Introduction to fitting methods

Ohm's law

The function `curve_fit()` is introduced for fitting linear data. You will perform voltage and current measurements on a resistor to find the resistance via Ohm's law.

This lab contains explicit questions for you to answer labelled with **Q**. However, you still need to maintain a notebook for the lab, make plots and write Python code.

Background knowledge for exercise 1

Python arrays, numpy, pyplot, scipy, `curve_fit()`
Error Analysis Chi squared (χ^2), goodness of fit
Physics Ohm's law. Review this from your first year text.

1 Introduction

In this exercise, we will introduce how to *fit* experimental data to a certain function, and get *quantitative* observations from the experiment.

The package "Notes on Error Analysis" can be found at http://www.physics.utoronto.ca/~phy225h/web-pages/Notes_Error.pdf. It introduces linear and non-linear regression methods, aimed at finding the values of parameters which give the closest agreement between the data and the theoretical model. You should read the section on least-squares fitting.

Suppose our data consisted of N measurements (y_i, x_i) with $i = 1, 2, \ldots, N$. A **first step is to plot the data with error bars representing the reading error** (or the standard deviation σ_i) on each point. This gives a visual check for what relation our data has.

Next, assume that we want to make a function f that will predict a value $\tilde{y}_i$ if we call the function with the measurement x_i and some set of parameters p_0,p_1... that represents the experiment (e.g. resistance, gravity, spring constant could all be parameters). Our goal is to find the value of the parameters that will allow the function f to make the best predictions given the available data.

We could use this function f to make predictions for each measurement pair (x_i, y_i), and make guesses about the parameters until we find 'good' agreement, but a more robust method is to use the data fitting capabilities of Python to find the parameter values for us, using a form of *linear regression*.

2 Curve fitting

To make use of Python in our regression, we need to write our mathematical relationship between x and y as a Python function. In this experiment we'll be looking at an experiment that we think is modeled by a linear equation $y = ax + b$, where a and b are the parameters we are trying to determine. In Python code this becomes

```
def model_function(x, a, b):
    return a*x + b
```

Note that the independent variable `x` must be the first argument. In order for your `model_function` to work it **must** be able to take a `numpy` array and return an array.

This function only makes a prediction of the measurement of y given x. The job of finding the parameters is done by the `scipy` package in Python, using the `curve_fit()` function from the `scipy.optimize` module.

Everytime you call `curve_fit()` you will give it 3 arguments, and usually more. The first argument is your `model_function` that `curve_fit()` will use to make predictions, followed by the x data, and the y data. `curve_fit()` can also take 3 additional arguments — You can provide an initial guess for your parameters using the `p0` keyword argument, the uncertainties in the *dependent* data in the `sigma` keyword

argument, and you should (almost always) and force `curve_fit()` to consider uncertinties as absolute values (instead of relative values) using the `absolute_sigma=True` keyword. Your code will then look like this:

```
def model_function(x, a, b):
    return a*x + b

p_opt, p_cov = curve_fit(model_function,
                         xdata, ydata, p0=initial_guess,
                         sigma=sigmadata, absolute_sigma=True)
```

`curve_fit()` returns two values. The first value `p_opt` contains the estimates of the parameters you used in the model (in this case `p_opt` has two elements). The second value `p_cov` contains the covariance matrix giving the uncertainties in estimates of the parameters you used in the model (in this case `p_cov` is a 2×2 matrix). In most experiments the important values in `p_cov` will be the diagonal elements, which represent the variance (the uncertainty–squared) of the parameters. An easy way to extract these values and convert them to uncertainties is :

```
p_std = sqrt(diag(p_cov))
```

where `sqrt` and `diag` are functions from `numpy` that calculate the square root, and extract the diagonal elements, respectically. `p_std` will now contain your parameter uncertainties.

3 Curve fitting internals

Internally, `curve_fit()` uses an algorithm to find the best parameters. It does this by defining a *metric* called χ^2 that it tries to minimize. Starting with your initial guess for the parameters, `curve_fit()` will calculate the value of χ^2 based on the experimental data and your model, and adjust the parameters systematically until it finds the minimum value of χ^2 within a (very small) uncertainty.

For simple equations, like the linear model above, `curve_fit()` will take about 7 tries to find the best set of parameters. With more complex non–linear functions it can take 50 tries *if you make a good initial guess*, and more than 1,000 tries *if you make a bad initial guess*. In the default setup `curve_fit()` is configured to stop trying after 800 tries, where it will report that it failed and stop the program.

