Introduction to Manim

Joon Kim
jkim@ivycollegiateschool.org
Ivy Collegiate School

1. Introduction

The visualization of mathematics had been a big issue between educators and mathematicians. There were various attempts for creating the mathematical visualization engine. Grant Sanderson, who is now widely known as 3Blue1Brown, was also interested in the visualization of mathematics. He created his own engine in the form of a Python package, and that was the start of the Mathematical Animation Engine, which is also called Manim.

Manim includes the elements of outside packages or codes such as NumPy or LaTeX. This also means the users should have some knowledge of those, including mathematical knowledge. For example, when the users want to return a function such as $\log(x)$, they should know the **numpy.log** function in the NumPy package. Furthermore, one of the Manim's main functions is the writing of mathematical equations in the videos, which requires the knowledge of TeX coding language.

2. Basic of Manim

The following code is the basic form of Manim code :

```
from manimlib.imports import *
class Video_Name(Scene) :
     def construct(self) :
          # Your Video Contents
```

In line 1, the code loads every function from the package called manimlib. This process is similar to the preparation of materials, which will be used for constructing the contents of the video.

Line 2 and line 3 can be interpreted as the base frame for the video, and next, the codes which build up the video will be written in these frames.

However, unlike other python modules, Manim requires a special method to run the code. To run the code, the user should use the command prompt of the PC. Let's assume that the Python code above is saved as "example.py" in the Manim folder directory. After opening the command prompt In the directory, following command should be inserted for running the code :

```
D:\Manim>python -m manim example.py
```

When the user runs the code in the cmd prompt, it returns the SyntaxError that unexpected EOF (end-of-file) is detected while the PC is parsing the given code. Simply, this means that there is no data in the code. Since there is no input, there must be no output. Now the following parts. will learn how to make the inputs for the video.

3. Text and TeX

```
from manimlib.imports import *
class Video_Name(Scene) :
      def construct(self) :
            Text = TextMobject("Hello World!")
            Tex = TexMobject("\\textrm{Hello World Again!}")
```

This part will introduce two different writing functions in the Manim: Text and TeX. In Manim, Text is just text. Nothing more. It is made by a function called **TextMobject**. On the other hand, TeX is mostly used for showing mathematical notations or equations. It is made by a function called TexMobject. However, while **TextMobject** only requires string-type objects for the function, **TexMobject** requires the string-typed LaTeX code in it. This means that the users should be able to work on LaTeX code, but even though some users might not know about the LaTeX, they can use the online LaTeX equation editors such as Codecogs.

```
self.play(Write(Text))                  # Animation 1
self.play(Transform(Text, Tex))         # Animation 2
self.wait(2)                            # Animation 3
```

Since the source for the video is assigned to the variables, the users should decide in which way they will use those sources. The functions above are representatives of the text-writing animations. What should be pointed out here is that the functions are quite intuitive. Definitely, the function **Write** is for writing the text object, and **Transform** is for transforming from Text to

Tex. Like this, most of the functions in Manim are intuitive.

Then if this code was ran in the command prompt in Manim folder directory :

```
File ready at D:\Manim\media\videos\example\1440p60\Video_Name.mp4
Played 3 animations
```

As this shows, 3 functions are inserted as input, the cmd prompt plays 3 animations as output. Furthermore, the video is saved in the media folder in the name which was assigned in the **class**.

4. Graphing the Function

```
from manimlib.imports import *
class Video_Name_2(GraphScene) :
      CONFIG = {
            "x_min": -5,
            "x_max": 5,
            "y_min": -4,
            "y_max": 4,
            "function_color": WHITE
      }
```

When the user wants to animate the function graphing, first changing the **Scene** to **GraphScene** is required. However, the most distinguishable change in the code is the variable CONFIG. This CONFIG is the same as the setting for the function and the graph. For example, the five lines in the variable determine the maximum and minimum values of x and y in the axis, or what the graph's color be in the animation. There are various options in CONFIG that you can control for the graphing, including the four options above.

