at the extreme end of the data. That might be justified if these contain few events anyway, they are noisy.

You do not have to use bins of the same sizes. When the counts get much rarer very fast as X gets bigger, then *logarithmic binning* is useful. This is where we set the bin boundaries to satisfy $X_b = rX_{b-1}$ for some ratio $r > 1$. Perhaps having one large bin for everything over a certain value of X is useful.

Maximum Likelihood

Apart from binning, the other theme of this section is the **Maximum Likelihood** approach to parameter estimation and curve fitting. This gives an alternative to the least squares method we considered above and we minimise a different loss function to find the best value for parameters. Again, see the notes from Statistics and Mesaurement course (also provided on Blackboard as part of this course) for details.

A quick reminder. Suppose we have a set of events where for each event i we measure some quantity x_i e.g. for the i-th collision at CERN we measure the energy x_i of some particle. (Note I have flipped my notation as everyone talks in terms of random variable x_i being measured at each event. I will now follow the same convention).

For each event our theory predicts that this event i happens with probability p_i where $p_i = f(x_i;\theta)dy$ if we have a continuous variable x or simply $f(x_i;\theta)$ if it is discrete. The probability $P_N(\theta)$ predicted by our theory with parameters θ that in N measurements we would have found the x_i data values actually found in the experiment is simply the product of the individual p_i. That is

$$P_n(\theta) = \prod_{i=1}^{N} p_i = \prod_{i=1}^{N} f(x_i;\theta)dx_i \,.$$

Note we set dx_i to be 1 for the discrete case. The best fit of our theory to the data comes from the parameters θ that maximise this probability.

Numerically is is much more convenient to minimise the negative log of this function which is guaranteed to give the same answer. So the cost function we use is the (negative) log likelihood L where

$$L = -\log(P_n(\theta)) = \sum_{i=1}^{N} \log(f(y_i;\theta)) \,.$$

Aside. The more observant will notice that we have dropped a term equal to the sum of $\log(dx_i)$. This is not present for discrete valued x ($dx = 1$ in this case). For continuous variables, this dx_i can be considered some sort of unceratinty. If it is the same for all variables, these add an irrelevant constant. In more sophisticated problems, we might need to reexamine this aspect more carefully. However, in all the problems we consider, we will use the form for L given here.

Example data

Let us start by generating some data we will use in the following examples. Note how we use binning to visualise this data here but the data is actually from a continuous distribution. We could have shown the cumulative distribution of the data and of the theoretical function to avoid binning.

To perform an unbinned fit, you must provide the (vectorized) PDF of the model i.e. a numpy function or a scipy function. The PDF must also be normalised.

We will consider an example of a Gaussian peak over a constant background. The corresponding model PDF is the weighted sum of the normal and uniform PDFs. We will only consider X values between 0 and 1, so the uniform component is parameter-

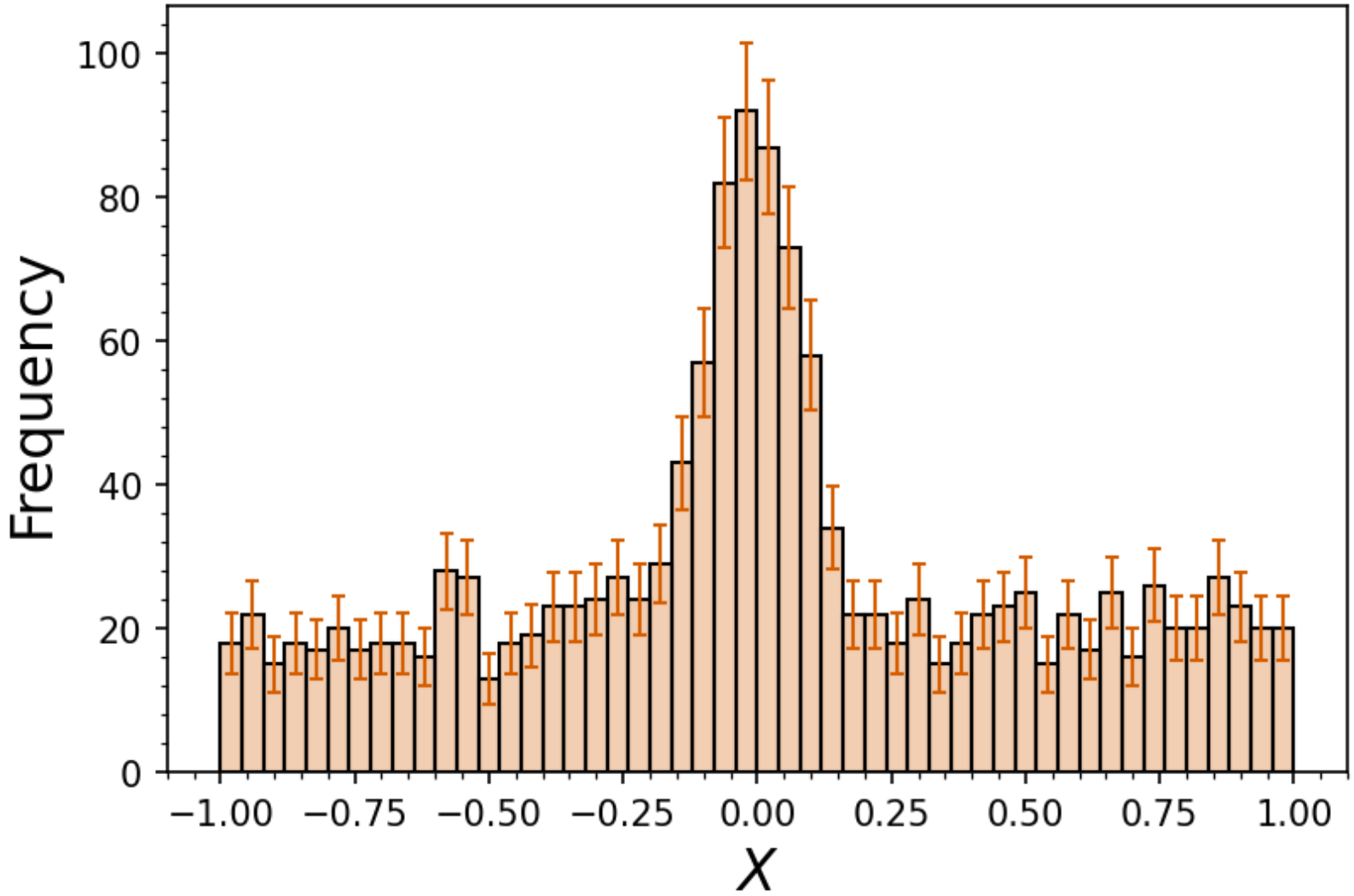
free. The parameters of the model are therefore μ and σ of the normal distribution, and z, the weighting factor which takes values between zero and one. We can write the total PDF as

$$f(X; z, \mu, \sigma) = z f_{\text{normal}}(X; \mu, \sigma) + (1 - z) f_{\text{uniform}}(X)$$

where $f_{\text{normal}}(X; \mu, \sigma)$ and $f_{\text{uniform}}(X)$ denote the normal and uniform PDFs respectively.

In [55]:
```
from scipy.stats import norm, uniform
from matplotlib.colors import to_rgba
from matplotlib.ticker import MultipleLocator

xrange = -1, 1
rng = np.random.default_rng(1)

mu_value = 0.0
sigma_value = 0.1
n_normal = 400
n_uniform = 1000
z_value =  n_normal/(n_normal+n_uniform)
print("Data generated using mu={:.3f}, sigma={:.3f},  z={:.3f} ".format(mu_value,sigma_value,z_value))
# EFS: we don't stop the Gaussian from ranging outside -2 to 2 range. Does it matter?
x_data = rng.normal( mu_value, sigma_value, size = n_normal)
x_data = np.append(x_data, rng.uniform(*xrange, size = n_uniform))

# n ,xe=
counts, bin_edges = np.histogram(x_data, bins = 50, range = xrange)
x_bin_centre = 0.5*(bin_edges[1:]+bin_edges[:-1])
dx = np.diff(bin_edges)

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.bar(x_bin_centre, counts, yerr = counts**0.5,
       width = dx,
       fc=to_rgba('#D55E00',0.3),edgecolor='black',
       error_kw={'ecolor':'#D55E00','capsize':2,'elinewidth':1,'capthick':1})
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('Frequency',fontsize = 16)
ax.xaxis.set_minor_locator(MultipleLocator(0.05))
ax.yaxis.set_minor_locator(MultipleLocator(4))
```

```
Data generated using mu=0.000, sigma=0.100,  z=0.286
```



Unbinned maximum-likelihood fits

We will use our mixture of data drawn from either a Gaussian or a uniform distribution.

