Basics of Mathematica for data analysis

Angelo Esposito
Physics Department, Columbia University

January 14, 2017

Abstract

In this brief tutorial I will introduce those features of Mathematica that are most useful to perform data analysis, namely plotting your data and performing fits with pretty much every functional form. You can obtain a license to download *Mathematica (student edition)* for free from the CUIT website. It is important to mention that, despite the present tutorial, if you are facing some problems with Mathematica or you do not know how to do something, the most efficient way to find the answer you are looking for is probably to use the Wolfram website. You can find most of the solutions with a simple Google search.

Since this is the first attemp to make some easy notes about Mathematica for data analysis, every comment and/or suggestion is more than welcome. You can always email me at `ae2458@columbia.edu`

Contents

1 Introduction

Wolfram Mathematica is a very powerful tool to complete a vast number of mathematical tasks. It is mostly famous for its ability to perform symbolic calculation, but it can also be used to perform numerical (approximate) integration and data analysis. Mathematica is designed to embed an incredibly large number of functionalities in a single software. It then follows that many of these tasks are not implemented in the most efficient way possible. However, it is extremely easy/intuitive to use and for the simple goals of our course is more than enough.

This tutorial is organized as follows. In Section 2 I will enumerate those aspects that are general to *any* Mathematica code and hence should be kept in mind through out the whole tutorial. In Section 3 I will introduce the most common functionalities of Mathematica. These will be fairly general and will come handy in the future, every time that you will decide to use the software. In Section 4 I will describe how to create a plot. This will mostly interests analytic functions of one variable and one dimensional data (also including error bars). Lastly, Section 5 explains how to perform a fit once data and errors are given. These last two will be likely the parts that you will need the most during the semester.

I will try to explain many of the ideas with some practical example, so that we can go a little beyond the "theoretical" aspects. Pictures of real Mathematica notebooks can be found at the end of this tutorial.

2 First things to know

Here I report some of the commands and syntactic features that are at the core of *any* Mathematica code. In particular:

1. Whenever you write a piece of code you can execute it by pressing `enter` (or `shift + return` if you do not have the `enter` key);

2. Mathematica allows you to divide your code in many sub-codes, each of them is enclosed in a square bracket on the right of the Mathematica window (see Fig. 1). Moreover, each sub-code can be executed independently from the others. It is typically a good habit to use them to enclosed different parts of your code. This way, it will be substantially easier to isolate bugs or mistakes whenever necessary;

3. The variables and functions defined in a certain Mathematica window are actually defined globally, *i.e.* they are common to all windows (also called *notebooks*). If you want to reset your variables you have to click *Evaluation* $\rightarrow$ *Quit Kernel* $\rightarrow$ *Local*, or type and run the `Quit[]` command on your notebook.

4. When you run a line of a Mathematica code, the software automatically plots on screen the output of that line, whatever it is. If you want to avoid this you will have to end the line with a *semicolon*;

5. Mathematica uses different colors for different kinds of codes. With time you will learn their meaning, and this will be quite useful to check for typos or bugs in

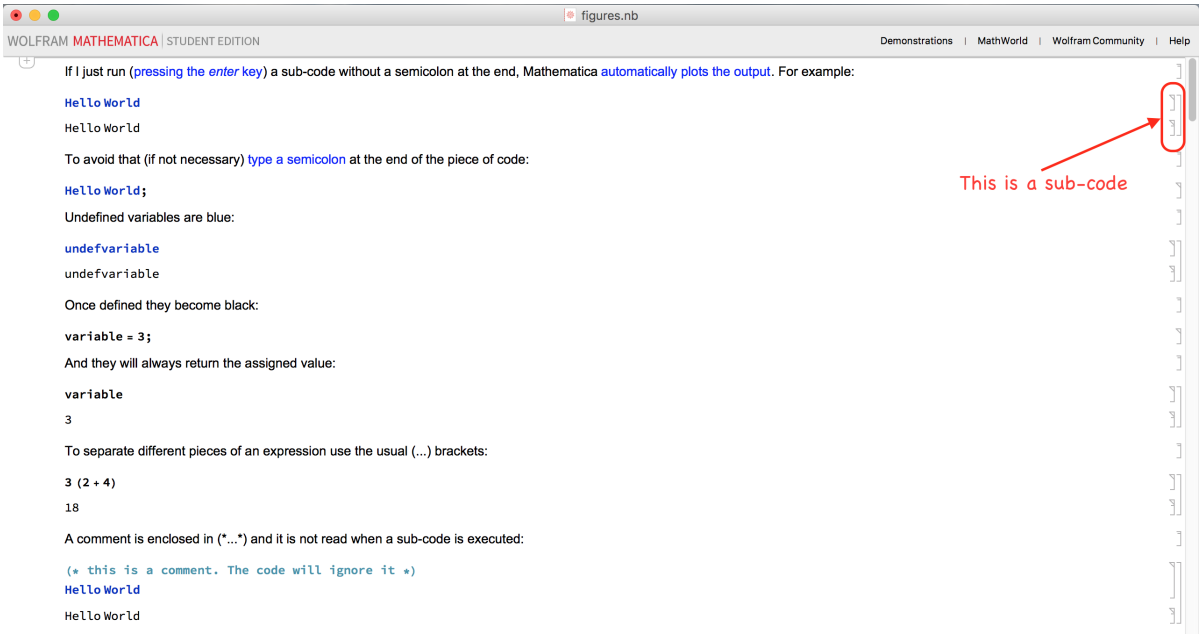


Figure 1: Simple illustration of some basic concepts

your code. In particular, a variable or function that is not yet defined[1] in the code is colored in *blue*, while a function or variable that has already been defined somewhere will appear in *black*. This can come very handy, especially to avoid overwriting;

6. Different kinds of brackets have different, important meanings. In particular curly brackets `{...}` are used in Mathematica to specify a table (see Sec. 4.2), square brackets `[...]` are used to enclose the arguments of a function (see Sec. 3), while normal brackets `(...)` have the usual mathematical meaning of separating different pieces of an expression;

7. Lastly, whenever you want to write some comment (for example to remind yourself what a certain part of the code does), you will have to enclose the text in `(*...*)`.

