Matplotlib for Publication

Rebecca Pitts

You may ask yourself...

- Which plot format best communicates my research results?
- How can I combine or overlay related data without sacrificing readability?
- How do I work with poorly-supported axis scales?
- How do I make my figures typographically consistent?
- How can I make my figures more accessible to readers with color blindness or dyslexia?
- What might be lost if my figure is printed (e.g., in grayscale or resized for A4 paper)?

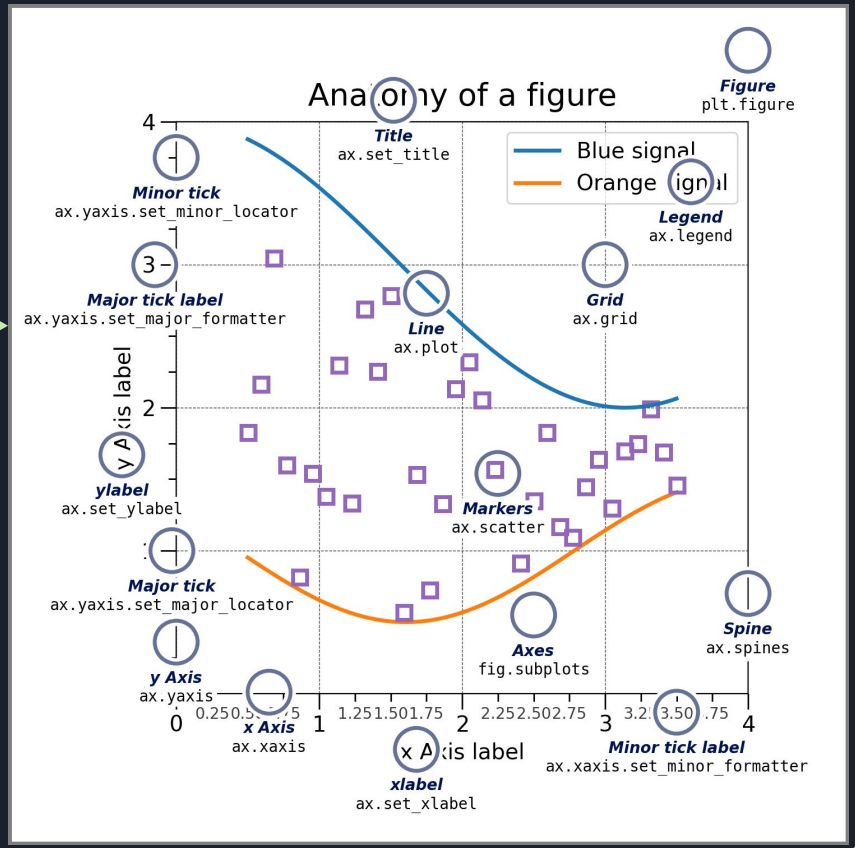
Definitions

Be familiar with “args” (positional arguments, usually mandatory), “kwargs” (keyword arguments, usually optional), & terms on this plot

2 ways to declare axes objects:

1. fig, ax = plt.subplots(nrows=nrows, ncols=ncols, **kwargs)
2. fig = plt.figure(**kwargs)
   ax=fig.add_subplot(nrows, ncols, n)

Former is preferred & better documented. Use plt.plot() for quickly viewing single datasets.



Overview

PART I

When choosing a plot type, consider:

1. The shape of your data
2. The number of parameters or dimensions
3. The number of points
4. The distribution of data in the parameter-space

PART II

The most printworthy figures have:

1. Typeset, easily readable text
2. Colorblind-friendly color maps and palettes
3. Consistent style throughout your paper (same fonts, grids, etc.)

Helps to use a .mplstyle file!

Part 1. Choosing a Plot Type

Standard representations for some data types:

Data type	Plot Format
Probabilities	Histograms or Kernel Density Estimations (KDEs)
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 or (pseudo)color images

When choosing a plot format, consider:

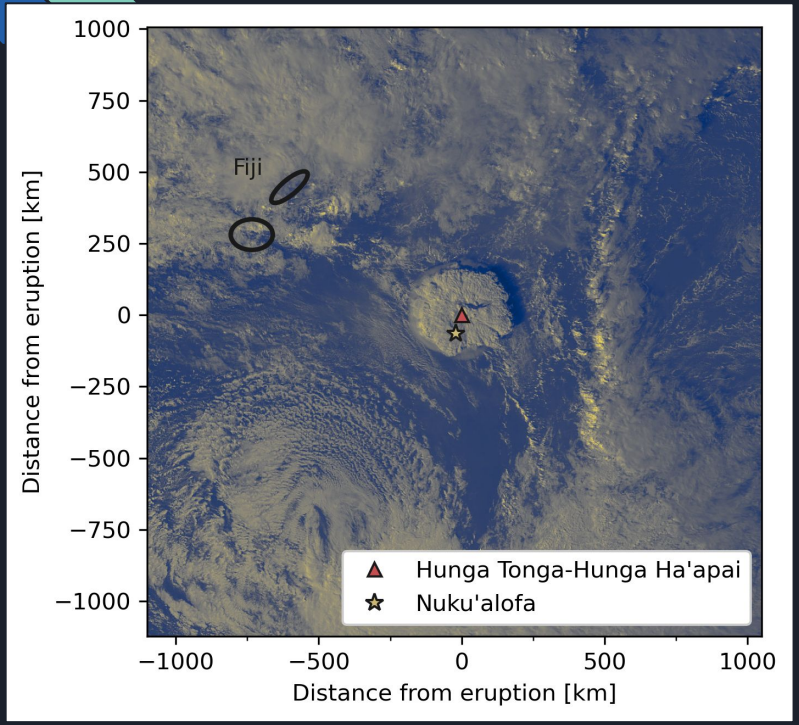
*Note: I assume there is a general understanding of which plots are fit for purpose for qualitative vs quantitative data.

Regularity

N-D array of values on a regular grid → image, field, or surface-like formats (2D or 3D projections)

- Challenge for 3D: find clearest, most informative viewing angles/slices that fit in your article
  - Not much to say here: it's trial and error.
- Online-only content: recommend interactive or animated graphics for 3D or temporally varying data

Image demo




```
import requests
from PIL import Image as img
mpl.rcParams.update(mpl.rcParamsDefault)
from matplotlib.patches import Ellipse

url='https://upload.wikimedia.org/wikipedia/commons/thumb/0/0b/Tonga_Vo
photo = np.asarray(img.open(requests.get(url, stream=True).raw))

fig,ax=plt.subplots(dpi=200, figsize=(4,4))
photplt = ax.imshow(photo,cmap=plt.colormaps['cividis'],
                    extent=[2.1*-524,2.1*500,2.1*-535,2.1*479])
ax.plot(0,0,'r^',ls='none',mec='k',label="Hunga Tonga-Hunga Ha'apai")
ax.plot(-20,-65,'y*',ls='none',mec='k',ms=8, label="Nuku'alofa")
ax.set_xlabel('Distance from eruption [km]')
ax.set_ylabel('Distance from eruption [km]')

viti=Ellipse( (-734,281), 147,107, fill=False, transform=ax.transData,
             ec='k', lw=2)
vanua=Ellipse( (-602,447), 160,53, angle=40, fill=False,
              transform=ax.transData, ec='k', lw=2)
ax.add_artist(viti)
ax.add_artist(vanua)
ax.xaxis.set_minor_locator(mpl.ticker.MultipleLocator(250))
ax.annotate('Fiji',(-800,490),color='k',xycoords=ax.transData)
ax.legend(loc=4,framealpha=1)
plt.show()
```

imshow() vs. other image-like formats

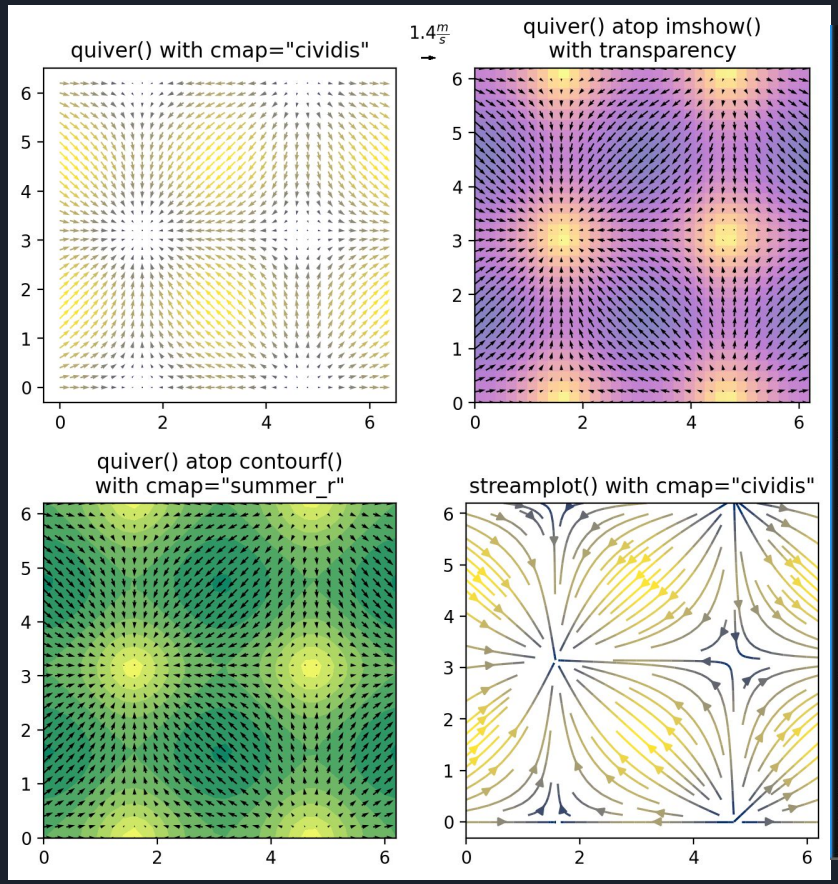
Function →	imshow()	pcolormesh()	pcolor()
Pixel shape	Square	Any quadrangle	Any quadrangle
Accepts RGB(A) input?	Yes	Yes	No
Supports masks	Values only	Values only	Coordinates & values
Supports map projections	Yes	No	No
# of interpolation options	19	4	3
Relative processing speed	Fastest	Medium	Slowest

Other Grid Formats

3 options for vector fields: streamplot(), quiver(), and barbs() (wind).

- Most “good” color maps will make some quiver() or barbs() arrows hard to see (colormaps work OK with streamplots). Recommendations:
  - Plot black arrows on translucent imshow() colormap of vector magnitudes (demo incoming)
  - Put quiverkey() outside plot area - no clean way to set the background color for the entire key, only the text.

