Tutorial 1: Data Plotting and Fitting with Python

Python is a high-level, cross platform, general-purpose, programming language that emphasizes code readability. It has a focus on open-source community collaboration and allows you to easily import prebuilt software bundles that facilitate powerful analyses. Because of its ease of use, it is the most widely used language in scientific computing.

Your Environment

The easiest way to get started with a project using python is though "virtual environment's". To get started, open the windows file explorer and open your project folder. Right click inside the folder and select "Open in Terminal". Once the terminal is pulled up use the following command

```
uv venv --python=3.13
```

UV will tell you in the terminal how you can activate your new environment. Use the command

```
.venv\Scripts\activate
```

Now to install the needed python packages, we will install matplotlib (plotting), pandas (data management), scipy (science functions) and numpy (numerical functions). Use the following command.

```
uv pip install matplotlib pandas scipy numpy
```

Notice you can install many packages at once by listing with spaces between each name.

Now install the packages jupyterlab and ezfit.

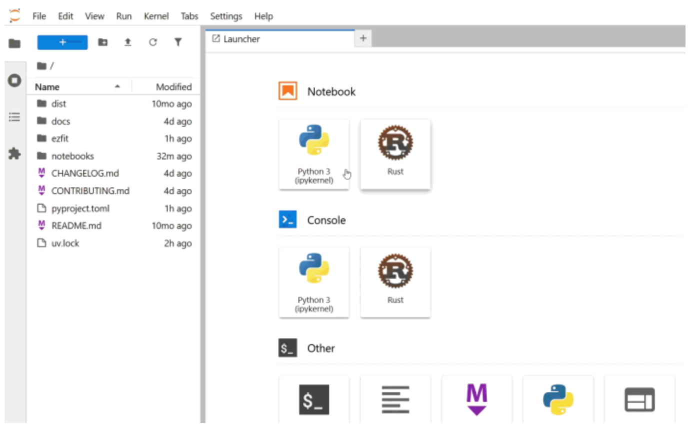
```
**Try this yourself**
```

Jupyter Notebooks

To start off you will want to launch a notebook. You can do this using the following command

```
jupyter lab
```

This may take some time to launch up, but once it does it will launch an instance of Jupyter Lab in your browser. Open a new Jupyter notebook by clicking on Python 3 under the notebook tab. Then rename the notebook.

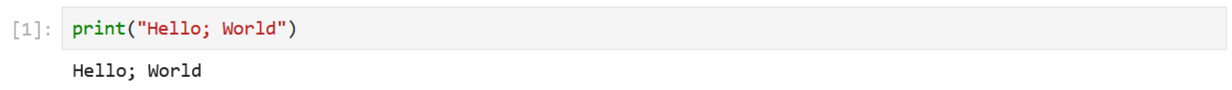


Name	Modified
dist	10mo ago
docs	4d ago
ezfit	1h ago
notebooks	32m ago
CHANGELOG.md	4d ago
CONTRIBUTING.md	4d ago
pyproject.toml	1h ago
README.md	10mo ago
uv.lock	2h ago

Notebooks are made of several code blocks that you can run sequentially. Click the first cell and type

```
print("Hello; World")
```

Run the cell by pressing shift+enter, or by clicking the play button at the top of the notebook. The output will be displayed below the cell



```
[1]: print("Hello; World")

     Hello; World
```

Some basic python

Functions are key to scientific analysis in python. We will make a basic power law function.

$$f(x, A, p) = Ax^p$$

In a new cell below the Hello World cell, make a new function using the following syntax.

```
# This is a comment
# Function declarations use the syntax def my_function_name(my_param):
def power_law(x, a, p): # x is the domain, a. and p are other parameters
    # After the declaration, use tabs to define the function internals
    # We use * for multiplication, and ** for exponentiation
    my_result = a*x**p
    return my_result
```

Once this cell is executed, you can use the function at any point in the future. To see how that works, make a new cell, and define variables for each of the function parameters.

```
a_test = 4
p_test = 2
print(power_law(x=10, a=a_test, p=p_test), power_law(2, a_test, p_test))
```

Execute the cell and make sure that it outputs “400, 16”

Notice you can pass any number of things into the print function.

Notice that I use “x=10”, “a=a_test”, and “p=p_test” in the first, but in the second, I just pass in the parameters in the correct positional order.

Libraries

Let’s grab some prebuilt tools from external python libraries. Import numpy and matplotlib in a new cell using the following command:

```
# import library_name as prefix_shortcut
import numpy as np
import matplotlib.pyplot as plt
```

Execute that cell and we now have numpy functions and objects, along with all the plotting utilities from matplotlib.

