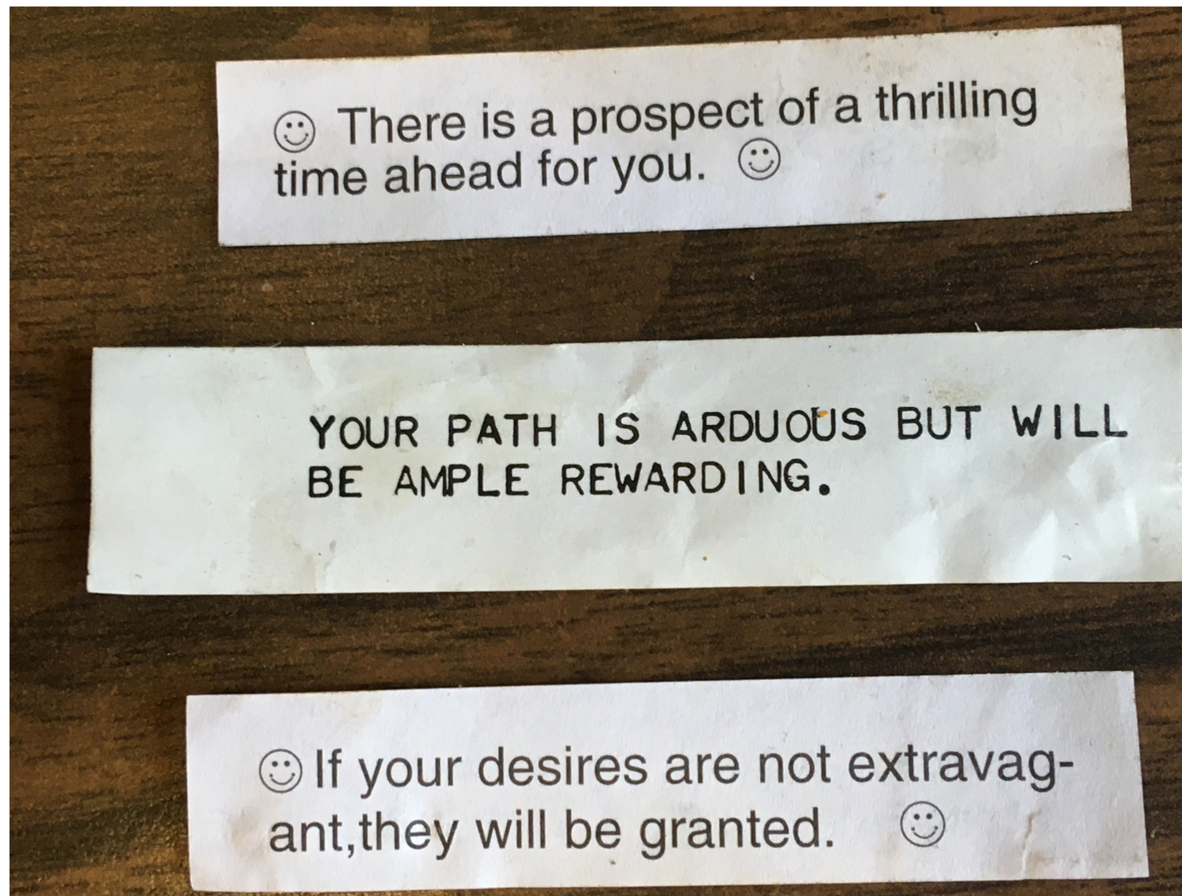
# Scientific animation with Python



# The best

The best questions are ones that you ask yourself.
The best images, still or animated, are ones you make for yourself (or with your partner).
Borrowing animations from Wikipedia only takes you so far!

The goals today are
* Show what is possible and why you may want it.
* Get you started.

# What's the big deal?

Technically there's no big deal. A video is a bunch of still frames presented in succession at over 10/second (looks smoother if it's over 25/second).
But psychologically it's a big deal: We process stories as narrative, with plots.

We'll see how you can just get Python to spit out a lot of individual frames, then show them to you at a chosen rate.

But you'll also want to share your joy – in a presentation; in the supplement to your big article; on social media; etc. For this you need to stitch them together into a file in one of the usual formats, typically mp4 or m4a.

- You could make a video screenshot of Python playing your animation, e.g. with screenshot.app. Usually poor resolution and wrong speed.
- Or you can emit individual frames and postprocess with ffmpeg, QuickTime, VLC, ImageJ, or another such free helper app.
- Today explore a third method: more convenient, once you get over the learning curve.

It's true that "play" can involve frustration.

But your presentations may never be the same.

* If your Python-of-choice is Google Colab, then you're set -- it knows ffmpeg.

* I prefer to run Python on my own laptop. It is easy and free to install the "Anaconda distribution" from
  https://www.anaconda.com/download
  (Anaconda also has a cloud resource, but I'm not familiar with it.)

Anaconda users need an extra step to get ffmpeg. I recommend doing this the Anaconda way, because my students who used other ways (pip, homebrew...) have not been able to connect it directly to Python.

**The hard, supposedly easy, way:**

Launch Anaconda Navigator. (See screenshot below.)
On the left click "Environments"
In the middle click "base (root)"
On the dropdown select "Installed"
In the search box enter "ffmpeg"
If ffmpeg appears in the results, you're already good to go. If not:

On the dropdown select "Not installed"
In the search box enter "ffmpeg"
If ffmpeg appears in the results, click its tickbox to select it.
Now at bottom right you should get a button called "Apply"; click it. This is the moment shown in the screenshot.
After a while a window pops up saying "these packages will be modified." Click the OK button.
Confirm by changing the dropdown menu to "Installed." You should now see ffmpeg.

**The easy, supposedly hard way:**

On macOS, Launch Terminal.app; on Windows launch Anaconda Prompt.
At the command prompt type

```
conda install ffmpeg
```

After a while you'll be asked to confirm; type

```
y<return>
```

After it finishes, confirm by typing

```
which ffmpeg
```

Now you should see something like

```
XXX/anaconda3/bin/ffmpeg
```

which indicates that ffmpeg is in the place where Python will look for it.

Anaconda Navigator

ANACONDA.NAVIGATOR

Connect

Home

Environments

Learning

Community

NEW

Anaconda Toolbox

Supercharged local notebooks.

Click the Toolbox tile to install.