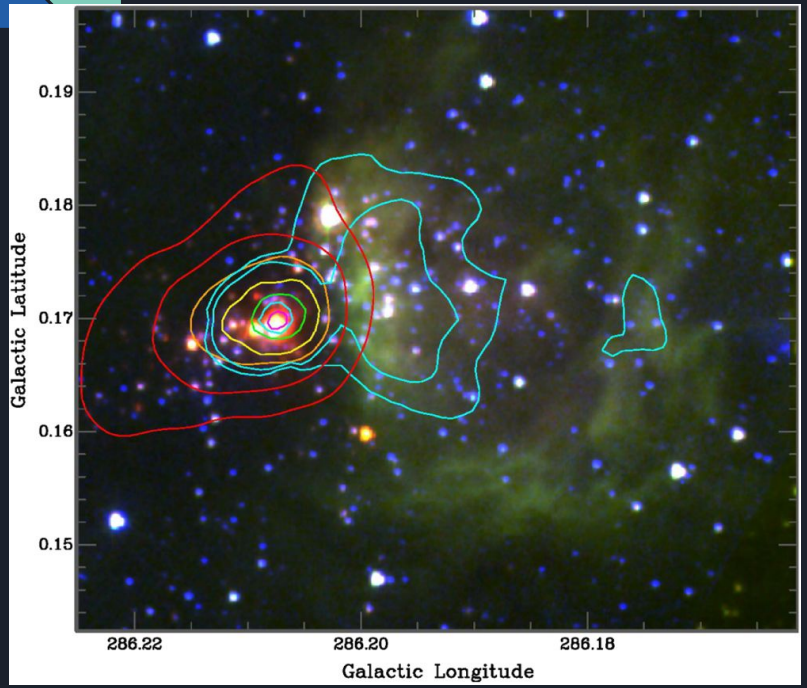
1 option (technically 2) for irregular grids: tricontour[f](). Otherwise must interpolate to regular grid (may have to anyway if data are in logspace)




```
X, Y = np.meshgrid(np.arange(0, 2 * np.pi, .2), np.arange(0, 2 * np.pi, .2))
U = np.cos(X)
V = np.sin(Y)

fig, axs = plt.subplots(ncols=2, nrows=2,dpi=200,figsize=(8,8))
fig.subplots_adjust(hspace=0.3)
M = np.hypot(U, V)
# Scale is inverse. Width is fraction of plot size; start around ~0.005
Q = axs[0,0].quiver(X, Y, U, V, M, scale_units='xy', scale=5, width=0.004,
                    cmap=plt.colormaps['cividis'])
qk = axs[0,0].quiverkey(Q, 0.51, 0.89, np.max(M),
                        r'${:.1f} \frac{{m}}{{s}}$'.format(np.max(M)),
                        labelpos='N',coordinates='figure')
#labelpos can be N, E, S, or W
axs[0,0].set_title('quiver() with cmap="cividis"')

C2 = axs[0,1].imshow(M,cmap='plasma_r',alpha=0.5,
                     extent=[np.min(X),np.max(X),np.min(Y),np.max(Y)])
Q2 = axs[0,1].quiver(X, Y, U, V, scale_units='inches',scale=12,width=0.004)
axs[0,1].set_title('quiver() on imshow()\nwith transparency')

C3 = axs[1,0].contourf(X, Y, M, levels=8, cmap='summer_r', zorder=0)
Q3 = axs[1,0].quiver(X, Y, U, V, scale_units='xy', scale=6, width=.004)
axs[1,0].set_title('quiver() on contourf()\nwith cmap="summer_r"')

SP = axs[1,1].streamplot(X, Y, U, V, color=M, linewidth=1.2,cmap='cividis')
axs[1,1].set_title('streamplot() with cmap="cividis"')
plt.show()
```

Quiver and streamplot demos

Dimensionality - Unstructured Data



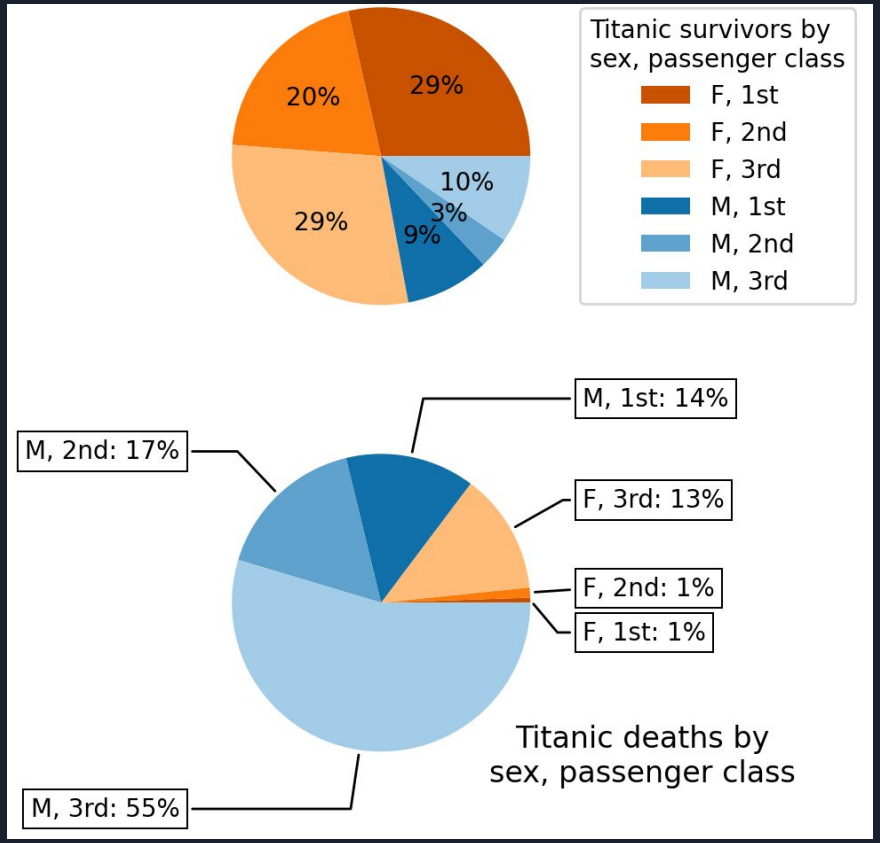
Greatest range of options for 3(-4) dimensions or parameters. You could...

- Add or overplot with extra axes
- Use different markers, colors, or line styles for extra qualitative variables
- Group, stack, color, and/or crosshatch bars or units of a proportional area chart
- Plot contours or markers on a RGB image
- Etc.

Let’s start with a non-example (+ alternatives)

Why should you NOT use pie or donut charts? Per many data science consultancy studies and other institutions...

- Pie & donut charts are inefficient (low info for space occupied)
- Hard to add other parameters except by grouped colors
- 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 or by numerical brackets.
- **State of belonging less important than relative or absolute quantities**

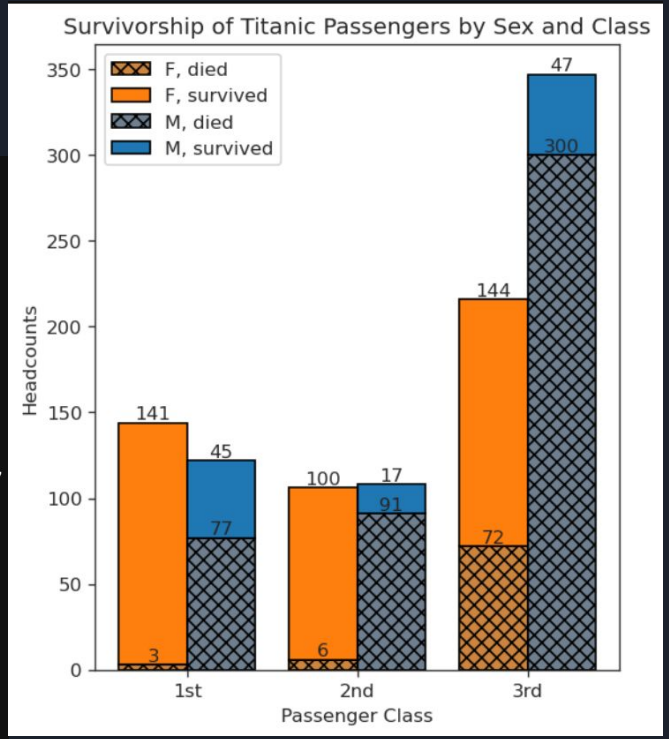


Pie chart for comparison

Better Alternatives: Bar Charts

If you're more interested in raw counts…

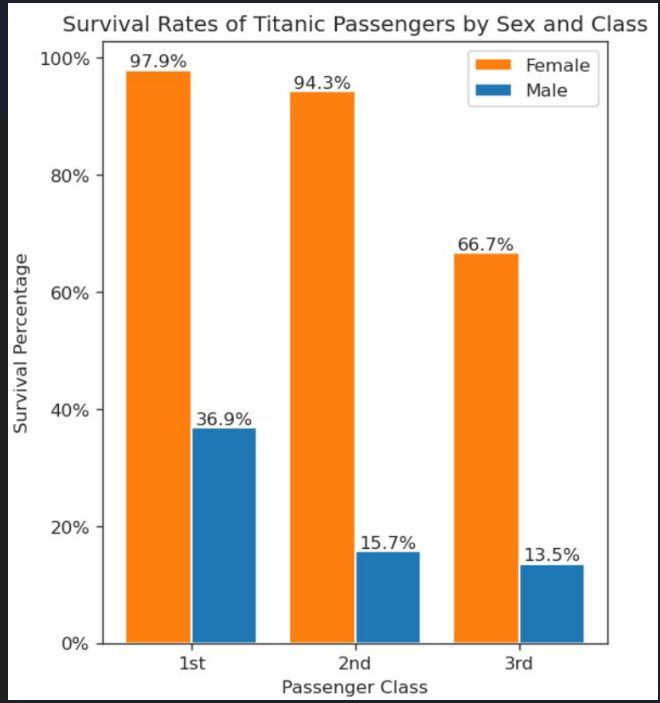
```
fig, (ax1,ax2) = plt.subplots(dpi=120,figsize=(12,6),ncols=2)
plt.subplots_adjust(wspace=0.4)
x = np.arange(3)  # classes
width = 0.4  # the width of the bars (there will be 2)
multiplier = 0

for sex, npclass in dead.items():
    offset = width * multiplier
    dead_recs = ax1.bar(x + offset, npclass, width, label=sex.capitalize()+', died',
                        ec='k',color='slategrey' if sex == 'm' else 'peru', hatch='xxx')
    ax1.bar_label(dead_recs, padding=0)
    live_recs = ax1.bar(x + offset, live[sex], width, label=sex.capitalize()+', survived',
                        ec='k',color='tab:blue' if sex == 'm' else 'tab:orange',
                        bottom=npclass)
    ax1.bar_label(live_recs, padding=0, labels=live[sex])
    multiplier += 1

ax1.set_xticks(x+0.5*offset, ['1st','2nd','3rd'])
ax1.set_title('Survivorship of Titanic Passengers by Sex and Class')
ax1.set_xlabel("Passenger Class")
ax1.set_ylabel("Headcounts")
ax1.legend()
```



Better Alternatives: Bar Charts

If you're more interested in percentages…

```
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()
```



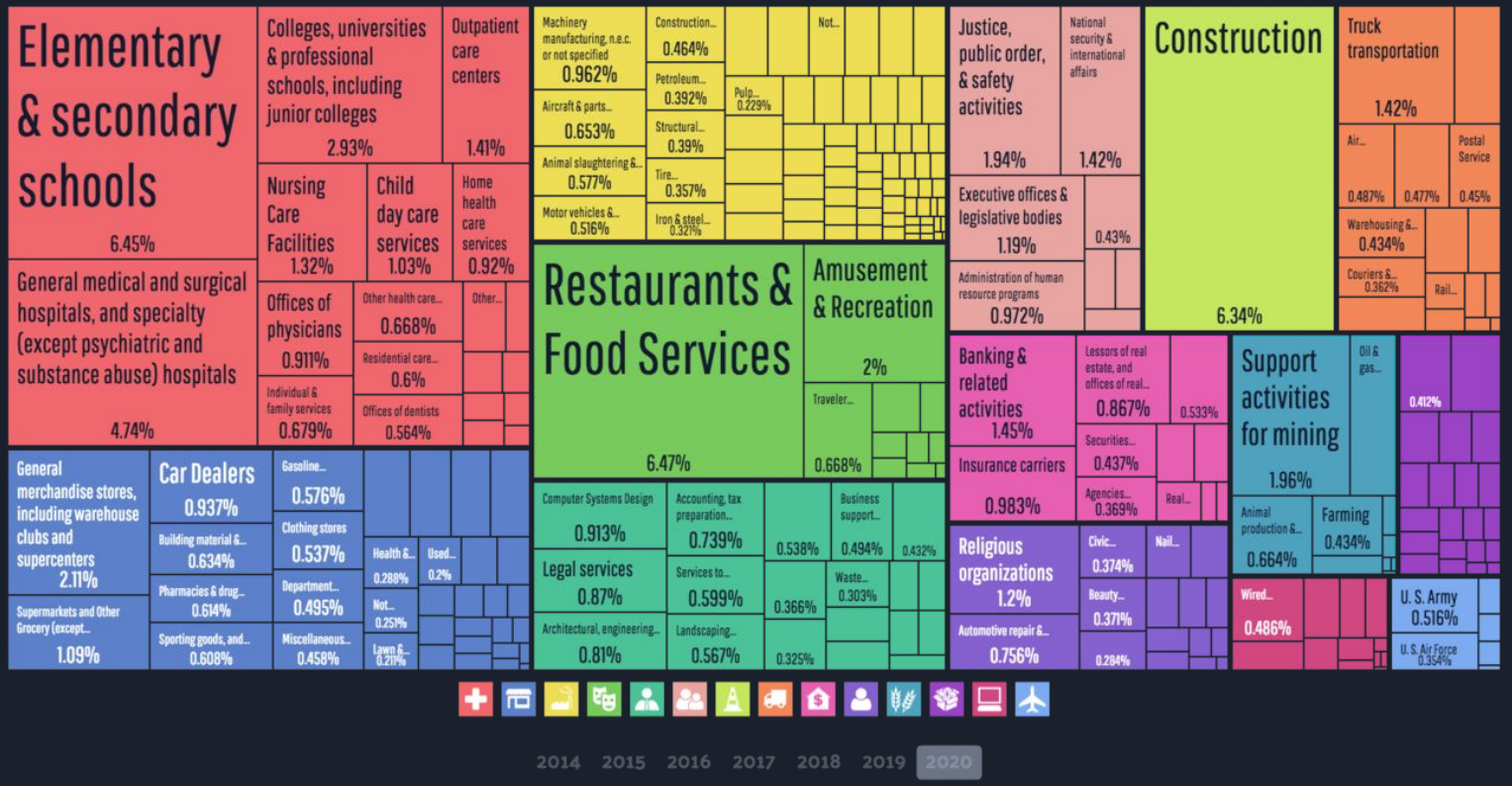
Other rectilinear proportional-area charts