Let’s create a vector/array of x-values to evaluate the power_law function over and then plot it. Make a vector using

```
# This array contains only the 5 listed elements
x_sparse = np.array([0, 2, 5, 7, 10])
```

```
# This array contains 100 evenly spaced elements from 0 – 10
x_dense = np.linspace(0, 10, 100)
```

Now evaluate these functions to create y arrays using the following syntax:

```
y_sparse = power_law(x_sparse, a_test, p_test)
y_dense = power_law(x_dense, a_test, p_test)
```

Add some print statements at the end of the cell, to make sure that y_sparse and y_dense arrays look the way you expect them to do. (Try printing y_sparse.shape, and y_dense.shape)

Plotting

Plot the data using the matplotlib plot function. This function takes parameters in the order domain, range, followed by style parameters. Add the following code to a new cell and execute it to make your first plot.

```
plt.plot(x_dense, y_dense)
plt.plot(x_sparse, y_sparse)
```

This will generate a static plot of the two kinds of datasets using the same figure.

To edit plots, we will change the code cell used to generate it and rerun the cell. Now edit the plt.plot(…) function associated with the sparse data to add in the positional style parameter. This comes after the range of your function.

```
plt.plot(x_dense, y_dense)
plt.plot(x_sparse, y_sparse, “o”)
```

Add a legend that labels the dense data by setting the label keyword and editing the plotting cell so that it reads

```
plt.plot(x_dense, y_dense, label = “dense”)
plt.plot(x_sparse, y_sparse, “o”,)
plt.legend() # this function will draw the legend on the plot
```

Now, add a label to the sparse data, do this on your own

The legend can be moved around using the key word loc. To place it in the lower right corner, use the notation

```
plt.legend(loc=”lower right”)
```

To add a grid, use the grid function at the end of the plotting cell.

```
plt.grid(True)
```

Choose a dashed “--”, solid “-”, dotted “:” or dot dashed “-.” style for the grid, and implement it by setting the ls keyword in the grid function. E.g. `plt.grid(True, ls=”-“)`

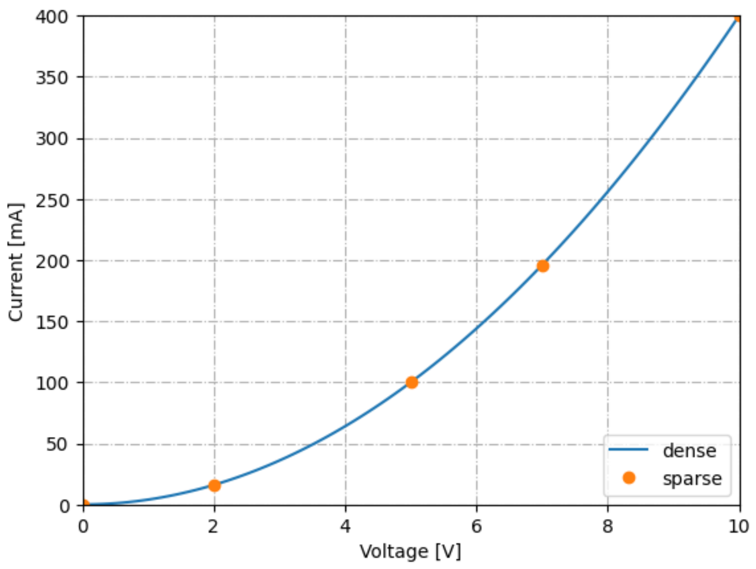
Set your data x-range and y-range using the following functions:

```
plt.xlim(left=0, right=10)
plt.ylim(bottom=0, top=400)
```

Always set your x and y axes labels with both a title and unit by adding the following functions:

```
plt.xlabel(“Voltage [V]”)
plt.ylabel("Current [mA]")
```

If you have properly implemented all the commands above, you should be left with a plot that looks like this, depending on your choice of grid format.



Now try to print this graph at a reasonable size that will fit neatly into your lab book, say ~3x4 inches. Save the figure as a png by adding to the end of your figure generating notebook cell the following final function.

```
plt.savefig("My first python figure")
```

Make sure you print a copy and tape it into your lab notebook

It is easy to google syntax for stylizing your plots.

