Week 3: Statistics, Hypothesis Testing and Goodness of Fit

Outline ^

Section One: Introduction to Statistics ^

Up to this point we have discussed probability, because in many physics experiments we are measuring quantities that are ultimately random variables. This allows us to predict how often we might measure the various possible outcomes of our experiment. Statistics, on the other hand, involves studying data we measure to try to determine the values and uncertainties of physical quantities.

We can therefore think of these two as inverse problems of one another:

- Studying probability allows us to take physical constants and predict likely outcomes of an experiment
- Studying statistics allows us to take the actual outcome of an experiment and find the values of physical constants that are compatible with data

We can make statistics that summarise data, e.g. find the mean and standard deviation of a dataset we have recorded.

Section Two: Hypothesis Testing ^

When we have measured data in an experiment, then we may wish to see if our data is consistent with a particular probability distribution, usually corresponding to a specific physical process.

The null hypothesis H_0

In general, the distribution may be parameterless or have some parameter we are interested in testing a value of. For example, consider a probability distribution with some parameter θ, that we think has the value θ_0. We want to see if this value θ_0 for the parameter θ is allowed by our data, is consistent withour data. We are not syaing (yet) if this is the best value, we just ask if it is a possible value. We refer to this as testing the **hypothesis** that $\theta = \theta_0$. We refer to this hypothesis as the **null hypothesis** and it is traditionally denoted as H_0 ("H" for hypothesis, "0" for null).

The word "null" is used because if the the null hypothesis is true then there is *no* statistically significant difference between our proposed distribution with parameter θ_0 and our data; any differences seen could be explained by chance alone. The proposed distribution with parameter θ_0 is called the "null model".

It is also often the case that the null model, the null hypotehsis, is some boring uninteresting answer and we are really hoping that this null hypotehsis is not true. So we are looking to "nullify" the null hypothesis.

In the end, the hypothesis testing method does not care what we do with the null hypothesis. So in practice the role and interpretation of a null model can be different in different studies.

Hypothesis Testing for Continuous Variables

What do we mean by "consistent with the data"?

When is our data consistent with a probability distribution? That is, how do we turn that into a quantifiable statement?

Consider some hypothesised PDF $f(X)$ for a continuous variable X, and a single measurement X_m.

- If the hypothesis is correct (i.e. that the random variable X is distributed according to $f(X)$), then we expect the measurement X_m to be in a region where the PDF is large

- If the measured value X_m lies in a region of the PDF with low probability, we would reject the hypothesis that this random variable is distributed according to this PDF

How do we quantify whether the PDF value $f(X_m)$ is small or large? Of course, the absolute value of the PDF is not useful, it is a probability density not a probability. To get a probablity value we need to integrate the PDF $f(X)$ over some region to determine the probability of a measurement flaiing in that region.

So we define a **rejection region** (or **critical region**) which is a range of values of X_m for which we reject the null hypothesis. The probability of a measurement falling in the rejection region is called the **significance level** α. That is α is defined as according to

$$\alpha = \int_{\text{Rejection region}} f(X)\, dX$$

i.e. the significance level is the integral of the PDF over the rejection region.

In practice we often choose a value for the significance level α. Then we define the shape but not the size of the rejection region. Finally we adjust the size of the rejection region until the probability of falling in the regection region is our chosen value of α.

For example, suppose we want a significance level of $\alpha = 0.05$ i.e. 5%. That is for a random variable X distributed according to our hypothesised PDF $f(X)$ there is a 5% chance that the value of this random variable X lies in our rejection region. We could therefore reject the hypothesis even if it is true (a **false negative**), and indeed would expect to do so 5% of the time. This is referred to as a **Type I error**, and is unavoidable as it is a consequence of probability.

It is also common to define something called a **confidence interval**, which is more important when we discuss parameter estimation next week but is commonly used terminology for finding rejection regions using `scipy`. Whereas we have defined the significance level α as the probability a measurement X lies in our rejection region, the confidence interval corresponds to all values of X *not* in the rejection region, which we often refer to with a probability as well. For example, the 95% confidence interval corresponds to the region in which we expect a measurement to be 95% of the time. If we make a measurement X that lies outside of this region, we could say we reject the hypothesis at a 5% significance level as that measurement is by definition in the rejection region with a 5% significance level.

Example of rejection region and significance level

Due to the Central Limit Theorem, a normal distribution is most common hypothesised PDF $f(X)$.

- So suppose our $f(X)$ is a standard normal distribution (mean $\mu = 0$, standard deviation $\sigma = 1$).

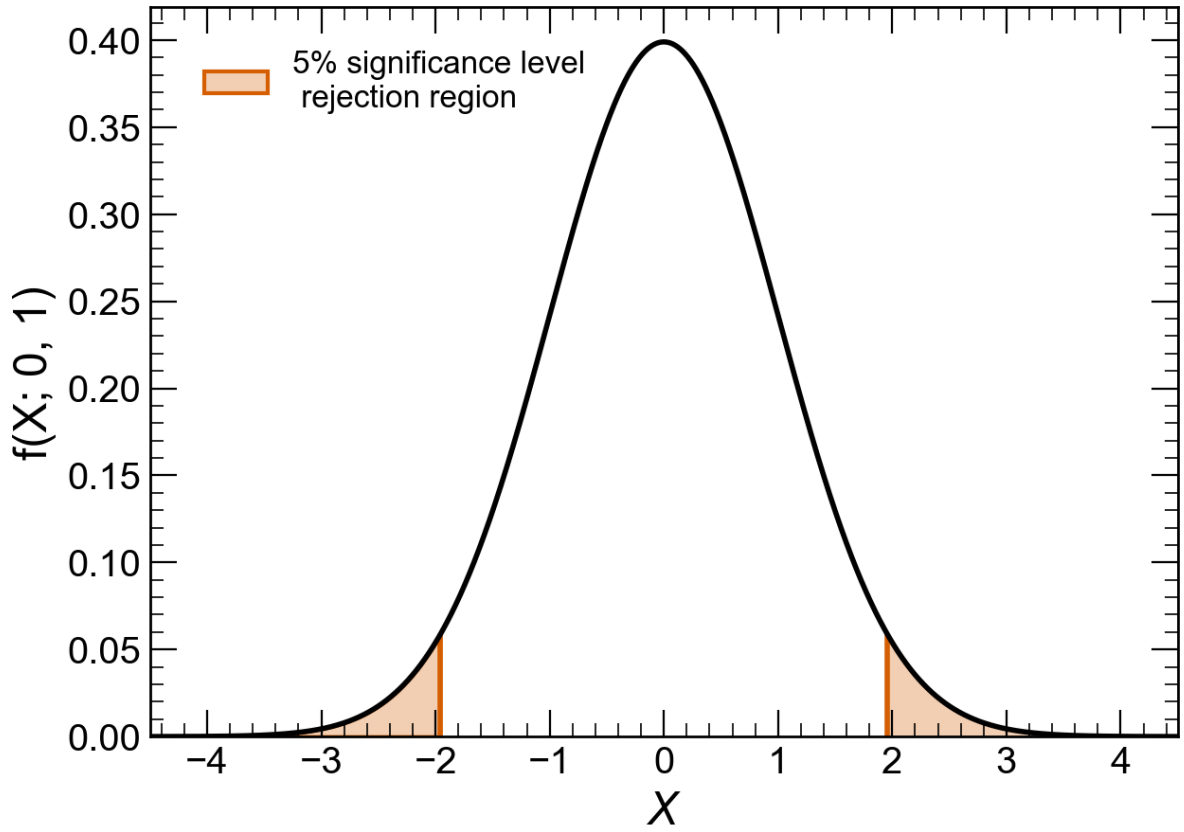
- We will choose our significance level α to be 5%.
- Next we decide that we will reject results in the tails of the distibution as the PDF is exponentially small for large X values. A simple way to define the shape of this region is to say it is where $|X| > X_r$, i.e. our rejeciton region has two parts, $X < -X_r$ and $X > +X_r$.
- Finally we have to adjust the size of the rejection region to get our 5% rejection rate. So here we choose the X_r that sets the boundary of the rejection region so that

$$\int_{-\infty}^{-X_r} \frac{1}{\sqrt{2\pi}} \exp(-x^2/2)\, dx \quad + \int_{+X_r}^{+\infty} \frac{1}{\sqrt{2\pi}} \exp(-x^2/2)\, dx \quad = \alpha\,.$$

Aside:- There is no simple way to represent these integrals but they appear so often that they have a standard name, the complementary error function erfc, and you can find standard routines to give their values. In fact to find X_r from a given α you need the inverse error function erf^{-1} as $\alpha = \mathrm{erfc}(X_r) = 1 - \mathrm{erf}(X_r)$ so $X_r = \mathrm{erf}^{-1}(1-\alpha)$. Luckily you can usually find a library function for the inverse error function erf^{-1} (try `scipy.special.erfinv`).

Below we show the rejection region for this example.

Note chosing α equal to 0.05 means we have 95% of the normal curve around the mean in the accepted region. That is about two standard deviations if you remember the properties of a normal distribution. And sure enough, we can see the rejected region starts at about $X = 1.96$ and $X = -1.96$ (i.e. at roughly 2σ as $\sigma = 1$ for this standard normal distribution).



Standard normal distribution with the rejection region corresponding to a 5% confidence interval represented by the shaded region. Any measurement X distributed according to $f(X)$ has a 5% probability of lying in the rejection region by chance. Note this showing a typical **two-tail rejection region** *as both very large positive and very large negative values are rejected. A* **one-tail rejection region** *would only work with one of these two, say just very large positive values are considered failures.*

You can reduce the frequency of Type I errors by reducing the size of the rejection region (and the significance level α), but this also makes it less likely we will reject the null hypothesis even if it is false. There is a balance between these two effects that must be considered.

It is important to not that we cannot prove a hypothesis is true! Even if the measured value X_m is in the highest probability region of the PDF, there is no guarantee that means that exact PDF is correct. As a result, all we can do is reject a hypothesis we deem to be false. If the measurement X_m does not reject a hypothesis even though it is false (a **false positive**), we make a **Type II error**. Without knowing the true PDF, we cannot evaluate how likely a Type II error is.

In [1]:
```
from scipy.stats import norm

# Standard normal distribution, confidence intervals
interval25 = norm.interval(0.25, loc = 0, scale = 1)
print("25% confidence interval +/- {0:4.2f} ".format(interval25[1]) )
interval50 = norm.interval(0.50, loc = 0, scale = 1)
print("50% confidence interval +/- {0:4.2f} ".format(interval50[1]) )
interval75 = norm.interval(0.75, loc = 0, scale = 1)
print("75% confidence interval +/- {0:4.2f} ".format(interval75[1]) )
```

```
25% confidence interval +/- 0.32
50% confidence interval +/- 0.67
75% confidence interval +/- 1.15
```

The alternate hypothesis H_1

Originally, only the null hypothesis was used in significance testing, an approach pioneered by one of the most important statisticians Ronald Fisher in the 1920's. Essentially, what we have described so far is Fisher's approach. In the 1950's Jerzy Neyman and Egon Pearson (confusingly, the son of the one who invented the Pearson correlation coefficient) attempted to improve Fisher's approach and this led to the use of an explicit alternative hypothesis H_1.

In many situations it is important to have a well defined alternative to the null hypothesis as the "not H_0" theory/model is often not defined in a useful way. So in this more modern approach, we have a well defined **alternate hypothesis** H_1 (H_a used elsewhere sometimes) for comparison with the null hypothesis H_0.

One way to see how an alternative hypothesis makes a difference is to note that our rejection regions for the null hypothesis H_0 often change significantly depending on our choice for the alternate hypothesis H_1. To see this consider the following process.

- Consider a random variable X
- Null hypothesis H_0: $X \sim f(X)$ for some probability distribution $f(X)$
- Alternate hypothesis H_1: $X \sim g(X)$ for some different probability distribution $g(X)$

- Make a measurement X_m of the random variable X that lies in a low PDF region for the null hypothesis
- Even if $f(X_m)$ is small, if $g(X_m)$ is *smaller* we say the null hypothesis is still still more believable than the alternate hypothesis, so we cannot reject the null hypothesis

As a result, we can see that any regions for which $g(X)$ is *smaller* than $f(X)$ should not be included in our rejection region regardless of how probable the values of X are under the null hypothesis. In such regions, we have no reason to prefer the alternate hypothesis over the null hypothesis and so cannot reject the null hypothesis.

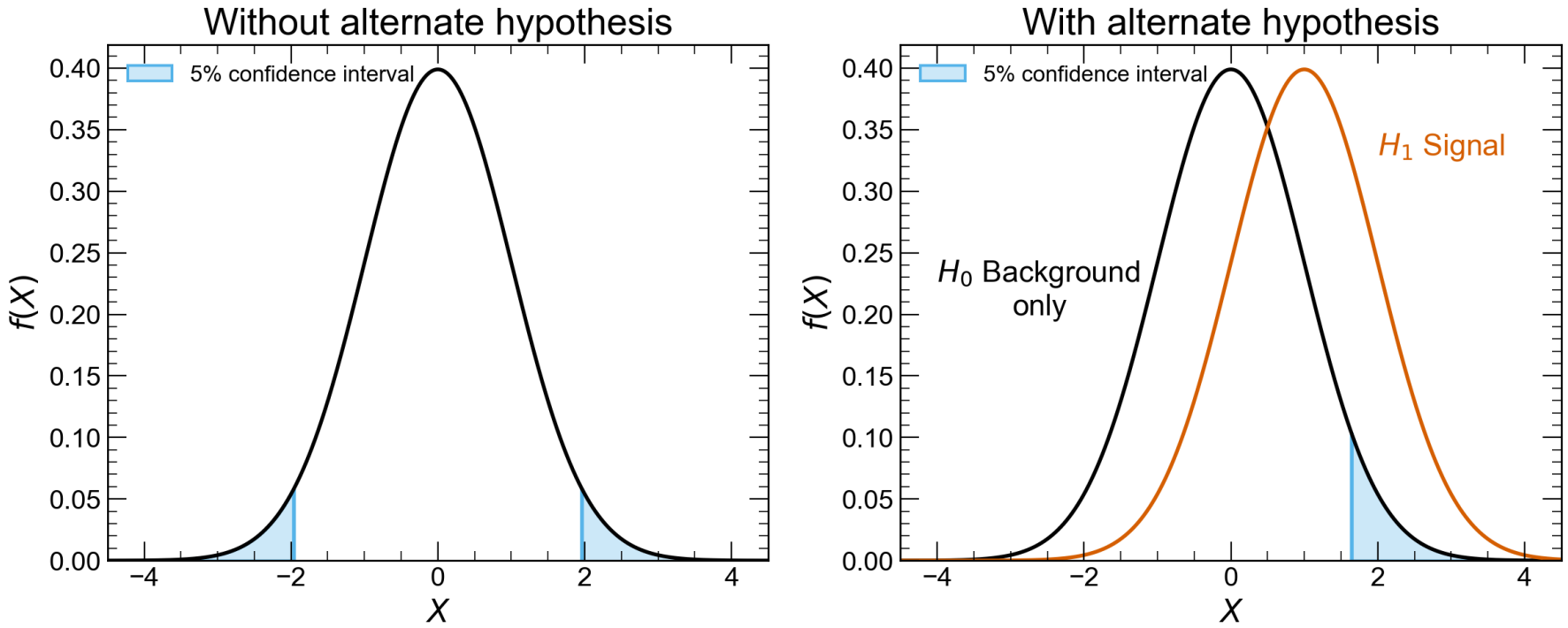
Note: much like before, we *cannot* accept the alternate hypothesis under a hypothesis test, only decide whether we do or do not reject the null hypothesis.

Example

For example, consider some potential new physics theory that predicts a some new signal in the measurement of X.

- In the absence of this new theory, we expect X to be distributed according to a standard Gaussian distribution centered at 0 and standard deviation $\sigma = 1.0$. This will be our null hypothesis H_0 that the PDF for X is the standard normal distribution $N(0, 1)$.
  - In particle physics experiments this would describe what is called the "background".
  - In this case we could imagine this this peak represents the production of a known particle, say the Higgs (so $X = (E - m_{\text{Higgs}})/\sigma_{\text{Higgs}}$ where we used the mass and lifetime/width of the Higgs to rescale eneregy E into dimensionaless X.
- On the other hand this new theory predicts X distributed according to a Gaussian centered at 1 and standard deviation $\sigma = 1.0$. This will be our alternative hypothesis $H(1)$ that the PDF for X is $N(1, 1)$.
  - In particle physics experiments this would describe what is called the "background" but it would normally be a smooth distribution without a peak in a realistic case.
  - have shifted energy axes to use the mass and deacy width of the set the energy to be zero and we are looking to see if virtual particles (heavier particle predicted by supersymmetry perhaps)
- For large positive values of X, we can safely reject the null hypothesis. The PDF for the alternative hypothesis H_1 is always greater than the null hypothesis for $X > 0.5$ (EFS prove this) so H_1 is always a better if sometimes a poor choice there.
- However, if we measure a large negative value of X, although this is unlikely under the null hypothesis H_0, it is even less likely under the alternate hypothesis H_1 and so we cannot reject the null hypothesis.
- As a result, we do *not* have a rejection region for large negative values of X.
- Instead, the size of the rejection region at large positive values is *increased* to cover the total desired significance level α as compared to the previous example where we had only the null hypothesis.