First we need to set limits on our parameters to prevent this problem from becoming mathematically undefined. In this case, we want our signal to be within the range $-1 < X < 1$, and so we require that

$$
\begin{aligned}
0 < z < 1 \\
-1 < \mu < 1 \\
\sigma > 0
\end{aligned}
$$

Now we have everything, we can try the fit. This uses the `UnbinnedNLL` ("Unbinned Negative Log-Likelihood) function fro iminuit.

In [56]:
```
from iminuit.cost import UnbinnedNLL

def model_pdf(x, z, mu, sigma):
    return (z*norm.pdf(x, mu, sigma) + (1-z)*uniform.pdf(x, xrange[0], xrange[1]-xrange[0]))

cost_function = UnbinnedNLL(x_data, model_pdf) # only need to pass the samples we have

m_obj = Minuit(cost_function, z = 0.4, mu = 0, sigma = 0.2) # arbitrary starting parameters
m_obj.limits['z'] = (0,1) # Set limit on z
m_obj.limits['mu'] = (-1, 1) # Set limit on mu
m_obj.limits['sigma'] = (0, None) # Set lower limit on sigma

m_obj.migrad()
```

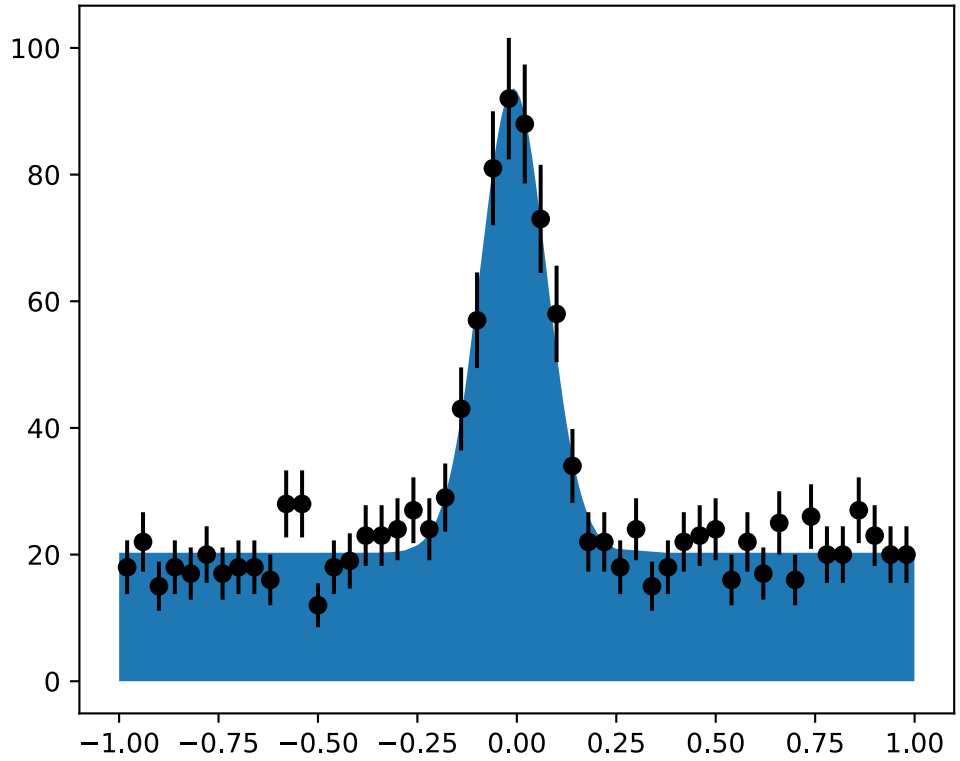
Out[56]:

Migrad

FCN = 1504	Nfcn = 83
EDM = 3.42e-05 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

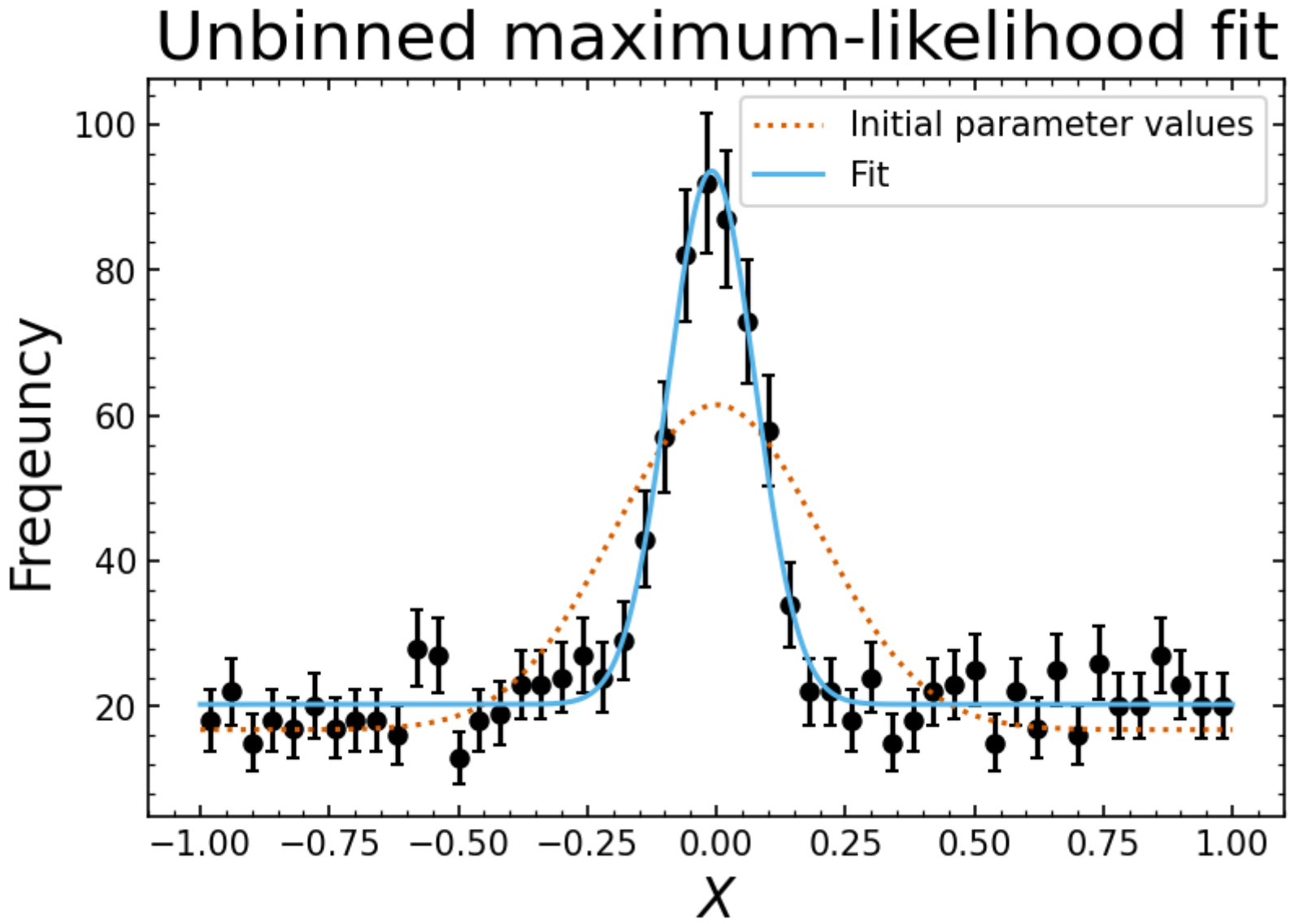
	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	z	0.275	0.017			0	1	
1	mu	-0.009	0.006			-1	1	
2	sigma	0.084	0.006			0		

	z	mu	sigma
z	0.000298	-0 **(-0.034)**	0.037e-3 **(0.381)**
mu	-0 **(-0.034)**	3.79e-05	-0.002e-3 **(-0.066)**
sigma	0.037e-3 **(0.381)**	-0.002e-3 **(-0.066)**	3.22e-05



We can then visualise the result a little better.