Then, like Text and Tex, what should be done first is setting up the variables and making the resources for the animation :

```
graph = self.get_graph(self.function, self.function_color)
label = self.get_graph_label(graph, label="y=10x+7")
self.play(ShowCreation(graph), Write(label))
```

In line 1, the function **self.get_graph** assigns the graph into the variable based on the defined function. For the next line, the users can create a label for the graph with **self.get_graph_label**. If you finished assigning the sources into the variable, you might want to create the animation

with those. Since the label is the string object, the users can use the **Write** function for displaying the label. For the graph, the function **ShowCreation** lets the code to display the animation of graph drawing. After creating those codes, defining the function is required. In the code above, the "function" is not defined. To define the "function," the users should define another function called "function" by the following code :

```
def function(self, x) :
      return (10 * x + 7)
```

According to this defined function "function," it returns the $10x + 7$, for any value of x. What this does in the **self.get_graph** is returning each y value (or $10x + 7$) for any x in "x_min" and "x_max"' you defined in the CONFIG and making a line with those points.

5. Drawing the Shapes

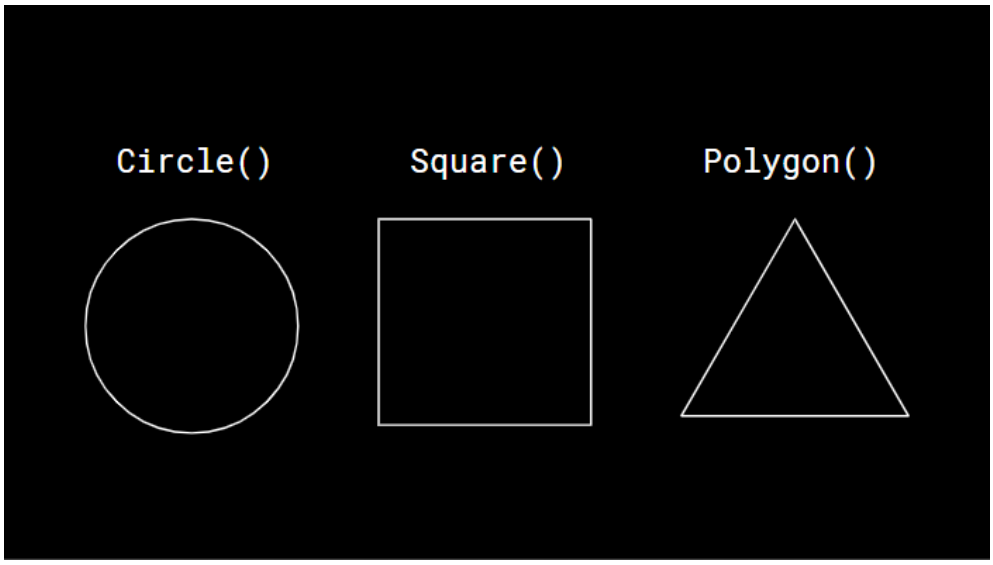

Figure 1 : The shapes made by Manim functions

The equations and graphs are not everything in mathematics. Mathematics also includes shapes or geometry in it. As the most of the Manim functions are intuitively showing its usage, the functions for shapes are also direct. The details for each shape can be written in the functions.

```
circle = Circle(color = "#42f5e3")
square = Square(fill_color = RED, fill_opacity = 1)
Polygon = (np.array(0,0,0), np.array(3,0,0), np.array(0,4,0))
rectangle = Rectangle(width = 3, height = 1)
```

```
rectangle.next_to(square, 1.2 * DOWN)
```

For the **Polygon** or **Line** function, the points should be assigned in it, and mainly the **array** function in NumPy is used for assigning the points. However, not only the points, but also the other options could be plugged into the function. For example, for shape functions, the users can set color options such as filled color, line color, and opacity. Furthermore, the width and height of rectangles or the radius of circle can also be controlled by the users. Nevertheless, in the Manim code, colors can be inserted in two ways: in color names or in hex color code. For example, the color "white" for Manim is **"#FFFFFF"** or **WHITE**. Another important function in Manim is **next_to**. The users can decide the position of the object based on the other object's location. In the function **next_to**, the distance between the other objects is determined by four directions: UP, DOWN, LEFT, RIGHT. Moreover, Manim has the fixed value of distance for those directions, but it can be decreased or increased by multiplying the constant to them.