Suppose we choose the same significance level of $\alpha = 5\%$ as before. The left plot shows the null hypothesis rejection region without considering an alternate hypothesis, exactrly as described in the previous example. However, the right shows the new rejection region after we have considered the alternate hypothesis H_1 shown here. Clearly, the alternate hypothesis significantly alters our rejection region. In particular, note that to get a 5% rejection but only using the positive X region, this positive rejection region is not simply $X > 1.96$ as it was before and shown on the left. Now we find the positive rejection region needs to be bigger, $X > 1.64$, as this is the only part of the rejection region.



Composite alternate hypotheses (Non-examinable)

While sometimes we may have a well defined PDF for our alternate hypothesis, this is not always the case. There are two main cases we can consider:

- In our signal-background example, we may not know the parameters of the signal distribution H_1 but we know that it should predict $X > 0$
  - As a result, know H_0 PDF is higher than H_1 PDF for $X \leq 0$
  - Use one-sided test

- Alternate hypothesis may be equally likely to shift value up or down
  - Have minimal information about alternate hypothesis, but as each direction is equally likely use a two-sided rejection region (as shown at first)
  - Effectively assuming a uniform distribution as the alternate hypothesis, to define the rejection region

Hypothesis testing for discrete distributions

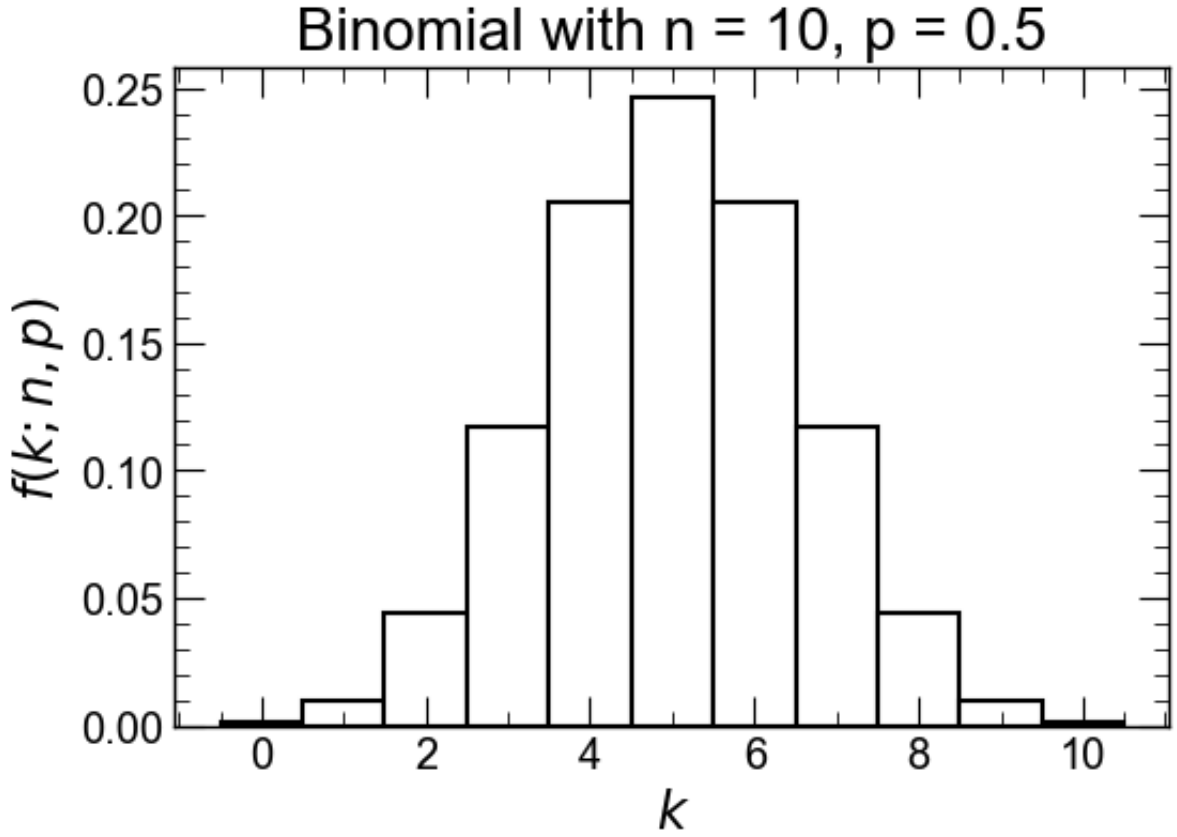
So far we have talked about finding intervals in a PDF for a continuous variable X with specific significance levels. In general, the same principles map well to to discrete distributions but some changes are needed. Since probabilities are no longer continuous, it is therefore not usually possible to find a rejection region with exactly our chosen significance level.

Example

For example, consider flipping a coin 10 times. We would like to know if the coinis unbiassed. We expect these measurements to obey a binomial distribution as there are two possible outcomes each time, heads or tails. If the coin is unbiased, we would expect the probability of a heads to be equal to the probability of a tails, $p = 0.5$. So

- Null hypothesis H_0: Binomial distribution with $p = 0.5$ and $n = 10$ (sometime written as $B(10, 0, 5)$).
- Alternate hypothesis H_1: coin is biased, i.e. $p < 0.5$ or $p > 0.5$.
- Hypothesis test is therefore *two-sided rejection region*.
- Choose significance level of $\alpha = 5\%$.

The figure below shows the distribution of the number of successes for the null hypothesis.



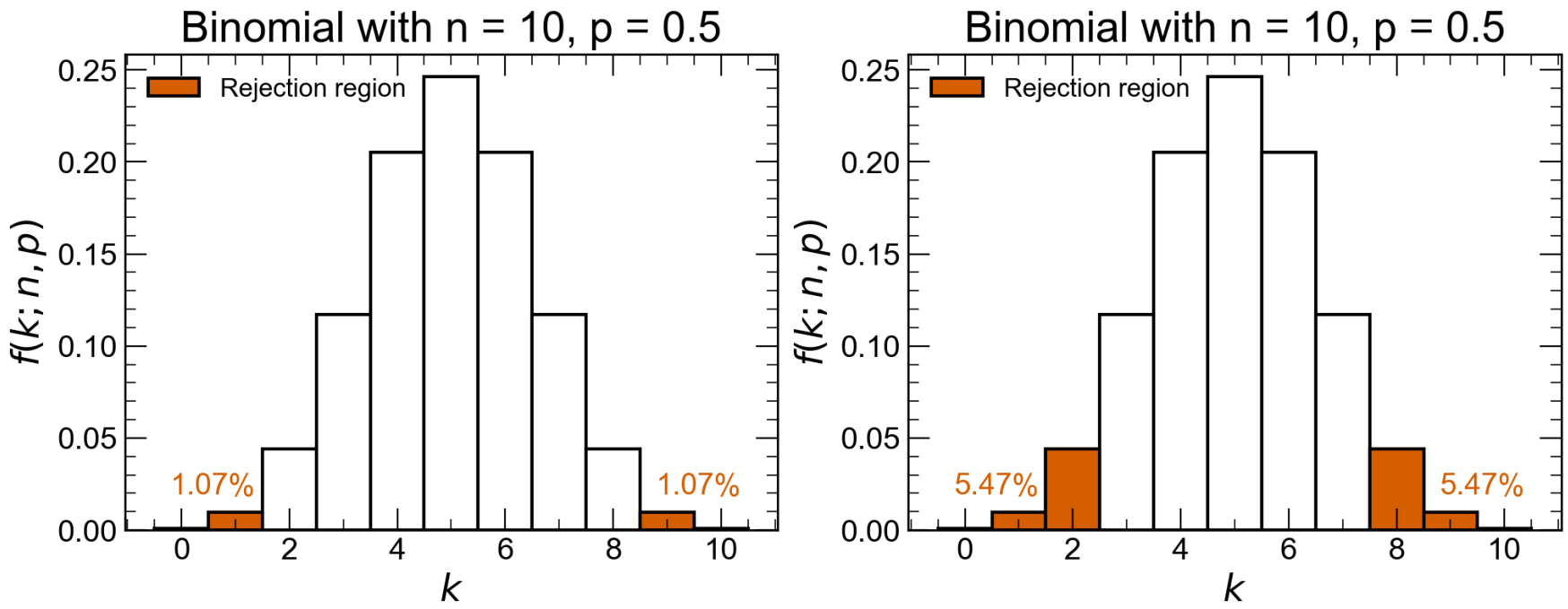
The binomial distribution with $p = 0.5$ and $n = 10$; our null hypothesis that the coin is unbiased, for 10 flips of the coin.

Because we are working with a two-tailed hypothesis test, we need to calculate probabilities symmetrically, i.e. probability of 10 heads or 10 tails, then probability of 9 heads or 9 tails, etc. Under the null hypothesis, these work out as follows:

- 10 heads or tails: 1/1024 for each, total: 2/1024.
- 9 heads or tails: 10/1024 for each, total: 20/1024.
- 8 heads or tails: 45/1024 for each, total: 90/1024.

If we set our rejection region as 10 heads, 10 tails, 9 heads or 9 tails we have a total probability of 22/1024 = 2.1%, which is obviously less than 5%. If we also include 8 heads or tails in the rejection region, the total probability is 112/1024 = 10.9% and so greater than 5%. How can we choose the rejection region?

The figure below shows the two-tailed rejection regions for the coin flipping experiment for the two possible rejection regions described above.



Two possible rejection regions for the coin flipping experiment, labelled by the significance level of the rejection regions.

In general, it is better to label our significance level with a value α which is larger than the actual probability of finding a result in the rejection region.

Consider the continuous case once again: say we choose a 5% rejection region, and make a measurement X_m that falls within this rejection region. However, if we had alternatively chosen a 10% (or 20%, 30% etc) rejection region, it would completely include the 5% rejection region and the measurement X_m would also reject the hypothesis at the 10% level (or 20%, 30% etc) as well as the 5% level. This isn't very helpful.

So in this example with at the $\alpha = 5\%$ we would choose to reject the null hypothesis if we find 9 or 10 of our ten coin tosses gave the same result. In fact we know an unbiassed coin will do that 2.1% of the time, less than $\alpha = 5$. Had we also regjected on 8 heads or 8 tails, then that happens around 10% of the time, larger than α.

Implementations of significance levels

In `scipy.stats`, it is easy to find the values of X to define the rejection region for a given significance level, for a given distribution. As we have seen in earlier code examples, the syntax for different probability distributions in scipy is very similar. All are based on the `rv_continuous` (or `rv_discrete`) class, which is used to define continuous (discrete) random variables. The two most useful methods for finding region regions are `ppf` and `interval`:

- `ppf`: returns the inverse of the CDF for the distribution; this is the same as the manually computed inverse functions we were looking at last week. `ppf(alpha)` of some value $0 \leq \mathrm{alpha} \leq 1$ returns the value of X that gives the cumulative probabilty `alpha` for your distribution
- `interval`: returns the confidence interval with equal areas around the median of the distribution. For desired confidence value `alpha`, uses the `ppf` function to to calculate the boundaries of the interval: returns `ppf([(1 - alpha)/2, (1+alpha)/2])`. The lower bound of the confidence interval is the upper bound on the lower rejection region, and the upper bound of the confidence interval is the lower bound on the upper rejection region.

An example of using this can be seen in the code cell below.

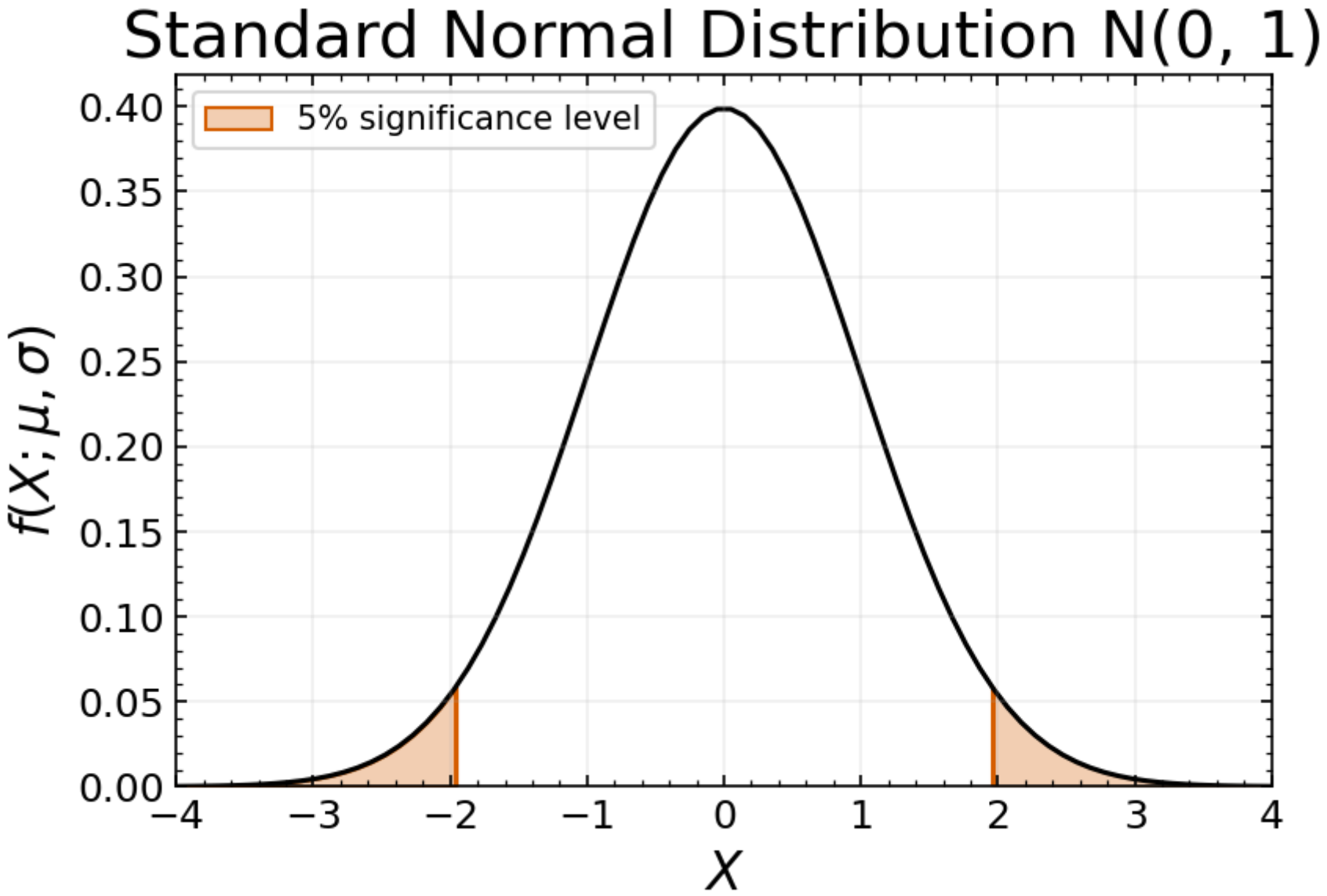
In [2]:
```
alpha = 0.05 # 5% significance level

# Standard normal distribution, 5% significance level corresponds to 95% confidence interval
interval = norm.interval(1 - alpha, loc = 0, scale = 1)
print(interval) # The boundary of this region ought to be the \pm 1.96 we were using in the Gaussian rejection region example
```

```
(-1.959963984540054, 1.959963984540054)
```

We can then plot the PDF and show the rejection region we have calculated:

In [3]:
```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import MultipleLocator
from matplotlib.colors import to_rgba

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.plot(np.linspace(-5,5,100),norm.pdf(np.linspace(-5,5,100)),color='black', zorder = 1)
ax.set_ylim(bottom = 0)
ax.set_xlim(-4,4)
ax.set_xlabel('$X$',fontsize = 16)
```

```
ax.set_ylabel('$f(X; \mu, \sigma)$',fontsize = 16)
ax.tick_params(direction='in',top=True,right=True,which='both',labelsize =12)
ax.plot([interval[0],interval[0]],[0,norm.pdf(interval[0])-0.001],color='#D55E00',zorder = 0)
ax.plot([interval[1],interval[1]],[0,norm.pdf(interval[1])-0.001],color='#D55E00',zorder = 0)
ax.fill_between(np.linspace(-5,interval[0],100),np.repeat(0,100),norm.pdf(np.linspace(-5,interval[0],100)),zorder = 0,fc=to_rg
ax.fill_between(np.linspace(interval[1],5,100),np.repeat(0,100),norm.pdf(np.linspace(interval[1],5,100)),zorder = 0,color='#D5
ax.xaxis.set_minor_locator(MultipleLocator(0.2))
ax.yaxis.set_minor_locator(MultipleLocator(0.01))
ax.legend(loc='upper left')
ax.set_title('Standard Normal Distribution N(0, 1)',fontsize = 20)
ax.grid('xkcd:dark blue',alpha = 0.2)
```



Summary

In this section, we have presented the basic concepts of hypothesis testing, including:

- Rejection regions and significance levels
- Alternate hypotheses
- One-sided and two-sided hypothesis tests
- Hypothesis tests of discrete distributions
- Basics of significance level implementations in `scipy`

In the following section, we will discuss two different tests for measuring the goodness of fit of a model to data, and how this can link back to hypothesis testing.

Section Three: Goodness of Fit^

Intrinsically related to hypothesis testing is the idea of **goodness of fit**. You will learn in detail about fitting and parameter estimation next week, but for now know that it concerns using data you have measured to determine the best parameters of a given model to fit to the data.