Examples of these basic concepts can be found in Fig. 1.

3 Main concepts

In this section and in the following ones I will assume that you are fairly familiar with the concept of variable and table (or array) as commonly used in any programming language. I will therefore start directly with what is linked to Mathematica itself.

Most of the functionalities of Mathematica are delivered in the form of *functions*. In this context, this term is somewhat broader than the one you might be used to. A function is in general an object that takes some input (*e.g.* the value of a variable, a set of raw data with errors, etc.) and gives back some output (*e.g.* a plot of the function, the

[1] In general, a variable is said to be *defined* or *assigned* when it is set equal to something. This something can be many different things: a number, a function, a plot and so on.

value of the fitted parameters, etc.). In general, the syntax for a function in Mathematica is of the form

$$\boxed{\texttt{NameOfFunction}[\{\texttt{arguments}\},\{\texttt{options}\}]}, \tag{1}$$

where both the `arguments` and the `options` are input quantities that you will have to provide to the function itself. Note that they do not necessarily have to be numbers. Sometimes they could be variables, strings, or even other functions. This should become clearer in a few pages.

Many of the most useful functions are already present in the Mathematica library. This means that you will not have to define them, you can just call them in your code and use them right away. Two built-in functions that you will probably use very often during the semester are

1. $\boxed{\texttt{Table}[\{\texttt{obj1},\texttt{obj2},\ldots,\texttt{objN}\},\{\texttt{i},\texttt{imin},\texttt{imax}\}]}$: This function creates a table with N columns filled with the quantities `obj1`, ..., `objN`. The variable `i` is the index indicating the row of the table, whose boundaries will be `imin` and `imax`. It then follows that the table will have $\texttt{imax}-\texttt{imin}+1$ rows. Clearly, the columns can depend on the index `i`. If you saved your table into a variable, say `mytab`, you can then access the element in the n-th column and m-th row with the syntax `mytab[[m,n]]`—see Fig. 2;

2. $\boxed{\texttt{Sin}[\texttt{x}],\texttt{Cos}[\texttt{x}],\texttt{Exp}[\texttt{x}],\ldots}$: These functions do exactly what you expect. They take a single variable `x` and they output the value of the corresponding mathematical expression. There is an enormous number of such expressions in Mathematica. You can find all of them in the *Help* → *Wolfram Documentation* window. Remember that for trigonometric functions the argument must be given in *radians*.

Although Mathematica has an incredible number of pre-existing functions, many times it is of utmost importance to be able to define your own. In particular, you will use it very often to create your own function to fit experimental data. The general syntax to define a function (any function) is

$$\boxed{\texttt{NameOfFunction}[\texttt{var1\_},\texttt{var2\_},\ldots,\texttt{varN\_}]:=\texttt{Expression}(\texttt{var1},\texttt{var2},\ldots,\texttt{varN})}. \tag{2}$$

The symbol "`:=`" is telling Mathematica that you are defining a new function and that it will have to save it somewhere. The input variables are specified right after the name by adding an *underscore* after them. Since you are creating this new function, it is up to you to decide which and how many input variables you want. Note that, during the definition, the name of the variables (here called `var1`, ..., `varN`) must be the same in the left and right hand sides of Eq. (2). However, later on, you are free to call them with different names. For example, you can use `x` and `y` in the definition but `u` and `v` in the rest of the code. Mathematica does not care.

The right hand side of Eq. (2) instead represents the function itself, *i.e.* what you want it to do. It can be, for example, an analytical expression, a plotting function, or anything you need at the moment.

Let's make a concrete example. Suppose you want to fit your data with a function given by[2]

$$f(x) = \left(\frac{\sin(x)}{x}\right)^2, \tag{3}$$

and you want to define it on your own. In this case the piece of code you need to write is the following

```
f[x_] := (Sin[x]/x)^2
```
. (4)

Now, in any other part of the code following the definition you can call the function with a given argument and read the result. For example, running `f[2.1]` will return $0.1689\ldots$. You can use your calculator to check that this is the correct result. Again, examples of user-defined functions are reported in Fig. 2.

Further things to know: Analytical expressions in Mathematica can easily become quite hard to read—see *e.g.* Eq. (4). Mathematica has some nice keyboard shortcuts to help you make the formulae look nicer and your code easier to read (as I have done in Fig. 2). You can find more about it `here`[3];

4 How to plot

This section is dedicated to the description of the different plotting functions available in Mathematica. These functions usually have an enormous number of different options. I will not describe all of them since this goes way beyond the aim of this tutorial (and probably beyond the ability of a single person...). Nevertheless I will present what I believe are the features that are necessary to your data analysis. If you are curious enough and want to make your plots fancier you can always read the documentation associated with each of the following functions.

4.1 Plot analytic functions

The easiest way to plot one or more analytic functions is using

```
Plot[{function1[x], function2[x], ...}, {x, xmin, xmax}, {options}]
```
, (5)

whose name was totally unexpected. Here `function1[x]`, `function2[x]`, etc. are the expressions that you want to plot. Mathematica will show all of them on the same canvas (*i.e.* on the same cartesian plane). The variable `x` will range from `xmin` to `xmax`, and the section `option` is left for you to specify all the features that will make your plot look better. A very basic example of plot is shown in Fig. 3.

