Introduction to Data Analysis and Plotting with Jupyter

Dustin Wheeler

January 20, 2023

> The goal of this exercise is to familiarize the student with basic data manipulation and plotting using Jupyter notebooks. Many of the concepts are general and can be translated to other software applications. By the end of this exercise, you should be able to
>
> - perform basic manipulations on data columns (formatting, transformations)
> - perform simple mathematical operations over a set of data
> - plot columns of data in a clear and visually pleasing manner
> - perform simple statistical operations on a set of data
> - get a line-of-best-fit for multiple datasets
> - import data from plain text files (CSV, TSV, DAT, etc.), like those created by many spectrometers and logging instruments

Introduction

Why do we plot data? Edward Tufte may have said it best:

> At their best, graphics are instruments for reasoning about quantitative information. Often the most effective way to describe, explore, and summarize a set of numbers—even a very large set—is to look at pictures of those numbers.

A Little Bit About Plots

Plots (also called graphs or charts, though these are not *quite* the same thing) come in a variety of styles. The can be used to explore data as your are performing analyses or to convey results to a reader as part of a report. There are a variety of "standard" types which should cover your needs in this course. The main types we will use are:

Scatter plot Used to show correlation between two variables, each point generally represents an **independent** measurement.

Points are not **connected** by lines, lines are instead used to represent a mathematical model fitting the data.

Line graph Indicates continuity in the x dimension, often suggests a single collection session for all data on the "line" (e.g., a UV-Vis or NMR spectrum). Individual points are generally not shown. Lines

should never be smoothed (or "splined"), as this treatment no longer accurately represents the data.

Really, a line graph is just a special case of a scatter plot where the independent variable is continuous.

Histogram Represents the distribution of data in a series of "bins", shows the frequency of a measured event. Closely related to density plots.

Here are the important parts of a plot, along with a brief description and guidelines for formatting.

Title Should be descriptive and represent *all* data in the plot.

Axes Should be clearly labeled (category, units, tick numbers). Should have an appropriate range (from slightly less than your minimum value to slightly more than your maximum value).

Fit line Should cover the whole spread of the data, but should not predict beyond the data range (you don't know anything about measurements in that region).

Legend Should clearly show which set of data is which, trend lines should clearly correlate to a given data set.

Basic Plotting and Linear Fits

Your first task is to take the data in the file `data.txt` and import it into a new Jupyter notebook.[1]

As the first step, please use this link to clone the files into your Jupyter directory:

`https://tinyurl.com/chem357-plotting`

This link will direct your browser to the class JupyterHub and download a folder containing a number of resources you'll need for this exercise.

The folder `raw_data` contains a few datasets in various formats. The file extension doesn't always tell you what type of information is in a file and can sometimes lead you astray. For this exercise, the file `data.txt` contains a set of "tab-delimited" data, meaning that the values in a row are separated by tab characters. Another common format is "comma-delimited" or "comma-separated" data, such as the dataset in `anscombe.csv`. You will find that different instruments have different defaults for exporting data, but these two are the most common. Sometimes, a `.dat` extension might be used for a plain-text file to indicate 'data' without indicating the underlying format or delimiter type.

[1] Note: prior to performing the following exercises, you should run through the introductory documentation to Jupyter to familiarize yourself with the concepts and interface.

Make sure that you are located in the top folder for this lesson (i.e., inside of "357_plot_intro") and open a new Jupyter template by clicking on the Template icon, then select "pchem_templates" in the top box and "PChem_Lab.ipynb" in the second box. This will open a new Jupyter notebook with a series of cells pre-populated with information to get you started. Give this notebook a title (as instructed in the first cell), then click on File in the JupyterLab menubar, select "Rename Notebook…", and give the notebook a title following the instructions in the first cell.

In the notebook, execute the code cell under the "Library Import" section. "Libraries" are just collections of useful Python code that individuals or groups have created. They generally cover a specific area or hold to a theme. In our notebook template, we use the following:

Numpy A general purpose numerical library that speeds up number based operations, allows functions to operate on lists of numbers, and gives a large collection of useful mathematical functions (things like polynomial fitting, matrix operations, and common algebraic and trigonometric functions). The core functionality is based around Numpy arrays, very fast n-dimensional arrays of data (a core requirement for scientific computation).

