Plotting data using Mathematica

Eric D. Black
California Institute of Technology
v1.0

1 Introduction

If you've never used Mathematica before it can seem a little alien. It is very different from just about any other software paradigm out there. Computer programs written in traditional languages such as Python or C++ consist of a series of statements or commands that get executed in order when you run the program, and their output is either displayed on the screen or written to a file. Spreadsheets such as Microsoft Excel display an array of numbers, text, or formulas, and some of the elements in that array contain the results of calculations that use other elements as input. If you change one element the whole sheet updates automatically. Both of these force you to conform your notation and commands to a particular format that is convenient to the computer, or more accurately to the people who wrote the program. Mathematica is designed the other way around. Here, the computer attempts to conform to the way you would do a calculation with pen and paper. It's not perfect, of course, but it is pretty good once you get the hang of it. Oh, and it can do a *lot* more than any spreadsheet or even a traditional programming language with a lot less work on your part.

Imagine you were doing some math with pen and paper, but instead of regular paper you are using a "smart" paper. On this special paper you can write down, say, an arithmetic problem or an integral, but instead of you having to come up with the answer you can just ask the paper to do it for you. Of course you have to be very explicit and ask the paper only well-specified questions. Also, you may have scribbled quite a bit on the paper, and you only want the answer to one specific question or series of questions. For this you will have to draw a box around exactly what you want answered, but as long as you're clear on what you want, the paper can give it to you.

Mathematica documents are called *Notebooks*, and each notebook is organized into *cells*. The notebook is the equivalent of our smart paper, and a cell is the equivalent of a box drawn around a statement or question. Mathematica evaluates only what is in a given cell at any one time. It does remember the results of all of the cells it has evaluated during a session, but unlike a spreadsheet it does not update them all as soon as one cell is edited.

Mathematica's syntax does have a bit of a learning curve, but once you are comfortable with it you will find it easier to use than a spreadsheet for most tasks.

2 Opening a file, entering data, and producing a plot

This is what the Mathematica icon looks like. If you are using one of the imacs in the lab, it should already be in the dock. The program starts with either a new, empty notebook or a dialog box asking you what you want to do. If you get the dialog box, click on "New Document," and a new notebook window will appear.

At this point you can type your data in by hand, or you can open an existing data file. For this example I have downloaded the data file from the ph3 website to the desktop on my mac, and the first thing I am going to do is set Mathematica's working (default) directory to that location. To do this use the command,

```
SetDirectory["~/Desktop"]
```

All commands in Mathematica begin with capital letters, and their arguments are contained in square brackets. After you have typed the command, hit "Enter" on your numeric keypad or "Shift-Return" to execute it. (Just hitting "Return" makes a new line inside the cell.) The output is the full

Figure 1: Opening dialog in Mathematica. For this tutorial, if you see this, just click on "New Document."

path of the directory, along with some complaining from Mathematica which I will ignore (see Figure 2). The brackets on the far right, near the edge of the window, delineate the first cell. You can have as many or as few commands in a cell as you want, and Mathematica will run them all in order when you tell it to execute that cell.

Next import the data from the file Data1.txt into Mathematica, using the `Import` command,

```
data = Import["Data1.txt", "Table"]
```

The arguments of the `Import` command are first, the data file itself, and second, the format of the data. The output is a *list* that we will store in the variable `data`. This list is a collection of pairs of numbers, the first of which is the x value of the data, and the second its y value. Those of you familiar

with computer science will recognize that `data` is now a two-dimensional array, and that Mathematica took care of the dimensioning, formatting, etc. for you.

Now plot the data using the command,

```
dataplot = ListPlot[data]
```

Again, we are sending the output of this command to be stored in a variable, in this case `dataplot`. Note that this time the variable type is a graph, rather than a number or array. You can store just about anything in a variable, which comes in handy in a lot of ways. More on this later.