Read the Docs

Documentation

Anaconda Blog

Search Environments

base (root)

anaconda3

Create Clone Import Backup Remove

Not installed | Channels | Update index... | ffmpeg

| Name | T | Description | Version |
|---|---|---|---|
| ffmpeg | | Cross-platform solution to record, convert and stream audio and video. | 4.2.2 |

1 package available matching "ffmpeg"  1 package selected

Apply Clear

heart

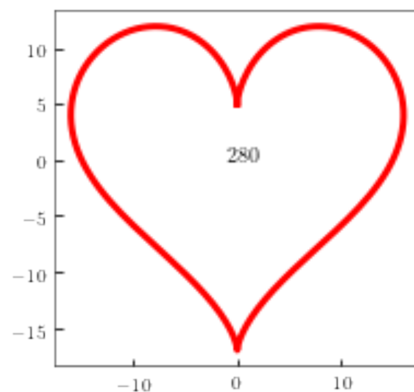
Run Code

In [3]:
```
import numpy as np; import matplotlib.pyplot as plt

# Generate plotting values
t = np.linspace(0, 2*np.pi, 200)
x = 16 * np.sin(t)**3
y = 13 * np.cos(t) - 5 * np.cos(2*t) - 2 * np.cos(3*t) - np.cos(4*t)

# Make the plot
plt.figure(figsize=(3,3))
plt.plot(x, y, 'r', linewidth=3)
plt.text(-1,0,'280')
```

Out[3]:
```
Text(-1, 0, '280')
```




I'd like a more impactful valentine, one that grows and shrinks over time. Right away, I face a problem: My assistant, trying to please me, will rescale the axes in every frame so that the heart fills the frame! Instead of a fluctuating heart, I'll get fluctuating axes labels! There are various workarounds, but the general-purpose insight is that I want to make the axes *once*, then serially replace the contents always leaving the axes unchanged.

# Pythonic matters

`plt.figure()` creates a "figure object," i.e an "object" in the class `figure`, and makes it the "current figure."

Objects can contain other objects. For graphing, we want our figure object to "own" an "axes object."
`plt.axes()` creates such an object in the current figure and makes it the "current axes."
An axes object can in turn contain, e.g. the lines that we usually think of as the axes, but also tick marks, labels, as well as data represented as symbols, curves, bars, etc.

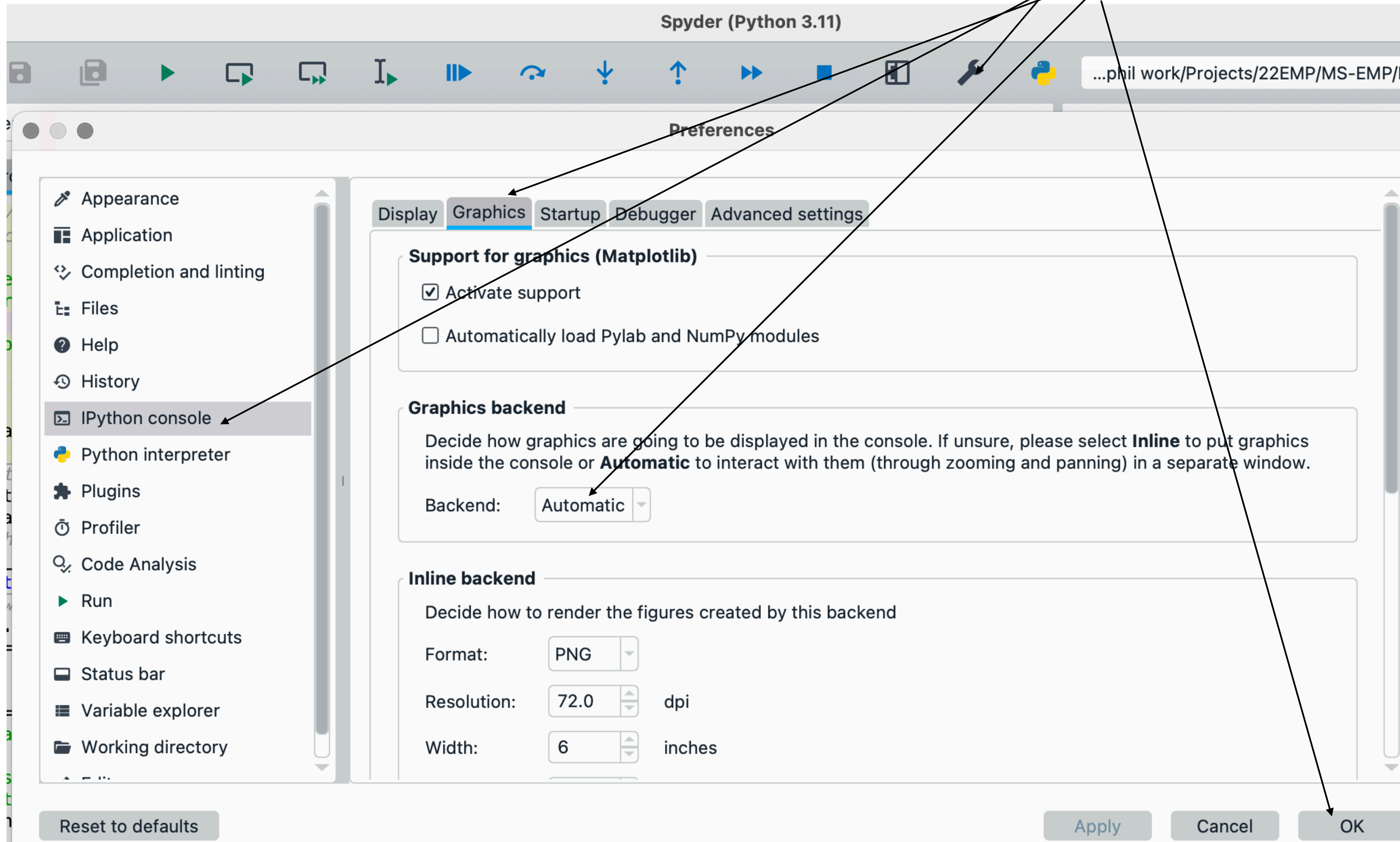
`plt.plot` conveniently combines several operations:

- create a figure object if none exists and make it "current figure" (otherwise use the existing current figure).
- add an axes object to the current figure object if none exists and make it "current axes."
- add symbols and/or curves to that axes object to represent data.
- revise the limits and labels as needed to accommodate that plot and any others already present.

But those operations can be ***unbundled*** for greater control.
In particular, we may wish to attach names to certain subobjects, so that we can go back and modify them by calling their methods.
Thus, if `my_ax` is the name of an axes object then its method `my_ax.plot()` will draw a plot in that object, regardless of whether it is "current." It also returns a tuple containing the object(s) it created (lines, symbols, etc.); if we wish we can assign a name to it. Later, we can then use that handle to change subobjects of the plot without completely redrawing it.
Next slide uses a more subtle version of that idea.