2 other good options:

- Treemaps (next slide)
- Waffle charts (example below)

Both much easier to make with external packages: Squarify (treemaps) & PyWaffle (used below)

https://github.com/gyli/PyWaffle



Treemap of employment in Oklahoma in 2020 (public)

Dimensionality - Unstructured Data 2

More dimensions/parameters can make your choice easier because:

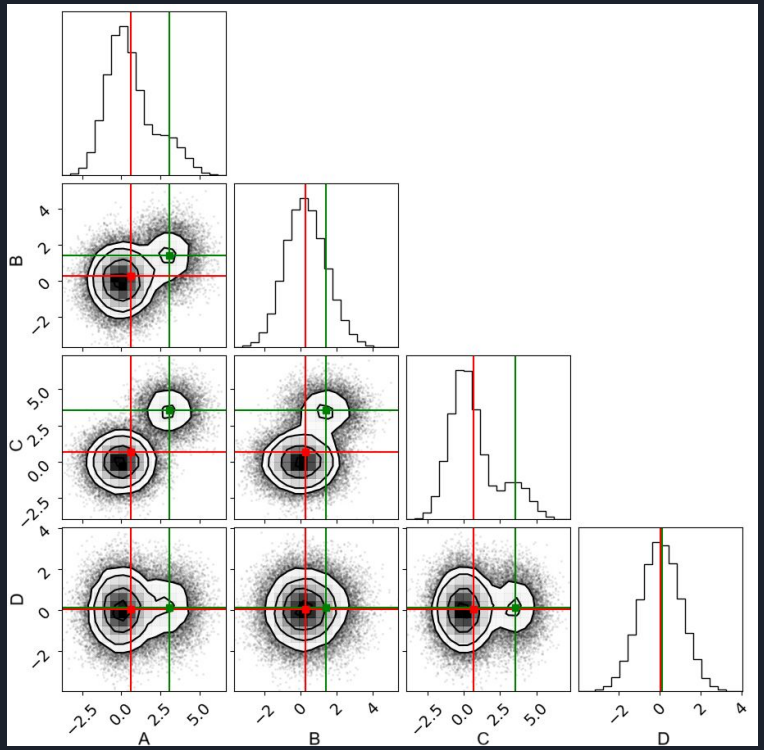
- Display options that can accommodate all variables are limited.
- **Journals have hard size and resolution limits.**

For **4-6 numerical parameters**, if data are scattered or include outputs of Markov Chain Monte-Carlo (MCMC)* simulations, corner plots or pair grids are typically best - & best made with extra packages

*MCMC simulations are a way of modelling the probability distributions of model parameters, both individually and given the distributions of other parameters, in the Bayesian framework of statistics where, instead of the data, it's the model that is assumed to be uncertain.

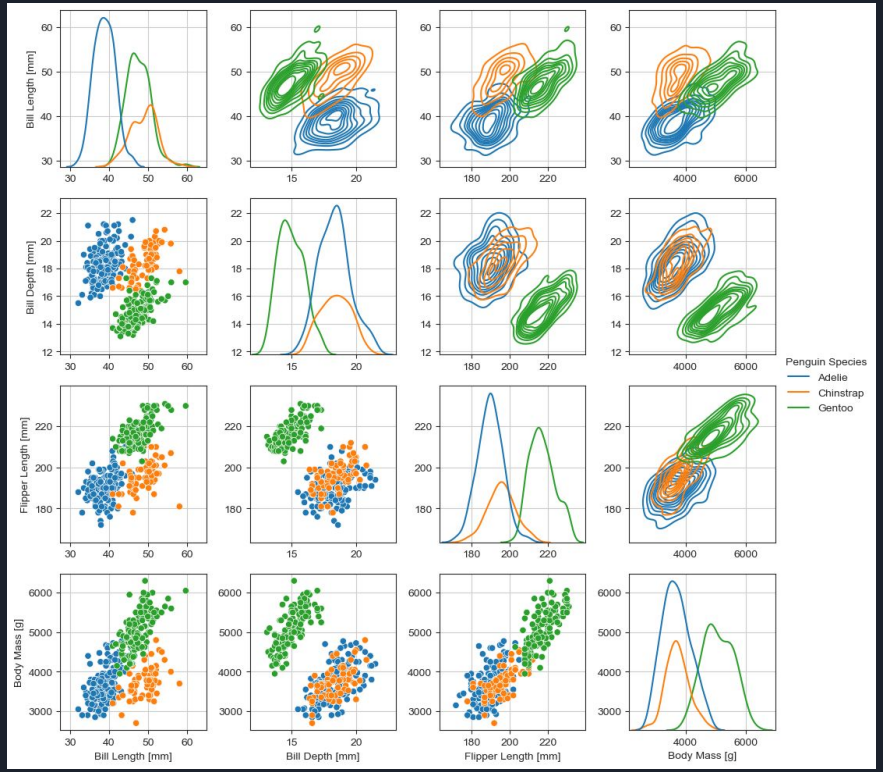
Corner Plots

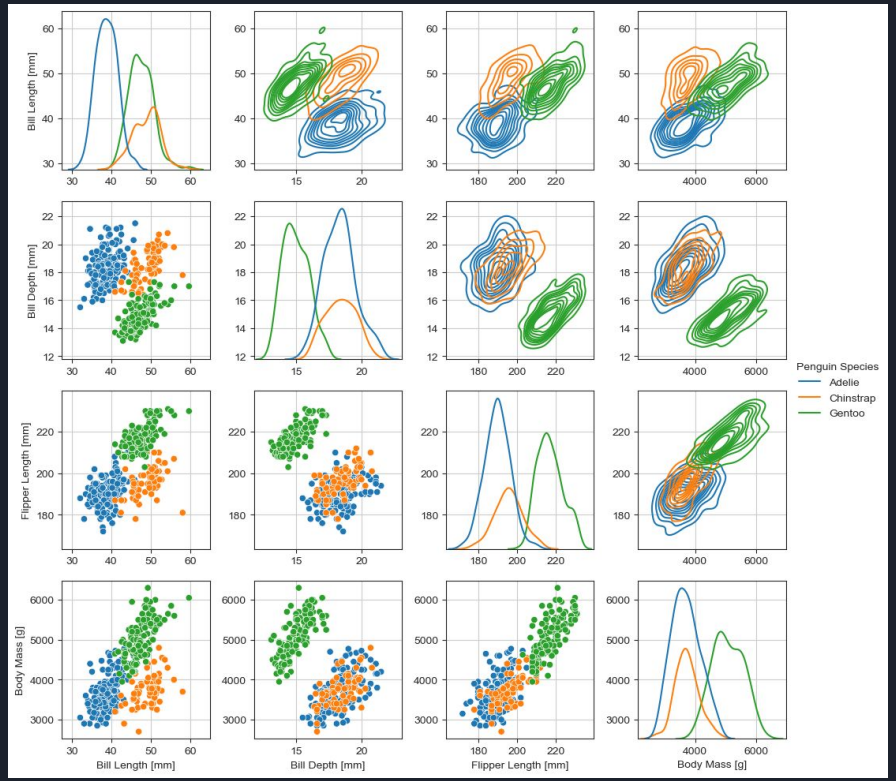
- Standard plot format for MCMC sims of any size
- Good for 2D projections of large datasets with up to ~5 parameters
- Lightweight external package - **corner.py** is pure matplotlib
- Note switch from scatter to contours at density threshold



Pair Plots/Grids

- Easy to make with Seaborn package - if you know Pandas
- Ideal for numerical variables + 1 categorical variable (hue)
- MANY format options for diagonal, upper & lower off-diagonal panels
- **Downside:** have to override axis labels to typeset them well (tricky)





seaborn.PairGrid()

Main

```
import pandas as pd
import seaborn as sb

penguins = sb.load_dataset("penguins")
print(penguins.head()) #standard preview function for pandas dataframes

g = sb.PairGrid(penguins, hue='species', diag_sharey=False, #corner=True,
                despine=False)
g.map_lower(sb.scatterplot)
g.map_diag(sb.kdeplot)
g.map_upper(sb.kdeplot)
g.add_legend(title='Penguin Species',loc='center right')#,fontsize=14)
sb.set_style('ticks')
plt.subplots_adjust(wspace=0.3, hspace=0.2)
```

Typesetting

```
import string
for i in range(4):
    for j in range(4):
        try:
            xlabel = g.axes[i,j].xaxis.get_label_text()
            ylabel = g.axes[i,j].yaxis.get_label_text()
            g.axes[i,j].set_xlabel(string.capwords(xlabel.replace('_',' ')
                                                   .replace('mm','[mm]')
                                                   .replace(' g',' [g]')))
            g.axes[i,j].set_ylabel(string.capwords(ylabel.replace('_',' ')
                                                   .replace('mm','[mm]')
                                                   .replace(' g',' [g]')))
            g.axes[i,j].tick_params(axis='both',
                                    labelleft=True, labelbottom=True)
            g.axes[i,j].grid()
        except AttributeError:
            pass
```