In [57]:
```
fig, ax = plt.subplots(1,1,figsize=(6,4),dpi = 150)
ax.errorbar(x_bin_centre, counts, counts**0.5, fmt='ok',ms = 5,capsize = 2,zorder = 0)

xm = np.linspace(bin_edges[0],bin_edges[-1],1000)
ax.plot(xm, model_pdf(xm, *[p.value for p in m_obj.init_params])*len(x_data)*dx[0],
            ls = ':', label='Initial parameter values',color='#D55E00',zorder = 1)
ax.plot(xm, model_pdf(xm, *m_obj.values)*len(x_data)*dx[0],
            label='Fit',color='#56B4E9',zorder = 1)
ax.legend(loc='upper right')
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('Freqeuncy',fontsize = 16)
ax.set_title('Unbinned maximum-likelihood fit',fontsize = 20)
ax.xaxis.set_minor_locator(MultipleLocator(0.05))
ax.yaxis.set_minor_locator(MultipleLocator(4))
ax.tick_params(direction='in',which='both',top=True,right=True)
```



Finally, we can look at the uncertainty in the best fit parameters. Comparing with the values used to generate the data, we can see the best fit values are statistically consistent.

We could go through all the hypothesis testing, p-values etc if we wanted to make this conclusion more precise.

In [58]:
```
# Calculate error using hesse

hhh = m_obj.hesse() # calculate parameter uncertainties

for param in m_obj.params:
    print('{} = {:.2f} +/- {:.2f}'.format(param.name, param.value, param.error))
```

```
z = 0.28 +/- 0.02
mu = -0.01 +/- 0.01
sigma = 0.08 +/- 0.01
```

Binned maximum-likelihood fits

When we have large data samples, binned fits are more computationally efficient and numerically stable. However, you have to choose an appropriate binning, which is non-unique.

For our example, we have a reasonably large sample (1400 events) and 50 bins is sufficiently fine to retain all information without having too few counts per bin. The maximum likelihood method applied to binned data gives correct results even if bins have no entries, so a binning that is very fine is not a problem and only increases computational cost.

The cost functions for binned fits that are implemented in Minuit assumes that the bin counts are independently Poisson distributed about some unkown expected value per bin. This is correct for ordinary histograms.

For a given PDF $f(X;\boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$, we can write the expected count λ_i in a given bin with boundaries a_i and b_i as

$$\lambda_i = \int_a^b f(X;\boldsymbol{\theta}) dX$$

The cost function for a binned maximum-likelihood fit is the sum of the logarithms of the Poisson probability for the count in each bin, as a function of the expected counts λ_i. This is again multiplied by -1 to go from a maximisation problem to a minimisation problem. However, instead of supplying a PDF we must provide a CDF in this case. We can approximate the CDF as bin-width multiplied by the PDF evaluated at the bin center if it is difficult to calculate, but this is only an approximation. Using an exact CDF is better.

Now we know what we need, we can define the CDF and run the fit:

In [59]:

```
from iminuit.cost import BinnedNLL

def model_cdf(xe, z, mu, sigma):
    return (z * norm.cdf(xe, mu, sigma) +
            (1-z) * uniform.cdf(xe, xrange[0], xrange[1] - xrange[0]))

bnll_cost_function = BinnedNLL(counts, bin_edges, model_cdf)

bnll_mobj = Minuit(bnll_cost_function, z = 0.4, mu = 0, sigma = 0.2)

bnll_mobj.limits['z'] = (0,1)
bnll_mobj.limits['mu'] = (-1,1)
bnll_mobj.limits['sigma'] = (0, None)

bnll_mobj.migrad()
```

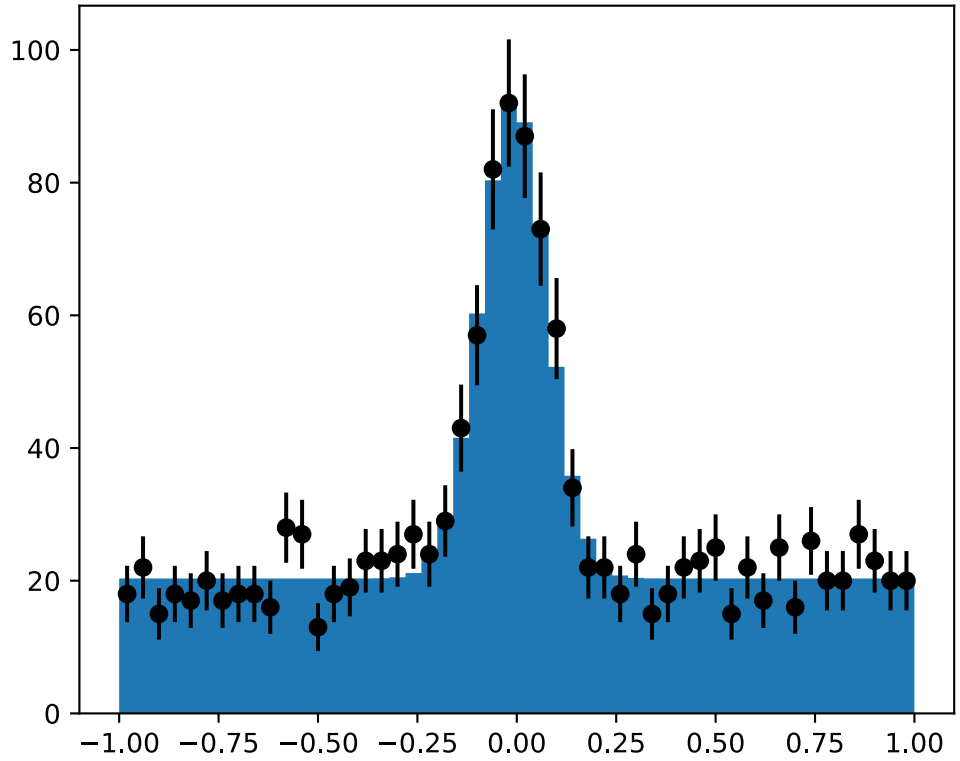
Out[59]:

Migrad

FCN = 29.74 (χ^2/ndof = 0.6)	Nfcn = 82
EDM = 1.88e-05 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	z	0.274	0.017			0	1	
1	mu	-0.008	0.006			-1	1	
2	sigma	0.083	0.006			0		

	z	mu	sigma
z	0.000302	-0 **(-0.045)**	0.041e-3 **(0.395)**
mu	-0 **(-0.045)**	3.9e-05	-0.003e-3 **(-0.088)**
sigma	0.041e-3 **(0.395)**	-0.003e-3 **(-0.088)**	3.49e-05



In [60]:
```
# Calculate error using hesse
hhh = bnll_mobj.hesse() # calculate parameter uncertainties

for param in bnll_mobj.params:
    print('{} = {:.2f} +/- {:.2f}'.format(param.name, param.value, param.error))
```

```
z = 0.27 +/- 0.02
mu = -0.01 +/- 0.01
sigma = 0.08 +/- 0.01
```

The parameter values and uncertainty estimates are not identical to the unbinned fit, but they are very close. In statistical terms, our results are equivalent. This shows that the binning we have chosen is good enough to reveal the essential information in the data.

We can then plot the result in a nicer format.