Loading Data

Make a new cell to import the pandas data science library. Import pandas using the nickname/prefix pd and execute the cell.

```
import pandas as pd
```

Read in a table or DataFrame using the read_csv function, and assign this object to a variable:

```
df_current = pd.read_csv("current_data.csv")
df_voltage = pd.read_csv("voltage_data.csv")
```

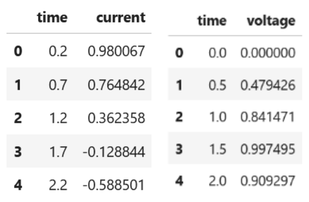
Now, we want to inspect this data and see what it looks like. DataFrames can be very large (there is a contest about trillion-record DataFrames), so let's look at just the first few rows, or the "head" of the DataFrame. Execute the following at the end of that cell.

```
print(df_current.head())
print(df_voltage.head())
```

For some objects such as DataFrames, you may prefer to use the display function to generate a nicely formatted version of the print results. Try this out with the two data frames in a new cell.

```
display(df_current.head())
display(df_voltage.head())
```

You should get a table that looks like this



	time	current
0	0.2	0.980067
1	0.7	0.764842
2	1.2	0.362358
3	1.7	-0.128844
4	2.2	-0.588501

	time	voltage
0	0.0	0.000000
1	0.5	0.479426
2	1.0	0.841471
3	1.5	0.997495
4	2.0	0.909297

Let’s consider a contrived problem where we want to calculate the power from this dataset. This will take multiplying the current times the voltage at every time. But in this instance, the time stamps do not line up! We need to interpolate the data onto a common axis so that we can do math on it.

Let’s choose the voltage data frame’s timestamp as our preferred axis. Using the NumPy interpolation function, calculate the interpolated current values.

```
time = df_voltage[“time”] # use df[“col”] to grab a specific column
common_current = np.interp(
        time,
        df_current[“time”],
        df_current[“current”]
)
```

Now create a new DataFrame containing the interpolated data

```
df_current_interp = pd.DataFrame({        # use new lines to keep things readable
        “time”: common_t,                 # if you are wrapped in () you
        “current”: common_current         # are good to make new lines!
        })
```

Display the dataset table to make sure they line up.

```
**Try this on your own**
```

Now that the datasets use a common time step, we can merge the two tables together “on” the time column.

```
df_merged = pd.merge(df_voltage, df_current_interp, on=”time”)
```

Display this table to see what we just did.

Now we can do math on the table to calculate the power column. This is done in probably the way you would expect to do it.

```
df_merged[“power”] = df_merged[“current”] * df_merged["voltage”]
```

Once each of these steps has been completed and you have a merged DataFrame, it is easy to plot the results. Using the plot “method” with the following notation will generate a plot, i.e.

```
df_merged.plot(x=”time”, style=[“s-“, “o-“, “*-“])
```

This will use the time column as the domain for every other trace and then iterate over the style list to prescribe a style for each trace. You can combine this with all the other visualization tools we have built so far.

Visual Analysis

Let's load in some more test experimental data and see how it works.

```
df_fitting = pd.read_csv("powerlaw.csv")
```

Display the head of this dataset to see what the data looks like.

Plot the data using the pandas scatterplot plotting method.

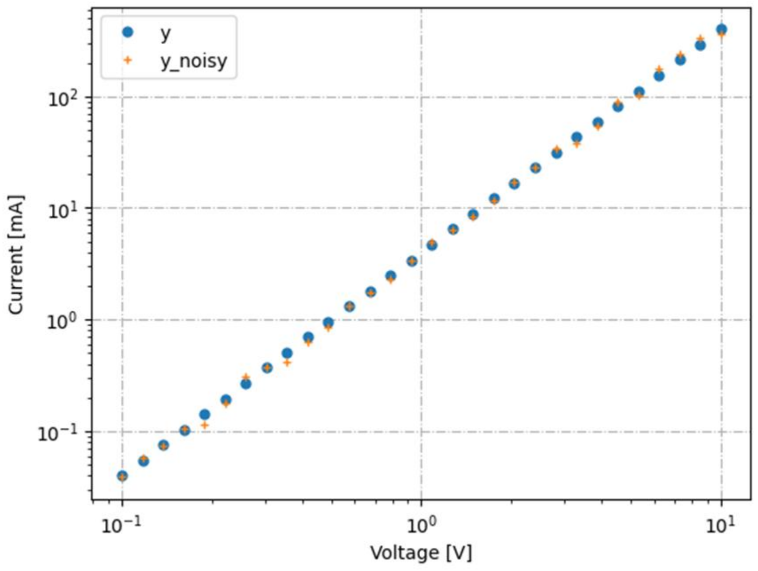
```
df_data.plot(x="x", style=["o", "+"], ms=5)
```

Next, we need to learn how to plot data on a log-log scale. This is done by passing loglog=True into the df.plot method.

```
df_data.plot(x="x", style=["o", "+"], ms=5, loglog=True)
plt.grid(True) # use the style you like for ls
…
```

Use the plotting syntax we used previously to label the x-axis Voltage y-axis Current, giving both correct units.

Notice that both axes became logarithmically displayed. Also notice what happened to the curvy trace: It became straight!!



