

# Matplotlib for Publication

Rebecca Pitts



LUND  
UNIVERSITY

INAISS



LUND  
UNIVERSITY

# Part 1. Basics of Matplotlib



# Introducing Matplotlib

Matplotlib is *the* standard Python library for visualizing data for print or digital media.

- Figures may be static, interactive, animated, and/or embedded in a Jupyter notebook
- Plots can be 2D or 3D, single or tiled as part of a larger figure
- Cartesian, polar, and many other coordinate systems supported, including map projections and sky coordinates
- [Excellent documentation with many, MANY tutorials](#)



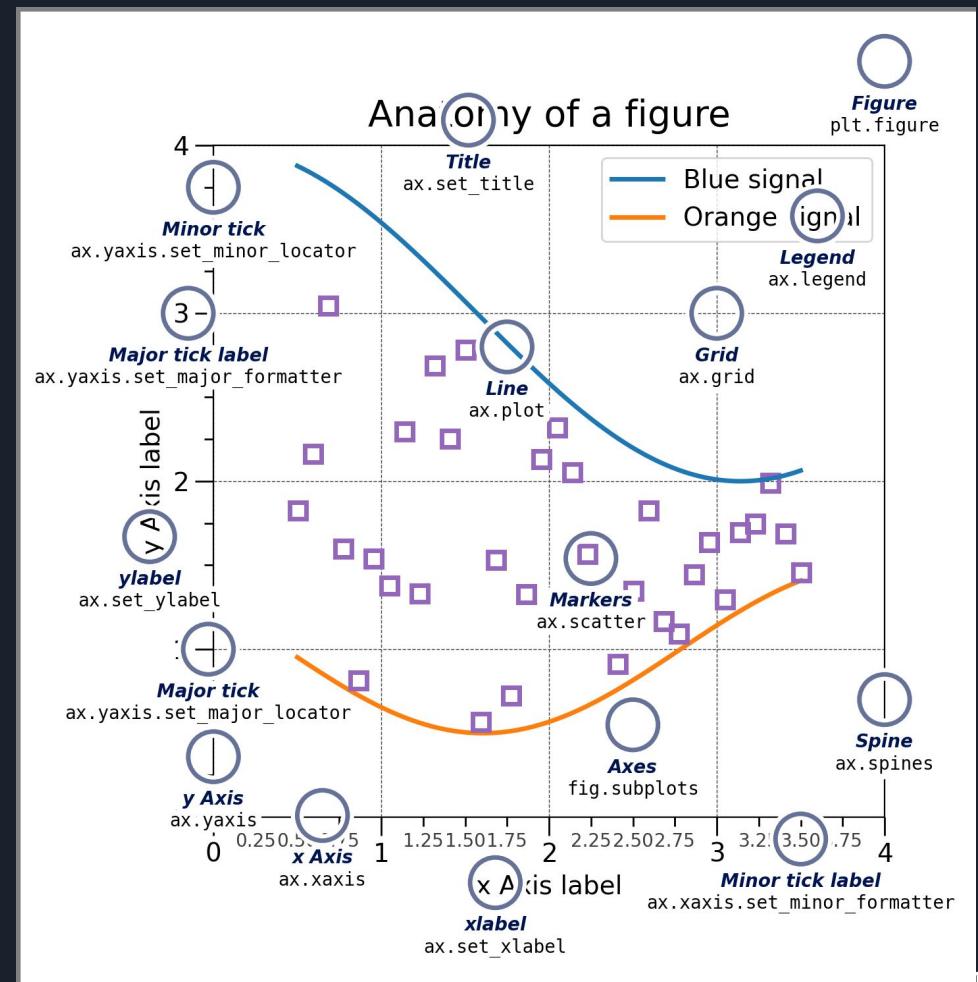
# Basic Terms

Names of plot elements→

- **Figure (fig)** = frame & all (sub)plots inside (may have >1 set of axes)
- **Axis instance (ax)** = one pair (or trio) of axes, their labels, & all data enclosed

Terms to know for functions:

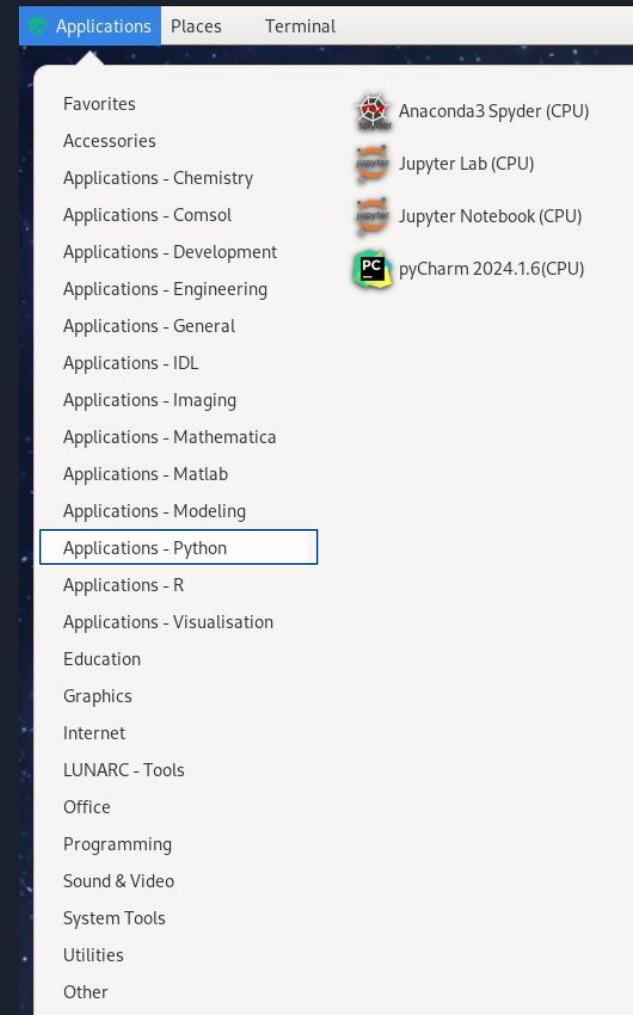
- **args** = positional **arguments**; usually mandatory
- **kwags** = **keyword arguments**; usually optional



# Run Matplotlib On-Demand

LUNARC HPC Desktop (COSMOS):  
recommended to use Spyder or Jupyter  
from the On-Demand Applications menu  
to use Matplotlib interactively.

Jupyter modules default to latest  
Anaconda3 version. Spyder currently  
uses Anaconda3 with Python/3.11.7.  
Exercises were written with  
Python/3.11.5.





# Load & Run Matplotlib Module

Command line or SLURM script: modules to load depend on whether you use Anaconda or PyPi

- **Anaconda:** `ml` Anaconda3 loads everything needed (but environments can mess with your `.bashrc` & disrupt access to other modules)
- **PyPi:** GCC determines version of Matplotlib & all dependencies
  - Use `ml spider <dependency>` to check Python or other dependency versions to see which GCC you need
  - Then `ml GCC/1x.x.x matplotlib` loads with all dependencies.
  - Matplotlib/3.9.2 has a malfunctioning display backend on COSMOS





LUND  
UNIVERSITY

# Code along to get started!

Project: lu2025-7-37 on COSMOS



# Pyplot: the Workhorse

The minimum working matplotlib code will virtually always require you to import `matplotlib.pyplot` (and usually also NumPy)

- Provides MATLAB-like functions to create or alter plots and their components
- In older code you may see `matplotlib.pylab`, which is now deprecated and risks overwriting some built-in functions.

**Standard call:** `from matplotlib import pyplot as plt`  
If in Jupyter, add `%matplotlib inline` after import line



# Minimal working examples

**Single data series, no typesetting  
(implicit application programming  
interface, or API):**

```
from matplotlib import pyplot as plt

plt.plot( x, y, 'k-' )
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

**Format string may include single-letter  
color specifier & linestyle specifier ('-', ':',  
'-.', '--'). Will show more options later.**

**Any data you want to add to, typeset,  
or include as a subplot (explicit API):**

```
from matplotlib import pyplot as plt

fig, ax = plt.subplots(**kwargs)
ax.plot( x, y, 'k-' )
ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()
```

**Formatting commands for axis objects  
typically have `set_` in front**



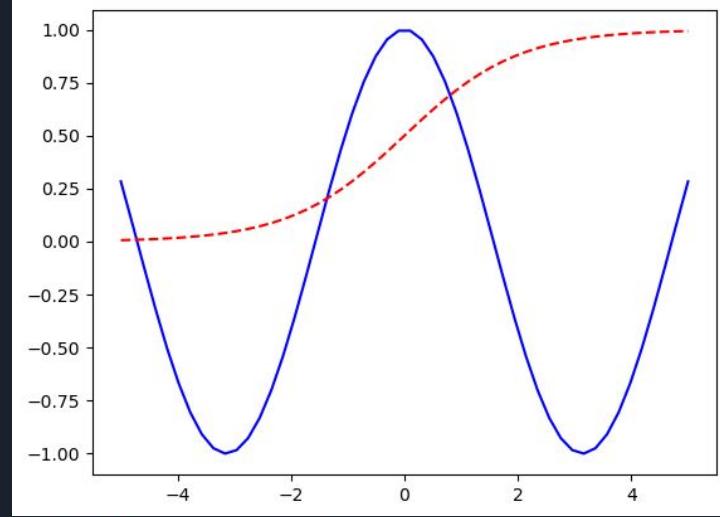
# Now you try!

Let `x = np.linspace(-5, 5, 50)`.

On the same axes, plot

- `np.cos(x)` as a solid blue line, &
- `1/(1-np.exp(-x))` as a dashed red line

Use the *explicit API*. We will come back to this plot later to add to it.



```
import numpy as np
import matplotlib.pyplot as plt
# import matplotlib as mpl
#mpl.use('TkAgg') #use at command line
#mpl.use('Qt5Agg') #if PyQt5 loaded

x = np.linspace(-5,5)
fig, ax = plt.subplots()
ax.plot(x, np.cos(x), 'b-')
ax.plot(x, 1/(1+np.exp(-x)), 'r-')
plt.show()
```



# Labels, Legends, & Titles

Set x & y axis labels & plot title:

```
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('[Sub]plot Title')
```

for the *explicit* API, or

```
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Plot Title')
```

for the *implicit* API.

To title a figure with subplots (always explicit API), use `fig.suptitle()`

Most plotting functions include a `label` kwarg to pass to `ax.legend()`.

- Control legend position with `loc` (in plot area) or `bbox_to_anchor` (in or out of plot area; more exact)
- Some plotting functions (e.g. `.bar()`) take array of labels as 1st arg instead of `label` kwarg
- Can explicitly pass lines/data handles & labels for e.g. multiple sets of contours



# Placement of Text, Legends, etc.

2 functions for adding text to plots at arbitrary points: `.annotate()` & `.text()`

- `.text()` is base function; it only adds & formats text
- `.annotate()` adds & formats connectors (e.g., arrows) from points to text
- Positions for both are in data coordinates by default unless `ax.transAxes` is used

`ax.transAxes` switches from data coordinates to axes-relative coordinates where x & y axes are unit length & (0,0) is lower left corner.

Legend placement via `bbox_to_anchor` uses unit-axes coordinates by default.

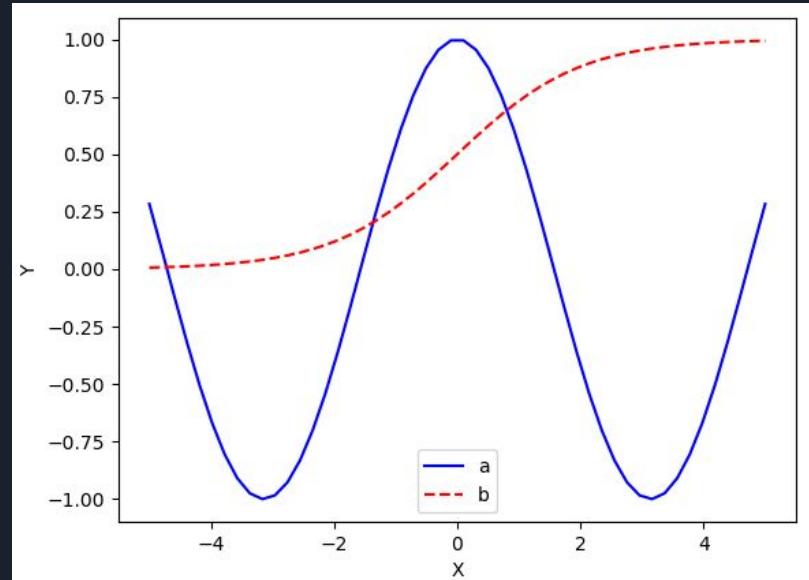
- Legend position `loc` can be set to integer or 2-word string like 'lower left' or 'upper center'
- Whole-figure legends can use `loc='outside <va> <ha>'`



# Now you try!

Return to your earlier plotting code. Label the horizontal & vertical axes “X” & “Y”, respectively. Label the blue solid curve “a” & the red dashed curve “b”. Add a legend to the plot in the lower center.

\*If you plotted both lines in the same `plot()` command earlier, you will have to split them up.



```
x = np.linspace(-5,5)
fig, ax = plt.subplots()
ax.plot(x, np.cos(x), 'b-', label='a')
ax.plot(x, 1/(1+np.exp(-x)), 'r--',label='b')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.legend(loc='lower center')
plt.show()
```

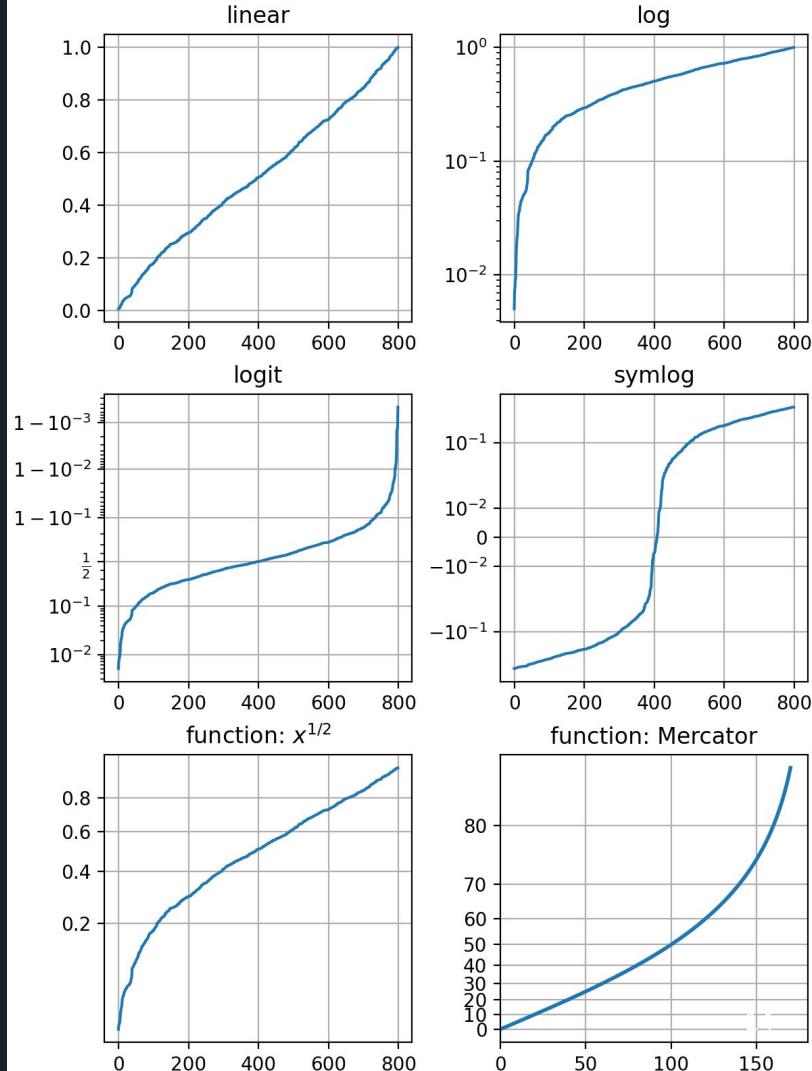


# Axis Scales

Some plotting functions have `xscale` & `yscale` kwargs. For others, you can set `ax.set_xscale()`/`ax.set_yscale()`:

- If scale arg is ‘function’, must define both forward & reverse transformations & pass as tuple of function names
- Axes scales can be shared across subplots

Usually automatic tick spacing is fine. If not, refer to [these examples](#)



# Subplots

Axes created with `plt.subplots()` are **iterable** if `nrows` &/or `ncols` are `>1`. These can be set to share x & y axes labels, & `.subplots_adjust()` can adjust or remove column/row spacing:

```
fig, axes = plt.subplots(nrows=3, ncols=2, sharex=True,  
                        sharey=True)  
plt.subplots_adjust(hspace=0, wspace=0)  
for i in range(3):  
    for j in range(2):  
        k = (i%3)*2+j  
        axes[i][j]=plt.plot(x,y[k])  
        if j==0:  
            axes[i][j].set_ylabel('Y')  
        if i==2:  
            axes[i][j].set_xlabel('X')
```

`plt.subplot()` (singular) is more tedious but allows separate projections for each plot.

Example:

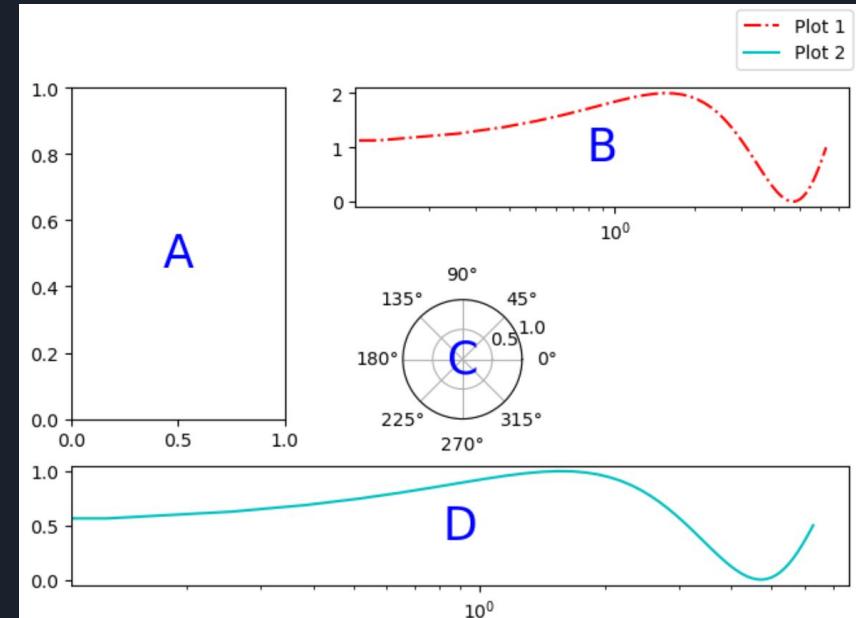
```
fig = plt.figure(figsize=(8,4))  
ax1 = plt.subplot(121)  
ax1.plot(x, 3+3*np.sin(x), 'b-')  
ax1.set_xlabel('x [rads]')  
ax1.set_ylabel('y')  
ax2 = plt.subplot(122, projection= 'polar')  
ax2.plot(x, 3+3*np.sin(x), 'b-')
```



# Subplot Mosaics

The API for subplot mosaics lets you lay out subplots so some plots span multiple rows or columns ("." denotes gaps). Example:

```
fig, axd = plt.subplot_mosaic(  
    """  
    ABB  
    AC.  
    DDD  
    """ , layout="constrained",  
    per_subplot_kw={"C": {"projection": "polar"},  
                   ('B', 'D'): {'xscale': 'log'}})  
  
for k, ax in axd.items():  
    ax.text(0.5, 0.5, k, transform=ax.transAxes,  
            ha="center", va="center", color="b",  
            fontsize=25)  
axd['B'].plot(x, 1+np.sin(x), 'r-.',  
              label='Plot 1')  
axd['D'].plot(x, 0.5+0.5*np.sin(x), 'c-',  
              label='Plot 2')  
fig.legend(loc='outside upper right')  
  
This layout can also be rendered  
"ABB;AC.;DDD"
```



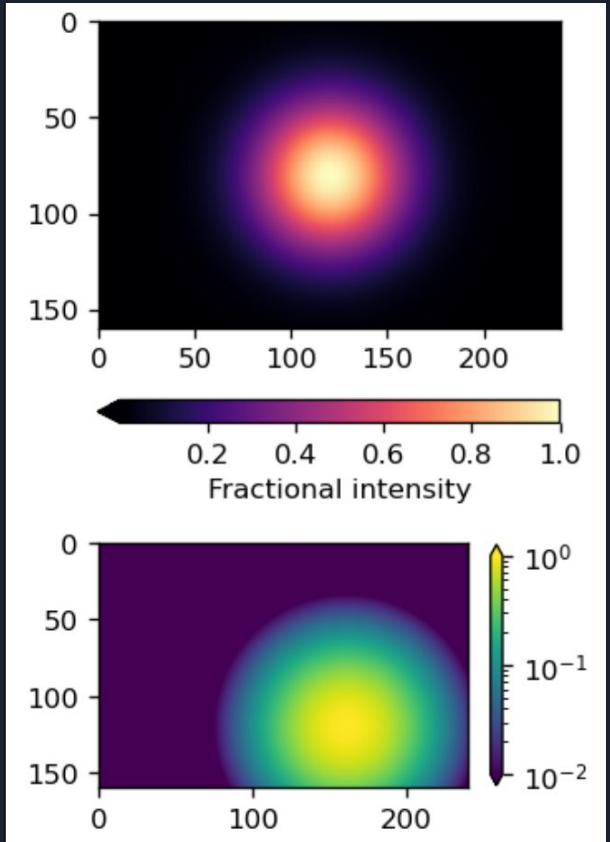
# Colorbars

Colorbars are methods of `Figure`, not `Axes`. Axis objects must be passed to `colorbar()` explicitly, & first arg must be a mappable (a plot; [check docs](#)).

Example:

```
fig, (ax1, ax2) = plt.subplots(nrows=2,
                               figsize=[3, 6],
                               dpi=120)
plt.subplots_adjust(hspace=-0.1)
img1 = ax1.imshow(Z1, cmap='magma')
img2 = ax2.imshow(Z2, norm='log', vmin=0.01)
cbar1 = fig.colorbar(img1, ax=ax1, extend='min',
                     orientation='horizontal')
cbar1.set_label('Fractional intensity')

cbar2 = fig.colorbar(img2, ax=ax2, shrink=0.5,
                     extend='both')
```



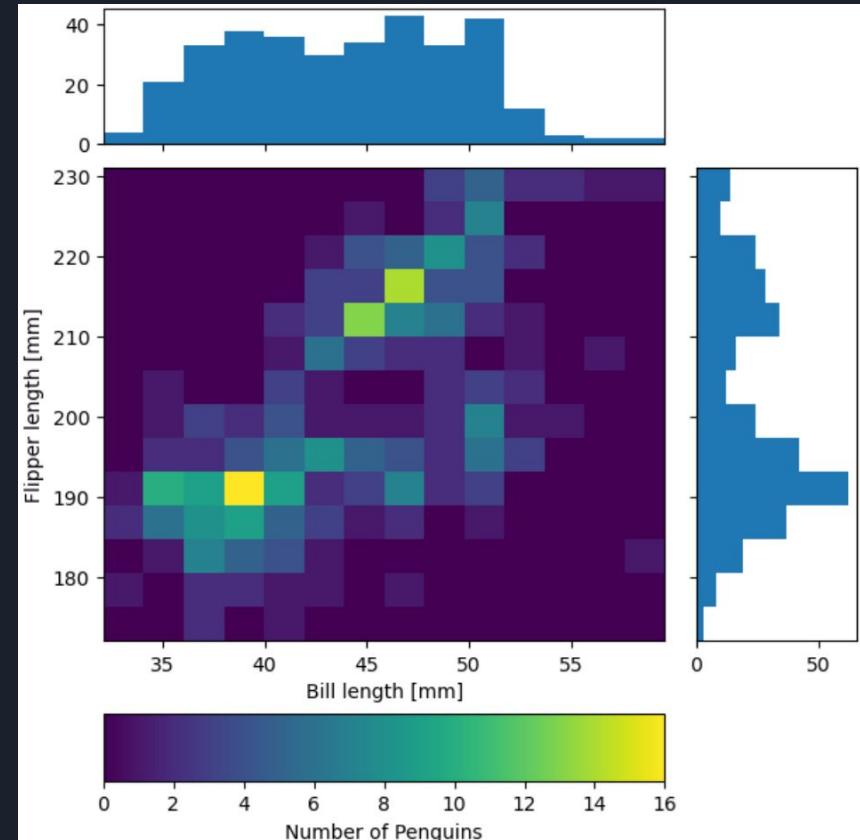
```

def corner_2p(x, y, ax2d, ax_histx, ax_histy):
    # no labels
    ax_histx.tick_params(axis="x", labelbottom=False)
    ax_histy.tick_params(axis="y", labelleft=False)

    nbins = int(np.ceil(2*len(x)**(1/3))) # Rice binning rule
    # the central 2D histogram:
    n,xb,yb,img = ax2d.hist2d(x, y, bins = [nbins,nbins])
    #use x- & y-bins from 2D histogram to align them
    ax_histx.hist(x, bins=xb)
    ax_histy.hist(y, bins=yb, orientation='horizontal')
    ax_histx.sharex(ax2d)
    ax_histy.sharey(ax2d)
    return img

fig, axd = plt.subplot_mosaic("a.;Bc;d.", layout="constrained",
                             height_ratios=[1, 3.5, 0.5],
                             width_ratios=[3.5, 1],
                             figsize=(6,6), dpi=120)
jointhist = corner_2p(penguins.dropna()['bill_length_mm'],
                      penguins.dropna()['flipper_length_mm'],
                      axd['B'], axd['a'], axd['c'])
axd['B'].set_xlabel('Bill length [mm]')
axd['B'].set_ylabel('Flipper length [mm]')
cb = fig.colorbar(jointhist,cax=axd['d'],
                  orientation='horizontal')
cb.set_label('Number of Penguins')

```



Example: 1- & 2D histograms on a scaled subplot mosaic with a separate colorbar axis.  
Note that `hist2d()` returns 3 parameters before the mappable. [Adapted from this](#).



# Figure Output Formats

Matplotlib's default figure size is 6.4" x 4.8" (16.26 x 12.19 cm) with 100 dpi resolution. Jupyter Lab's inline backend defaults to 80 dpi (usually too low).

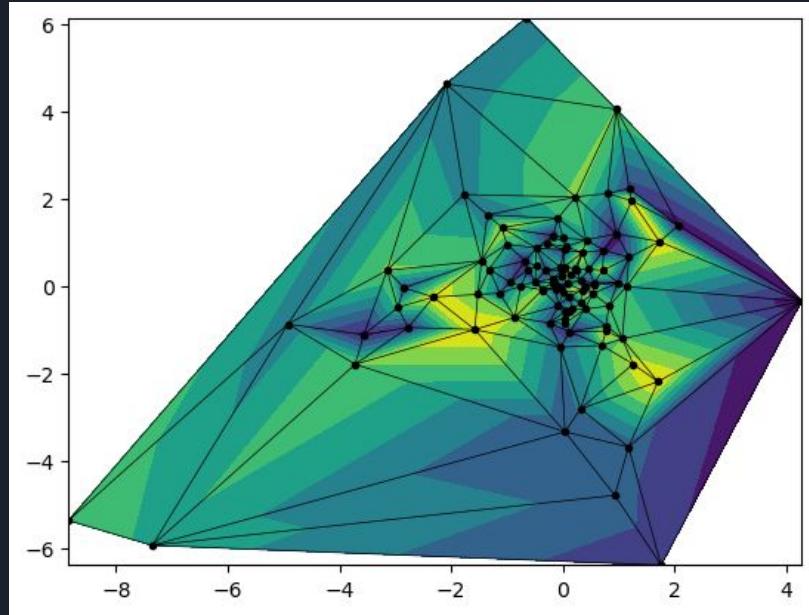
- Adjust figure resolution, size, & aspect ratio with `dpi` & `figsize` kwargs (`figsize` *in inches*) in `plt.figure()` or `plt.subplots()` at runtime, or `plt.savefig()` or `plt.imsave()` (RGB[A] images) for only the saved figure
- **Recommended output format: PDF (widely supported & efficient).**
  - PNG (default) is large in memory, JPEG is lossy, & (E)PS requires special viewers & lacks transparency support.



Full plot type browser with documentation here:

[https://matplotlib.org/stable/  
plot\\_types/index.html](https://matplotlib.org/stable/plot_types/index.html)

# Note on Irregularly Gridded Data



tricontour, & tricontourf on one plot

triplot, tripcolor, tricontour, & tricontourf are all based on triangular meshes ( $x,y$ ) with values at vertices ( $z$ )

- Only contour or shade the mesh if points are spatially correlated; otherwise, indicate  $z$  (value at  $x,y$ ) with marker size/color or something else.
- **Not to be used for contouring/shading by point density!** If you want that, you want a histogram or KDE plot, & will have to interpolate to a regular grid.

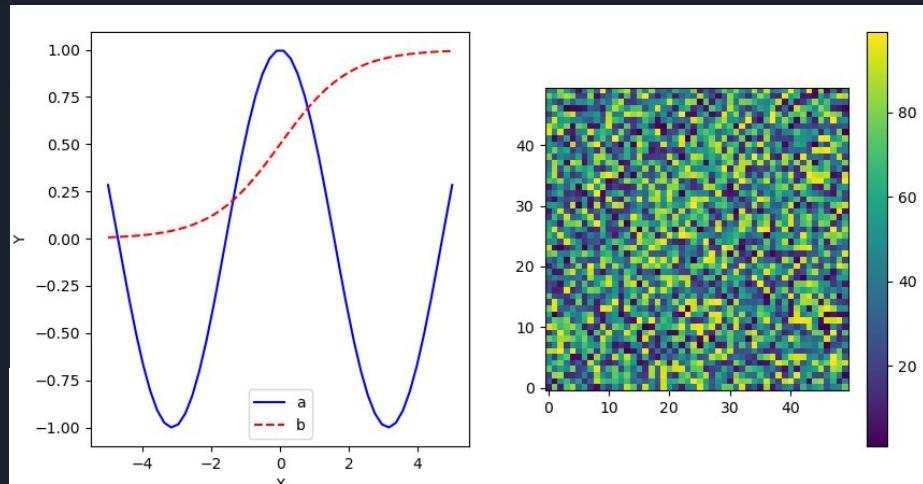


# Now you try!

Return to your earlier plot code. Let  $Z = \text{np.random.randint}(1, 100, (50,50))$

Plot  $Z$  with `imshow()` on a second set of axes, with the bottom left corner at  $(0,0)$ , & add a colorbar. Adjust `figsize` to keep the subplots roughly square.