In the following I report some of the options that will probably be more useful to you:

[2] Question: in which experiment will you have to use a function like that?

[3] If by any chance you printed the pdf and the reference does not work, here is the link: https://reference.wolfram.com/language/tutorial/KeyboardShortcutListing.html.

- `Axes → False, Frame → True` : This command will allow you to eliminate from the plot those (quite horrible) horizontal and vertical axes and replace them with a much nicer and more professional frame;

- `ImageSize → w` : This changes the size of the plot, which now has width specified by `w` (a number of your choice). It is quite useful to make your plot visible;

- `FrameLabel → {Style["xax", sx], Style["yax", sy]}` : If you eliminated the axes and replaced them with a frame, this option will allow you to label the x and y variables. The label on the horizontal axis will be `xax` (some text of your choice) with size `sx` (a number of your choice), while the one on the vertical axis will be `yax` with size `sy`;

- `PlotLabel → Style["labelname", s]` : Similarly to the previous option, this will label the whole plot. The label will appear on top of it as `labelname` with size `s`;

- `PlotStyle → {opt1, opt2, ...}` : This will help you modify the features of the plotted functions, *e.g.* color, thickness, etc. `opt1`, `opt2` and so on will be the options given to `function1[x]`, `function2[x]`, etc. respectively. The number of possible features here is too large. You are strongly encouraged to look at the examples presented here, and at the full Mathematica documentation.

In Fig. 3 I report some examples of plotting functions. The first is the most basic one, where the function is simply plotted with default parameters. In the second example you can instead appreciate some of the options that I have previously explained.

4.2 Plot experimental data

Let us now see how to plot experimental data, *i.e.* how to make a scatter plot of a list of points (x_i, y_i). Suppose that for some reason you have a table of data saved in the variable `data`. In order to be plotted, the table has to be organized as follows

$$\boxed{\texttt{data} = \{\{\texttt{x1}, \texttt{y1}\}, \{\texttt{x2}, \texttt{y2}\}, \ldots, \{\texttt{xN}, \texttt{yN}\}\}}, \tag{6}$$

where the (x_i, y_i) can respectively represent, *e.g.* an angle and a transmitted intensity, the thickness of aluminum and the number of β rays that manage to go through it, etc. Most of the times you will have to fill the table with the raw data obtained from the experiment, in which case you can define it explicitly exactly as in Eq. (6). However, other times this table might be the result of some mathematical manipulation that you had to perform on the raw data before getting to the physical quantity of interest.

Whatever produced the table `mydata`, plotting it is extremely simple. You need to use the function

$$\boxed{\texttt{ListPlot}[\{\texttt{table1}, \texttt{table2}, \ldots\}, \{\texttt{options}\}]}. \tag{7}$$

Beside some minor differences this function works exactly like `Plot[]`, except for the range along the x axis which is now computed automatically starting from the data x_i. An example of this can be found in Fig. 4.

4.3 Plot experimental data with error bars

As you will learn during the course, a set of data without a properly assigned set of errors is meaningless[4]. For the same reason, plotting a set of data without error bars is a mistake.

Mathematica produces fairly nice plots with error bars, but the procedure to obtain them is a little more involved than what we have seen so far. As a first thing you will need to load a particular *package* called `ErrorBarPlots`. This is done with the following command:

$$\boxed{\texttt{Needs["ErrorBarPlots`"]}}. \tag{8}$$

This procedure will add some additional functions to your code.

In order to plot the error bars, the structure of the table containing data and errors has to be different from the one presented in Eq. (6). In particular, if your data are (x_i, y_i), and the errors on the y variable are s_i, the table should be organized as

$$\boxed{\texttt{list} = \{\,\{\{\texttt{x1},\texttt{y1}\},\texttt{ErrorBar[s1]}\},\,\{\{\texttt{x2},\texttt{y2}\},\texttt{ErrorBar[s2]}\}\,,\ldots\}}. \tag{9}$$

Beside being used to plot error bars, I am not aware of any other instance where the function `ErrorBar[]` is useful.

In general, you are free to create the list by hand, by explicitly writing down the syntax in Eq. (9) for every data point. However, I do not recommend that. It is generally good to *keep data and errors separated in two different tables*, since this is what you will need to make a fit. You can alway create a separate table whose only goal is to be used for plotting.

A useful piece of code that does that is the following. Suppose that you saved your data in the variable `data={{x1,y1},...,{xN,yN}}`, and your error in `err={s1,...,sN}`. Clearly the two lists need to have the same number of elements. You can then combine them together to create a new table in the form of Eq. (9), with the following code:

$$\boxed{\texttt{list} = \texttt{Table}[\{\texttt{data}[[\texttt{i}]],\texttt{ErrorBar}[\,\texttt{err}[[\texttt{i}]]\,]\},\{\texttt{i},1,\texttt{Length}[\texttt{data}]\}]}, \tag{10}$$

Once the list with the correct format has been created, you can obtain your nice plot with error bars by using the function

$$\boxed{\texttt{ErrorListPlot}[\{\texttt{list1},\texttt{list2},\ldots\},\{\texttt{options}\}]}. \tag{11}$$

The options available here are exactly the same as for `Plot[]` and `ListPlot[]` and hence I will not repeat them. As usual, an example is reported in Fig. 5.