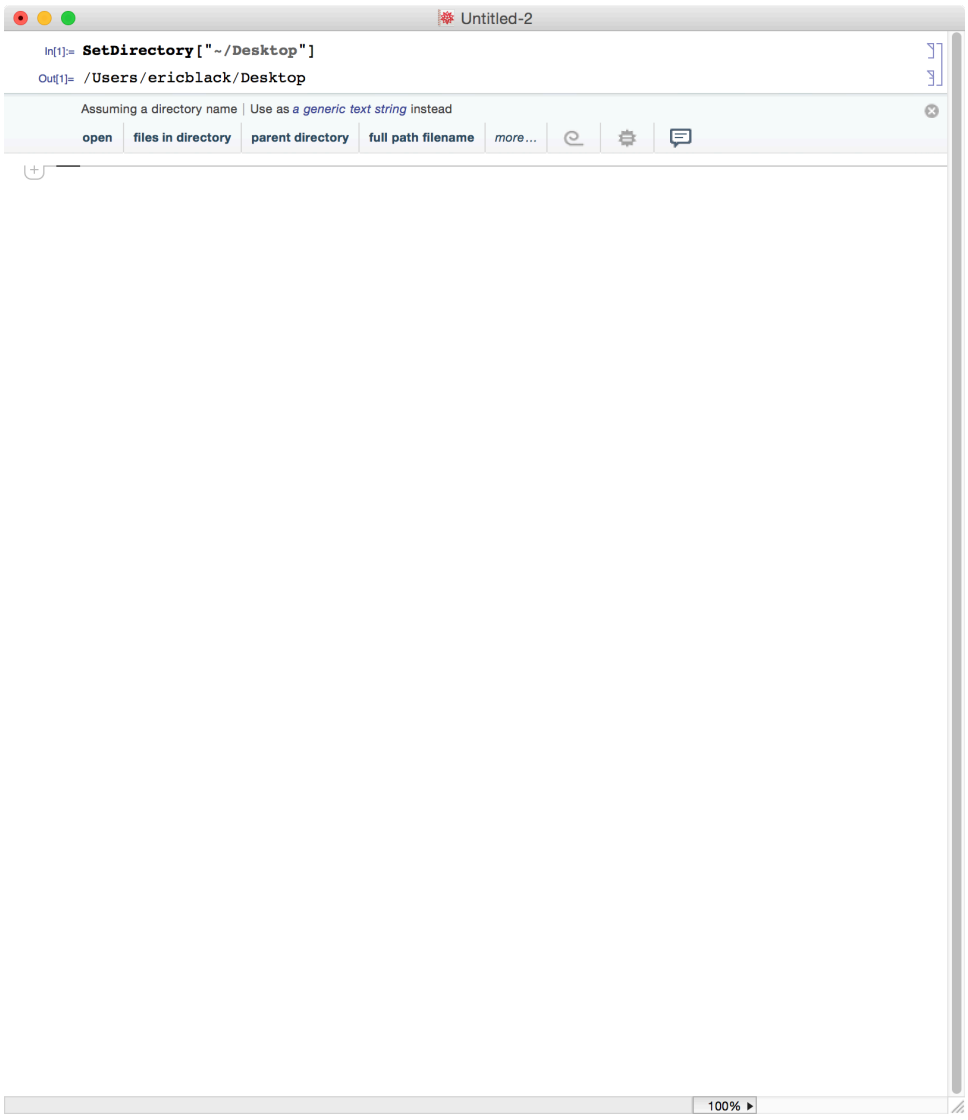

Figure 2: New notebook in Mathematica. The first command is to set the working directory to where the data file is located, in this case the desktop on my mac.