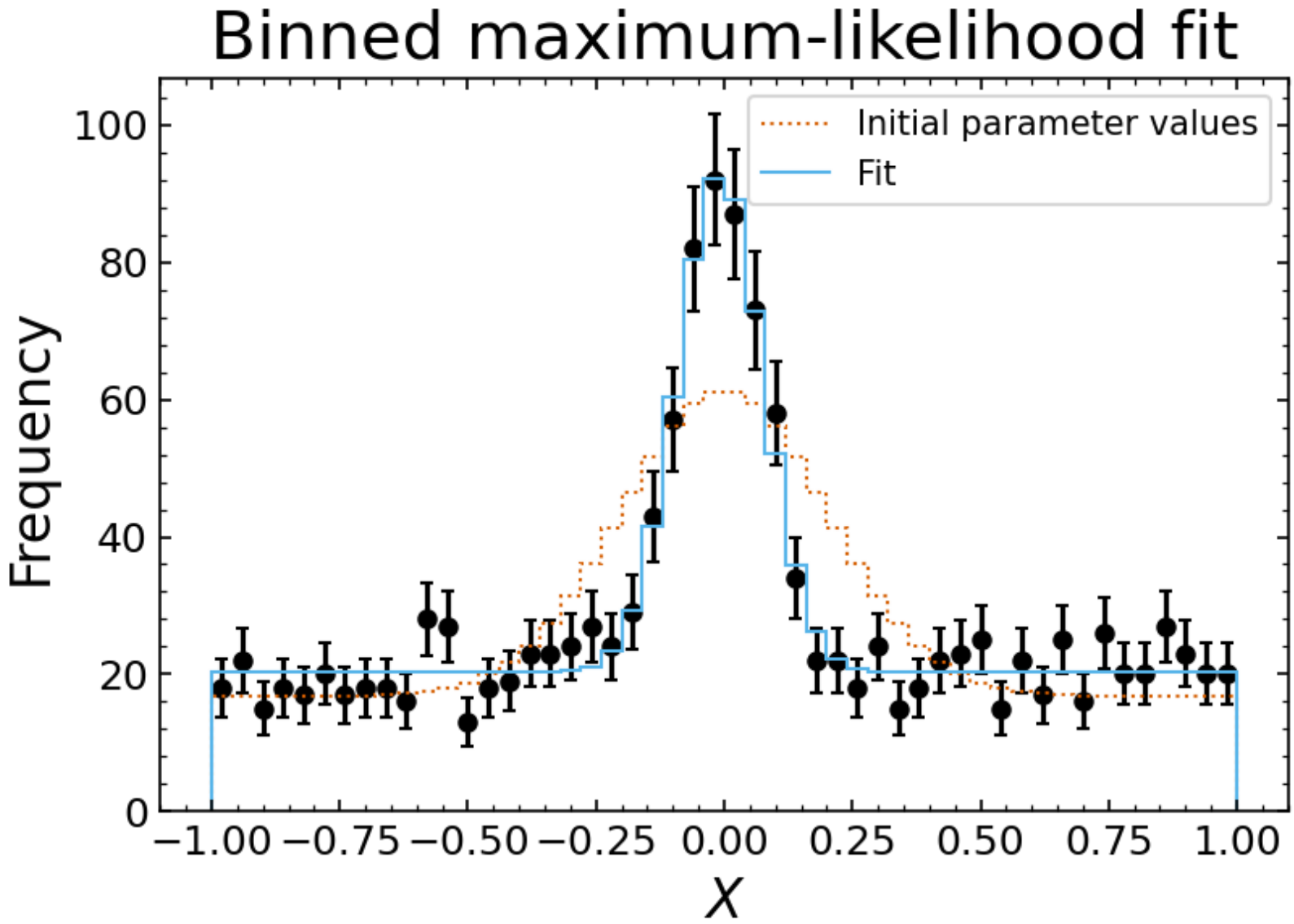
Aside: thinking about this, I think it would be better to put the uncertainty on the fitted values. The data points are exactly that, we have exactly one data point per bin. The uncertainites on the data points are estimated from a theoretical principle.

On the other hand, the theoretical work suggests we have a Binomial distribution in each bin with the analytic formula giving the mean value for the distribution in each bin. We should be plotting the mean value and then the known uncertainties (e.g. 1 sigma band) of this fitted theoretical probability distribution. Given the counts in each bin the Poisson distribution is a good approximation and much simpler to work with.

Also there is more uncertainty in the fitted form than we show. We only show the form using the parameters that give the best fit here.

However, what I show here is a fair enough representation.

In [61]:
```
fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.errorbar(x_bin_centre, counts, counts**0.5, fmt='ok',ms = 5,capsize = 2,zorder = 0)
ax.stairs(np.diff(model_cdf(bin_edges, *[p.value for p in bnll_mobj.init_params]))*len(x_data),
          bin_edges,
          ls=':',label='Initial parameter values',zorder = 1,color='#D55E00')
ax.stairs(np.diff(model_cdf(bin_edges, *bnll_mobj.values))*len(x_data), bin_edges,
          label='Fit',zorder = 1,color='#56B4E9')
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('Frequency',fontsize = 16)
ax.set_title('Binned maximum-likelihood fit',fontsize = 20)
ax.xaxis.set_minor_locator(MultipleLocator(0.05))
ax.yaxis.set_minor_locator(MultipleLocator(4))
ax.legend(loc='upper right')
ax.tick_params(labelsize = 12, direction='in',top=True,right=True,which='both')
```



Summary

In this section, you have seen how to do fitting using `iminuit`, including:

- The basics of fitting
- Details of the Minuit object
- How to set limits on parameters and fix values during fitting
- How to calculate and plot confidence intervals on fitted parameters
- Binned and unbinned maximum-likelihood fits

While we have covered many things here, this is by no means a comprehensive summary of `iminuit`. If you want to learn more, look at the documentation. The tutorials are in general very useful for understanding different approaches/techniques for fitting.

The following section covers the exercises for you to work through this week.

Exercises ^

For the exercises this week, you will work through a problem that is very close to some real data.

Note that some of these answers may contain more information and text than we would expect from students, say in an assessment. We sometimes use our answers to highlight issues and to illustrate ideas, sometimes outside the range of the course, and to do that we may add a wider range of numerical outputs and text.

Data for remaining exercises

The data you will use for the remaining exercises is a list of the energies in GeV (our X variable) of particles seen in an experiment. The theory is that in general we will get a "background" of particle production from all sorts of boring physics we are not interested in. What we are looking for are rare events, the "signal", where a new particle has been produced. These rare events will be seen as extra events on top of the background but these extra events are all at energies centred around the mass of the new particle and with a small spead (the width is related to the decay rate of the particle, it won't be stable).

To find the new particle we will have a theoretical pdf representing some prediction for the shape of the background plus signal. This pdf has six parameters which we must fit to the data, e.g. the centre of the signal peak is a parameter representing the mass of the new particle.

We do not try to fit the raw data directly to the pdf (EFS: we could use a method from week 3 to do this, which one?). Instead we will first bin the data and then we will try to find values for the six parameters of our theory that give the best fit to the binned data.

The first job is to look at the data and then to decide how to bin it and what part is useable in order to give us a reasonable chance of success. That is what we start with in exercise 3. Each exercise builds up step by step until by exercise 7 we hopefully find out the mass of the new particle if the data suggests one is present.

Exercise 3

The data you will use for the remaining exercises is saved in a file named "signal". The code cell below will load this data in using `pickle`.

Investigate this data. This is what you should always try to do before diving into analysis. You want to get a quick feel for the data. However, in thei exercise you must decide on two aspects of the analysis of this data. You must fix aspects (the parameter used to encode this are examples of hyper parameters in the ML langauge we use later):

- The number of bins to use for the data, equivalent to fixing the bin width. If we have few but very wide bins, we might not see the peak representing the new particle. If there are lots of small width bins, many bins will have small counts or even zero, so they will be too noisy to use.
- The range of data to include. We don't want to include regions of low counts as this will be noisy and will not help us find the signal for the new particle (if any).

To do this plot a histogram of the raw data sample from this file to fix:

- an `n_bins` parameter, the number of bins used,
- choose the range of X (energy/GeV) values to include. Use as much of the data as you can.

You must use a Markdown cell to explain your reasons for your choice of these two parameters.

Finally produce the data in the form you will use in the remaining exercises. That is, the energy values and counts (frequency) in each bin.

You may use any data structure and any names you like. I used two one-dimensional numpy arrays `data_x` and `data_y` of the same length with, respectively, the energy values and counts (frequency) in each bin. It might help you to do the same when comparing against my answers.

In [62]:
```
# Read in data
import pickle as pk

with open('signal','rb') as infile:
    signal_data = pk.load(infile)

# Start with some basic measurements of the data
print(signal_data.shape)
print(" Number of data points is "+str(len(signal_data)))
print(" X (energy/GeV): min={:.2f}, max={:.2f}".format(min(signal_data),max(signal_data)))
```

```
(30550,)
 Number of data points is 30550
 X (energy/GeV): min=90.00, max=226.12
```

In [63]:
```
# Your plotting code here
```

Explain your choice of the number of bins used amd the range of X (energy/GeV) values to included for the fit.

Exercise 4

Now that you have prepared the data you think we should analyse, we will start by trying to fit a polynomial function using the `LeastSquares` cost function. We will use `np.polyval` to define our model. Try using different polynomial orders up to 10. Remember you can extract the value of the cost function from your Minuit object, using `m.fmin`, which will also tell you your reduced chi-squared value. What is the best reduced chi-squared you can get?

Hint: to use the chi square statistic, you need to provide errors on the counst in each bin. WHat will you use for these errors and why?

In [64]:
```
# Cost function doesn't like any error = 0, think about how to avoid this.
```

In [65]:
```
# FROM NOW ON WE CAN REUSE THE CODE FROM EXERCISE 1 from this week.

# Our old friend the n-order polynomial
```

```
def line_np(x, par):
    return np.polyval(par, x)
```

In [66]:
```
# Define cost function a.k.a. loss function
```

In [67]:
```
# Define Minuit object
```

In [68]:
```
# Run the minimisation, as in exercise 1
```

In [69]:
```
# Plot result, as in exercise 1.
```

Comments on results of exercise 4

Exercise 5

Now try fitting the data with an exponential, $a\exp((x-b)/c)$, again with the `LeastSquares` cost function. The function has been defined for you below. Discuss with the person next to you what you think good starting parameter values might be.