```
fig, axes = plt.subplots(ncols=2, figsize=(10,5))
axes[0].plot(x, np.cos(x), 'b-', label='a')
axes[0].plot(x, 1/(1+np.exp(-x)), 'r--', label='b')
axes[0].set_xlabel('X')
axes[0].set_ylabel('Y')
axes[0].legend(loc='lower center')
Z = np.random.randint(1, 100, (50,50))
img = axes[1].imshow(Z, origin='lower')
cb = plt.colorbar(img)
plt.show()
```





LUND  
UNIVERSITY

# Questions?

To get exercises:

wget

<https://github.com/rpitts/Matplotlib4Publication/raw/refs/heads/main/mpl4pub-exercises.zip>



LUND  
UNIVERSITY

# Part 2. Formatting & Accessibility



Best practices are fairly consistent  
across plot types

Clear &  
distinct data  
sets

Depends strongly on plot  
type & data

Legible &  
correct labels,  
symbols, etc.

Suitable axes  
scales, grids,  
& ticks

Suitable  
colors or  
color maps

## Elements of Plot Accessibility

# Figure sizes

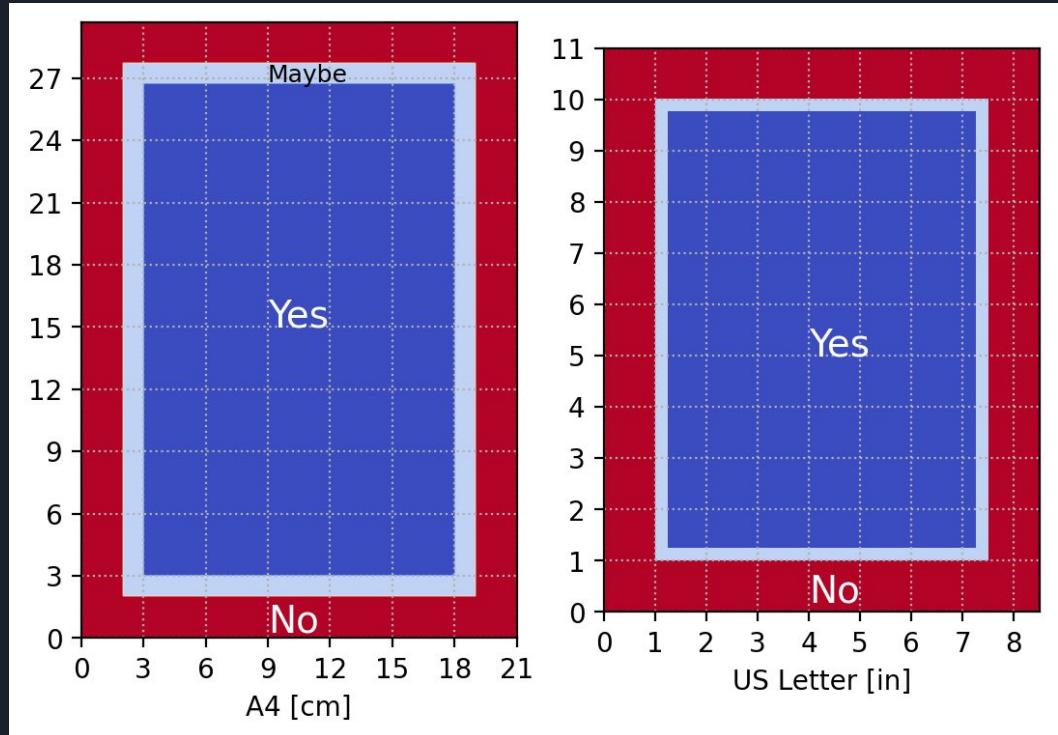
Maximum figure size is determined by the journal's printable area, especially the width along the minor axis.

Format	Dimensions	Margins	Printable area
A4	21 × 29.7 cm 8.3" × 11.7"	2-3 cm 0.8-1.2"	15-17 × 23.7-25.7 cm 6-6.5" × 9.3-10.1"
US Letter	8.5" × 11" 21.6 × 27.9 cm	1-1.25" 2.5-3.2 cm	6-6.5" × 8.5-9" 15-17 × 21.6-22.9 cm



# Sub-Figure Sizes

- Minimum subfigure size varies, but try to avoid panels <1" or <3 cm on a side (labels get too small)
- Color bars, axis labels, & captions take up space, too, so roughly 4x6 subplots max



# Matplotlib & LaTeX

Need to know LaTeX basics to typeset plot text & symbols as they appear in journal text:

- LaTeX expressions (even just italicized variables) must be bracketed by '\$ \$' (US dollar signs); e.g., "sin(\$A\$)" to render  $\sin(A)$
- Syntax of most LaTeX functions to modify characters: "\$\backslash function{char}\$"
  - Superscript (" $^$ ") & subscript (" $_$ ") operators without "{}" only work on the first character that follows.
- Conventionally italicized (math-text) letters indicate variables. Use `\mathrm{char}` to un-italicize letters in, e.g. super- or subscripts
- See Matplotlib's [mathtext](#) docs for symbols, operators, & other commands





# Matplotlib & LaTeX (Cont.)

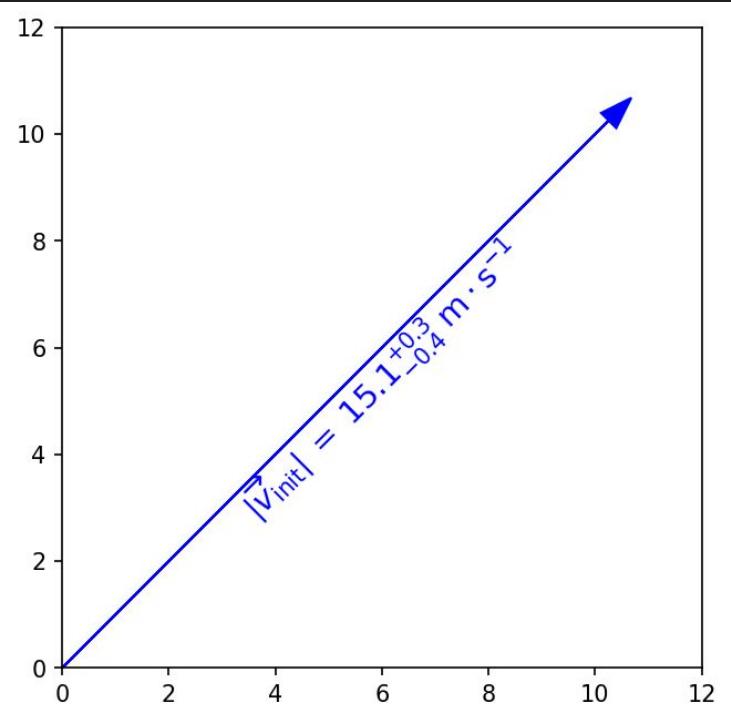
- String insertion ('{}').format()) creates conflict between Matplotlib & LaTeX → must double all {} associated with LaTeX commands
- Most Greek letters &  $\pm/\mp$  symbols require the input string to have `r` before the first quote mark (e.g., `r'\$\backslash beta$'` prints  $\beta$ )
- Operators with special meanings in LaTeX or Python (e.g. %, \$, [], &, ~, #) must have backslash(es) (\ for LaTeX commands, \\ for Python operators) to render as text.
- Spaces inside \$\$ are not rendered: use \, or \,: or \\; to insert them
  - \! removes unwanted space (e.g. after > or <)





```
v_init=15.1
error_arr=[-0.4,0.3]
fig,ax=plt.subplots(dpi=150,figsize=(5,5))
ax.set_aspect('equal') #arrowheads will slant if axes are not equal
ax.arrow(0,0,10.68,10.68,length_includes_head=True,color='b',
         head_width=0.4)
ax.text(6, 5.4, r"\vec{v}_{init} = 15.1^{+0.3}_{-0.4} m\cdot s^{-1}"
        .format(v_init,*error_arr),
        ha='center',va='center',rotation=45.,size=14, color='b')

ax.set_xlim(0,12)
ax.set_ylim(0,12)
plt.show()
```



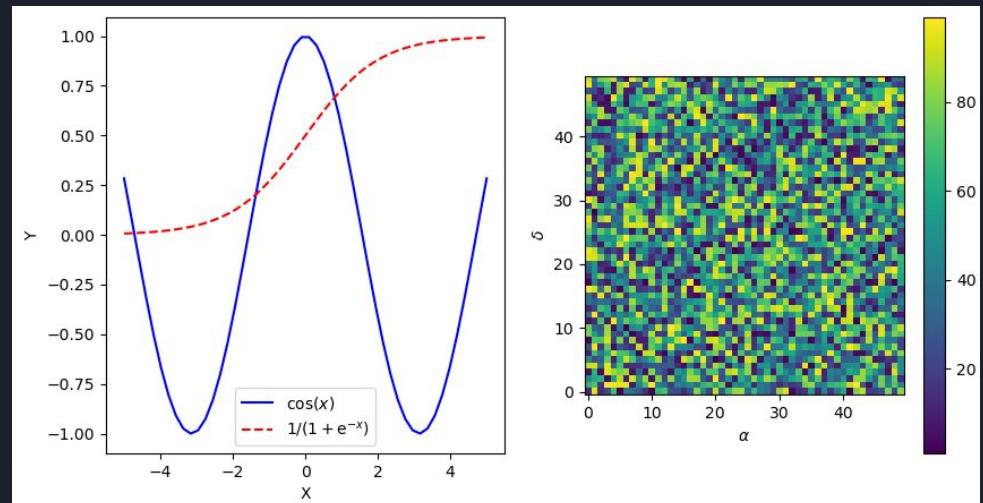
No, you cannot break the string over multiple lines with a backslash, even outside of a TeX expression. A backslash with nothing after it is treated like \n but leaves the '\ in the output.

# Now You Try!

Return to your plotting code from the first lecture. Replace the “a” & “b” line labels with mathtext typeset equations (in-line fractions are fine). For the image, label the horizontal axis “ $\alpha$ ” (Greek alpha) & the vertical axis “ $\delta$ ” (delta).

```
fig, axes = plt.subplots(ncols=2, figsize=(10,5))
axes[0].plot(x, np.cos(x), 'b-', label=r'cos($x$)')
axes[0].plot(x, 1/(1+np.exp(-x)), 'r--',
              label=r'$1/(1+\mathrm{e}^{-x})$')
axes[0].set_xlabel('X', fontsize=14)
axes[0].set_ylabel('Y', fontsize=14)
axes[0].legend(loc='lower center')

Z = np.random.randint(1, 100, (50,50))
img = axes[1].imshow(Z, origin='lower')
axes[1].set_xlabel(r'$\alpha$', fontsize=14)
axes[1].set_ylabel(r'$\delta$', fontsize=14)
cb = plt.colorbar(img)
plt.show()
```



# Axis Ticks

Matplotlib's default is outward-pointing ticks on the left & bottom.  
Ticks on all sides are preferred for readability.

- To add outward ticks on top and right:

```
ax.tick_params(axis='both', which='both', top=True, right=True)
```

- To set inward ticks all the way around:

```
ax.xaxis.set_tick_params(direction='in', which='both')
```

```
ax.yaxis.set_tick_params(direction='in', which='both')
```

```
ax.tick_params(axis='both', which='both', top=True, right=True)
```

Remember: **Be consistent throughout your article!**





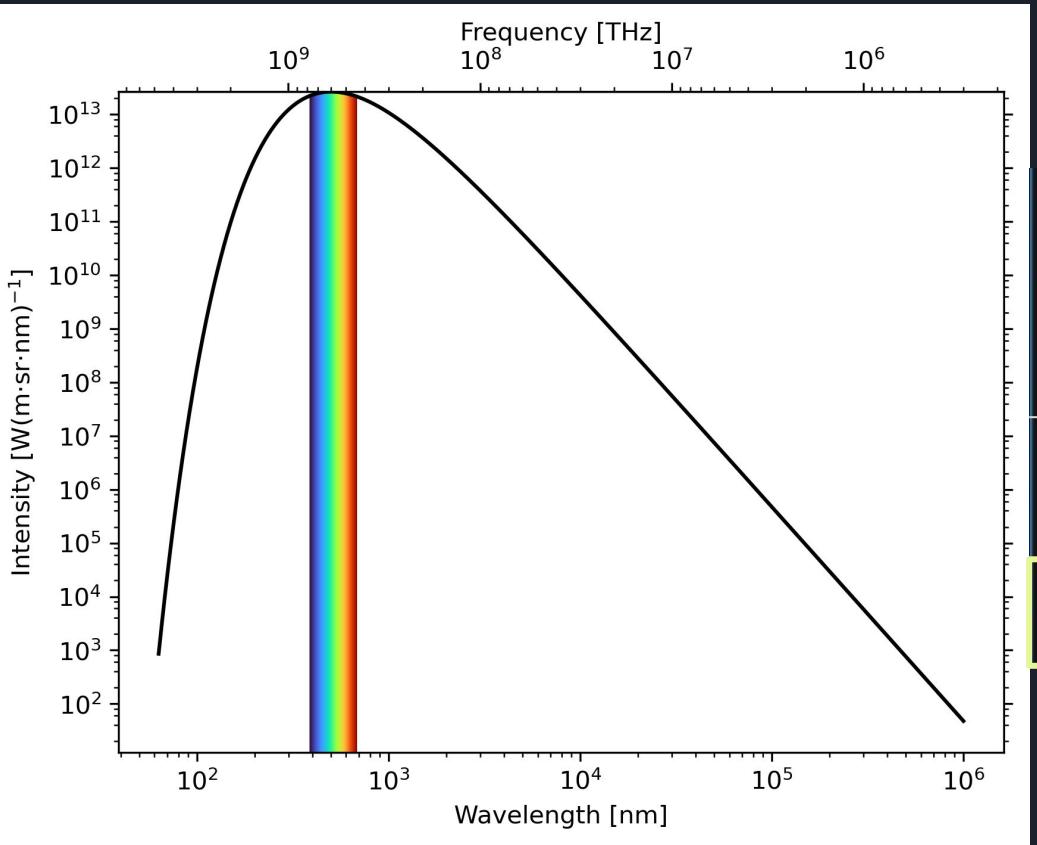
# Tick Labels & Locators

May need to modify or manually set tick locators & labels if you want:

- Units with special formats or symbols (dates, fractions, money, etc.)
- Categorical variables (e.g. countries, species, sex, etc.)
- Axis tick labels centered between major ticks
- Secondary axes that are transformations of the primary axes
- Custom or power-law axis scales
- Log-, symlog-, or asinh scaling with labels on every decade & visible minor ticks over  $>7$  decades

on one or more axes, or if you want any of the above on a colorbar.





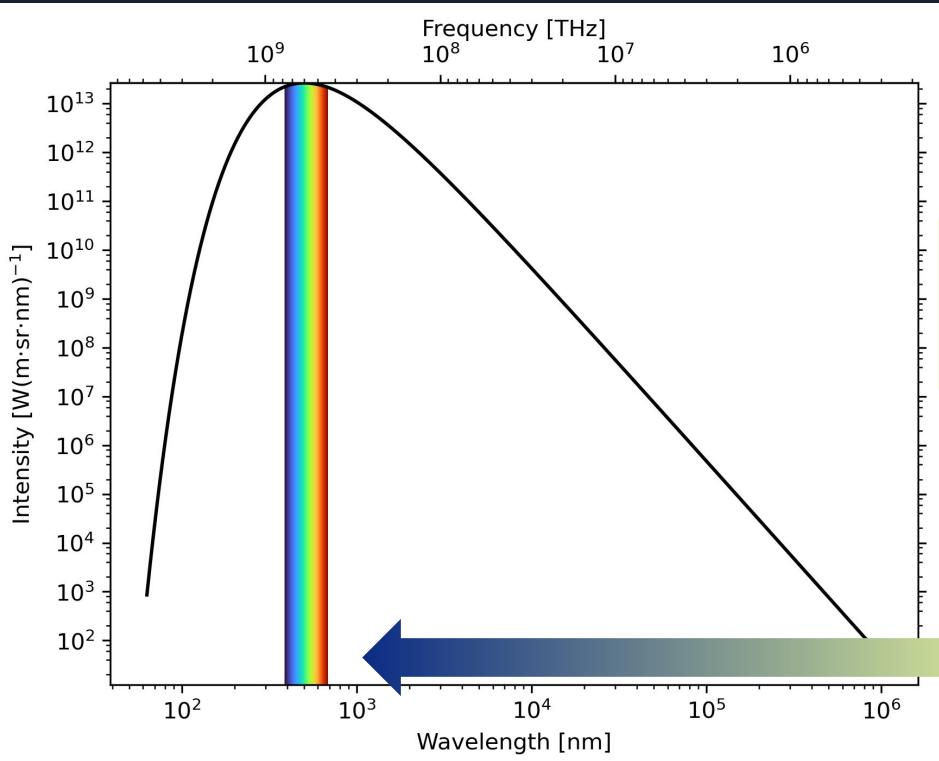
```
c = 2.998*10**8.
k_b = 1.380649*10**-23.
hc = (2.998*10**8.)*(6.626*10**-34.)
def bb(wvl,T):
    return ((2*hc*c)/(wvl**5)) * 1/(np.exp(hc/(wvl*k_b*T)) - 1)

wvs = np.logspace(-7.2,-3.0,471)
bb5777 = bb(wvs,5777.)

import matplotlib as mpl
import matplotlib.ticker as ticks
fig, ax = plt.subplots(dpi=300)
ax.plot(wvs*10**9,bb5777,'k-')

# 1 nm = 10^-9 m, 1 THz = 10^12 Hz
secax = ax.secondary_xaxis('top',functions=(lambda x: 1000*c/x,
                                             lambda x: 0.001*c/x))
```

Example of `ax.secondary_xaxis('top', functions=(prim2sec,sec2prim))` & `mpl.ticker.LogLocator()`



```
import matplotlib as mpl
import matplotlib.ticker as ticks

ax.set_xscale('log')
ax.set_yscale('log')