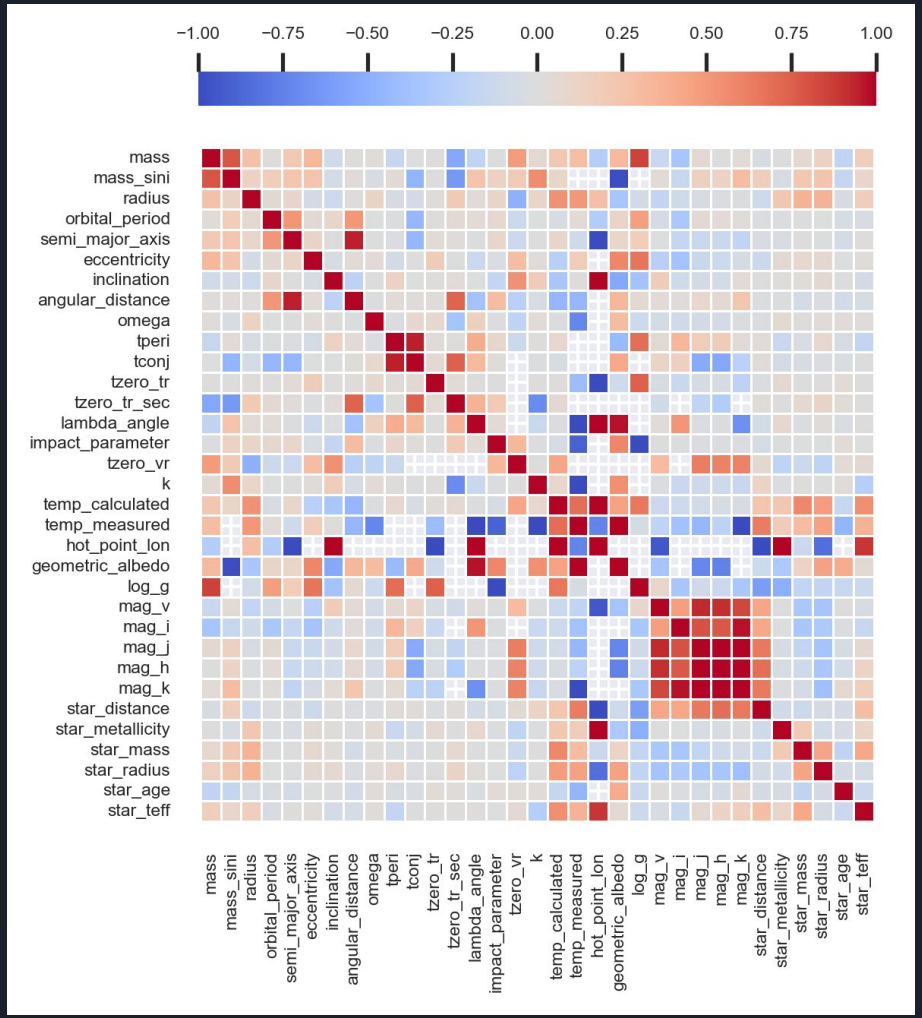
Larger ND Parameter Spaces

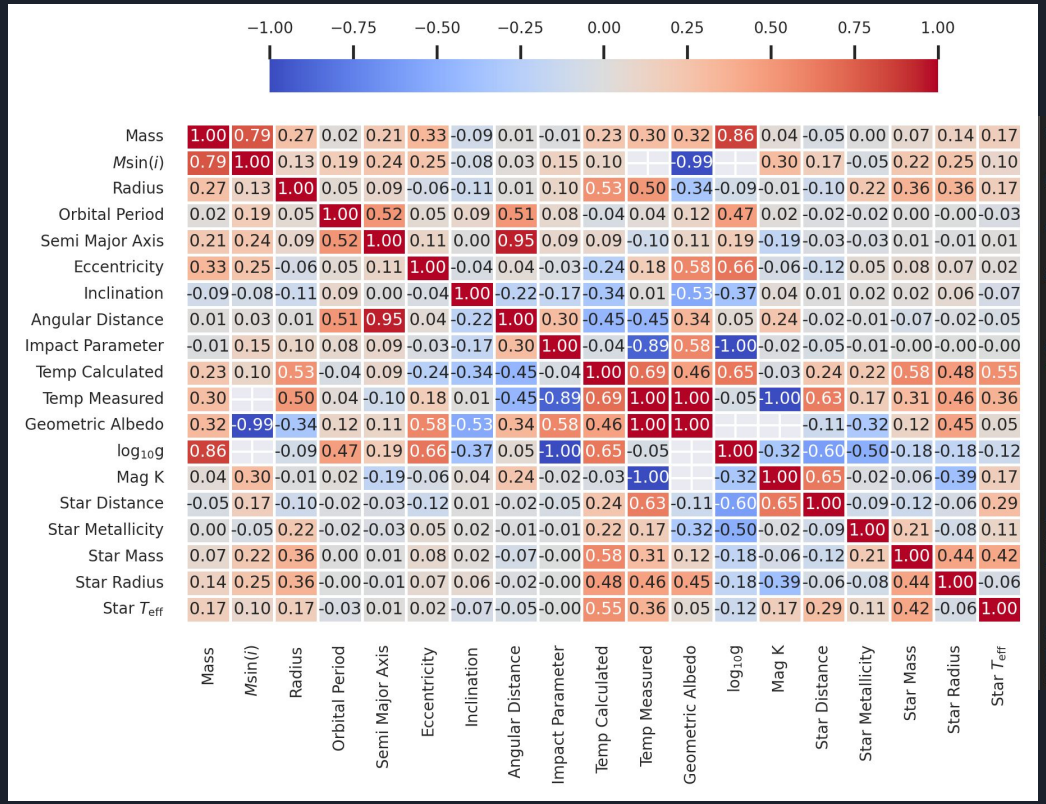
Options decline steeply for >6 parameters

- 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?
- If you absolutely must, use either imshow()-based heatmap (more flexible) or seaborn.heatmap() (more straightforward if you know Pandas)

Heatmap demo with Seaborn

```
import requests
url='https://exoplanet.eu/catalog/csv/'
exops = pd.read_csv(url) #98 columns!
dropkeys = [title for title in exops.columns if
            'error' in title or 'name' in title]
exops1=exops.drop(columns=[*dropkeys,'planet_status','ra','dec',
                           'discovered']) #paring columns down
corrs=exops1.corr(numeric_only=True)
print(corrs.shape) # = 33 x 33

plt.figure(dpi=300)
sb.set_style('ticks')
sb.set(font_scale=0.5)
sb.heatmap(corrs, cmap="coolwarm", annot=False, linewidth=.3,
           cbar_kws={'shrink':0.6, 'location':'top'},
           xticklabels=True, yticklabels=True, square=True)
plt.show()
```






```
dropkeys2 = [title for title in exops1.columns if
          'tzero' in title or 'mag_' in title]
exops2=exops1.drop(columns=[*dropkeys2[:-1],'tperi','tconj','omega',
                            'star_age','hot_point_lon','lambda_angle','k'])
corrs2=exops2.corr(numeric_only=True) # 19 x 19

annot_labels = np.ma.masked_inside(corrs2.to_numpy(),-0.1,0.1)
replacers = {'Mass Sini':r'$M$sin($i$)', 'Log G':r'log$_{{10}}$g',
             'Teff':r'$T_{{\mathrm{{eff}}}}$'}
axlabs = corrs2.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'$T_{{\mathrm{{eff}}}}$')

plt.figure(dpi=300)
sb.set(font_scale=0.6)
hm = sb.heatmap(corrs2, cmap="coolwarm", annot=annot_labels,fmt='.2f',
           linewidth=0.75, cbar_kws={'shrink':0.8, 'location':'top'},
           xticklabels=axlabs, yticklabels=axlabs, square=False)
sb.set_style('ticks')
plt.show()
sb.set(font_scale=1)
```

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

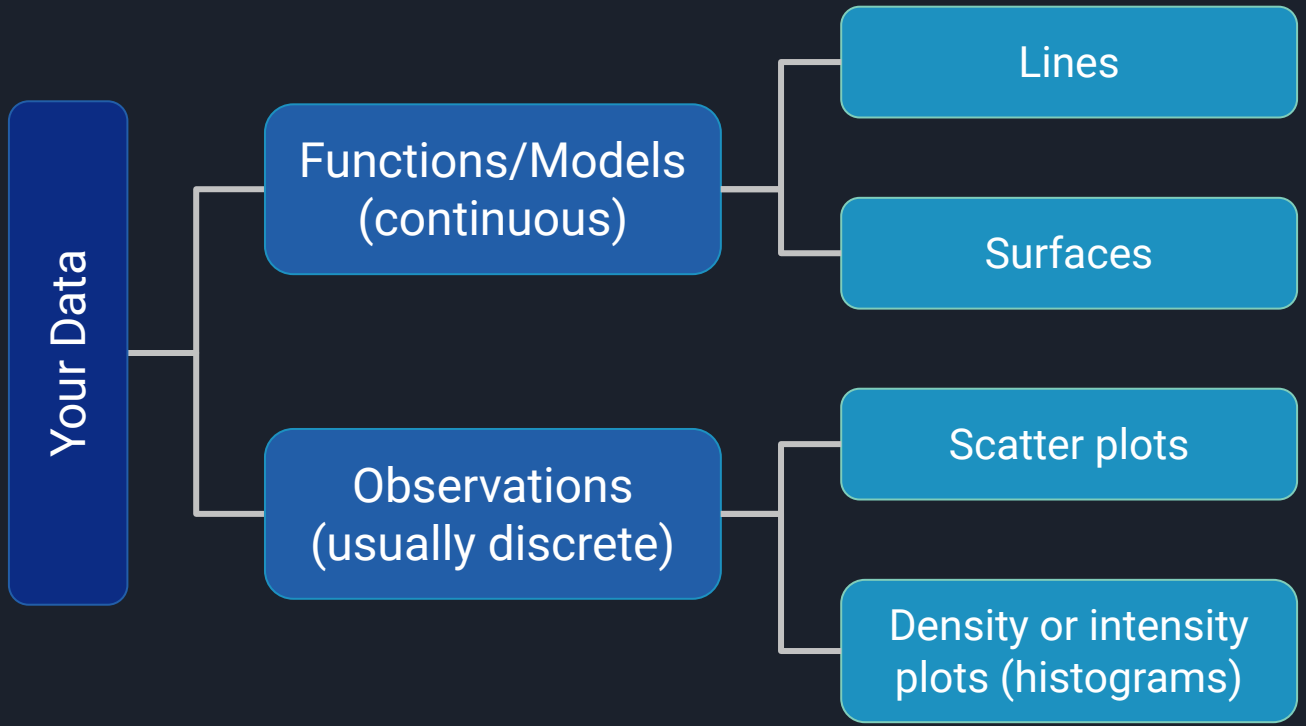
Better if you don't plot more variables than heatmap() can annotate

Quantity & Density

Best approach depends on:

- Number of data points or lines
- Distribution across parameter space
  - This also determines if or where you include a legend
- What other data/models need to be shown on the same plot
- **What features you want to emphasize about the data**

Quantity & Density

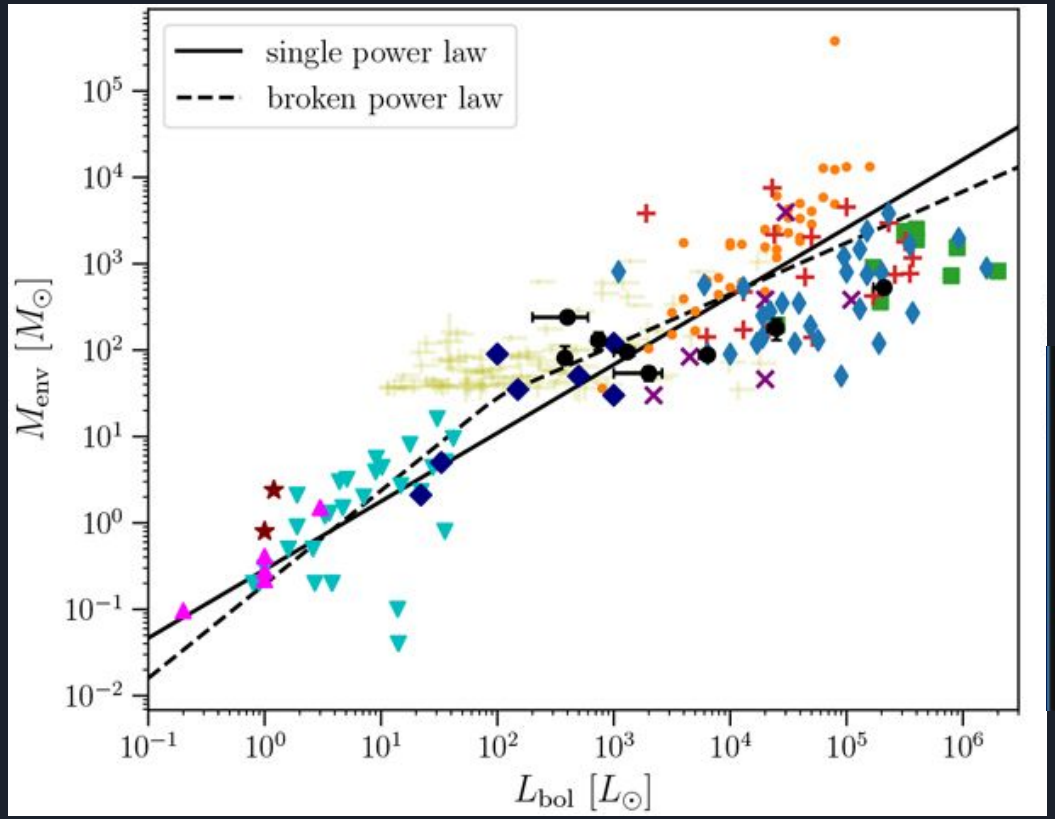


Overcrowding is the biggest risk you face with scatter plots or plots featuring multiple models.