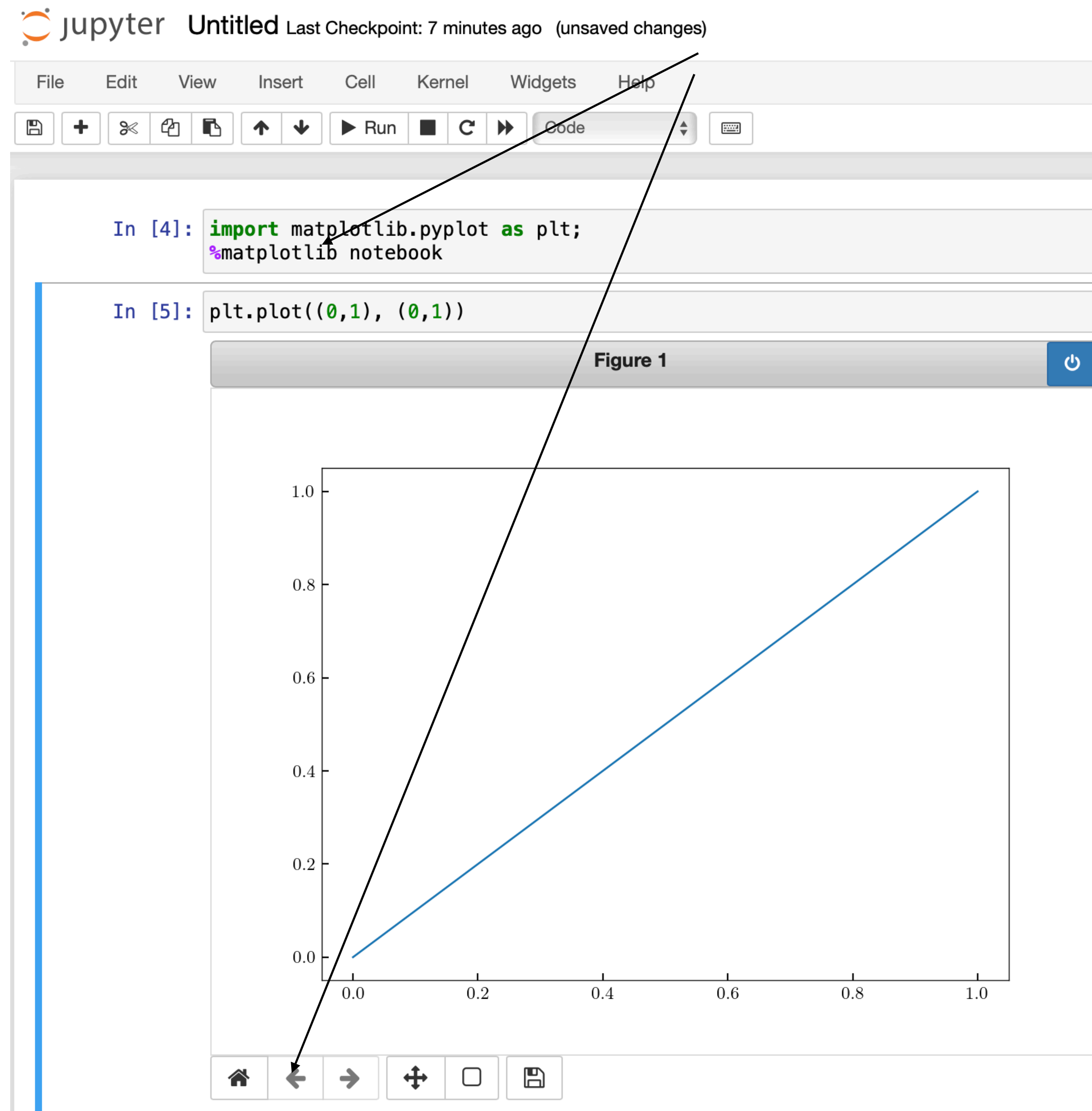
# Making animation on your screen

First set your environment to give "live plots." In Spyder, you only need to do thls once:



# Making animation on your screen

First set your environment to give "live plots." In Jupyter, you must do this every session:



## heartThrob.py:

In Python, a function has access to variables defined in the surrounding code. Sticklers may prefer to transmit them via the "fargs" keyword in FuncAnimation.

A "return" statement here is optional.

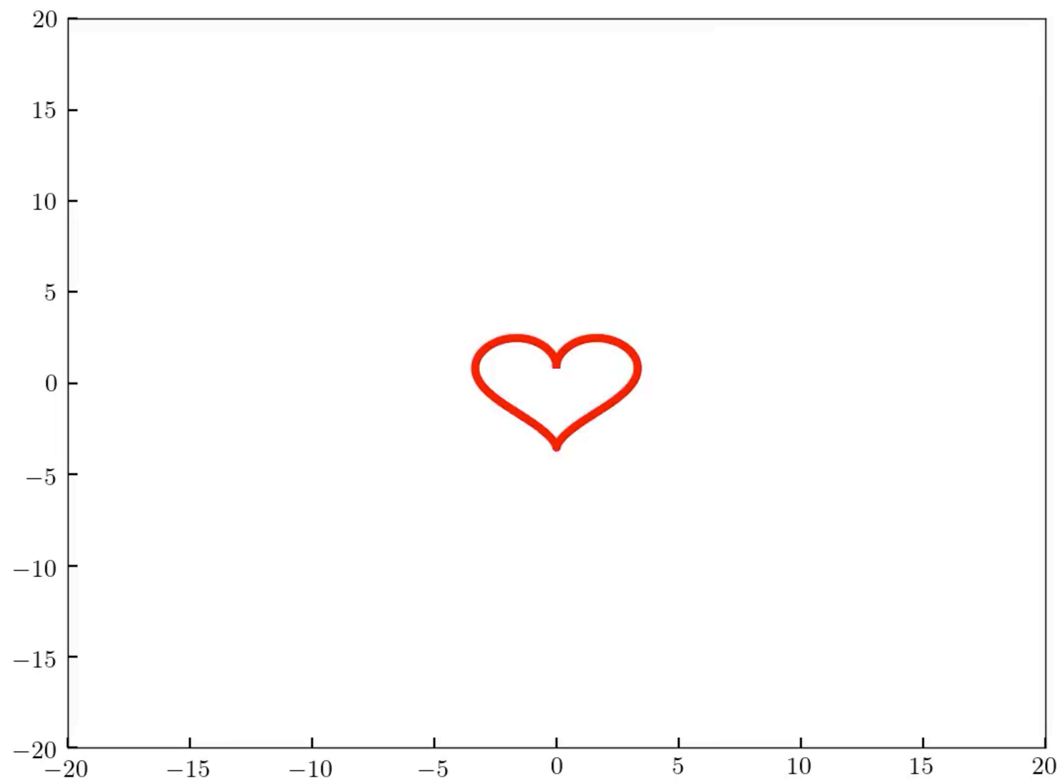
Give a name to the first (and only) line object in the plot.

```
from matplotlib import animation
#%% set up: get_step draws a frame and is called by FuncAnimation below:
def get_step(n):
    scale = np.abs(np.sin(2*np.pi*(n/30))) # this changes for each frame
  # heart and my_line are defined outside the function but available inside it
    my_line.set_data(scale*heart[0], scale*heart[1])

#%% now begin the main code: set generic graph values:
t = np.linspace(0, 2*np.pi, 200)
heart = [16 * np.sin(t)**3, 13 * np.cos(t) - 5 * np.cos(2*t) - 2 * np.cos(3*t)␣
 ↪- np.cos(4*t)]

my_fig = plt.figure()
my_ax = plt.axes(xlim=(-20,20), ylim=(-20,20)) # axes will be exactly same in␣
 ↪every frame
"""create an empty curve, which will be replaced for every frame, and assign it
a name so that we can manipulate it. Note that the plot method of our axis ax
returns a tuple with one element for each line drawn (here there's just one).
We must unpack that tuple to get access to the line object:  """
(my_line, ) = my_ax.plot([], [], lw=3, color='red')

"""Now make the animation:
Tell FuncAnimation which figure window, what frame-drawing function to use, how
many frames:  """
my_movie = animation.FuncAnimation(my_fig, get_step, frames=60)
```