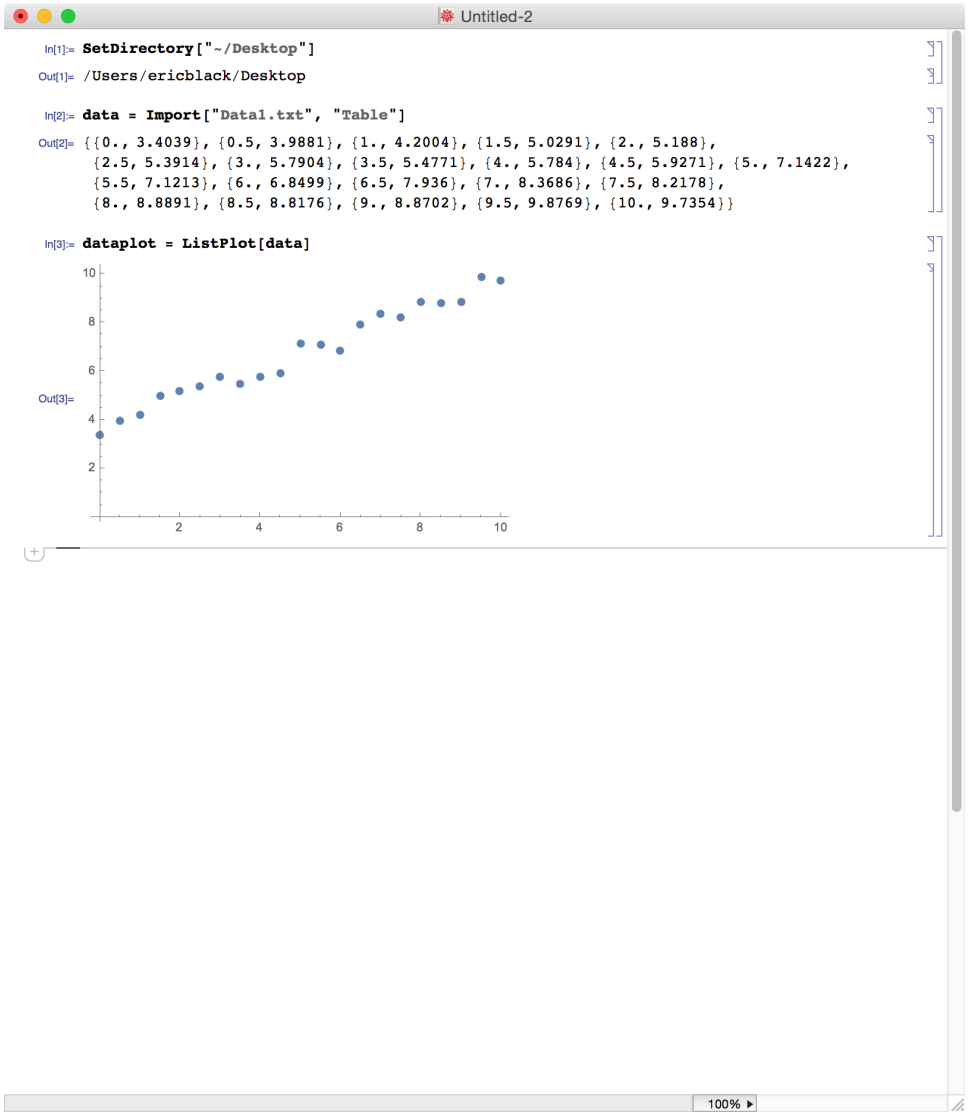

Figure 3: Importing and plotting data from a file. This notebook now has three cells, as the square brackets on the right show. Each cell is subdivided into an input and an output, and there are square brackets with slightly-different formats to show this.

3 Error bars

Mathematica has a special command for plotting data points with error bars, and it is called `ErrorListPlot`. Most commands are available immediately when you start Mathematica, but this one requires you to load a special package first. To do that, type in the following command, and execute it.

```
Needs["ErrorBarPlots`"]
```

Be sure to include the backwards single quotation mark (`), which is usually in the top-left corner of your keyboard, just to the left of the number one key and just below the escape key. Once this package is loaded you can now use the command `ErrorListPlot`. Before we try this command out, however, we are going to have to do some work to construct its argument, which must be a list of the form

```
{{{x1, y1}, ErrorBar[s1]}, {{x2, y2}, ErrorBar[s2]}, ... }
```

where $x1$ and $y1$ are the x and y values of the first data point, and s1 is the uncertainty in y1, etc. (You can add uncertainties in x as well, but we won't need them for this example.) You can construct this list by hand if you only have a few data points, but constructing the list using a single command will give me an excuse to show you some useful techniques for manipulating elements of a list. We will construct the argument for the `ErrorListPlot` command the following way.

```
darg = Table[{{data[[i,1]], data[[i,2]]}, ErrorBar[0.3]},
       {i,1,Length[data]}]
```

The output should look like the list shown in Figure 4. There are several things to notice about this command.

1. The command `Table` generates a list.

2. We are storing that list in the variable `darg` (short for "data argument," but you could call it anything you want).

3. We can extract the particular elements of the array `data` by appending what amounts to subscripts inside double square brackets. The name of the number that resides in the i-th row and j-th column of the array is

```
data[[i,j]]
```

4. The index for the table is i, and its range goes from 1 up to the number of data points you have. You could have counted them, gotten a total of 21, and plugged the range in as

   ```
   {i, 1, 21}
   ```

   However, it is much easier to tell Mathematica to do the counting for you by using the `Length` command, so the range becomes

   ```
   {i, 1, Length[data]}
   ```

5. I have used the same error for each data point (0.3), but you could just as easily have put in a different uncertainty for each point.