When adding data to a figure that other research groups have contributed to, **it is OK to deemphasize other data & highlight your own.**

To emphasize your contributions to a common plot, consider...

- 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 a custom color cycler*.
    - *Module cycler works with colors & line styles, but adding a marker cycle disables the whole cycler instance → use itertools.cycle
- Making line(s) you want readers to focus on thicker & more solid, or diminishing less important lines by making them dotted.



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

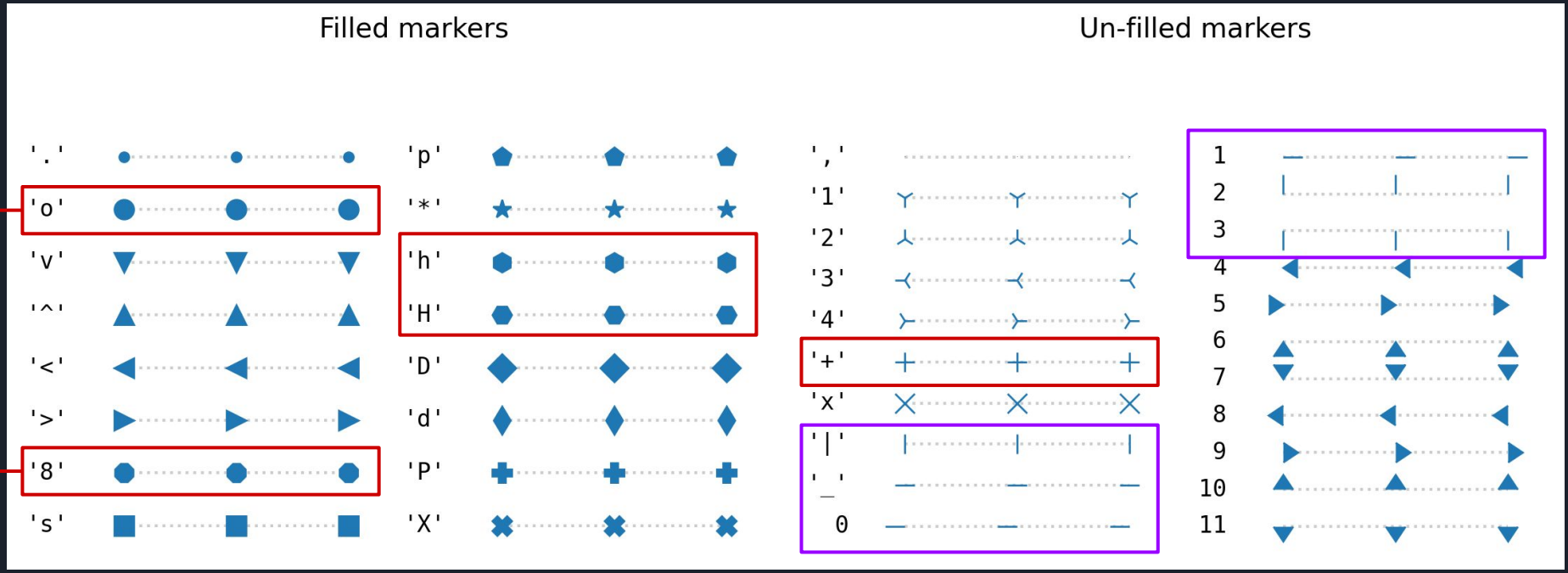
```
from itertools import cycle
c_cycle = cycle(['tab:orange','tab:blue','tab:red',
                 'tab:green','tab:cyan','m',# m = magenta
                 'navy','maroon','purple'])
m_cycle = cycle(['.', 'd', '+',
                 's', 'v', '^',
                 'D', '*', 'x'])
#pretend we have a 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)

Scattered Data by the Numbers 1.

A few × 10 to a few × 100 data points

- Vary color (c kwarg of scatter()) &/or marker style to denote different datasets or values of a categorical parameter
- Only color by numerical 3rd parameter (cmap kwarg of scatter()) if it's correlated with at 1+ axis parameter
- If feasible, vary colors & markers together for colorblind accessibility & so data are still distinct in grayscale
    - Keep in mind: some markers are hard to tell apart



- **Avoid reusing shapes with different rotations** (can be problematic for dyslexic readers), or octagons with big circles
- “+” can be confused for error bars if error bars don’t have caps.
- Reserve | & – for specialty plots (event plots, rug plots, etc.)

Scattered Data by the Numbers 2.

A few × 100 to a few × 1000 data points

Recommend scatter() with transparency or histogram with logarithmic bins if outliers are important

- Reduce scatter plot opacity: set alpha kwarg ~0.2-0.5
- **Many rules for histogram binning, but generally, distribute *n* data points over $n^{1/2}$ to $n^{1/3}$ bins** (empty bins in 2D space don't count)
  - hexbin() & hist2d() normally handle binning well internally
- Minimally overlapping distinct datasets can be shown with different colors or as line contours (harder)

Scattered Data by the Numbers 3.

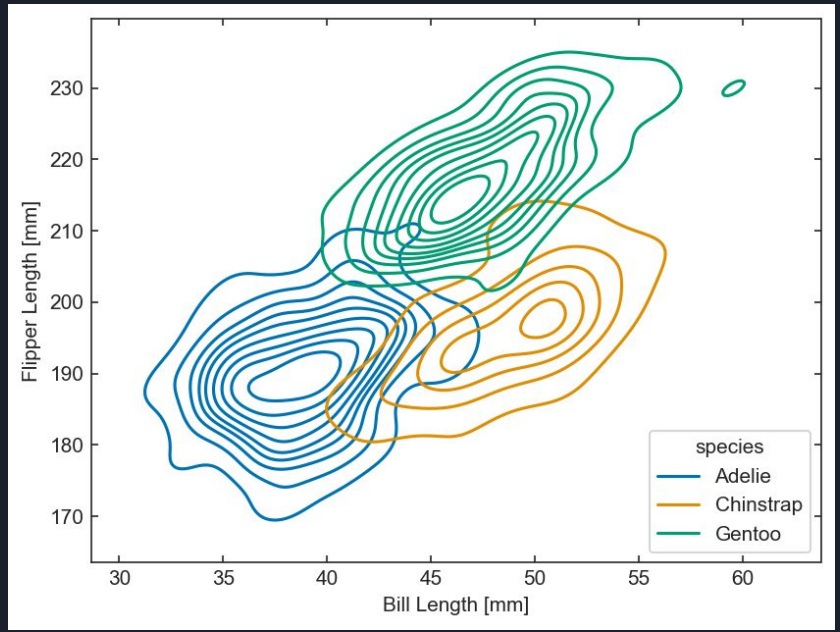
1000s of data points or more

Use histograms, Gaussian Kernel Density Estimations (KDEs), or contours, or corner.hist2d() if outliers are important

- Gaussian KDEs ≈ histograms smoothed by a treating every point as a bell curve of unit height and width = kernel bandwidth, & summing them up
  - **Pros:** continuous functions, essential for smooth contours of scattered data
  - **Cons:** require SciPy or Seaborn; slow processing

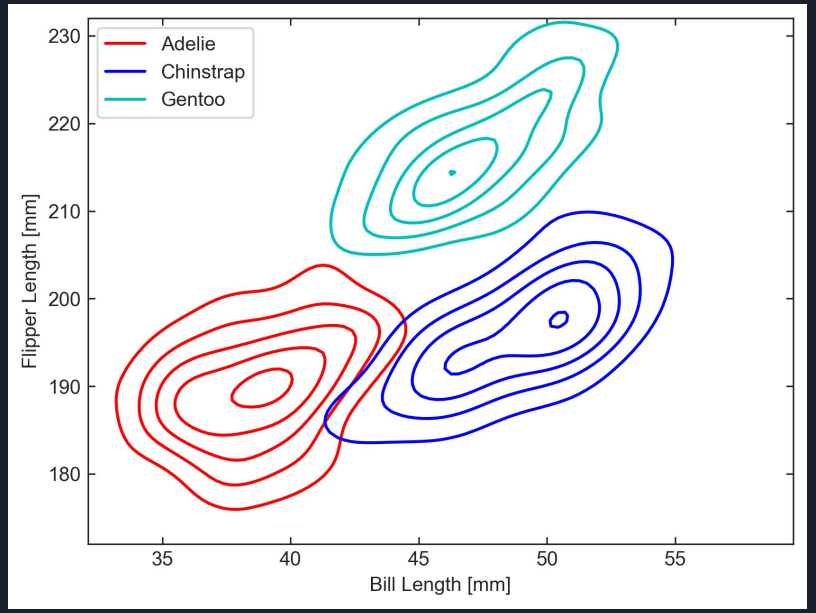
KDE example with Seaborn

```
plt.figure(dpi=150)
sb.set_style('ticks')
sb.set_palette('colorblind')
kde = sb.kdeplot(penguins, x='bill_length_mm',
                 y='flipper_length_mm', hue='species')
sb.move_legend(kde,'lower right')
plt.xlabel('Bill Length [mm]')
plt.ylabel('Flipper Length [mm]')
plt.tick_params(direction='in',right=True, top=True)
plt.show()
```



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()
```



Notes on Contours

Matplotlib's contouring functions, [tri]contour[f](), don't accept scattered data directly; must first make KDE on regular grid, & contour that (don't try to contour histograms!)

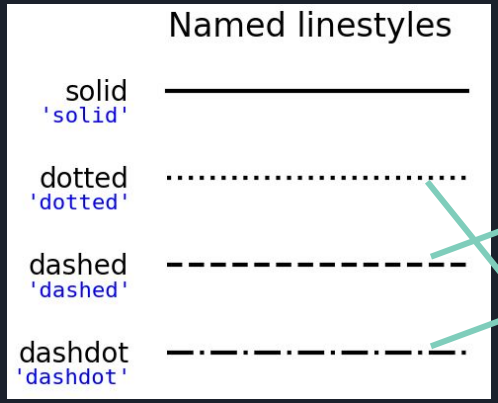
- Filled contours ([tri]contourf()) are more intuitive, less confusing if you need to plot linear or scattered data on top.
- Line contours are better to overlay on images, other contours, or other continuous 2D distributions.
- Legends are not natively supported for contours: if you put >1 set of contours on 1 plot, you have to make a proxy artist

Speaking 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** on a hierarchy, with minimal crossing
  - Many if data are correlated & evolving with a 3rd variable (e.g., monthly mean temperatures over years - let older curves fade out)

Lines Cont.