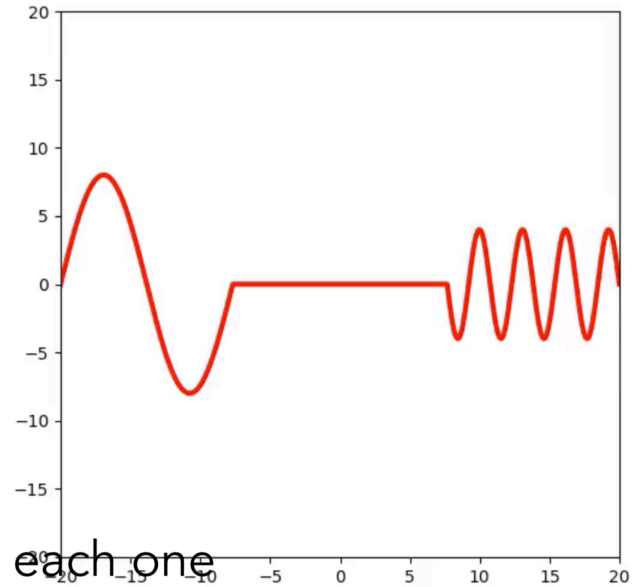

You can now just do a video screen shot of the animation while Python is displaying it.

# Your Turn

*Challenge:* Draw a fixed Lissajous figure, then an animated dot that traces it. Then do something cool on you own initiative.

*Challenge:* Draw a fixed epicycloid or hypcycloid figure, then an animated dot that traces it. Then do something cool on you own initiative.

# Traveling waves (sound in air or light in vacuum):




*Challenge:* make a delta function out of cosine waves. Let each one evolve via the Schrodinger equation, and thus see how that delta wavepacket spreads.

Actually, forget about the SE: All you need to know is that each component wave has frequency related to its wavenumber via $\omega = k^2$. And your starting superposition is a bunch of cosines all with equal weight (the Fourier transform of a delta function).

# Time evolution of a distribution

Molecular diffusion involves the spread of a distribution. It's instructive to look at the randomness in a single instance. But instead of a single histogram at final time, why not make a video of the time development of the histogram?

```
myfig = plt.figure() # set up one invariant axes for all frames
movie_ax = plt.axes(xlim=(-1,Nbins+1), ylim=(0,Nwalk/2)) # stays constant over all frames
my_bars = movie_ax.bar(range(Nbins), binpops[:,0])    # first frame
```

Each bar is a separate object; my_bars is an array of them all.

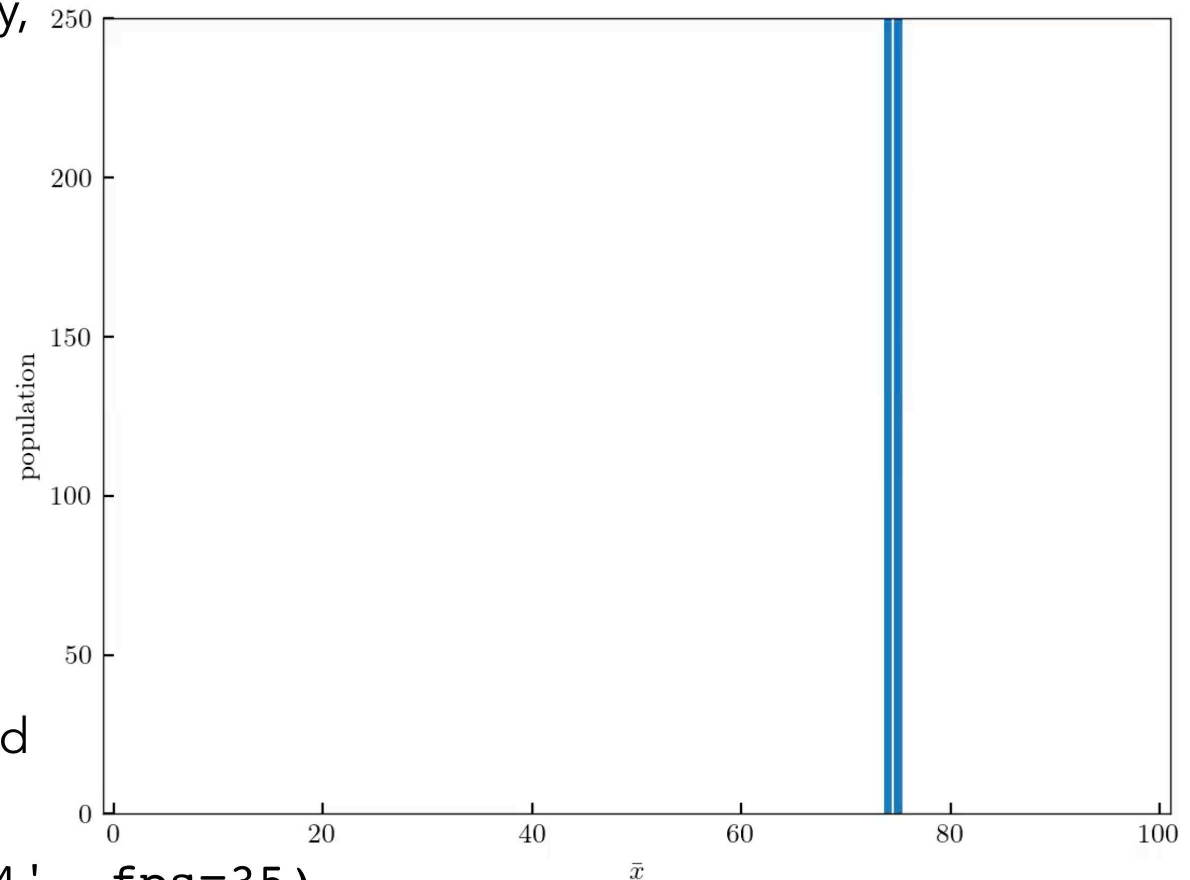
<< blah blah, create `binpops[which bin, which time]` >>

```
def get_step(n):
    for i in range(Nbins):
        my_bars[i].set_height(binpops[i,n])
```

To animate bar plot, in each frame reset the array of bar heights.