After we have determined some estimate for the parameters of the model for our data, we want to determine how well the model fits the data. This is referred to as finding the goodness of fit. We will talk first about two common tests for goodness of fit

- χ^2 test
- Kolmogorov-Smirnov test Then see how they relate to hypothesis testing and discuss **p-values**.

The χ^2 goodness of fit test

The χ^2 test is probably the best known test used for goodness of fit. Formally, for a set of N_{data} measurements $\{X_i, y_i\}$ (**independent variable** X_i controlled by us while the **dependent variable** y_i is the output we measure) and some fit function $f(X; \boldsymbol{\theta})$ with list of parameters $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_d)$, we want to work out if the function $f(X; \boldsymbol{\theta})$ models the data well. That is any deviations in the data from what our fit function f predicts can be understood as expected statistical noise.

To calculate the χ^2, we first must calculate the **pull** of each point, which is the difference between the data y_i and the prediction at the same point $f(X_i; \boldsymbol{\theta})$, weighted by the uncertainty σ_i on the measurement y_i. This is written as:

$$p_i = \frac{y_i - f(X_i; \boldsymbol{\theta})}{\sigma_i},$$

where p_i is the pull for the i-th data point.

We can then define the χ^2 as a function of the parameters $\boldsymbol{\theta}$, given the measurements y_i, as the sum of the square of the pulls:

$$\chi^2(\boldsymbol{\theta}; y_i) = \sum_i p_i^2 = \sum_i \frac{(y_i - f(X_i; \boldsymbol{\theta}))^2}{(\sigma_i)^2},$$

where here we have explicitly written the χ^2 as a function of the parameters $\boldsymbol{\theta}$ of the fit function $f(X; \boldsymbol{\theta})$. We can use the χ^2 to estimate parameters, which you will see next week.

As well as being able to use this quantity to optimise the choice of parameters $\boldsymbol{\theta}$, the value of the χ^2 for specific $\boldsymbol{\theta}$ values tells us about the quality of the fit.

We will first make one key assumption: that if our fit is a good fit, the y_i are random variables distributed according to a normal distribution $N(f(X_i; \boldsymbol{\theta}), \sigma_i)$ i.e. with mean $f(X_i; \boldsymbol{\theta})$ and standard deviation σ_i. The pulls p_i are therefore standard normally distributed random variables.

If we have a good fit, we expect the y_i to deviate from the mean by values similar to σ_i, under this assumption of a normal distribution. We therefore expect the square of each pull $p_i^2 \sim 1$, so the χ^2 value for a good fit should be $\sim N_{\text{data}}$ i.e. similar to the number of data points.

For such a basic statistic, χ-square can be very confusing, see the optional note below for some more comments. Part of the problem is that we have *two* distributions here which are completely unrelated.

The first distribution we have is $f(X; \boldsymbol{\theta})$. The f distribution is our model with parameters $\boldsymbol{\theta}$ telling us that for a given X the expected *mean* value for what we measure is $f(X; \boldsymbol{\theta})$. Note in many cases, X is no longer a random variable, it is some experimental input variable we control. In that case, f is now our theoretical model that depends on some unknown parameters $\boldsymbol{\theta}$ we wish to measure.

The second distribution we have is the Gaussian probability distribution where we say that even if we have a measurement at X_i we are not guaranteed to measure the mean value $f(X; \boldsymbol{\theta})$ but we know that the deviation from the mean $(y_i - f(X; \boldsymbol{\theta}))$ is Gaussian distributed with a standard deviation σ_i. We can estimate σ_i by repeating our measurement at X_i and derive σ_i from these several experimental y values. However, be careful, if you use a mean value from several experiments at X_i to give you your y_i value then you must use the standard error of the mean to set σ_i.

Degrees of freedom and χ^2_ν

In practice, rather than comparing the noise in the results to that expected from N_{data} Gaussians, we actually need to compare to the number of **degrees of freedom** N_{dof}, which is the difference between the number of data points N_{data} and the number of fit parameters N_{params}:

$$N_{\text{dof}} = N_{\text{data}} - N_{\text{params}}$$

This is because we can think of each measurement y_i as giving us an equation for $f(X_i; \boldsymbol{\theta}), \sigma_i)$. So at the very least if we have N_{params} parameters to fix we will need N_{params} pieces of information y_i to fix them. For $N_{\text{dof}} < 0$, the system is underconstrained and cannot be uniquely solved.

In general we have many more data points than parameters $N_{\text{dof}} > 0$ and a fit is needed to find the best of many possible solutions. However, even in this case, we can think of the N_{params} parameters chosen as giving us N_{params} equations limiting the data so really only $N_{\text{dof}} = N_{\text{data}} - N_{\text{params}}$ peices of information are unconstrained and so these remaining measurements can have some random noise (assumed Gaussian in this context). So when we are trying to estimate if the data shows a reasonable amount of randomness, we need to think of the noise as coming from N_{dof} variables, in this case normal distributed.

This is why it is useful to define what is called the **reduced chi-square statistic** χ^2_ν

$$\chi^2_\nu = \frac{\chi^2}{N_{\text{dof}}}$$

which is the χ^2 per degree of freedom. Our naive estimate for χ^2 suggests that this reduced chi-square statistic χ^2_ν should *always* be around 1.0. In more detail:-

- If χ^2_ν is around 1.0 then we may have a good fit.

- For values of χ^2_ν much greater than 1, average deviation is much greater than 1σ:
    1. The chosen model could have too few parameters to model the data well; you could be **underfitting**.
    2. The choice of parameters $\boldsymbol{\theta}$ may not be correct even if the type of model is correct.
    3. The uncertainty on measurements σ_i could be underestimated.

- For values of χ^2_ν much less than 1, the average deviation is much less than 1σ:
    1. The chosen model could have too many parameters for the data; you could be **overfitting**.

2. The uncertainty on each data point could be overestimated.

Because there are multiple possible explanations for different values of the reduced χ^2, you need to be careful how you interpret your results. Look at the optional note below on some of the common problems.

Optional exercise: find the $\chi^2(N_{\rm dof}=1)$ distribution

A more precise estimate of the probability that we accept/reject a given value comes from knowing the probability distribution of a sums of squares of $N_{\rm dof}$ random variables, where each variable comes from a standard normal distribution. You can write this expression down. If you know the integral representation of the Gamma function $\Gamma(z)$ you might even derive an expression for the probability distribution for the expected values of χ^2 with $\chi^2(N_{\rm dof}=1)$, i.e. the PDF for the square of a single gaussian random variable. However the distribution of expected χ^2 values for any given $N_{\rm dof}$ is used so often that this is is a standard named function, the chi-squared distribution mentioned in week 2, and library routines exist to do all the hard work you might need.

WARNING: Common problems when using χ^2

The literature and even python library functions are often very confused about the correct general form of the χ^2 statistic which is what we have defined carefully here.

> The simple rule is that if you do not have explicit values for σ_i in your numerical routine for χ^2, one for every measurement y_i, then you are probably making assumptions which may well be wrong.

There are two common problems.

- The routine you are using will accept a list of σ_i values but it also runs without them, setting default values of $\sigma_i=1$ if you don't give explicit vaues. This default just gives you a simple sum of square differences which normally has no particular role in statistics.
- Your χ^2 routine does *not* have the option to provide any σ_i values because it sets $(\sigma_i)^2=y_i$. The reason for this is that the routine is assuming Poisson statistics where the mean is equal to the variance. This is appropriate when each measurement y_i is a count and the count is reasonably large (say greater than 5 as a rough rule of thumb) so that the Poisson is a good approximation for a Binomial distribution (which is what we really have). For example y_i could be the number of people in the Physics department who have heights h in centimetres which fall in the range $10*(i)\leq h<10*(i+1)$. The form where $(\sigma_i)^2=y_i$ is known as the **Pearson chi-squared statistic** but it is often incorrectly referred to as *the* chi-squared statistic.

Even if you do have to give σ_i values to your routine, make sure they are standard deviations for the actual values y_i. A common problem is if y_i is some sort of average (e.g. over several repreated measuyrements) you use the standard deviation obtained from the set of measurements for y_i not the standard error of the mean.

A simple check is to look at the reduced chi-square statistic χ^2_ν. If χ^2_ν is ridiculously small or far too large given what you can see in the data, you have probably made a mistake.

Kolmogorov-Smirnov tests

Named after Andrey Kolmogorov and Nikolai Smirnov, the Kolmogorov-Smirnov test, often just the K-S test or the KS test, is one of the most common goodness of fit tests. It is used in two ways.

The **one-sample** KS test compares a data sample with a reference probablity distribution. The question this test asks is

- how likely is it that we would see these data samples if the data is drawn from that probability distribution?

The **two-sample** KS test compares two different data samples. This test asks

- how likely is it that we would see these two sets of data samples if they are drawn from the same (but unknown) probability distribution?

The KS test is based around the CDF (cumulative probability distribution) not the PDF (probability density function). So to test empirical data, we first need to define the the **empirical cumulative distribution** of the data $F_{\rm data}(X)$. This is defined as follows:

$$F_{\rm data}(X) = \frac{\text{number of data values} < X}{n},$$

where n is the number of measurements in the dataset.

We will describe the KS tests in terms of the following simple model. Consider the standard normal distribution $N(0,1)$. We can generate random samples from this dataset using `scipy.stats`.

We start by finding the empirical cumulative distribution for our data.

In [4]:
```
# This code cell creates the data
from scipy.stats import norm
```

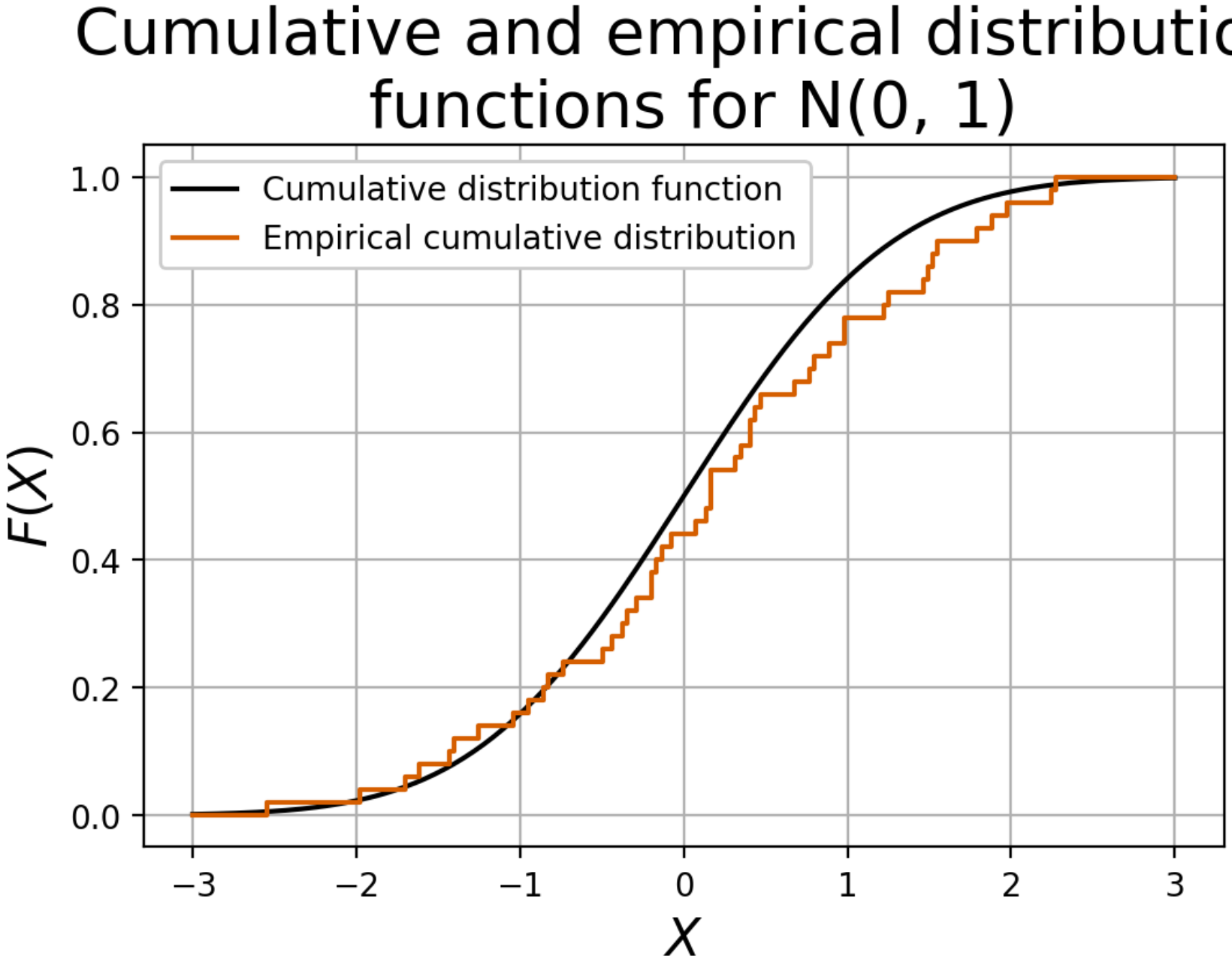
```
import numpy as np

# Set the random seed for consistency. scipy.stats uses the numpy random number generator, so why not use 0?
np.random.seed(0)

number_values = 50 # this is the number of data values to use
samples = norm.rvs(loc=0.0, scale=1.0, size = number_values) # the mean is loc=0, the standard deviation is scale=1.0

def Fd(X):
    """ This gives the empirical cumulative distribution of the data in "samples" for given X value """
    return np.count_nonzero(samples<X)/len(samples)

# Values of X to use here when **plotting**, it has nothing to do with location of data in samples
Xmin=-3.0
Xmax=3.0
Xvalues = np.linspace(Xmin, Xmax, number_values*4)

empirical = [Fd(X) for X in Xvalues] # This gives the  empirical cumulative distribution for many values of X
```

In [5]:

```
# This code cell plots empirical cumulative distribution from our data with the theoretical CDF
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 200)
ax.plot(Xvalues,norm.cdf(Xvalues),color='black', label = 'Cumulative distribution function')
""" OK now comes a slight cheat.
The empircal CDF is defined everywhere but will have steps at each value of X in teh data.
However the function below has steps at each value in the `Xvalue` array which is purely the
list of points used for PLOTTING, graphics, nothing to do with the actual data.
Really I should use the ax.stair function and use the actual X values in the data  `samples`.
"""
ax.step(Xvalues,empirical,color='#D55E00',label='Empirical cumulative distribution',where='post')
ax.legend(loc='upper left',framealpha = 1)
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('$F(X)$',fontsize =16)
ax.grid()
ax.set_title('Cumulative and empirical distribution\n functions for N(0, 1)',fontsize = 20)
```

Out[5]:

```
Text(0.5, 1.0, 'Cumulative and empirical distribution\n functions for N(0, 1)')
```


Cumulative and empirical distribution functions for N(0, 1)
Cumulative distribution function
Empirical cumulative distribution
1.0
0.8
0.6
0.4
0.2
0.0
F(X)
−3
−2
−1
0
1
2
3
X

As you can see, while the empirical distribution is close to the cumulative distribution it does vary. If we increase the amount of data, the empirical distribution will get closer to the true cumulative distribution function. In fact, in general the Kolmogorov-Smirnov test performs best for large amounts of data, because then the empirical distribution better approximates the true distribution that describes the data, regardless of whether that is the same as the distribution you are comparing to or not.

One-sample KS test

For the one-sample test, we then wish to compute the maximum absolute distance between the empirical cumulative distribution $F_{\rm data}(X)$ and the cumulative distribution function $G(X)$. This can be written as

$$D = \max_X |F_{\rm data}(X) - G(X)|,$$

where $\max_X$ means that we take the largest value of $|F_{\rm data}(X) - G(X)|$ over all values of X in the empirical data set.

In general, larger values of D indicate worse agreement between the CDFs. We can calculate this for our standard normal data:

In [6]:
```
samples_cumulative = [Fd(X) for X in samples]

D = np.max(np.abs(samples_cumulative - norm.cdf(samples)))

print(D)
```

```
0.10706475374815838
```

We can all try to change the random number seed above (e.g. try the last three digits of your CID number) and repeat to see what difference it makes. I found $D \approx 0.107$ for seed zero on my machine but the same seed on different machines may generate different answers.