Line styles vary in clarity → unspoken hierarchy to consider for your narrative:

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

Uncertainty Handling

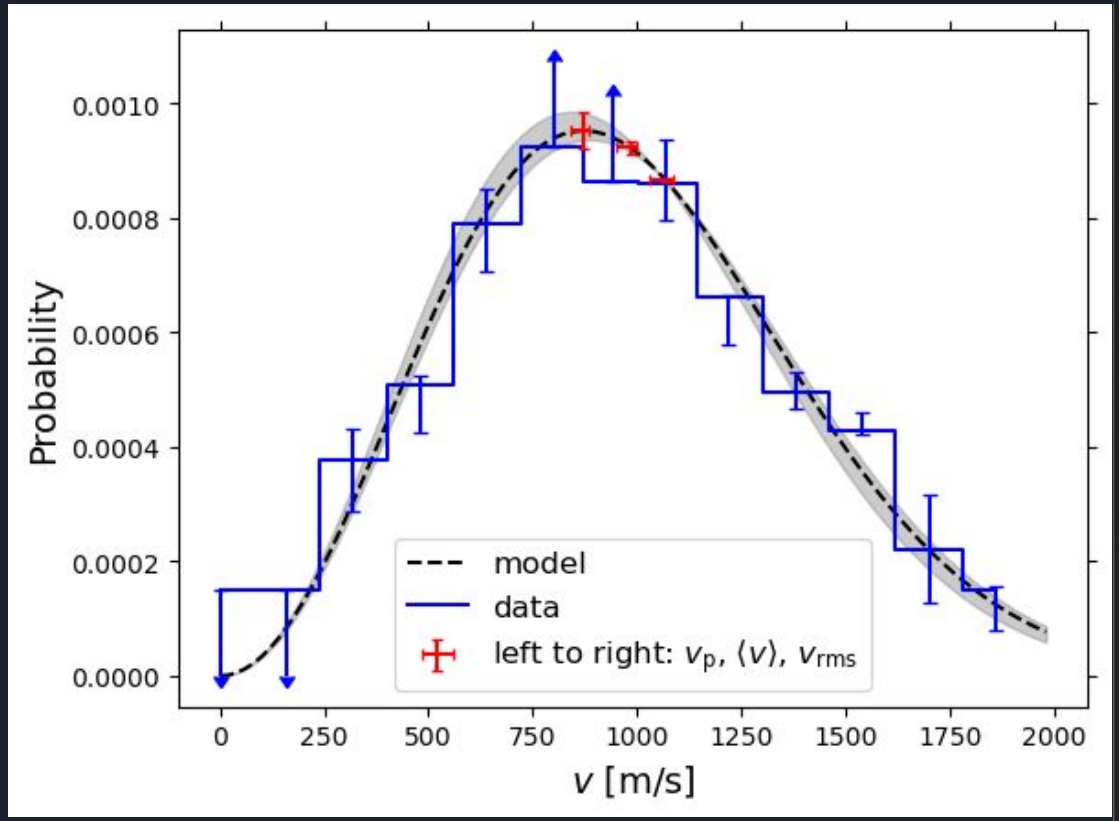
# of data points	Best error bars/limits format
Up to a few 10s	Error bars/limits on every point, with caps if large
Few 10s to ~100	Error bars on every point; omit caps
Few 100s	Error bars on every nth point (errorevery kwarg)
Many 100s & up	Representative error bar(s) in an unoccupied corner

Lines: favor fillbetween(x, yerr_lo, yerr_up) in a light or translucent color with the main line on top. Separate upper & lower margin lines are less intuitive.

Complex error bars

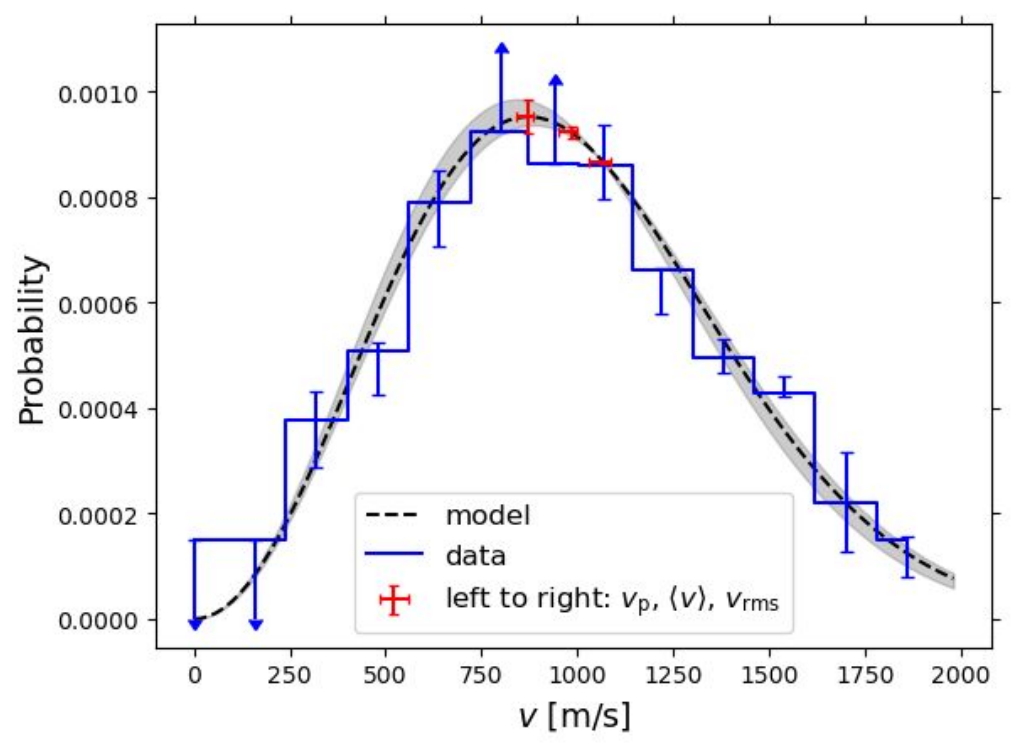
Error bars can be tricky if asymmetric or mixed with limits.

- 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).




```
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
```

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




```
perrs = np.array([np.random.default_rng()\
                  .uniform(0,1,len(vs[::8]))*0.0001,
                  np.random.default_rng()\
                  .uniform(0,1,len(vs[::8]))*0.0001])
plolims = np.zeros(probs.shape)
puplims = np.zeros(probs.shape)
puplims[0:2]=1
plolims[5:7]=1
perrs[1,5:7]=0.00015
perrs[0,0:2]=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 (cont.)

LUND
UNIVERSITY

Part 2. Global Format Constraints

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.
- Color bars, axis labels, & captions take up space, too, so roughly 4×6 subplot grid max

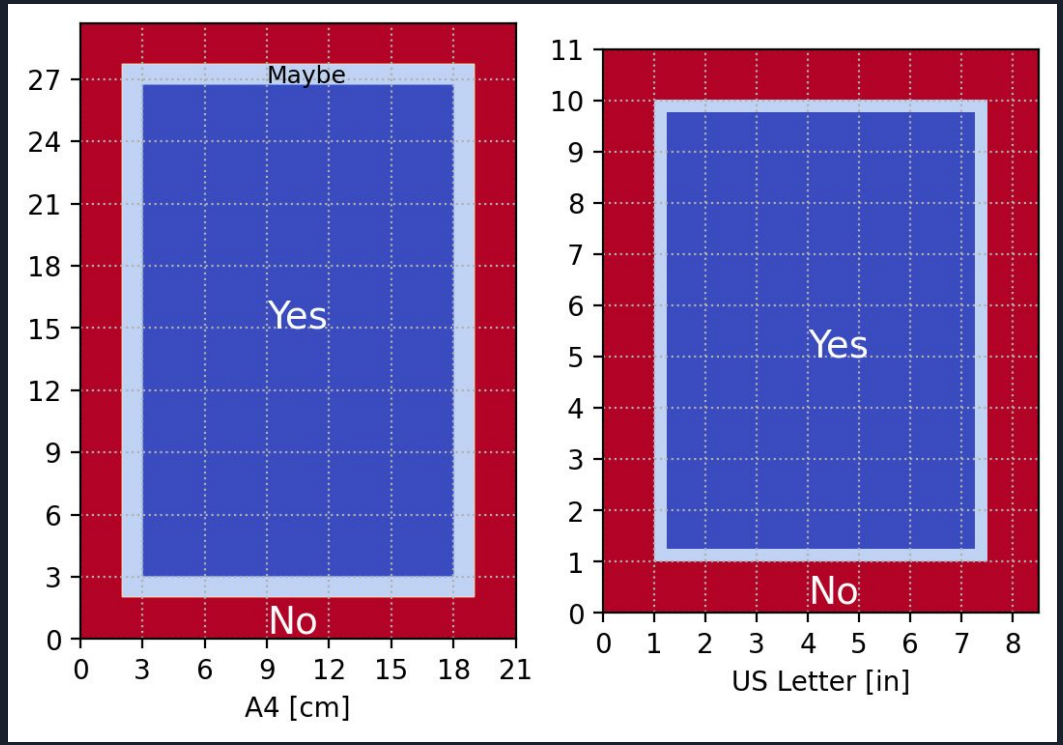


Figure Output Formats

Preferred image formats vary by journal. Matplotlib supports PDF, (E)PS, PNG, & JP(E)G.

- **Recommended:** PDF. Most widely supported & scales well.
- PS & EPS are some journals' top format choices, but require special viewing software & may not support transparency.
- **Matplotlib default:** PNG. Accepted but discouraged (large file sizes).
- JPG/JPEGs have small file sizes but degrade when edited or resized.
- PIL (pillow) module supports SVG and TIFF output formats

Figure Output Formats Cont.

Journals often request 300 dpi images. 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.

- Can set dpi=300 in plt.figure() or plt.subplots() at runtime, or in plt.savefig() or plt.imsave() for the saved version only
    - Set mpl.rcParams['figure.dpi'] = 300 to apply to all figures in current session (or until kernel restarts)
- Can adjust figure size & aspect ratio with figsize = (i,j) kwarg in figure(), subplots(), savefig(), and imsave(). *Note: figsize is in inches*

Fonts & Font Sizes

- No consensus on whether serif or sans-serif fonts are more readable.
- Pick a common font (or similar) & stick with it.
- Most journals set hard minimum font size ~9-10 pt.
- **11-14 pt font preferred.** If in doubt, err larger for older readers.

Setting Fonts with rcParams

```
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)
```

Matplotlib & LaTeX

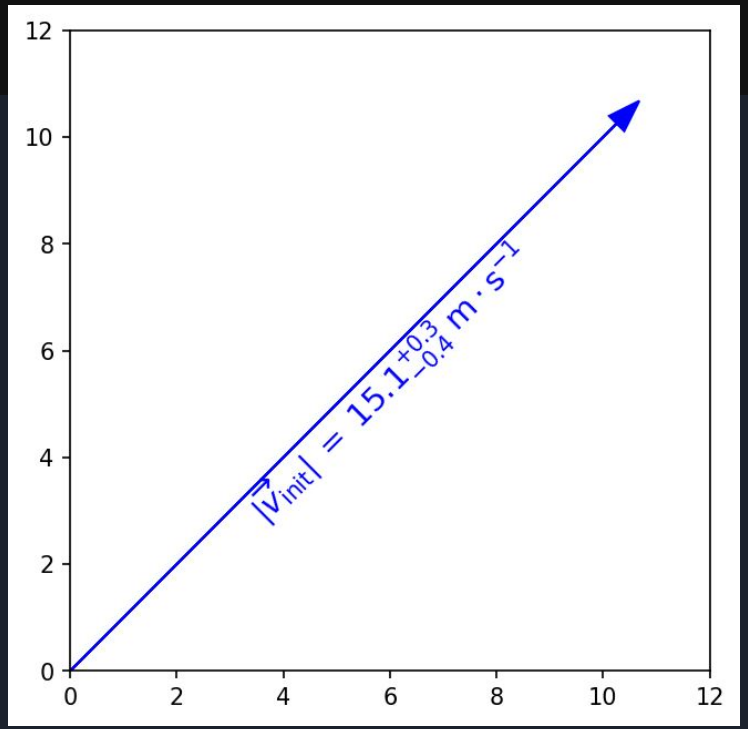
Need to know LaTeX basics to typeset plot text like journal text:

- LaTeX expressions must be bracketed by “$ $” (dollar signs)
- Syntax of most LaTeX functions to modify characters: “$\function{char}$”
    - Superscript (“^”) & subscript (“_”) operators can work without “{}” but only 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 online tables for Greek letters, operators, & other symbols

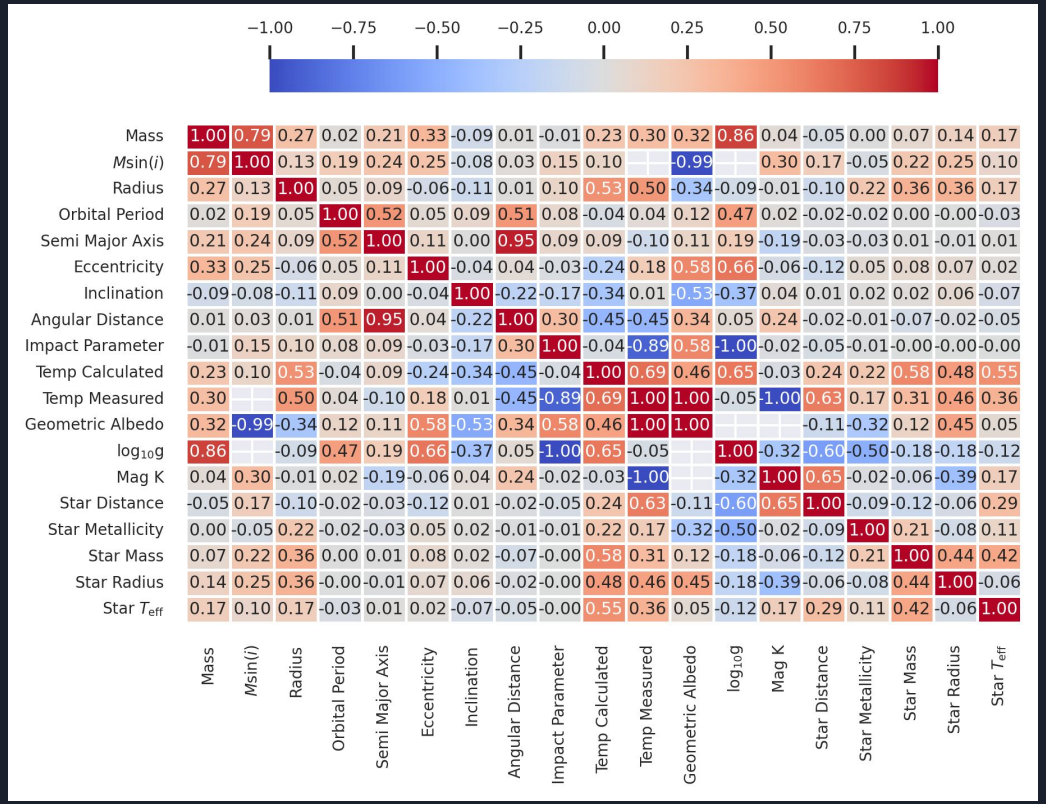
Matplotlib & LaTeX (Cont.)

- String insertion (‘{ }’.format()) creates conflict between Matplotlib & LaTeX → must double all {} associated with LaTeX commands, including nested braces
- Most Greek letters & ±/∓ symbols require the input string to have r before the first quote mark
- Operators with special meanings in LaTeX or Python (e.g. % and $) must have backslash(es) (\ for LaTeX commands, \\ for Python operators) to render as text.
- Spaces within $$ are not rendered: use \, or \; to insert them