Further things to know:

1. To make your plots easier to read when you have multiple functions or datasets represented on the same canvas, it is useful to create a legend. So far, I have

[4]This does not necessarily mean that every quantity you measure comes with an error. However, it is crucial to understand that the error is *never* zero. When it is omitted it actually means that it is negligible with respect to the other uncertainties in play.

found a fair number of different ways of doing that. My favorite one to use the (quite complicated) `PlotLegends → Placed[SwatchLegend[]]` option. If you are interested in making legends, you should probably spend a few minutes navigating Google to find the option that you prefer. The example in Fig. 5 reports my personal choice;

2. When you have performed a fit, it is a good habit to superimpose to the data with error bars a plot of the fitted function. To do that one can use the function `Show[]`. In this case, for example, you have to use `Show[ErrorListPlot[...], Plot[...]]`. This will produce a single canvas with both the data points (produced by the function `ErrorListPlot[]`) and the fitting function (plotted using `Plot[]`). Note that the options given to the first argument will override those of the second argument;

3. Once you have created a plot (of a function or a set of data), you can simply save it as a pdf by right-clicking on it.

5 Fitting experimental data

Now that you know how to make a table, use Mathematica functions and plot different kinds of objects, there is only one item left from your data analysis list: how to perform a least squares fit.

5.1 A quick reminder

You can find more details about how a least squares fit works in the last section of the lab manual, or in any book of statistical analysis. Nevertheless, for the sake of completeness, I will quickly review here the main ideas.

Suppose you have a set of N data points, (x_i, y_i), with errors s_i, and that for some reason (*e.g.* theoretical predictions) you know that they should be described by some function $f(a, b; x)$. While x is a variable, a and b are parameters[5]. You could for example have

$$f(a,b;x) = ax + b \quad \text{or} \quad f(a,b;x) = a\frac{\sin(bx)}{bx} \quad \text{or} \quad f(a,b;x) = ae^{bx} \quad \text{etc.} \tag{12}$$

The idea is to find the value of a and b such that your function $f(a, b; x)$—now intended as a function of x—is the best possible description of the data.

A quantity that somehow gives a measure of "how close your function is to the data" is the so-called *chi squared*

$$\chi^2 = \sum_{i=1}^{N} \frac{\left(y_i - f(a,b;x_i)\right)^2}{s_i^2}. \tag{13}$$

Its interpretation is easy. The numerator is simply the distance between the function computed at x_i and the actual value of the data. The presence of the square tells you that we are interested in the "absolute closeness", regardless if the function is above or

[5] I am using only two parameters for a matter of concreteness. The whole idea easily generalizes to an arbitrary number of them.

below the data point. Lastly, the denominator is used to weight the different y_i. Data points with a smaller error s_i are more reliable and hence contribute more to the sum.

You now want to find the values of a and b that make your function the closest possible to the data, *i.e.* you want to minimize the χ^2. This means that you will have to solve

$$\frac{\partial \chi^2}{\partial a} = 0 \qquad \text{and} \qquad \frac{\partial \chi^2}{\partial b} = 0. \tag{14}$$

Solving the previous equations will give you the *best fit* value of the parameters, let us call them $\hat{a}$ and $\hat{b}$, which will now be dependent on the data points and their errors:

$$\hat{a} \equiv a(x_i, y_i, s_i) \qquad \text{and} \qquad \hat{b} \equiv b(x_i, y_i, s_i). \tag{15}$$

The previous equation gives the actual value of the best fit parameters. Their errors can be found as usual. If every point y_i has an associate error s_i then the propagated uncertainty on the parameters is simply

$$s_{\hat{a}} = \sqrt{\sum_{i=1}^{N} \left(\frac{\partial \hat{a}}{\partial y_i} s_i \right)^2} \qquad \text{and} \qquad s_{\hat{b}} = \sqrt{\sum_{i=1}^{N} \left(\frac{\partial \hat{b}}{\partial y_i} s_i \right)^2}. \tag{16}$$

Do not worry if you did not understand everything and/or if it looks terribly tedious (it is...). Mathematica will do all of it for you behind close doors.

5.2 Least squares fit with Mathematica

5.2.1 The NonlinearModelFit function and its options

Let us suppose again that you saved your data in a table, `data={{x1,y1},...,{xN,yN}}`, and the errors in another, `err={s1,...,sN}`. There are at least three different functions in the Mathematica library that perform fits. However, I will only focus on one of them since I believe it is the most complete one and can be used with every functional form for the fitting function. This function is

$$\boxed{\texttt{NonlinearModelFit[data, fitfunc[x], \{parameters\}, \{variables\}, \{options\}]}}. \tag{17}$$

The syntax is fairly simple to understand. The first argument is simply your `data` table, while the second one is the functional form of the fitting function. This has to depend both on x and on the parameters $(a, b, c, \dots)$. In the third argument you have to specify what the parameters to be fitted are. You will have to provide them as a table as well, *i.e.* `{a,b,c,...}`. The fourth argument specifies what quantities are you considering as variables. Although you can have more general situations, in every case of interest for you this will just be `{x}`.

Now we have to provide the options, which in this case are essential. We have to firstly tell Mathematica that we want our fit to be weighted with the squared errors as in Eq. (13). To do that, we must specify the option $\boxed{\texttt{Weights} \to \texttt{1/err}^2}$. However, it turns out that this is not enough to tell Mathematica to perform a least squares analysis as the one outlined in Sec. 5.1. If we do not give any other option, the `NonlinearModelFit`

function will find the right value of the fitted parameters but underestimate their errors. To find the right uncertainties as in Eq. (16) we also have to give the option `VarianceEstimatorFunction` $\rightarrow$ `(1&)`. The reasons why this is the right option to provide are rather obscure and I personally learned it empirically. You can however find some kind of explanation `here`[6].