Now you can pass this list to the function `ErrorListPlot`, and the output should be a plot similar to what `ListPlot` generated, except with error bars on each data point.

```
dataplot2 = ErrorListPlot[darg]
```

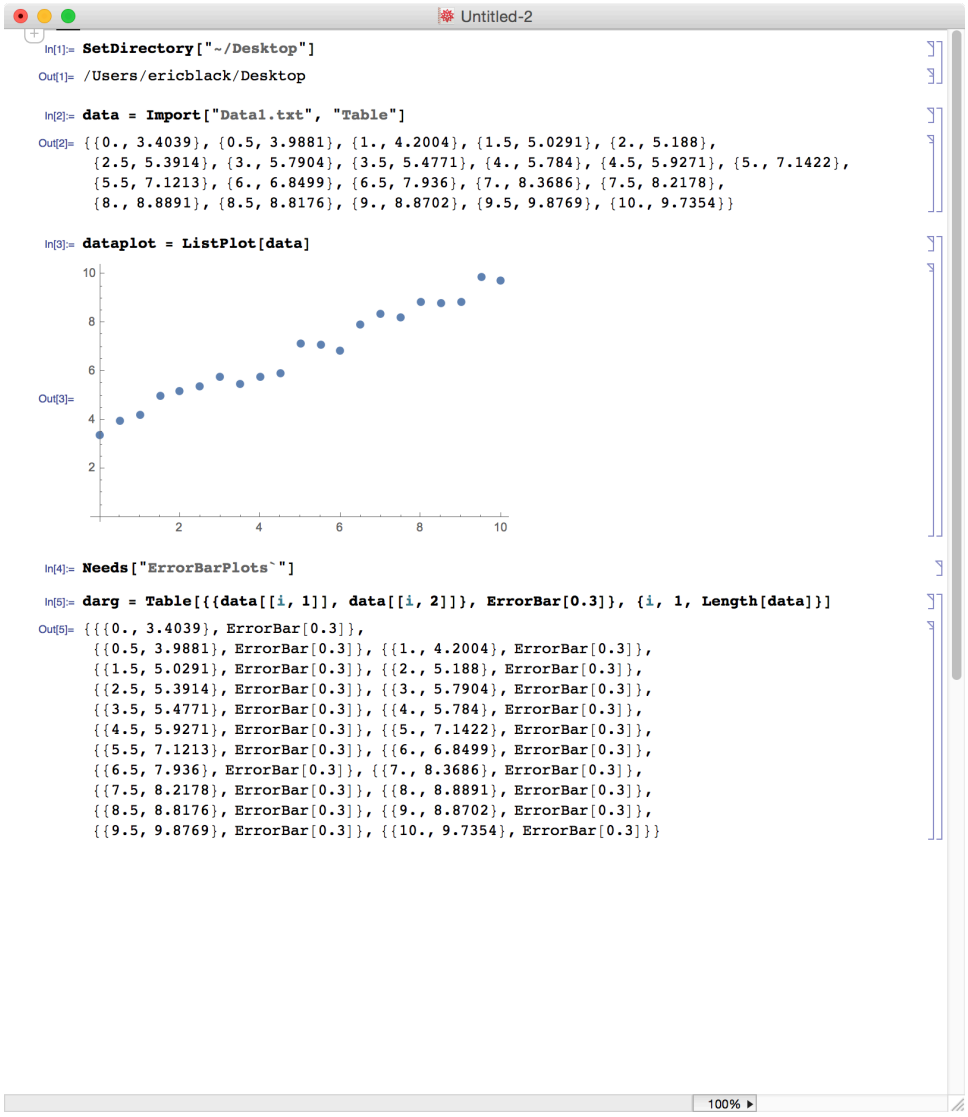

Figure 4: Constructing a list of data points with error bars.

4 Adding a theory curve

Now you know enough to go on with, and we can start doing some useful things rather quickly.

First, we can plot a line through the data using the `Plot` command. This one, like `ListPlot`, is intrinsic to Mathematica, and you don't have to load any special packages to use it. Before we use it, though, we will want to define a function to plot. Let's start with a line of the form

$$f(x) = ax + b$$

Before we can ask Mathematica to plot this we have to have definitions of the slope a and y-intercept b. We are fortunate that our first data point has an x-value of zero, so a good first guess for the y-intercept is that first y-value.

$$b = y_1$$

The slope a is going to be approximately the rise over run between the first and last data points,

$$a = \frac{y_N - y_1}{x_N - x_1}$$

The way to write these expressions in Mathematica is

```
b = data[[1,2]]

a = (data[[Length[data],2]]-data[[1,2]])/
    (data[[Length[data],1]]-data[[1,1]])

f[x_] := a*x + b
```

You can put all three statements into a single cell, but you ought to put the definitions of a and b before that of the function f so Mathematica will know what those coefficients are. Note the underscore after the x and the colon-equals := in the definition of the function.