```
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"$|\overrightarrow{{v}}_{{\mathrm{{init}}}}|$ = ${:.1f}_{{{:.1}}}^{{+{:.1}}}\;\mathrm{{m\cdot s}}^{{-1}}$".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.




```
dropkeys2 = [title for title in exops1.columns if
             'tzero' in title or 'mag_' in title]
exops2=exops1.drop(columns=[*dropkeys2[:-1],'tperi','tconj','omega',
                            'star_age','hot_point_lon','lambda_angle','k'])
corrs2=exops2.corr(numeric_only=True) # 19 x 19

annot_labels = np.ma.masked_inside(corrs2.to_numpy(),-0.1,0.1)
replacers = {'Mass Sini':r'$M$sin($i$)', 'Log G':r'log$_{{10}}$g',
             'Teff':r'$T_{{\mathrm{{eff}}}}$'}
axlabs = corrs2.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'$T_{{\mathrm{{eff}}}}$')

plt.figure(dpi=300)
sb.set(font_scale=0.6)
hm = sb.heatmap(corrs2, cmap="coolwarm", annot=annot_labels,fmt='.2f',
           linewidth=0.75, cbar_kws={'shrink':0.8, 'location':'top'},
           xticklabels=axlabs, yticklabels=axlabs, square=False)
sb.set_style('ticks')
plt.show()
sb.set(font_scale=1)
```

str.replace() also clashes with LaTeX: ‘{}’.format() under the hood
→ dict() of replacements raises TypeError

Axis Ticks

Matplotlib's default is outward-pointing ticks on the left & bottom. Ticks on all sides are preferred for readability, but not required.

- 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)
  ```

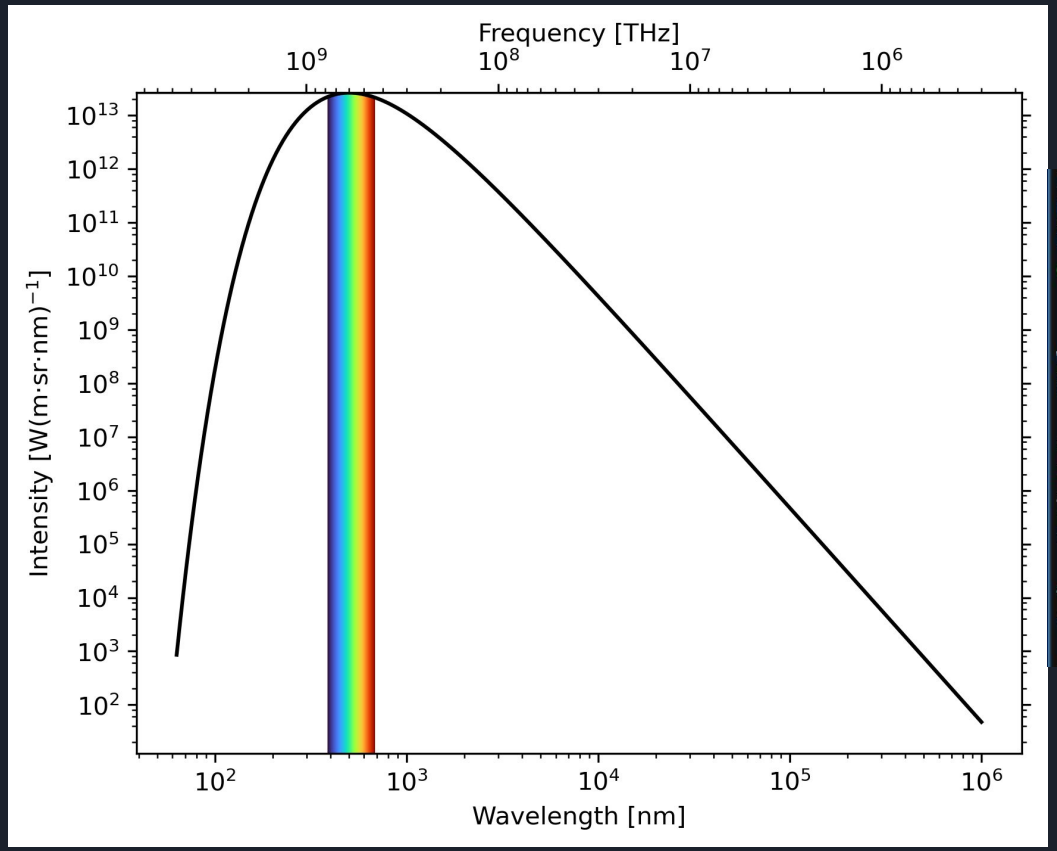
Remember: Be consistent!

Tick Labels & Locators

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

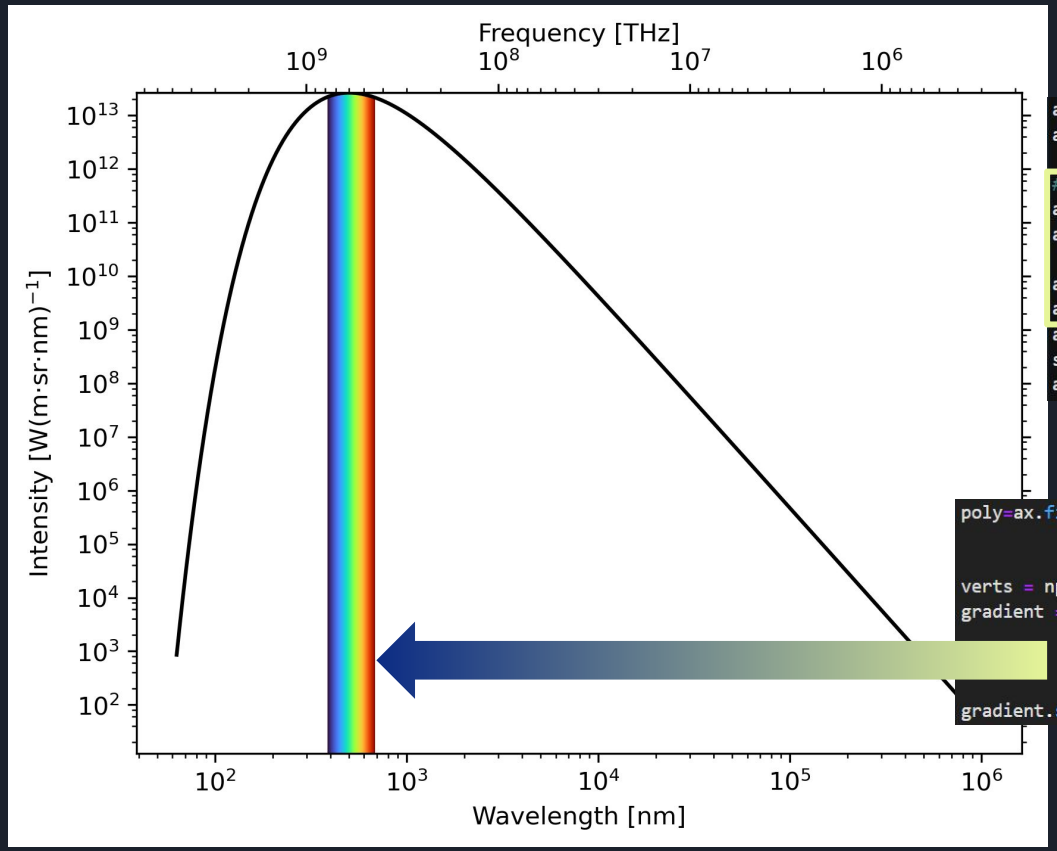
- Units with special formats or symbols (dates, fractions, money, etc.)
- Categorical variables (e.g. countries, species, relative size labels, etc.)
- Axis tick labels centered between major ticks
- Secondary axes that are transformations of the primary axes
- Custom or power-law axis scales
- Log-, 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.