This is because our data is a power-law function. This is an important rule to remember. We essentially took a log of both sides of our equation. Notice what happens mathematically when doing this to an arbitrary power-law function:

$$y = Ax^P$$

$$\log y = \log A + P \log x$$

Notice this is exactly the following:

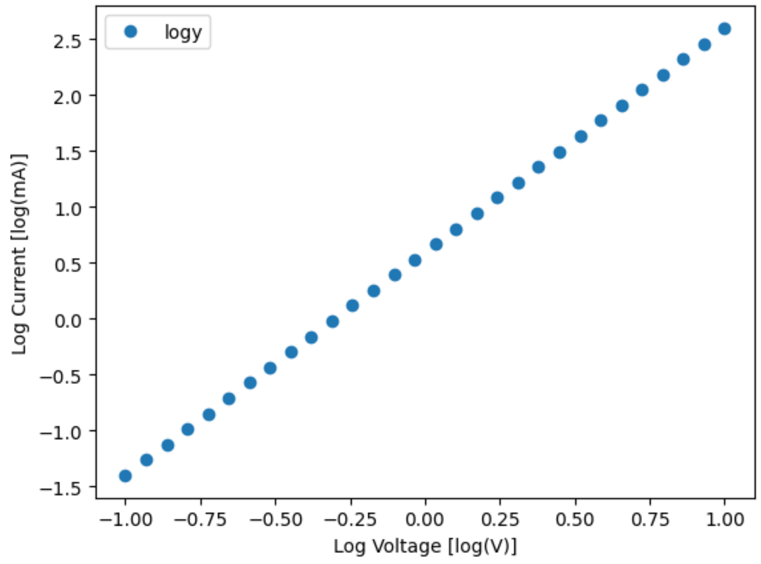
$$y' = mx' + b$$

where $b = \log A$ and $m = P$. where $b = \log A$ and $m = P$. (Also $y' = \log y$ and $x' = \log x$.) Notice the trivial relationship between the slope of the linear function and the power of the power law.

Print this graph out and tape it into your lab notebook. Title it “Log-Log Display of the Data”

Now we will do this explicitly, by manually computing the log of the x and y data. This looks like

```
df_data[“logx”] = np.log10(df_data[“x”])
df_data[“logy”] – np.log10(df_data[“y”])
df_data.plot(x=”logx”, y=”logy”, style=”o”)
plt.ylabel(“Log Current [log(mA)]”)
plt.xlabel(“Log Voltage [log(V)]”)
```



Note that we use np.log10(…) rather than np.log(…) since log resolves as the natural log of the dataset.

You can see that where the points go haven’t changed at all from our earlier rendering, only the axis information. For example, on the x-axis, what used to go from 0.1 to 100 now goes from -1 to 1, which makes sense in terms of powers (0.1 equals 10 raised to the -1 power).

Print out this graph and tape it below the earlier one. Title it “Log of the Data”

Fitting

Fitting models to data is one of the core aspects of experimental physics. Thus, you're going to have to fit a lot of data in this class to extract important physical parameters.

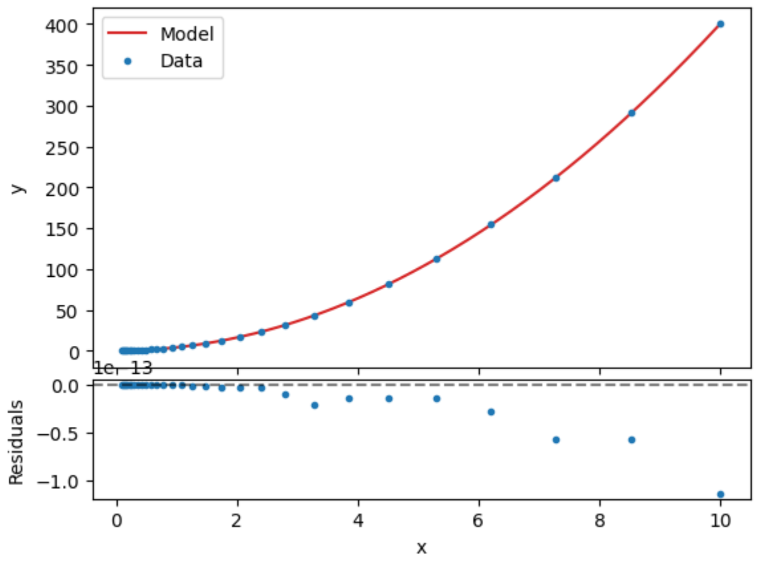
To do this, we need to import ezfit into the notebook. Let’s not give it a nickname, so import and execute the following:

```
import ezfit
```

This adds a new functionality to every DataFrame object that will facilitate fitting. Let’s fit the data we loaded up using the power_law function we defined before. To use ezfit, your function needs to be defined to take the domain (x) as the first argument of the function.