# Random walkers in potential trap

The distribution spreads at first, then stops spreading. It also migrates, slowly, eventually becoming centered on the bottom of the potential energy well (x=50).



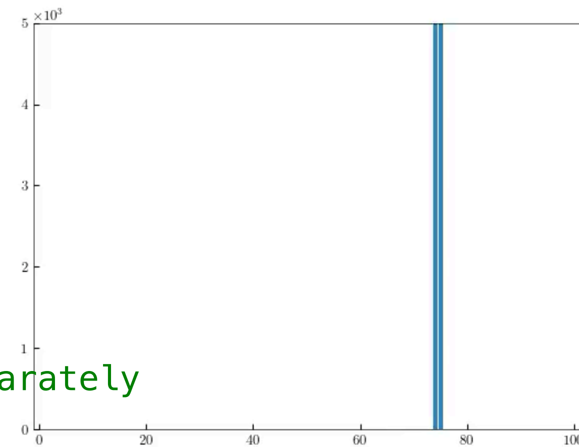

A movie object returned by FuncAnimation contains a "save" method.
This one line renders the animation and writes it to a file for use elsewhere:

```
my_movie.save('harmonicRW.mp4', fps=35)
```

# Brownian in a trap: Details

```python
myfig = plt.figure() # set up one invariant axes for all frames
movie_ax = plt.axes(xlim=(-1,Nbins+1), ylim=(0,Nwalk/2)) # must stay constant over many frames
binpops = np.zeros((Nbins,Nstep)) #history of histogram
half = Nwalk//2
binpops[3*Nbins//4,0] = half
binpops[-1+3*Nbins//4,0] = Nwalk - half
my_bars = movie_ax.bar(range(Nbins), binpops[:,0])    # first frame
#%%
for time in range(1,Nstep):
    temp = np.zeros(Nbins)
    temp[1] = binpops[0,time-1] #handle left edge separately: all bounce
    temp[-2] = binpops[-1,time-1] #handle right edge separately
    for xbar in range(1, Nbins-1): # exclude ends which were handled separately
        Pplus = (1 - (xbar - Nbins//2)/400 )/2
        m = brn(binpops[xbar,time-1], Pplus) # partition walkers
        temp[xbar+1] += m
        temp[xbar-1] += binpops[xbar,time-1] - m
    binpops[:,time] = temp
    if temp.sum() != Nwalk: print("oops", time, binpops.sum()) #should never happen but check
#%%

def get_step(n):
    for i in range(Nbins):
        my_bars[i].set_height(binpops[i,n])
```




How to animate bar plot

# Your Turn

*Challenge*: simulate Ehrenfest's Fleas, and display the results as an animated bar chart. Then do something cool on you own initiative.

# Second visualization: The Swarm

This time, compute the actual trajectories of just 20 walkers. Release the walkers at a variety of initial positions, say, evenly spaced at $\bar{x} = 2, 7, 12, \ldots 97$. Where do they end up?

```
myfig = plt.figure(figsize=(6,1)) # set up one invariant axes for all frames
movie_ax = plt.axes(xlim=(-1,Nbins+1)) # must stay constant over many frames
my_gnats = movie_ax.scatter(trajects[:,0], np.zeros(Nwalk))
```

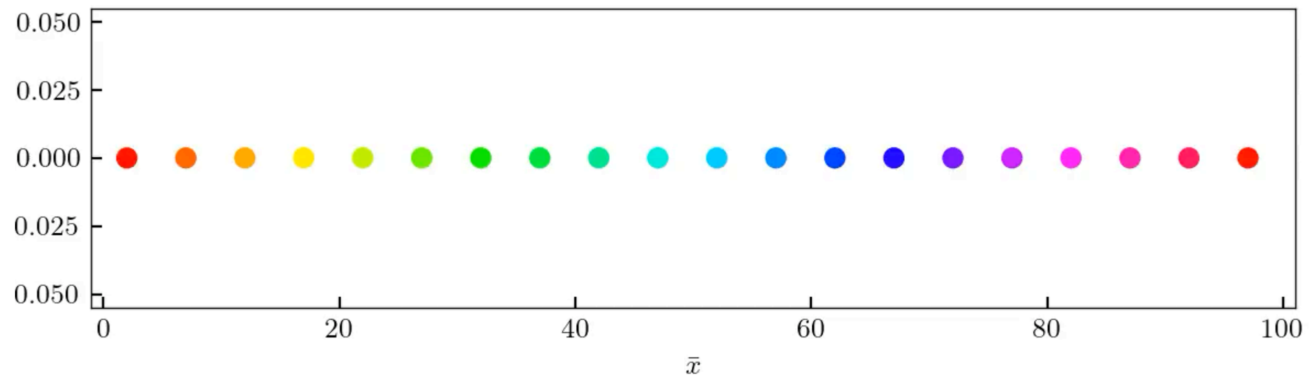
<< blah blah, create `trajects[which walker, which time]` >>

```
def get_step(n):
    my_gnats.set_offsets(np.vstack((trajects[:,n],np.zeros(Nwalk))).T)
```

To animate scatterplot, in each frame reset the xy values by supplying an array with 20 rows and two columns.

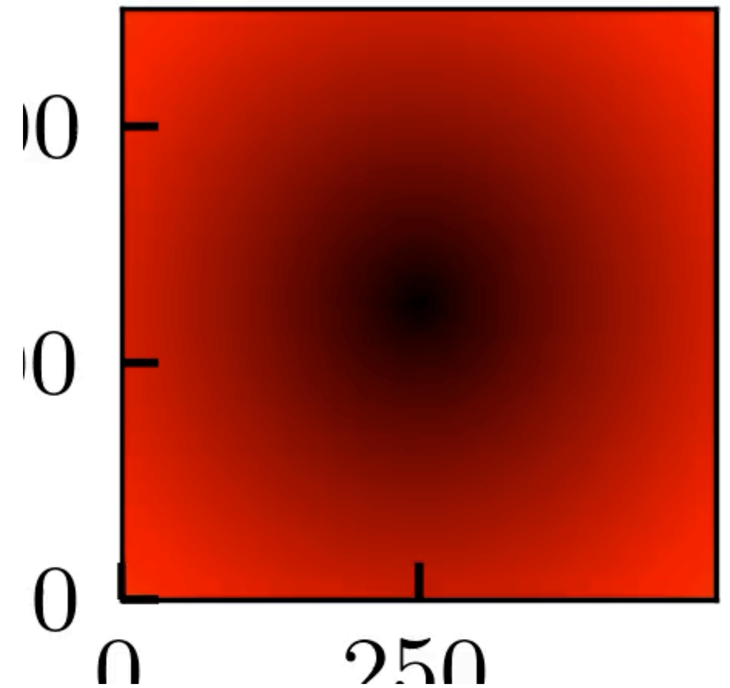
Finally, let's distinguish each walker by giving each its own color. There are various ways to do this. [Hint: Check the documentation for `scatter` for its keyword argument `color`.]

The walkers never stop getting transiently pushed out to large excursions.

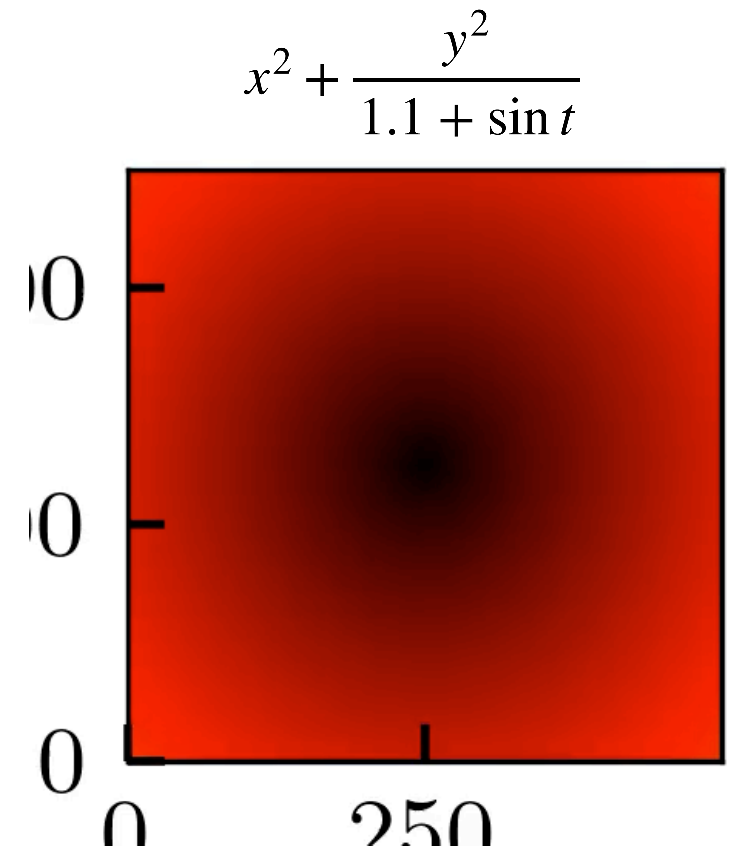


# Raster=bitmap=heatmap Animation

$$x^2 + \frac{y^2}{1.1 + \sin t}$$



# Your turn




*Challenge:* Instead, show 2D diffusion from a point (or something more interesting) in such a representation:
$c(t, \vec{x}) = t^{-1} \exp(-\|\vec{x}\|^2/t)$

*Challenge:* Instead, show the real part of a p orbital in the *xy* plane:
$\mathrm{Re}\,\Psi(r, \varphi, t) = r\mathrm{e}^{-r} \cos(\varphi - t)$

*Challenge:* The same, but this time as a surface plot.

# Exporting animation by writing many still images

Here is a method that makes no use of matplotlib.animation:

```
nmesh = 500 # number of mesh points
tmin = -np.pi # start
tmax = np.pi # end
dt = 0.15 #

def x(i): # converts index to physical distance
    return 2.*(i/nmesh) - 1.0
def y(j): return 2.*(j/nmesh) - 1.0

values = np.zeros((nmesh,nmesh)) # allocate
nframe = 0
for t in np.arange(tmin,tmax,dt):
    nframe += 1
    for i in range(nmesh):
        for j in range(nmesh):
            values[i,j] = np.sqrt(x(i)**2 + y(j)**2/(np.sin(t)+1.1))
    plt.imshow(values.T, cmap='hot', interpolation='nearest', origin='lower')
    plt.text(20, 50, 't='+format(t,'.2f'))
    plt.savefig('rasterMovie'+format(nframe,'05d')+'.png'); plt.close('all')
```

Then use an external helper app to postprocess the resulting image files. For example, Anaconda users can install FFmpeg via the Anaconda Navigator app, or by

```
$ conda install ffmpeg
```

(Windows: Can issue this command in Anaconda Prompt app.) (Mac: Use the Terminal app.)

Then use it:

```
$ ffmpeg -i rasterMovie%05d.png -pix_fmt yuv420p rasterMovie.mp4
```

In case of error, may need (see `https://stackoverflow.com/questions/20847674/ffmpeg-libx264-height-not-divisible-by-2` )

```
$ ffmpeg -i rasterMovie%05d.png -pix_fmt yuv420p -vf "pad=ceil(iw/2)*2:ceil(ih/2)*2" rasterMovie.mp4
```

# Exporting animation by linking FFmpeg to Python.

```
from matplotlib.animation import FuncAnimation

nmesh = 500 # number of mesh points
tmin = -np.pi # start
tmax = np.pi # end
dt = 0.15 #

all_times = np.arange(tmin, tmax, dt)
total_number_of_frames = len(all_times)
def x(i): # converts index to physical distance
    return 2.*(i/nmesh) - 1.
def y(j): return 2.*(j/nmesh) - 1.

values = np.zeros((total_number_of_frames, nmesh, nmesh)) # allocate
nframe = -1
for t in all_times:
    nframe += 1
    for i in range(nmesh):
        for j in range(nmesh):
            values[nframe,i,j] = np.sqrt(x(i)**2 + y(j)**2/(np.sin(t)+1.1))
theTop = values.max(); theBot = values.min()

def animate(frame):
    """
    Animation function.    """
    global values, image # Not strictly neccessary
    image.set_array(values[frame].T)
    return image # return whatever you changed
animation = FuncAnimation(fig, animate, np.arange(total_number_of_frames),
    interval=1000 / 25)
# set the DPI to the actual number of pixels you're plotting to avoid interpol
animation.save("rasterMovie2.mp4", dpi=nmesh)
```

http://joshborrow.com/blog/posts/making_research_movies_in_python/

# Animation, *plus ultra*

```
fig, ax = plt.subplots(1, figsize=(3.6, 2.9))
image = ax.imshow(<<first frame>>.T)         # to be changed each frame
mobilepoint, = ax.plot([],[],'g*', ms=3) # to be changed each frame
mytext = ax.text(6,4,'variable label')   # to be changed each frame
plt.xlabel(r'$x$\ [a.u.]')                 # fixed stuff
ax.text(6, 2.6, 'fixed label')             # fixed stuff

def animate(k):    # make video frame k by changing what needs changing
    image.set_array(<<frame k>>.T)
    mobilepoint.set_data([position(k)], [0.])
    mytext.set_text(str(k))
    return image, mobilepoint, mytext    # return changed objects
```

make x,y what you expect

dynamic text

http://joshborrow.com/blog/posts/making_research_movies_in_python/

# Boring kinetics

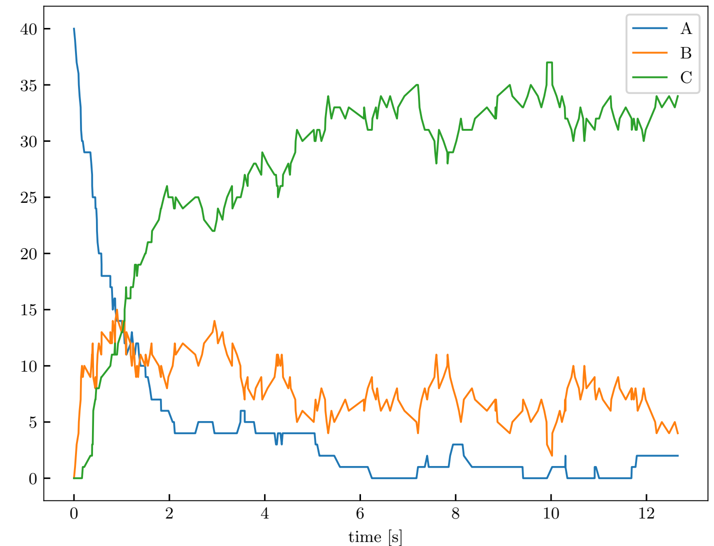
```
"""Description: demoThreestates.py  Gillespie simulation of A<-->B<-->C model"""
import numpy as np; import matplotlib.pyplot as plt; plt.close('all')
from numpy.random import random as rng

"""Three states defining the cols of stoichiometry matrix:
     0 = A; 1 = B; 2 = C
Four reactions, all first-order, defining the four rows of stoichometry matrix:
     0 = A-->B; 1 = B-->A; 2 = B-->C; 3 = C-->B   """
stoich = np.array([[-1,1,0], [1,-1,0], [0,-1,1], [0,1,-1]]) # each row sums to 0
# rate constants:
ks = np.array([[1,0,0], [0,0.2,0], [0,1,0], [0,0,0.2]]) # only one entry in each row nonzero

Mtot = 40 # total number of molecs is constant
Ntrans = 250 # number of steps to simulate
pops = np.zeros((Ntrans+1, 3)) # allocate for populations in states A, B, C
pops[0, 0] = Mtot # initialize
ts = np.zeros(Ntrans+1) # allocate

rxnchooser = rng(Ntrans)
timechooser = rng(Ntrans)

for j in range(Ntrans):
    propens = np.sum(pops[j,:]*ks, axis=1) # propensities for each rxn
    norm = propens.sum() # prob/time for anything to occur
    breakpoints = np.cumsum(propens/norm)
    which_event = np.searchsorted(breakpoints, rxnchooser[j])
    pops[j+1] = pops[j] + stoich[which_event,:]
    ts[j+1] = ts[j] - np.log(timechooser[j])/norm
plt.figure(figsize=(3,3))
plt.plot(ts, pops)
plt.legend(('A','B','C'))
plt.xlabel('time [s]')
```



# Kinetics as a thrilling *story*

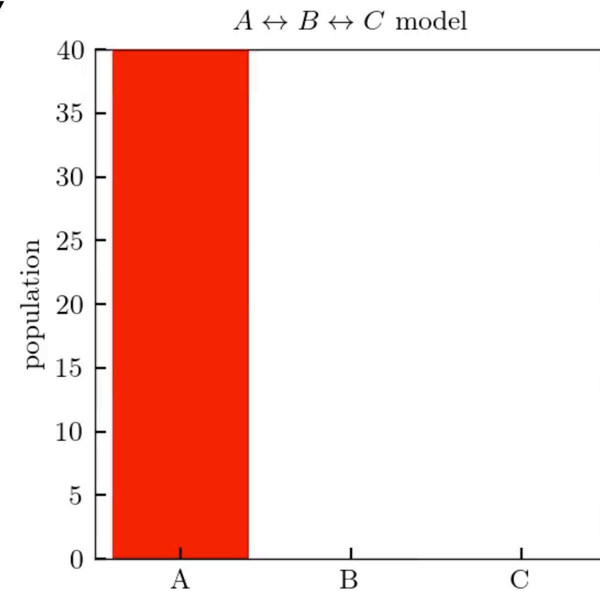
*Challenge:* Upgrade it to make a dancing bar-chart (or roll your own example). To get started, initialize with

```
my_bars = movie_ax.bar(np.arange(0,3), np.zeros(3))
```

Then, in the animation function, modify the bars on each frame using

```
    for i in range(3):
        my_bars[i].set_height(h[i,n])
```

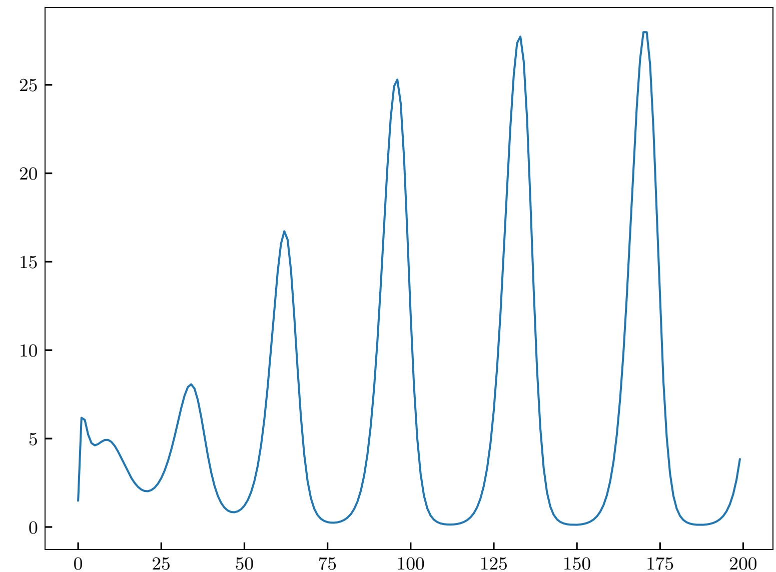
where h is the result from your simulation.



