使用 Manim 与 Python 3.7 创建数学动画

（初稿）

原始发布人：talkingphysics

翻译：wizardor

目　录

一、前言

1、历史介绍

重要提示：来自于2018年12月的基于Python 3.7的最后的更新包含了所有的信息。

本文是基于Python 2.7 的Manim 引擎的动画实现，最新的版本是基于Python 3.7实现的。来自3Blue1Brown（https://www.3blue1brown.com/）的格兰特·桑德森（Grant Sanderson）拥有一系列令人惊叹的、涉及各种数学主题的视频。 我尤为喜欢的是，他能够帮助你对线性代数，微积分和其他几个主题的一些数学概念有更深刻的直觉认识。 同样令人敬畏的是，他为Python中的视频开发了名为manim 的动画引擎工具，这启发了我尝试为自己的一些视频制作动画。 不幸的是，有关如何使用manim （我认为它能够代表数学动画）没有多少资源可以借鉴，所以我决定着手并亲自实践来学会使用它。 manim 的代码可在Github上免费获得。 我注意到许多人表示有兴趣来学习使用manim ，因此我想通过一系列文章来分享我的学习之旅。以下内容，都是假定你对Python 面向对象的编程已有了足够的了解。 另请注意，这是基于2018年5月9日的manim 提交版本--将来的版本可能会有所不同。在学习之前，我想指出，这绝不是manim 工作原理的权威手册。这只是我学习如何使用manim 的旅程的叙述。 我的理解可能是错误的，我的执行可能不是最佳的。 牢记这些警告，让我们开始学习如何创建自己的精美数学动画。

2、新版本修订

我以前写过一系列博客文章，详细介绍了如何使用manim ， manim 是由3Blue1Brown的Grant Sanderson所创建的数学动画程序包。 自从我写完这些文章以来，manim 已经进行了许多更改，包括切换到Python 3.7， 我仔细阅读了截至2018年12月的Manim 版

本信息的更新。在Manim 代码没有更改的情况下，很多信息重复了我以前的文章，我前期系列文章中的主要更改也与Manim 的更改有关，主要是在处理3D场景方面。 请注意，未来的版本可能会突破其中一些命令，但是找出问题原因所在，将是学习manim 内部如何工作的最佳方法。

本文初译完成于 2019 年 10 月 5 日，限于译者水平，错误在所难免，希望读者指正。

反馈邮箱：wizardor@gmail.com

为便于读者学习，本教程视频打包下载地址如下：

地址 1、地址 2

二、内容（中文）

1.0 安装 manim 引擎

Brian Howell 已经把所有相关安装manim 的资料都已经打包到了如下链接：

http://bhowell4.com/manic-install-tutorial-for-mac/.

确保所有工作都有条不紊的一个有益提示是，请通过使用虚拟环境来进行安装。如果你安装manim 遇到了问题，请通过如下论坛链接（https://github.com/3b1b/manim /issues）来寻求进一步的帮助。

Github上的readme 文件也会对你安装manim 有所帮助。

为了确保你安装 manim 后能够正常工作，通过

python -m manim example_scenes.py -pl

该命令可以检验你是否成功安装、并是否能够正确运行了示例代码，如果你运行出错了，请到 github 上的链接

https://github.com/3b1b/manim /issues?utf8=%E2%9C%93&q=is%3Aissue

寻求帮助，也许其他人的常见问题列表中已经包含了你所希望解决的问题。

2.0 创建首个场景

将下面的代码

https://github.com/zimmermant/manim_tutorial/blob/master/manim_tutorial_P37.py

复制并粘贴到新的文本文件中，将其另存为 manim _tutorial_P37.py 文件并置于 manim 的根目录中。 .py 扩展名会告诉你的操作系统这是一个 Python 文件。

打开命令控制台窗口，切换到 manim 的根目录，然后输入：

python -m manim pymanim _tutorial_P37.py Shapes -pl

现在通过输入 python 命令来调用 Python 语言解释器。 传递给 Python 语言的第一个参数 extract_scence.py 为运行脚本并创建视频文件的 manim 代码的一部分。 创建视频总需要调用此参数。 第二个参数 manim _tutorial_1.py 是包含控制脚本的文件（即模块）的名称。 第三个参数 Shapes 是文件中定义的类的名称（即场景名称），该类描述了有关如何构建场景的说明。 最后一个参数 -pl 告诉 extract_scene 脚本在完成动画后通过播放动画来预览（p）该动画，并以低质量（l）来渲染动画，这会加快创建动画的时间。有关能够传递给 extract_scene 的其他参数的说明，具体可以参见 README.md 文件。