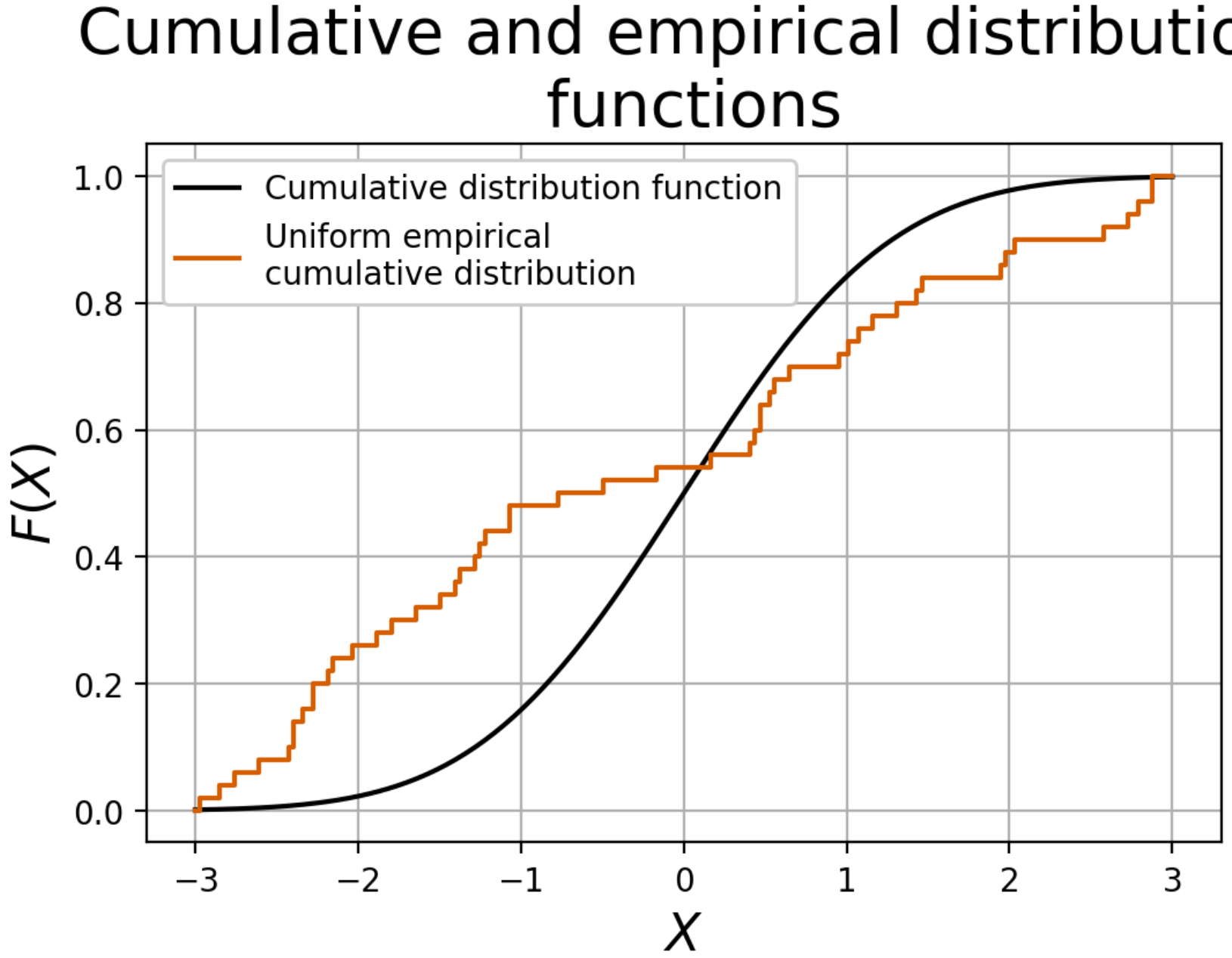
So far we have generated empirical data from the distribution we are examining so the data will be a good fit to the theoretical distribution.

Let us try instead generating data from a uniform distribution between -3 and 3, denoted $U(-3, 3)$, and comparing the distribution of the data with the standard normal distribution.

In [7]:
```
# Now repeat analysis above but for uniform distribution, all combined into this one cell.
from scipy.stats import uniform
```

```
# Number of X values and the values of X are set above Xvalues = np.linspace(Xmin, Xmax, number_values*2) etc
# number_values=50, Xmin=-3.0, Xmax=3.0, Xvalues = np.linspace(Xmin, Xmax, number_values*2)
uniform_samples = uniform.rvs(loc = Xmin, scale = Xmax-Xmin, size = number_values)

def Fd_uniform(X):
    return np.count_nonzero(uniform_samples < X)/len(uniform_samples)

empirical_uniform = [Fd_uniform(X) for X in uniform_samples]

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 200)
ax.plot(Xvalues ,norm.cdf(Xvalues),color='black', label = 'Cumulative distribution function')
ax.step(Xvalues,[Fd_uniform(X) for X in Xvalues],color='#D55E00',label='Uniform empirical \ncumulative distribution',where
ax.legend(loc='upper left',framealpha = 1)
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('$F(X)$',fontsize =16)
ax.grid()
ax.set_title('Cumulative and empirical distribution\n functions',fontsize = 20)

D_uniform = np.max(np.abs(empirical_uniform - norm.cdf(uniform_samples)))

print(D_uniform)
```

```
0.32183181629623536
```


Cumulative and empirical distribution functions
Cumulative distribution function
Uniform empirical cumulative distribution
1.0
0.8
0.6
0.4
0.2
0.0
F(X)
−3
−2
−1
0
1
2
3
X

We can see that the value for D in this case is significantly larger than for data generated according to the standard normal distribution. I found $D \approx 0.321$ when comparing Uniform distribution data to the normal distribution while $D \approx 0.107$ when comparing using normal distribution data and the theoretical normal distribution.

How do we decide what value for D indicates a good fit? We will see later how to do this formally.

EFS: Well a quick and dirty way to do this is to change the seed and rerun the code to see how much the D values change. Do this once and see if you think that the data from a uniform distribution will always be significantly worse fit than the data from a standard normal distribution to the theoretical normal distribution.

So far we have written our own code to do the KS test. It wasn't too hard. However, trying to work out in a more rigorous way what the D values mean is much harder. Of course all this work has been done for us.

In fact, the one-sample Kolmogorov-Smirnov test is implemented in `scipy.stats` as `ks_1samp`. We can use it as follows:

In [8]:
```
from scipy.stats import ks_1samp

d1_D, d1_p = ks_1samp(samples, norm.cdf)
unif_D,unif_p = ks_1samp(uniform_samples, norm.cdf)

print("For sample from normal distribution:  D = {:.3f} and p-value = {}".format(d1_D,d1_p))
print("For sample from uniform distribution: D = {:.3f} and p-value = {}".format(unif_D,unif_p))
```

```
For sample from normal distribution:  D = 0.107 and p-value = 0.5781417630622738
For sample from uniform distribution: D = 0.342 and p-value = 1.0171478278197228e-05
```

This returns not only the value of D, but also the p-value of the test (like we have seen for hypothesis testing). We will see how we can calculate p-values ourself for Kolmogorov-Smirnov tests later. We can clearly see that the normally distributed sample is much more likely to belong to the hypothesised distribution than the uniformly distributed sample (as we expected).

Two-sample KS test

For a two-sample Kolmogorov-Smirnov test, rather than comparing empirical data with a well-defined distribution, we are comparing two different sets of empirical data. In this case, we need to compute the empirical cumulative distribution for both datasets. This is particularly useful in machine learning, to help evaluate performance of your model; you can try prediction on some of your training data, and compare the predicted values with some test data. You will see more of these concepts later in this course.

Here we will denote the empirical cumulative distributions for the two samples as $F_{\rm data}(X)$ and $G_{\rm data}(X)$ respectively. This test will allow us to determine if the two different empirical datasets are from the same underlying distribution. The distance D_2 is calculated according to

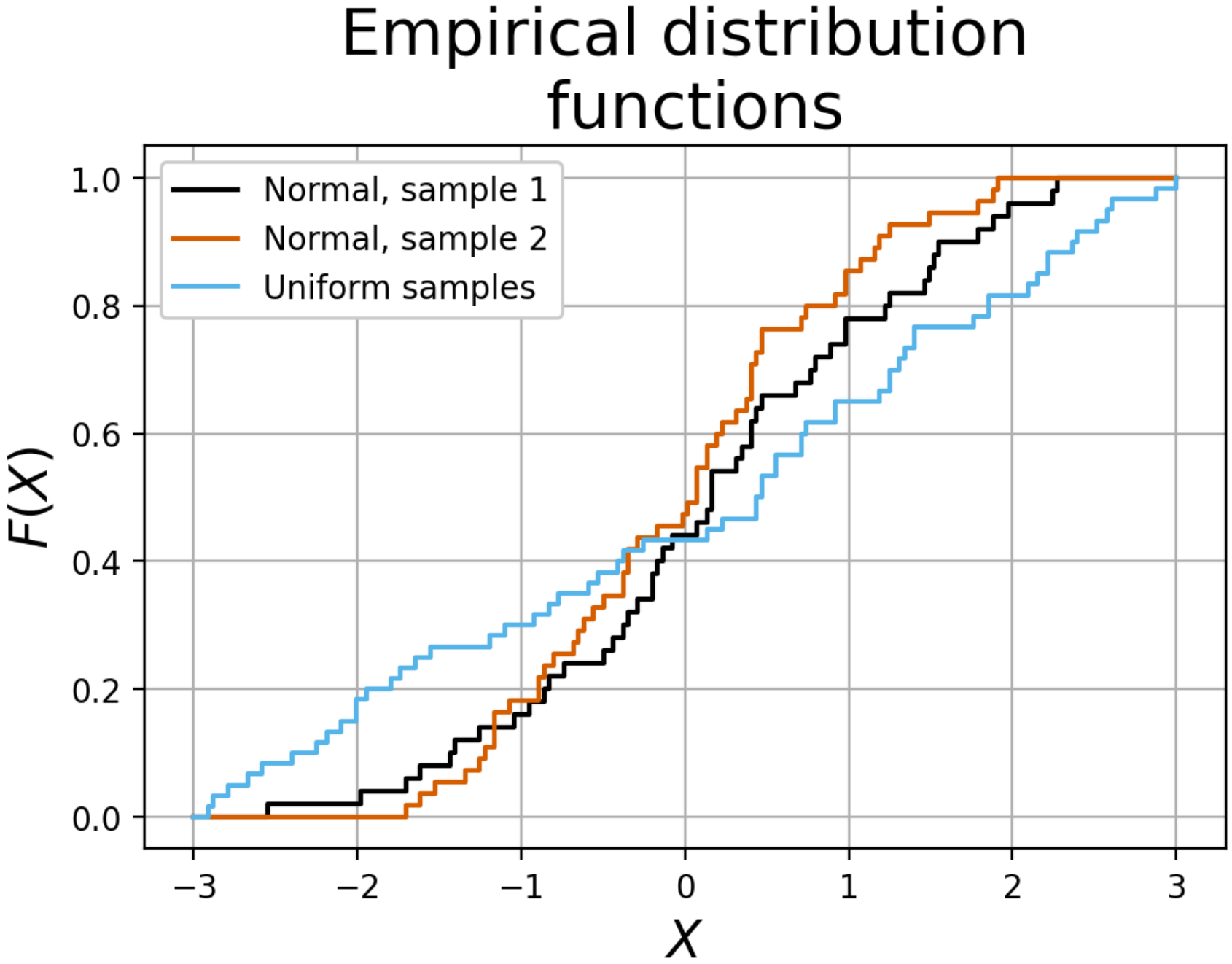
$$D_2 = \sup_X |F_{\rm data}(X) - G_{\rm data}(X)|.$$

where symbols have the same meanings as before. Once again, larger values of D_2 indicate greater disagreement between the two distributions. We can see this by comparing some normally distributed data with either more normally distributed data, or uniformly distributed data:

In [9]:
```
np.random.seed(0) # Set the random seed for consistency

# Range of X values in the data are as above, that is Xmin=-3.0, Xmax=3.0 for uniform distribution.
# The exponential cutoff in the normal should exclude values outside this range.

# Remember Xvalues is only used to choose what points to use when plotting.

# NOTE KS 2 sample does NOT require the same number of points in each sample, number_values=50,
# so I will keep the same X range but change the number of points in each case

samples_normal_1 = norm.rvs(size = number_values)
samples_normal_2 = norm.rvs(size = number_values+5)
samples_uniform = uniform.rvs(loc=Xmin, scale=Xmax-Xmin,size = number_values+10)

def Fd_normal_1(X):
    return np.count_nonzero(samples_normal_1 < X)/len(samples_normal_1)

def Fd_normal_2(X):
    return np.count_nonzero(samples_normal_2 < X)/len(samples_normal_2)

def Fd_uniform(X):
    return np.count_nonzero(samples_uniform  < X)/len(samples_uniform)

# Note Xvalues is just used to get points when plotting, has nothing to do with where data is.
empirical_normal_1 = [Fd_normal_1(X) for X in Xvalues]
empirical_normal_2 = [Fd_normal_2(X) for X in Xvalues]
empirical_uniform  = [Fd_uniform(X)  for X in Xvalues]

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 200)
ax.step(Xvalues, empirical_normal_1, color='black',   label = 'Normal, sample 1', where='post')
ax.step(Xvalues, empirical_normal_2, color='#D55E00', label = 'Normal, sample 2', where='post')
```

```
ax.step(Xvalues, empirical_uniform,  color='#56B4E9', label = 'Uniform samples',  where='post')

ax.legend(loc='upper left',framealpha = 1)
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('$F(X)$',fontsize =16)
ax.grid()
ax.set_title('Empirical distribution\n functions',fontsize = 20)
```

Out[9]:
```
Text(0.5, 1.0, 'Empirical distribution\n functions')
```


Empirical distribution functions
Normal, sample 1
Normal, sample 2
Uniform samples
1.0
0.8
0.6
0.4
0.2
0.0
F(X)
−3
−2
−1
0
1
2
3
X

To evaluate the Kolmogorov-Smirnov statistic for the two sample case, we should evaluate each CDF at each point in both sets of samples.

In [10]:
```
# The next two lists give the X values where there is data from either one of the two data sets being tested.
all_samples_normal  = np.concatenate([samples_normal_1, samples_normal_2])
all_samples_uniform = np.concatenate([samples_normal_1, samples_uniform])
all_samples_uniform2= np.concatenate([samples_normal_2, samples_uniform])

D_normal  = np.max(np.abs(np.array([Fd_normal_1(X) for X in all_samples_normal])   - np.array([Fd_normal_2(X) for X in a
D_uniform = np.max(np.abs(np.array([Fd_normal_1(X) for X in all_samples_uniform])  - np.array([Fd_uniform(X)  for X in a
D_uniform2= np.max(np.abs(np.array([Fd_normal_2(X) for X in all_samples_uniform2]) - np.array([Fd_uniform(X)  for X in a

print("Comparing normal sample 1 and normal sample 2: D = {}".format(D_normal))
print("Comparing normal sample 1 and uniform sample:  D = {}".format(D_uniform))
print("Comparing normal sample 2 and uniform sample:  D = {}".format(D_uniform2))
```

```
Comparing normal sample 1 and normal sample 2: D = 0.12909090909090903
Comparing normal sample 1 and uniform sample:  D = 0.19333333333333333
Comparing normal sample 2 and uniform sample:  D = 0.2606060606060606
```

Clearly, the agreement is much better between normal sample 1 and normal sample 2 than between normal sample 1 and the uniform sample (as you would expect).

Once again, the 2-sample KS test is implemented in `scipy.stats` as `ks_2samp`. Again, this also returns the p-value of the test, which we will discuss later.

In [11]:
```
from scipy.stats import ks_2samp

print('Comparison of normal 1 and normal 2: D = {}, p-value = {}'.format(*ks_2samp(samples_normal_1, samples_normal_2)))
print('Comparison of normal 1 and uniform : D = {}, p-value = {}'.format(*ks_2samp(samples_normal_1, samples_uniform)))
print('Comparison of normal 2 and uniform : D = {}, p-value = {}'.format(*ks_2samp(samples_normal_2, samples_uniform)))
```

```
Comparison of normal 1 and normal 2: D = 0.1290909090909091, p-value = 0.7151099547593596
Comparison of normal 1 and uniform : D = 0.1933333333333333, p-value = 0.2280575658333467
Comparison of normal 2 and uniform : D = 0.2606060606060606, p-value = 0.03229736254064738
```

Test statistics and hypothesis testing

So far we have talked about a couple different tests for goodness of fit, but not really talked about how we can identify what values of these tests give a good fit. In fact, both the Kolmogorov-Smirnov tests and the χ^2 test link back to hypothesis testing.

In general, when we are evaluating a fit, we calculate a **test statistic** that describes our data, in the context of the model we are trying to fit/compare with. In the case of the Kolmogorov-Smirnov test, the test-statistic is the distance D, while in the case of the χ^2 test the value of the χ^2 itself is the test-statistic. Because these test-statistics are functions of random variables, they themselves are random variables and are therefore samples from some distribution.

p-values

A **p-value** gives the probability p of getting the observed value t or a worse value for a test statistic. The statistic value t is calculated from some data. The probability distribution used to find p is the distribution for the test statistic given that the data comes from null model, our null hypothesis H_0.

If the the probability density distribution for the possible test statistic values is written as $p(t|H_0)$ (indicating this is the probability for t assuming H_0 is true) and if larger t values are worse, then the p-value is simply found from the CDF

$$p = P(t|H_0) = \int_t^\infty dt'\, p(t'|H_0)\,.$$

Note $p(t|H_0)$ is *not* our PDF $f(X)$ as we wrote in our first discussion of hypothesis testing. This distribution $p(t|H_0)$ needs to be derived from $f(X)$ and will change depending on the test statistic used. So more work needs to be done and this work is typically not trivial and often it is not even analytically tractable. Again for standard cases, the work has been done for us.

Roughly speaking, a low p-value means we reject the null hypothesis.

Dire warning about p-values

These p-values are widely used in statistical analysis but they are often misunderstood, misused and abused. They are often produced by library functions and then reproduced without any thought, even in published academic papers. These p-values can carry meaning but don't simply accept them at face value; think about how they were obtained.