An example of how to perform a fit is reported in Figs. 6 and 7.

5.2.2 How to access the results of the fit

Suppose that you saved the result of your fit in the variable `myfit` by typing

$$\texttt{myfit} = \texttt{NonlinearModelFit}[\dots]. \tag{18}$$

You are now able to access pretty much all the information you want about your fit. In the following I report the quantities that you will use more often during the semester and how to access them. See again Figs. 6 and 7 for concrete examples.

- *Fitted function*: The functional form of the best fit function is straightforwardly obtained as `myfit[x]`;

- *Overview of the results*: By giving the command `myfit["ParameterTable"]`, Mathematica will plot a nice and elegant table containing all the values of the best fit parameters together with their errors and other quantities (whose meaning is not interesting to us);

- *Best fit parameters*: The complete values of the parameters are stored in `myfit` under `myfit["BestFitParameters"]`. For some obscure reason, Mathematica presents them in the form of *replacement rules*[7]. To actually access the numerical value of the parameter, you will have to type its name followed by "`/.`", *e.g.* `a /. myfit[''BestFitParameters'']`;

- *Errors on the parameters*: The errors resulting from the best fit are instead saved in `myfit["ParameterErrors"]`. In this case, this will be a table and hence you can obtain the value of a certain element as for any other table. For example, `myfit[''ParameterErrors''][[1]]` will give the error on the first parameter and so on;

- *Residuals*: The residuals are defined as the difference between y_i and the fitted function computed at x_i, *i.e.* $\text{res}_i = y_i - f(a, b; x_i)$. The list of the residuals corresponding to each data point is stored in `myfit["FitResiduals"]`. Again, you can access the i-th residual with `myfit[''FitResiduals''][[i]]`.

5.2.3 How to check the goodness of your fit

Once you have performed a fit, it is always a good habit to check whether or not it is good. This will typically help you understand if you are using the right fitting function,

[6] https://reference.wolfram.com/language/howto/FitModelsWithMeasurementErrors.html.

[7] In general, something is a Mathamatica's replacement rule if it appears in the form `variable` $\rightarrow$ `value`.

if you are estimating the errors correctly, and so on. There are two things that you can do to make sure that your fit worked correctly: plot the residuals and check the value of the χ^2.

If your fit is properly done then the residuals will be more or less randomly distributed around zero—see Fig. 7. This is because if the fitting function is the right one the data points will end up some times above the function and some times below it. If you observe a clear pattern in the residuals this is a bad sign. It can mean, for example, that your function is not the right one or that you did not estimate the errors correctly.

Computing the χ^2 is also a very good habit. You can see from Eq. (13) that it can be related to the residuals by

$$\chi^2 = \sum_{i=1}^{N} \left(\frac{\mathrm{res}_i}{s_i} \right)^2 . \tag{19}$$

In particular, this means that it can be computed in Mathematica using the function `Sum[]` as in the following[8]:

```
Sum[ ( myfit["FitResiduals"][[i]] / err[[i]] )^2 , {i, 1, Length[err]} ] .
```
(20)

It is clear that the smaller the χ^2 the closer your fitted function is to the actual data points. If a fit is reasonable one would like to have $\chi^2/\mathrm{DOF} \lesssim 1$, where DOF are the so-called *degrees of freedom*, which are defined as

$$\mathrm{DOF} = \text{number of data points} - \text{number of fitting parameters.} \tag{21}$$

You should learn to look at your value for χ^2/DOF carefully. In particular, if it is too much bigger than unity this might be typically due to two reasons:

1. Your fitting function is the wrong one and hence it is very far from the experimental points. Make sure that you are using the right function. If that is the case, it means that there is little you can do about it. You do not have a good fit, but this is not a major problem.

2. The errors you assigned are too small. Sometimes, even if the fitting function is correct, the χ^2/DOF might still be quite large because the error bars are underestimated. This means that you are not properly taking into account all the sources of errors. Try to correct that.

6 Short conclusions

I have presented here the most important aspects that you need to know about Mathematica and its functions in order to perform the basic data analysis required by this course. Even though Mathematica is typically very user-friendly and intuitive, you might at first have to struggle with it a little bit, as it often happens with new programming languages. I personally believe that there is no better way to learn a new software than to get your hands dirty, produce mistake and understand how to solve them. Typically, no tutorial or textbook can help you with that. However, if you succeed in the task, as I am sure you will, it will be another useful information to add to your personal toolbox.

[8] The syntax of the function `Sum[]` is very straightforward and I will not explain it in detail.

Tables

A boring table

Create a table of zeros with 3 columns and 10 rows. Let us assign it to the variable `mytab':

```
In[1]:= mytab = Table[{0, 0, 0}, {i, 1, 10}];
```

Show the table by simply writing its name and running the code:

```
In[2]:= mytab
Out[2]= {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}}
```

The number of elements of the table can be found using the function `Length[]':

```
In[3]:= Length[mytab]
Out[3]= 10
```

A more interesting table

Create a table of two columns where we save the values of the index and of its square:

```
In[4]:= mytab2 = Table[{i, i^2}, {i, 1, 10}];
        mytab2
Out[5]= {{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49}, {8, 64}, {9, 81}, {10, 100}}
```

You can access the element in the n-th column and m-th row with the syntax `mytab2[[m,n]]':

```
In[6]:= mytab2[[2, 1]]
Out[6]= 2

In[7]:= mytab2[[7, 2]]
Out[7]= 49
```

Mathematical functions

Mathematical functions are easy to play with. Just choose one and feed it with some argument. Run the code to obtain the value of the function:

```
In[8]:= Sin[2 π]
Out[8]= 0

In[9]:= Exp[3.06]
Out[9]= 21.3276

In[10]:= Log[0]
Out[10]= -∞
```