通过输入 python -m manim --help 命令可以给你更为完整的命令参数列表。

```
from big_ol_pile_of_manim _imports import *


class Shapes(Scene):
```

```
#A few simple shapes
def construct(self):
        circle = Circle()
        square = Square()
        line=Line(np.array([3,0,0]),np.array([5,0,0]))
        triangle=Polygon(np.array([0,0,0]),np.array([1,1,0]),np.array([1,-1,0]))

        self.add(line)
        self.play(ShowCreation(circle))
        self.play(FadeOut(circle))
        self.play(GrowFromCenter(square))
        self.play(Transform(square,triangle))
```

如果一切正常，你在输出终端中应该可以看到如下信息：

该视频的输出结果如下：

【视频:】Shapes - manim Tutorial Example 1
https://youtu.be/V5iF-7RAAuc

各种 manim 基础核心模块都包含在 big_ol_pile_of_manim _imports.py 中，因此 import 该模块将为你提供 manim 的所有基本功能。 注意这并不包括 manim 的所

有模块，但包含了核心模块。 你需要抽出时间来研究一下这些模块，了解它们是如何组织在一起的。 通过尝试弄清楚 Python 是如何工作的，我受益匪浅。顺便说一句，我发现使用 https://github.com/3b1b/manim 页面上的搜索框，对于查找不同的类，并弄清楚它们接受什么参数以及它们如何运作是非常有益的。

2.1 场景及动画

场景脚本可以告诉 manim 在屏幕上如何放置和设置对象并生成动画。 我看到每个3blue1brown视频都是作为单独场景创建的，这些场景再使用视频编辑软件进行拼接。你必须将每个场景定义一个单独的类，该类是Scene的子类。此类必须具有Construct()方法，以及创建对象，将其添加到屏幕以及对其进行动画处理所需的任何其他代码。当通过 extract_scene.py 运行动画时，construct()方法本质上是该类中的主要方法。 它类似于初始化方法__init__；创建类的实例时，将自动调用该方法。在此方法中，你应该定义所有对象、控制对象所需的任何代码以及将对象置于屏幕上并对其进行动画控制处理的代码。

在第一个场景中，我们创建了一个圆、一个正方形、一条直线和一个三角形。注意，使用 numpy 数组 np.array() 可以指定坐标。你可以传递一个类似（3,0,0）的三元组，该三元组有时会起作用，但是某些转换方法依赖于 坐标是 numpy 数组。

Scene()类中一个最重要的方法是 play()方法。 play()负责处理你 manim 中的各种动画的执行过程。我最喜欢的动画是“变形”，它出色地完成了将一个mobject变形为另一个对象的工作。 该场景展示了一个正方形如何渐变为一个三角形，而你也可以使用该变换将两个对象变形在一起。 要使对象显示在屏幕上而没有任何动画，可以使用add()放置它

们。该行已添加并显示在第一帧中，而其他对象则会淡入或增长。转换的命名已非常直接明了，这是显而易见的。

更多尝试：

–更改add()和play()命令的顺序；

–尝试在其他形状上使用Transform()方法；

–分析位于/manim /mobject/ 文件夹中的geometry.py 文件中定义的形状。

3.0 更多的几何形状

你可以使用 manim 创建几乎任何几何形状，包括圆形，正方形，矩形，椭圆形，直线和箭头。 让我们看一下如何绘制其中的一些形状。

你可以从以下链接下载完整的教程代码：https://github.com/zimmermant/manim_tutorial/blob/master/manim _tutorial_P37.py

下载完成后，你需要定位到 manim 目录的根目录处，然后在控制台终端中输入如下命令运行场景：

```
python -m manim  manim _tutorial_P37.py MoreShapes -pl
```

```
class MoreShapes(Scene):
    def construct(self):
        circle = Circle(color=PURPLE_A)
        square = Square(fill_color=GOLD_B, fill_opacity=1, color=GOLD_A)
        square.move_to(UP+LEFT)
        circle.surround(square)
        rectangle = Rectangle(height=2, width=3)
        ellipse=Ellipse(width=3, height=1, color=RED)
        ellipse.shift(2*DOWN+2*RIGHT)
        pointer = CurvedArrow(2*RIGHT,5*RIGHT,color=MAROON_C)
        arrow = Arrow(LEFT,UP)
```

```
        arrow.next_to(circle,DOWN+LEFT)
        rectangle.next_to(arrow,DOWN+LEFT)
        ring=Annulus(inner_radius=.5, outer_radius=1, color=BLUE)
        ring.next_to(ellipse, RIGHT)

        self.add(pointer)
        self.play(FadeIn(square))
        self.play(Rotating(square),FadeIn(circle))
        self.play(GrowArrow(arrow))
        self.play(GrowFromCenter(rectangle), GrowFromCenter(ellipse),
GrowFromCenter(ring))
```