Calculating p-values from χ^2 tests

The test statistic here is χ^2, the sum of the square of the pulls $p_i = y_i - f(X_i)$

$$\chi^2 = \sum_i \frac{(y_i - f(X_i;\boldsymbol{\theta}))^2}{(\sigma_i)^2}$$

where $f(X)$ is the PDF, our null model being tested, y_i is the measured value obtained at X_i, and σ_i is our estimate of the standard deviation in the measurement of y_i.

Last week, we saw that under the null model that the χ^2 statistic is the sum of the squares of standard normal random variables, then the possible values of the test statistic χ^2 is given by a standard distribution known as the **chi-squared distribution** which has one parameter, the number of degrees of freedom.

Notice how this one family of chi-squared distributions always gives the the distribution of χ^2 values regardless of the shape of the PDF $f(X;\boldsymbol{\theta})$ whose form is part of our null model.

Because the χ^2 is a sample from a probability distribution, we can calculate a p-value for the fit of $f(X_i;\boldsymbol{\theta})$ to our data for a given set of parameter values $\boldsymbol{\theta}$. That is we can ask how likely is it that we would draw a value equal to or higher than the value of χ^2 that we measure. Effectively, our χ^2 test is a hypothesis test with the null hypothesis being that each measurement y_i is distributed according to a normal distribution with mean $f(X_i;\boldsymbol{\theta})$ and standard deviation σ_i.

For a given dataset with N_{dof} degrees of freedom, we can calculate the p-value of the fit parameters $\boldsymbol{\theta}$ using the χ^2. We can do this with a one-sided test of the χ^2 distribution and our value of χ^2.

Example

Consider a fit of some data to a given theory where we have $N_{\text{dof}} = 5$ and $\chi^2 = 7.64$.

We start with a quick rough check and we see that the reduced chi-squared (to 2 d.p.) is

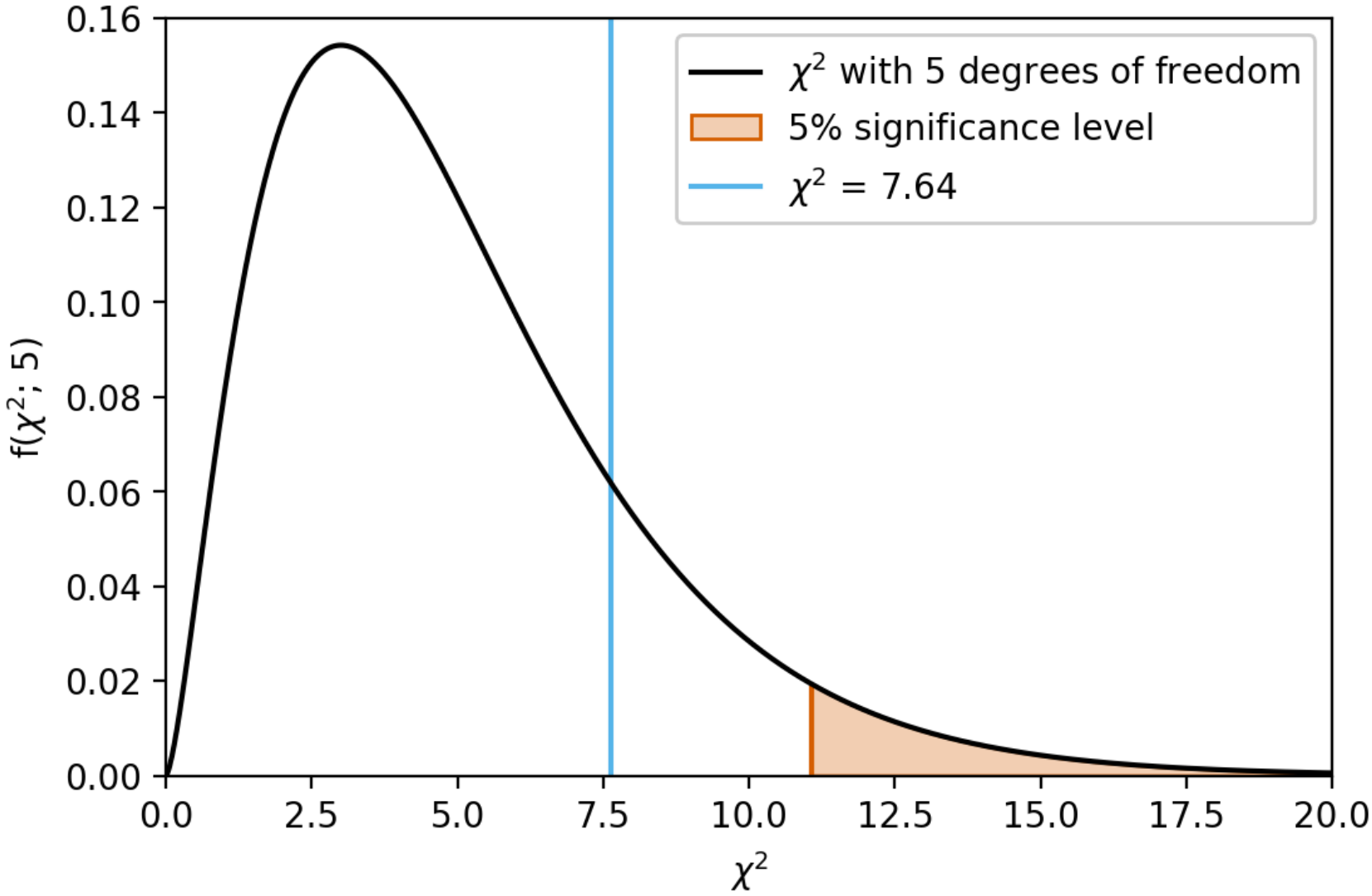
$$\chi^2_\nu = \chi^2 / N_{\text{dof}} = 7.64/5 = 1.53$$

This is reasonably close to 1.0 so we might guess that the theory is is not a bad fit to the data.

To be more precise, we can choose a significance level of 5% and perform a hypothesis test that our test statistic χ^2 is distributed according to the χ^2 distribution with 5 degrees of freedom:

In [12]:
```
from scipy.stats import chi2 # This is the chi-square distribution, not the chi-square statistic
from matplotlib.colors import to_rgba

chi2_value = 7.64

X = np.linspace(0,20,1000)
fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 200)
ax.plot(X, chi2.pdf(X, 5), label='$\chi^2$ with 5 degrees of freedom',color='black',zorder = 1)
ax.plot([chi2.ppf(0.95,5),chi2.ppf(0.95,5)],[0,chi2.pdf(chi2.ppf(0.95,5),5)-0.001],zorder = 0,color='#D55E00')
ax.fill_between(np.linspace(chi2.ppf(0.95,5),25,1000),np.repeat(0,1000),chi2.pdf(np.linspace(chi2.ppf(0.95,5),25,1000),
ax.plot([chi2_value, chi2_value],[0,0.17],color='#56B4E9',zorder = 0,label=r'$\chi^2$ = {:4.2f}'.format(chi2_value))
ax.set_ylim(bottom = 0, top = 0.16)
ax.set_xlim(0,20)
ax.legend(loc='upper right',framealpha = 1)
ax.set_xlabel('$\chi^2$')
ax.set_ylabel('f($\chi^2$; 5)')
```

Out[12]:
```
Text(0, 0.5, 'f($\\chi^2$; 5)')
```



Clearly, our value of χ^2 is outside of our chosen rejection region and so we do not reject the hypothesis that our χ^2 is distributed according to the χ^2 distribution with 5 degrees of freedom. We therefore do not reject the hypothesis that our data is normally distributed about our model with standard deviation σ_i.

What is the p-value for our data? We can calculate it using `chi2.sf` (sf = "survival function"), which is equivalent to 1 - `chi2.cdf`:

In [13]:
```
print(chi2.sf(chi2_value, 5))
```

```
0.17722382548124416
```

We therefore say that there is a 17.7% chance our data is distributed according to our null hypothesis. This is perhaps worse that we might have hoped from our reduced chi-square value but still not enough to reject our null hypothesis given our significance level.

Calculating p-values from Kolmogorov-Smirnov scores

Much like we have calculated p-values from χ^2 values, we can do the same for Kolmogorov-Smirnov tests. We need to do this slightly differently for either one-sample or two-sample tests.

We will first setup up some common formalism between both problems. Let us denote the true cumulative distribution of the data as $F(X)$, and the other cumulative distribution is $G(X)$. This $G(X)$ which can either be the analytic distribution we are testing against for a one-sample Kolmogorov-Smirnov test, or the true distribution of the second sample we are testing against for a two-sample Kolmogorov-Smirnov test.

We can look at performing either a one-sided or a two-sided hypothesis test on the Kolmogorov-Smirnov statistic. These correlate to different null and alternate hypotheses:

- Two-sided test: Null hypothesis is $F(X) = G(X)$; alternate hypothesis is $F(X) \neq G(X)$
- One-sided test: there are two possible tests:
    1. Null hypothesis: $F(X) \geq G(X)$ for all X; alternate hypothesis $F(X) < G(X)$ for at least one value of X
    2. Null hypothesis: $F(X) \leq G(X)$ for all X; alternate hypothesis $F(X) > G(X)$ for at least one value of X

The choice of specific hypothesis test also slightly changes our choice of test statistic; for example, if we are only interested in testing the null hypothesis that $F(X) \geq G(X)$, we don't care about any cases where $F(X) > G(X)$ and instead the points that influence our test the most are the points where $F(X) < G(X)$. As a result, we can define two other test statistics. As before, $F_{\text{data}}(X)$ denotes the cumulative distribution function of data sampled from some probability distribution with true CDF $F(X)$.

$$D_+ = \sup_X(F_{\text{data}}(X) - G(X))$$
$$D_- = \sup_X(G(X) - F_{\text{data}}(X)),$$

where these statistics are written for the one-sample case but are defined equivalently for the two-sample case, just replacing $G(X)$ with $G_{\text{data}}(X)$.

The two-sided test statistic D can then be seen as $\max(D_+, D_-)$.

P-values for one-sample Kolomogorov-Smirnov tests

The one-sample Kolmogorov-Smirnov test aims to determine if a set of experimentally measured data $\{X_i\}$ is distributed according to some probability distribution $f(X)$ with corresponding CDF $F(X)$. Depending on our choice of a one-sided or two-sided test, the null hypothesis may vary but typically is that the data *is* distributed according to the probability distribution $f(X)$.

We will consider a set of data $\{X_i\}$ and find the D value, the largest absoluted difference between X_i and the correpsonding $f(X_i)$. We want to calculate the probability that this value is consistent with the probability distribution for the X, that is $f(X)$.

If the null hypothesis is true, we expect D to be a random variable distributed according to the Kolmogorov-Smirnov distribution. This distribution depends on the number of measurements we had but it has a horrible functional form that we will not present here, but it is implemented well in `scipy.stats` for both one-sided and two-sided tests, as `scipy.stats.ksone` and `scipy.stats.kstwo` respectively.

First, we will calculate the p-value for a two-sided test for a sample generated from a standard normal distribution, the $\{X_i\}$, and we compare this to the standard normal distribution itself, the $f(X)$.

In [14]:
```
from scipy.stats import norm, kstwo
np.random.seed(0) # Set the random seed for consistency

samples = norm.rvs(loc = 0, scale = 1, size = 50) # Generate the sample data we wish to test

xi_cumulative = [np.count_nonzero(samples < X)/len(samples) for X in samples] # Calculate empirical cumulative distrib

D = np.max(np.abs(xi_cumulative - norm.cdf(samples,loc = 0, scale = 1))) # Calculate D

p_value = kstwo.sf(D,len(samples))

print("D = {:.3f}, p-value = {}".format(D,p_value))
```

```
D = 0.107, p-value = 0.5781417630622738
```

Of course, as this is a common task, there is a single command that will do this for us. So we can compare the value we calculate step by step above with one we find using `scipy.stats.kstest`, which is a function that can be used for both one-sample and two-sample Kolmogorov-Smirnov tests.

In [15]:
```
from scipy.stats import kstest
#kstest(samples, norm.cdf)
ks_result = kstest(samples, norm.cdf)
print("Now D = {:.3f}, p-value = {}".format(ks_result.statistic,ks_result.pvalue))
print("Slight difference in p-value = {}".format(p_value-ks_result.pvalue))
```

```
Now D = 0.107, p-value = 0.5781417630622738
Slight difference in p-value = 0.0
```

Note: it is possible that you might see some slight discrepancies between these values. This is because the implementation in `scipy` treats the empirical cumulative distribution differently for D_+ and D_-, the reason for which is beyond the scope of this course. For high sample sizes, these results will converge. In practise you should use the `scipy` implementation rather than coding a method yourself.

We can set significance levels for Kolmogorov-Smirnov values much like we can in any hypothesis test. For example, if we chose a 5% significance level for the data we have just computed the statistic for, we would accept the null hypothesis that the data is distributed according to the normal distribution.

We can also calculate the statistic and the p-value for an empirical dataset that should not agree with the normal distribution. We will as before generate data corresponding to a uniform distribution between -3 and 3, denoted $U(-3,3)$.

In [16]:
```
np.random.seed(0) # Set the random seed for consistency

u_samples = uniform.rvs(loc = -3, scale = 6, size = 50) # Now data is from the uniform distribution

u_cumulative = [np.count_nonzero(u_samples < X)/len(u_samples) for X in u_samples]

D_u = np.max(np.abs(u_cumulative - norm.cdf(u_samples, loc = 0, scale = 1)))

p_value_u = kstwo.sf(D_u,len(u_samples))

print("D = {:.3f}, p-value = {}".format(D_u,p_value_u))
```

```
D = 0.211, p-value = 0.019511592180534776
```

For this empirical dataset, we would therefore reject the null hypothesis that the data is distributed according to the uniform distribution at the 5% significance level.

P-values for two-sample Kolmogorov-Smirnov tests

While in the one-sample case we can compare our value of D to the Kolmogorov-Smirnov distribution, in the two-sample case things are a bit more complicated and the K-S distribution is not a good approximation until we reach very large sample sizes. Instead, we must rely on a method called the *inside method*. This calculates the distribution of possible values of D through combinatorics, and is beyond the scope of this course. It is well-implemented in `scipy` as the backend for `ks_2samp`. We can calculate the p-values for comparing our two normally distributed samples and one of those samples with a uniform sample:

In [17]:
```
from scipy.stats import ks_2samp

print(ks_2samp(samples_normal_1, samples_normal_2))
print(ks_2samp(samples_normal_1, samples_uniform))
```

```
KstestResult(statistic=0.1290909090909091, pvalue=0.7151099547593596)
KstestResult(statistic=0.1933333333333333, pvalue=0.2280575658333467)
```

We can see from the p-values that the two normally distributed samples are much more likely to be from the same distribution.

However, at the 5% significance level, we see that both `samples_normal_2` and `samples_uniform` are both acceptable. The second case of a comparison with the uniform distribution is an example of a Type II error (false positive). I suspect if we had more data then we would see a statistically significant difference between the sample from the normal distribution and the sample from the uniform distribution.

We could also look at this from the point of view of formal hypothesis testing. If we said that our null hypothesis H_0 was that `samples_normal_1` came from a normal distribution and our alternate hypothesis H_a was that `samples_normal_1` came from a uniform distribution, then using that approach we would accept the normal distribution was the best distribution for `samples_normal_1`. In this case it would have made more sense to use the 1-sample KS test and compare `samples_normal_1` data to the exact distribution functions but there will be situations where the two distributions used to define the two hypotheses are not available exactly and are only know as some numerical sample of the distribution.

