Arrays and Plotting

The foundation for everything else in this section: holding measurements in NumPy arrays, reading and writing data files, and making plots you can put in a report. Assumes you can write a function — see Python Basics.

Two libraries do nearly all the work. **NumPy** holds your measurements and does arithmetic on all of them at once. **Matplotlib** draws them. Every other page in this section builds on both.

All examples assume:

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

1 Why arrays instead of lists

A Python list can hold numbers, but you can't do arithmetic on it — `[1, 2, 3] * 2` repeats the list rather than doubling the values. A NumPy array behaves the way you'd want a column of measurements to behave:

```
voltages = np.array([0.10, 0.52, 1.18, 2.41])

voltages * 2          # array([0.2 , 1.04, 2.36, 4.82])
voltages / 50         # every element divided by 50 — currents through a 50 Ω load
voltages - 0.03       # subtract an offset from every point
```

Operating on a whole array at once is called *vectorising*. It's shorter than a loop, much faster on large datasets, and — the part that matters most — it reads like the physics.

2 Making arrays

```
np.array([1.0, 2.0, 3.0])     # from a list you already have
np.linspace(0, 10, 100)       # 100 points from 0 to 10, inclusive of both ends
np.arange(0, 10, 0.5)         # 0 to 10 in steps of 0.5, excluding the endpoint
np.zeros(50)                  # 50 zeros — useful to fill in as you measure
np.ones(50)
```

`linspace` and `arange` cover different needs, and the difference catches people out: `linspace` takes a *number of points*, `arange` takes a *step size* and stops before the endpoint. When you want a smooth curve for plotting a model, `linspace` is almost always the one you want.

3 Arithmetic and maths functions

NumPy's maths functions apply element-wise:

```
x = np.linspace(0, 2 * np.pi, 200)

np.sin(x)
np.exp(-x)
np.sqrt(x)
```

```
np.log10(x + 1)      # log10 for decibels and Bode plots; np.log is natural log
np.abs(x - np.pi)
```

Use `np.sin`, not the `math` module's `sin` — the NumPy versions work on whole arrays, the `math` ones only on single numbers.

4 Summary statistics

```
readings = np.array([4.98, 5.02, 4.95, 5.07, 5.01])

readings.mean()          # 5.006
readings.std(ddof=1)     # sample standard deviation
readings.min(), readings.max()
len(readings)
```

`ddof=1` for measured data

By default `np.std` divides by *N*, which describes the spread of the numbers you have. For a sample of measurements estimating an underlying quantity, you want `ddof=1` — dividing by *N* − 1. It's the difference between describing your data and estimating the population it came from, and with small *N* the gap is not subtle: for five points, the default underestimates by about 11%.

The standard error of the mean — the uncertainty on the average — is then `readings.std(ddof=1) / np.sqrt(len(readings))`.

5 Selecting parts of an array

Indexing and slicing work like lists:

```
t = np.linspace(0, 1, 1000)

t[0]        # first element
t[-1]       # last
t[100:200]  # a slice
```

You can also select by condition, which is how you filter data:

```
signal = np.array([0.1, 5.2, 0.3, 4.9, 0.2])

signal > 1                # array([False, True, False, True, False])
signal[signal > 1]        # array([5.2, 4.9]) — just the values above 1
t[signal > 1]             # the *times* at which that happened
```

That last line is the useful pattern: build a mask from one array, apply it to another of the same length to pull out matching points.

6 Reading a data file

Most instruments write CSV or plain columns of numbers.

```
# Simple numeric CSV, no header
data = np.loadtxt("run3.csv", delimiter=",")

# With a header row to skip
data = np.loadtxt("run3.csv", delimiter=",", skiprows=1)

time = data[:, 0]      # first column  — ":" means every row
voltage = data[:, 1]   # second column
```

`np.loadtxt` fails on missing values or non-numeric fields. When a file is messier than it should be, `np.genfromtxt` is the tolerant version — it fills gaps with `nan` rather than raising:

```
data = np.genfromtxt("messy.csv", delimiter=",", skip_header=1)
clean = data[~np.isnan(data).any(axis=1)]   # drop rows containing a nan
```

Unpacking straight into named variables

`unpack=True` transposes the result, so you can name your columns as you read them — clearer than remembering that column 2 was the photodiode:

```
time, voltage = np.loadtxt("run3.csv", delimiter=",", skiprows=1, unpack=True)
```

7 Writing a data file

```
np.savetxt(
    "processed.csv",
    np.column_stack((time, voltage)),
    delimiter=",",
    header="time_s,voltage_V",
    comments="",          # without this, the header line starts with "# "
)
```

`np.column_stack` turns separate 1-D arrays into columns of one 2-D array. Putting units in the header costs nothing and saves the argument about what the second column meant.