Is this a good fit? Look at the visual output and calculate a goodness of fit measure.

In [70]:
```
def exp_distrib(x, a, b, c):
    return a*np.exp(-(x-b)/c)
```

In [71]:
```
# Define cost function a.k.a. loss function

signal_lsq_exp_obj = None
```

In [72]:
```
# Define Minuit object
```

In [73]:
```
# Fit Minuit object
```

In [74]:
```
# check fit valid but it looks good
```

In [75]:
```
# Plot result, as in exercise 1.
```

In [76]:
```
# Now look at measures of goodness of fit, p-value
```

Comments on results of Exercise 5.

Exercise 6

Now, we will try a more sophisticated model, composed of an exponential and a Gaussian, namely

$$f(X) = A\exp(-(X-B)/C)) + D\exp(-(x-\mu)^2/2\sigma^2)\,.$$

First, try fitting all of the data simultaneously. The function has been defined in the code cell below. Make sure to plot your results.

Take care with your choice of initial parameters, as this will significantly affect your results.

In [77]:
```
# Exponential and gaussian
def exp_plus_gauss(x, A, B, C, D, mu, sigma):
    return A*np.exp(-(x-B)/C) + D*np.exp(-(x - mu)**2/(2*sigma**2))
```

In [78]:
```
# Define cost function a.k.a. loss function

signal_lsq_eg_obj = None
```

In [79]:
```
# Define Minuit object
```

In [80]:
```
# Fit using Minuit object
# minimise the least_squares cost function, i.e. find the best fit
```

In [81]:
```
# calculate parameter uncertainties
```

In [82]:
```
# Fit looks good but is it?
```

In [83]:
```
# Plot result, as in exercise 1.
```

Comments of results of exercise 6.

Exercise 7

Now we will use the same function

$$f(X) = A\exp(-(X-B)/C)) + D\exp(-(x-\mu)^2/\sigma^2)\,.$$

but we will do the fit in stages. We will start by fitting the exponential background without the part of the data coming from the signal (the region $120 \leq X \leq 130$). So follow these steps:

- Fix the signal parameters `D`, `mu`, and `sigma`
- Set physical limits on background parameters `A`, `B`, and `C` (`A` and `C` should be strictly positive)
- Mask the signal region $120 \leq X \leq 130$, i.e. don't include it in the data.
- Fit the model so that we are only effectively fitting the background.
- Fix background parameters and release signal parameters.
- Set physical limits on signal parameters (`D` and `sigma` should be positive, `mu` should lie in the range of the data)
- Remove the mask, i.e. now run on all the data.
- Fit the model.
- Finally, release all parameters and fit one final time.

Take care with your choice of initial parameters, as this can make a large difference to the result of your fit. Make sure to plot your final result.

In [84]:
```
# Define a new cost function, a.k.a. loss function, to start afresh but it is as before

signal_lsq_eg_obj2 = None
```

In [85]:
```
# Define Minuit object
```

In [86]:
```
# Fix signal parameters and set limits on background parameters
```

In [87]:
```
# Mask the signal region as specified in question
```

In [88]:
```
# Fit the model
```

In [89]:
```
# check fit valid. It looks good

# goodness of fit and p-values
```

In [90]:
```
# Release the signal parameters, fix background parameters, set limits on signal parameters,
# remove the mask

# Release the signal parameters,

# Fix background parameters

# set limits on signal parameters

# remove the mask of the signal region

# Just fit to signal region
```

In [91]:
```
# Fit the model
```

In [92]:
```
# check fit valid but it looks good away from signal

# goodness of fit, p-values
```

In [93]:
```
# Release all parameters, refit the whole model & calculate error using hesse

# Fix background parameters

# remove the mask, fit all data

# Fit the model
```

In [94]:
```
# check fit valid but it looks good

# goodness of fit, p-values
```

In [95]:
```
# Calculate errors in parameters using hesse
```

In [96]:
```
# Plot result.
```

In [97]:
```
# OPTIONAL. I just thought I'd check the pull values
```

Does the masking approach improve the quality of your fit? What is the value of μ for your Gaussian signal? Answer in the Markdown cell below.

In [98]:
```
# Goodness of fit, p-values etc
```

Your comments on results of exercise 7

Exercise 8

Exercise 8a

Now repeat fitting the background and signal simultaneously, but this time using a binned negative log likelihood fit. We will use a linear combination of an exponential pdf $f_{\exp}(x; B, C)$ and a Gaussian pdf $f_{\mathrm{g}}(x; \mu, \sigma)$ as we did with least squares. That is our pdf will be $f_{\mathrm{BNLL}}(x; \mu, \sigma, B, C, z)$ where

$$\begin{aligned} f_{\mathrm{BNLL}}(x; \mu, \sigma, B, C, z) &= z.\, f_{\exp}(x; B, C) + (1 - z).\, f_{\mathrm{g}}(x; \mu, \sigma) \\ &= z.\, C \exp((x - B)/C) + z.\, \frac{1}{\sqrt{2\pi}\sigma} \exp((x - \mu)^2/\sigma^2)\,. \end{aligned}$$

The first code cell below defines a CDF you can use for this purpose using built in functions.

There are various issues mathematically with $f_{\mathrm{BNLL}}(x; \mu, \sigma, B, C, z)$ and with the normalisation which is why we have replaced A and D parameters used in the least square exercises by a single variable z here. Some optional comments on this follow but this form will work.

To solve the problem using binned negative log likelihood fit, use the same data as you did above. Remember to set physical limits on your parameters, including:

- `mu` within the range of the data.
- `sigma` and `C` as strictly > 0.
- $0 \leq$ `z` ≤ 1

Remember to use the `BinnedNLL` cost function.

Exercise 8b

Once you have the best fit using log likelihood, comment on how good the solution is.

Also, compare the log likelihood solution to the least squares solution. For instance, you can use the same `hesse` method and the same built in functions to get the errors on bets fit values for parameters and the χ square values. So you sould be able to copy code used in the least squares case.

Optional notes on Exercise 8: Normalisation and Log Likelihood.

Normalisation is an issue here. The exponential distribution is normalised for $0 \leq x \leq \infty$ while Gaussians are defined for any real x. In any case, we are only really interested in the limited range used in our data and then the standard functions we use from the libraries are definitely not normalised for these limited ranges.

If we have normalised pdf $f_1(x)$ and $f_2(x)$ defined for the same range of x values, then $z.\, f_1(x) + (1-z).\, f_2(x)$ is also a valid pdf in the same range of x if $0 \leq z \leq 1$. However here each individual function is not normalised over the range of our data so adding them together with a z and (=$(1-z)$ prefactors where z lies between 0 and 1 makes no sense.

We could normalise the functions, numerically in the Gaussian, but then these factors would also depend on the parameters making definition of the functions much more involved.

In fact for log likelihood, the overall freedom to change the normalisation of a function is an irrelevant factor and you should never include this. Suppose $f(x;\theta)$ is a normalised pdf but we choose to work with the log likelihood of $g(x;\lambda,\theta) = \lambda f(x;\theta)$ where λ is an extra parameters. We can see that the negative log likelihood working with f and with g only differ by a factor of $N\ln(\lambda)$ for N data points. This is minimised by taking λ to zero which is clearly silly. Even if we limit λ to be greater than some positive number c we will always chose $\lambda = c$ as this is the smallest value for this extra factor. So we should never allow the overall normalisation of our pdf to change in log likelihood analysis though it need not be equal to 1. As long as λ is fixed, minimising with f or g will give the same result for parameters as the log likelihoods differ by a constant $N\ln(\lambda)$. So the overall normalisation is irrelevant as long as it is fixed. So we don't care if the nromaalisation is messed up as long as we factor this out and work with z and $(1-z)$ prefactors with $0 \leq z \leq 1$ rather than the independent A and D normalisations used in least squares.

In [99]:
```
# Define the cdf we need
```

In [100…
```
# First get the data we will use.
# Perhaps simply reuse the parameters and data used for least squares.
```

In [101…
```
# Define your cost function
from iminuit.cost import BinnedNLL

signal_bnll_eg_cost = None
```

In [102…
```
# Define your Minuit object & set limits on parameters
```

In [103…
```
# Fit your function
```

In [104…
```
# Plot your results
```

In [105…
```
# Calculate errors using hesse
```

In [106…
```
# Goodness of fit, p-values

```

Comparison of Least Squares and BNLL solutions

FINISHED! <!-- NO TOO BORING, DONE ENOUGH.