Summary

In this section, we have covered two different tests of goodness-of-fit, including:

- One and two sample Kolmogorov-Smirnov tests
- The chi-squared goodness-of-fit test
- How to calculate p-values from both K-S tests and chi-squared tests

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

Exercises ^

Exercise 1

We are going to work with three different probability distributions, with specific parameters. The three distributions are as follows:

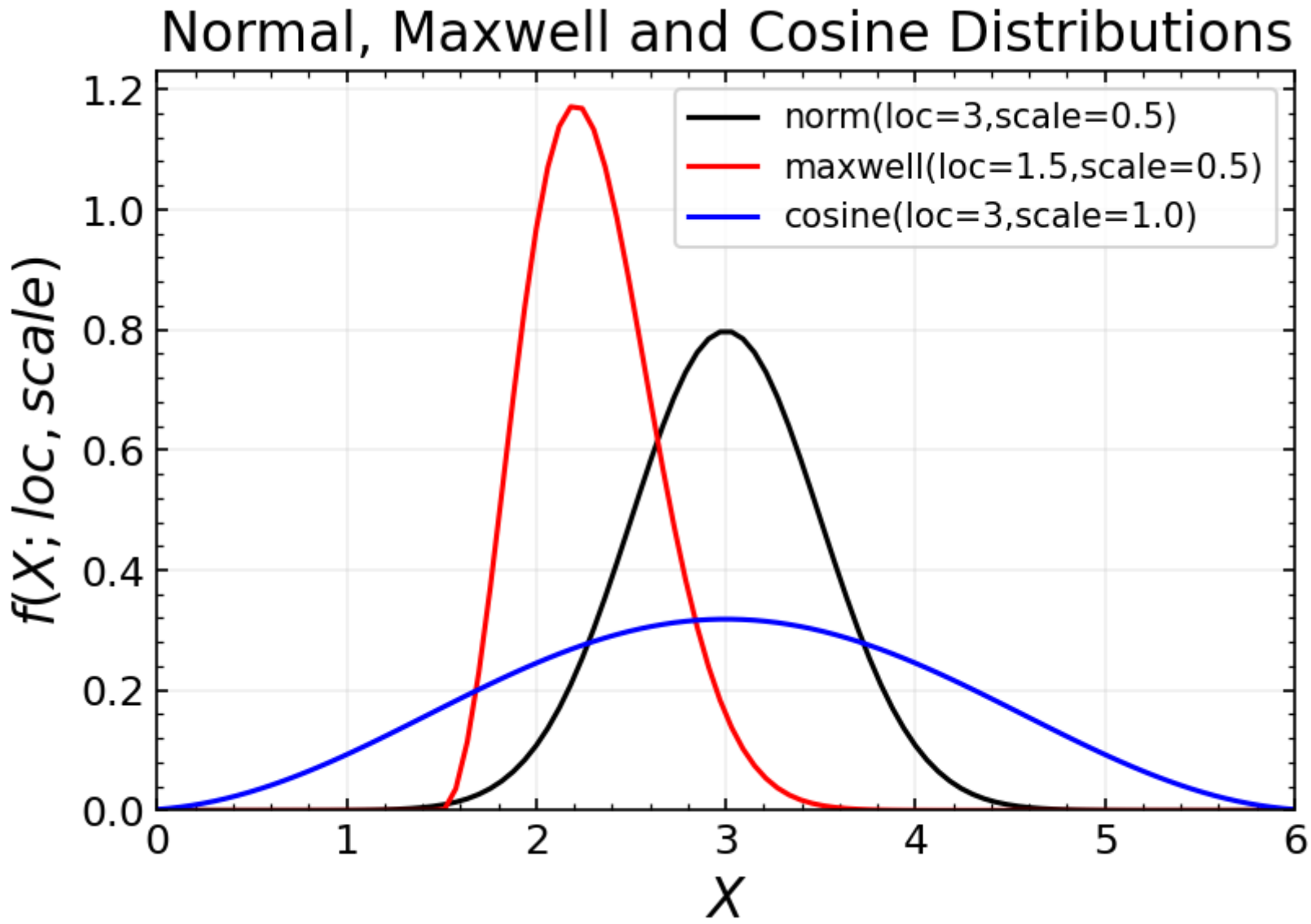
- `scipy.norm` with `loc = 3` and `scale = 0.5`
- `scipy.maxwell` with `loc = 1.5` and `scale = 0.5`
- `scipy.cosine` with `loc = 3` and `scale = 1`

Plot the PDFs of each of these distributions for the range $0 \leq X \leq 6$. Make sure your plots are clear and readable, including:

- Axis labels
- A legend labelling each curve (or display them in different subplots)
- Clear colours

In [18]:
```
from scipy.stats import norm, maxwell, cosine


# Your plotting code here

import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import AutoLocator,MultipleLocator
from matplotlib.colors import to_rgba
```

```
# 0 <= X <= 6 wanted
X_min=0.0
X_max=6.0
# Some X values to use when plotting
X_range = np.linspace(X_min, X_max, 100)

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 150)
ax.plot(X_range,    norm.pdf(X_range,loc=3.0,scale=0.5), color='black', label = "norm(loc=3,scale=0.5)")
ax.plot(X_range, maxwell.pdf(X_range,loc=1.5,scale=0.5), color='red',   label = "maxwell(loc=1.5,scale=0.5)")
ax.plot(X_range,  cosine.pdf(X_range,loc=3.0,scale=1.0), color='blue',  label = "cosine(loc=3,scale=1.0)")


ax.set_ylim(bottom = 0)
ax.set_xlim(X_min,X_max)

# set labels etc
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('$f(X; loc, scale)$',fontsize = 16)
ax.tick_params(direction='in',top=True,right=True,which='both',labelsize =12)
ax.xaxis.set_minor_locator(MultipleLocator(0.2))
ax.yaxis.set_minor_locator(MultipleLocator(0.04))
#ax.xaxis.set_minor_locator(AutoLocator())
#ax.yaxis.set_minor_locator(AutoLocator())
ax.legend(loc='upper right')
ax.set_title('Normal, Maxwell and Cosine Distributions',fontsize = 16)
ax.grid('xkcd:dark blue',alpha = 0.2)
```



Exercise 2

The file 'week03threedistributions500.npy' contains three data sets in a single numpy array of shape [3, 500]; each of these datasets corresponds to one of the probability distributions you plotted in the previous exercise.

1. Load the dataset in 'week03threedistributions500.npy'.
2. Using one-sample Kolmogorov-Smirnov tests, find the values of the Kolmogorov-Smirnov test statistic D and the corresponding p-values for each comparison i.e. for each data set and each distribution so we need 9 such comparisons.
3. Find which data set was generated by which theoretical distribution. Why are they not perfect matches? Write your answer in Markdown in a text cell.

Optional Exercise 2b. Long and tedious so please skip this. You should always really take a quick look at data so plot a histogram for each of the three data samples and add the theoretical distributions for comparison.

In [19]:
```
# Load the data.
# The text and the .npy extension tells us these are numpy arrays
# so we use np.load as follows
sample_1, sample_2, sample_3 = np.load('week03threedistributions500.npy')

# quick check to see if we read these in properly
print(len(sample_1),len(sample_1),len(sample_3))
```

```
500 500 500
```

In [20]:
```
from scipy.stats import ks_1samp

# For example see
# https://stackoverflow.com/questions/37575270/what-arguments-to-use-while-doing-a-ks-test-in-python-with-studen
# to see how to set arguments of cdf function when passing to the ks_1samp method.

"""
It is much simpler just to write the six lines for one sample then to copy these lines,
change the sample used for the next two samples. No need to get the computer to find the best fit.
With just 9 examples, a human can easily pick out the answer.

Of course, if we had many more samples or distributions, there are ways to use automate this
so that just one line in the centre of some loops is doing the KS fit but it is probably not
worth doing that here. You would lose time.
"""

print("\n+++ Testing sample 1:-")
d1_D, d1_p = ks_1samp(sample_1, norm.cdf,    (3.0,0.5) )
print("  Compared to norm(loc=3,scale=0.5) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_p))
d1_D, d1_p = ks_1samp(sample_1, maxwell.cdf, (1.5,0.5) )
print("  Compared to maxwell(loc=1.5,scale=0.5) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_
d1_D, d1_p = ks_1samp(sample_1, cosine.cdf,  (3.0,1.0) )
```

```
print("  Compared to cosine(loc=3,scale=1.0) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_p))

print("\n+++ Testing sample 2")
d1_D, d1_p = ks_1samp(sample_2, norm.cdf,    (3.0,0.5) )
print("Compared to norm(loc=3,scale=0.5) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_p))
d1_D, d1_p = ks_1samp(sample_2, maxwell.cdf, (1.5,0.5) )
print("Compared to maxwell(loc=1.5,scale=0.5) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_p)
d1_D, d1_p = ks_1samp(sample_2, cosine.cdf,  (3.0,1.0) )
print("Compared to cosine(loc=3,scale=1.0) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_p))

print("\n+++ Testing sample 3")
d1_D, d1_p = ks_1samp(sample_3, norm.cdf,    (3.0,0.5) )
print("  Compared to norm(loc=3,scale=0.5) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_p))
d1_D, d1_p = ks_1samp(sample_3, maxwell.cdf, (1.5,0.5) )
print("  Compared to maxwell(loc=1.5,scale=0.5) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_
d1_D, d1_p = ks_1samp(sample_3, cosine.cdf,  (3.0,1.0) )
print("  Compared to cosine(loc=3,scale=1.0) distribution:  D = {:.3f} and p-value = {:5.3f}".format(d1_D,d1_p))
```

```
+++ Testing sample 1:-
  Compared to norm(loc=3,scale=0.5) distribution:  D = 0.629 and p-value = 0.000
  Compared to maxwell(loc=1.5,scale=0.5) distribution:  D = 0.054 and p-value = 0.104
  Compared to cosine(loc=3,scale=1.0) distribution:  D = 0.504 and p-value = 0.000

+++ Testing sample 2
Compared to norm(loc=3,scale=0.5) distribution:  D = 0.207 and p-value = 0.000
Compared to maxwell(loc=1.5,scale=0.5) distribution:  D = 0.478 and p-value = 0.000
Compared to cosine(loc=3,scale=1.0) distribution:  D = 0.035 and p-value = 0.543

+++ Testing sample 3
  Compared to norm(loc=3,scale=0.5) distribution:  D = 0.032 and p-value = 0.663
  Compared to maxwell(loc=1.5,scale=0.5) distribution:  D = 0.628 and p-value = 0.000
  Compared to cosine(loc=3,scale=1.0) distribution:  D = 0.206 and p-value = 0.000
```

Interpretation of Exercise 2 results

Let us work at the 5% acceptance level.

Sample 1 is best fit by the Maxwell distribution, maxwell(loc=1.5,scale=0.5) but not by much. That is there is just 10% chance that this KS test D-value or worse (larger) would come from 500 points drawn from this Maxwell distribution.

Sample 2 is best fit by the cosine distribution, cosine(loc=3,scale=1). This is a better fit tha we had with sample 1 as you can see in the smaller D value. In this case there is a 54% chance that this KS test D-value or worse (larger) would come from 500

points drawn from this cosine distribution.

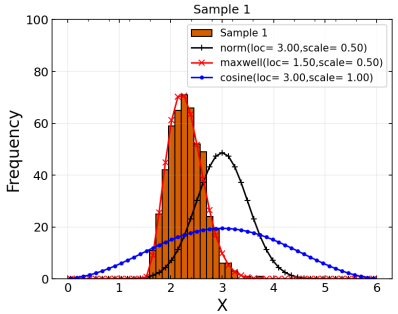
Sample 3 gives the best fit all of three data sets, in this case to the normal distribution norm(loc=3,scale=0.5). Note it isn't a perfect fit as the random noise means it should never be perfect. The D value here is the smallest of all 9 comparisons and there is a 66% chance of finding this level of fit or worse (larger D) from this normal distribution.

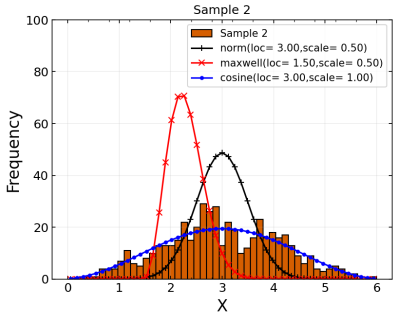
Additional Information on Exercise 2

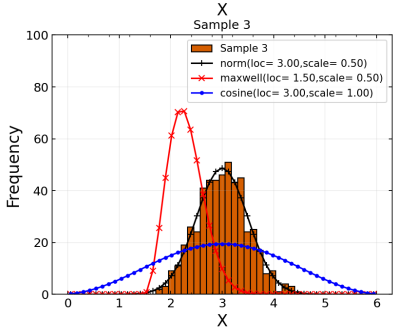
In fact you should probably always take a look at distributions when analysing them. When I first tried this exercise the data file was incorrect and the data barely matched the suggested theoretical distributions according to the KS test. To find out what was happening I started with some basic statistics (mean, standard deviation for instance) and quickly found the data didn't match the distributions. Note that the `loc` and `scale` are not always the mean and standard deviation for a distribution but there are methods that give these from the built-in functions e.g. `norm(loc=3.0,scale=0.5).mean`. I went ahead and (while not difficult, it took some time(I found the plots of histograms of the data and theoretical versions did not match.

The plot below shows the current state of the data in the file against the theoretical distributions I suggested. They now do show a good match.

Note: the lines in the theoretical distributions should *not* be there unless you tell me these are "interpolations of the data points for visual guidance only". These interpolations between points are made by the plot routine and are not controlled by me and it is unclear exactly what the value of the interpolated line represents - at best they give some approximation of the pdf. The data points correctly show a histogram of bin values as this is what we have from the data. The data points from the theoretical distributions do *not* and should not give the continuous PDF at the centre of the bin (though that might be a fair approximation). Do not put scientifically meaningless lines between data points without at least careful note as I have done here.







Exercise 3

The files 'two_sample_test_to_match.npy' and 'two_sample_test_options.npy' contain samples from several normal distributions, with different μ and σ. Your task is to to see which of the three test samples in 'two_sample_test_options.npy' is most likely to be from the same distribution as the target sample in 'two_sample_test_to_match.npy'. To do this, you should:

- Load the data
- Define the empirical cumulative functions for each test sample, and plot them against the target sample
- Use `ks_2samp` to calculate the Kolmogorov-Smirnov test statistic and the p-value for each sample

- Finally, write your answer as to which test sample is most probable to match the target sample in the Markdown cell after the code cells

In [21]:
```
# Load the data, you can read it into a tuple of three, no options needed

target_sample =np.load('two_sample_test_to_match.npy')

sample_1, sample_2, sample_3 = np.load('two_sample_test_options.npy')

# quick check using shape (dimensions of numpy matrix) to see if we read these in properly
print(target_sample.shape)
print(sample_1.shape,sample_2.shape,sample_3.shape)
```

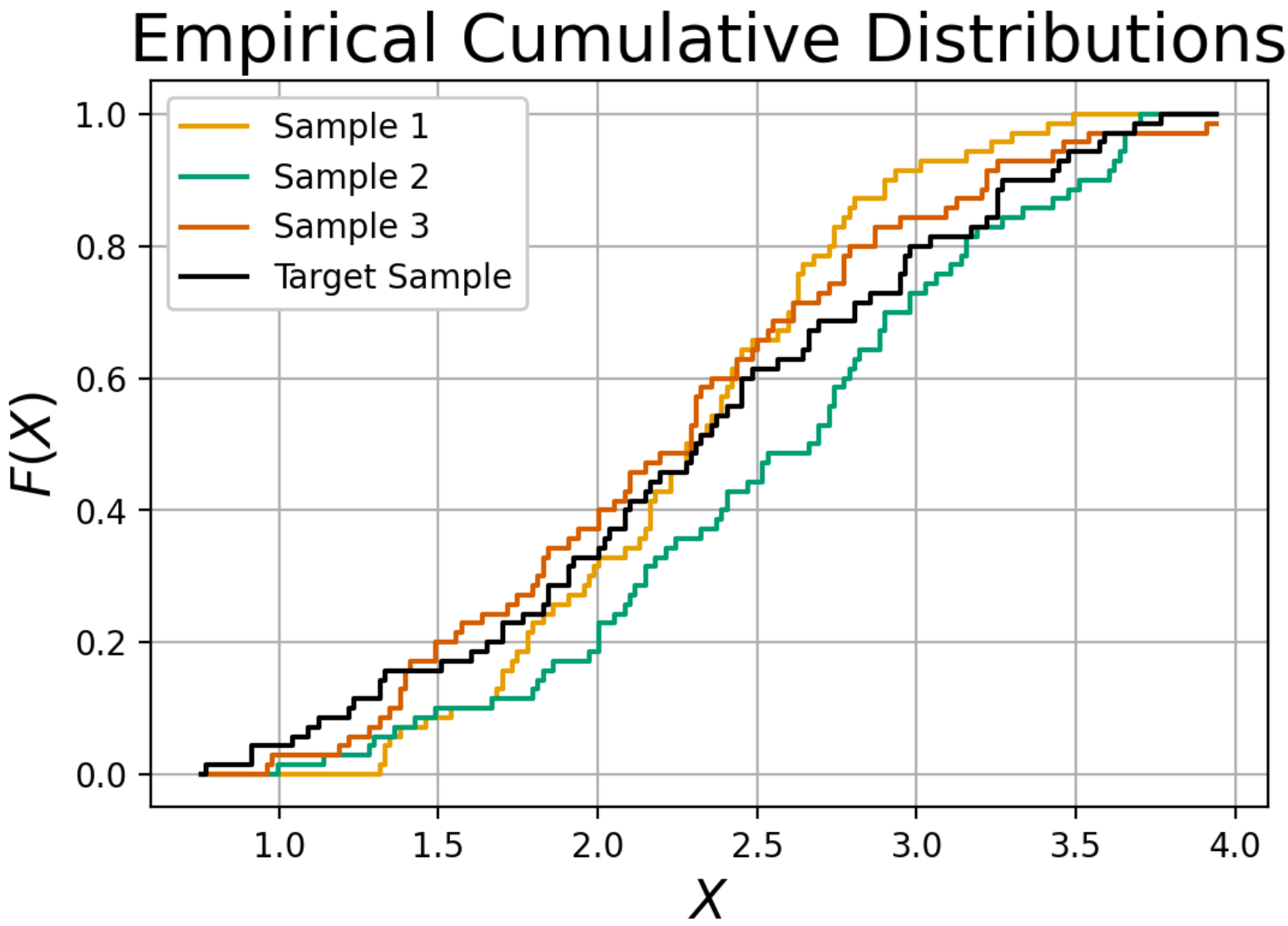
```
(70,)
(70,) (70,) (70,)
```

In [22]:
```
# Define empirical cumulative functions.
# Crude way but effective, you don't need to make it fancy for four distributions
# Note that these methods use global variables sample_1 etc which is poor practice
# but for a simple quick solution it works here.

def F1(X):
    """ This gives the empirical cumulative distribution of the data in "sample_1" for given X value """
    return np.count_nonzero(sample_1<X)/len(sample_1)
def F2(X):
    """ This gives the empirical cumulative distribution of the data in "sample_2" for given X value """
    return np.count_nonzero(sample_2<X)/len(sample_2)
def F3(X):
    """ This gives the empirical cumulative distribution of the data in "sample_3" for given X value """
    return np.count_nonzero(sample_3<X)/len(sample_3)
def Ft(X):
    """ This gives the empirical cumulative distribution of the data in "target_sample" for given X value """
    return np.count_nonzero(target_sample<X)/len(target_sample)
```

In [23]:
```
# This code cell plots empirical cumulative distribution from our data with the theoretical CDF
import matplotlib.pyplot as plt

Xmin=min(target_sample)
Xmax=max(target_sample)
# Values of X to use here when plotting
for sample in [sample_1,sample_2,sample_3]:
    Xmin=min(min(sample),Xmin)
    Xmax=max(max(sample),Xmax)
```

```
Xvalues = np.linspace(Xmin, Xmax, 200)

CDF1 = [F1(X) for X in Xvalues]
CDF2 = [F2(X) for X in Xvalues]
CDF3 = [F3(X) for X in Xvalues]
CDFt = [Ft(X) for X in Xvalues]

# Now plot using some colour blind friendly colours https://zenodo.org/records/3381072
# From Color Universal Design (CUD): https://jfly.uni-koeln.de/color/
# Okabe_Ito = ["#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC79A7", "#000000"]

fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 200)
ax.step(Xvalues, CDF1, color="#E69F00", label='Sample 1', where='post')
ax.step(Xvalues, CDF2, color="#009E73", label='Sample 2', where='post')
ax.step(Xvalues, CDF3, color='#D55E00', label='Sample 3', where='post')
ax.step(Xvalues, CDFt, color='#000000', label='Target Sample', where='post')
ax.legend(loc='upper left',framealpha = 1)
ax.set_xlabel('$X$',fontsize = 16)
ax.set_ylabel('$F(X)$',fontsize =16)
ax.grid()
ax.set_title('Empirical Cumulative Distributions',fontsize = 20)
```

Out[23]:
```
Text(0.5, 1.0, 'Empirical Cumulative Distributions')
```



Looking at this plot it seems that for most X values the red curve from sample 3 is either the closest curve above, or the closest below the black curve of the target_sample.