Let’s make a guess for this function.

```
fit_result, *axes = df_data.fit(
        power_law,              # name of your fit function
        x=”x”,                  # experimental domain
        y=”y”,                  # experimental data
        a={“value”:0.4},        # parameter specification
        b={“value":2},          # parameter specification
)
```



The output of this function is ‘fit_result’, containing the fitted model (in this case a power law), and ‘*axes’. The axes use the * in front of it to unpack the remaining function outputs into a single variable containing the top and bottom axes in this plot, though editing these uses a slightly different notation than defined above.

To look at the fit results, print our fit_results in a new cell. This is very important and shows the final fit parameters retrieved from the fits.

```
power_law
a : (value = 4.000000000000001 ± 1.6e-15, bounds = (-inf, inf))
p : (value = 2.0 ± 1.9e-16, bounds = (-inf, inf))
…
```

These are the final fit values and their uncertainty from the fit. Notice it got them perfectly and the uncertainties are TINY! The uncertainty of a measurement or analysis like this determines the significant figure (sig fig) of the result, so this says $p = 2.00000000000000000 \pm 0.00000000000000019$. By the way, 1e-16 is generally considered “machine precision”. Uncertainty values should be rounded to one sig fig unless that digit is one, in which cases you should quote two sig figs as shown here.

Let’s make the data more realistic by adding ~10% noise to it. The y_noisy column is the same data with just this noise added. Fit this data by altering the previous fit command. Now the output will yield more realistic uncertainties:

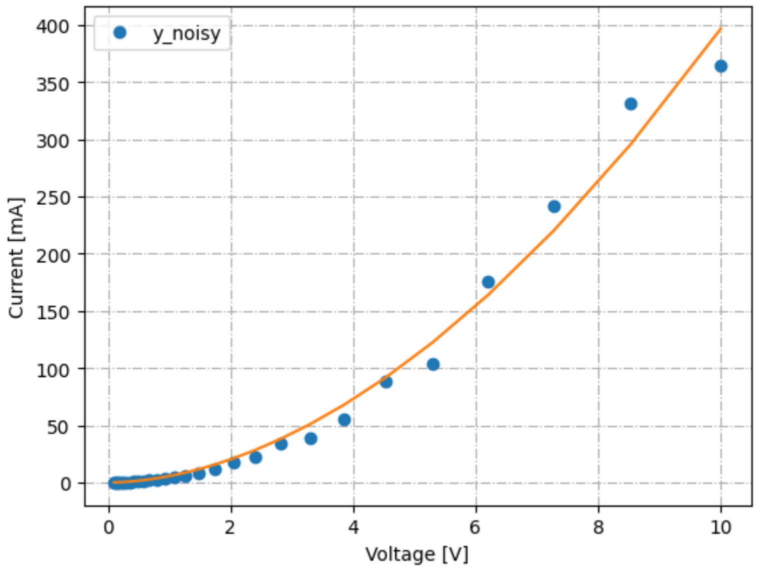
```
power_law
a : (value = 5.6 ± 0.86, bounds = (-inf, inf))
p : (value = 1.85 ± 0.073, bounds = (-inf, inf))
…
```

For proper reporting of p, you would write $p = 1.85 \pm 0.7$ or $p = 1.85(7)$ to show that the final decimal place has an uncertainty of 7.

To make your own plot of the results, you can treat the output fit_results object like it's own function and plot the data along with this result.

```
df_data.plot(x="x", y="y_noisy", style="o")
plt.plot(df_data["x"], fit_results(df_data["x"]))
…
```

Combining this with the other styles we have built so far, we can see the following results



Print and tape this graph into your logbook. Title it "My First Fit"

We can also fit a portion of the dataset to another model. Say we want to fit the data for x>5 to a linear model. We first define a function for $f(x, m, b) = mx + b$

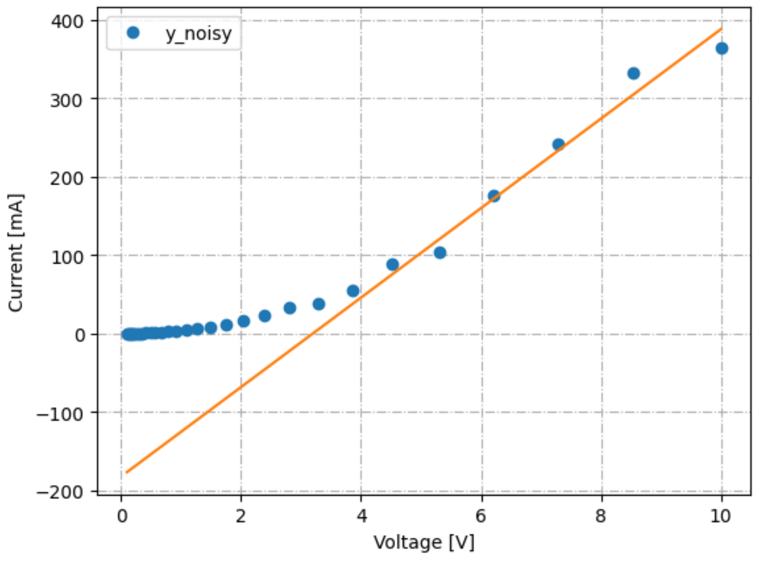
```
def line(x, m, b): # remember to put the domain first!
    return m*x+b
```