However, it is possible for `curve_fit()` to stop 'successfully' and have an ill fitting model if your model function is inappropriate, your data is inconsistent, or your initial parameter guesses were bad. In this case you might see that the parameters haven't been changed from your initial guess, or the variances are infinite (`inf`) or "not a number"(`nan`).

We will use the metric χ^2 later to measure how good the model function fits the data, but you don't need to define it for `curve_fit()` to work. In words, χ^2 can be written as

$$\chi^2 = \sum_{\text{measurements}} \left(\frac{\text{dependent data value} - \text{predicted value}}{\text{dependent data measurement error}} \right)^2 . \tag{1}$$

For a 'good' fit we want our model prediction to be about as close to each measurement (the numerator) as the uncertainty in each measurement (the denominator). We square the function because a model that is *too high* is just as bad as a model that it *too low*.

Mathematically, χ^2 is written as

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - f(x_i)}{\sigma_i} \right)^2 . \tag{2}$$

where x_i and y_i are the data from the i^{th} measurement, $f(x_i)$ is the prediction from the model of the value y_i, and σ_i is the uncertainty of the i^{th} measurement (**not the standard deviation of all of the measurements**).

There are many keyword arguments to `curve_fit()`, which can be found in the documentation. Commonly, you may use `maxfev` to control the maximum number of function calls. The default value is 800.

4 The experiment

We will be testing Ohm's Law – the linear relation between voltage and current in electrical circuits. The voltage is our independent variable, current is the dependent variable, and resistance is the parameter we'd like to measure. There are two parts to the experiment: testing a known resistor, and testing a potentiometer.

4.1 Uncertainty from multimeters

Our later analysis requires an idea of the uncertainty in the measurements. Digital multimeters have two types of uncertainty:

Error of accuracy which depends on function (DCV, ACV, DCA, ACA and Ohm (Ω)). This error is provided by the instrument manufacturer. This error is expressed as a percentage of the reading. Please check the link below for complete instrument specifications. The reading can be done on different scales of the same function, which changes the number of digits on display and also the error of accuracy.

Error of precision which is due to fluctuations at the level of the last digit. Take one digit (with position on the last digit and value 1) to be the error of precision.

The uncertainty is taken to be the largest of the two errors above.

For example, using a multimeter with an error of accuracy of 3%, you read 2.06 mA of the instrument. Error of accuracy is 3% of reading:

$$2.06\text{mA} \times 0.03 = 0.06\text{mA}$$

Error of precision is 0.01 (fluctuations in the last digit). Therefore, by choosing the largest of the two errors, we state that the uncertainty in reading 2.06mA is 0.06mA. The instrument specifications for the multimeter you'll use can be found at http://www.upscale.utoronto.ca/specs/tegam130a.html

Be sure to track the uncertainty in measurements from the multimeter. Also, keep in mind that there are other sources of uncertainty that are harder to measure, like the quality of electrical connections.

4.2 The experiment

You will be provided with two multimeters, a board with electrical components (see Figure 1), and enough wires to connect everything. Choose one resistor, and one potentiometer for the experiment.

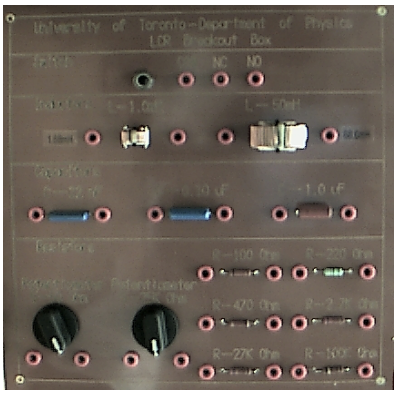

Figure 1: The electrical components board.

For the known resistor, perform the following.

1. Connect the ameter, voltmeter, and power supply to the resistor (Figure 2). *Ask the TA if you're in doubt*
2. Vary the voltage on the power supply.
3. Record the voltage and current from the multimeters, along with uncertainty.

4. Change the voltage, and record the new values – repeat.
5. Save the data on a memory stick as a text file (extension `.txt`) with the data in two columns: the first column being the independent variable (voltage).
6. After performing all the above measurements, disconnect the power, and switch the voltmeter to resistance. This will give you a reference value to compare against.

Repeat these measurements for a potentiometer, making sure not to move the dial between measurements. If your resistor board has no potentiometer, use another resister.