In [24]:
```
# Run Kolmogorov-Smirnov tests

D1, p1 =  ks_2samp(target_sample, sample_1)
```

```
print("  target to sample 1:  D = {:.3f} and p-value = {:5.3f}".format(D1,p1))

D2, p2 =  ks_2samp(target_sample, sample_2)
print("  target to sample 2:  D = {:.3f} and p-value = {:5.3f}".format(D2,p2))

D3, p3 =  ks_2samp(target_sample, sample_3)
print("  target to sample 3:  D = {:.3f} and p-value = {:5.3f}".format(D3,p3))
```

```
  target to sample 1:  D = 0.186 and p-value = 0.179
  target to sample 2:  D = 0.171 and p-value = 0.256
  target to sample 3:  D = 0.114 and p-value = 0.754
```

The best fit to the target data is clearly sample 3 as the lowest D value and highest p value show. This is what we observed above from the CDF plot.

However you have to be careful. There is still an 18% chance that if we had random samples from the undelrying distributions of the target sample and sample 1 (the worst fitting curve) that we would still get a worse (higher) D value. That is about once out of five trials, pure random chance would give us this D value for sample 1 and the target value even if they were identical.

Exercise 4

We are now going to return to the simple pendulum data you plotted in Week 1. As a reminder, this data should obey the relation between pendulum length and period, given as

$$T = 2\pi\sqrt{\frac{L}{g}}$$

where L is the pendulum length in m, T is the pendulum time period in s, and g is the acceleration due to gravity, in ms^{-2}.

Using the value of g provided in the file, perform a χ^2 goodness of fit test for this data. What is the value of the chi-squared and the reduced chi-squared? What is the p-value for this model/parameter value?

In [25]:
```
# Load the data; remember you need to use the pickle option as the data is in a dictionary

pendulum_data = np.load('pendulum_data.npy',allow_pickle = True).item()
```

```
# simple check we read it in
print(pendulum_data.keys())
```

```
dict_keys(['length', 'period', 'g', 'sig_g', 'period_err', 'length_err'])
```

In [26]:
```
print( len(pendulum_data['length']),
       len(pendulum_data['period']),
       (pendulum_data['period_err']),
       (pendulum_data['length_err'])
     )
```

```
21 21 0.05 0.1
```

NOTE a routine for chi-square should be built into `scipy` but `scipy.stats.chisquare` is for only Pearson chi-square statistics. So make your own.

In [27]:
```
# Make your own chi square routine

def chisquare(yvalues,fvalues,svalues):
    """ Find the general chi-square statistic
    \chi^2 = \sum_i [ (y_i-f_i)/ s_i ]^2

    Note no checks. All lists must be of equal length and with well defined values.
    Slow but good enough here. Can write a faster vectorised form using numpy arrays.

    Input
    -----
    yvalues - list of yvalues
    fvalues - list of expected values
    yvalues - list of standard deviations (expected errors)

    Return
    ------
    tuple of
    chi-square statistic
    and a list of pull squared values.

    """

    chisqvalues = [ (yvalues[i]-fvalues[i])*(yvalues[i]-fvalues[i])/(svalues[i]*svalues[i]) for i in range(le
    return sum(chisqvalues), chisqvalues
```

I will choose to look at the function

$$y(T/L^2) = \frac{4\pi^2 L^2}{T}\,, \Rightarrow\, y(X) = \frac{4\pi^2 L^2}{T}, \quad X = \frac{L^2}{T}$$

as looking for a null result is best and looking for a constant is the next best thing. We expect $y(X) = g$ for all values of L and T.

The uncertainty in $y(X)$ values have to be found by error propagation from the T and L uncertainties which we are given so $$

(\sigma_y)^2

\left(\frac{\partial y}{\partial T} \right)^2 (\sigma_T)^2

- \left(\frac{\partial y}{\partial L} \right)^2 (\sigma_L)^2 \, .

$$Then we find that$$

(\sigma_y)^2

\left(-\frac{4 \pi^2 L^2}{T^2} \right)^2 (\sigma_T)^2

- \left(\frac{8 \pi^2 L}{T} \right)^2 (\sigma_L)^2 \, .

$$Rearranging for numerical purposes we can write this as$$

(\sigma_y)^2

y^2 \left(- \left(\frac{ \sigma_T}{T} \right)^2
```
   + \left( \frac{2 \sigma_L}{L} \right)^2
   \right)
```
\, .

$$So then$$

\sigma_y

y(L) \left(\left(\frac{ \sigma_T}{T} \right)^2
+ \left(\frac{2 \sigma_L}{L} \right)^2
\right)^{1/2}
\ . $$

In [28]:
```
length_values = pendulum_data['length']/100
print()
```

In [29]:
```
# Calculate data for the chi-squared metric

number_values = len(pendulum_data['length'])

# WORK IN SI UNITS TO MAKES THING CONSISTENT
# SO MUST CONVERT LENGTH TO SI UNITS!
length_values = pendulum_data['length']/100
sigma_L = pendulum_data['length_err']/100


y_values = [ length_values[i] * ( 2 * np.pi / pendulum_data['period'][i] )**2
             for i in range(number_values)
           ]

"""y_sigma_values =[   ( 2 * np.pi / length_values[i] )**2
                * np.sqrt(  (pendulum_data['period_err'])**2
                          + (2 * pendulum_data['period'][i] * sigma_L / length_values[i] )**2
                         )
             for i in range(number_values)
           ]
"""

y_sigma_values =[   y_values[i]
                * np.sqrt(  (pendulum_data['period_err'] / pendulum_data['period'][i] )**2
                          + (2*sigma_L / length_values[i] )**2
                         )
             for i in range(number_values)
           ]

# does no harm to check the data looks OK
```

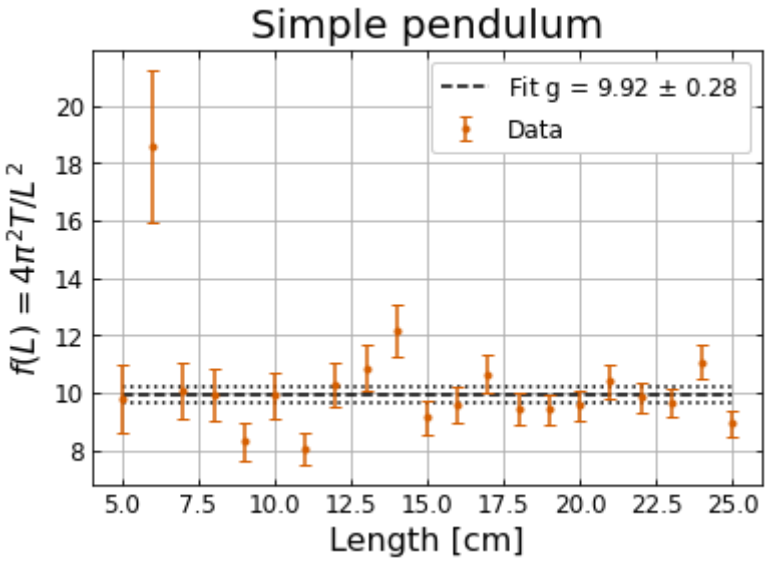
```
print(length_values)
print(sigma_L)
print(y_values)
print(y_sigma_values)
```

```
[0.05 0.06 0.07 0.08 0.09 0.1  0.11 0.12 0.13 0.14 0.15 0.16 0.17 0.18
 0.19 0.2  0.21 0.22 0.23 0.24 0.25]
0.001
[9.78234716989105, 18.588688799168633, 10.055945466344664, 9.944412282700906, 8.283536740365765, 9.8939807062
8444, 8.01305607189093, 10.286132279778828, 10.838604556995403, 12.145683191910695, 9.125662279437384, 9.5829
59368912853, 10.65108788916072, 9.439950499116742, 9.406569047499605, 9.54306366981664, 10.391884767254545,
9.826738415191905, 9.644773287855179, 11.066032388014497, 8.913038917205728]
[1.1570282764567112, 2.6763937243999476, 1.0012344923513252, 0.9166523739320908, 0.6586471154830339, 0.807765
6035761028, 0.5634040691187038, 0.776989057877519, 0.8050098804441445, 0.9168099253793148, 0.579344681142798
9, 0.6022072227185306, 0.6824999783161662, 0.5540312188914572, 0.5359227423416457, 0.5331838573974216, 0.5900
895827921386, 0.530208315696827, 0.5040354346821632, 0.6050294192510889, 0.42946598523608875]
```

In [30]:

```
# OPTIONAL even better to plot the data.
# As this is an exercise not a scientific study, only do this if you have time.

fig = plt.figure(figsize = (6,4)) # set a figure size similar to the default
ax = fig.add_subplot(111)

# If you want a grid, make sure you add this first.
ax.grid() # Make sure we add the grid before our other plots, otherwise it will be displayed on top of other

ax.plot( pendulum_data['length'], [pendulum_data['g'] ] * number_values, "k--",
         label = 'Fit g = {:.2f} $\pm$ {:.2f} '.format(pendulum_data['g'], pendulum_data['sig_g']),
         )
# Simple way to show uncertainty in $g$ from fit
ax.plot( pendulum_data['length'], [pendulum_data['g'] + pendulum_data['sig_g'] ] * number_values, "k:",
         label = None
         )
ax.plot( pendulum_data['length'], [pendulum_data['g'] - pendulum_data['sig_g']] * number_values, "k:",
        label=None)

# plot data on TOP of errors
ax.errorbar(pendulum_data['length'], y_values, yerr = y_sigma_values,
            fmt = '.', label='Data', color='#D55E00',capsize = 3)



ax.set_xlabel('Length [cm]',fontsize = 16)
ax.set_ylabel(r'$f(L) = 4\pi^2 T / L^2 $',fontsize= 16)
```

```
ax.set_title('Simple pendulum',fontsize= 20)

handles, labels = ax.get_legend_handles_labels()

ax.legend(loc='upper right',fontsize = 12, framealpha = 1) # Framealpha prevents grid lines from showing thro

ax.tick_params(axis='both',labelsize = 12, direction='in',top = True, right = True, which='both')

# These next three lines make more sophisticated ticks on the axes
# I would normally put imports at the start but I place this import just before I use the imported methods.
from matplotlib.ticker import MultipleLocator # makes placing ticks easier; can put major/minor ticks at fixe

#ax.xaxis.set_minor_locator(MultipleLocator(0.5)) # tells it to place the small ticks every 0.5 on the horizo
#ax.yaxis.set_minor_locator(MultipleLocator(0.02)) # tells it to place the small ticks every 0.02 on the vert
```



In [31]:
```
# At last find the chi-square values

# NOTE we do NOT use the error in g provided as that comes from a numerical fit, not an experimental error
chisqvalue, chisqvalue_list = chisquare(y_values, [pendulum_data['g']] * number_values, y_sigma_values)
print(chisqvalue_list)
print("\n *** The chi-square value for this data is {:5.2f}".format(chisqvalue))
print("\n *** The number of data points is {:d} and here the number of degrees of freedom is one less".format
print("       So the reduced chi-square value for this data is {:5.2f}".format(chisqvalue/(number_values-1)))
```

```
[0.013382807945006922, 10.499963684023564, 0.01948152427941834, 0.00094746166633093, 6.144492488632341, 0.000
7564317476462717, 11.410427277711536, 0.22668451892473349, 1.3129353738205782, 5.913586975819388, 1.861948962
0554766, 0.30620760534713787, 1.1594193013266936, 0.7389158669686994, 0.9042780449359027, 0.4897494304702987,
0.6498426326321061, 0.02846756315299587, 0.28998344670020365, 3.6117560544279628, 5.456085254454431]

 *** The chi-square value for this data is 51.04

 *** The number of data points is 21 and here the number of degrees of freedom is one less
       So the reduced chi-square value for this data is  2.55
```

Looking at the pull squared from each measurement, each value for L, we see that the second point which looks way off the expected results is bad but does not give the worst fit to the data. That comes from the seventh point at L=11cm. This is because the uncertainty in the second point is large anyway.

We have a subtlety. The value of g, the acceleration due to gravity on Earth, provided comes from a fit to the same experimental data that we have been given (see comments in week 1). So our value for the expected value of y, the f_i at each data point in our chi-square formula, has already used the data to fix this number. This value of g is clearly going to be a better fit to our data than some number given to us from elsewhere because we have used our own data, good or bad, to get the closest fit to our data. Essentially we have used up one degree of freedom to choose the expected value $f_i = f(L)$ that we used in our chi-square. So the number of degrees of freedom is one less than the number of measurements.

In practice, here with twenty or so points, subtracting one from the number of degree of freedom you use in your formula is not going to make any difference to our conclusions.

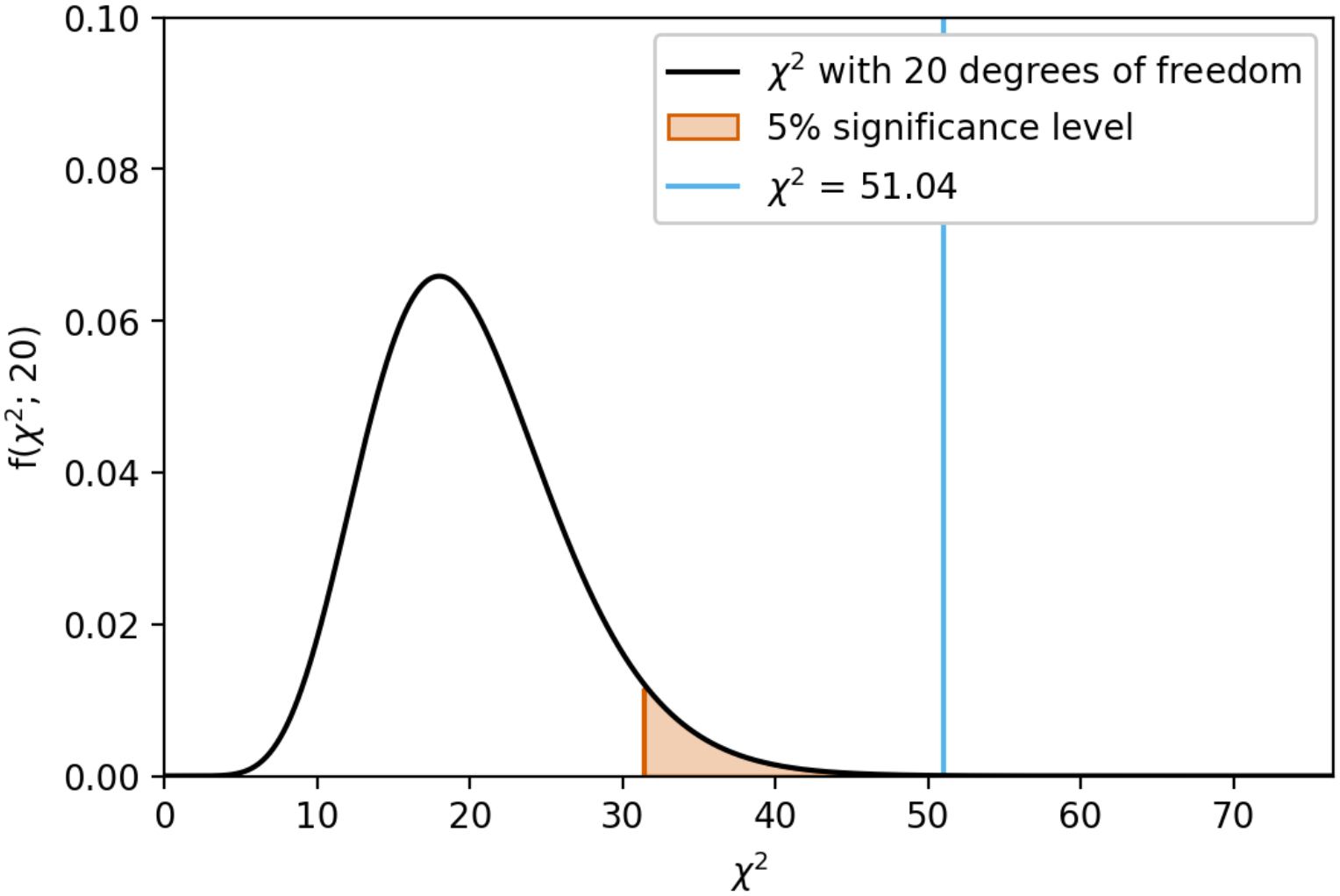
The bottom line with a reduced chi-square of close to three, this data is nearly three sigma away from what the best fit result gets. That suggest there is a large variation in the results and so very unlikely to be found by the random fluctuations in an excellent experiment. Lets be more precise and find the p-value.

