

Curve Fitting

Curve fitting is all about finding the best function to match some data. These functions typically have variability on unknown parameters, and so once we've chosen a family of functions to model our data on, we need to determine the actual values of the parameters that best fits our data.

Today, we'll be trying to fit some simulated data, shown below, to a family of functions of the form:

$$f(x; a, b) = ax e^{-bx^2}$$

Here, a 'family' of functions just refers to the fact that we could get lots of different functions by choosing different values of parameters a and b .

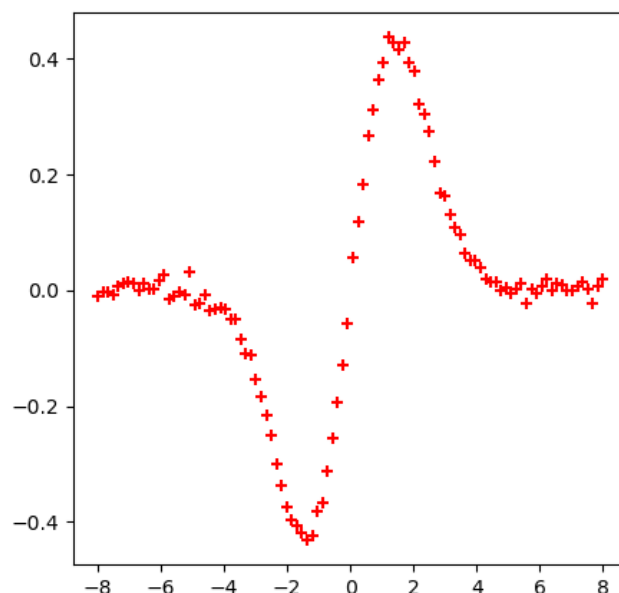


Figure 1: Some data that we want to fit too

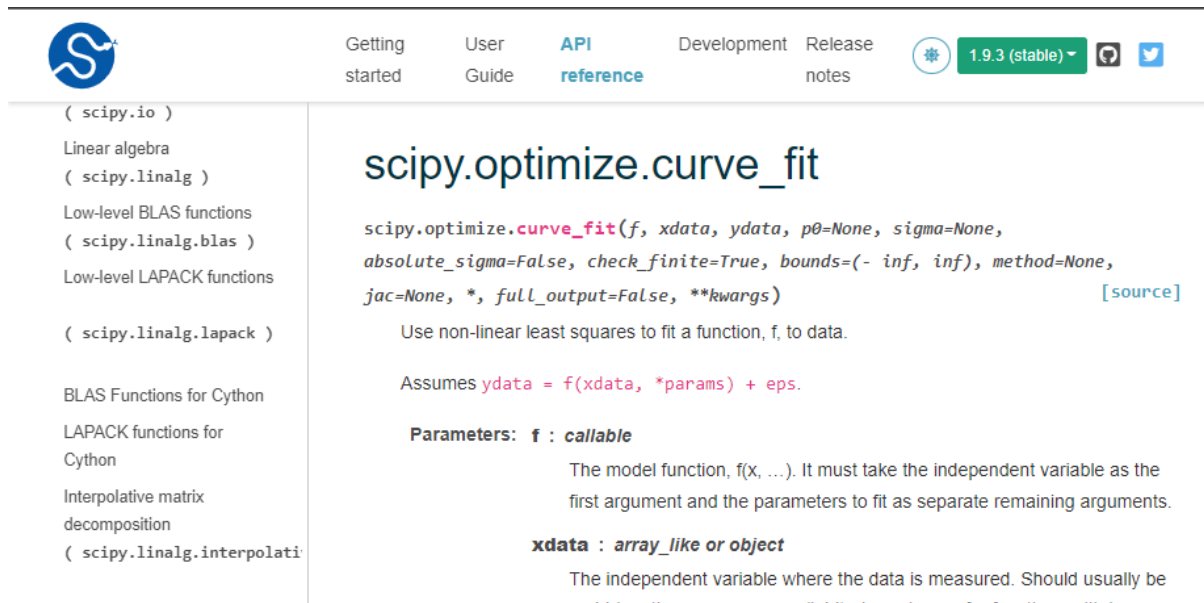
Our job is to find the *specific* values of a and b that best fit the data above.