User-defined functions

Define a function of one variable:

```
In[11]:= myfunc[x_] := Cos[x] + Sin[x]^2 e^x;
```

From now on, I can call the variable with whatever name I prefer:

```
In[12]:= myfunc[snoopy]
Out[12]= Cos[snoopy] + e^snoopy Sin[snoopy]^2
```

Read the value of the function for whatever argument you want:

```
In[13]:= myfunc[1.]
Out[13]= 2.46505

In[14]:= myfunc[13.2]
Out[14]= 189 426.
```

A similar syntax can be used for a multi-dimensional function. For example:

```
In[15]:= myfunc3D[x_, y_] := (x^2 + y^2) Sin[x - y] Cos[x + y];
```

Read the value for different points (x,y):

```
In[16]:= myfunc3D[1., 1.]
Out[16]= 0.

In[18]:= myfunc3D[π, 1.3]
Out[18]= -2.9795
```

Figure 2: Example of the use of simple functions. This is a real Mathematica screenshot.

The most basic kind of plot

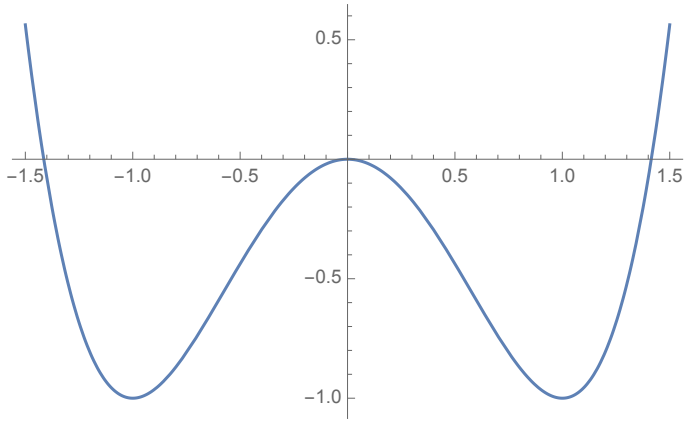
Let's define a function:

```
In[17]:= higgs[x_] := -2 x^2 + x^4;
```

And now let's just plot the function without any fancy option:

```
In[18]:= Plot[higgs[x], {x, -1.5, 1.5}]
```

Out[18]=



A nicer plot of many functions

Define the functions we want to plot together:

```
In[19]:= f1[x_] := x Sin[x];
         f2[x_] := x Cos[x];
```

Now plot them together. The options I am adding here will kill the axes and create the frame, produce the labels, change the size of the image, change the color of the lines:

```
In[21]:= Plot[{f1[x], f2[x]}, {x, 0, 4 π}, Axes → False, Frame → True, ImageSize → 450,
          FrameLabel → {Style["x", 14], Style["f1,2 (x) ", 14]}, PlotStyle → {Red, Green}]
```

Out[21]=

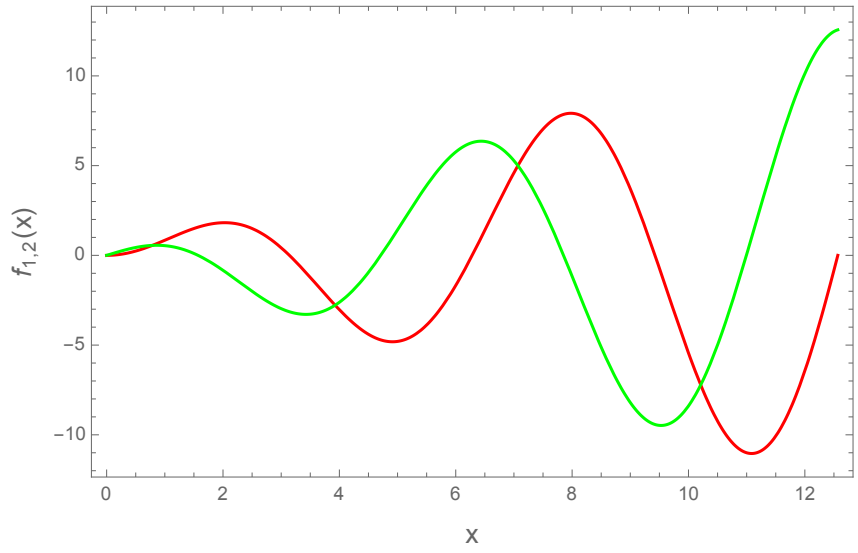


Figure 3: Example of how to plot one or more mathematical expressions, without and with customized features.

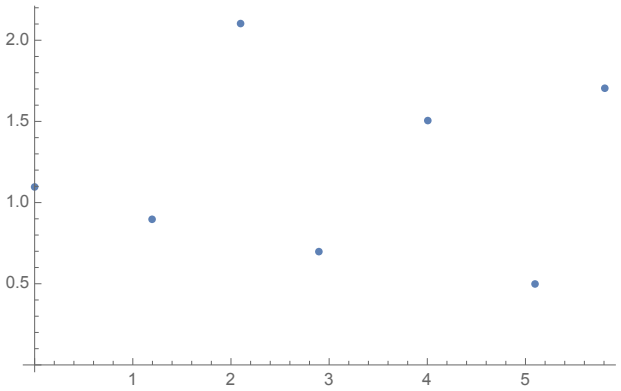
Plot of data

Define explicitly the table containing the data.
Each (x_i, y_i) pair has to appear in between curly brackets `{xi,yi}':