Now we fit the line to the dataset

… continued

```
line_results, *axs = df_data.query(
        “x>5”
).fit(
        line,
        x=”x”,
        y=”y_noisy”
)
```

This will generate a plot of only the data for x>5. If we want to see the model compared to the data we did not use for fitting, we can do that using df_data.plot(…) along with the visualization tools used for the last plot.



Print and tape this graph into your logbook. Title it “Fit Over a Partial Range”

Fitting Diagnostics

Sometimes our fit results hide demons about the results. Let’s fit the column y_noisy_2. Your results should look something like this.

```
power_law
a : (value = 0.49 ± 0.062, bounds = (-inf, inf))
p : (value = 3.36 ± 0.057, bounds = (-inf, inf))
covariance:
[[ 0.0038 -0.0035]
 [-0.0035  0.0032]]
correlation:
[[ 1.    -0.998]
 [-0.998 1.    ]]
```

Notice that the error on 'a' is large (relative uncertainty of about 15%!). This is because the parameters of a power-law are highly correlated as you can see in the correlation matrix printed just below the fit values shown above. This matrix tells you if any parameter is correlated with another one in your model. The diagonal is always perfectly 1 because the parameter is correlated with itself. However, the off-diagonal elements tell us here that p is almost perfectly inversely correlated with 99.2% correlation! You can see the same results in the off-diagonal entries of the covariance matrix telling a similar story. This means that if you simultaneously lower p and raise a, you get about the same chisq error. Thus, the algorithm can't tell very well what the parameter values should be.

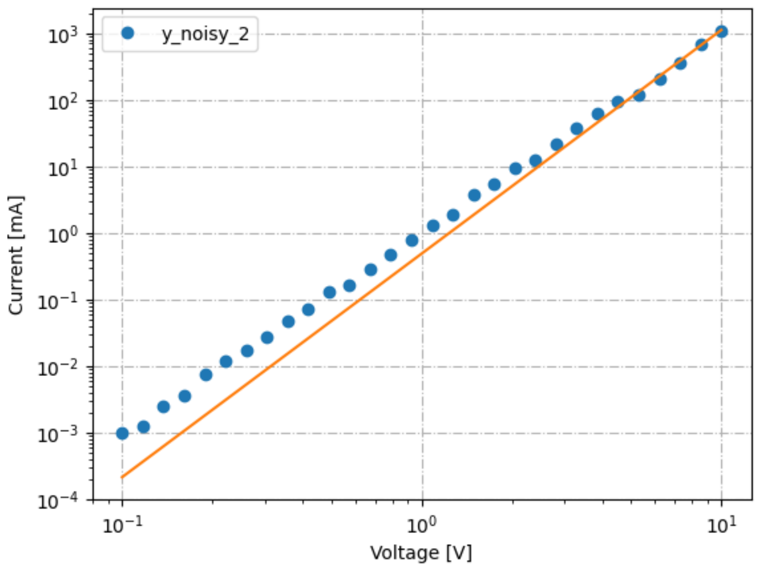
Indeed, the power obtained from this fit is WRONG. We can say it is definitively wrong because I know that I made the data with a power of 3, and ***the value of 3 does not fall inside the uncertainty range of our fit***. In fact, we can say that our fit value is off by

$$error\ [sigma] = \frac{error}{uncertainty} = \frac{3.36 - 3}{0.06} = 6.0\sigma$$

We are six standard deviations off of the expected value!

But the fit looks good, right?

Let's check out the log scale of this fit result. Remember you can do this using the loglog=True command.



Both the data and fit look like lines, but boy is that slope off! This is because the errors on the left side of the graph are tiny in value compared to those on the right, so the computer effectively ignores the left half of the data!