【视频:】MoreShapes - manim Tutorial Series Example 3
https://youtu.be/QGJQerSELDo

你会注意到我们有一些新形状，并使用了几个新的命令。以前，我们看到过Circle()，Square()，Line()和Polygon()类。现在，我们添加了Rectangle()，Ellipse()，Annulus()，Arrow()和CurvedArrow()。除直线和箭头外，所有形状均是从屏幕中心原点（0,0,0）处创建的。对于线条和箭头，创建时你还需要指定两端的位置。

首先，我们使用关键字设定参数color =为正方形指定了颜色。大多数形状是VMobject的子类，VMobject代表矢量化的数学对象。 VMobject本身是数学对象类Mobject的子类。确定可以传递给类的关键字参数的最佳方法是查看VMobject和Mobject类的允许参数列表。一些可能的关键字包括半径，高度，宽度，颜色，填充色和填充透明度。对于

Annulus()类，我们为关键字参数设置了inner_radius和outside_radius。

你可以在constant.py文件中的COLOR_MAP词典中找到已命名颜色的列表。命名的颜色是COLOR_MAP字典的键值，该字典使用了十六进制颜色代码表示法。你也可以使用十六进制颜色代码选择器来创建自己的颜色，并将其键值添加到COLOR_MAP中去。

3.1 方向向量

constants.py 文件包含了其他有用的定义，例如可用于将对象放置在场景中的方向向量。例如，UP是一个numpy数组（0,1,0），对应于1个距离单位。为了遵守manim 中使用的命名约定，我决定将距离单位称为MUnit或数学单位（这是我自己的术语，而不是manim术语）。因此，默认屏幕高度为8 MUnit（在constants.py中定义）。默认屏幕宽度为14.2 MUnit。

如果考虑x，y和z坐标，则UP是沿y轴正方向指向的向量。 RIGHT是数组（1,0,0）或沿正x轴指向的向量。其他方向向量是LEFT，DOWN，IN和OUT。每个向量的长度为1 MUnit。创建对象实例后，可以使用.move_to()方法将对象移动到屏幕上的特定位置。请注意，方向向量可以加在一起（例如UP + LEFT），也可以乘以标量来放大（例如2 * RIGHT）。换句话说，方向矢量的行为就像你期望数学矢量的行为一样。如果要指定自己的向量，则它们将必须是具有三个分量的numpy数组。每个屏幕侧的中心边缘也由矢量TOP，BOTTOM，LEFT_SIDE和RIGHT_SIDE进行了定义。

向量的整体比例（像素和MUnit之间的关系）由constants.py中定义的FRAME_HEIGHT变量设置。 缺省值为8。这意味着你必须将对象以8 * UP的比例关系从屏幕底部移动到屏幕

顶部。 目前，除了在constants.py中进行定义更改外，我没有看到有其他的更改方法。

还可以使用next_to()方法将Mobject相对于另一个对象进行定位。 命令arrow.next_to（circle，DOWN + LEFT）将箭头向下移动一个MUnit，将箭头向左移动一个MUnit。最后，箭头将位于圆的左下一个MUnit单位的对角位置。

Circle()类也具有Surround()的方法，该方法使你可以创建一个完全包围另一个mobject的圆。 圆的大小将取决于所包围对象的最大尺寸。

3.2 制作简单动画

如前所述，.add()方法在场景开始时将mobject放置在屏幕上。 .play()方法可用于动画场景中的对象控制。

动画的名称（例如FadeIn或GrowFromCenter）不言自喻。你应该注意的是，动画将按照列出的顺序来依序播放，并且如果要同时生成多个动画，则应将所有这些动画包括在单个.play()命令的参数中，并以逗号分隔。 在本系列教程的后续文章中，我将向你展示如何使用动画列表来同时播放多个动画。

更多尝试：

–使用Polygon()类创建其他形状；

–尝试在屏幕上不同位置来放置多个对象；

–看看/animation/transforms.py 中可用的不同类型的转换

4.0 创建文本对象

Mobject有一个特殊的子类，称为TextMobject（文本数学对象）。 将以下类添加到你的tex_mobject.py 文件中，然后在控制台命令行中输入

python -m manim manim _tutorial_P37.py AddingText -pl

请注意，文本看起来真的很模糊，因为我们是以低质量渲染动画配置（pl）来加快渲染处理速度的。 使用这样的小文件，你可以在不花费太多时间的情况下以全分辨率来呈现它。为此，将-pl替换为-p（保留低分辨率标记）。

```
class AddingText(Scene):
    #Adding text on the screen
    def construct(self):
        my_first_text=TextMobject("Writing with manim  is fun")
        second_line=TextMobject("and easy to do!")
        second_line.next_to(my_first_text,DOWN)
        third_line=TextMobject("for me and you!")
        third_line.next_to(my_first_text,DOWN)

        self.add(my_first_text, second_line)
        self.wait(2)
        self.play(Transform(second_line,third_line))
        self.wait(2)
        second_line.shift(3*DOWN)
```

```
        self.play(ApplyMethod(my_first_text.shift,3*UP))
```