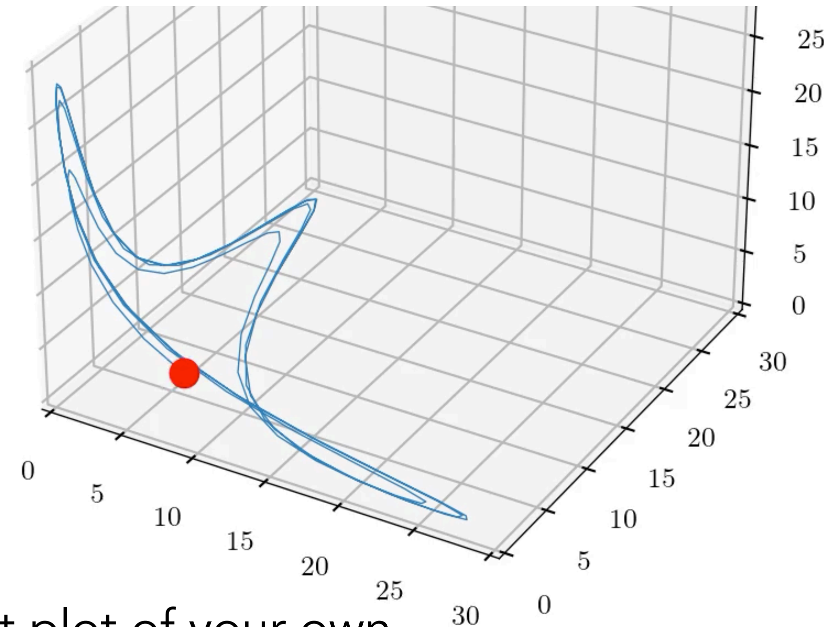

Populations of the three species ultimately equilibrate to the relative values predicted by the Boltzmann distribution, but they never stop fluctuating, and the fluctuations are big if the total numbers are small.

# Boring ODEs

```
from mpl_toolkits.mplot3d import Axes3D
param = [50, 0, 0.2, 2]
# Initial conditions.
y0 = [1.5, 0.5, 1, 1.5, 2, 2]
# Set number of points and frames to use, frame rate.
num = 200
max_frames = 80
rate = 20
# Times at which solution to ODE will be evaluated.
times = np.arange(0, num)
# Function to use with odeint: dy/dt = F(y,t)
def repressilatorVF(y, t):
    # input: y = array of 6 dynamical variables
    # returns: vector field VF of derivatives
    VF = np.zeros(6)
    VF[0] = -y[0] + param[0]/(1.+y[5]**param[3])+ param[1];
    VF[1] = -y[1] + param[0]/(1.+y[3]**param[3])+ param[1];
    VF[2] = -y[2] + param[0]/(1.+y[4]**param[3])+ param[1];
    VF[3] = -param[2]*(y[3]-y[0]);
    VF[4] = -param[2]*(y[4]-y[1]);
    VF[5] = -param[2]*(y[5]-y[2])
    return VF
#%% Solve the ODE.
y = odeint(repressilatorVF, y0, times)
```



# Thrilling ODEs




*Challenge:* Make some sort of 3D animated line or point plot of your own (maybe an explicit function, not the solution to an ODE). This time, the key is that you must create the axes with

```
ax3d  = plt.figure().add_subplot(projection = '3d')
```

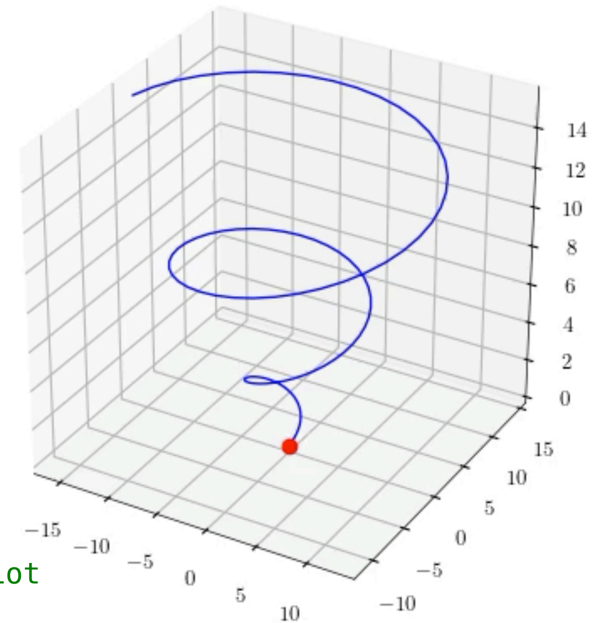
... then initialize the moving point:

```
my_point, = ax3d.plot([], [], [], 'ro', ms=9)
```

... then in the rendering function:

```
my_point.set_data_3d((y[now, 0],), (y[now, 1],), (y[now, 2],))
```

# Useful shortcut




```
t = np.linspace(0, 5*np.pi, 101)         # define parameter for parametric plot

ax.plot3D(t * np.cos(t), t * np.sin(t), t)                   # generate 3D plot

# onward to animation
"""Easy matplotlib animation.
https://github.com/jwkvam/celluloid/blob/master/celluloid.py :"""
from celluloid import Camera
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')# create 3D plotting object attached to figure
ax.set_aspect('equal')                              # distortion? no thanks
camera = Camera(fig)

for thist in t:
    ax.plot3D(t * np.cos(t), t * np.sin(t), t, 'b')                   # same in every fram
    ax.plot3D([thist * np.cos(thist)], [thist * np.sin(thist)], [thist], 'or')
    camera.snap()
animation = camera.animate(interval=100, blit=True)
```

# The best

The best questions are ones that you ask yourself.
The best images, still or animated, are ones you make for yourself (or with your partner).
Go ahead.

A STUDENT'S GUIDE TO

# PYTHON

FOR PHYSICAL MODELING

SECOND EDITION

JESSE M. KINDER

PHILIP NELSON

```
X = np.arange(x_min, x_max+ds, ds)
Y = np.arange(y_min, y_max+ds, ds)
data = np.zeros( (X.size, Y.size), dtype=
for i in range(X.size):
    for j in range(Y.size):
        x0, y0 = X[i], Y[j]
        x,y = x0, y0
        count = 0
        while count < max_iterations:
            x,y = (x0 + x*x - y*y, y0 + 2*x*y)
```

<-- Princeton Univ Press, August 2021

Pine, D J. *Introduction to Python for science and engineering*. CRC Press 2019.

J W-B Lin, H Aizenman. E M Cartas Espinel, K Gunnerson, and J Liu, *Introduction to Python programming for scientists and engineers*. Cambridge Univ. Press 2022.

C Hill, *Learning scientific programming with Python*. Cambridge Univ. Press 2020.