Exercise 9

Finally, repeat the unbinned maximum-likelihood fit with the same masking procedure we used for the `LeastSquares` cost function, but this time using the CDF we have defined and the `BinnedNLL` cost function. Remember the key steps:

- Fix the signal parameters `mu`, and `sigma`
- Set physical limits on background parameters `C`, and `D`
- Mask the relevant region
- Fit the model
- Fix background parameters and release signal parameters

- Set physical limits on signal parameters
- Remove the mask
- Fit the model
- Finally, release all parameters and fit one final time

How does the result from this compare with the unmasked approach? How does it compare with the `LeastSquares` results? What about the uncertainty on the fitted parameters? Write your answers in the Markdown cell after the code cells.

-->

Appendices ^

Appendix: Bug or feature when using functions with multiple parameters in `migrad`

The `line_np` routine used above to fit polynomials of any given degree works as long as you don't visualise the result in iminuit which the `migrad` method does in Jupyter notebooks. Given our first example comes from the iminuit DOCS it seems like a bug. The solution used above was to assign a variable to the Minuit object when using migrad.

Alternatively this version 2 below does work with the visualisation in iminuit. Here we use `*par` in our definition to pack all parameters passed (when we use the Minuit object) into a single tuple, to pass to `np.polyval` . This prevents a bug with Minuit's built-in visualisation. However, note you now call this routine with an explicit list of parameter values `line_np_version2(x, p0, p1, p2, p3)` as opposed to `line_np(x, [p0,p1,p2] )` as we did originally.

In [107…

```
def line_np_version2(x, *par):
    return np.polyval(par, x)  # for len(par) == 2, this is a line
```

Appendix: Approximating the predicted count in a bin

Here won't write the dependence on parameters (θ) explicity. We then note that for the CDF $F(X)$ we have that

$$F(X + \Delta X) \approx F(X) + \Delta X \, \frac{\partial F}{\partial X} = F(X; \theta) + \Delta X \, f(X) \, .$$

Note we use an identity that the derivative of the CDF $F(X)$ is the PDF $f(X)$ which you prove from the definition of the CDF in terms of the intergal of the PDF (EFS: prove it).

Let p_b be the predicted probability that one event will be in bin b. Then this is given by

$$p_b = F(X_b) - F(X_{b-1})$$

Suppose we let $\bar{X}_b = (X_b + X_{b-1})/2$, i.e. the location of the centre X value of the bin. Then

$$\begin{aligned} & p_b \approx \big(F(\bar{X}_b) + (X_b - \bar{X}_b) \, f(\bar{X}_b)\big) \quad - \quad \big(F(\bar{X}_b) + (X_{b-1} - \bar{X}_b) \, f(\bar{X}_b)\big) \\ \Rightarrow \quad & p_b \approx (X_b - X_{b-1}) \, f(\bar{X}_b) \, . \end{aligned}$$

Appendix: Extended unbinned and binned maximum-likelihood fits

Sometimes, rather than just finding the shape parameters of some underlying probability distributions, we want to also find the integral of the probability distribution. This is particularly common in particle physics where we are trying to estimate physical observables, like the cross section of an interaction.

For the unbinned case, the model has to return not only the probability density (as before) but also the integral of the density, which corresponds to the total number of counts. For the model we have used so far, this is equivalent to replacing the z term before the normal distriubtion with n_{sig}, the number of signal events, and the $(1 - z)$ term before the uniform distribution with n_{bkg}, the number of background events.

For the binned case, we still only need to return the CDF, but we still must make the parameter update as before. In both cases, we are fitting one extra parameter.

We can run these fits using the cost functions `ExtendedUnbinnedNLL` and `ExtendedBinnedNLL` from `iminuit.cost`.

Temporary masking for fitting

When we have a complex data set with one or multiple peak(s) and a background, it can be useful to fit in several stages. It is common to start by masking the signal region, so we can fit only the background region. This means we can hide some region of our data, e.g. around where we expect our signal, so we can just fit the background. After we have successfully fit the background, we can then keep the background parameters fixed and fit the signal parameters of the whole distribution.

This is easy to do in iminuit, using the mask attribute of cost functions.

We can demonstrate this for a binned fit below, where now instead of fitting the fraction of signal and background z we fit for the total number of signal and background events, $n$sig and $n$bkg respectively. This kind of fit can be referred to as an extended binned maximum-likelihood fit, and is useful when we care about the total number of events as well as the shape of the probability distributions. Extended maximum-likelihood fits are non-examinable and will not be discussed further, but are useful as an example for masking.

In [108…

```
# This reproduces the data used in the Maximum Likelihood section.
from scipy.stats import norm, uniform
from matplotlib.colors import to_rgba
from matplotlib.ticker import MultipleLocator

xrange = -1, 1
rng = np.random.default_rng(1)

mu_value = 0.0
sigma_value = 0.1
n_normal = 400
n_uniform = 1000
z_value =  n_normal/(n_normal+n_uniform)
print("Data generated using mu={:.3f}, sigma={:.3f},  z={:.3f} ".format(mu_value,sigma_value
# EFS: we don't stop the Gaussian from ranging outside -2 to 2 range. Does it matter?
x_data = rng.normal( mu_value, sigma_value, size = n_normal)
x_data = np.append(x_data, rng.uniform(*xrange, size = n_uniform))
```

```
n, xe = np.histogram(x_data, bins = 50, range = xrange)
x_bin_centre = 0.5*(xe[1:]+xe[:-1])
dx = np.diff(xe)
```

```
Data generated using mu=0.000, sigma=0.100,  z=0.286
```

In [109…

```
from iminuit.cost import ExtendedBinnedNLL

def model_density_cdf(x, nsig, nbkg, mu, sigma):
    return (nsig*norm.cdf(x, mu, sigma) + nbkg*uniform.cdf(x, xrange[0], xrange[1]-xrange[0]

c = ExtendedBinnedNLL(n, xe, model_density_cdf)

m = Minuit(c, nsig = 300, nbkg = 1500, mu = 0, sigma = 0.2)

m.limits['mu'] = (-1,1)
m.limits['nsig','nbkg','sigma'] = (0, None)
m.fixed['mu','sigma','nsig'] = True # Fix mu, sigma, and the total number of signal events

c.mask = (x_bin_centre < -0.5) | (0.5 < x_bin_centre) # Mask the signal between -0.5 and 0.5

m.migrad()
```

Out[109]:

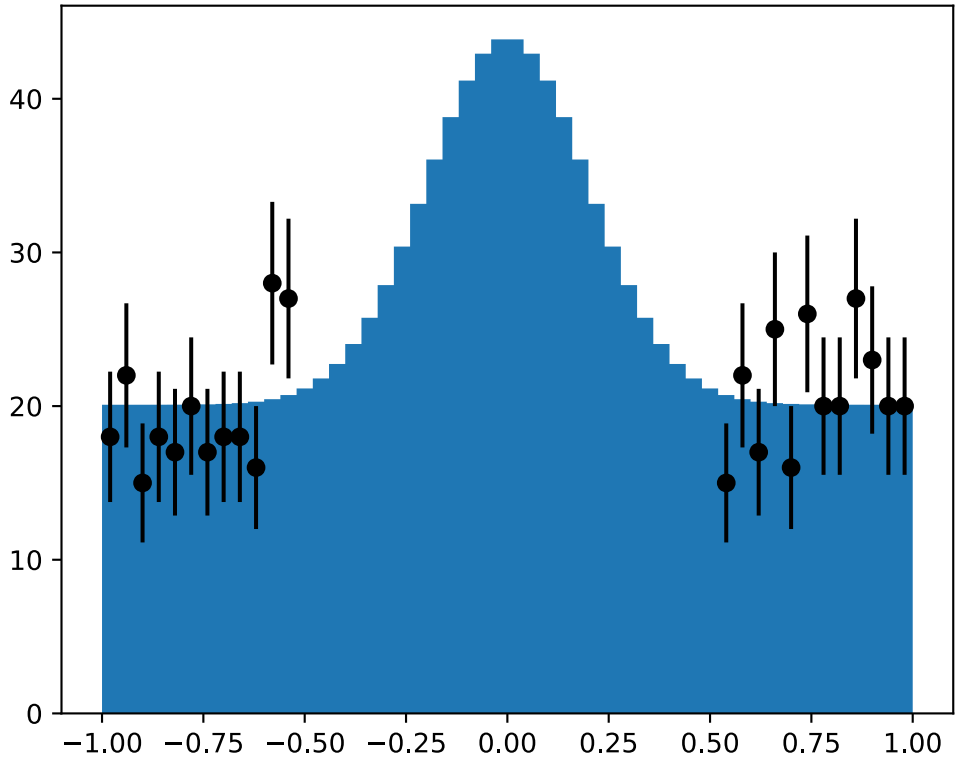
Migrad

FCN = 17.28 (χ^2/ndof = 0.8)	Nfcn = 20
EDM = 2.39e-10 (Goal: 0.0002)	

Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	nsig	300	3			0		yes
1	nbkg	1.00e3	0.05e3			0		
2	mu	0.0	0.1			-1	1	yes
3	sigma	0.200	0.002			0		yes

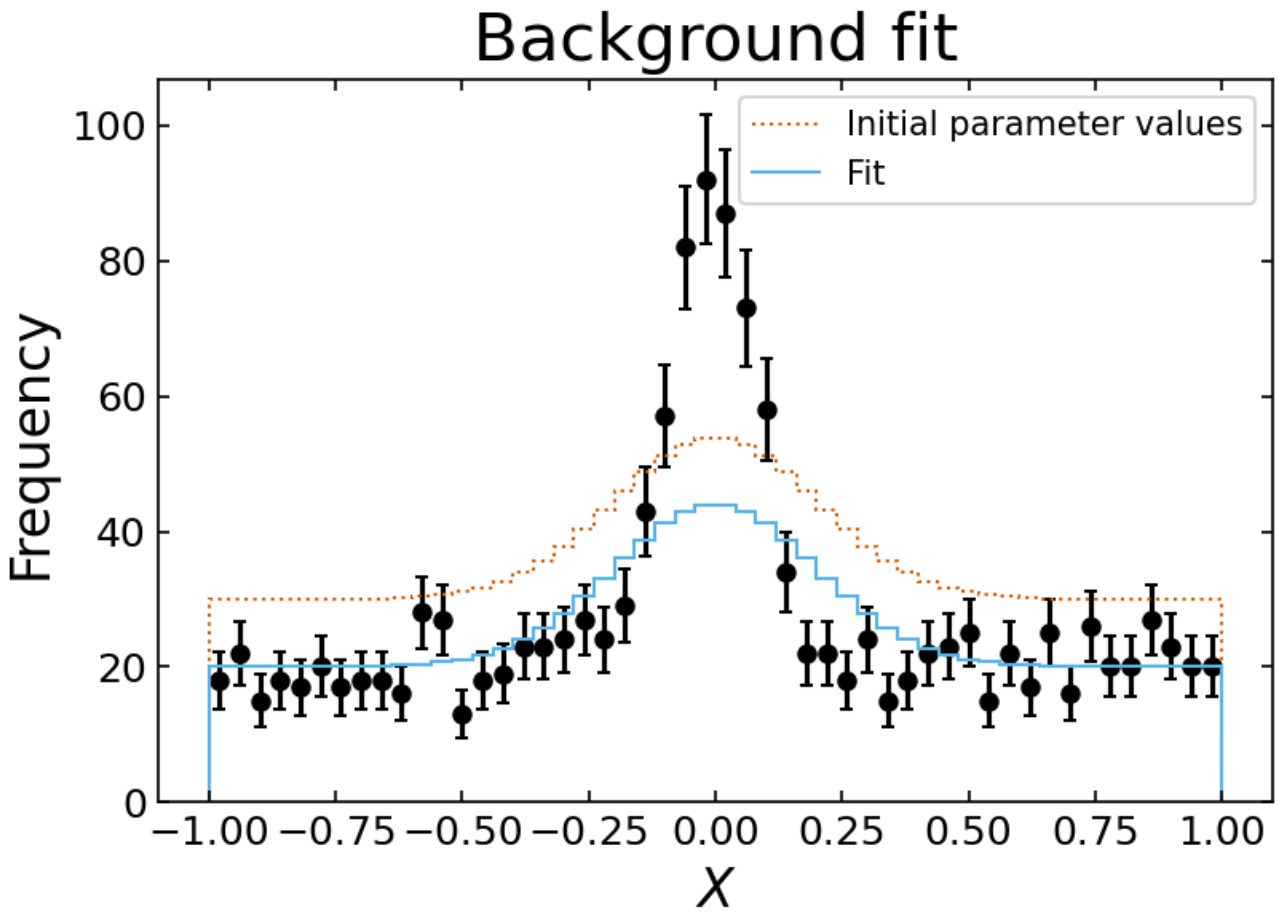
	nsig	nbkg	mu	sigma
nsig	0	0	0	0
nbkg	0	2.1e+03	0	0
mu	0	0	0	0
sigma	0	0	0	0



We can now plot the background result we have fitted:

In [110…

```
fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.errorbar(x_bin_centre, n, n**0.5, fmt='ok',ms = 5,capsize = 2,zorder = 0)
ax.stairs(np.diff(model_density_cdf(xe, *[p.value for p in m.init_params])), xe, ls=':',lab
ax.stairs(np.diff(model_density_cdf(xe, *m.values)), xe, label='Fit',zorder = 1,color='#56B
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('Frequency',fontsize = 16)
ax.set_title('Background fit',fontsize = 20)
# ax.xaxis.set_minor_locator(MultipleLocator(0.05))
# ax.yaxis.set_minor_locator(MultipleLocator(4))
ax.legend(loc='upper right')
ax.tick_params(labelsize = 12, direction='in',top=True,right=True,which='both')
```



We can see the background leve looks about right, but obviously our signal region has not been fitted. We can now fix the background, remove the mask, and fit only the signal parameter.

In [111…

```
c.mask = None # Remove the mask of the signal region
m.fixed = False # Release all parameters
m.fixed['nbkg'] = True # Fix the background amplitude

m.migrad()
```

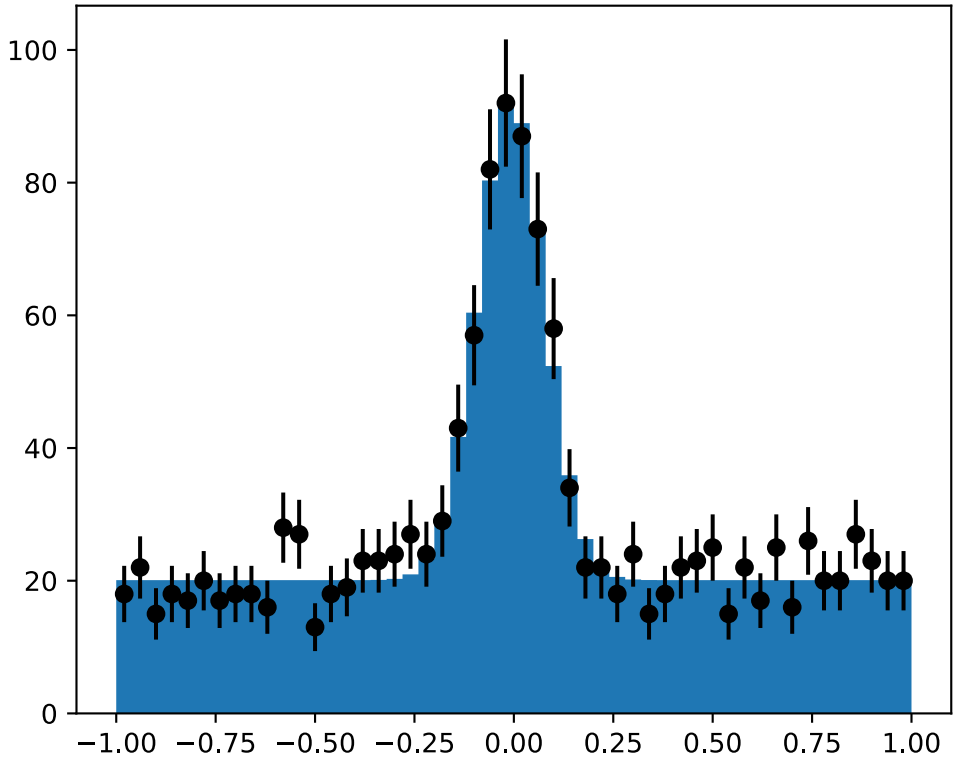
Out[111]:

Migrad

FCN = 29.86 (χ^2/ndof = 0.6)	Nfcn = 91
EDM = 8.78e-05 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	nsig	386	25			0		
1	nbkg	1.00e3	0.05e3			0		yes
2	mu	-0.008	0.006			-1	1	
3	sigma	0.084	0.006			0		

	nsig	nbkg	mu	sigma
nsig	625	0	-5.29e-3 **(-0.034)**	43.938e-3 **(0.306)**
nbkg	0	0	0	0
mu	-5.29e-3 **(-0.034)**	0	3.9e-05	-0.003e-3 **(-0.084)**
sigma	43.938e-3 **(0.306)**	0	-0.003e-3 **(-0.084)**	3.29e-05



Finally, we release all of the parameters and run the fit again to get correct estimates of the uncertainty.