【视频：】AddingText: manim Tutorial Series Example 4.1

https://youtu.be/XyA85M1YaWo

要创建textmobject，必须将有效的字符串作为参数来传递给它。文本渲染基于Latex，因此你可以使用许多Latex的排版功能。我将在以后的文章中介绍。作为mobject的子类，任何方法（例如move_to，shift和next_to）都可以与 textmobjects 一起来使用。

wait()方法将设置阻止场景的下一个命令执行所需的秒数。默认时间为1秒，因此调用self.wait()将等待1秒，然后再执行脚本中的下一个命令。

你应该注意到了，在动画过程中，第二行文本跳下来，而第一行文本则轻轻地向上滑动。这与以下事实有关：我们将shift()方法应用于第二行文本，但我们创建了第一行文本移位的动画。为mobject()方法设置动画时，需要ApplyMethod()动画。请注意，ApplyMethod()的参数是该方法的指针（在本例中为my_first_text.shift，不带任何括号），后跟逗号，然后是通常包含作为shift()方法的参数的内容。换言之，ApplyMethod（my_first_text.shift，3 * UP）将创建一个动画，将my_first_text向上移动三个MUnit。

4.1 改变文本对象

尝试运行 AddMoreText 场景。

```
class AddingMoreText(Scene):
```

```
        #Playing around with text properties
        def construct(self):
                quote = TextMobject("Imagination is more important than knowledge")
                quote.set_color(RED)
                quote.to_edge(UP)
                quote2 = TextMobject("A person who never made a mistake never tried
anything new")
                quote2.set_color(YELLOW)
                author=TextMobject("-Albert Einstein")
                author.scale(0.75)
                author.next_to(quote.get_corner(DOWN+RIGHT),DOWN)

                self.add(quote)
                self.add(author)
                self.wait(2)
                self.play(Transform(quote,quote2),

ApplyMethod(author.move_to,quote2.get_corner(DOWN+RIGHT)+DOWN+2*LEFT
))

                self.play(ApplyMethod(author.scale,1.5))
                author.match_color(quote2)
```

```
        self.play(FadeOut(quote))
```

【视频：】AddingMoreText: manim Tutorial Series Example 4.2

https://youtu.be/gWuOD4mHBFs

在这里，我们将学会如何使用set_color()更改文本的颜色。这将使用与绘制几何形状时相关的相同颜色，其中许多颜色在constant.py中的COLOR_MAP词典中进行了定义。除了设置颜色外，还可以将颜色与另一个对象匹配。在以上代码的倒数第二行中，我们使用match_color()更改文本对象“作者”的颜色以匹配quote2。

你也可以使用scale()来更改文本的大小。该方法将mobject按给定的数值因子放大。因此，scale（2）将使mobject的大小增加一倍，而scale（0.3）将mobject缩小至其大小的30%。似乎你不能在动画中直接使用scale()，如ApplyMethod（author.scale，2），这会不起作用。是否有一种更好的方法，如果我想放大或缩小动画，我该怎么做呢？那就是创建第二个textmobject对象。

这将会是我想要的最终大小的文本对象，使用Transform()可以从一个缩放尺寸平滑过渡到另一个缩放尺寸。希望以后我能找到更好的方法。

你可以对齐对象，通过让对象是否UP，DOWN，LEFT或RIGHT的操作来使mobject与屏幕边缘以to_edge()方式进行中心对齐。 你也可以使用to_corner()，在这种情况下，你需要组合两个方向（例如UP + LEFT）以指示拐角处理。

每个mobject都有一个边界框，你可以使用get_corner()并指定方向来获取该边界框的角

坐标。 因此，get_corner（DOWN + LEFT）将返回mobject左下角的位置。 在我们的示例中，我们找到文本对象“报价”的右下角，然后将文本对象“作者”从该位置下移一个单位。 稍后，我们将文本对象“作者”移至quote2的左下方。