To fix this, let's fit the log of the data to a line:

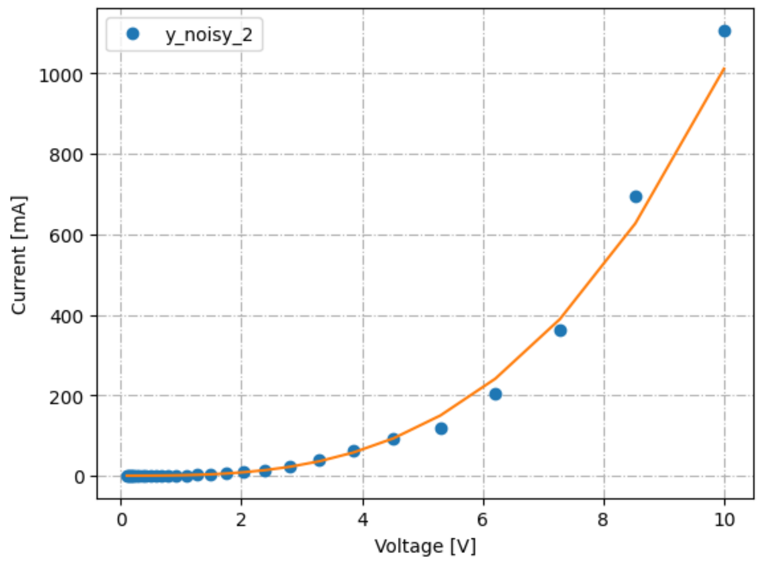
```
df_data["logy_2"] = np.log10(df_data["y_noisy_2"])
fit_result_linear, *axes = df_data.fit(
        line,                   # name of your fit function
        x="logx",               # experimental domain
        y=" logy_2",            # experimental data
)
```

Looking at the fit results, we need to be careful about the log log mapping.

```
line
m : (value = 3.01 ± 0.014, bounds = (-inf, inf))
b : (value = -0.005 ± 0.0081, bounds = (-inf, inf))
χ2: inf
reduced χ2: inf
covariance:
[[ 0.0002 -0.    ]
 [-0.      0.0001]]
correlation:
[[ 1. -0.]
 [-0.  1.]]
```

Now you have $m = 3.010(14)$ which is MUCH closer to the expected value (still not within one standard deviation, but that's ok as this is not real data). Also, notice that the correlation between parameters b and m equals 0.000. Because linear fits are completely uncorrelated, physicists often try to convert their data into a linear function before the fit the key parameters to make sure they obtained accurate fit parameters with no correlations

Now convert these fit parameters into A and P for the power law and update those variables in Gnuplot. (Go back to the math that we did before if you don't remember the conversion between each variable.) Plot the new fit and data together to get the following graph:



Print and tape this graph into your logbook. Title it "My first Log Fit." Also, write the values of A and P on your graph.

Not bad huh?! We'll do fits like these for your analysis in many of the lab experiments this semester.

*****Have a TA check that you've finished this tutorial before starting the next part.*****

Good Resources

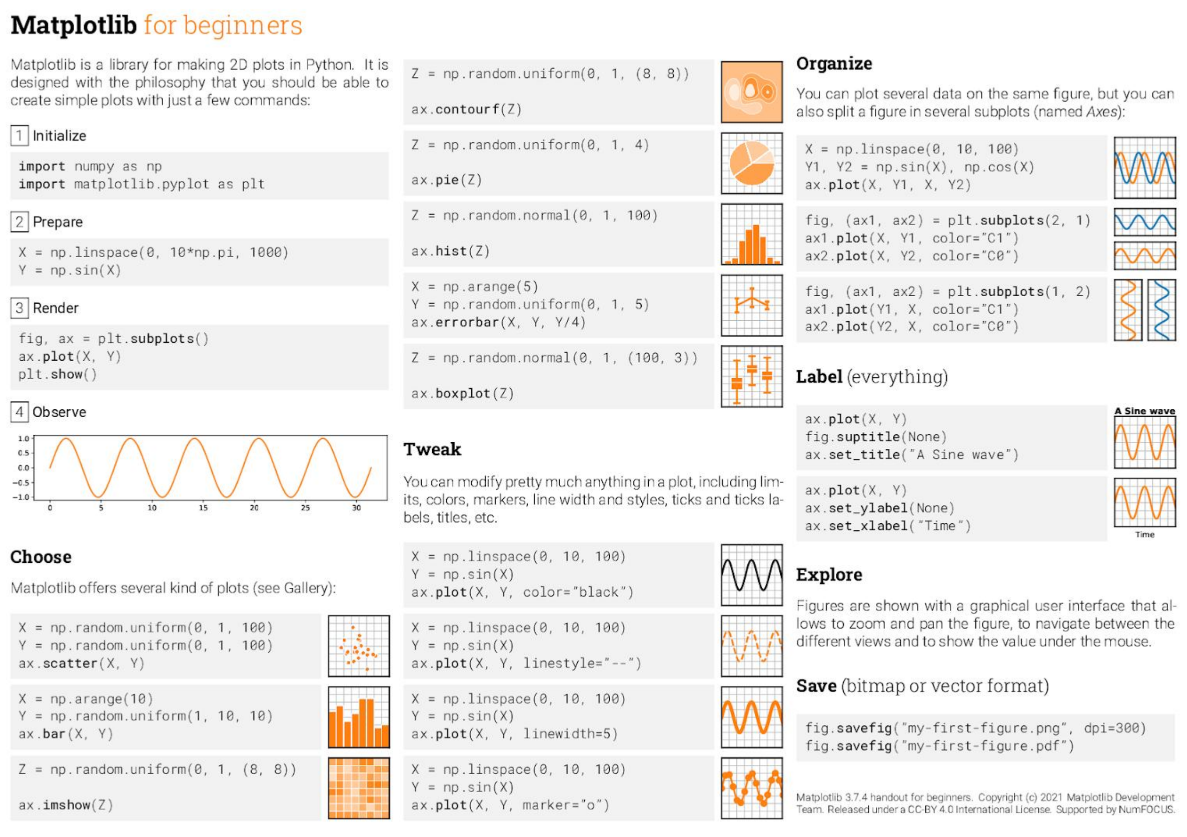
As mentioned above this is all in python, and you should spend some time learning how to do basic things in python. W3 schools is a great resource for learning some of these basics.