Plotting this function is easy.

```
theoryplot = Plot[f[x], {x, -1, 11}]
```

We have been storing our graphs in variables, and now you will see why. Use the `Show` command to overlay multiple plots.

```
Show[dataplot2, theoryplot]
```

5 Residuals

You now know enough to calculate and plot your residuals. If you've been paying attention the following commands should be familiar enough.

```
residuals =
 Table[{{data[[i, 1]], data[[i, 2]] - f[data[[i, 1]]]},
   ErrorBar[0.3]}, {i, 1, Length[data]}]

ErrorListPlot[residuals]
```

6 Reduced chi-squared

The only new command I have to introduce you to here is `Sum`, and it works exactly like you'd expect. To write the expression,

$$\widetilde{\chi}^2 = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{y_i - f(x_i)}{0.3}\right]^2$$

we use the following Mathematica syntax.

```
rchisq = Sum[((data[[i, 2]] - f[data[[i, 1]]])/0.3)^2, {i, 1,
    Length[data]}]/Length[data]
```

7 Least-squares fitting

Your reduced chi-squared in this case probably won't be all that close to 1, because we haven't yet optimized our values for a and b. You can calculate these coefficients using the summation formulas in Equations 2 and 3 of the main handout (you know enough commands to do that now), but you can also have Mathematica do it for you using the `Fit` command.

```
lsq = Fit[data, {1, x}, x]
```

Here, as usual, we are storing the output of the `Fit` command in the variable `lsq`. The arguments may require a little explanation. First is the data, here given by the variable `data` that we created when we first imported the data from its file on the Desktop. Second is a shorthand notation for the form of

the function we are fitting to the data, in this case a first-order polynomial in the variable x. Finally, the variable itself x. The `Fit` function can fit to a polynomial of any order by changing the second argument. For example, you could fit to a quadratic by typing

```
Fit[data, {1, x, x^2}, x]
```

Mathematica can do fits many different ways, most of which are beyond the scope of this discussion, but you should know they exist. One in particular I want to point out to you is this. If you want a quick fit and residuals plot, and you don't care about error bars, try the following.

```
lsq2 = LinearModelFit[data, x, x]
ListPlot[lsq2["FitResiduals"]]
```

This gives a plot of the residuals without error bars, and I honestly don't know if it's possible to add them to this. If you've followed along, however, you can make your own residuals plot with error bars using `ErrorListPlot`, as we did earlier.

8 Summary

8.1 File-handling commands

1. `SetDirectory["path"]` - Set the working directory.
2. `Import["file", "type"]` - Read data from a file. There are literally hundreds of file types supported. For a list of these, see *http://reference.wolfram.com/language/guide/ListingOfAllFormats.html*. We are primarily interested in `"Table"`.
3. `Export["file", data]` - Write data to a file. We didn't cover this on in the writeup, but you may as well know about it. Again, literally hundreds of formats are supported. Much like a variable, you can store just about anything in a file.

8.2 Plotting commands

1. `Plot[f[x], {x, min, max}]` - Plot the function $f(x)$ from $x = min$ to $x = max$. The vertical scale is set automatically, unless you add the option

```
Plot[f[x], {x, min, max},
   PlotRange -> {{xmin, xmax}, {ymin, ymax}}]
```

2. `ListPlot[data]` - Plot a list of data. `PlotRange` works here the same as it does in `Plot`.

3. `ErrorListPlot[data]` - Plot data with error bars. Requires loading a package before you can use this one. Command is

   ```
   Needs["ErrorBarPlots`"]
   ```

   Elements of the data list must be of the form

   ```
   {{x, y}, ErrorBar[s]}
   ```

4. `Show[plot1, plot2]` - Overlay plots `plot1` and `plot2`, and show them on the same graph. Supports essentially any number of plots.

8.3 List-related commands

1. `Table[expr[i], {i, min, max}]` - Generate a list of values of the expression *expr* as the index i goes from min to max.

2. `Length[list]` - Length of `list`.

3. `list[[i,j]]` - Value of element i, j of `list`.

8.4 Arithmetic

1. `Sum[expr[i], {i, min, max}]` - Sum of `expr[i]` over $i = min$ to $i = max$.

8.5 Fitting commands

1. `Fit[data, model, variable]` - Least-squares fit of `model` to `data`.

2. `LinearModelFit[data, model, variable]` - Similar to above, except with more options.

3. `NonlinearModelFit[data, model, variable]` - Fit a nonlinear model to a set of data. For more information see http://reference.wolfram.com/language/ref/NonlinearModelFit.html