需要注意的重要一点是，Transform()动画仍将mobject引用保留在屏幕上，但只是将其显示文本和属性更改为quote2。 这就是为什么FadeOut()引用quote而不引用quote2的原因。 但是，quote2的角是它的原始位置，这就是为什么我们必须找到quote2的角以将作者移到正确的位置。 请记住，当你使用Tranform时，涉及的对象的属性可能不是你所期待的那样。

另一个有用的信息是，scale()方法更改了当前对象的大小。 换句话说，使用scale（.5）后跟scale（.25）会导致对象是原始大小的$ 0.5 * 0.25 = 0.125 $倍

更多尝试：

–尝试使用to_corner()方法

–在constants.py文件中找到 COLOR_MAP 并更改文本的颜色

4.2 旋转及高亮文本对象

下面的代码将演示如何旋转文本对象并给文本对象加上一些小挑战。继续运行如下命令：

```
python -m manim    manim _tutorial_P37.py RotateAndHighlight -p
```

```
class RotateAndHighlight(Scene):
```

```
        #Rotation of text and highlighting with surrounding geometries
        def construct(self):
            square=Square(side_length=5,fill_color=YELLOW, fill_opacity=1)
            label=TextMobject("Text at an angle")
            label.bg=BackgroundRectangle(label,fill_opacity=1)
            label_group=VGroup(label.bg,label)    #Order matters
            label_group.rotate(TAU/8)
            label2=TextMobject("Boxed text",color=BLACK)
            label2.bg=SurroundingRectangle(label2,color=BLUE,fill_color=RED,
fill_opacity=.5)
            label2_group=VGroup(label2,label2.bg)
            label2_group.next_to(label_group,DOWN)
            label3=TextMobject("Rainbow")
            label3.scale(2)
            label3.set_color_by_gradient(RED, ORANGE, YELLOW, GREEN, BLUE,
PURPLE)
            label3.to_edge(DOWN)

            self.add(square)
            self.play(FadeIn(label_group))
            self.play(FadeIn(label2_group))
            self.play(FadeIn(label3))
```

【视频：】RotateAndHighlight: manim Series:Example 4.3

https://youtu.be/saM2hTsMyQ8

我们在背景中添加了一个正方形，以显示BackgroundRectangle的作用。 请注意，填充颜色的不透明度默认为零，因此，如果你未定义fill_opacity，则只能看到正方形的边缘。要创建背景矩形，你需要指定要应用此方法的textmobject以及不透明度。 你不能将颜色背景更改为黑色。

VGroup类允许你将多个mobject组合为一个矢量化的数学对象。 这使你可以将任何VMobject方法应用于分成组的所有元素。 将原始对象成组之后，你仍然可以更改它们的属性。 换句话说，原始mobject不会被破坏，vmobject只是mobject的更高层次的分组。 通过将文本和背景矩形分组，我们可以使用rotate()一起更改两个对象的方向。 请注意，TAU等于$ 2 \ pi $（请参见Tau宣言，其中一些观点比较有趣）。

next_to()方法可以看作是相对于其他对象的偏移，因此label2_group.next_to（label_group，DOWN）将label2_group从label1_group将下移一个单位（请记住，距离单位是由FRAME_HEIGHT变量在constants.py 常量文件中设置的）。你可以使用set_color_by_gradient()来创建颜色渐变，可以 将任意数量的颜色（用逗号分隔）传递给该方法。

更多尝试：

–尝试更改正方形与背景矩形的填充的透明度

–尝试将背景矩形与文本分开旋转

–更改label2的颜色以查看其如何影响文本的可读性

–更改“彩虹”的颜色

–将“彩虹”文本放置在屏幕的另一边缘。

5.0 数学方程式

如果一个数学动画工具无法包含美观的数学方程式，那么该数学动画工具就不会有太大用处。 我所知道的最好的方程式排版方法是使用LaTeX()功能，manim 引擎会使用该工具。 如果你你想了解有关使用LaTeX进行排版的更多信息，我会推荐ShareLaTeX上的教程作为基本介绍，但是你不需要了解有关LaTeX的使用manim 的太多知识。 你可以在此处找到常用符号的列表，这些几乎是你需要了解的所有符号了。

在 manim 下运行如下的场景：

```
class BasicEquations(Scene):
    #A short script showing how to use Latex commands
    def construct(self):
        eq1=TextMobject("$\\vec{X}_0 \\cdot \\vec{Y}_1 = 3$")
        eq1.shift(2*UP)
        eq2=TexMobject(r"\vec{F}_{net} = \sum_i \vec{F}_i")
        eq2.shift(2*DOWN)

        self.play(Write(eq1))
        self.play(Write(eq2))
```