In[29]:=
```
mydata = {{0, 1.1}, {1.2, 0.9}, {2.1, 2.1}, {2.9, 0.7}, {4, 1.5}, {5.1, 0.5}, {5.8, 1.7}};
```

In[30]:=
```
ListPlot[mydata]
```

Out[30]=



Let us specify some options to make the previous plot less ugly:

In[35]:=
```
ListPlot[mydata, Axes → False, Frame → True, ImageSize → 450,
 PlotLabel → Style["mydata", 20, FontFamily → Times],
 FrameLabel → {Style["x", 16], Style["y", 16]}, PlotStyle → {Blue, PointSize[0.015]}]
```

Out[35]=

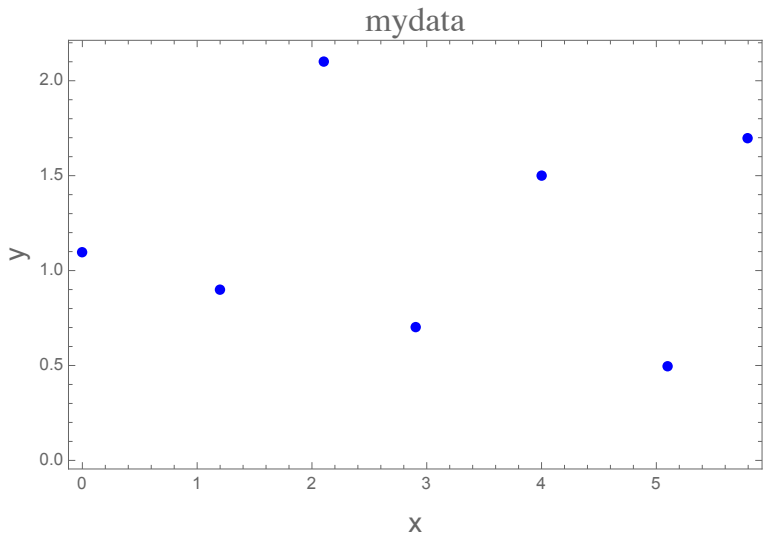


Figure 4: Example of functions to plot a list of (x_i, y_i) data.

Load package

As a first thing we need to load the additonal functions:

In[1]:=
```
Needs["ErrorBarPlots`"]
```

Define data and errors

Define two variables containing the tables for data and errors. For example:

In[2]:=
```
data = {{0, 0.03}, {1, 0.98}, {2, 2.00}, {3, 3.01}, {4, 4.05}, {5, 4.92}};
err = {0.12, 0.11, 0.13, 0.09, 0.12, 0.09};
```

It is essential for the two tables to have the same number of elements. This is true, in fact:

In[4]:=
```
Length[data]
Length[err]
```

Out[4]= 6

Out[5]= 6

Now join the two tables together to create a 'plottable' list:

In[6]:=
```
list = Table[{data[[i]], ErrorBar[err[[i]]]}, {i, 1, Length[data]}]
```

Out[6]=
```
{{{0, 0.03}, ErrorBar[0.12]}, {{1, 0.98}, ErrorBar[0.11]}, {{2, 2.}, ErrorBar[0.13]},
 {{3, 3.01}, ErrorBar[0.09]}, {{4, 4.05}, ErrorBar[0.12]}, {{5, 4.92}, ErrorBar[0.09]}}
```

Plot

Finally make the error bar plot. Also, create a legend (this is my favore method but it might not be the only one):

In[7]:=
```
ErrorListPlot[list, Axes → False, Frame → True, ImageSize → 500,
 PlotStyle → {Blue, PointSize[0.009]},
 FrameLabel → {Style["x", 16], Style["y", 16]},
 PlotLegends → Placed[SwatchLegend[{Blue}, {"My experimental points"}], {0.2, 0.86}]]
```

Out[7]=

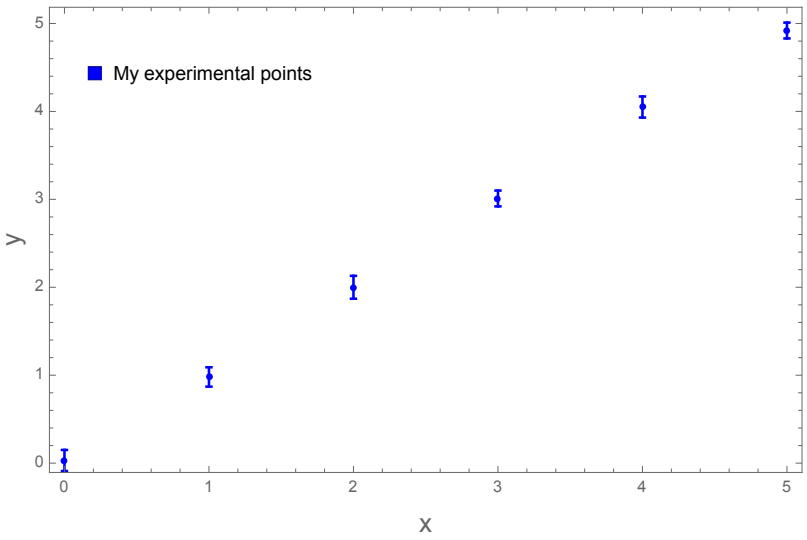


Figure 5: How to plot data and error bars together.