```python
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()




```
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=mpl.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 - have to work in log units & use tick label formatters to override the labels

Grids

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

- Newer releases include matplotlib.projections.geo → supports Aitoff, Hammer, Lambert, & Mollweide axes projections, & has a customizable GeoAxes base class. Docs haven't caught up.
- Matplotlib also accepts AstroPy WCS instances as axes transformations.
- Syntax sample: ax.grid(True, ls=':', color='silver', lw=2) turns grid on with defaults altered

Grids & data format

Images & 2D histograms: use contrasting but neutral color, dotted lines to deemphasize grid.

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
- For 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 which kwarg of grid()

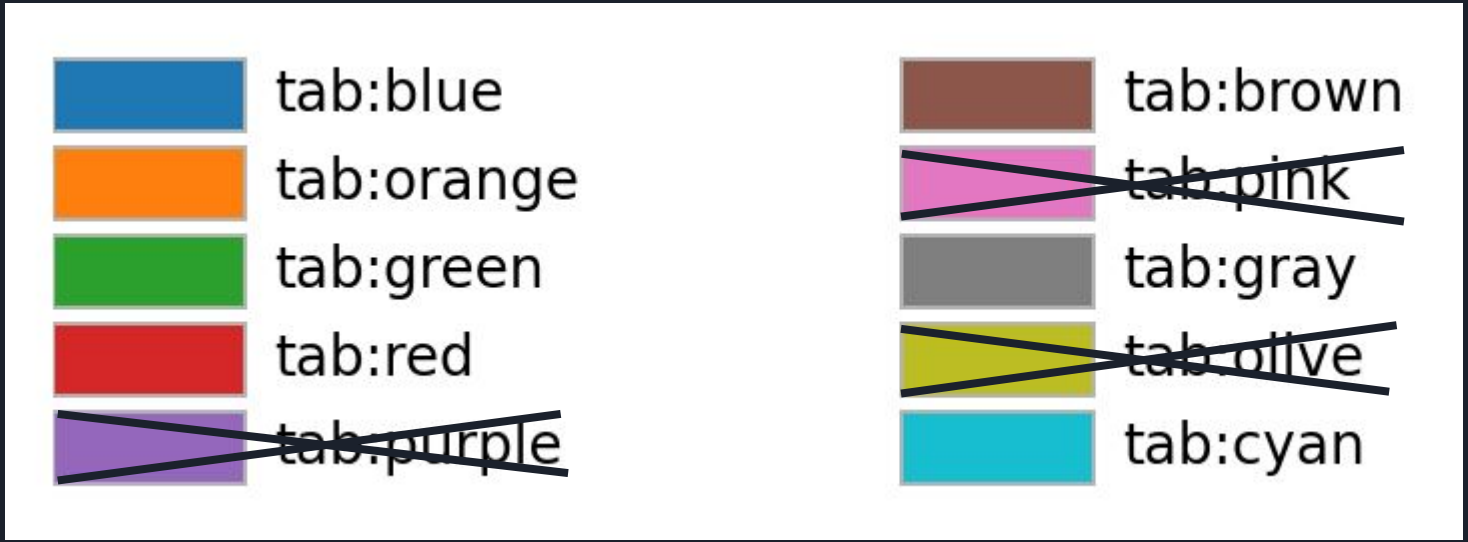
Choosing colors

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 → I like 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 a poor choice for >4 colors

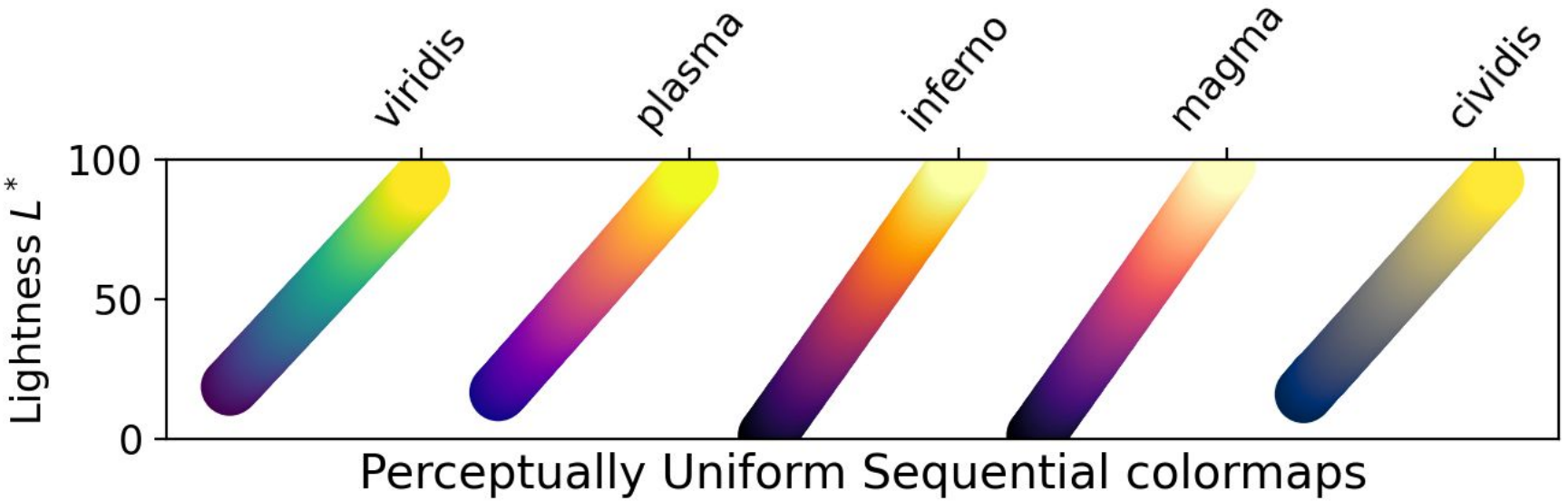
But should vary marker/line styles for >4 data sets anyway

Choosing Colormaps

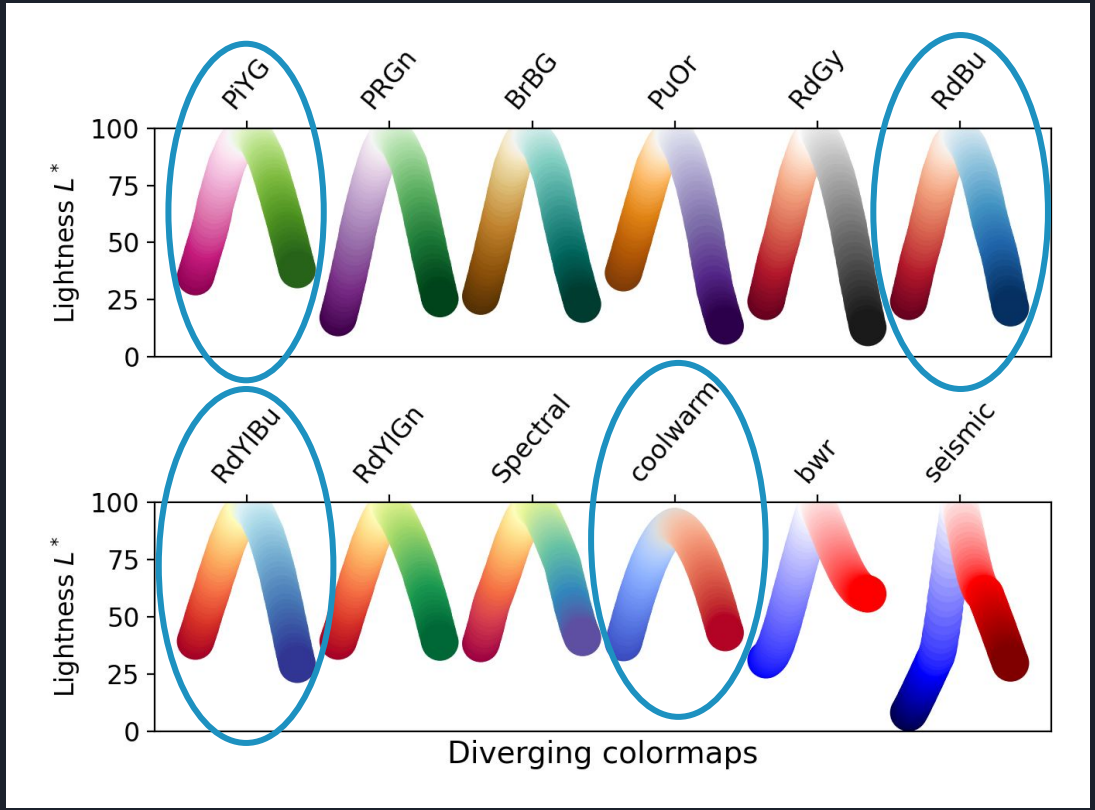
Matplotlib docs mostly cover this. In short, consider:

- **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?

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



For sequential colormaps, these, ‘binary’, ‘gray’, & ‘bone’ are good. The rest…not so much. Avoid ‘rainbow’ & ‘jet’.



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

Putting it all together: rcParams & style files

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

- figure size, backends, aspect ratio, padding, resolution, & subplot spacing
- fonts, font sizes, weights & letter spacing (e.g. monospace)
- marker sizes, fill styles, default shape, color- & line-cycles
- axes tick lengths, widths, colors, visibility, & orientations

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

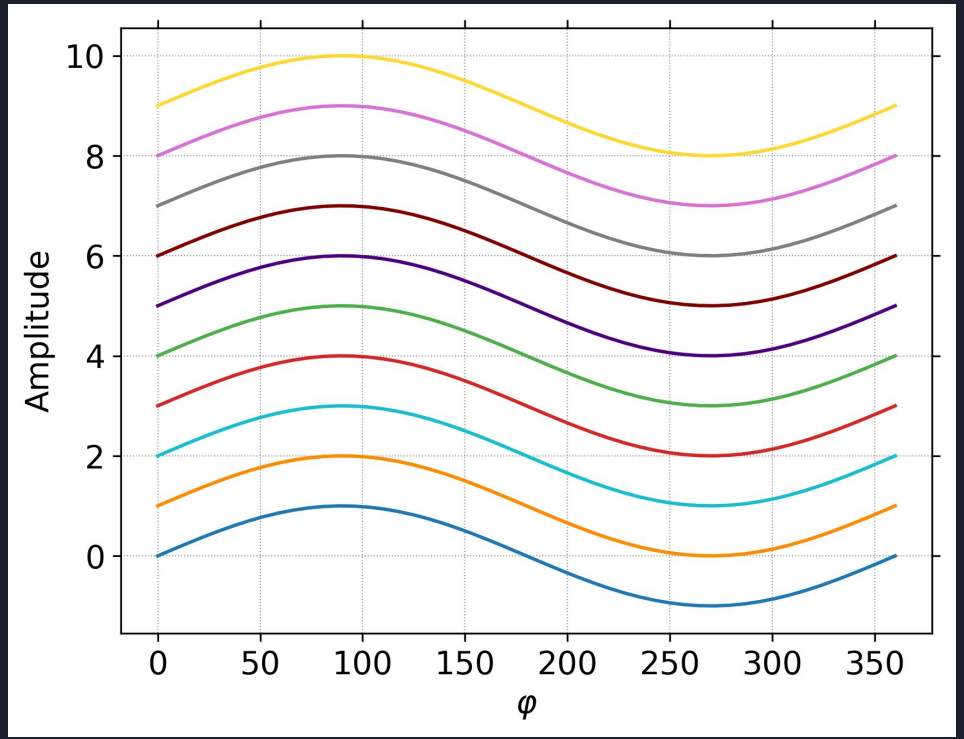
rcParams & style files

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, each of which addresses different set of rcParams.
- Need only include rcParams to be changed from defaults
- Built-in style sheets can be called by name without full path

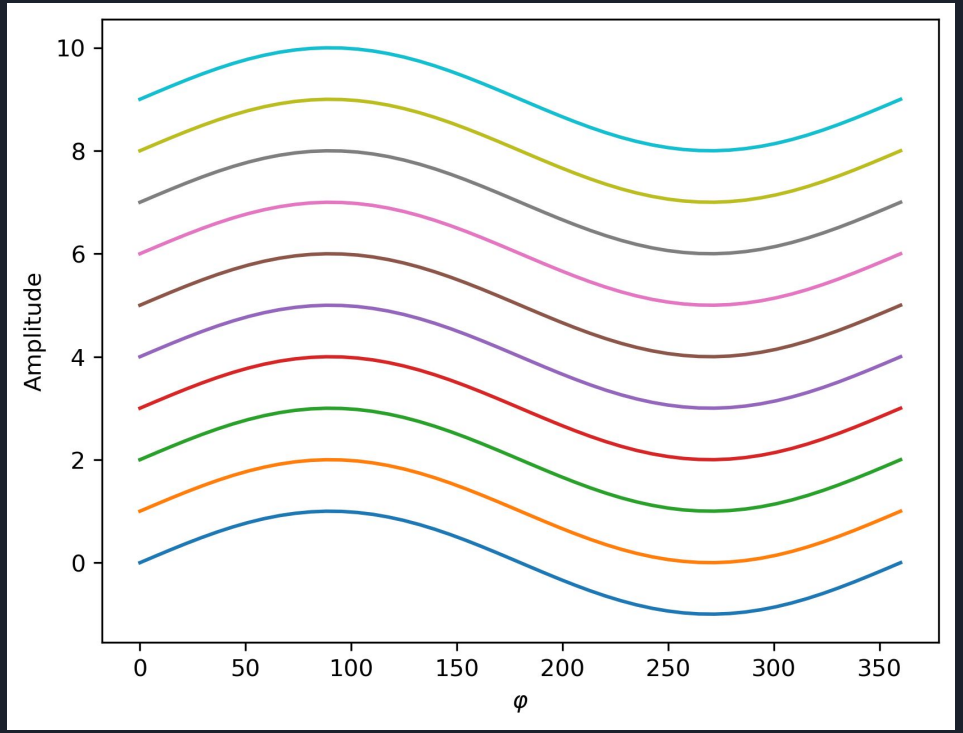
Set for whole session: plt.style.use('./your_style.mplstyle')

Set 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




```python
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)