In [112…

```
m.fixed = None
m.migrad()
```

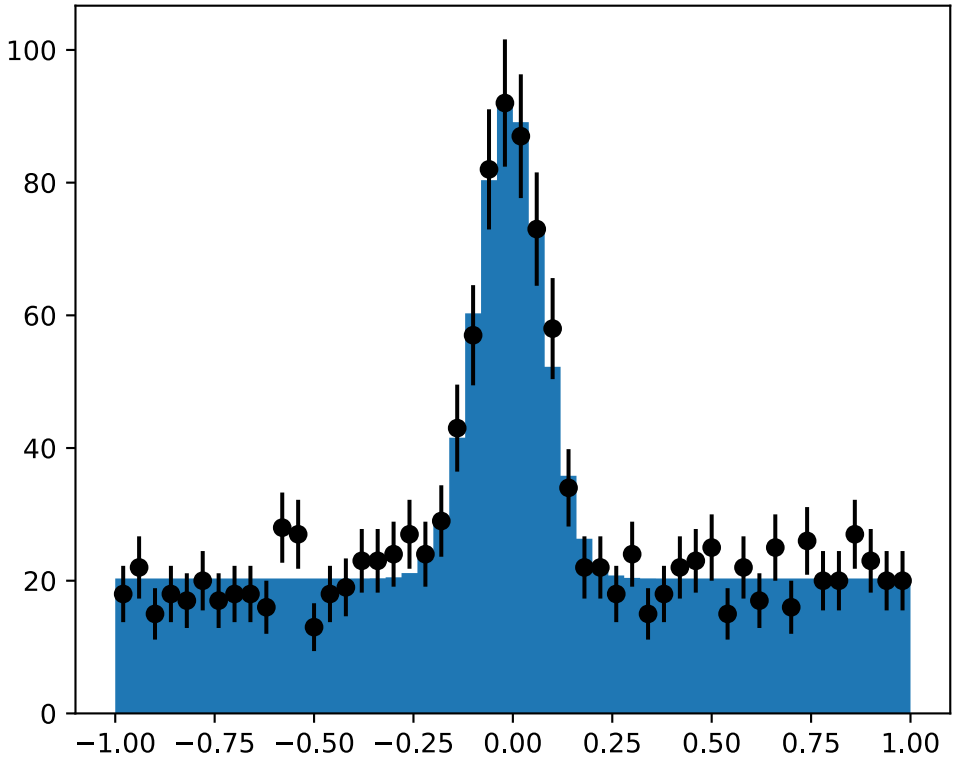
Out[112]:

Migrad

FCN = 29.74 (χ^2/ndof = 0.6)	Nfcn = 158
EDM = 5.84e-06 (Goal: 0.0002)	
Valid Minimum	Below EDM threshold (goal x 10)
No parameters at limit	Below call limit
Hesse ok	Covariance accurate

	Name	Value	Hesse Error	Minos Error-	Minos Error+	Limit-	Limit+	Fixed
0	nsig	383	26			0		
1	nbkg	1.02e3	0.04e3			0		
2	mu	-0.008	0.006			-1	1	
3	sigma	0.083	0.006			0		

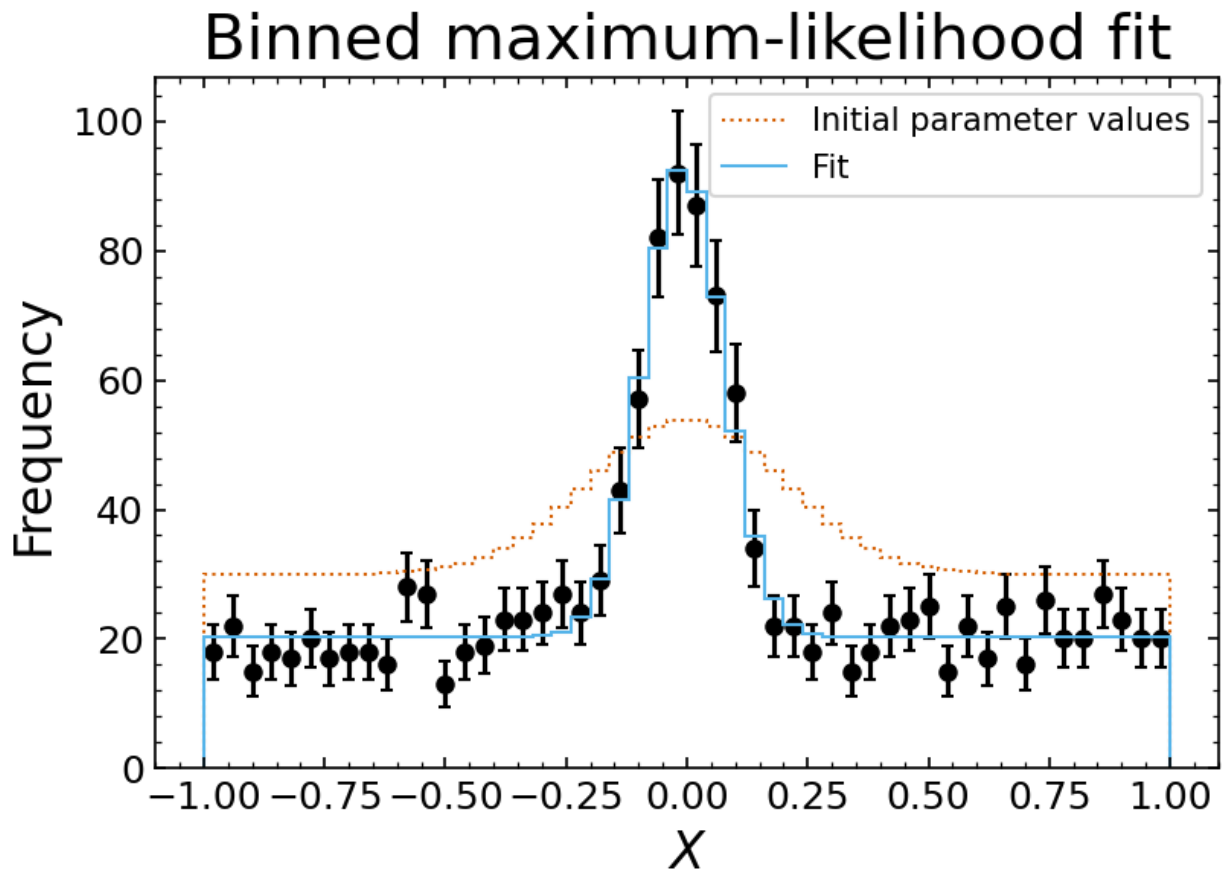
	nsig	nbkg	mu	sigma
nsig	696	-0.3e3 **(-0.325)**	-6.76e-3 **(-0.041)**	56.718e-3 **(0.364)**
nbkg	-0.3e3 **(-0.325)**	1.33e+03	6.76e-3 **(0.030)**	-56.713e-3 **(-0.263)**
mu	-6.76e-3 **(-0.041)**	6.76e-3 **(0.030)**	3.89e-05	-0.003e-3 **(-0.089)**
sigma	56.718e-3 **(0.364)**	-56.713e-3 **(-0.263)**	-0.003e-3 **(-0.089)**	3.49e-05



We can now plot the full result:

In [113…

```
fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.errorbar(x_bin_centre, n, n**0.5, fmt='ok',ms = 5,capsize = 2,zorder = 0)
ax.stairs(np.diff(model_density_cdf(xe, *[p.value for p in m.init_params])), xe, ls=':'
ax.stairs(np.diff(model_density_cdf(xe, *m.values)), xe, label='Fit',zorder = 1,color='
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('Frequency',fontsize = 16)
ax.set_title('Binned maximum-likelihood fit',fontsize = 20)
ax.xaxis.set_minor_locator(MultipleLocator(0.05))
ax.yaxis.set_minor_locator(MultipleLocator(4))
ax.legend(loc='upper right')
ax.tick_params(labelsize = 12, direction='in',top=True,right=True,which='both')
```



As we can see, this is exactly the same as when we did the fit all in one go. This is because we have used a simple example, but in general this can be very helpful to fit lots of histograms without having to adjust each fit manually.