- W3 Python
- W3 Numpy
- W3 Matplotlib
- W3 Pandas

Visualization

When it comes to making very good and high quality visualizations in python, there are several options as well. Matplotlib is not the only method, but it is the basis for most plotting libraries. Another popular one is seaborn, specialized for statistical visualization. But there are also popular libraries for interactive visualization, and other fun things. There is too much to cover here.

- Matplotlib's quickstart
- Matplotlib for beginners



Matplotlib for beginners

Matplotlib is a library for making 2D plots in Python. It is designed with the philosophy that you should be able to create simple plots with just a few commands:

1 Initialize

```
import numpy as np
import matplotlib.pyplot as plt
```

2 Prepare

```
X = np.linspace(0, 10*np.pi, 1000)
Y = np.sin(X)
```

3 Render

```
fig, ax = plt.subplots()
ax.plot(X, Y)
plt.show()
```

4 Observe

Choose

Matplotlib offers several kind of plots (see Gallery):

```
X = np.random.uniform(0, 1, 100)
Y = np.random.uniform(0, 1, 100)
ax.scatter(X, Y)
```

```
X = np.arange(10)
Y = np.random.uniform(1, 10, 10)
ax.bar(X, Y)
```

```
Z = np.random.uniform(0, 1, (8, 8))

ax.imshow(Z)
```

```
Z = np.random.uniform(0, 1, (8, 8))

ax.contourf(Z)
```

```
Z = np.random.uniform(0, 1, 4)

ax.pie(Z)
```

```
Z = np.random.normal(0, 1, 100)

ax.hist(Z)
```

```
X = np.arange(5)
Y = np.random.uniform(0, 1, 5)
ax.errorbar(X, Y, Y/4)
```

```
Z = np.random.normal(0, 1, (100, 3))

ax.boxplot(Z)
```

Tweak

You can modify pretty much anything in a plot, including limits, colors, markers, line width and styles, ticks and ticks labels, titles, etc.

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, color="black")
```

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linestyle="--")
```

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, linewidth=5)
```

```
X = np.linspace(0, 10, 100)
Y = np.sin(X)
ax.plot(X, Y, marker="o")
```

Organize

You can plot several data on the same figure, but you can also split a figure in several subplots (named *Axes*):

```
X = np.linspace(0, 10, 100)
Y1, Y2 = np.sin(X), np.cos(X)
ax.plot(X, Y1, X, Y2)
```

```
fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.plot(X, Y1, color="C1")
ax2.plot(X, Y2, color="C0")
```

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(Y1, X, color="C1")
ax2.plot(Y2, X, color="C0")
```

Label (everything)

```
ax.plot(X, Y)
fig.suptitle(None)
ax.set_title("A Sine wave")
```

```
ax.plot(X, Y)
ax.set_ylabel(None)
ax.set_xlabel("Time")
```

Explore

Figures are shown with a graphical user interface that allows to zoom and pan the figure, to navigate between the different views and to show the value under the mouse.

Save (bitmap or vector format)

```
fig.savefig("my-first-figure.png", dpi=300)
fig.savefig("my-first-figure.pdf")
```

Matplotlib 3.7.4 handout for beginners. Copyright (c) 2021 Matplotlib Development Team. Released under a CC-BY 4.0 International License. Supported by NumFOCUS.

- Matplotlib cheat sheet
- Matplotlib gallery
- Seaborn - matplotlib all dressed up
- Seaborn gallery

Pandas

You will learn more about working with data as the class progresses, but pandas is a very powerful tool for working with any type of data. To learn more about how to use it, see the following tutorials

- Getting started with Pandas
- Coming from excel
- Pandas cheat sheet