【视频：】Basic Equations: manim Series: 5.1

https://youtu.be/r_4RUwYAAdk

在LaTeX中，通常使用美元符号$$围起来的方程式来表示方程式，在这里也可以使用。 主要区别在于，由于manim 解析文本的方式，所有LaTeX命令之前都必须包含一个额外的反斜杠用以转义。 例如，可以在LaTeX中创建希腊字母，方法是输入字母名称后加反斜杠； 小写字母alpha为$ \ alpha $，角度theta为$ \ theta $。 但是，在manim 中，需要使用双反斜杠进行转义，因此将是$ \\ alpha $并将被写为$ \\ theta $。

你可以将矢量箭头放在变量上，例如使用\ vec {A}，或者，由于manim 需要双反斜杠，因此可以将\\ vec {A}放置在变量上。 无论你将什么放在花括号内，都将在屏幕上显示箭头上方的箭头。 下标由下划线表示，因此应写为$ \ vec {X} _0 $。 如果下标包含多个字符，则可以将下标括在方括号中。 因此，习惯上是$ \\ vec {F} _ {net} $。

必须始终包含美元符号会很麻烦，因此TexMobject类假定所有字符串都是数学对象。TEX()是基于LaTeX的排版语言，因此我假设TexMobject是为TEX命名的。TextMobject()和TexMobject之间的主要区别在于，除非你使用美元符号指定方程式，否则文本数学对象会假定所有内容都是纯文本，除非使用\\ text {}指定文本是Tex数学对象，否则引擎会假定所有内容都是方程式，除非你指定的内容是纯文本。

创建任何类型的mobject时，默认位置似乎都是屏幕的中心。创建后，你可以使用shift()或move_to更改mobject的屏幕位置。在上面的示例中，我将方程式向上移动了两个MUnit或向下移动了两个MUnit（请记住，MUnit或数学单位是我在Manim 中称呼的长度度量单位）。由于屏幕高度默认设置为8 MUnit，因此2 MUnit的偏移量大约相当于屏幕高度的四分之一。

Write()方法是ShowCreation的子类，它采用TextMobject或TexMobject并以动画方式在屏幕上写入文本。你还可以将字符串传递给Write()方法，它将为你创建TextMobject对象。 Write()方法必须位于play()控制域内，以便对其进行动画处理。

5.1 给等式着色

```
class ColoringEquations(Scene):
    #Grouping and coloring parts of equations
    def construct(self):
        line1=TexMobject(r"\text{The vector } \vec{F}_{net} \text{ is the net }",r"\text{force }",r"\text{on object of mass }")
        line1.set_color_by_tex("force", BLUE)
        line2=TexMobject("m", "\\text{ and acceleration }", "\\vec{a}", ".  ")
        line2.set_color_by_tex_to_color_map({
            "m": YELLOW,
            "{a}": RED
        })
        sentence=VGroup(line1,line2)
        sentence.arrange_submobjects(DOWN, buff=MED_LARGE_BUFF)
        self.play(Write(sentence))
```

【视频：】Coloring Equations: manim Series 5.2

https://youtu.be/UDs9BGFBvoQ

在此示例中，我们将文本分为纯文本和方程式块，这让我们可以使用set_color_by_tex()或set_color_by_tex_to_color_map()对方程进行着色。 set_color_by_tex()方法以你想要的颜色作为参数对单个字符串进行颜色设定。看起来，你只需要指定字符串进行部分匹配，即可让整个字符串都会变成设定的颜色。例如，如果我们输入line1.set_color_by_tex（“F”，BLUE），则大写F出现的唯一位置是在“F_”变量中，因此整个“F_”字符串变量都会被着成蓝色。如果相反，我们尝试使用line1.set_color_by_tex（“net”，BLUE）进行精准匹配，因为“net”在line1的两个位置中出现，因此两个位置包含“net”的部分都将会被着成蓝色。如果要更改texmobjects列表中多个元素的颜色，可以使用set_color_by_tex_to_color_map()和字典。字典的键应该是我们想要着色的文本（或字符串的唯一部分），值应该是所需的颜色。

请注意，由于我们使用的是texmobject而不是textmobject，因此必须将纯文本包含在LaTeX命令\\ text {}中。如果你不这样做，假定文本是方程式的一部分，则文本的字体和间距看起来会很有趣。因此，“the net force on object of mas”看起来像这样，在等式环境下是无法识别单词之间的空格、不同的字体、以及字母与普通文本的不同间距的。