# PAY SPECIAL ATTENTION TO THE NEXT 4 LINES
ax.yaxis.set_major_locator(ticks.LogLocator(base=10,numticks=99))
ax.yaxis.set_minor_locator(ticks.LogLocator(base=10.0,subs=(0.2,0.4,0.6,0.8),
                                         numticks=99))

ax.yaxis.set_minor_formatter(ticks.NullFormatter())
ax.tick_params(axis='y',which='both',right=True)
ax.set_xlabel('Wavelength [nm]')
secax.set_xlabel('Frequency [THz]')
ax.set_ylabel('Intensity [W(m$\cdot$sr$\cdot$nm)$^{-1}$]')

poly=ax.fill_between(wvs[np.where(np.logical_and(wvs>3.8*10**-7,wvs<7*10**-7))]*10**9,
                      bb5777[np.where(np.logical_and(wvs>3.8*10**-7,wvs<7*10**-7))],
                      color='none') #mark off polygon to fill later
verts = np.vstack([p.vertices for p in poly.get_paths()])
gradient = plt.imshow(np.linspace(0,1, 256).reshape(1, -1),
                      cmap=matplotlib.colormaps['turbo'], aspect='auto',
                      extent=[verts[:, 0].min(), verts[:, 0].max(),
                              verts[:, 1].min(), verts[:, 1].max()])
gradient.set_clip_path(poly.get_paths()[0], transform=plt.gca().transData)
```

In case you're  
interested in the  
rainbow

Example of  
`ax.secondary_xaxis('top',functions=(prim2sec,sec2prim)) &`  
`mpl.ticker.LogLocator()`



# Final notes on ticks & tick locators

For color bars, use `ticks` & `format` kwargs of `colorbar()`

- Format kwarg accepts all the same locator functions as  
`ax.[x|y]axis.set_[major|minor]_locator()`

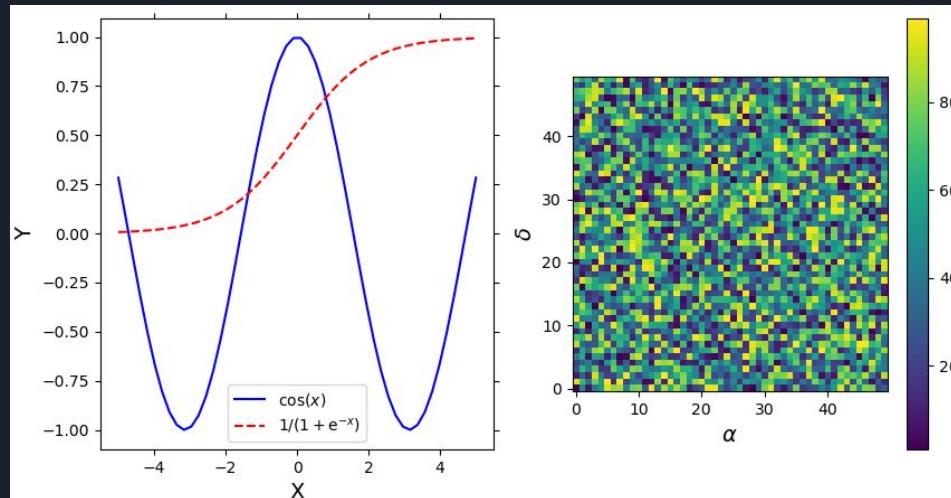
Scales that are neither linear nor logarithmic are not suitable for histograms, contours, or image-like data

- Contours don't work well with log axes either; must work in log units & use tick label formatters to override the labels



# Code Along!

Call `.tick_params(axis='both', which='both', top=True, right=True)` on the first (left) axis object of your plotting code to turn on axis ticks on the right and top axes.





# Grids

**Essential for non-Cartesian coordinate systems**, discouraged otherwise. Matplotlib defaults reflect this (for polar & axes3d plots)

- Newer releases include `matplotlib.projections.geo` → support for map projections (like Mollweide), with customizable `GeoAxes` base class (examples lacking in docs).
- Matplotlib also accepts AstroPy WCS as axes transformations.
- Syntax sample: `ax.grid(ls=':', color='silver', lw=2)` turns grid on with defaults altered. Leave () empty to turn on default grid





# Grids & data format

**Images & 2D histograms:** use contrasting but neutral color; best line style depends on variability of image data

**Scattered data:** choose light, thin, solid lines in a neutral color.

- Set `zorder=0` to put grid behind sparse data. Leave grid atop dense data
- Online-only figures: can use styles with light grids on dark backdrop

**Lines, bars, & stems:** thin lines in neutral color set behind data

- **Bar & stem plots:** only turn on grid lines perpendicular to the bars/stems using the `axis` kwarg of `grid()`



# Now You Try!

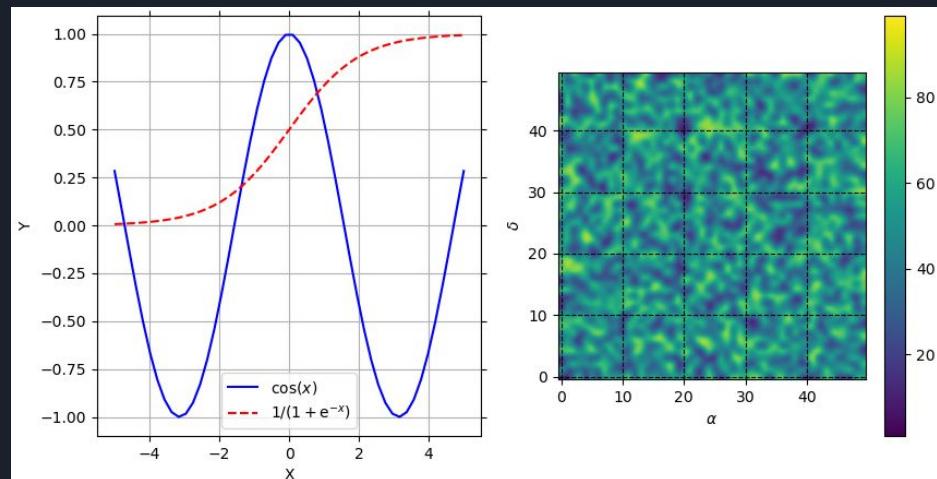
Return to your code-along code & add `interpolation='bicubic'` as a kwarg to the `imshow()` plot (it makes gridlines easier to see). Then turn on grids for both plots. Use the defaults on the left plot & black dashed lines on the right.

## Solutions:

Left: `axes[0].grid()`

Right:

```
axes[1].grid(color='k', ls='--')
```





LUND  
UNIVERSITY

# Questions?



LUND  
UNIVERSITY

# Part 2. Formatting & Accessibility (Continued)



# Fonts & Font Sizes

Text base class & all derivatives (titles, legends, etc.) have `fontsize` kwarg.

- Most journals set hard minimum font size ~9-10 pt. Prefer **11-14 pt font & err larger for older readers.**
- Pick a common font style (or similar) & stick with it. Default is fine.
  - No consensus whether serif or sans-serif fonts are more legible



# Setting Fonts with rcParams

`plt.rcParams['font.size'] = 14` sets all labels to size 14 font, for example.  
Can set font family as shown below:

```
import matplotlib as mpl
### To select a sans-serif font:
mpl.rc('font', **{'family':'sans-serif','sans-serif':['Helvetica']})
### To select a serif font:
#mpl.rc('font', **{'family':'serif','serif':['Liberation Serif']})
mpl.rc('text', usetex=False)
```

Don't set `usetex=True` unless you want to ensure italicized serif math text AND you have very obscure LaTeX symbols to typeset.

```
### To reset to defaults:
mpl.rcParams.update(mpl.rcParamsDefault)|
```

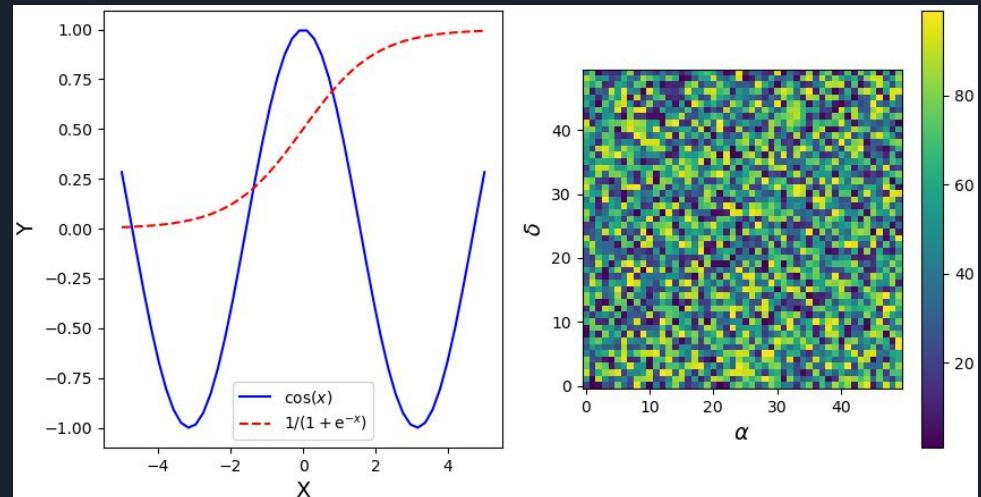


# Now You Try!

Return to your plotting code from the past several “Now You Try!” exercises. Increase the font sizes of the X & Y axes labels to size 14.

```
fig, axes = plt.subplots(ncols=2, figsize=(10,5))
axes[0].plot(x, np.cos(x), 'b-', label=r'$\cos(x)$')
axes[0].plot(x, 1/(1+np.exp(-x)), 'r--',
              label=r'$1/(1+\mathit{e}^{-x})$')
axes[0].set_xlabel('X', fontsize=14)
axes[0].set_ylabel('Y', fontsize=14)
axes[0].legend(loc='lower center')

Z = np.random.randint(1, 100, (50,50))
img = axes[1].imshow(Z, origin='lower')
axes[1].set_xlabel(r'$\alpha$', fontsize=14)
axes[1].set_ylabel(r'$\delta$', fontsize=14)
cb = plt.colorbar(img)
plt.show()
```





# Choosing color palettes

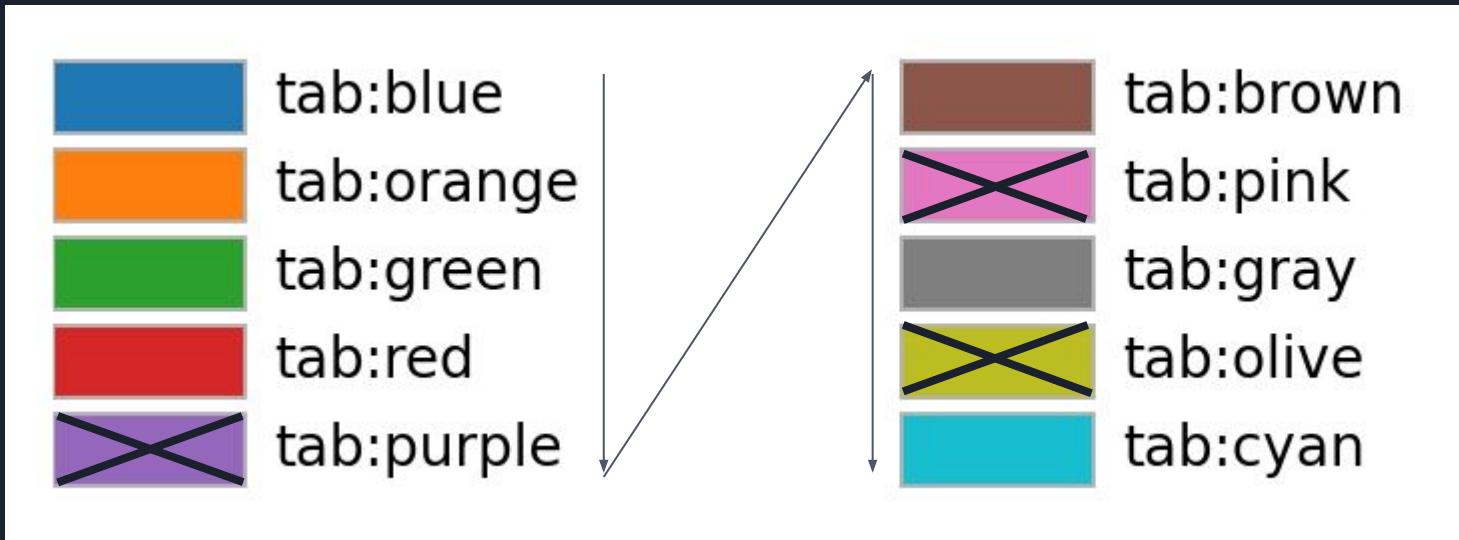
Color blindness affects 1/12 people with 1 X-chromosome & 1/200 with at least 2 X chromosomes; try to make plots colorblind friendly.

- Lots of online simulators to help → [DaltonLens](#), [Sknettech](#), [Coblis](#), ...

Shades of **blue** & **orange** are most easily distinguished colors for all 3 types of dichromacy (& the best 3rd color is either gray or **cyan**).

- Can import Tableau's colorblind10 palette with  
`plt.style.use('tableau-colorblind10')` (more on styles later)
- Can use `plt.colormaps['your_cmap'](np.linspace(0.1, 0.9, n))` to generate  $n$ -color palettes from colormap





Default color cycle is not great for >4 colors  
But should vary marker/line styles for >4 data sets anyway



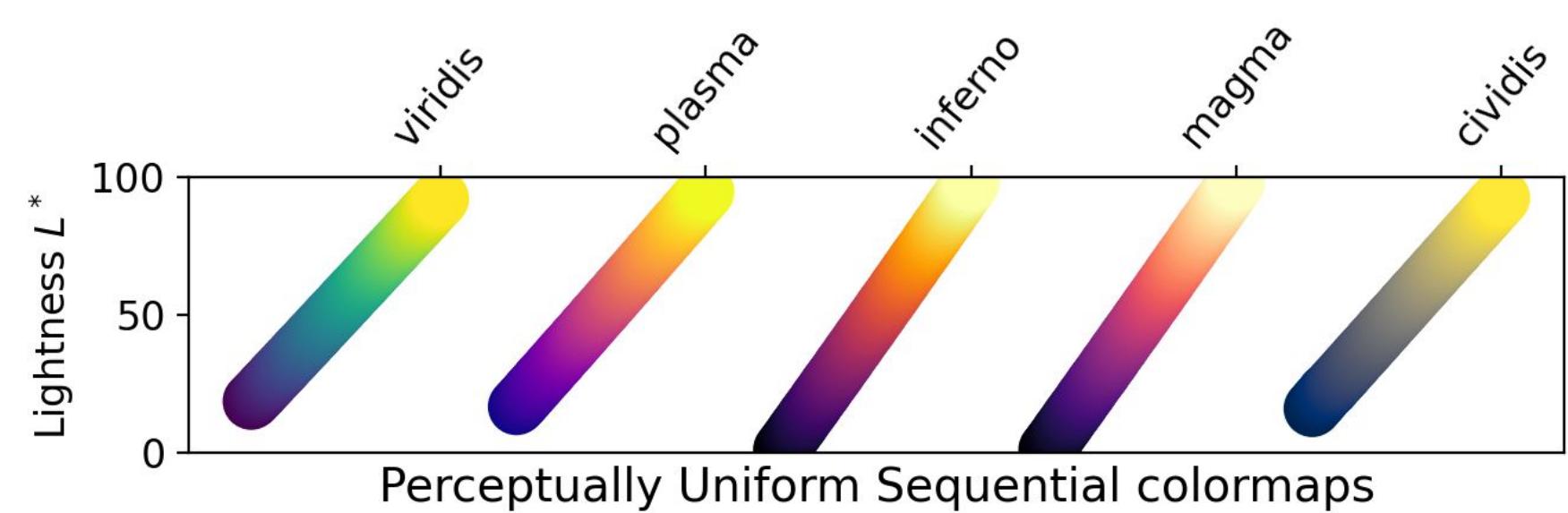
# Choosing colormaps

Matplotlib docs mostly cover this. In short, consider:

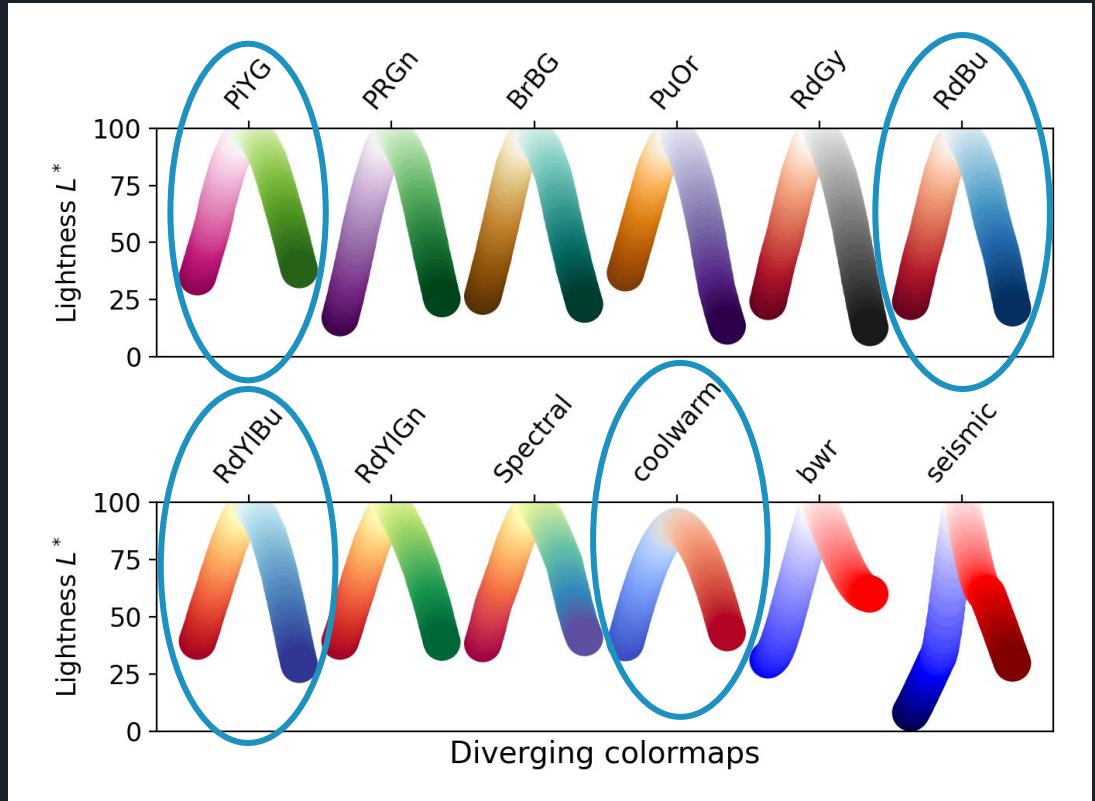
- **Data type requirements:** should colormap be sequential, diverging, cyclical (only 1 good option), or categorical?
- **Dynamic range:** enough contrast between lights & darks?
- **Monotonicity:** does brightness vary smoothly & linearly so that it still reads correctly in black & white?

**Note:** all colormaps can be reversed by appending '\_r' to colormap name string.





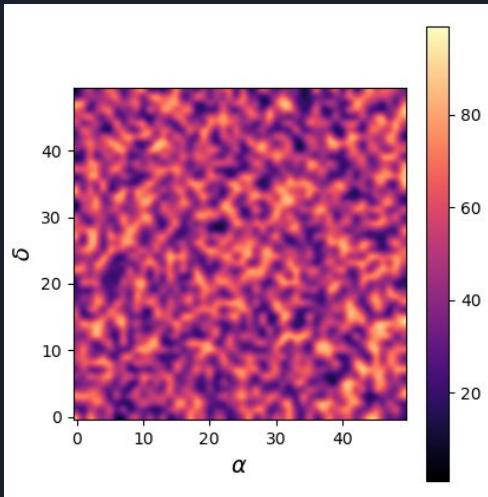
These 5, 'binary', 'gray', & 'bone' are good sequential colormaps.  
The rest...not so much. Avoid 'rainbow' & 'jet'.



Avoid diverging color maps with very different lightness at each end.  
For that & colorblind accessibility, avoid the colormaps not circled.

# Now You Try!

Return to your plotting code & change the color map for the image to 'magma'.



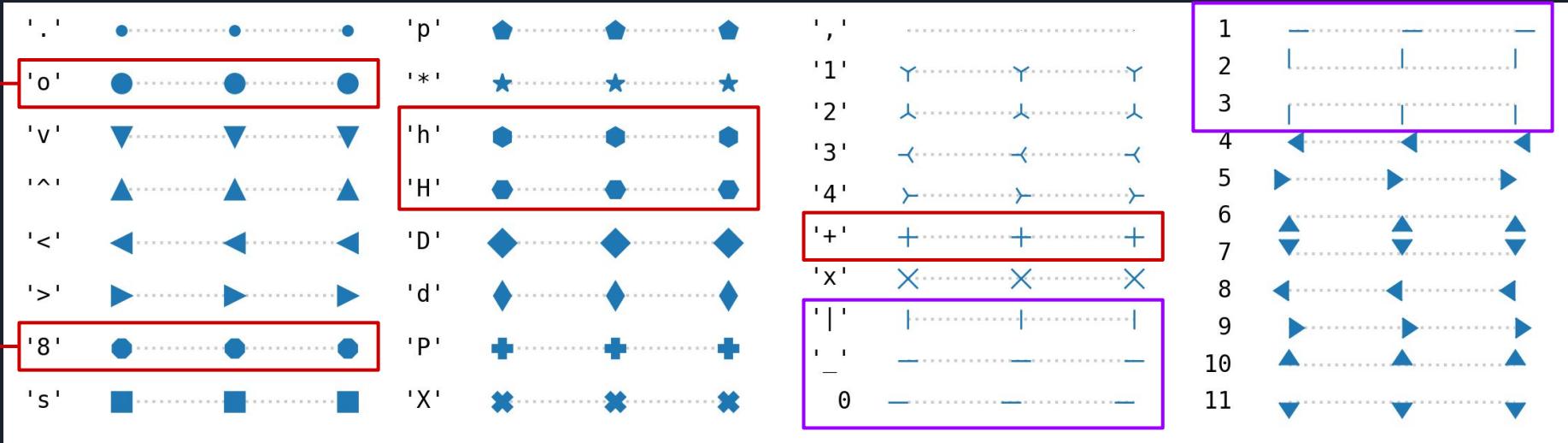
All you had to do was add `cmap='magma'` to the list of `kwargs` in `imshow()`.

Most plots now have a `cmap` kwarg if it makes sense & accept color map names as strings. Older versions & other color-related functions may require you to import `matplotlib.colors` & select the colormap like a dict entry, e.g.  
`matplotlib.colors['magma']`





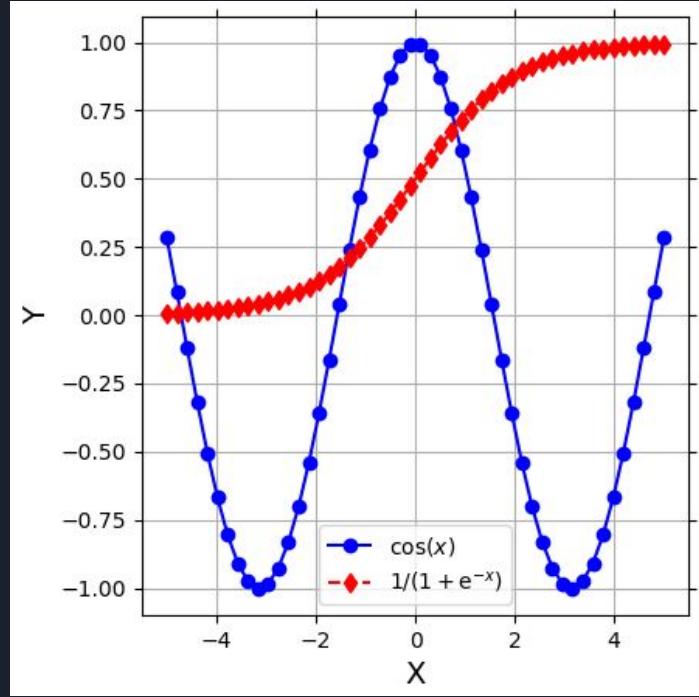
# Choosing Markers



- If using easily confused shapes or the same shape in different rotations, choose different colors with different values (brightness)
- "+" can be confused for error bars if error bars don't have caps.
- Reserve | & - for specialty plots (event plots & rug plots)

# Now you try!

Return to your plotting code & add markers to the red & blue lines in the left panel. Use large circles ('o') for the blue cosine curve, & diamonds/rhombuses ('d') for the exponential curve. All you have to do is add a shape code to each format string.