Least squares fit

In[7]:=
```
Needs["ErrorBarPlots`"]
```

Suppose my data and errors are:

In[8]:=
```
data = {{-4, 9.03}, {-3, 3.97}, {-2, 1.01}, {-1, -0.02}, {0, 0.99}, {1, 4.03}, {2, 8.99}};
err = {0.17, 0.15, 0.13, 0.16, 0.12, 0.18, 0.13};
```

From theoretical expectations I know that these data should be described by a parabola of the kind:
$a\,x^2 + b\,x + c$
I will therefore perform the fit in the following way and save the result in the variable `myfit':

In[10]:=
```
myfit = NonlinearModelFit[data, a x^2 + b x + c, {a, b, c}, {x}, Weights → 1/err^2, VarianceEstimatorFunction → (1 &)];
```

I can take a look at the parameters and their errors in an elegant form by doing:

In[11]:=
```
myfit["ParameterTable"]
```

Out[11]=

	Estimate	Standard Error	t-Statistic	P-Value
a	1.00147	0.0157838	63.4494	3.69591×10^{-7}
b	2.00134	0.0391237	51.1541	8.74028×10^{-7}
c	0.992066	0.0760084	13.0521	0.000198897

If I want to access the actual value of the parameters I will do:

In[12]:=
```
abestfit = a /. myfit["BestFitParameters"]
bbestfit = b /. myfit["BestFitParameters"]
cbestfit = c /. myfit["BestFitParameters"]
```

Out[12]= 1.00147

Out[13]= 2.00134

Out[14]= 0.992066

While I can read the errors on the parameters as:

In[15]:=
```
sa = myfit["ParameterErrors"][[1]]
sb = myfit["ParameterErrors"][[2]]
sc = myfit["ParameterErrors"][[3]]
```

Out[15]= 0.0157838

Out[16]= 0.0391237

Out[17]= 0.0760084

To plot the fitted function together with the data and error bars first make the `plottable' list:

In[18]:=
```
list = Table[{data[[i]], ErrorBar[err[[i]]]}, {i, 1, Length[data]}];
```

Then use Show[] to plot function and data together:

In[19]:=
```
Show[
 ErrorListPlot[list, Axes → False, Frame → True, ImageSize → 500, PlotStyle → {Blue, PointSize[0.008]},
  FrameLabel → {Style["x", 14], Style["y", 14]}, PlotLabel → Style["Best fit + data", 18]],
 Plot[myfit[x], {x, -4, 2}, PlotStyle → {Red, Thickness[0.001]}]
]
```

Out[19]=

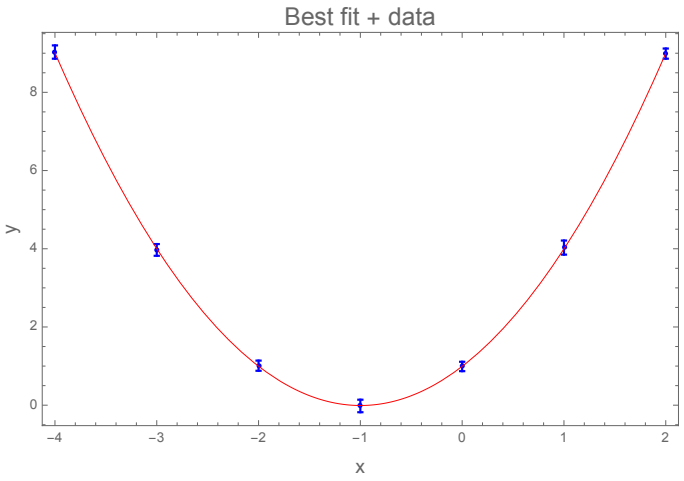


Figure 6: Least square fit with Mathematica and plot of the fitting function + data.

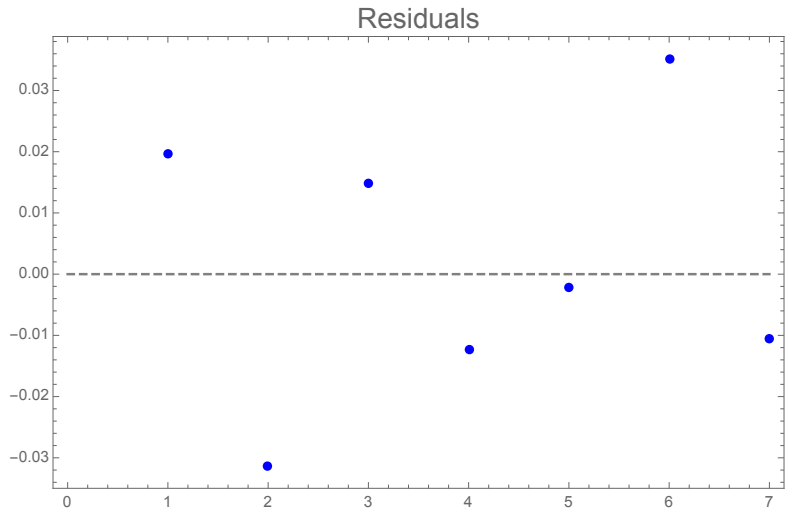
The residuals can be found with:

```
In[20]:= myfit["FitResiduals"]
Out[20]= {0.0197542, -0.031291, 0.0147239, -0.0122012, -0.00206627, 0.0351287, -0.0106162}
```

and therefore can be plotted with (it is a good habit to also show a dashed line corresponding to zero for reference):

```
In[21]:= Show[
 ListPlot[myfit["FitResiduals"], Axes → False, Frame → True, ImageSize → 500, PlotStyle → {Blue},
  PlotLabel → Style["Residuals", 18]],
 Plot[0, {x, 0, 7}, PlotStyle → {Gray, Dashed}]
]
```

Out[21]=



Lastly, given the residuals compute the χ^2:

```
In[22]:= χ2 = Sum[((myfit["FitResiduals"][[i]])/(err[[i]]))^2, {i, 1, Length[err]}]
Out[22]= 0.120715
```

Figure 7: Fig. 6 continued.