通过使用VGroup()将两行组合在一起，我们可以使用 arrange_submobjects()方法来分隔这两行。第一个参数是希望对象间隔开的方向，第二个参数buff（我假设）是mobject之间的缓冲区距离。在constants.py文件中定义了多个默认缓冲区距离，但你也可以指定一个数值来重新定义。最小的默认缓冲区距离为SMALL_BUFF = 0.1，最大的为LARGE_BUFF = 1。尽管我没有深入研究代码，但我认为缓冲区距离的工作方式是作为主要方向向量（例如UP，DOWN，LEFT，RIGHT）的乘法因子，因此指定SMALL_BUFF和LEFT相当于$ 0.1 *（-1,0,0）=（-0.1,0,0）$。

更多尝试：

–尝试使用

https://www.sharelatex.com/learn/List_of_Greek_letters_and_math_symbols 中的符号创建自己的方程式

–尝试更改方程式不同部分的颜色

6.0 对齐文本对象与使用花括号

本篇章将展示如何使用花括号以可视化的方式将方程式或文本分组在一起，以及如何对齐文本元素。 我们将首先编写一个程序来对齐两个方程式的元素，但是有点拙作， 这不是以最优雅的方法来完成此任务。 看完第一个版本后，我们将以更简洁的方式重写代码，从而使所有内容更加统一。

你可以从如下地址下载教程文件： https://github.com/zimmermant/manim_tutorial/blob/master/manim _tutorial_P37.py

```
class UsingBraces(Scene):
    #Using braces to group text together
    def construct(self):
        eq1A = TextMobject("4x + 3y")
        eq1B = TextMobject("=")
        eq1C = TextMobject("0")
        eq2A = TextMobject("5x -2y")
        eq2B = TextMobject("=")
        eq2C = TextMobject("3")
        eq1B.next_to(eq1A,RIGHT)
        eq1C.next_to(eq1B,RIGHT)
        eq2A.shift(DOWN)
        eq2B.shift(DOWN)
        eq2C.shift(DOWN)
```

```
eq2A.align_to(eq1A,LEFT)

eq2B.align_to(eq1B,LEFT)

eq2C.align_to(eq1C,LEFT)


eq_group=VGroup(eq1A,eq2A)

braces=Brace(eq_group,LEFT)

eq_text = braces.get_text("A pair of equations")


self.add(eq1A, eq1B, eq1C)

self.add(eq2A, eq2B, eq2C)

self.play(GrowFromCenter(braces),Write(eq_text))
```

【视频：】Using Braces: manim Series 6.1

https://youtu.be/By0Zjj35tdI

为了在屏幕上对齐部分方程式，我们使用next_to()和align_to()方法。在本示例中，我们将等式分解为较小的部分，然后使用next_to()将每个等式的子部分彼此相邻放置，最后使用align_to()将等式的每个部分进行左对齐。 你还可以使用UP，DOWN和RIGHT对齐mobject的不同边界。我们还添加了括号，以显示如何直观地分组一组方程组。为了使用花括号，我们必须使用VGroup()方法来组合这些方程式。当我们实例化括号时，第一个参数是组，第二个参数是括号相对于分组的位置。 你可以使用get_text()在大括号旁边设置文本。此方法不会在屏幕上绘制文本，它仅用于设置文本相对于大括号的位置，因此，你仍需要将文本添加到屏幕上。

```
class UsingBracesConcise(Scene):
    #A more concise block of code with all columns aligned
    def construct(self):
        eq1_text=["4","x","+","3","y","=","0"]
        eq2_text=["5","x","-","2","y","=","3"]
        eq1_mob=TexMobject(*eq1_text)
        eq2_mob=TexMobject(*eq2_text)
        eq1_mob.set_color_by_tex_to_color_map({
            "x":RED_B,
            "y":GREEN_C
            })
        eq2_mob.set_color_by_tex_to_color_map({
            "x":RED_B,
            "y":GREEN_C
            })
        for i,item in enumerate(eq2_mob):
            item.align_to(eq1_mob[i],LEFT)
        eq1=VGroup(*eq1_mob)
        eq2=VGroup(*eq2_mob)
        eq2.shift(DOWN)
        eq_group=VGroup(eq1,eq2)
        braces=Brace(eq_group,LEFT)
```

```
eq_text = braces.get_text("A pair of equations")


self.play(Write(eq1),Write(eq2))
self.play(GrowFromCenter(braces),Write(eq_text))
```

【视频：】Using Braces Concise: manim Series 6.2
https://youtu.be/FlzrUqyne8A

这是比之前代码的更简洁版本。每个方程式以列表形式给出，方程式的每个部分都作为独立的字符串，以便这样可以更好控制两个方程式各部分的垂直对齐关系。在for循环内，我们使用align_to()将eq1和eq2中元素的左边界进行对齐。