```
axes[0].plot(x, np.cos(x), 'bo-', label=r'cos($x$)')
axes[0].plot(x, 1/(1+np.exp(-x)), 'rd--',
label=r'$1/(1+\mathrm{e}^{-x})$')
```



# Choosing Line Styles

Line styles vary in clarity → unspoken hierarchy to consider for your narrative (subject to change if you adjust dash/dot spacing)

1. Solid lines (`ls='-'`): clearest → **most important**
2. Dashed lines (`ls='--'`) (could argue if #2 or #3)
3. Dash-dotted lines (`ls='-.'`) (↑ ditto)
4. Dotted lines (`ls=':'`): easiest to lose among scattered data → **least important**





# Practical limits of lines

Max # of lines per figure depends heavily on other content.

- Lines with error margins &/or underlying scattered data: **4-5 curves** depending on whether overlying data have color
- Lines without error margins & no underlying scattered data:
  - **8-10 independent curves** of similar importance
  - **10-20 curves** in a hierarchy, with little to no crossing
  - Many if data are correlated & evolving with a 3rd variable (e.g., monthly mean temperatures over years - let older curves fade out)





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

import matplotlib as mpl

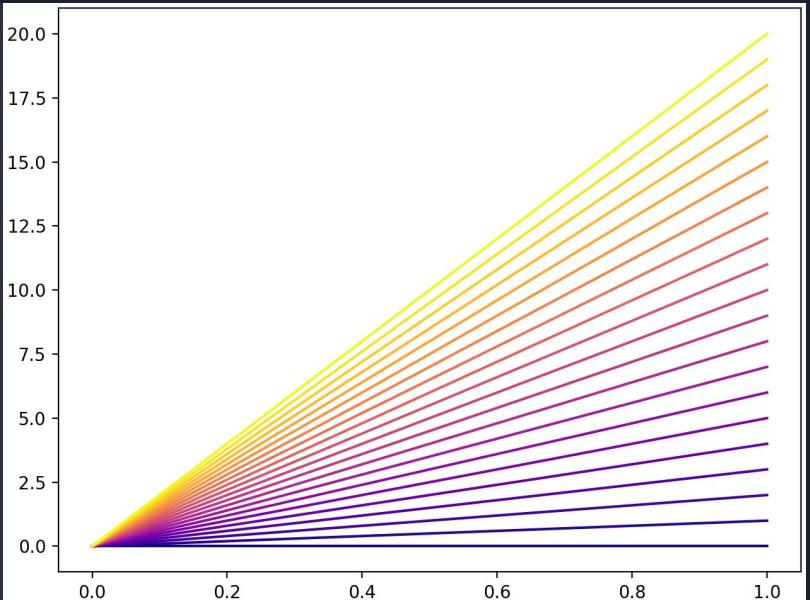
n_lines = 21
cmap = mpl.colormaps['plasma']

# Take colors at regular intervals spanning the colormap.
colors = cmap(np.linspace(0, 1, n_lines))

fig, ax = plt.subplots(layout='constrained')

for i, color in enumerate(colors):
    ax.plot([0, i], color=color)

plt.show()
```



Example: 21 lines of increasing slope, with colors drawn from a colormap



LUND  
UNIVERSITY

# Questions?



LUND  
UNIVERSITY

# Part 3. Dos and Don'ts of Data Representation



# Standard representations for most data:

Data type	Plot Format
Time Series	Line, stem, or stack plots (like stacked <code>fill_between</code> )
Probability Distributions	Histograms or Kernel Density Estimations (KDEs); Box plots or violin plots to compare different populations; Corner plots for MCMC simulations of model parameters
Budgets & Fractions	Proportional area charts (pie chart is most [in]famous)
Vector fields	Quiver or stream plots
Spectra	Line of intensity vs frequency, wavelength, or wave number
Elevation or Intensity	Contour maps, (pseudo)color images, or surface plots



# Main Challenges

- How best to combine different data types/sets for ease of comparison, &
- How to make the key result(s) easy to see & understand

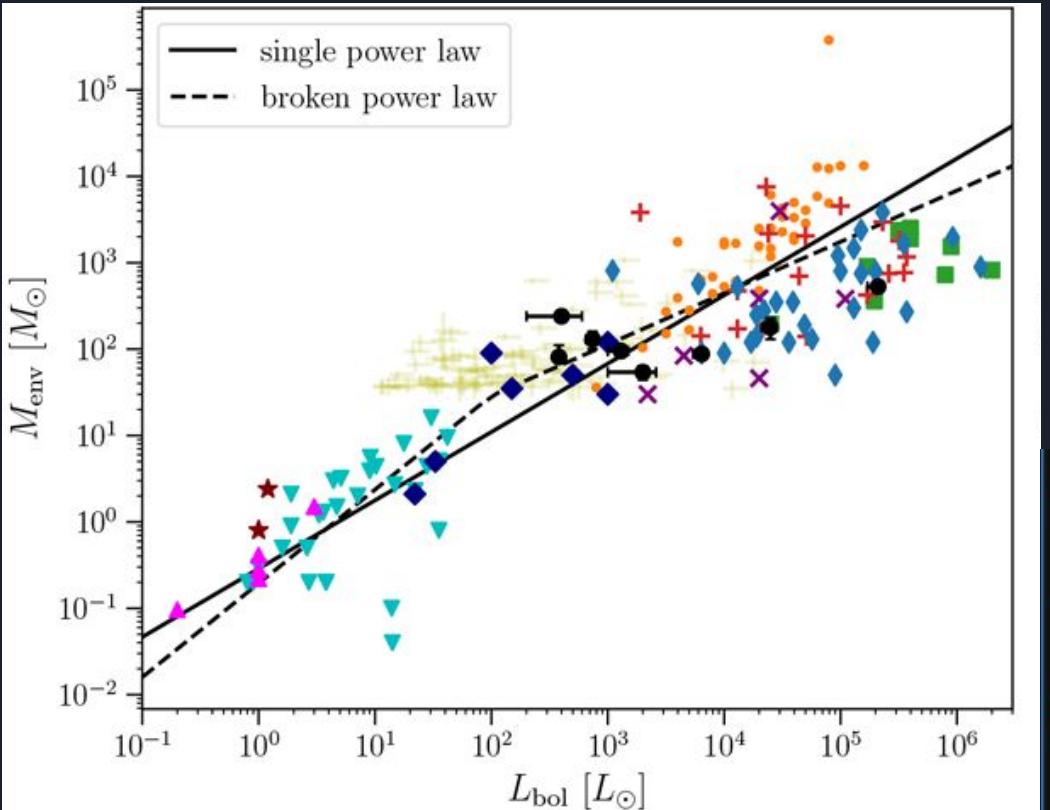


# Do: distinguish your data on communal plots

When adding data to a figure from (others') previous research, **it is OK to deemphasize other data & highlight what is yours &/or new by...**

- Decreasing the opacity (`alpha`) or marker sizes of other data.
- Showing error bars only for your own data.
- Coloring your data black & all other data lighter colors, e.g. with custom color cycler from `itertools.cycle`
- Emphasizing line(s) with thicker, darker, or more solid line style



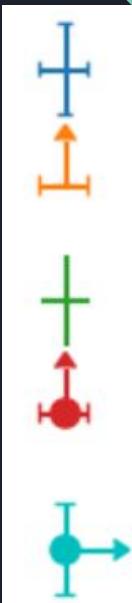


I set the markers & colors in the ASCII table, but this is how you might handle it in a script:

```
from itertools import cycle
c_cycle = cycle(['tab:orange', 'tab:blue', 'tab:red',
                 'tab:green', 'tab:cyan', 'm', # m = magenta
                 'navy', 'maroon', 'purple'])
m_cycle = cycle(['.', 'd', '+',
                 's', 'v', '^',
                 'D', '*', 'x'])
#pretend we have a dict of studies & data by author group
for authors,study in studies.items():
    plt.scatter(study[:,0],study[:,1], c = next(c_cycle),
                ms = next(m_cycle), ls='', label=authors)
```

Example plot from R. L. Pitts et al. 2021b using many of the previous tips (data & markers were labelled in a caption)

# Do: include error bars however feasible

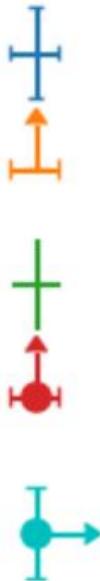


# of data points	Best error bars/limits format
Up to a few 10s	Error bars/limits on every point, with caps if large
Few 10s to ~100	Error bars on every point; omit caps
Few 100s	Error bars on every nth point ( <code>errorevery</code> kwarg)
Many 100s & up	Representative error bar(s) in an unoccupied corner

**Lines:** use `fill_between(x, yerr_lo, yerr_up)` in a light or translucent color under the main line. Separate upper & lower margin lines are less intuitive.



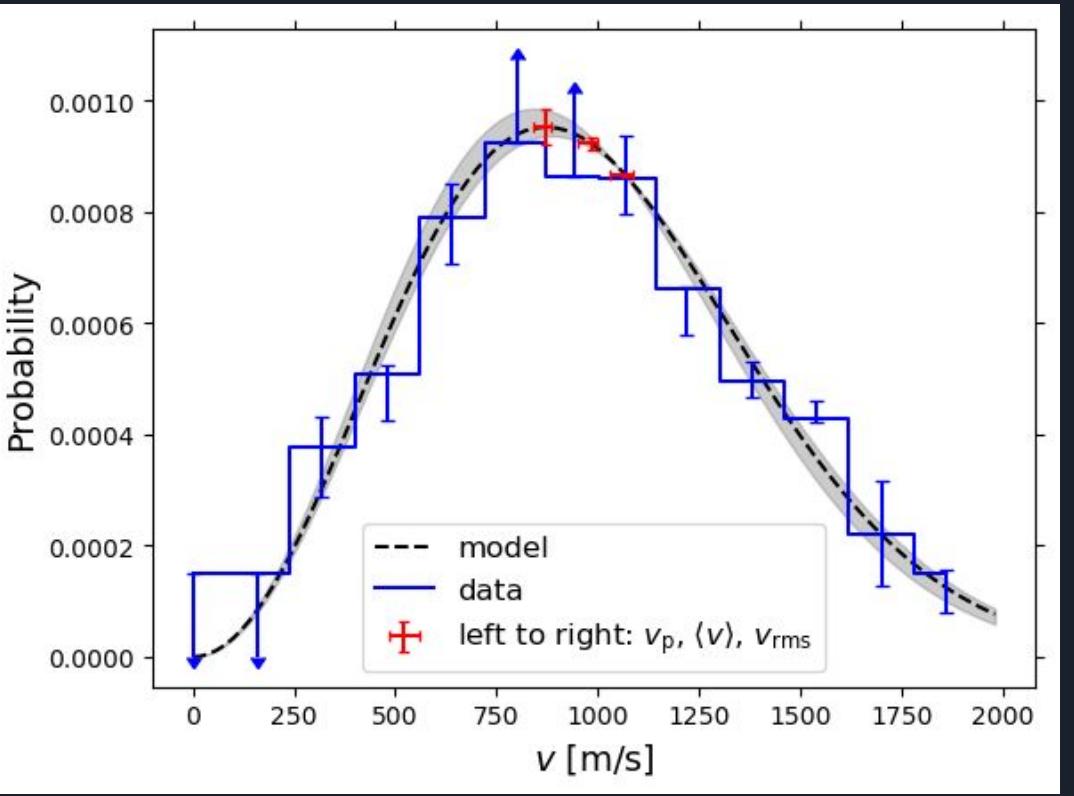
# Don't: set error margins to 0 for limits



Error bars can be tricky if asymmetric or mixed with limits.