Resistors are marked with coloured bars to indicate their nominal resistance through a code (there are many calculators for this online). As part of this exercise, you may also compare your results against the nominal resistance ± the tolerance.

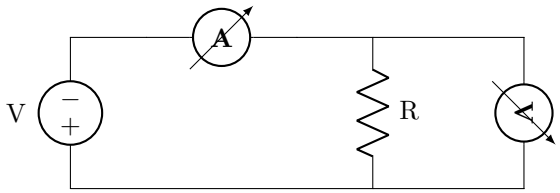

Figure 2: Circuit diagram of the experimental setup.

5 Analyzing the data

We will analyze the dependency of the current on the voltage using a linear fitting program. Ohm's law for resistors sates

$$I = \frac{1}{R}V\,. \tag{3}$$

Thus, the resistance of a linear component can be determined by plotting I vs. V, and measuring the slope.

There are other electrical components that have nonlinear voltage-current relations, like diodes, transistors, and amplifiers. Using a linear model for these would result in a bad fit. In a later exercise, we will perform a nonlinear regression with a lightbulb.

6 Build a linear fitting program

Linear fitting is done via the Levenberg-Marquedt algorithm (an advanced least-squares method), as implemented in the `scipy.optimize` module. Today, you will use the `curve_fit()` function (i.e. `scipy.optimize.curve_fit`). Calculation of the statistics from the ouput of `curve_fit()` will be done in the next exercise.

Here is the outline of the Python program for Exercise 1a:

- import the necessary modules and functions (e.g. `numpy` and `curve_fit()`)
- Read the data file into an array with the `loadtxt()` function.
- Extract the variables from the array columns.
- Define the model function $f(x) = ax + b$.
- Call `curve_fit()` with the function, data, and initial guess for the parameters.
- Output the calculated parameters.
- Create relevant plots.

Write a program to perform a linear fit for the data you collected in Section 4. It should create a plot of current vs. voltage with errorbars and the line of best fit, and output the calculated value of resistance (the slope of the line of best fit). Run this program using the data for the resistor, and again using the data for the potentiometer.

Q1 In Ohm's law, $V = IR$, the line should pass through $I = V = 0$. Did your linear fit pass through zero as well? If it didn't, why do you think that happened?

Q2 What result do you find for resistance if you force the line to pass through zero? (i.e. try the model function $f(x, a) = ax$)

Q3 How does your resistance from using `curve_fit()` compare to the value measured with the multimeter?

7 Goodness of fit - reduced chi-squared

Recall that the χ^2 distribution gives a measure of how unlikely a measurement is. The more a model deviates from the measurements, the higher the value of χ^2. But if χ^2 is too small, it's also an indication of a problem: usually that there weren't enough samples.

This best (most likely) value of χ^2 depends on the number of degrees of freedom of the data, $\nu = N - n$, where N is the number of observations and n is the number of parameters. This dependence is removed with the *reduced chi-squared* distribution,

$$\chi^2_{\text{red}} = \frac{1}{\nu} \sum_{i=1}^{N} \left(\frac{y_i - y(x_i)}{\sigma_i} \right)^2 , \tag{4}$$

where y_i is the *dependent* data you measured, x_i is the *independent* data, σ_i is the measurement error in the *dependent* data.

For χ^2_{red}, the best value is 1:

$\chi^2_{\text{red}} \gg 1$ indicates that the model is a poor fit;
$\chi^2_{\text{red}} > 1$ indicates an incomplete fit or underestimated error variance;
$\chi^2_{\text{red}} < 1$ indicates an over-fit model (not enough data, or the model somehow is fitting the noise).

Q4 Add a function to your program to calculate χ^2_{red}. What values were computed? How would you interpret your results?

Your submission for this exercise should include:

- A lab book describing what you did, including the setup of your electronic circuit, the settings on the multimeter, the band colors on the resistors you used, and any other details that might be relevant (was the power supply working properly? did you swap multimeters part way through? did you remove data points?...),
- Answers to questions 1-4,
- plots of current vs. voltage for each of the resistor and potentiometer,
- the calculated resistance values, including uncertainties,
- the final version of your program, and
- the recorded data from the experiment.

Resistors for electrical circuits are designed to be very linear and reliable, so the three measurements for resistance will likely be very close. The lines may even be indistinguishable on the plot.

Prepared by John Ladan and Ruxandra Serbanescu, 2016.