In [32]:
```
# Calculate p-value

from scipy.stats import chi2 # This is the chi-square distribution, not the chi-square statistic

p_value = chi2.pdf(chisqvalue, (number_values-1))

print("\n *** The chi-square value for this data is {:5.2f}".format(chisqvalue))
print("       The number of data points is {:d} and here the number of degrees of freedom is one less".format
print("       The reduced chi-square value for this data is {:5.2f}".format(chisqvalue/(number_values-1)))
```

```
print("       The p-value for this data is {:5.3f}".format(p_value) )

# I did not ask for this but we can update the plot we used earlier
# to illustrate chi square distribution for this case.
from matplotlib.colors import to_rgba

ndof = number_values-1
x_max = chisqvalue*1.5

X = np.linspace(0,x_max,1000)
fig, ax = plt.subplots(1,1,figsize = (6,4),dpi = 200)
ax.plot(X, chi2.pdf(X, ndof), label='$\chi^2$ with {:d} degrees of freedom'.format(ndof),
                              color='black',zorder = 1)
ax.plot([chi2.ppf(0.95,ndof),chi2.ppf(0.95,ndof)],[0,chi2.pdf(chi2.ppf(0.95,ndof),ndof)-0.001],zorder = 0,col
ax.fill_between( np.linspace(chi2.ppf(0.95,ndof), x_max, 1000),
                 np.repeat(0,1000),
                 chi2.pdf( np.linspace(chi2.ppf(0.95,ndof),x_max,1000),ndof),
                 fc=to_rgba('#D55E00',0.3),
                label='5% significance level', edgecolor='#D55E00', zorder = 0)
ax.plot([chisqvalue, chisqvalue], [0,0.1],
        color='#56B4E9', zorder = 0,
        label=r'$\chi^2$ = {:4.2f}'.format(chisqvalue))

ax.set_ylim(bottom = 0, top = 0.1)
ax.set_xlim(0,x_max)
ax.legend(loc='upper right',framealpha = 1)
ax.set_xlabel('$\chi^2$')
ax.set_ylabel('f($\chi^2$; {:d})'.format(ndof))
```

```
 *** The chi-square value for this data is 51.04
       The number of data points is 21 and here the number of degrees of freedom is one less
       The reduced chi-square value for this data is  2.55
       The p-value for this data is 0.000
```

Out[32]:
```
Text(0, 0.5, 'f($\\chi^2$; 20)')
```



On the surface, these experimental results look good, the best fit gives a reasonable value for g. However, what the statistics is showing is that the variation in our experimental results is too large to have come from Gaussian statistics. This is worrying. We probably need to look at how we estimated our uncertainties. Perhaps one or two of the measurements at certain L values were flawed, e.g. these outliers were the results from experiments the professor did when they should have left it to their PhD student?

In [ ]:

Exercise 5

This exercise concerns a particle physics experiment looking to find some new particle. Some data gathered by this experiment can be found in the file "particle_data.npy". This data is in a dictionary, so you must use `allow_pickle = True` to load the data in as you did with the pendulum data used in an exercise above. The data you get is a dictionary with two keys, `values` and `bins`. The values give the counts in each bin, the bins are the edges of the bins in GeV.

The data is the count of detected particles in different energy bins, along with the specific energy bins. The density of the *number* of counts expected from background events in this data is modelled by the following function:

$$f(X) = A\exp(-X/B) + C$$

where X is the energy in units of GeV and the coefficients A, B, and C take the values

$$A = 250$$
$$B = 400$$
$$C = 50$$

Note: with these values you get an unnormalised PDF fX but this gives exactly the correct expected *number density* of events. You can just integrate this over a range of energy values to find the number of expected events in that region.

1. First, find the number n_b of particles in bin b which has values of energy X that lie in $X_b \leq X < X_{b+1}$. Show your working in a markdown cell.
2. Next, read in the data from "particle_data.npy" and get the counts in each bin and the bin edges stored in the dictionary.
3. Now find the expected number of background events in each bin. The bins are equally spaced.
    A. Note that we may need to rescale our predicted data to make sure that the total of counts predicted matches that in the data exactly. This is noted in the code box below.
4. Visualise the data using `plt.stairs` and show the expected background values.
5. For each bin, what is the distribution of the values expected in that bin? For instance suppose we could repeat the same experiment ten time, how would the ten value for one bin be distributed? (This is a hypothetical question as you would just combine all the data into one data set).
6. Using a 5% significance level, does the data with the hypothesis of background-only model? Calculate the value of the chi-square statistic of the data relative to the background model to do this hypothesis test. Write your final answer in

the Markdown cell below.

Remember the following steps:

- Evaluate the background-only model at the center of each energy bin
- Assume each data bin is distributed according to a Poisson distribution
- Calculate the number of degrees of freedom of the background-only model

The number n_b of particles in bin b is given by $$

n_b

\int{X{b}}^{X_{b+1}} dZ \, (A \exp (-Z/B) + C)

[-A B \exp (-Z/B) + C \, Z]{X{b}}^{X{b+1}} $$ where bin b has energy values X that lie in $X{b} \leq X < X_{b+1}$. $Thisisequalton_b = AB\left(\exp(-X_b/B) - \exp(-X_{b+1}B)\right) + C\left(X_{b+1} - X_b\right)$.$

In practice, it is common to approximate the number n_b of particles in bin b as

$$n_b \approx (X_{b+1} - X_b) f((X_{b+1} + X_b)/2) \, .$$

That is we multiply the bin width by the PDF (probability *density* function) evaluated at the centre of the bin. That will work well enough here but you should understand that this is an approximation. EFS: can you see why this approximation often works? EFS: Can you derive this approximation? (i.e. what is the first small correction?)

In [33]:
```
# Load the file "particle_data.npy" using "allow_pickle = True" AND to get a dictionary

particle_data = np.load('particle_data.npy',allow_pickle = True).item()

# Check keys of this dictionary
print("Dictionary keys are ",particle_data.keys())

# Get the bin edges in this dictionary
bin_edges = particle_data['bins']
print("Found "+str(len(bin_edges))+" bin edges")
```

```
# Get the counts in each bin and check there is one less entries than the number of bin edges
count_values  = particle_data['values']
print("Found "+str(len(count_values))+" counts")
```

```
Dictionary keys are  dict_keys(['bins', 'values'])
Found 251 bin edges
Found 250 counts
```

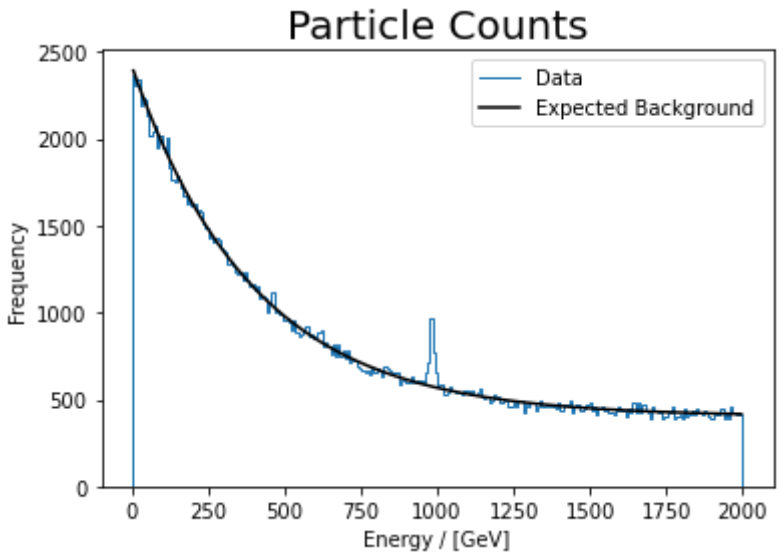
In [34]:
```
# Now find the expected number of background events in each bin.
def bin_prob_unnorm (Xmin,Xmax,A,B,C):
    return C*(Xmax-Xmin) + A*B*(np.exp(-Xmin/B) - np.exp(-Xmax/B))

# Don't forget that while this f(x) is not normalised,
# we are told these values give us the actual numbers expected in each bin.
A = 250
B = 400
C = 50

# Assume bins are equally spaced.
Xfirst= bin_edges[0]
Xlast= bin_edges[-1]
expected_bkg_values = [bin_prob_unnorm(bin_edges[b-1],bin_edges[b],A,B,C)
                       for b in range(1,len(bin_edges)) ]
```

In [35]:
```
"""
*** PROBLEM ***
The scipy.stats.chisquare we use below must have an almost perfect match
between the total count in the data and the total count in the expected data.
Various issues such as the stochastic nature of the way created the artifical data
means the parameters we used to create the data for this exercise do not give
the same number of particles in the artifical data each time we create it.
So we need to rescale the data by a tiny amount.
"""

# Optional: as a check lets see if the total number in the expected counts and actual counts match up
total_expected_bkg_count = sum(expected_bkg_values)
total_count = np.sum(count_values) # numbers of particles actually counted in the data
count_correction = total_expected_bkg_count/total_count
print("Total number in expected count / actual number of particles = {:5.3f}".format(count_correction))
# The match isn't perfect but its pretty close.

expected_bkg_values = [ c/count_correction for c in expected_bkg_values ]
```

```
Total number in expected count / actual number of particles = 0.995
```

In [36]:
```
#Visualise the data using plt.stairs and show the expected background values.

# We will plot the expected values at the centre of the bin
bin_centres = [ (bin_edges[b]+bin_edges[b-1])/2 for b in range(1,len(bin_edges)) ]

fig, ax = plt.subplots(1,1,figsize = (6,4))

# With only one plot, no subscript on ax nedded, so don't use ax[0] here
ax.stairs(count_values, bin_edges, label='Data')

# Unfortunately there were too many points so data points didn't work well.
# So using meaningless lines connecting the points of the expected counts is the best approach here.
ax.plot(bin_centres, expected_bkg_values, color="#000000" , label='Expected Background')
ax.legend()
ax.set_title('Particle Counts',fontsize = 20)
ax.set_xlabel('Energy / [GeV]')
ax.set_ylabel('Frequency')
```

Out[36]:
```
Text(0, 0.5, 'Frequency')
```



In [37]:
```
# OPTIONAL ADDITIONAL INFORMATION.

"""
The simple analytical form for the background distribution
```

```
means we can do a quick check on the values we were given.
More useful for those setting the question than answering it.

First assume that any interesting signal is not at the
extreme energy values (as we can see in the plot). So the first
and last bins are counts from pure background process.
The count predicted at $X=0$ which is in the first bin is
f(X=0) = A+C
Since B is much less than the largest X value (Xmax=2000)
then we can approximate that
f(X=Xmax) = C
These are densities but the bin width (call it W) is constant so we predict that
the first bin contains W.(A+C) and the last bin contains W.C.
"""

bin_width = bin_edges[1]- bin_edges[0]
print("Bin width {:6.2f}".format(bin_width))

C_est = count_values[-1] / bin_width
print("C estimate {:6.2f}, C value used {:6.2f}".format(C_est,C))

A_est = (count_values[0] - count_values[-1] ) / bin_width
print("A estimate {:6.2f}, A value used {:6.2f}".format(A_est,A))

# You should see these match up pretty well to the values we were given.
```

```
Bin width   8.00
C estimate  52.00, C value used  50.00
A estimate 243.25, A value used 250.00
```

OPTIONAL

This is first to remind you that counts in each bin should be thought of as random numbers drawn from a Binomial distribution.

That is if we find a particle at energy E in then there is a probability p_B of finding it in bin p. That means with N total particles measures (N trials) we have a Binomial distribution for the probability $f(b)$ of finding n_b out of the N particles in bin b where

$$f(b) = \binom{N}{n_b} (p_b)^{n_b} (1-p_b)^{n_b - k} .$$

The expected number of particles n_b in bin b is then $p_b N$ and the variance is $(\sigma_b)^2 = p_b N(1-p_b)$ For small p_b the variance is equal to the mean which is case you have when the Binomial distribution approximates the Poisson distribution. With many bins here, this will approximation works for us here.

This leads me to my warning that you must be *VERY CAREFUL* with chi square tests. The form given in scipy.stats.chisquare is for the Pearson chi-square test and is not the most general. In week 4 we will use the full form general via another package. The Pearson test assumes Poisson distributions for the y values so that the variance of any y value is assumed to be given by the mean for that bin, and we estimate the mean value to be equal to the value we measure (it is our best estimate). However in our case, this is the situation we have, the counts in each bin are the y values and we estimate the "error" (uncertainty, one standard deviation) to be $\sqrt{y}$. That is all assumed in `scipy.stats.chisquare`.

In [38]:
```
# AT LAST
# Now let us find chi square using the scipy version as we do want the Pearson chisquare test.
from scipy.stats import chisquare

chi2_result = chisquare(count_values, expected_bkg_values)
reduced_chi2_value = chi2_result.statistic/len(bin_edges)
print ("reduced chi square value {:5.3f}, p-value {:5.3f}, from {:d} values".format(reduced_chi2_value,chi2_
```

```
reduced chi square value 2.459, p-value 0.000, from 250 values
```

The background distribution given is not a good fit to the data at the 5% level.

So it appears that while not a terrible fit (we can see a good fit in the plot for most energy values), the fit is *not good enough*. The probability that the data could be produced from the background distribution given to us is less than 5% ($p<0.05$) which was our significance level.

Probably, that small peak in the data around 1000 GeV is causing this small p-value, large reduced chi square value, so perhaps that peak is a significant new feature!

Exercise 6

Another experiment has now made a measurement of the mass of the new particle you are looking for, reporting a mass of $1020 \pm 17\ \mathrm{GeV/c^2}$, where the uncertainty is assumed to be normally distributed. You quickly fit the anomolous peak

seen in the data of the last exercise to a Briet-Wigner distribution and your first estimate is that the mass of the new particle you have detected is about $987\ \mathrm{GeV}/c^2$ but you have not had time to assess the errors in your masss value.

Using a two-tailed hypothesis test, does your measurement agree with the other experiment at a 10% significance level? What about a 5% significance level? Write your answer in the Markdown cells below.

What if we had an estimate of the error in our mass estimate. The bins we used are 8GeV so I would guess our errors are of that size.

In [39]:
```
# This is a straight copy of the code from the earlier discussion
from scipy.stats import norm # Note "norm" is the name for the "normal" or "Gaussian" distribution

# find the distance, the number of std.deviations between the two results
distance = (1010.0 - 987.0)/17.0
print("These two results are {:4.2f} sigma apart".format(distance))
for alpha in [0.05,0.1]:

    # Standard normal distribution, 5% significance level corresponds to 95% confidence interval
    interval = norm.interval(1 - alpha, loc = 0, scale = 1)
    print("\nThe {:4.1f}% confidence level interval where we accept the result is\n".format(alpha*100)+str

    if distance<interval[1] and distance>interval[0]:
        print("*** The two masses are the SAME at the {:4.1f}% confidence level".format(alpha*100))
    else:
        print("### The two masses are DIFFERENT at the {:4.1f}% confidence level".format(alpha*100))
```

```
These two results are 1.35 sigma apart

The  5.0% confidence level interval where we accept the result is
(-1.959963984540054, 1.959963984540054)
*** The two masses are the SAME at the  5.0% confidence level

The 10.0% confidence level interval where we accept the result is
(-1.6448536269514729, 1.6448536269514722)
*** The two masses are the SAME at the 10.0% confidence level
```

The two mass values are 1.35σ (standard deviation) apart. At a 10% confidence level, we would expect when trying to estimate the masses of the same particle in two different experiments that at least 1 in 10 of experiments would throw up results further apart than these two mass values just due to statistical fluctuations.

So we can accept at the 5% confidence level (the harder test to pass) that these two experiments are measuring the same mass.

We would hope that they are the same new particle but there would be a lot more physics to studny before we can be sure.

If we had an estimate of the error in our mass estimate the uncertainty in the difference of the mass values is even larger. As we are intereted in the difference of the two masses, the variance for the difference is simply the sum of the two variances in the two mass measurements. Hence the stadrad deviation is larger if we include the error in our own mass. Makes it even more likely that random fluctuations could have produce these two mass values from the same underlying distribution.

In [ ]: