Manim Animations

Intro

Visualizations are essential when trying to communicate scientific results. While text is often required to be mathematically precise, giving intuition and an initial understanding is far easier with a good picture. The `manim` package allows us to take this one step further with animations of mathematical concepts. If you have seen the Youtube channel 3blue1brown, these animations will look familiar because that is what he uses. Today we will learn some of the basics.

Installation

The starting code for this lab comes with a `pyproject.toml` file that specifies the `manim` package as a dependency. On top of that, we are also going to use `jax` in this lab, which is also included in the pyproject.toml file.

python environment

If you decided to use `virtualenv` for this lab, that is to create an environment with `python -m venv [ENV_NAME]` and activate it with `source [ENV_NAME]/bin/activate`, then you can install the dependencies with `pip install -r pyproject.toml`.

If you have `uv` installed, you can also use `uv` to install the dependencies with `uv sync` to create the environment.

Extras

There will be some additional dependencies that you need depending on your operation system, check out the instruction on manim's website to make sure you have all the prerequisites installed.

To ensure `manim` is installed correctly, on the command line run `manim --version`. You should see some text like `Manim Community v[Version]`.

Basic Examples

Creating Shapes

Create a file `quickstart.py`. Inside this file, add the following:

```python
import manim as mn

class CreateShapes(mn.Scene):
  def construct(self):
    # your code here
```

Each animation is an `mn.Scene`. Each scene will have a `construct` function which includes all the code to create and animate the objects in the scene.

Now to create shapes. There are many common shapes available in manim, such as `Circle`, `Square`, `Triangle`, `RegularPolygon`, etc. Create a circle with `circle = mn.Circle()`. This just initializes the object in code, so to play the animation, call `self.play(mn.Create(circle))`.

(Alternate ways to create: https://docs.manim.community/en/stable/reference/manim.animation.creation.html)

Manim Animations

Now that the code is written, we need to run it. In your terminal, run `manim -pql quickstart.py CreateShapes`. This should automatically open up a player where you can run the animation. It will also save it in `./media/videos/quickstart/480p15/`.

You can create multiple shapes by declaring multiple shapes, then calling them all at once like `self.play(mn.Create(shape1), mn.Create(shape2), ...)` and so on. Try a couple now, they should be created on top of each other.

Transformations

The `manim` package can be used to automatically transform shapes into other shapes. Create another class `SquareToCircle`, and in its `construct` function create a circle and a square. Animate the square creation as in the previous section. To transform the square into a circle, use the `Transform(old_object, new_object)` function inside a `self.play` call. This will take the attributes of new_object and give them to old_object, modifying old_object.

For a more challenging task, create a new class `CircleConvergence`. The goal is to inscribe a n-gon inside a circle using `RegularPolygon`, then slowly transform it by adding more sides until the n-gon looks indistinguishable from the circle. Use the code block below as a skeleton.

```
class CircleConvergence(mn.Scene):
  def construct(self):
    # initialize the circle and the list of regular n-gons
    # play the animation to create the circle and first n-gon (probably a triangle)

    # loop through the ngons, skipping the first one
    for ngon in sequence_of_ngons[1:]:
        # transform the current ngon to the next ngon
```

Attributes and Set Functions

In addition to different kinds of shapes, each shape also has a number of attributes that can either be defined at construction, or set later with an appropriate set function. Some options include border color, fill color, rotation, or position. For example, suppose we have a `circle` and a `square`. Then we can:

`circle.set_fill(mn.PINK, opacity=0.5) # set the color fill and opacity`

`circle.rotate(mn.PI / 3) # rotate the circle`

`square.next_to(circle, mn.DOWN, buff=0.5) # put square below circle, with 0.5 buffer`

`circle.shift(mn.LEFT) # shift the circle left`

Construct a few shapes and play around with all of these. All these are static attributes of the shapes, but we can also animate them. For any of these calls, we can prepend `.animate` to return an animation that will go in a self.play call. For example,
`self.play(circle.animate.rotate(mn.PI / 3))` will animate the rotation of circle.

Construct 4 different shapes in grid with different colors and rotations. Make some of the attributes initial, and some animated.

Update the `CircleConvergence` script to display the area of the n-gon as it trasnforms, which is an estimate of π. As the number of sides increase, the area should approach π. You can do this with a text

object `label = mn.Text(f'pi estimate{area:0.5f}')` which is created with `mn.Write(label)` inside a play call.

Changing rendering configuration

You may notice the output video is kind of low res and choppy, that is not because of manim sucks but because of the default rendering configuration. You can change the rendering configuration by creating a `manim.cfg` file in the same directory as your script. Here is an example of a `manim.cfg` file that calls the `-pql` flags that we were using before:

```
[CLI]
preview = True
quality = low_quality
```

Modify the `manim.cfg` to set the `background_color` to `WHITE` and the quality to `high_quality`. Then you can run `manim <file.py> <Scene class>` to render the new content.

Animating your gradient descent with `jax` and `manim`

Here is a relevant example from the `manim-community` gallery.

Step 1: Create the target function with `jax`

The first step is to create the target function that you want to minimize. Go ahead and pick your favorite function that has a scalar input and a scalar output. You can use `jax` to create the function such that later on when we need the gradient, you can just use `jax.grad` to get the gradient.

Once you have define the function. Let's create the axes and plot the function itself

Step 2: Plot the static elements

We need to add the axes, labels, and the graph of the target function. Pick sensible values for the ranges of your graph depending on your chosen target function.

```
ax = mn.Axes(
    x_range=[min_x, max_x, x_ticks],
    y_range=[min_y, max_y, y_ticks],
    axis_config={"include_tip": False},
)
labels = ax.get_axis_labels(x_label="x", y_label="f(x)")
graph = ax.plot(target_function, color=mn.MAROON)
```

Since these are static elements, we can add them with `self.add(ax, labels, graph)`, rather than animating them.

Step 3: Initialize a point in the domain

Next, let's draw an initialization point in the domain. Generate a random number in the x domain using jax.random, or pick a point. We will use a `ValueTracker` to allow for animating this point along the target function curve.

```
t = mn.ValueTracker(initial_x)
```

To plot this point, we will use the `Dot` class. It takes a Point3D as a constructor, which we need to convert from the x/y coordinates on the axes using the function `ax.coords_to_point(x,y)` or `ax.c2p(x,y)` for short. Put this together it will be, for the appropriate `x,y`:

```
dot = mn.Dot(point=ax.c2p(x,y))
```

Finally, we have to set dot to update properly as the value tracker changes:

```
dot.add_updater(
    lambda x: x.move_to(ax.c2p(t.get_value(), target_function(t.get_value()))),
)
```

Step 4: Run the gradient descent

Now we will run the gradient descent. Recall that the formula is $x_{i+1} = x_i - \eta \nabla f(x_i)$ where η is the learning rate. Loop this for some number of steps, and in each step, animate the following:

Step 4a: Render the gradient vector

The next step is to evaluate the gradient and draw it in manim. This link could be useful for that.

If you want to be fancy, you can first draw the original gradient vector, then scale it by the step size. You can also highlight the gradient vector using `manim`'s `indication` function. Make sure when you create the arrow to use `ax.c2p` to get the proper locations for the start and end points. Also make sure to create `Arrow` with `buff=0`.

Step 4b: Update the point

Once we have shown the audience how the gradient look like, let's use the gradient to update the point. This is down in a single step with `self.play(t.animate.set_value(x_next))` where `x_next` is the next x value.

Step 4c: Remove the gradient vector

You can remove the old gradient vector with the `self.remove` function.

Step 5: Speed up the animation

In order to converge, we probably need to run thousands of steps. It would be a pain to keep it at the same speed when we are illustrating the idea for that long. So the last step is to speed up the animation.

The can be done by choosing the run time of the animation to be shorter than the actual time it takes to run the animation.

Further Animations

What would you like to animate?

- Something from your project?
- The Nintendo GameCube opening animation?
- Some kind of 3d surface? https://docs.manim.community/en/stable/reference/manim.mobject.three_d.three_dimensions.Surface.html
- etc.