- `lolims`, `uplims`, `xlolims`, & `xuplims` kwargs take boolean mask arrays.
- Where limit masks are True, `xerr` &/or `yerr` arrays *must still have values for limit arrow lengths*.
- If error bars are asymmetric, wherever there is a limit, `xerr` or `yerr` must have a value  $>0$  in the direction that the limit arrow points.
  - Any value on the other side will be ignored unless it raises a `ValueError` (e.g. 0 on a log-log plot).





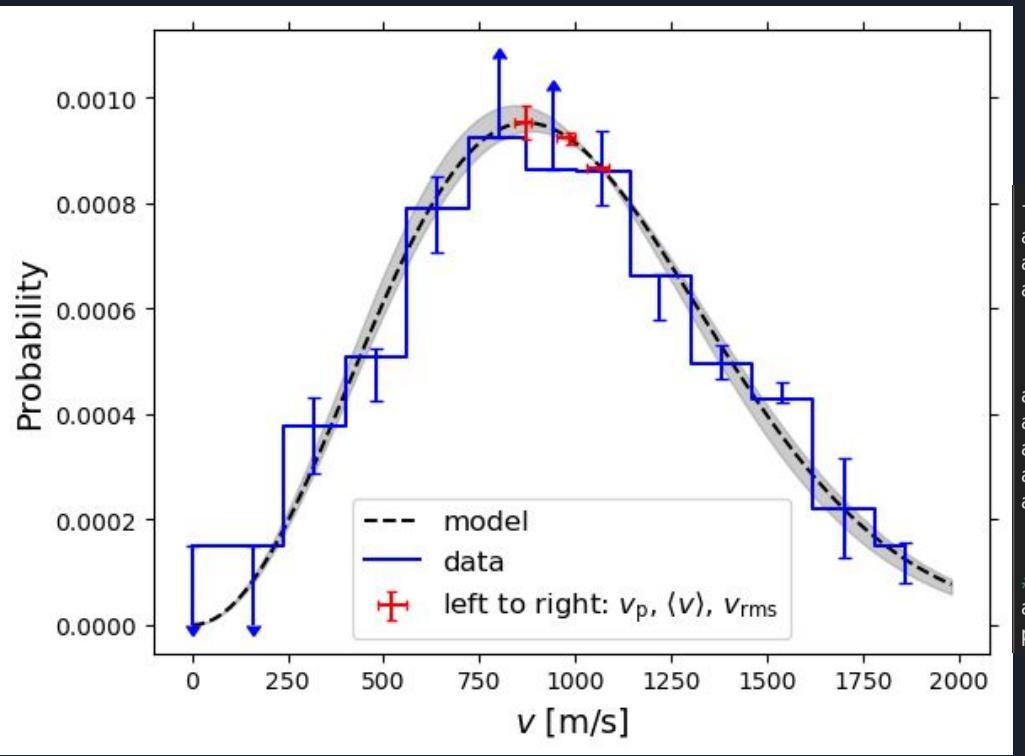
Example: faux Maxwell-Boltzmann distribution of a hot (5600-6200 K) Xenon arc lamp

```

k_B = 1.380649*10**-23
m_amu = 1.660539*10**-27 #atomic mass unit
def max_boltz_pdf(v,m_u,T):
    m2kT = m_u*m_amu / (2*k_B*T)
    return ((m2kT / np.pi)**1.5) * (4*np.pi*v**2) \
           * np.exp(-(v**2) * m2kT))
mpvs = np.sqrt(2*k_B*np.array([5600.,6000.,6200.]) / \
                (131.293*m_amu)) # = 872 m/s at T=6000 K
mv = mpvs*2/np.sqrt(np.pi)
rmsv = mpvs*np.sqrt(1.5) #most probable, mean, & rms velocities

x = np.array([mpvs[1],mv[1],rmsv[1]])
vs = np.sort(np.concatenate((np.arange(0.,2000.,20.),x)))
mbd_6000 = max_boltz_pdf(vs,131.293,6000.)
mbd_5600 = max_boltz_pdf(vs,131.293,5600.)
mbd_6200 = max_boltz_pdf(vs,131.293,6200.)
y = max_boltz_pdf(x,131.293,6000.)
#Mock up some errors
xerrs = [x-np.array([mpvs[0],mv[0],rmsv[0]]), \
          np.array([mpvs[2],mv[2],rmsv[2]])-x]
yerrs = np.zeros((2,3))
for i,xi in enumerate(x):
    vi = np.where(vs == xi)
    if mbd_6000[vi]-mbd_5600[vi] > 0:
        yerrs[0][i] = mbd_6000[vi]-mbd_5600[vi]
    else:
        yerrs[1][i] = abs(mbd_6000[vi]-mbd_5600[vi])
    if mbd_6200[vi]-mbd_6000[vi] > 0:
        yerrs[1][i] = mbd_6200[vi]-mbd_6000[vi]
    else:
        yerrs[0][i] = abs(mbd_6000[vi]-mbd_5600[vi])
# fudge some noise and limits
probs = mbd_6000[::8]+np.random.default_rng()\
    .uniform(-1,1,len(vs[::8]))*0.0001
probs[np.where(probs<0.00015)]=0.00015

```



```
fig, ax = plt.subplots(dpi=100)
ax.fill_between(vs, mbd_5600, mbd_6200, alpha=0.4, color='gray')
ax.plot(vs, mbd_6000, 'k--', label='model')
ax.errorbar(x, y, xerr=xerrs, yerr=yerrs, linestyle='',
            ecolor='r', capsize=2, zorder=20,
            label=r'left to right: $v_{\mathrm{p}}$, $\langle v \rangle$, $v_{\mathrm{rms}}$')
ax.set_xlabel('$v$ [m/s]', fontsize=14)
ax.set_ylabel('Probability', fontsize=14)
ax.tick_params(top=True, right=True)
ax.step(vs[:8], probs,color='b', where='mid', label='data')
ax.errorbar(vs[:8], probs, yerr=perrs,
            uplims=puplims, lolims=plolims,
            linestyle='', ecolor='b', capsize=3)
#Note: capsize also controls arrow size
ax.legend(loc=8,fontsize=12)
plt.show()
```

Example: faux Maxwell-Boltzmann distribution of a hot (5600-6200 K)  
Xenon arc lamp



# Scattered Data by the Numbers 1.

## A few $\times$ 10 to a few $\times$ 100 data points

- **Do:** Vary color (`c` kwarg of `scatter()`) &/or marker style for different datasets or values of a categorical parameter
  - If feasible, vary both colors & markers together for colorblind accessibility & so data are still distinct in grayscale
  - Keep in mind: some markers are hard to tell apart (e.g., large circles & octagons)
- **Don't:** color by numerical 3rd parameter (`cmap` kwarg of `scatter()`) unless it's correlated with at least 1 axis parameter





# Scattered Data by the Numbers 2.

## A few $\times$ 100 to a few $\times$ 1000 data points

**Do:** use `scatter()` with transparency, or `hist2d()` or `hexbin()` with logarithmic bins if outliers are important

- Reduce scatter plot opacity: set `alpha` kwarg  $\sim 0.2\text{-}0.5$
- **Many rules for histogram binning, but generally, distribute  $n$  data points over  $n^{1/2}$  to  $n^{1/3}$  bins** (empty bins don't count)
- Distinct datasets can be shown in different colors as points or line contours

**Don't:** try to contour a histogram or overlay multiple 2D histograms





# Scattered Data by the Numbers 3. 1000s of data points or more

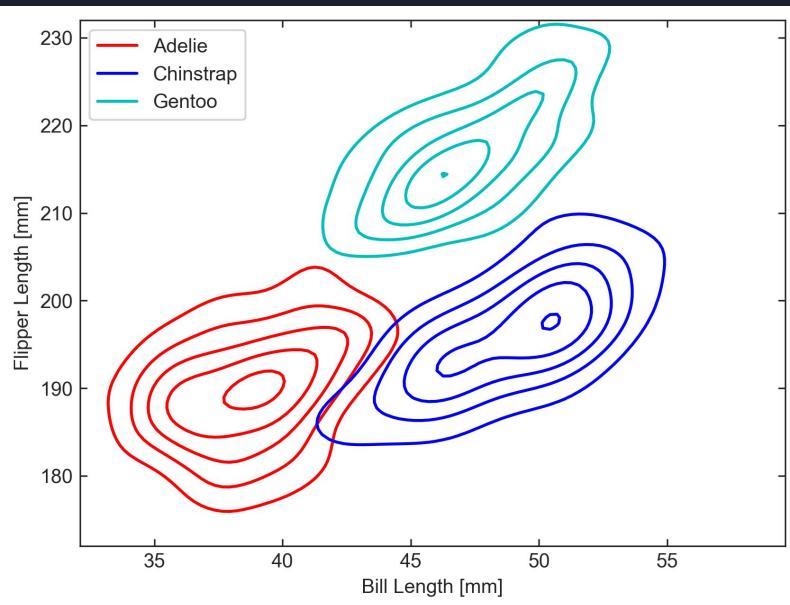
**Do:** Use histograms, Gaussian Kernel Density Estimations (KDEs), or contours\*, or `corner.hist2d()` if outliers are important

- Gaussian KDEs  $\approx$  histograms smoothed by summing bell curves of unit height & width = kernel bandwidth centered at each point.
  - **Pros:** continuous functions, smooth contours of scattered data
  - **Cons:** require SciPy or Seaborn (much easier); slow processing

\*Without Seaborn, multiple sets of contours require a [proxy artist](#) to be added to a legend.



# 2D KDE example with SciPy



```
lenbill, lenflip = penguins[['bill_length_mm',
                             'flipper_length_mm']].to_numpy().T
species = penguins['species'].to_numpy()

from scipy.stats import gaussian_kde

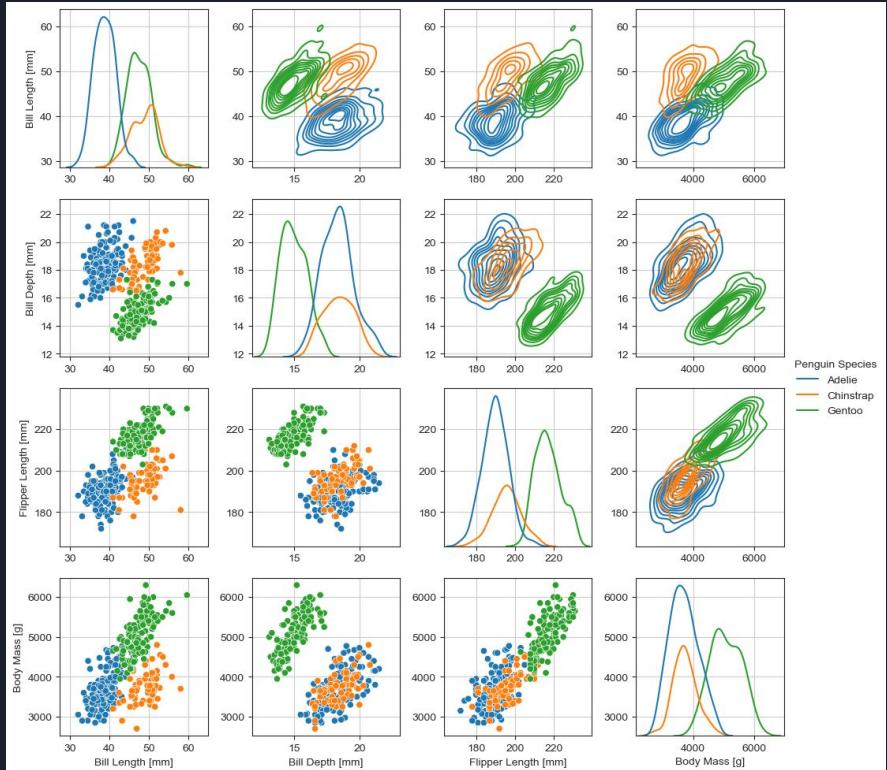
X, Y = np.mgrid[min(lenbill):max(lenbill):100j,
                  min(lenflip):max(lenflip)+1:100j]
pos = np.vstack([X.ravel(), Y.ravel()])

lines = []
fig, ax = plt.subplots(dpi=300)
labels = sorted(list(set(species)))
cc = ['r', 'b', 'c']
for j,s in enumerate(labels):
    inds = np.where(np.logical_and(species == s, np.isfinite(lenbill)))
    kernel = gaussian_kde(np.array([lenbill[inds],lenflip[inds]]))
    Z = np.reshape(kernel(pos).T, X.shape)
    pcp = ax.contour(X,Y,Z, colors=cc[j], levels=5)
    lines.append( mpl.lines.Line2D([], [], color=cc[j], label=s) )
ax.legend(handles=lines, loc=2)
ax.set_xlabel('Bill Length [mm]')
ax.set_ylabel('Flipper Length [mm]')
ax.tick_params(direction='in', right=True, top=True)
plt.show()
```



# Do: Use Corner/Pair Plots for (some) Correlated Variables

- Great for showing correlations between up to ~5 parameters
- Easy with **Seaborn** package if you know Pandas (but can be hard to typeset) →
  - Built-in method to color by categorical variable
- Can also use [corner.py](#) package

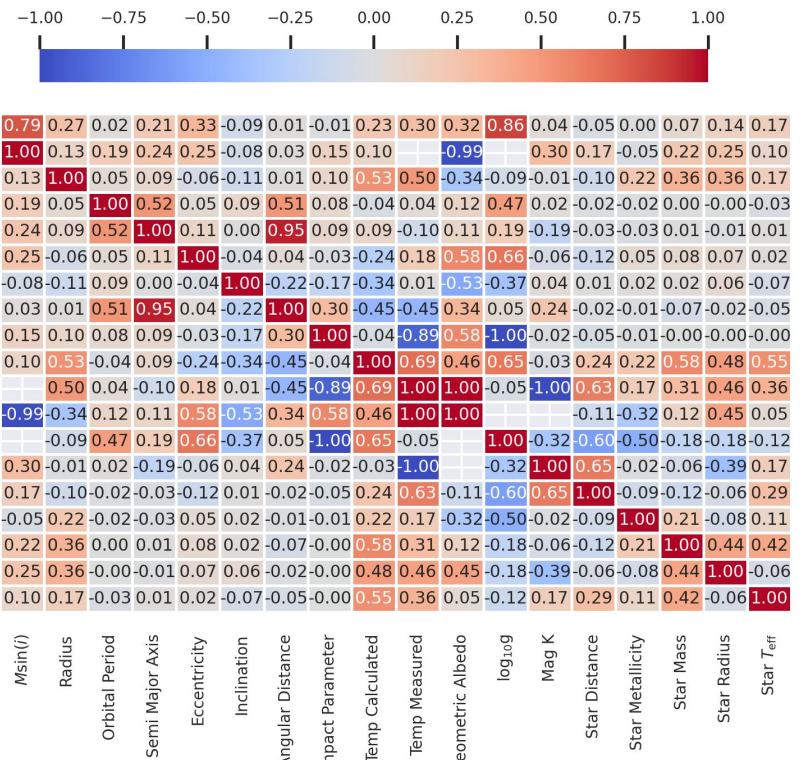




# Don't: Use Corner/Pair plots for >>6 parameters

- For ~10-30 numerical parameters, only a heatmap of correlation coefficients will fit
  - Journal-mandated minimum font size sets hard upper limit
- Ask yourself: “Do I *really* have to plot that many variables on one figure? Can I not break them up?”
- If you absolutely must, use either `imshow()`-based heatmap (more flexible) or `seaborn.heatmap()` (more straightforward if you know Pandas)





```

import requests, string
import seaborn as sb
exops = pd.read_csv('https://exoplanet.eu/catalog/csv/') #98 cols!
dropkeys = [title for title in exops.columns if any(
    t in title for t in ['error', 'name',
    'tzero', 'mag'])]
exops1=exops.drop(columns=[*dropkeys, 'planet_status', 'omega',
    'ra', 'dec', 'k', 'discovered',
    'tperi', 'tconj', 'star_age',
    'hot_point_lon', 'lambda_angle'])
corrs=exops1.corr(numeric_only=True)
print(corrs.shape) # = 18 x 18

annot_labels = np.ma.masked_inside(corrs.to_numpy(), -0.1, 0.1)
axlabs = corrs.columns.to_numpy()
for i, lbl in enumerate(axlabs):
    axlabs[i] = string.capwords(lbl.replace('_', ' '))
        .replace('Mass Sini', r'M$\sin($i$)')
        .replace('Log G', r'\log_{\{10\}}g')
        .replace('Teff', r'T_{\{\mathrm{\{eff\}}\}}$')
#can't use replacement by dict due to implicit '{}'.format()
plt.figure(dpi=300)
sb.set(font_scale=0.6)
hm = sb.heatmap(corrs, cmap="coolwarm", annot=annot_labels,
    cbar_kws={'shrink':0.8, 'location':'top'},
    xticklabels=axlabs, yticklabels=axlabs,
    linewidth=0.75, fmt='.2f', square=False)
sb.set_style('ticks')
plt.show()
sb.set(font_scale=1)