To do this, we use a least-squares fitting function from SciPy called `curve_fit`, the documentation for which is available [here](#).

This might be the first time you've had to read any documentation, so before we begin coding, I'll just explain how I read documentation.

Reading Documentation

Here is a screenshot of the documentation of the `curve_fit` function taken from the website:



The screenshot shows the SciPy documentation page for `scipy.optimize.curve_fit`. The page has a header with navigation links: "Getting started", "User Guide", "API reference" (highlighted), "Development", and "Release notes". There is also a version selector showing "1.9.3 (stable)". The left sidebar contains a navigation menu with links to various SciPy modules: `(scipy.io)`, Linear algebra (`(scipy.linalg)`), Low-level BLAS functions (`(scipy.linalg.blas)`), Low-level LAPACK functions (`(scipy.linalg.lapack)`), BLAS Functions for Cython, LAPACK functions for Cython, Interpolative matrix decomposition (`(scipy.linalg.interpolati`), and others. The main content area shows the function signature: `scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False, check_finite=True, bounds=(- inf, inf), method=None, jac=None, *, full_output=False, **kwargs)` with a "[source]" link. Below the signature is a brief description: "Use non-linear least squares to fit a function, f, to data." and an assumption: "Assumes `ydata = f(xdata, *params) + eps`." The "Parameters" section lists: `f : callable` (The model function, `f(x, ...)`. It must take the independent variable as the first argument and the parameters to fit as separate remaining arguments.) and `xdata : array_like or object` (The independent variable where the data is measured. Should usually be an M-length sequence or an (K, M) shaped array for functions with K).

All that information may seem a bit scary, but we don't need to read it all! The most important bits of all documentation are the following:

1. The function's **DESCRIPTION**
2. It's **INPUTS** (or parameters)
3. It's **OUTPUTS** (which you'll need to scroll down for!)

The first thing we learn about `curve_fit` is that it 'uses non-linear least squares to a fit a function, `f`, to data' – this is good, we want to fit data to a model function!

Secondly, the first 3 parameters (or inputs) tell us what the function wants. It wants:

- a. a model function `f(x, ...)` – this needs to be a *callable*, which is basically just a function that we can *call* – and it'll depend on some parameters (e.g. `a` and `b` from above)
- b. some `xdata`
- c. some `ydata` – this will be the data we'll be fitting! (and it asks for them in *array_like* form, we'll just use numpy arrays).

Some functions have lots of inputs, but often we don't need to use all of them. You'll notice that `curve_fit` has a lot of inputs which take default values, and often we can just ignore them.

Scrolling down, we'll also learn about `curve_fit`'s outputs. Don't be put off by how far you have to scroll down – just look for where it says 'outputs' or 'returns'. Most important to use are the two arrays,

1. `popt` – an array of optimal values for the parameters (so that the error between the model function and the data is minimised)

2. `p cov` – a 2d array with estimated errors – you'll only ever have to use the items on the diagonal, which represent the estimated variances on the optimal parameter values.

Most documentation will also have some examples showing the functions in action, which can be super useful too!

Big Coding Tip: Being able to read documentation is really important, as it'll allow you to use functions that other people have written. Often you don't need to understand how a particular function works, only how to *use* it, and this is what documentation is for – teaching you how to *use* other people's code.

Also, this takes practice! Just try and remember that you only really need three bits of information from documentation: its *description*, its *inputs* and its *outputs*.

Fitting to axe^{-bx^2}

Now we're going to walk you through the process of fitting the above data to our model function $f(x) = axe^{-bx^2}$. Ultimately, we want to plot the data shown in Fig. 1, overlaid with a nice smooth curve that fits the data. You should have been sent a .csv file containing all the data points, let a demo know if not or send me and email at hga946@student.bham.ac.uk.

Let's **plan**:

First things first, import some libraries:

1. Import `numpy`, `matplotlib` and `scipy.optimize`.

Then, let's make sure we can read the .csv file into a numpy array:

2. Read in the .csv file to a numpy array

To check we've loaded in it right,

3. Try plotting the data – it should look like Figure 1.

Now to do the fitting, since `curve_fit` needs a callable $f(x, \dots)$ (which acts as the model function) we also need to:

4. Write a function `model_function(x, a, b)` that takes `x` as a dependent variable, as well as parameters `a` and `b`, and returns axe^{-bx^2} .

