Making Animations with Matplotlib

How to Train Your Robot

Chapter 4

Brandon Rohrer

How to Train Your Robot

About This Project

How to Train Your Robot is a side project I've been working on for 20 years. It's a consistent source of satisfaction. A big part of the joy is sharing what I learn as I go. And who knows? Maybe someone will find it useful.

We're gathering steam now.

Brandon
Boston, USA
November 30, 2022

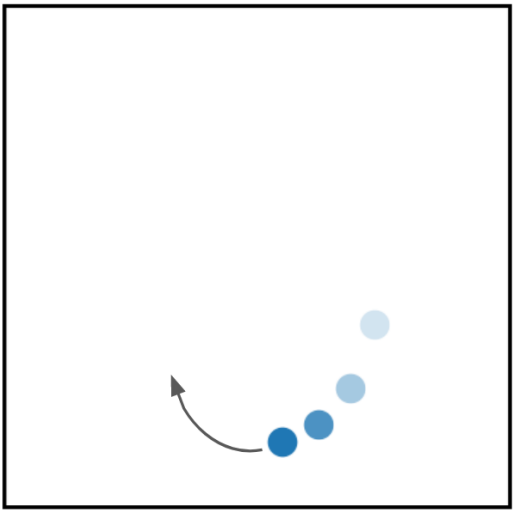
Making Animations with Matplotlib

Chapter 4

In which we re-create the moving picture.

First, an example

This oscillating ball demonstrates the most important tricks we're going to talk about in this chapter. Take a minute and watch the video at tyr.fyi/4lissajous.



Here's the code behind it. (Don't worry about understanding it yet. We'll come back to it, one bite at a time.) This and all the rest of the code for this chapter is in a git repository at tyr.fyi/4files. Download them and run them for yourself!

```
import numpy as np
import matplotlib.pyplot as plt
from pacemaker import Pacemaker

pacemaker = Pacemaker(24)
fig = plt.figure(figsize=(4, 4))
ax = fig.add_axes((0, 0, 1, 1))
lines = ax.plot(0, 0, marker="o", markersize=16)
ball = lines[0]
ax.set_xlim(-1.1, 1.1)
ax.set_ylim(-1.1, 1.1)
plt.ion()
plt.show()

for i in range(10000):
    x = np.sin(i / 10)
    y = np.sin(i / 9.01)
    ball.set_xdata(x)
    ball.set_ydata(y)
    fig.canvas.flush_events()
    pacemaker.beat()
```

lissajous_ball.py

At first glance, the video is nothing special–a blue ball moving smoothly through shifting ellipses. But the clockwork behind it has all of the cogs that we'll need to make far more sophisticated animations.

The ability to draw, move, redraw on a regular cadence will let us represent robot arms flailing in all their configurations. It will let us draw a robotic pup wandering around its world.

It will let us turn the abstract machinations of learning algorithms into movies. With this groundwork in place we'll be limited only by our patience and imagination.

Caveat

I have to start with a big disclaimer. This chapter is not about how to do Animation with a capital A. It's very specifically how to make lines and shapes bob around on a screen in a Python-friendly way that will help us do robotics. There are plenty more sophisticated tools for doing all of this. If you were going to create a Studio Ghibli-style animated movie, you would use a different tool. If you were creating a user interface for a mobile app you would use a different tool. If you wanted to create interactive animations on the website, you'd use a different tool. If you wanted to do large scale, physically realistic object deformation, collision, or fluid dynamics you would use a different tool.

The approach that we're going to work with here is only special because it is the simplest way I could find. This is nothing more than a janky homespun animation tool. The reason to build our own here, rather than going with something out of the box, is that it gives us more transparency and control. We can see exactly what it's doing at any moment. When it doesn't work as expected, we can dig into it and find out why. We can be certain that it's not going to trip up any of the rest of the robotics computation we have running because we're going to design it from the ground up. Also, we avoid building in any dependencies on large animation platforms.

This is a good trade-off for us, particularly if we're only planning to take advantage of a tiny sliver of what bigger platforms can do. And my hope is that by basing it on some widely available Python tools, Numpy and Matplotlib, it will run on almost any computer. That's not always the case for animation software. So we're not doing this out of pure hubris, or the desire to reinvent the wheel. It just so happens that the exact thing we need isn't out there yet, and so it's worth our while to build it from scratch.

Why Animation for Robotics?

Aside from being fun to look at, animations are an invaluable tool for working with robots. For simulated robots it helps to see what's going on and how it's playing out in real time. I've spent way too much time looking at joint angles, printed in ASCII, scrolling rapidly down the console. As soon as I embraced the need to make rudimentary animations, the answers to my questions practically popped out of the screen. We're working with things that are changing over time and moving around in our world (or a virtual world that resembles it). Putting them in an animation lets us tap into our highly specialized visual hardware, neural circuitry that is designed to make use of exactly this type of information.

Even when working with physical robots, being able to animate other things that we can't see, like contact forces and derived features, greatly enriches our mental model of how everything is working. It can also be instructive to have a physical robot and simulated robot side-by-side, one mirroring the other. Any difference between the two becomes immediately apparent. It can bring to light subtle

bugs, like encoder failures, wheel slippage, calibration errors, and sign errors very deep in our calculations.

Running animations will take full advantage of our building blocks so far: the pacemaker from Chapter 2 and interprocess communication through Queues and the multiprocessing package from Chapter 3. Animations are nothing more than a sequence of images updated at a fixed rhythm with subtle differences between them.

Frame Rate

The rhythm matters a lot. If it is uneven the animation looks clunky and sloppy, but as long as the pace is consistent the refresh rate doesn't need to be extremely high. 100 frames per second (FPS) are past the limit of human perception. Anything faster than that is lost on us entirely. Even 60 FPS, a common option on televisions, it's nearly continuous to our eyes. 30 FPS (updates every 33 ms) is closer to the classical television frame rate. It looks smooth to our eye, but somehow less lifelike than 60 FPS. It has a distinctive look and feel. Cinematic frame rates of 24 FPS are even more distinctive. Having grown up with the big screen as the ultimate in movie presentation, I still harbor very positive feelings about that 24 FPS aesthetic. In the absence of reason to do otherwise, it's the format I gravitate toward. High frame rate movies of 48 FPS are also available in the theater, but these feel off to me, even though they are closer to my real world experience.

At 24 frames per second, our brains still stitch the frames into smooth motion. If you know what you're looking for, you can catch fleeting artifacts of individual frames, but it's

not immediately apparent. However once we get much slower than that, it becomes possible to distinguish individual frames. In situations where data size is minimized, like security cam footage, frame rates in the range of 10 FPS to 16 FPS are common. At these rates, it's still possible to see imagined motion, but it's stuttered. Anything 6 Hz or below gives a very strong impression of successive still images. Still useful, but a different experience than observing an object in motion.

We won't be doing anything here that is too complex or computationally intensive, so we should be able to run the frame rate as high as we'd like. By default, I'll set it at 24 frames per second, but I encourage you to play with it and see what you prefer.

There are a couple of things that can block us as we try to make our animation smoother. It can take a while to create the next image. This step is called rendering, actually placing all of the pixels where they need to be. When making complex images like for a Pixar film, rendering a single frame can take thousands of processor hours. Luckily we will be keeping things simple, so that shouldn't be a problem for us.

The other step that can be time consuming is simulation. This comes before a rendering. It's the step where we figure out, for instance, where all of the pieces of the robot arm should be in space. We'll dive into this in the next chapter, so we won't have to worry about this for now. Instead of doing any simulation in this chapter, we'll skip this step and tell the animation exactly where every shape and line needs to be.

It will make sense for us to use our pacemaker to track missed frames. Too many missed frames will clue us in to the fact that our simulation and rendering processes can't keep up with our frame refresh rate. Because this will be running in a separate process from everything else it shouldn't munge things up, but it will be worth throwing a warning to the developer that the frame rate is too high. If this happens, the solution is either to speed things up by streamlining the simulation and rendering computations, or slow things down by decreasing the frame rate.

We are also going to take full advantage of interprocess communication. Animation can be flaky. Anytime we are interacting with physical hardware, even if it is a display, we become subject to physical limitations. Often we'll have to compete with other processes for the resources we need. If another process from the operating system with higher priority swaps in and blocks the display update, then our animation will glitch. This is more likely to happen on some systems than others, and it depends a lot on the specific hardware you have and the specific combination of programs you're running. But it's safe to assume that it's going to happen now and again, so we want to make sure that when it does it won't break everything else we have going on. Keeping the animation in a separate process is a good way to do this.

In this configuration the simulation and rendering can each be isolated in their own sandbox. If they hiccup, hang, or even freeze, the worst that will happen is that the user will have a bad experience. But it won't prevent the rest of the robot simulation or control code from running as it should.

Matplotlib

In Chapter 3 we briefly introduced Numpy and Matplotlib for working with data in Python. Now we are going to get better acquainted with them.

Before Numpy and Matplotlib there was a program called MATLAB (MATrix LABoratory) that specialized in scientific computing. Specifically, it is very good at manipulating arrays. The company that makes it is still around: MathWorks just outside of Boston. MATLAB will always have a special place in my heart. It was the first language that I got comfortable enough with to make algorithmic mischief. It got me through grad school and kicked me off as a researcher. But when I started hacking around at home and writing code that I wanted to share with the rest of the world, the high price tag became a barrier. That's when I discovered Python, Numpy, and Matplotlib. They were entirely free of charge, and as a bonus Matplotlib was written to be very similar to MATLAB's plotting routines. I never looked back.

The mimicked MATLAB functionality in Matplotlib is just a gateway drug. The bones of Matplotlib are a powerful and low level drawing capability. You can literally manipulate individual pixels if you are so inclined. This is what makes it a good fit for our work in drawing and animating robots and their environments. To unlock this power, we need to move past the high level functions that draw lines and scatterplots and delve into the inner workings. This means embracing objects and object oriented programming.

Object-Oriented Programming

If you're familiar with object-oriented programming you can skip this part. If you're not, we'll cover just enough to meet our needs. But know that it is a large topic the subject of many volumes and much discussion.

To illustrate how Object Oriented Programming (OOP) works and why it is useful, imagine we have a circle we'd like to draw. There are a few things that we will need to know about this circle like size, position, color. We can keep track of all these in a collection of variables.

```
radius = 3
center_x_position = 1
center_y_position = 1
edge_linewidth = 2
edge_color = "darkblue"
face_color = "blue"
```

All of these things are necessary for fully specifying what to draw and how to draw it. If we want to write code that animates this circle, we can carry this collection of variables around. Every time we need to find out something about the circle, we can read these variables' values. Every time we want to modify some aspect of the circle, we can change one of their values.

This approach is fine. It's totally legal. But it's a little awkward. All these variables are describing different attributes of the same thing. But instead of keeping track of that one big thing, we have to keep track of six little things. If we pass the circle to a function, we have to pass six arguments instead of one. It's unwieldy.

Now image if instead of 6 attributes, it was 60. What was unwieldy becomes almost unmanageably complex. Now imagine a dozen different circles instead of one. There become so many variables to keep track of that the code becomes unreadable and the probability of introducing a bug approaches one.

To keep things cleaner, we can do some bundling. We can create an object and gather the attributes into it. To kick this off, we create a new object class, which we'll name Circle.

```
class Circle:
```

Now we have a Circle class, but it's still an empty shell. We haven't defined anything about it. To fix that, we add an initialization function. This function is special. It has to be named `__init__()`. That's "init" with two underscores before and after it. Surrounding a name with double underscores is a convention Python uses to set aside particular names for its own purposes. This double-underscore-init is one such special name. If you call it "dunder init", you really sound like you're in the know.

Dunder init is called every time we create an object of type Circle, so it's a great place to set all the attributes we want to keep track of.

```
class Circle:
    def __init__(self):
        self.radius = 3
        self.center_x_position = 1
        self.center_y_position = 1
        self.edge_linewidth = 2
        self.edge_color = "darkblue"
        self.face_color = "blue"
```

In a confusing twist of syntax, `__init__()` takes a `self` argument, which refers to the very object being created, the new Circle object. Every attribute we want to assign to this object, we declare with `self.<variable_name>`.

We can create and initialize an object of type Circle by calling the class itself.

```
circle = Circle()
```

Calling `Circle()` automatically runs `__init__()` and assigns the result to the new object variable, `circle`. Then we can access its attributes with the `circle.<attribute>` syntax.

```
print(circle.center_x_position)
print(circle.edge_color)
```

```
1
darkblue
```

And we can modify them the same way.

```
print(circle.edge_linewidth)
```

```
2
```

```
circle.edge_linewidth = 1
print(circle.edge_linewidth)
```

```
1
```

Collecting and managing attributes are just two of the useful things objects and classes do for us. They also let us define functions that are specific to the type of object we're working with.

Imagine we wanted to implement a "go home" behavior for our ball. Whatever its x- and y-position, it will take a small step toward the (0, 0) home position, moving 10% of the way toward it.

```
def go_home(circle):
    circle.center_x_position = .9 * circle.center_x_position
    circle.center_y_position = .9 * circle.center_y_position
```

We could also write this exact thing more concisely.

```
def go_home(circle):
    circle.center_x_position *= .9
    circle.center_y_position *= .9
```

This is a short function, easy to understand and read. Now consider the case of a more complicated function. Let's say that the size of the step that the circle takes toward the home

position is determined by its radius and color in some convoluted way.

```
def go_home(circle):
    if circle.face_color == "darkblue":
        rate = 1 / (1 + circle.radius ** 2)
    else:
        rate = 1 / (1 + 2 * circle.radius)

    circle.center_x_position *= rate
    circle.center_y_position *= rate
```

It's perfectly fine to represent this as a freestanding function, but take a look at how closely it's associated with the circle object. It accepts one argument of type Circle, it accesses four different attributes of that circle, and it modifies two of those attributes. It is a method for doing something circle related and doesn't have much to do with the rest of the code. We can reflect this close coupling by pulling this function into the object itself. Just like we can bundle attributes, we can also bundle functions that are specific to this type of object.

To do this in our code, we move the function inside the class definition, and instead of an input argument `circle`, we substitute `self` again.

```
def go_home(self):
    if self.face_color == "darkblue":
        rate = 1 / (1 + self.radius ** 2)
    else:
        rate = 1 / (1 + 2 * self.radius)

    self.center_x_position *= rate
    self.center_y_position *= rate
```

Just like the way classes manage attributes, gathering object-specific functions into a class is a matter of convenience. It doesn't change what we can do or how well it works, but it can be really helpful for keeping related code together. Now, if I want to modify the `go_home()` behavior of a circle, I don't have to search through all of my code to find where I implemented it. I know that it's right there with my class definition. If I want to work with circles, I just have to find the Circle class code. I don't have to also seek out all of the functions I created for manipulating it.

By the way, if you want to be proper, there is particular terminology to use when working with classes. It's not that important, but it can be helpful to know.

Normally they're called	But when they're part of a class, they're called
variables	attributes
functions	methods

Object oriented programming isn't the one right way to write code, but it can be helpful in some situations. Whether it makes sense to break your code up into classes depends on practical considerations, like how complex it is and who will be maintaining it. This decision can also be driven by the aesthetic and philosophical preferences of the authors. There are some beautiful ideas behind the object oriented approach. Sometimes authors are so enamored by the ideas behind it that they force their code into classes whether it benefits from it or not. Depending on who you talk to, there

can be strong feelings on all sides of this conversation. Just a heads up.

Objects in OOP don't have to be visible things. They can be as abstract as we want. We could create an object that represents a customer that contains attributes like their contact information and methods for managing communication and transactions with them. The pacemaker we're using is based on the `Pacemaker` class (from tyr.fyi/4files) we used in Chapter 3, which contains attributes like `clock_period` and `start_time` as well as a `beat()` method to help it keep time. We can even have an object represent an entire robot, with attributes representing each of its components that are themselves objects. (Spoiler: This is exactly what we're going to do.)

For better or for worse, Matplotlib is heavily object oriented. Everything important is an object. The Figure object is the basis for everything. You can think of it as an art gallery wall. It's a place where you place art, but you can't draw on it directly. The Axes object is like a canvas. You can make it any size you want and place it anywhere in the Figure. You can even add several of them. Then, when you go to make visual elements, these can take a few different forms, such as a Line or a Patch, which are also objects. All of this means that comfort with the concepts of classes, attributes, and methods will serve you well.

Having a ball

We now know just enough about OOP to dive into Matplotlib. We begin by importing the pyplot package, and by convention, abbreviating it as plt.

```
import matplotlib.pyplot as plt
```

Then we create a new figure. This returns an object of type Figure.

```
fig = plt.figure()
```

We can pass the optional argument figsize to specify (in inches) how large we expect the figure to be. (Yes, it is in inches, not centimeters. I don't make the rules.) For a 4 inch by 4 inch figure:

```
fig = plt.figure(figsize=(4, 4))
```

And we call the `add_axes()` method of the Figure class to create a new set of axes.

```
ax = fig.add_axes((0, 0, 1, 1))
```

The (0, 0, 1, 1) argument indicates that the axes should start at the far bottom left of the figure window and extend the full width and height. We are strong-arming our plot to fill all the available visible space, the entire gallery wall. We're turning the entire figure into a movie screen.

Now for the critical bit.

```
lines = ax.plot(0, 0, marker="o", markersize=16)
ball = lines[0]
```

The call to the `plot()` method of the axes draws a single circular marker of size 16 at the x, y position of (0, 0). It returns a list of all the lines that were drawn. In our case, there is only one "line", and in fact just one point on that line, so we pull off the first and only element off the list and call it `ball`.

To round out the setup of the axes, these two lines of code declare that the left edge of the figure corresponds to x = -1.1, the right edge to x = 1.1, the bottom to y = -1.1 and the top to y = 1.1.

```
ax.set_xlim(-1.1, 1.1)
ax.set_ylim(-1.1, 1.1)
```

The last two steps in setting up our animation are turning on interactive mode with `ion()` and displaying the figure on the desktop with `show()`.

```
plt.ion()
plt.show()
```

This gets us our ball! We've drawn a ball! This is a major milestone and we should all take a moment to revel in it.

I'm happy for you! Also, don't take my ball.

Ballroom dance

So far we have done a bit of drawing, but no animation. Our ball is just sitting there. To animate it we have three more steps: calculate a new position for the ball, move it there, and update the plot. Then we repeat that for as long as we want the ball to stay in motion.

To generate a pattern of movement, we'll fall back to a classic–sinusoidal oscillations with different frequencies in the x- and y-directions, also called Lissajous curves.

```
for i in range(10000):
    x = np.sin(i / 10)
    y = np.sin(i / 9.01)
```

As the step counter, `i`, continues to grow, `x` and `y` wiggle back and forth at different frequencies. This will give the ball a wobble that starts as a diagonal line, fattens to a circle, then

collapses back to a diagonal line leaning the opposite directions, and repeats. You can play with the two constants here, 10 and 9.01, and generate a whole family of behaviors.

The next step is to update the ball position. This is where we get to take advantage of working with objects. Our ball has some internal attributes that represent its x- and y-positions. The code does some clever data handling things, so it won't let us reach in and change the x and y data directly, but it does provide a couple of methods that let us set their values, `set_xdata()` and `set_ydata()`. This pattern of setting and getting attribute data through methods is so common that these special purpose methods have their own names: setters and getters.

```
ball.set_xdata(x)
ball.set_ydata(y)
```

Updating the display takes some effort and time from the processor, and it doesn't happen automatically every time we make a change to our ball's position. We have to specifically ask it to update the screen with the changes we've made. This is also an opportune time to check in with our pacemaker and keep our animation updates on the beat.

```
fig.canvas.flush_events()
pacemaker.beat()
```

And that's the whole enchilada. Now when we run this, we see the gracefully swinging ball at tyr.fyi/4lissajous . It contains all the basic ingredients of any animation we'll ever make: the implementation of the draw-move-update-repeat loop. Here's the full code for the ball moving in Lissajous

curves. Now that we've worked through it, it should feel more familiar.

```
import numpy as np
import matplotlib.pyplot as plt
from pacemaker import Pacemaker

pacemaker = Pacemaker(24)
fig = plt.figure(figsize=(4, 4))
ax = fig.add_axes((0, 0, 1, 1))
lines = ax.plot(0, 0, marker="o", markersize=16)
ball = lines[0]
ax.set_xlim(-1.1, 1.1)
ax.set_ylim(-1.1, 1.1)
plt.ion()
plt.show()

for i in range(10000):
    x = np.sin(i / 10)
    y = np.sin(i / 9.01)
    ball.set_xdata(x)
    ball.set_ydata(y)
    fig.canvas.flush_events()
    pacemaker.beat()
```

lissajous_ball.py

A Note for Notebook users

There is one change you'll need to make to get this to run in a Python notebook. At the very beginning make sure to include this magic.

```
%matplotlib notebook
```

Keep this trick in mind for all the animation scripts from here on out.

Juggling

Now that we can make one ball flit around the screen, let's try the same trick with several of them at once. It's a straightforward repetition of what we've already done. Instead of creating just one ball, we copy, paste, and tweak that line to create four of them.

```
ball = ax.plot(0, 0, marker="o", markersize=16)[0]
shadow_1 = ax.plot(0, 0, marker="o", markersize=16)[0]
shadow_2 = ax.plot(0, 0, marker="o", markersize=16)[0]
shadow_3 = ax.plot(0, 0, marker="o", markersize=16)[0]
```

And instead of moving one ball, we can move all four of them at once. Here we make the adjustment of staggering the balls a bit, so that each of the shadows is trailing the main ball by a few steps.

```
ball.set_xdata(np.sin(i / 16))
ball.set_ydata(np.sin(i / 14))

shadow_1.set_xdata(np.sin((i - 1) / 16))
shadow_1.set_ydata(np.sin((i - 1) / 14))

shadow_2.set_xdata(np.sin((i - 3) / 16))
shadow_2.set_ydata(np.sin((i - 3) / 14))

shadow_3.set_xdata(np.sin((i - 6) / 16))
shadow_3.set_ydata(np.sin((i - 6) / 14))
```

We can create as many objects as we like and move them however we want! It's an exciting time.

Color coordination

If you run the code like this, you'll notice that each ball is a different color. We haven't told Matplotlib what color to make each ball, so it has fallen back to its default palette. If we'd like to exert some control over the colors, we certainly have that option. We can use the named keyword `color` and pass a value for it to `plot()`.

```
ball = ax.plot(
    0, 0,
    color="darkblue",
    marker="o",
    markersize=16,
)[0]
```

There are a few different ways we can tell Matplotlib what color to use. The most intuitive of these is to call out a color by name. Not every color name you can think of is supported, but a surprising number of them are. For instance in the blue family there are `slateblue`, `darkblue`, `lightblue`, `midnightblue`, `cornflowerblue`, and a bunch of others.

If you are so inclined, you can also use a tuple of three values between 0 and 255. These get interpreted as red, green, and blue values in the RGB color space. For example, (0, 0, 0) is black, (255, 255, 255) is white, and (255, 0, 0) is as red as red can get.

An alternative way to specify a color by RGB values is to use its hex code. In hexadecimal, 255 is ff, so white is (ff, ff, ff) in hex RGB space. This gets further shortened to look like #ffffff. This is a concise way to express any color we will ever want to render. The website hexcolorcodes.org has a fun

interface that you can use to experiment with and learn the relationship between a hex code and the color it generates.

For a deeper dive into all of your Matplotlib color options, you can go to tyr.fyi/4color and for a yet deeper dive into how colors are represented with RGB values, check out tyr.fyi/4numbers.

I have mixed feelings about the use of color in animations. On the one hand, I am a bit red-green color blind, so there are differences between blues and purples and greens and browns that I don't pick up on. There are several varieties of color blindness, so any color display will be understood differently by different viewers. The pragmatic part of me thinks it might just be easier to do everything in grayscale.

However, there is no denying that color is an effective way to get viewers to feel something. Just like changing the background music in a movie scene gives it an entirely different feel, changing the color palette in an animation gives it a wholly different vibe. Color combinations are a powerful way to tap into feelings and associations that the viewer isn’t even aware of.

Because I’m a little bit color challenged, I like to go online and look at color palettes that designers have shared. There are a million of them. If you search "color palette" and add a few descriptive words like "outer space" or "cozy fireplace", you'll get a lot of options. I don’t always know what I’m looking for before I find it, but often when I stumble across the right combination of colors it makes me feel a certain way, and I know that’s the color combination I want to use for the project I'm working on. They’re usually accompanied

by their hex color codes, which I copy into my Python script, and I try to include the link back to the original source.

For our ball with its shadows, we'll stick with variations on the theme of blue. Each trailing ball will be a little lighter shade to give us a sense that time has passed and the impression of the ball has faded: #4682b4, #a7c4dd, #d3e2ee, #e9f0f6. You're not expected to have a mental picture of the color just by looking at these, but notice how the number pairs get higher as we go. They are getting lighter, approaching white, #ffffff. Also notice how the first two digits, the red channel, are always the lowest of the three and the last two digits, the blue channel, are always the highest. It's not a pure blue, but it's more blue than anything else. Here's what the four circle callouts look like now.

```
ball = ax.plot(
    0, 0,
    color="#4682b4",
    marker="o",
    markersize=24)[0]
shadow_1 = ax.plot(
    0, 0,
    color="#a7c4dd",
    marker="o",
    markersize=24)[0]
shadow_2 = ax.plot(
    0, 0,
    color="#d3e2ee",
    marker="o",
    markersize=24)[0]
shadow_3 = ax.plot(
    0, 0,
    color="#e9f0f6",
    marker="o",
    markersize=24)[0]
```

Stacking order

If we're not careful, Matplotlib will draw the shadows on top of the original ball rather than underneath it. We want to make sure that the youngest shadows are on top of the older ones, and that the ball itself is on top of everything else. There are two ways to do this. One is by re-ordering our lines of code, so that Matplotlib's default handling happens to get them in the right order. This is fiddley and error-prone, and can spike your stress levels. Instead, we are going to tell Matplotlib exactly what order to stack these in.

The named argument `zorder` gives explicit instructions as to what falls on top of what. It works like a z-coordinate, a position out of the page. Whatever has the highest z-order will be on top, the lowest z-order will be on the bottom. It can be any number you like. In our case, we just have to make sure that the ball has the highest z-order, and that each progressively older shadow has a lower z-order than the one before.

Here's what the final code looks like for specifying the ball, its shadows, their colors, and their order.

```
ball = ax.plot(
    0, 0,
    color="#4682b4",
    marker="o",
    markersize=24,
    zorder=3)[0]
shadow_1 = ax.plot(
    0, 0,
    color="#a7c4dd",
    marker="o",
    markersize=24,
    zorder=2)[0]
shadow_2 = ax.plot(
    0, 0,
    color="#d3e2ee",
    marker="o",
    markersize=24,
    zorder=1)[0]
shadow_3 = ax.plot(
    0, 0,
    color="#e9f0f6",
    marker="o",
    markersize=24,
    zorder=0)[0]
```

Now that we've flexed our muscles and told Matplotlib exactly what we want, our code has gotten a little bit longer. This is an ongoing trade-off in Matplotlib. It offers complete control, and the price we pay is more lines of code. It's a reasonable trade-off in my opinion. But it means that we may need to employ some strategies to manage longer visualization code.

The first line of defense is breaking the code up into functions. Our code does a couple of important things here. There's one chunk where we create the ball and all of its shadows. There is another chunk where we update the position of the ball and each of its shadows. Then there is the overarching code that initializes the figure and iterates through the frames. These three chunks are a reasonable way to split up the code. We can break the code out into three functions, `create_circles()`, `move_circles()`, and `main()`. If we look ahead a little bit, this structure has the advantage that it will let us swap out new code for the `create_circles()` and `move_circles()` portions without having to change the overall structure. We've taken a step toward making our code modular. Pieces of it are reusable for future projects.

When it comes to breaking code up into functions, there are a lot of different preferences and philosophies. As with everything, there is no one right answer. The winning solution is whatever works best for who's going to be writing, reading, and maintaining the code. If that's just going to be you, then your opinion is the only one that matters.

I tried to write my code for the future version of me that will have forgotten everything I did and why I did it. For future me, an ideal Python file is readable start-to-finish. It begins with the highest level function, the one that executes the main purpose of the code. In our case that is `main()`.

Then, as I'm reading through `main()`, I come across functions that aren't yet defined, like `create_circles()` and `make_circles()`. The names of these functions give me a

rough sense of what they do, and, as I read further down through the code, I can fill in the remaining gaps in my knowledge by reading the functions themselves. More complex code may have several levels of functions calling other functions. If I'm really on my game, each function will take up no more than one screen on my computer, allowing me to read it as a single page. That lets me read only as deeply as I want and then skim over the lower level details after that.

The one thing I have to remember to make this game work is to call the `main()` function at the very end of the script so that it will run when I call the script.

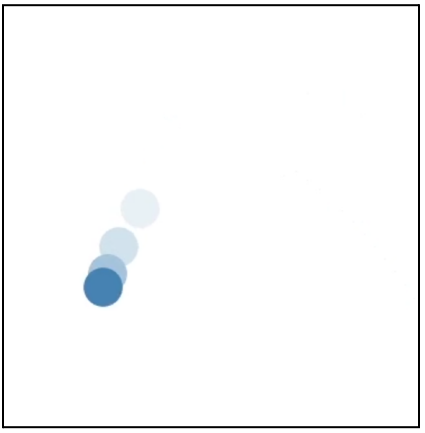
With all of this in place, here's what our finished animation code looks like.

```
import numpy as np
import matplotlib.pyplot as plt
from pacemaker import Pacemaker

def main():
    pacemaker = Pacemaker(24)
    fig = plt.figure(figsize=(5, 5))
    ax = fig.add_axes((0, 0, 1, 1))
    ax.set_xlim(-1.5, 1.5)
    ax.set_ylim(-1.5, 1.5)
    circles = draw_circles(ax)
    plt.ion()
    plt.show()

    for i in range(1000):
        move_circles(circles, i)
        fig.canvas.flush_events()
        pacemaker.beat()

def draw_circles(ax):
    ball = ax.plot(
        0,
        0,
        color="#4682b4",
        marker="o",
        markersize=24,
        zorder=3,
    )[0]
    shadow_1 = ax.plot(
        0,
        0,
        color="#a7c4dd",
        marker="o",
        markersize=24,
        zorder=2,
    )[0]
```

```
    shadow_2 = ax.plot(
        0,
        0,
        color="#d3e2ee",
        marker="o",
        markersize=24,
        zorder=1,
    )[0]
    shadow_3 = ax.plot(
        0,
        0,
        color="#e9f0f6",
        marker="o",
        markersize=24,
        zorder=0,
    )[0]
    return (ball, shadow_1, shadow_2, shadow_3)

def move_circles(circles, i):
    (ball, shadow_1, shadow_2, shadow_3) = circles

    ball.set_xdata(np.sin(i / 16))
    ball.set_ydata(np.sin(i / 14))

    shadow_1.set_xdata(np.sin((i - 1) / 16))
    shadow_1.set_ydata(np.sin((i - 1) / 14))

    shadow_2.set_xdata(np.sin((i - 3) / 16))
    shadow_2.set_ydata(np.sin((i - 3) / 14))

    shadow_3.set_xdata(np.sin((i - 6) / 16))
    shadow_3.set_ydata(np.sin((i - 6) / 14))


main()
```

shadowed_circle.py

Just a reminder, you can find this and all of the other scripts from this chapter at tyr.fyi/4files. If you would like to copy and paste from the GitHub repository, or fork it and do something fancier, please help yourself.



Watch the show at tyr.fyi/4shadows.

When we run this, we see a graceful blue ball, swaying back-and-forth across the screen trailed by three shadows, each lighter than the last. Thanks to our careful color selection, they are nicely coordinated. Thanks to our careful ordering, each shadow falls under the one that came before. These finishing touches give it a clean look and are well worth the extra trouble.

Once you have the code running, play with it. Change the size of the balls, play with the position update to make the movement faster or slower, make wider or sharper wiggles. Change the colors. Add some more shadows. Make a second ball that moves in a different pattern. Tweaking things and see what happens is an excellent way to learn about what's going on

Patches

Circles are great, but it's very likely that we will eventually want to draw something else. There are some shortcuts for jumping to triangles, rectangles, and stars, but we're going to skip right to the power solution: patches. A patch is an arbitrary polygon. It's a connect-the-dots outline of a shape. It can have as many points in its outline as we want. When it comes to drawing shapes, patches are the Key To The City.

To define our patch, we have to create a path. This is a sequence of dots that gets connected to draw the shape. A path to draw a triangle would look like this.

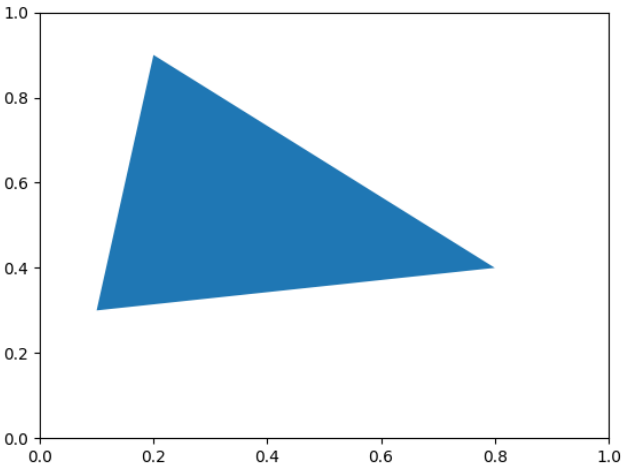
```
path = [
    [.1, .3],
    [.2, .9],
    [.8, .4],
]
```

Each row represents one point, its x-value and its y-value. After they are all stacked together, the path is an array with two columns. To turn this into a patch on the drawing, we have to invoke Matplotlib patch object, and use the path to initialize it.

```
import matplotlib.patches as patches
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.gca()
path = [
    [.1, .3],
    [.2, .9],
    [.8, .4],
]
ax.add_patch(patches.Polygon(path))
fig.savefig("patch.png")
```

triangle_patch.py

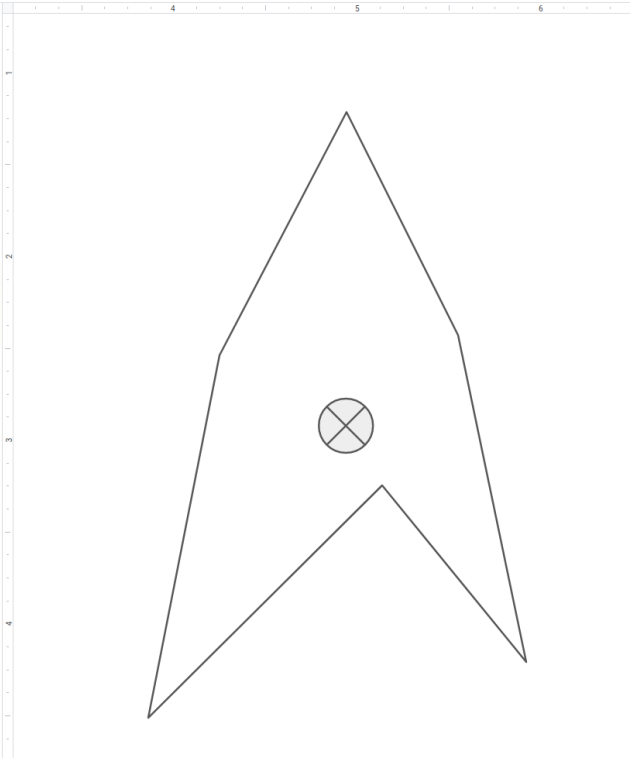
Then we can have it show up in our plot with each vertex being at the specified path coordinate.



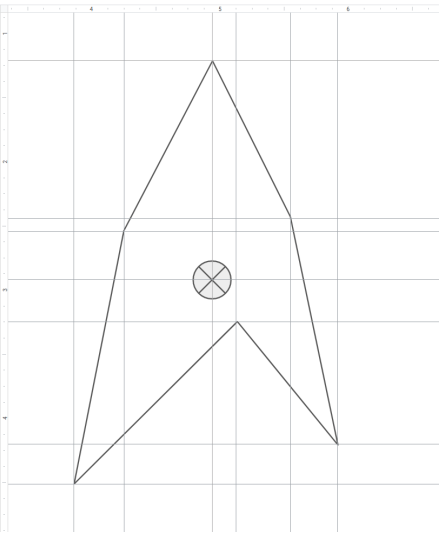
We can have as many points as we want in our path. Going from the shape in your imagination to a numerical path can

be a bumpy road. There are several ways to go about it, but none of them are painless. Here's a method I regularly use for slightly more complex shapes.

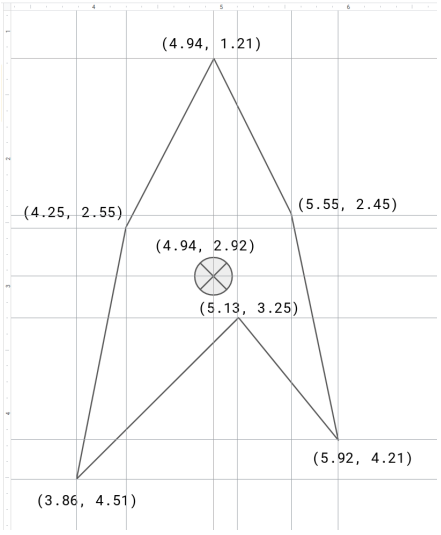
1. Start by hand sketching a shape in Google Slides or a similar app. Here I also included the center point that I want to use as the (0, 0) position in my drawing's coordinate system.



2. Make sure there is a ruler of some sort visible and project each vertex back to its horizontal and vertical positions.



3. Find the x- and y-coordinates of each vertex in the shape.



4. Pick any vertex and go around the shape in order. Type each (x, y) pair into an array.

```
uncentered_path = np.array([
    [4.94, 1.21],
    [5.55, 2.45],
    [5.92, 4.21],
    [5.13, 3.25],
    [3.86, 4.51],
    [4.25, 2.55]
])
```

Subtract the x and y values of the center point from every other point in the path to center it over $(0, 0)$.

```
center = np.array([4.94, 2.92])
path = uncentered_path - center
```

Flip the y-axis to account for the fact that the positive y-direction is down for our ruler.

```
path[:, 1] = -1 * path[:, 1]
```

There are also some keywords we can use to customize the face color and the color and thickness of the outline.

```
ax.add_patch(patches.Polygon(
    path,
    facecolor="none",
    edgecolor="black",
    linewidth=2))
```

Et voila! Our hand sketch is now immortalized in Python code.

Translation

The five-dollar word for moving something around on the page is **translation**. It very specifically means moving a thing without spinning it or bending it or changing it in any other way. When we were working with a ball, whose position was defined by a single point at its center, translation was trivial. We just moved the center to a new location and the rest of the ball followed. Now that we are working with patches we have to be more thoughtful. The patch we just created is defined by a path with six points, each with their own coordinates. Moving it around on the page just became a bit trickier.

When moving a thing, the first step is to figure out how far we want it to go. If we're looking at changing the x-position of a thing, then this change will be Δx. If we know where a thing is, x_{start}, and where we want it to end up, x_{end}, then

$$\Delta x = x_{end} - x_{start}$$

Armed with the change we want to see in the world, the next step is to apply it to each point in the path, one at a time.

$$x_{i,end} = x_{i,start} + \Delta x$$

for all i in the path.

Translation in the y-direction works exactly the same way.

$$y_{i,end} = y_{i,start} + \Delta y$$

If we want to get a little fancier, we can write this out as a set of matrix operations.

$$\begin{bmatrix} x_{i,end} \\ y_{i,end} \end{bmatrix} = \begin{bmatrix} x_{i,start} \\ y_{i,start} \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

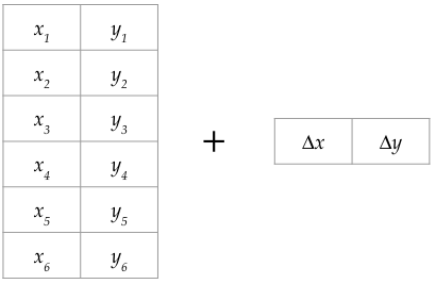
It means exactly the same thing, it's just a way to combine the two equations into one. For now it's decorative, but later in the chapter it will prove useful.

Matrix notation is also useful because it mimics the Python code. Numpy's foundational data structure, the array, is a matrix as far as we are concerned. The implementation of translation looks similar to the matrix equation.

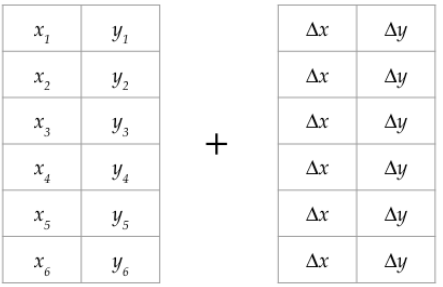
```
dx = 2
dy = -1
translation = np.array([[dx, dy]])
translated_path = path + translation
```

Even though we are adding the same `dx` and `dy` to every point in our six-point path, we only have to create one (`dx`, `dy`) pair. When we go to add our path array with six rows and two columns to our translation array with one row and two columns, Numpy automatically infers that we meant to have six copies of that row, adding the same `dx` and `dy` to each x and y in the path.

When we tell Numpy



it knows we really mean



This assumption is called **broadcasting** and saves us a lot of tedious replication. It is one of the many useful things Numpy does for us. When we work with arrays of different shapes, it makes a lot of labor-saving assumptions about what we intended.

A terminology note: **matrix** is the official mathematical term and **array** is the official computational term. Pedants from both camps will tell you they are different things. And they're right. But we're going to gloss over all that and use the terms interchangeably because their differences don't matter to us right now.

Scaling

Now that we're working with patches, we have a lot more freedom of expression. Not only can we move them around, but we can also make them bigger and smaller.

Getting started is straightforward enough. If we want to double the size of a patch, we can multiply every x and every y by two. This will make it twice as wide and twice as high. If we want the final height and width to be 1/10 of the original, we multiply everything by .1. It's no coincidence that the word scaling is often used as a synonym for multiplication.

The difficulty comes when the object we are scaling is not situated at (0, 0). If it's centered at (1, 1) for instance, then after multiplying all of the x's and y's by two, the new patch would be centered at (2, 2). Multiplication not only causes the patch to grow or shrink, but also causes it to get further away from or closer to the origin.

The solution to this is to

1) first move the object to the origin,
2) then scale it,
3) then move it back to where it started.

Luckily, moving things around is a problem we just finished solving with translation. The only question that remains is where exactly the origin should be. This is the point on the object that will stay put no matter how we blow it up or shrink it, how we zoom in or out. This is the object's **anchor**.

Anchors aweigh

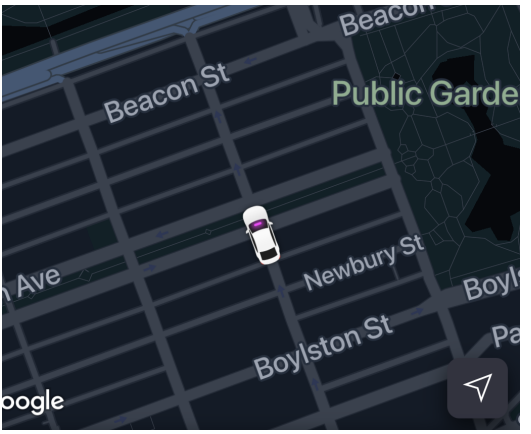
The anchor can be any point we choose. In a lot of cases, it makes sense to put it in the middle of the object. This can be the average of all the path points' x-positions and y-positions. It can also be the geometric centroid of the patch or just a hand-picked spot that sits visually near the center. With a target near the center, the middle of the object will appear to stay put no matter how we scale it.

Intuitively, the middle of an object is what we think of as its location. It's the thing we don't want to change even as it changes sizes. But sometimes that's not the case. Consider a mouse pointer in the shape of an arrow. In this case, if we change the size of the arrow, it is the point of it that we want to make sure stays in one spot. The tip of the arrow is the intuitive anchor. This also goes for pins on a map. It's not the center of the tag we care about so much as the very point that is identifying a location.

When trying to determine the anchor
of an object just ask yourself,
"What part of it would I use to boop an Ewok?"

In the maps of ride scheduling apps, for instance, cars are represented with anchors at the dead center, right on top of the travel mug full of iced coffee.

No matter how you zoom in or out, the center of the car represents its estimated position. As a result, when you're zoomed out far enough, it looks like a car stopped at a stoplight is straddling the entire intersection. Free consulting advice to ride sharing apps: A better anchor might be the front of the car.

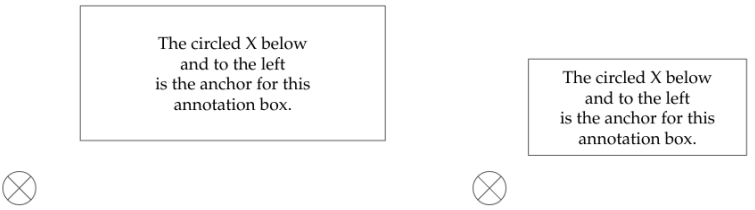


If this poor driver were actually sitting
in the middle of the Eastbound lane
of Boston's Commonwealth Avenue,
they would be having a very bad day.

In the case of text, there is a baseline that we want to stay fixed, no matter how each character is scaled. It will fall at the bottom of most lower case letters, but for those with tails, like p and q, the baseline falls closer to the midpoint of the character. For these, it's important that the anchor not be set at the bottom of the character, but rather at the baseline. That

will help the rounded bottoms of the p and q to stay in line with the rest of the text, even when they are rendered in a much larger font, as p and q.

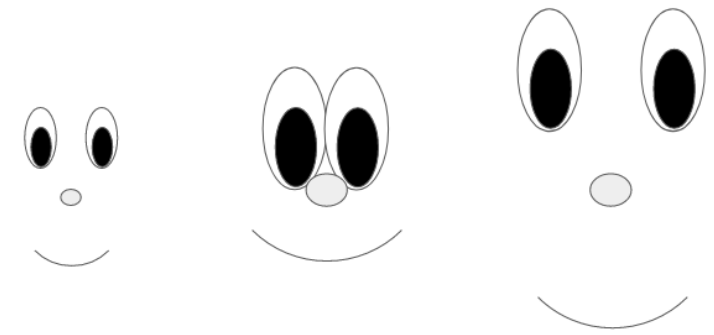
An anchor doesn't even have to fall within an object. For an annotation box, for instance, we may want to make sure that it always stays just to the right of the thing it's annotating. Whether we make it larger or smaller, we want to maintain a small offset from the object it's referencing. In this case, the anchor would fall outside the box entirely.



The anchor falls below and to the left of the annotation box.

Another common case is where we have a collection of patches that need to maintain a fixed relationship with each other, for instance, the eyes, nose, and mouth in a drawing of a face. As we scale the features up and down, we want to make sure that they maintain the same relationship so that the whole face looks like a larger or smaller version of itself, rather than looking like eyes are growing and shrinking in an otherwise unchanging face. The solution to this is to give every patch in the group the same anchor point, perhaps the bottom of the nose. Even though this anchor falls well outside both of the eyes, having them share an anchor means

that, if the eyes double in size, the space between them will double as well. All the relationships will be maintained.



Original face with four features.

Face scaled by 2x. Each feature anchored at its center.

Face scaled by 2x. All features anchored at a common point.

All this is to say that every object needs to have an anchor, and the location of that anchor will depend entirely on what the object is, and how it's meant to be interpreted by the viewer.

Now that we have an anchor point, one advantage it gives is allowing us to describe the position of a complex object with a single (x, y) pair. All we really need to know is where the anchor is. We can figure out where every other vertex is after that if we need to.

For scaling, the anchor tells us exactly what part of our patch needs to be at (0, 0) before we do any multiplication. Our scaling operation breaks down to three steps:

1) translate the anchor point to (0, 0)
2) scale every point in the path
3) translate the anchor point back to where it was.

In matrix notation it looks like this. If the original anchor point is $(x_{\text{anchor}}, y_{\text{anchor}})$ and the scaling factor is a, then

$$\begin{bmatrix} x_{scaled} \\ y_{scaled} \end{bmatrix} = \begin{bmatrix} x - x_{anchor} \\ y - y_{anchor} \end{bmatrix} \times a + \begin{bmatrix} x_{anchor} \\ y_{anchor} \end{bmatrix}$$

In code it looks similar, although it ends up being more convenient to represent each (x, y) as a 1 x 2 (1 row and 2 column) matrix, rather than a 2 x 1 (2 row and 1 column) matrix as we did above. It might seem like a small difference, but small differences have brought down many mighty programs. The 1 x 2 `anchor` array broadcasts naturally with our 6 x 2 `path` array.

```
anchor = np.array([[x_anchor, y_anchor]])
scale = 3
scaled_path = (path - anchor) * scale + anchor
```

With this new trick up our sleeve, we can scale an object however we want in an animation. Let's say we want to take our original arrowhead object and make it beat like a heart.

First, we would need to define how the scale will change over time. To get a rhythmic motion, we can combine a couple of sine waves whose frequencies are small multiples of each other. The "angle" that we feed the sine functions is time. For this we'll use a multiple of the frame counter, `i`.

```
theta = i / 7
scale = 3 + 0.6 * (
    np.sin(theta) + 0.4 * np.sin(3 * theta))
```

Having a scale that is a sinusoidal function of `theta` means that every time `theta` passes $2 \times \pi$, about 6.3, the cyclical pattern will complete and start over. (Keep in mind that Numpy does all of its trigonometry in radians, so a complete cycle is 2π radians, rather than 360 degrees.) Because we are using `i/7` as `theta`, a complete cycle will take about 44 frames. And because we are animating this at 24 frames per second, a complete cycle will take a little over 1.8 seconds, about 33 beats per minute. This arrowhead is extremely chill.

Now that the heartbeat-like scale has been crafted, we can perform the scaling transformation.

```
scaled_path = (path - anchor) * scale + anchor
```

Here we can make a simplification. We intentionally defined our path so that the anchor was at (0, 0). That lets us perform the scaling more concisely.

```
scaled_path = path * scale
```

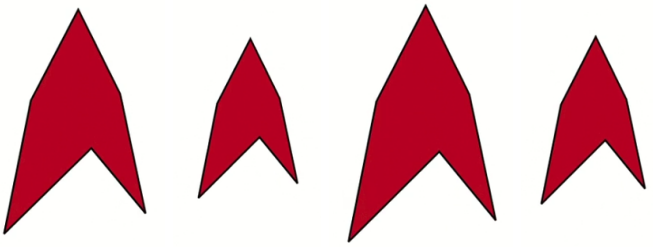
And finally, we update the patch object to complete the job.

```
patch.set_xy(scaled_path)
```

Here are all the snippets above, condensed into a single function as it appears in the animation code. I haven't bothered to repeat the rest of the code here, since it is quite similar to the code we've already walked through.

```
def pulse_patch(patch, path, i):
    theta = i / 7
    scale = 3 + 0.6 * (
        np.sin(theta)
        + 0.4 * np.sin(3 * theta))
    patch.set_xy(path * scale)
```

patch_scaling.py at tyr.fyi/4files



The arrowhead gets bigger and smaller. You get the idea. Check out the video at tyr.fyi/4pulsing. Trust me, it's way better than this picture.

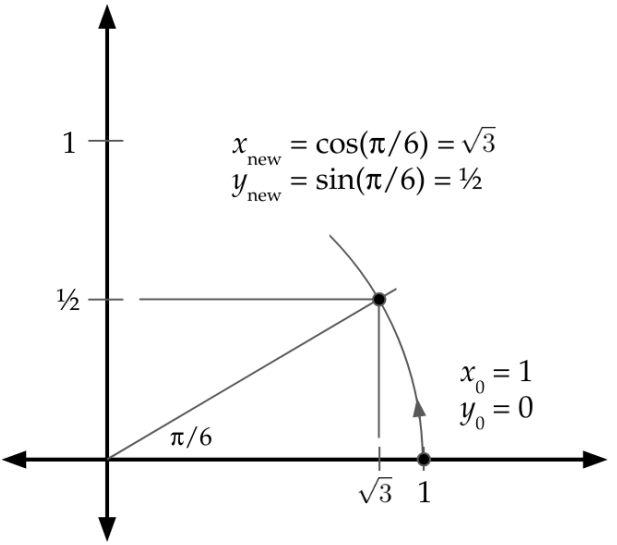
Notice how the center of the object stays put. Every point in the path is moving in a different direction, but the overall effect is that the object is motionless, growing and shrinking to the beat. Cool, right?

Rotation

The third of our three musketeers of 2-D transformations is rotation–getting a thing to twist and spin. We get to reuse the notion of an anchor here. This is the point that stays put as we spin the object. As with scaling, every point in the path will move in a somewhat different way. It will all depend on where it sits relative to the anchor.

Rotation is particularly fun, because the math starts to look impressive. It has both trigonometry and matrix multiplication. On first exposure it can all seem pretty intimidating, but don't let that scare you off. We'll take it one step at a time.

We can build a gut feel for how rotation works by starting with some well chosen examples. Consider a point sitting on the x-axis at (1, 0). If we want to rotate it by some angle, say 30° (or $\pi/6$ radians) about the origin, it would move to a new position. The new x would be $\cos(\pi/6)$ or $\sqrt{3}$. The new y would be $\sin(\pi/6)$ or ½.

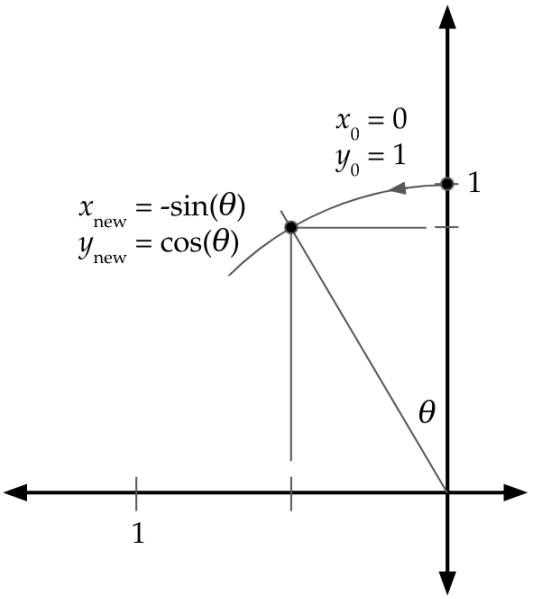


We can generalize that out to any angle of rotation, θ, and say that the new x will be the cosine of θ, and the new y will be the sine of θ. This works for any angle, positive, negative, greater than 360° (2π rad), it doesn't matter.

If our point on the x-axis is at some other location, x_0, we can extend our initial result by multiplying everything by x_0.

$$x_{new} = x_0 \cos(\theta)$$
$$y_{new} = x_0 \sin(\theta)$$

We can do a similar exercise for a point that sits on the y-axis. A point at (0, 1), rotated by θ will end up with x_{new} as the negative sine of θ, and the new y_{new} as the cosine of θ.



And for a point at (0, y_0) this extends to

$$x_{new} = -y_0 \sin(\theta)$$
$$y_{new} = y_0 \cos(\theta)$$

Because of the fabulous properties of math, the fact that our x- and y-axes are orthogonal (rotated 90° from each other)

means that we can just squash these two results together to figure out what would happen if we started with a point at (x_0, y_0) and rotated it by θ.

$$x_{new} = x_0 \cos(\theta) - y_0 \sin(\theta)$$
$$y_{new} = x_0 \sin(\theta) + y_0 \cos(\theta)$$

Using matrix notation, we can rewrite this as

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} x_0 \cos(\theta) & - & y_0 \sin(\theta) \\ x_0 \sin(\theta) & + & y_0 \cos(\theta) \end{bmatrix}$$

which is the same thing, but in matrices.

The matrix on the right hand of the equals sign has some nice structure to it. We can actually break it apart and express it as the matrix product of two separate matrices.

$$\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

If you've never seen it before, matrix multiplication is a choreographed sequence of multiplications and additions. No individual piece of it is very fancy, but the end result can look quite fancy indeed. If you'd like to take 15 minutes to get a peek at the inner workings, please take a look at tyr.fyi/4matmul.

You may be wondering why we're going out of our way to get mathy here. Matrix multiplications are a convenient way to represent transformations of many sorts. They come up again and again in robotics, signal processing, and machine learning. In the world of math, they go by the terrifying name of **linear algebra**. But when you get up close, all those

snarling teeth turn into a giant cuddly bear. They're just a bunch of multiplications and additions.

When we sit down to write this in Python, it ends up making things easier if we swap our rows for our columns, and vice versa. This swap is called **transposition**, and it comes up a lot in linear algebra. The only catch is that when we take the transpose of a matrix multiplication, we have to change around the order of the matrices. After taking the transpose of both sides of our rotation transformation, it gets a bit rearranged.

$$\begin{bmatrix} x_{new} & y_{new} \end{bmatrix} = \begin{bmatrix} x_0 & y_0 \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

Thanks to the cleverness of matrix multiplications we can use this form to rotate a whole collection of points at once.

$$\begin{bmatrix} x_{0\,new} & y_{0\,new} \\ x_{1\,new} & y_{1\,new} \\ x_{2\,new} & y_{2\,new} \\ x_{3\,new} & y_{3\,new} \\ x_{4\,new} & y_{4\,new} \\ x_{5\,new} & y_{5\,new} \end{bmatrix} = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_5 & y_5 \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$$

This will come in handy as we are working with our path, a 2-column matrix with one row for each point. In Python the rotation transformation matrix is easily recognizable.

```
rotation = np.array([
    [np.cos(angle), np.sin(angle)],
    [-np.sin(angle), np.cos(angle)],
])
```

To do the matrix multiplication, we can take advantage of the `@` operator, which Numpy has hijacked and uses for this purpose.

```
rotated_path = path @ rotation
```

If you prefer a function, you can use `matmul()` and get the exact same result.

```
rotated_path = np.matmul(path, rotation)
```

In either case the most important thing to remember is that order of the arrays matters. `A@B` and `B@A` are different things.

Now that we have a rotation worked out, there's nothing that prevents us from doing several transformations on the same shape. We can stack them, doing one after the other until we get the effect we want.

This example shows our arrowhead shape performing both a rhythmic rocking, a rotation, and bobbing up and down, a translation.

This arrowhead swings back and forth
while bouncing up and down.
Check out the video at tyr.fyi/4rocking.

In code, we do the same thing we would do for a rotation and a translation in isolation, but we do them one after the other. The fact that we use the anchor point as a reference and track it as it undergoes various transformations buys us this flexibility.

Here's the whole sequence. It's a lot. We'll break it down.

```
def step_animation(patch, path, i):
    anchor = np.array([[0, 1]])
    anchored_path = path - anchor

    scale = 2.5
    scaled_path = anchored_path * scale

    theta = i / 10
    angle = 0.5 * np.sin(theta)
    rotation = np.array([
        [np.cos(angle), np.sin(angle)],
        [-np.sin(angle), np.cos(angle)],
    ])
    rotated_path = scaled_path @ rotation

    x_translation = 0
    y_translation = 20 * (1 - np.cos(angle))
    translation = np.array([[
        x_translation, y_translation]])
    translated_path = rotated_path + translation

    transformed_path = translated_path + anchor

    patch.set_xy(transformed_path)
```

step_animation() from patch_rotation.py at tyr.fyi/4files

Step 1. Choose the point (0, 1) as the anchor, and move the path so that the anchor sits at (0, 0) .

```
anchor = np.array([[0, 1]])
anchored_path = path - anchor
```

Step 2. Scale the path up by a factor of 2.5.

```
scale = 2.5
scaled_path = anchored_path * scale
```

Step 3. Rotate the arrowhead in a back and forth rocking manner. Use an intermediate variable `theta` to track progress through each rocking cycle. Every time `theta` reaches a multiple of 2π the cycle will start over. The variable `angle` is the actual pendulum angle, measured as a deviation from the original position.

```
theta = i / 10
angle = 0.5 * np.sin(theta)
rotation = np.array([
    [np.cos(angle), np.sin(angle)],
    [-np.sin(angle), np.cos(angle)],
])
rotated_path = scaled_path @ rotation
```

Step 4. Re-use the variable `angle` here to get a cyclical up-and-down translation that is in sync with the rotation.

```
x_translation = 0
y_translation = 20 * (1 - np.cos(angle))
translation = np.array([[
    x_translation, y_translation]])
translated_path = rotated_path + translation
```

Step 5. Undo Step 1 to restore the original position.

```
transformed_path = translated_path + anchor
```

And that's the whole sequence of operations for doing a combined scaling, rotation, and translation.

This is an important milestone! While there are technically other transformations we could do to a 2-D object, these are all of the ones that preserve its shape. In math terminology, they are similarity-preserving, which means that the angle at each vertex remains unchanged. It is a fantastic starter kit for programmatic animation.

That was a lot. I need a nap.

Point-to-point movement

The next item of business is to consider how to move something around. In the examples so far we have modified scale, angle, and position as sinusoidal oscillations. This is useful, but not complete. We are also going to want to move a thing from one point to another and have it stay there, at least until we tell it to move again.

Let's put our arrowhead in space and call it a ship. Our goal is to get this ship to fly a triangular path. It has three points in space that it needs to hit. This translates to six poses, because each time it reaches a point, the ship needs to then rotate in place so that it can fly in the direction of its next point.

To put some concrete numbers to this, let's say the three points are (3, 4), (5, -7), and (-5, 0). The heading that the ship needs to take to get from one point to the next, measured as rotations from the original orientation in which we drew the arrowhead, are approximately 1.7π, 1.05π, and .3π radians.

(We are going to be sticking with radians instead of degrees. They are what Numpy uses, and it will save us from having to do a lot of back and forth conversions. If you are still building an intuition for radians a good starting point is remembering π = 180°. Everything else falls out from that: π/2 = 90°, π/4 = 45°, π/6 = 30°, 2π = 360°.)

We also need to specify how much time we want the ship to take getting from one pose to the next. The most natural unit of time in animation is number of frames. Because we're working at 24 frames per second, transition times in the

range of 15 - 35 frames will give us movement that is slow enough to watch but fast enough to be interesting.

Here is our full set of pose points.

point	coordinates	heading (rad)	frames to get here
A	(-5, 0)	1.7π	1
B	(3, 4)	1.7π	30
B	(3, 4)	1.05π	18
C	(5, -7)	1.05π	29
C	(5, -7)	.3π	17
A	(-5, 0)	.3π	33
A	(-5, 0)	-.3π	19

This will play on repeat. Note how the final angle is different from the initial angle by 2π. This is to "uncoil" the angle as the ship goes around in a circle. This full-spin as we jump from the last row back up to the first happens in a single frame and so it leaves no visual evidence.

The notion that there are a few important poses that need to be smoothly connected is a useful concept. The details about what objects do in between those poses can really affect the feel of the animation. In artistic animation these important poses are called keyframes, and the smooth movement between them is called easing. Subtle differences in easing functions can give animations a distinctly different feel.

Given two points and a set amount of time to move an object from one to the other, a reasonable starting point is a

constant velocity trajectory along a straight line path between them. To traverse a path between A and B in 37 frames, we might draw a line between them and chop it up into 37 equal steps. Then we can advance the object one step per frame.

Moving between pose (`x_i`, `y_i`, `theta_i`) and (`x_j`, `y_j`, `theta_j`) in `n_frames` is three lines of code in Numpy.

```
xs = np.linspace(x_i, x_j, n_frames)
ys = np.linspace(y_i, y_j, n_frames)
thetas = np.linspace(theta_i, theta_j, n_frames)
```

This approach dedicates one frame to the starting point and one frame to the end point.

For our spaceship we can see what this looks like for each leg of the triangle. We can also use a constant rotational velocity to pivot between legs.

See the video at tyr.fyi/4constant.
The full code is in spaceship_triangle.py at tyr.fyi/4files.

The resulting motion certainly does the job of getting the spaceship where it needs to go, but the sudden starts and stops coupled with the artificially constant movement between them leaves the animation looking clunky and cheap.

It's also weird to our eyes because it violates the laws of physics. Anything with mass, such as a spaceship, can't go from a dead stop to a steady speed in an instant. It would require infinite force and infinite energy. Our eyes aren't used to seeing that in the world, so our brains interpret this as a very fake spaceship.

We can do one step better by imagining how the spaceship would move if it had mass. We will cover this in a lot more detail in the next chapter on simulation, but for now it's enough to claim that if we were to push on our spaceship with a constant thrust, it would exhibit a constant acceleration, a smoothly ramping velocity. To move from point A and come to a stop at point B, we need a triangular velocity profile. A smoothly ramping to a peak at the midpoint, then smoothly ramping back down, will do the job. It will show constant acceleration for the first half of the journey, then constant acceleration of equal magnitude in the opposite direction for the second half of the journey. This is actually not too far off from how an actual spaceship would go about making this trip.

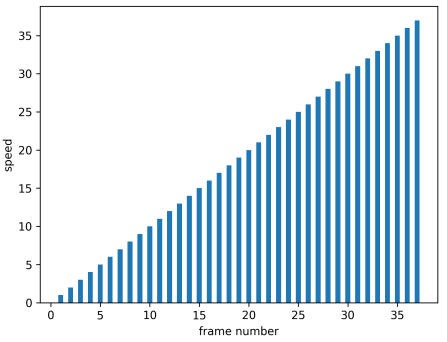
It's worth a walk through it because we'll use a similar process in the future. To move between (`x_i`, `y_i`, `theta_i`) and (`x_j`, `y_j`, `theta_j`) in `dt` frames, first calculate how much x, y, and θ are going to change.

```
dx = x_j - x_i
dy = y_j - y_i
dtheta = theta_j - theta_i
```

To construct the nice symmetric isosceles triangle of speed that we want, we'll do some array juggling. First construct the ramp-up portion with its peak on the far right by generating an array of all ones, then doing a cumulative sum across them using the unfortunately named `np.cumsum()`. The cumulative sum is the discrete version of the cumulative integral. The value at each point is the sum of all the original array values that came before.

```
right_triangle = np.cumsum(np.ones(dt))
```

Here's the pattern that results.

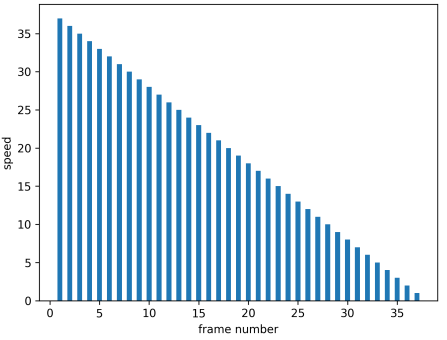


To create the trailing edge we can use the Numpy indexing trick for reversing an array, `[::-1]`. It says to start at the end

of the array and walk back toward the beginning one element at a time.

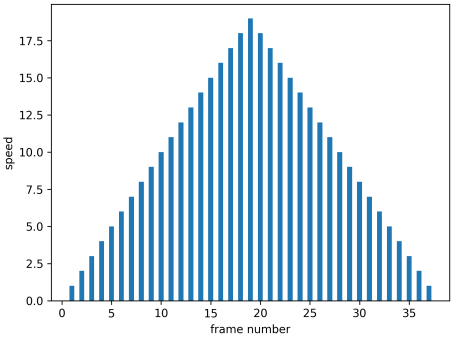
```
left_triangle = np.cumsum(np.ones(dt))[::-1]
```

And after verifying this in a bar chart, we can see that it makes a perfect left-leaning triangle.



The `np.minimum()` function will combine them in the way we want, taking the lower of the two values at each point.

```
triangle = np.minimum(right_triangle, left_triangle)
```

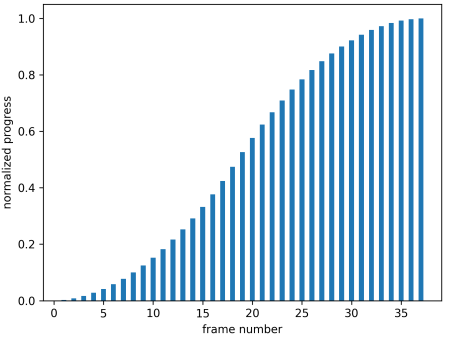


This triangular shape is exactly what we are after. The only question that remains is how tall it should be. We're going to take a shortcut on the math for figuring out the velocity and position at each time step. We just need a velocity profile of the right shape. It's OK if it is not scaled correctly, because our next step is to normalize it — to divide every value by the sum of all the values.

```
triangle *= 1 / np.sum(triangle)
```

After normalizing, the sum of all the velocities over all the time steps is one. We don't know what the peak velocity is and we don't have to care. The next step is to take the cumulative sum of these to turn them into a progress tracker.

```
progress = np.cumsum(triangle)
```



This results in a set of points that progress from 0 to 1. Notice they start slow, move quickly through the middle, and then end slow again. This is the fraction of total distance covered. In this form we can scale it with the total distance covered in each dimension–`dx`, `dy`, and `dtheta`–to get the position in each dimension over time. Adding these to the initial values of each gives us the position at each time point

in each dimension for a trajectory with a triangular velocity profile.

```
x_t = x_i + progress * dx
y_t = y_i + progress * dy
theta_t = theta_i + progress * dtheta
```

These last two steps we will re-use. For a speed profile of any shape, or we can generate an array describing it. We don't need to worry about scaling, because the next step is just to normalize the speed profile so that the whole thing sums up to one. In this form, it describes what fraction of the trip should be taken in each time step.

You can run this yourself or take a look at the result at tyr.fyi/4triangular. Notice how smoothly the ship stops and starts. It has a sense of mass, a feeling of momentum. Watch it back to back with the constant velocity animation to get a sense of how much of an improvement it is.

You could stop right here

This concludes our coverage of the tools we'll need to animate robots. The ability to plot points and polygons and move them around at will gives us the power to transmit a ton of information to a human eye. It's also great for showing off cool things you've built. We're going to have a lot of fun with this in subsequent chapters.

The rest of this chapter focuses on how to make movement look like a human waving their hand around in the air. This is a rabbit hole I got to crawl down during my PhD research, and I couldn't resist sharing it with all of you, however

tenuously relevant. If you tell me you read and enjoyed it out of politeness, I promise not to ask any uncomfortable follow up questions.

Minimum-jerk

How things move matters. As humans we have a few perceptual superpowers, things that we are especially good at picking up on. One of these is patterns and quirks of movement. We can identify a friend from a long way off by how they walk, before they are close enough to see their face. You can even tell a lot about their state of mind. They'll move in noticeably different ways when they are angry, exhausted, nervous, sad, or jubilant.

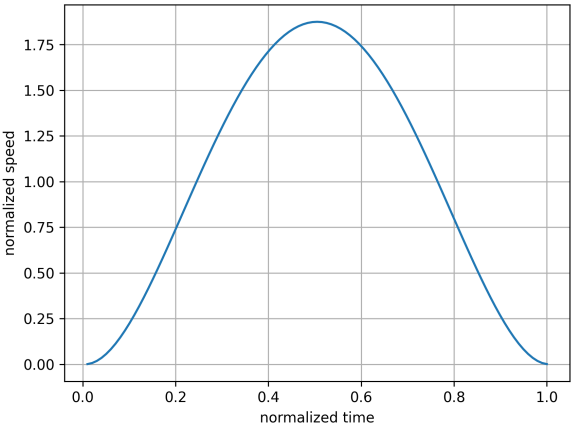
Looking very closely at how a human makes simple movements, moving their hand in front of them from one point to another, shows a fascinating pattern. It's a smooth movement, starting from rest, ramping up to a peak speed, then easing back to rest again. But when measured and analyzed, it's quite distinct from the triangle velocity curve we just created. It's bell-shaped. There are no sudden jumps in acceleration.

After comparing a lot of candidate models for describing this, one that does a surprisingly good job is a minimum-jerk curve. For the curious, it is the velocity curve that results from minimizing the integral of the square of jerk, the derivative of acceleration. But that's not the important part. What matters is that the *progress* from point-to-point, the fraction of the total distance covered, can be expressed as a concise 5^{th}-order polynomial

$$progress(\tau) = 10(\tau)^3 - 15(\tau)^4 + 6(\tau)^5$$

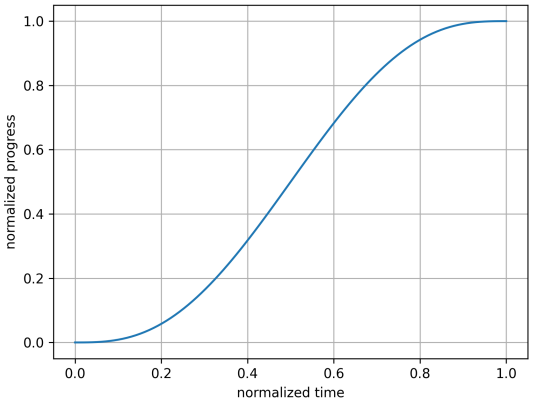
where τ is the normalized time, 0 at the start of the movement and 1 at the end. Substituting in a 0 for τ gives $progress(0) = 0$ and substituting in a 1 for τ gives $progress(1) = 1$, as we would hope. This is a continuous version of the progress plot we made two pages ago.

Looking at the slope of the progress curve, the normalized speed profile, we can observe how it is similar to the triangular speed profile and how it's different.



Both of them start and end with a speed of zero, reaching a peak at the halfway point. Both are symmetric about the center. But where the peak of the triangle is sharp, the peak of the minimum jerk curve is smooth and rounded. And the start and the end of the min-jerk curve are somewhat flattened. They take their time getting going and settling back down.

These translate into some subtle but noticeable differences in the min-jerk progress curve.



Because the speed curve's peak is rounded and spends more time close to its maximum value, the center portion of the progress curve resembles more of a straight line. And because of the flattened ends of the speed curve, the ends of the progress curve are also flattened. The min-jerk curve takes more of its movement time to get going and to slow back down.

Now take a look at the spaceship animation with minimum jerk point-to-point movements at tyr.fyi/4minjerk. If you toggle between it and the triangular speed profile animation (tyr.fyi/4triangular) you can see a subtle difference in the quality of the movement. To my eye, the min-jerk movements look more organic, and the triangular speed profiles look more like a precision robot at work.

Your opinion on the aesthetics may be different. The important thing to notice here is that subtle differences in how a thing moves between two points conveys a different feel.

The difference is more than aesthetic. If we were to break down both the triangular and minimum jerk speed profile into their respective sinusoidal frequency components by means of a Fourier transform, we would find that the triangular profile has more high frequency content. In order to make those sudden changes in acceleration, you have to include a lot of high frequency sine waves. This is more than a mathematical curiosity. In practice, this means that any high frequency dynamics of your robot can be excited when you move. It's the same thing that happens when ringing a bell. A hammer strike has most of its energy at higher frequencies. This allows it to excite the high frequency vibrational modes of the bell, to get that bell really resonating for us, making a pleasing peal. In a robot this is not what we want. There are lots of mechanical pieces of all sizes that we hope will stay rigid. We like to pretend that robot arms do not bend and that sensors move steadily and gradually, rather than shaking rapidly. Exciting high frequency modes of vibration in our apparatus is an invitation to fatigue, malfunction, and catastrophic failure.

A minimum jerk speed profile, on the other hand, can be represented using mostly lower frequencies, due to its smoothness. Even when executed at high speed it doesn't excite high frequency vibration nearly as much as a triangular speed profile. There are good practical reasons to

make smooth point-to-point movements when working with mechanical hardware. It's more than just looking cool.

I promise I'm not bored. This is my interested face.

Logit-normal

Interestingly (to almost no one besides myself), minimum jerk is not even the smoothest of smooth speed profiles. There is a more complex variant shown to fit human movements even better than minimum jerk, called the logit-normal. Although in fairness to minimum jerk, logit-normal cheats by being able to modify its skew (leaning from one side or the other) and kurtosis (thickness of the central peak).

The difference between logit-normal and minimum jerk is subtle, and probably not observably different to the human eye, but it has some fun properties. Our constant velocity movement was discontinuous. Velocity jumped suddenly between zero and a steady value. Another way to say that is that it was discontinuous in the first derivative of position.

The velocity of our triangular velocity speed profile was continuous, but its acceleration was discontinuous. It jumped between zero, a positive value, a negative value, and back to zero. It was discontinuous in the second derivative of position.

The velocity of our minimum jerk curve was continuous, as was the acceleration, but it did have a discontinuity in the third derivative of position, jerk.

Logit-normal is infinitely differentiable. There is no derivative of it that is discontinuous at any point. I don't know if that's practically significant, but it's pretty damn cool.

The shape of the speed profile can be expressed directly as a function of normalized time, τ, which varies between 0 at the start of the movement and 1 at the end. It is given by

$$v(\tau) = \frac{1}{\tau(1-\tau)} e^{-\frac{\left(\log\left(\frac{\tau}{1-\tau}\right)-\mu\right)^2}{2\sigma^2}}$$

for $\tau > 0$ and $\tau < 1$ and is 0 otherwise.

This seems like a sprawl of symbols (and it is) but we can break it down a little bit. It might remind you of the Normal distribution.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

We can ignore the factor out front.

$$\frac{1}{\sigma\sqrt{2\pi}}$$

It's a constant, and we are focused on the shape of the curve, rather than its amplitude. It gets rescaled away when we normalize it.

We can get one step closer if we take out the x and substitute in the logit function.

$$logit(\tau) = \log\left(\frac{\tau}{1-\tau}\right)$$

The logit is good for lots of things in statistical modeling, but it's particularly well known as a log-odds-ratio. Notice that when τ is 0 or 1, it behaves terribly. In fact, it is only defined on the open interval (0, 1).

And the last step to building the logit-normal is to add a $1/\tau(1 - \tau)$ term to the beginning of the equation. Mind-twisting factoid: this happens to be the derivative of the logit! I'm sure this is important for some reason, and I'm awaiting the day when a patient mathematician will explain it to me.

The logit-normal has quite a bit of flexibility in the shape it can take. The μ parameter controls skew. $\mu > 0$ makes the peak lean to the right (the maximum speed occurs later in the movement) and $\mu < 0$ does the reverse. $\mu = 0$ gives a symmetric curve. The σ parameter controls the fatness of the central curve. $\sigma = .7$ is an approximate match to the shape of a minimum jerk curve. A higher value of σ gives a thicker peak–it inches closer to a constant velocity movement. A lower value of σ gives a thinner peak. It gets going slowly, rushes through the middle of the movement, then takes a longer time to slow back down.

To make point to point movements look just a little more organic, we can randomly vary μ and σ a little bit. When you fit logit-normal curves to actual human movements, there is some variation in their skewness and kurtosis. Adding in this subtle variation gives the movements an extra texture and helps them seem as if someone is puppeting the object by hand.

Another trick is to overlap subsequent movements a little bit, rather than having the object come to a complete stop. This is a thing that humans do in the course of making normal movements. In fact, the more skilled they are, the more overlapped and blended together these movements become. Because they become small components of a larger, composite movement, it's helpful to refer to these individual pieces as **submovements**.

Take a moment to watch the end result at tyr.fyi/4logit.

To my eye it almost looks like a toddler waving a toy spaceship around, making PEW PEW PEW noises. The overlapped submovements blend the legs of the triangle together, and make the rotations smoothly integrated with the rest of the movement.

Making an animation as human-like as we can is a fun exercise, but probably more for its entertainment value than anything else. Its real value will come later, when we're actually making hardware move. When commanding an arm or a leg or a finger or a tentacle to move from one point to another, we have to be very specific about how we tell it to get there. Specifying a super-smooth logit-normal speed profile will minimize wear and tear on our hardware. It will keep us from unnecessarily making it "ring" at higher frequencies and avoid some really pathological failure modes.

Historically, robots have gotten around issues of shaking, bending, and vibration by being built more rigidly. Linkages are thicker and heavier, bearings are more tightly toleranced, gears are more precisely machined. To make all this move in

a controllable way, actuators also have to get beefier, and power supplies have to grow proportionally as well. The result is a chonky behemoth–heavy, expensive, and despite its over-engineering, delicate.

Making smoother movements is a way to break this cycle. Floppy robot arms and jittery sensors behave better when moved in a way that doesn't excessively excite high frequency resonances. Lower rigidity requirements means hardware becomes lighter, cheaper, and less power hungry. Smooth movements open up entire new platforms and price points for experimentation. It's a democratizing force.

In addition, it's also nice to have hardware that moves in a predictable way. We humans are so familiar with our own smooth movements, as well as those of others, that we can predict the end point of a movement in flight with surprising accuracy. This short term prediction capability allows us to make corrections for our own movement errors before we're even done committing the error. And it makes it easier to interact with other movement-makers when you can guess where they're headed in the next half second. By making robots move in a way we're already programmed to recognize, it might make them more natural to interact with.

What's next?

I can't believe you stuck around for that whole tangent. As a reward, here's a sneak preview of the next chapter.

Now that we can animate things, we need to think more about the worlds that we will be animating. They will all be based on the physical world. Robot arms and wheels and grippers and linkages and actuators will all be doing their thing in an imagined world, patterned after our own. Objects will have mass and occupy space. Newton's Laws will be in force. We hope that whatever cleverness we manage to encode to navigate simulated worlds will help robots to do the same thing in our world. For this, the rules in simulations need to resemble those that physical robots face as closely as possible. This is what we'll tackle in the next chapter–simulations with plausible physics.

Recap

Matplotlib is a flexible plotting library for Python. It's tightly integrated with **Numpy**.

Matplotlib is built using an **Object Oriented Programming** approach. Almost everything important is defined as an object class, with its own attributes (variables) and methods (functions).

The **animation process** we use is to repeatedly modify the data underlying Line and Patch objects, updating the plot each time.

The similarity preserving **transformations** of scaling, rotation, and translation can be performed via matrix multiplications. It's important to properly account for the object's **anchor** point.

The way in which an object moves from **point to point** strongly affects how it is perceived by a human viewer. Smooth speed profiles, such as minimum-jerk and logit-normal, are particularly easy on hardware and reminiscent of human movement.

Resources

Matplotlib can do a lot, and it is a big package. Luckily there is some fabulous documentation at tyr.fyi/4matplotlib. I pulled out some of the pieces that I find myself searching for all the time and collected them at e2eml.school/133. It has all the tricks I use to get plots looking just the way I like.

Equations were generated in a Jupyter notebook equations.ipynb at tyr.fyi/4files.

Figures were generated using the Python script figures.py at tyr.fyi/4files.

Way back before he was my PhD advisor, Neville Hogan made the observation that minimum-jerk velocity profiles fit human movements surprisingly well and, as a bonus, have a strong motivating principle. Reza Shadmehr did a great overview and derivation of the concept at tyr.fyi/4shadmehr.
Neville Hogan (1984) Adaptive control of mechanical impedance by coactivation of antagonist muscles. IEEE Transactions on Automatic Control. AC-29: 681-690. tyr.fyi/4hogan

Nine years later Plamondon et al. compared a looooong list of speed profile candidates together and found that logit-normal (tyr.fyi/4logitnorm) fit better than any of them. They referred to it with the descriptive name "support-bounded lognormal" and rearranged the parameters, but it was the logit-normal underneath.
Réjean Plamondon, Adel M. Alimi, Pierre Yergeau, Franck Leclerc (1993) Modelling velocity profiles of rapid movements: a comparative study. Biological Cybernetics. 69, 119-128. tyr.fyi/4plamondon

About the Author

Robots made their way into Brandon's imagination as a child while he watched The Empire Strikes Back on the big screen, one buttery hand lying forgotten in a tub of popcorn. He went on to study robots and their ways at MIT and has been puzzling over them ever since. His lifetime goal is to make a robot as smart as his pup. The pup is skeptical.

To see more of his work, visit brandonrohrer.com.