8 A first plot

```
plt.figure(figsize=(7, 4.5))
plt.plot(time, voltage)
plt.xlabel("Time (s)")
plt.ylabel("Voltage (V)")
plt.grid(True, alpha=0.3)
plt.show()
```

`plt.show()` displays it. In a notebook the figure appears under the cell; from a script it opens a window.

> **Axis labels are not optional**
>
> Every axis gets a quantity and a unit. A plot without them can't be interpreted by anyone, including you next week, and it will cost you marks in a report. This is the single most common thing missing from student figures.

9 Data points, not lines

Measurements are discrete points. Drawing them joined by lines implies you know what happened in between, which you don't — reserve lines for models and fits.

```
plt.plot(current, voltage, "o", label="Measured")        # points
plt.plot(i_model, v_model, "-", label="Ohm's law fit")   # line
plt.xlabel("Current (A)")
plt.ylabel("Voltage (V)")
plt.legend()
```

The format string is terse: `"o"` circles, `"s"` squares, `"."` small dots, `"-"` solid line, `"--"` dashed. Combine them (`"o--"`) if you really do want both.

10 Error bars

```
plt.errorbar(
    current, voltage,
    yerr=v_uncertainty,      # also xerr= if the x values have uncertainty
    fmt="o",                 # marker style; without this you get connecting lines
    capsize=3,
    label="Measured",
)
```

`fmt="o"` matters — the default draws lines between points, which is rarely what you want for data.

11 Several plots together

For anything beyond one quick plot, use the `subplots` interface. It gives you explicit figure and axes objects instead of relying on which plot is "current", which is what stops figures from bleeding into each other:

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(7, 7), sharex=True)

ax1.plot(freq, gain, "o")
ax1.set_ylabel("Gain (dB)")
ax1.grid(True, alpha=0.3)

ax2.plot(freq, phase, "s")
ax2.set_xlabel("Frequency (Hz)")
```

```
ax2.set_ylabel("Phase (deg)")
ax2.grid(True, alpha=0.3)

fig.tight_layout()
```

Note the method names change: `ax.set_xlabel(...)` on an axes object where you'd write `plt.xlabel(...)` on the current figure. `sharex=True` links the axes so they zoom together and only the bottom one is labelled.

12 Log axes

Anything spanning decades — frequency response, decay, power laws — belongs on log axes:

```
ax.set_xscale("log")      # log frequency
ax.set_yscale("log")      # log-log, for power laws
```

On a log-log plot a power law is a straight line whose slope is the exponent, which makes it a quick way to check one. Log axes can't show zero or negative values; those points silently vanish.

13 Saving a figure

```
fig.savefig("gain_vs_frequency.png", dpi=300, bbox_inches="tight")
fig.savefig("gain_vs_frequency.pdf", bbox_inches="tight")
```

Use PNG at `dpi=300` for documents, PDF when you want it to stay sharp at any zoom. `bbox_inches="tight"` trims the surrounding whitespace, which otherwise crops your axis labels when you insert the figure.

Save before `plt.show()` — showing a figure can clear it, leaving you with a blank file.

14 Common surprises

Symptom	Cause
`ValueError: operands could not be broadcast together`	Two arrays of different lengths in one expression. Check `len()` on both — usually a mismatched slice
Plot window empty, or nothing appears	Missing `plt.show()`, or you saved after showing
Figures accumulating on top of each other	Reusing the current figure. Use `fig, ax = plt.subplots()` per plot
Integer division giving zeros	An array read as `int`. Force it with `data.astype(float)`
`nan` spreading through a result	One bad value in the input. `np.isnan(data).sum()` counts them

Back to Python for Lab Work