```

# Seaborn Heatmap demo

Best not to plot more variables than `heatmap()` can annotate

heatmap() has built-in kwargs to replace labels with typeset versions

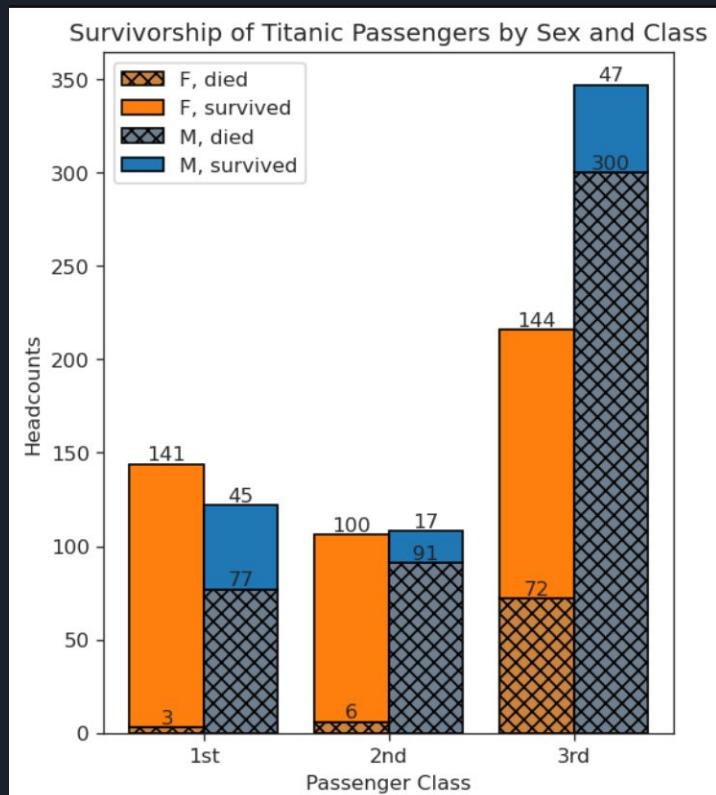
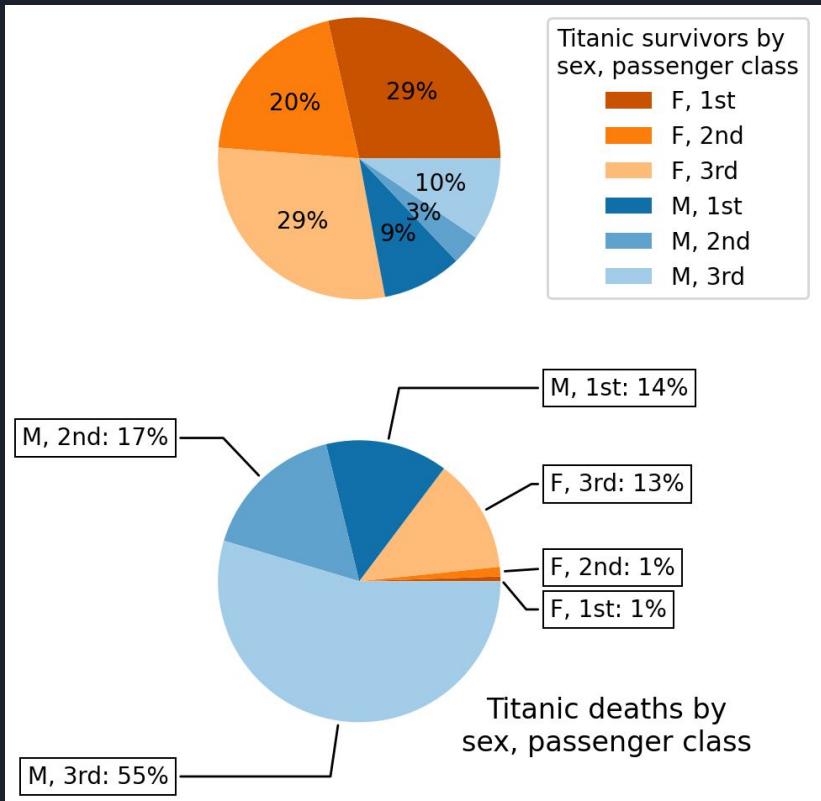


# Don't: Use Pie or Donut Charts

**Why should you NOT use pie or donut charts?** Per many studies...

- Pie & donut charts are inefficient (low info for space occupied)
- Humans are better at judging relative lengths than angles
- Pie charts are easier to read when categories are ordered by fractional size, which may conflict with more natural orders, e.g. alphabetical.
- **State of belonging is usually less important than actual quantities**
- **Inappropriate aggregation can obscure important category-specific details (this applies to many plot formats, e.g. stack/stream plots)**





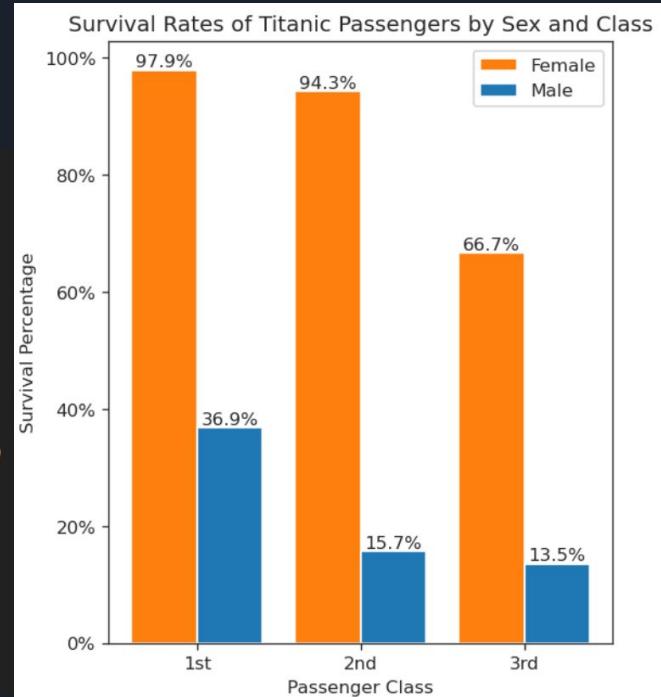
# Compare

# Do: Use Rectilinear Proportional Area Charts

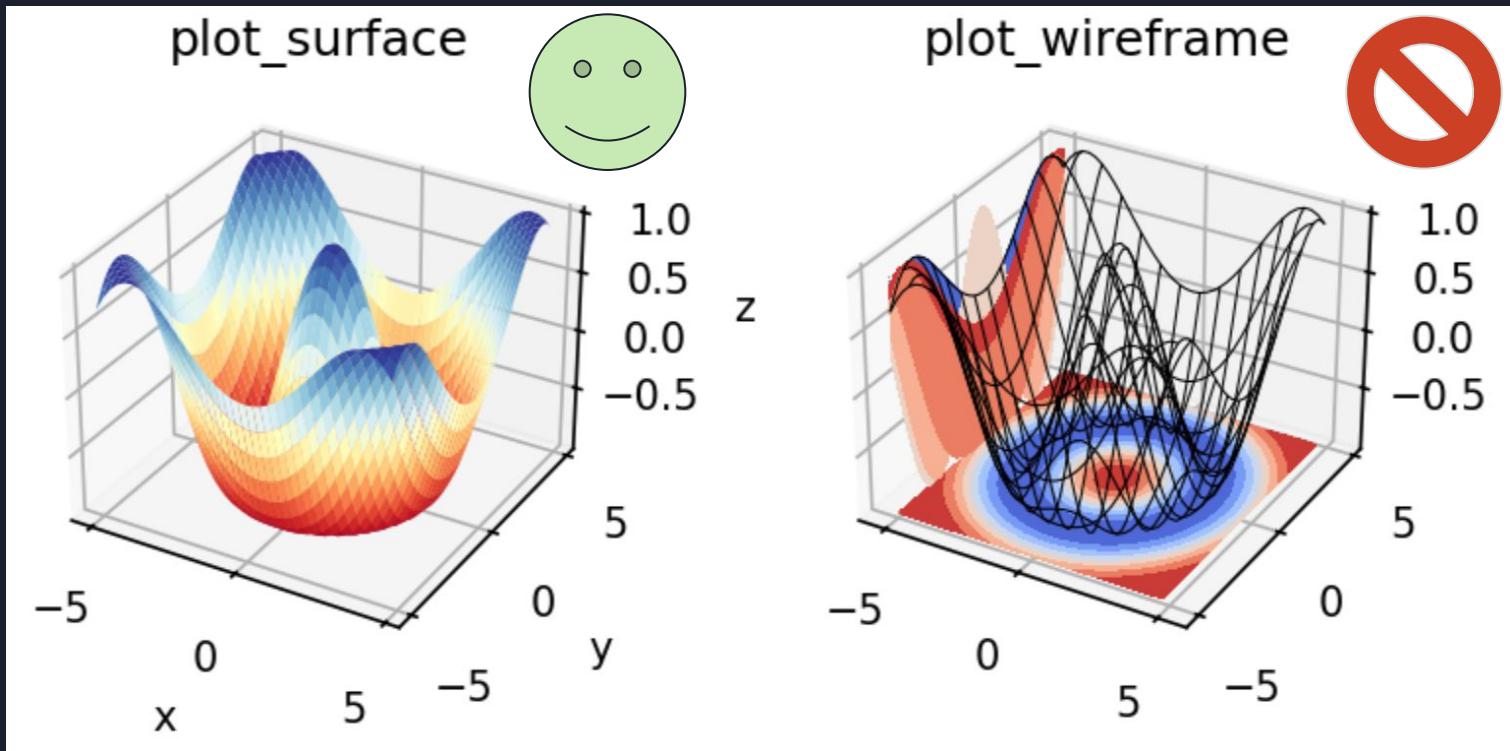
Bar charts are the classic. Others like waffle plots & treemaps require external packages.

```
x = np.arange(3) # classes
width = 0.4 # the width of the bars (there will be 2)
multiplier = 0

for sex, npclass in live.items():
    offset = width * multiplier
    pcts = npclass/(npclass+dead[sex])
    live_recs = ax2.bar(x + offset, pcts, width,
                        label='Male' if sex == 'm' else 'Female',
                        color='tab:blue' if sex == 'm' else 'tab:orange')
    ax2.bar_label(live_recs, labels = ['{:1%}'.format(p) for p in pcts])
    multiplier += 1
ax2.yaxis.set_major_formatter(mpl.ticker.PercentFormatter(1.0))
ax2.set_xticks(x+0.5*offset, ['1st','2nd','3rd'])
ax2.set_title('Survival Rates of Titanic Passengers by Sex and Class')
ax2.set_xlabel("Passenger Class")
ax2.set_ylabel("Survival Percentage")
ax2.legend()
plt.show()
```

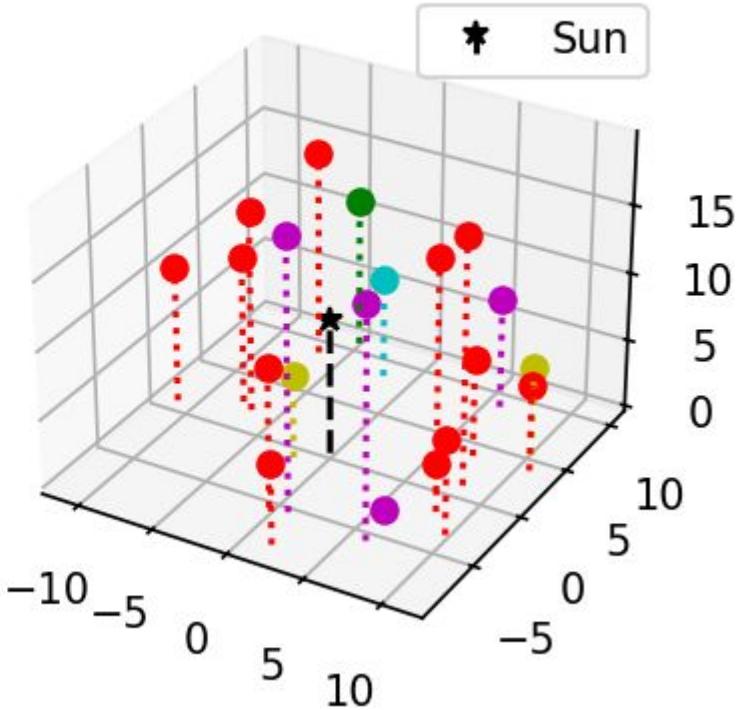
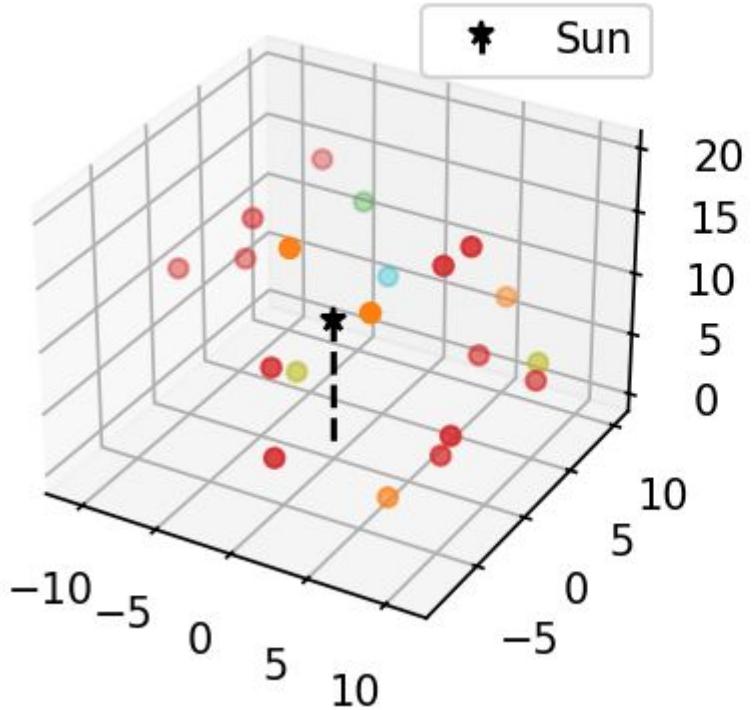


Do: consider depth rendering limitations in 3D





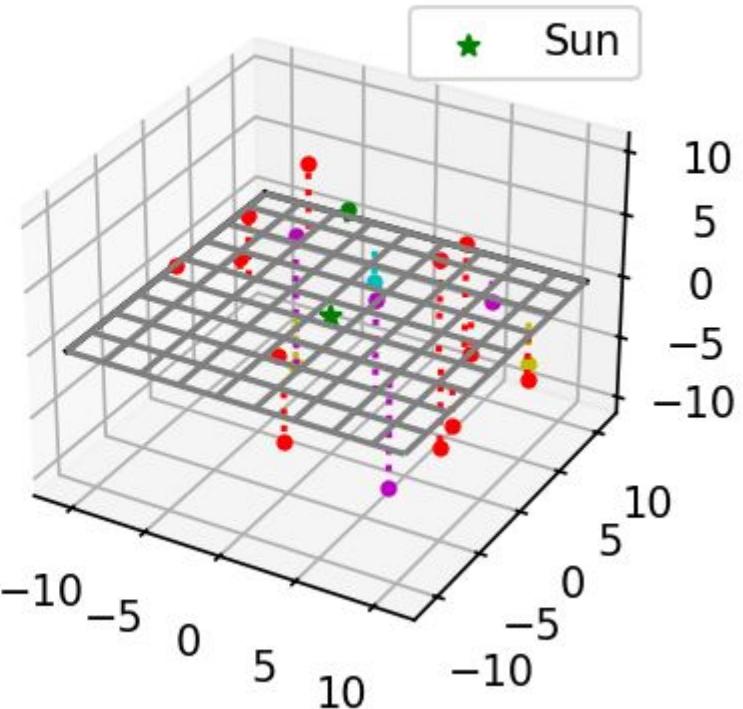
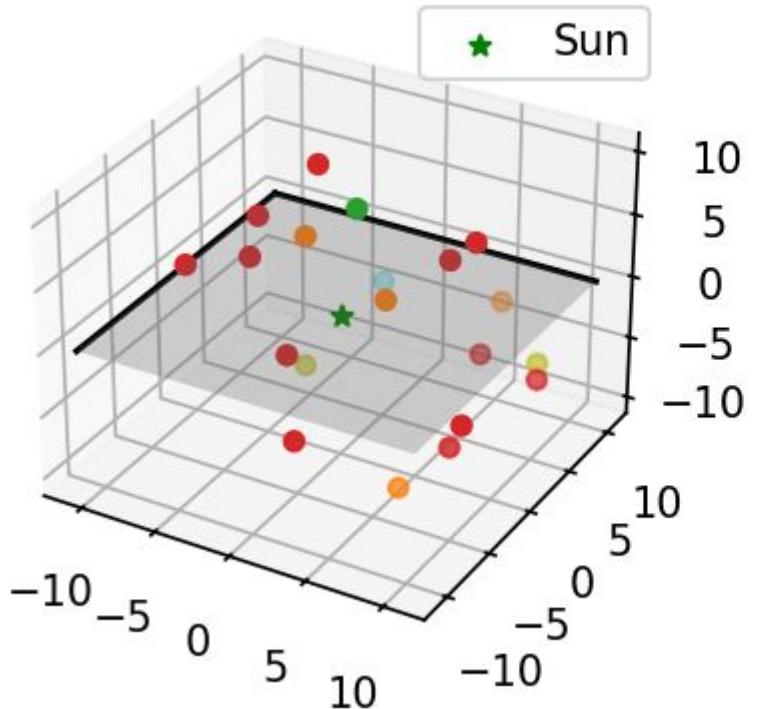
## Nearest 20 Stars (Scale in LY)