Scipy A fundamental library for scientific computing, builds on the Numpy library but includes numeric integration routines, non-linear curve fitting routines, interpolation, numerical optimization, and statistics.

Pandas A data management library that offers functions for importing, tidying, organizing, and filtering data using the DataFrame construct. Excellent resource for 2-dimensional data, starts breaking down with more dimensions (look at the xarray package, which is built on top of Pandas, for this purpose).

Matplotlib A comprehensive library for creating static, animated, and interactive visualizations in Python. It has become the de-facto standard library for plot visualizations in Python.

Seaborn A collection of routines and formats built on top of Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.

Now, create a new cell under the Data import section. Import the data from `data.txt` by using the import function of the "Pandas" (`pd`) library:

```
data = pd.read_csv("raw_data/data.txt", sep="\t")
```

This will take the contents of `data.txt` and turn it into an object called a "DataFrame" with the name `data`.[2] The `sep="\t"` argument is required because our data is "tab-separated", rather than "comma-separated", which is the default (as suggested by the function name `read_csv`). When in doubt, specify the field separator when you import data with python and the pandas library.

To view your new object, just type `data` on a new line and press Shift + Enter to evaluate the cell. By default Jupyter notebooks will display the results of the final line of a cell (if there is anything to display).[3] You should end up with two columns of data with 21 data points in each column. Once you have the data columns, create a new scatter plot from the data. The Pandas library automatically incorporates basic plotting functions from the Matplotlib library, so data in a DataFrame can be plotted by using the `plot()` function on a given DataFrame. In your case, you need only type `data.plot()` and evaluate the cell to see a plot of the data.[4] While you are exploring data, the default appearance is fine, but you'll probably want to start customizing the appearance of plots when you get them ready for publication.

Notice that the plot isn't plotting column X against column Y, but instead is plotting X and Y against the *index* (the bold column of numbers on the left side of the displayed data). There are two ways to correct your plot. The first is to import the data and define the index column during the import, like so:

```
data1 = pd.read_csv("raw_data/data.txt", sep="\t",
    index_col="X")
data1.plot()
```

This method is useful if you know the structure of the data ahead of time (and when you have a clear index column, such as a time index, wavelength, or frequency). The second way to fix the plot is to define the *x* and *y* data in the arguments of the `plot()` function. The `plot()` function takes a number of arguments (you can see them quickly by pressing Shift + Tab inside of the parentheses), including arguments for features like plot type (e.g., `kind="scatter"`), plot title (e.g., `title="My Plot"`), and axes labels (e.g., `xlabel="Time", ylabel="Effort"`).

With this in mind, create a scatter plot (`kind="scatter"`) of Y and a function of X, give the plot a title ("X vs. Y" is fine) and label each axis ("X", "Y").

[2] For more information, type `help(pd.DataFrame)` to get more information on the DataFrame object. Alternately, in the Jupyter notebook, you can press Shift + Tab at the end of a command to get help on the object (i.e., when your cursor is at the end of `pd.DataFrame` or your new object `data`.)

[3] They only display the results of intermediate lines of a cell if the `print()` command is used.

[4] *n.b.:* Pandas is designed first as a tool for finance and business operations. The default settings for plots are often inappropriate for scientific plots, so some tweaking may be necessary.

Polishing Plot Appearance

When creating more complex plots, once the basics are complete, remove any unnecessary "chart junk" to prepare your plot for publication. This includes things like background shading (Matplotlib is pretty good about this), grid lines (only necessary if you're trying to manually read data off the chart, we don't need this with the ubiquity of computers), and the border surrounding the full plot area (should only need axes to indicate the origin or provide an anchor for tick marks). If there is a legend (useful when there are multiple datasets present), it should usually be a part of the plot area, not pushed off to the side. Don't waste valuable data area on something like a legend than can be nestled in one of the empty parts of your plot. Finally, while color seems like a nice feature, plots shouldn't rely on color to convey meaning. There's always a chance that readers won't have access to color printouts, or that the reader may be colorblind. To keep plots accessible, use symbols, line weights, and dashing to differentiate your datasets. The lesson to learn from this paragraph is that simplicity is key to conveying your message. The sole purpose of your figures is to help the reader understand the message. Anything that distracts from that is harmful to your cause.