Once we've done this we can:

5. Call `curve_fit`, to calculate the optimal values of `a` and `b`

Finally:

6. Plot the data and overlay with a nice smooth curve that represents the best fit curve, using the parameter values found in part 4.

That's the overall plan, but let's go into a bit more detail for each step...

Firstly, step 1, remember we load in libraries by something like the following:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

For step 2, and loading in .csv files, firstly make sure you're in the same working directory as the 'simulated_data.csv' file, and then call:

```
17  ### load in data
18  data = np.loadtxt('simulated_data.csv', delimiter=',')
```

Being in the 'same working directory' is equivalent to saving the .csv and your .py file in the same folder. The 'delimiter' parameter here tells numpy what separates values in the .csv, so it knows what format to put the 'data' numpy array into.

Side note: The function `np.savetxt(...)` has the opposite effect of `np.loadtxt`, it will save a numpy array to a given file and I used it to create the 'simulated_data.csv' file that you're using. The documentation for [np.savetxt](#) and [np.loadtxt](#).

For steps 3 and 5, here's a template for plotting with matplotlib:

```
29
30  fig = plt.figure(figsize=(5,5))
31  ax = fig.add_subplot(111)
32  ax.scatter(x,y, color='red', marker='+', label='')
33  plt.show()
34
```

For step 4, just try calling `curve_fit(...)`, making sure you put the inputs in the right places (as per the documentation) If it doesn't work, see if you can understand the error it produces.

Ultimately, you should be able to plot something like this:

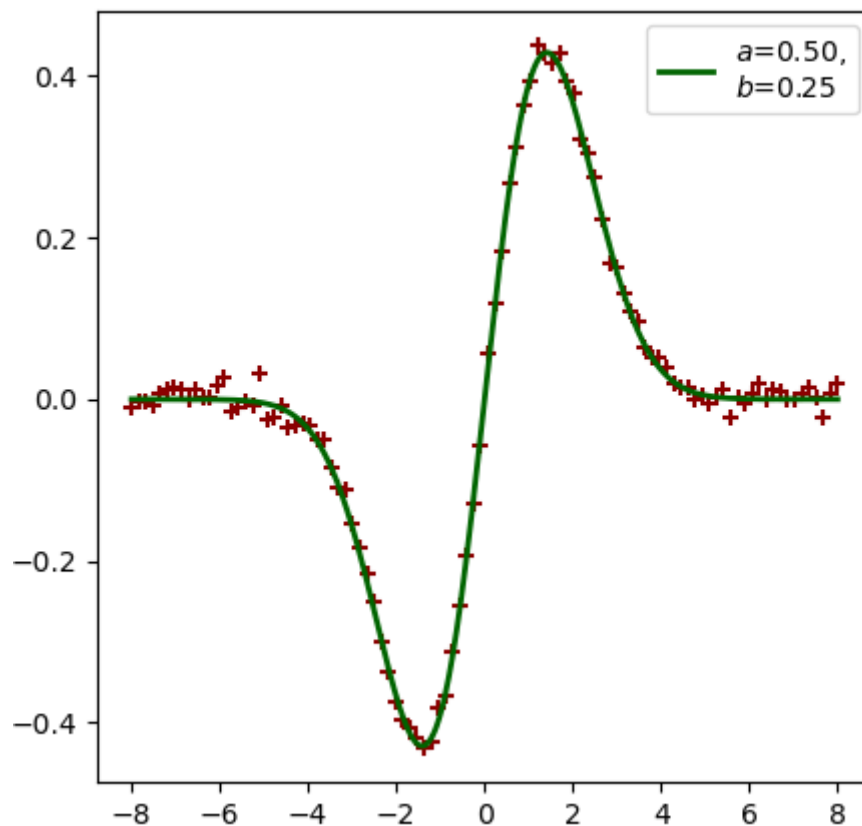


Figure 2: Data with best fit curve overlayed

Check you get $a = 0.5$, $b = 0.25$.

In case you're interested, the reason your best fit doesn't fit the curve perfectly is because I added a little noise in the y-direction to create this data :). Adding 'noise' is equivalent to adding a scaled normal distribution to each y-value – I called something like this:

```
20  
21 y = model_function(x, a, b) + 0.01*np.random.randn(len(x))
```