In 3D, `scatter()` renders depth within a dataset using transparency (no such effects for other 3D versions of 2D plots)

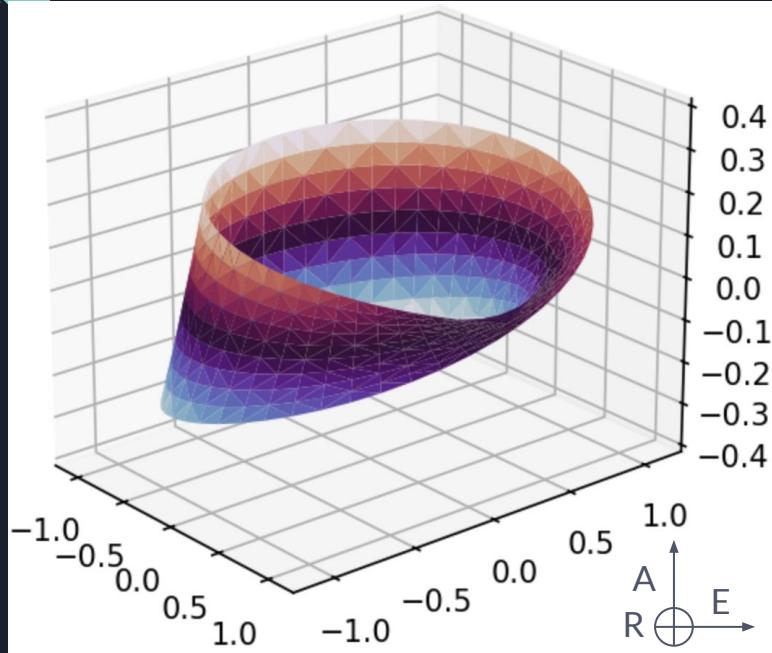


## Nearest 20 Stars (Scale in LY)



Matplotlib can layer data *within* 1 dataset, but multiple datasets tend to resist proper layering, even with `zorder` set for every element

# Do: Control viewing angle with view\_init()



set\_box\_aspect() & its zoom kwarg help with cropping

```
import matplotlib.tri as mtri

fig, ax = plt.subplots(dpi=150, subplot_kw = {"projection": "3d"})

# Make mesh in space of parameterisation variables (u, v)
u = np.linspace(0, 2.0 * np.pi, endpoint=True, num=50)
v = np.linspace(-0.5, 0.5, endpoint=True, num=10)
u, v = np.meshgrid(u, v)
u, v = u.flatten(), v.flatten()

# Möbius mapping: takes (u, v) pair and returns (x, y, z)
x = (1 + 0.5 * v * np.cos(u / 2.0)) * np.cos(u)
y = (1 + 0.5 * v * np.cos(u / 2.0)) * np.sin(u)
z = 0.5 * v * np.sin(u / 2.0)

# Triangulate parameter space
tri = mtri.Triangulation(u, v)

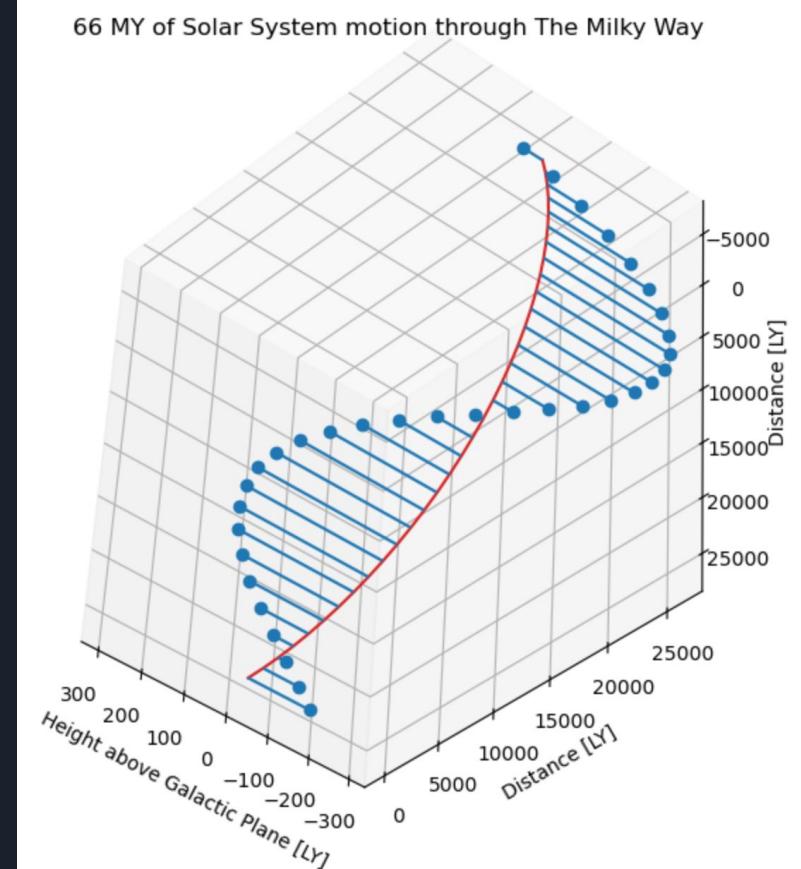
# Plot surface. Triangles in parameter space determine which
# (x, y, z) points are connected by edges.
ax.plot_trisurf(x, y, z, triangles=tri.triangles,
                 cmap=plt.cm.twilight),# , antialiased=False)
ax.set_xlim(-1.2, 1.2)
ax.set_ylim(-1.2, 1.2)
ax.set_zlim(-0.4, 0.4)
ax.view_init(elev=22, azim=-40, roll=0, vertical_axis='z')
ax.set_box_aspect((3, 4, 3), zoom=1.3)
```



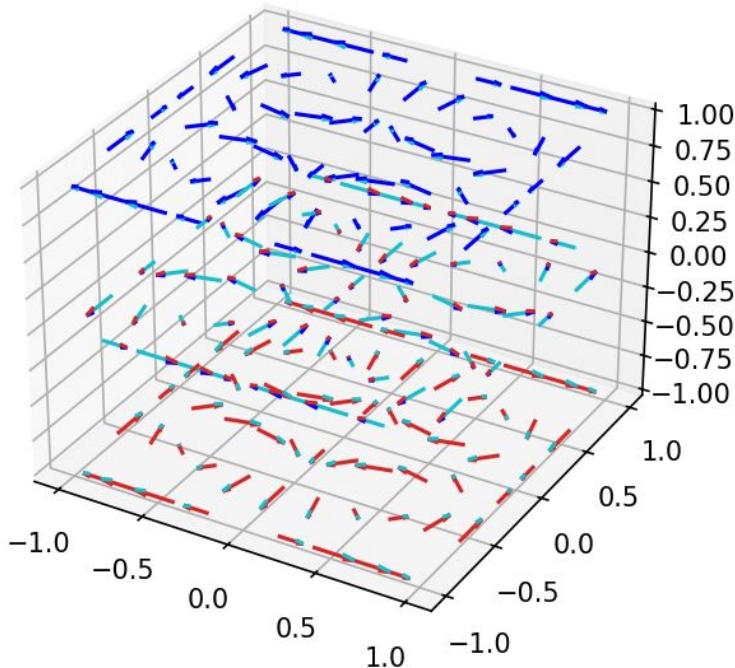
# Now you try!

In the day 1 exercises, you made a 3D stem plot of the Sun's motion through The Milky Way over the past 66 million years. (You can use the solutions file if you didn't finish.)

Set `roll=60` to see how that motion looks when standing (floating?) so that the plane of the Solar System is horizontal & celestial north is up.



# Don't: use 3D vector plots for the time being



```
x, y, z = np.meshgrid(np.arange(-1, 1.2, 0.2),
                      np.arange(-1, 1.2, 0.4),
                      np.arange(-1, 1.5, 1))

# Make direction data for arrows
u = np.sin(np.pi*x)*np.cos(np.pi*y)*np.cos(np.pi*z)
v = -np.cos(np.pi*x)*np.sin(np.pi*y)*np.cos(np.pi*z)
w = (np.sqrt(0.75)*np.cos(np.pi*x)*np.cos(np.pi*y) *
      np.sin(np.pi*z))
# Plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"}, dpi=150)
ax.quiver(x,y,z, u,v,w, length=0.25,
           color=['tab:red', 'tab:cyan', 'b'])
```

Arrows are blunt, drawn with 3 line segments,  
& colors are applied by line segment instead of  
to the whole arrow.

Wait for devs to figure out polygonal arrows.





# Do: Be consistent throughout your paper! (`rcParams` & `style` files can help)

MANY plot properties set internally by Matplotlib's default runtime configuration parameters, or `rcParams`, including but not limited to:

- (Sub)figure formatting
- Text/font formatting
- Marker/line formatting & color palettes/maps
- Axes tick configurations

Any can be set at runtime with dict-like syntax, or imported from file.





# rcParams & .mplstyle files

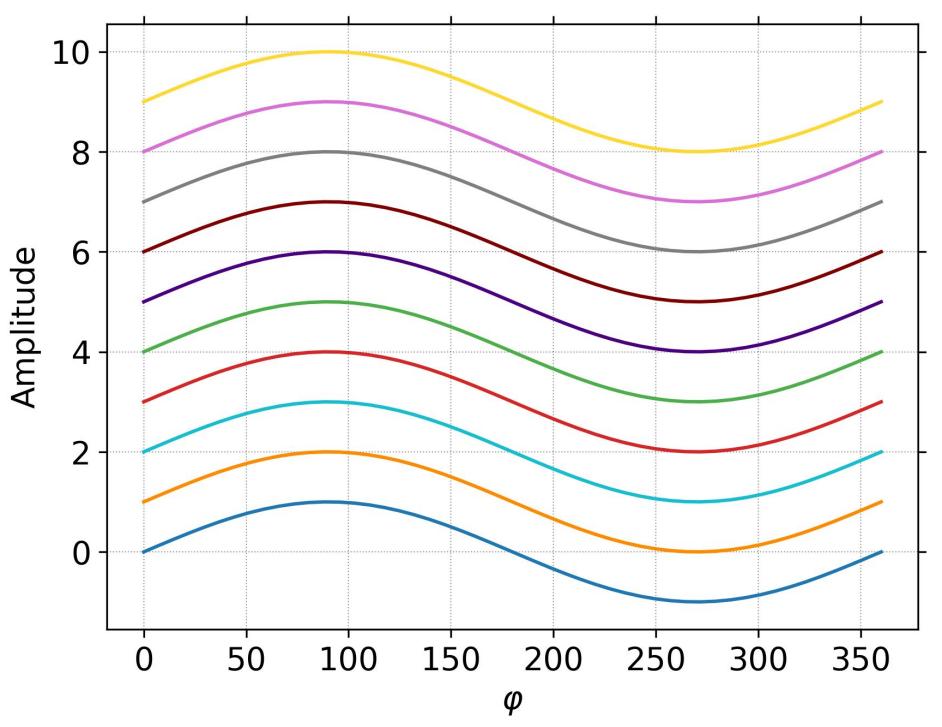
A .mplstyle file is a text file of key-value pairs where each key is the rcParam to modify. See, e.g., mpl4publication.mplstyle

- Can create & import list of style sheets for different sets of rcParams.
- **Need only include rcParams to be changed from defaults**
- Built-in style sheets can be called by name without full path

**Set for whole session:** plt.style.use('./your\_style.mplstyle')

**Set for one plot:** with plt.style.context('./your\_style.mplstyle'): ...





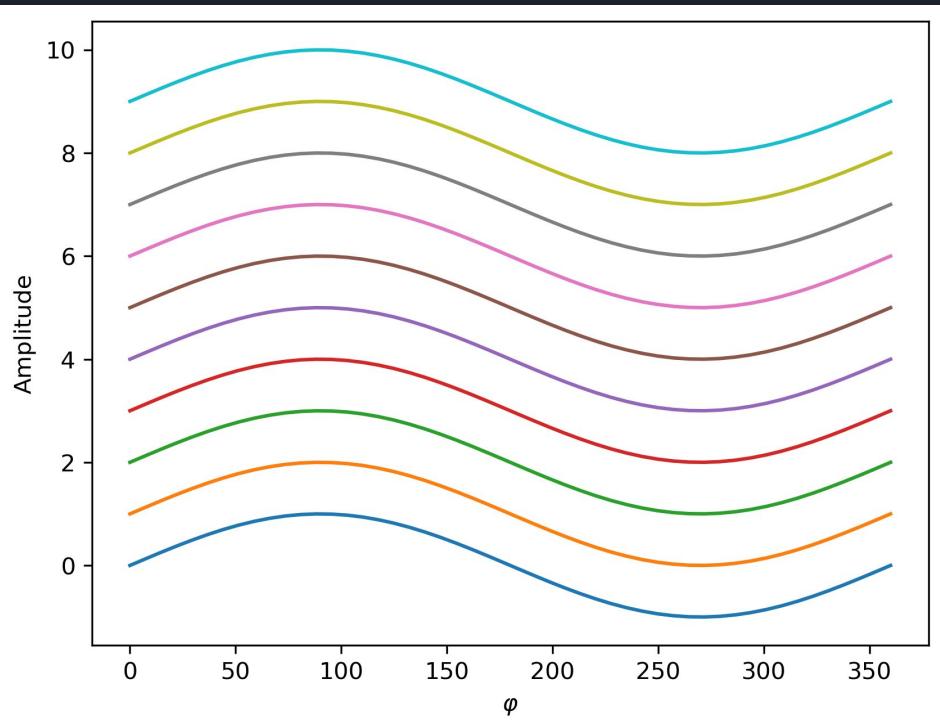
```

plt.figure(dpi=300)
with plt.style.context('..../mpl4publication.mplstyle'):
    for i in range(10):
        y = i+np.sin(np.linspace(0, 2 * np.pi))
        plt.plot(np.linspace(0,360,len(y)),y)
        plt.xlabel(r'$\varphi$')
        plt.ylabel('Amplitude')
plt.show()

#mpl.rcParams.update(mpl.rcParamsDefault)
plt.figure(dpi=300)
y = np.sin(np.linspace(0, 2 * np.pi))
for i in range(10):
    y = i+np.sin(np.linspace(0, 2 * np.pi))
    plt.plot(np.linspace(0,360,len(y)),y)
    plt.xlabel(r'$\varphi$')
    plt.ylabel('Amplitude')
plt.xlabel('Phase Angle')
plt.ylabel('Amplitude')
plt.show()

```

Example with `mpl4publication.mplstyle` (plot from `with` clause)



```

plt.figure(dpi=300)
with plt.style.context('..mpl4publication.mplstyle'):
    for i in range(10):
        y = i+np.sin(np.linspace(0, 2 * np.pi))
        plt.plot(np.linspace(0,360,len(y)),y)
        plt.xlabel(r'$\varphi$')
        plt.ylabel('Amplitude')
    plt.show()

#mpl.rcParams.update(mpl.rcParamsDefault)
plt.figure(dpi=300)
y = np.sin(np.linspace(0, 2 * np.pi))
for i in range(10):
    y = i+np.sin(np.linspace(0, 2 * np.pi))
    plt.plot(np.linspace(0,360,len(y)),y)
    plt.xlabel(r'$\varphi$')
    plt.ylabel('Amplitude')
plt.xlabel('Phase Angle')
plt.ylabel('Amplitude')
plt.show()

```

Example with `mpl4publication.mplstyle` (after with clause)



LUND  
UNIVERSITY

# Questions?