Now that the plot has been cleaned up, we'd like to add a trendline. The Seaborn library is a visual analysis package built on top of Matplotlib with an emphasis on data exploration and friendly figures. To plot our data with Seaborn, input the following:

```
1 sns.lmplot(data=data, x="X", y="Y", ci=None)
```

This creates a scatterplot using the X and Y columns from `data`, fits it with a linear model ("lm"), and disables the confidence interval (CI) in the plot.[5] A linear fit should be sufficient for this data. The `seaborn` is really designed for exploratory data analysis, not for complex curve fits and edge cases of data. It is generally inappropriate for spectral data, but very good for visual statistical explorations of data.

[5] "ci" can be set to a numeric value to represent the confidence interval of the associated fit, typically 95 % (`ci=95`).

Fitting Data

Now that you have a trendline, it's time to get some information about it. The Numpy library is a fundamental library in the numeric python community. It contains a vast array of functions and structures that are used by *many* other libraries.[6] One useful function in Numpy is the `polyfit()` function, which calculates a polynomial of best fit using a linear least-squares method. Examples can be seen on the linked page, but we will run through the simplest example, a straight line.

[6] The Numpy library (shortened to `np` by convention) has a host of useful functions, including a number of common mathematical functions. The square root function is one of these (`np.sqrt(num)`). Another useful one (that you'll need later) is `np.log10(num)`.

To use the `polyfit()` function, you need to specify the data ("X" and "Y") and the degree of the polynomial (1). All other arguments are optional, and just giving these three arguments will return the coefficients of the polynomial from the highest power to the lowest $(c_n, c_{n-1}, \ldots, c_1, c_0)$. The optional argument `cov=True` will return the *covariance matrix*, the diagonal values of which are the variances for each coefficient.

These (or, more precisely, their square roots) help you to see how "good" the returned fit is. Please read the help page to learn what statistics are returned and in what form.[7]

Implement this using the following code:[8]

```
# Fitting using the `numpy.polyfit()` function
# Function returns fit coefficients (1d array of values) and an n x n covariance matrix
fit_coeff, fit_cov = np.polyfit(data["X"], data["Y"], 1, cov=True)

# The standard error of each coefficient is just the square root of that diagonal element.
fit_err = np.sqrt(np.diag(fit_cov))

# Use formatted strings (f-strings) to display values with useful labels
print(f"slope: {fit_coeff[0]}, std. err.: {fit_err[0]}\n\
intercept: {fit_coeff[1]}, std. err.: {fit_err[1]}")
```

[7] If you're interested in doing more statistical modeling in Python, the Statsmodel package offers a number of descriptive statistics and flexible models.

[8] A couple of notes on Python features:

- everything following the number sign (#) on a line is considered a *comment*, meaning that it isn't evaluated. This is a great way to document your code as you work.
- The "f-strings" feature was introduced in Python 3.6. They are a far more convenient way to included code in printed strings, and result in much more readable code. Use them if you have the option, I won't bother teaching you older methods of formatting strings.
- Line 3 assigns multiple variables at once by using tuple assignment. Don't worry too much about the mechanics of this, just know that it's a handy tool when the right side of your assignment puts out multiple items.

Anscombe's Quartet

Repeat the process of importing data with the file `anscombe.csv`.[9] Name the new dataframe `anscombe` and display it by calling it alone in the last line of your cell.[10]

Now we want to calculate and print a linear fit and corresponding error for each of the four sets of data in `anscombe`. The columns in `anscombe` should have the names "x1", "y1", "x2", "y2", etc. For convenience, I've created a small loop for you to use on your data. This allows us to perform repetitive operations without worrying about being consistent in our typing from operation to operation.[11]

[9] There are two options here: you will need to change the delimiter to a comma (`delim=","`) for this file, or, since the default delimiter for `pd.read_csv()` is a comma, you can omit the `delim="," ` argument altogether (it is implicitly defined).

[10] Just like we did with the dataframe `data` in the previous section.

[11] If it's correct for the first set of data, it will be correct for each subsequent one

```
# We will make a loop to make this less repetitive (and reduce our chances of making an error)
fits = []
for x, y in (('x1','y1'),('x2','y2'),('x3','y3'),('x4','y4')):
```

```
    fit_coeff, fit_cov = np.polyfit(anscombe[x],
     ↪  anscombe[y], 1, cov=True)
    fit_err = np.sqrt(np.diag(fit_cov))
    fits.append([fit_coeff,fit_err])

# This operation makes my point regarding this data more
 ↪  clear, as you'll see
with np.printoptions(precision=2):
    print("Fit, Std. Err.")
    for x in fits:
        print(x)
```

Clearly, the data sets are different… You can see that from the numbers in the table. So what's going on? To learn more, plot each of the datasets in a separate plot. Discuss your observations with your lab partner, and figure out the best way to deal with information like this.