6. Adding External Files

Although Manim provides various functions from itself for the users, the limitation for the animations still remains. Especially, when the users want to display or use specific statistics or files, Manim doesn't have ability to create it in the code. However, this can be solved since Manim can load the external files in its "\media\designs" folder directory, and those files' extensions should be **.svg** (SVG files), **.png** (images), or **.wav** (audios). For each file extension, there are folders for it. The folder "svg_images" is for SVG files, another folder "raster_images" is for PNG images, and the last, "sounds" folder is for WAV audio files. To load the external files in Manim code, each of the files must be stored in the correct folder based on its file extension.

First, for audio files, the users should make a change in the **scene_file_writer.py** in the folder directory "\manimlib\scene" like this :

```
subprocess.call(["rm", sound_file_path])        # Original Code
→ # subprocess.call(["rm", sound_file_path])    # Change 1
→ os.remove(sound_file_path)                    # Change 2
```

After changing the code, if the audio file is correctly stored in the "sounds" folder, then the user can load the audio file by the function **self.add_sound("file name")**. Make sure the name of the audio file is in the function as a string, without the file extension. Then, in the animation made by the cmd prompt will play the audio file.

Image and SVG files are different from the audio file. It does not require advance preparations (except storing files into the folder). Another difference is that images or SVG files should be

first assigned to a variable. Assume that the user wants to load the image file “Einstein.png” and the SVG file “Godel.svg” in the code, then the user might use the following code :

```
einstein = ImageMobject("Einstein")
godel = SVGMobject("Godel")
godel.next_to(einstein, DOWN)
self.add(einstein, godel)
```

This is the way to load SVG files or image files in Manim. For deciding the animation there are various options not only **self.add**, but also **FadeIn**, **DrawBorderThenFill** functions that have different animation effects.

7. Vector in Number Plane

Manim is made for creating the Mathematical animations, but the functions Manim has and the visualization can be used for other studies, especially physics. Physics is also the study which introduces absurd topics such as electromagnetism or quantum mechanics. Nonetheless, that is why Manim could be also helpful for understanding them. One of the main elements in physics is the vector, a quantity that has both direction and magnitude. Since mathematics discusses vectors also, Manim has specific functions for displaying vector fields.

```
class Vector(Scene) :
      CONFIG = { “plane_kwargs” : {
            “color” : WHITE
      }, }
      def construct(self) :
            Plane = NumberPlane(**self.plane_kwargs)
            self.add(Plane)
```

Before manipulating the vectors in the code, the number plane must be created to display those vectors efficiently. Like the 2D graphing, there is CONFIG for the number plane also. Since function **NumberPlane** itself has its default value for the number plane, setting up the CONFIG is not necessary for the number plane, but still the situations when changing the settings is required exists. While using CONFIG, the “plane_kwargs” should be added into it like above.

```
tail= [x * RIGHT + y * UP
for x in np.arange(-3,3,1)
for y in np.arange(-3,3,1)
```

```
# np.arange(min, max, frequency)
]
```

The entire code determines the location of each vector's tail, and the number of the vectors. In the default setting, the frequency of x and y of the number plane is 1 (the unit of the grid equals to 1). The first line of the code determines the distance from the origin. For the other lines, **for** loops are used, to create multiple vectors in each location, with **np.arange** function.

```
vectors = []
for p in tail:
      head = 0.5 * RIGHT + 0.5 * UP
      vector = Vector(head).shift(p)
      vectors.append(vector)
vector_field = VGroup(*vectors)
self.play(FadeIn(vector_field))
```

Finally, the creation of the vector field on the number plane is done by this code. First, an empty list should be created to assign the vectors, created by using the **for** loop again. In the loop, the variable head determines the location of each vector's head. Finally the actual vectors are created by the function **Vector** and stored into the list with **append** function.

To display the formation of the vector field, the Manim function **VGroup** will unite every vector in the list as a one object. The **VGroup** function is useful for grouping the assigned objects, it can group not only vectors, but also any other objects such asSVG files or texts. Lastly, by the function **FadeIn**, the code will generate the vector field in the animation.

8. For Further Learning

For further learning, the YouTube channel "Theorem of Beethoven" gives the detailed guides from installation of Manim and basics. Moreover, there are different communities of Manim users, especially in GitHub and Reddit communities are recommended to the beginners. The most efficient way to learn Manim is checking source codes of others' animations and analyzing it. This might give the beginners better understanding of the functions in Manim.