The key takeaway from this exercise is that you should always plot your data, even if it's only a quick glance on the side of your spreadsheet! This data was taken from an article written by F.J. Anscombe in 1973 arguing for the necessity of graphs in statistical analysis. It is such a valuable dataset that the `seaborn` library is installed with a copy of the data (`sns.load_dataset("anscombe")`).

Column Manipulation and Plotting Spectra

Start a new section titled "Plotting Spectra" and import the file `ftir_data.csv`. Print out the top of the dataframe using the command `df.head()`, where `df` is the name of your dataframe. Notice that we have data in the form "Wavenumber (cm^-1)", "Transmittance (%)".

There is a very simple syntax in Pandas for making a new column:

```
df["new col name"] = func(df["old col name"])
```

In this case, `func()` just represents the transform that we are applying to the original column (multiplication, addition, inversion, etc.). Columns in dataframes will broadcast functions across every item in the column by default, so making a line-by-line transformation (e.g., doubling a column) is as easy as `df["B"] = 2 * df["A"]`. Using this syntax, create new calculated columns in our dataframe so we can plot the same spectrum as "Wavelength" vs "Absorbance (A)".[12]

To get you started converting wavenumber to wavelength, make the following transformation on the "Wavenumber" column[13]:

[12] What are the relevant units for %T and A?

[13] Notice that Python requires the use of "**" for exponentiation. This is because the caret character (^) is often used for other commands. This convention is common in programming languages.

```
df["Wavelength (??)"] = 1/(df["Wavenumber
 ↪  (cm^-1)"]/10**(-2)) * 10**(6)
```

You should be able to derive this equation quickly on your own…[14]

Perform the same steps to take the column for "Transmittance" and transform it into Absorbance values. Remember that Absorbance is $\log_{10}(1/T)$ while Transmittance is I/I_0.

Once you have created the new columns, create two plots. The first should plot wavenumber agains transmittance, while the second will plot wavelength against absorbance. Because these are spectra (data collected in a single experiment), it is appropriate to plot them as line plots, *without* plotting the individual points. Make sure that your plot reflects this.

Make sure your plots are labeled in all the appropriate locations and that your spectra look clean and presentable.

Congratulations, you have the basics of plotting and calculation using Jupyter! You are expected to take this information and build upon it as you write reports through the semester. All of these techniques can be translated to other applications, some of which might be more appropriate for the type of research being performed in a research group. You should be able to transfer your basic skills to any other software with ease, and the tips on presentability are universally applicable.

[14] Hint: label the units and work through the dimensional analysis. Fill in the question marks with the appropriate units.

Alternative Software

While Microsoft Excel may be the most common piece of software for plotting, that doesn't mean it's the best. You have seen that Python (by way of Jupyter) is capable of doing just about anything you can imagine, often in only a few lines. In the future, should you find that Jupyter doesn't meet your needs, here is a list of alternative software for creating scientific plots (I've left out general purpose spreadsheet software).

GraphPad Prism (CUNY site license available) Statistical analysis software for Windows and macOS. Capable of repeated analysis on huge datasets of life/social science data. Falls short in spectral analysis tools.

`https://www.graphpad.com`

Origin by OriginLab (no site license available) Professional data analysis and plotting software for scientists and engineers. Heavy focus on physical sciences and engineering. Only available on Windows.

`https://www.originlab.com`

Igor Pro by WaveMetrics (no site license available) Very similar to Origin, but available for macOS and Windows.

`https://www.wavemetrics.com/products/igorpro`

Mathematica (site license), Matlab (site license), JupyterLab (free) These are really programming environments, though they are designed for use by scientists. Mathematica and Matlab are commercial products, while Jupyter is an open-source option that uses the JUlia, PYThon, and R languages. They all have entry-output interfaces so that live interaction can be done with data. While the learning curve is steep, the payout for learning a programming language will last a lifetime.

`http://www.wolfram.com/mathematica/`

`https://www.mathworks.com/products/matlab.html`

`https://jupyter.org`