请注意，在创建texmobjects时，我们传递了列表的变量名称，并在其前面带有星号 eq1_mob = TexMobject（* eq1_text）。星号是一个Python命令，用于解压缩列表并将参数视为以逗号分隔的列表。 因此，eq1_mob = TexMobject（* eq1_text）与 eq1_mob = TexMobject（“4”， “x”， “+”， “3”， “y”， “=”， “0”）效果相同。

更多尝试：

–在屏幕上排列方程式

–在方程式周围添加一些形状

7.0 绘图函数

绘制图形的最简单方法是基于GraphScene（）场景类。 场景创建了一组坐标轴，并具有创建图形的各类方法。一开始让我困惑的是，坐标轴属于你自己的场景类，因此你将需要使用self来访问与坐标轴相关的方法，这在初始阶段就给我带来了一些问题。

我们首先研究它是如何创建坐标轴和图形的，我们得看一下CONFIG {}字典文件，在manim 中该字典经常用于初始化许多类变量。

```
class PlotFunctions(GraphScene):
    CONFIG = {
        "x_min" : -10,
        "x_max" : 10.3,
        "y_min" : -1.5,
        "y_max" : 1.5,
        "graph_origin" : ORIGIN ,
        "function_color" : RED ,
        "axes_color" : GREEN,
        "x_labeled_nums" :range(-10,12,2),
    }
    def construct(self):
        self.setup_axes(animate=True)
        func_graph=self.get_graph(self.func_to_graph,self.function_color)
        func_graph2=self.get_graph(self.func_to_graph2)
```

```
        vert_line = self.get_vertical_line_to_graph(TAU,func_graph,color=YELLOW)
        graph_lab = self.get_graph_label(func_graph, label = "\\cos(x)")
        graph_lab2=self.get_graph_label(func_graph2,label = "\\sin(x)", x_val=-10,
direction=UP/2)
        two_pi = TexMobject("x = 2 \\pi")
        label_coord = self.input_to_graph_point(TAU,func_graph)
        two_pi.next_to(label_coord,RIGHT+UP)

        self.play(ShowCreation(func_graph),ShowCreation(func_graph2))
        self.play(ShowCreation(vert_line), ShowCreation(graph_lab),
ShowCreation(graph_lab2),ShowCreation(two_pi))

    def func_to_graph(self,x):
        return np.cos(x)

    def func_to_graph2(self,x):
        return np.sin(x)
```

【视频：】PlotFunctions - manim series 7.1

https://youtu.be/zP5HAKXHu3I

在construct方法下，第一行是self.setup_axes（），它将在屏幕上创建一组坐标轴。 除了创建的内容是否为动画外，所有其他坐标轴变量均使用CONFIG {}中的变量进行配置，

我稍后将予以解释。 GraphScene（）类的默认值配置（位于graph_scene.py文件中）如下所示：

```
CONFIG = {
    "x_min": -1,
    "x_max": 10,
    "x_axis_width": 9,
    "x_tick_frequency": 1,
    "x_leftmost_tick": None, # Change if different from x_min
    "x_labeled_nums": None,
    "x_axis_label": "$x$",
    "y_min": -1,
    "y_max": 10,
    "y_axis_height": 6,
    "y_tick_frequency": 1,
    "y_bottom_tick": None, # Change if different from y_min
    "y_labeled_nums": None,
    "y_axis_label": "$y$",
    "axes_color": GREY,
    "graph_origin": 2.5 * DOWN + 4 * LEFT,
    "exclude_zero_label": True,
    "num_graph_anchor_points": 25,
```

```
    "default_graph_colors": [BLUE, GREEN, YELLOW],
    "default_derivative_color": GREEN,
    "default_input_color": YELLOW,
    "default_riemann_start_color": BLUE,
    "default_riemann_end_color": GREEN,
    "area_opacity": 0.8,
    "num_rects": 50,
}
```

在我们的示例中，改变了x_min，x_max，y_min，y_max，graph_origin，axes_color和x_labeled_num的数值。在我们calss中分配的值优先于其在parent class设置的值，已在parent class中定义的值是不会自动改变的。 x_labeled_num标签属性代表其沿x轴的数值刻度。 我们已经使用range（-10,12,2）生成了一个以-10为步长从2到+10的值的列表。我在y轴上注意到一个问题，如果你沿任一轴设置的最小值不是0.5的整数倍，那么将会导致沿该轴的刻度线与零点不对称（例如，尝试y_min = -1.2）。我不确定这是怎么回事，但是如果你坚持使用0.5的整数倍，那就不会有此问题。

在设置好轴之后，就可以使用self.get_graph（）来绘制函数图了。 get_graph（）的参数必须是一个指向函数的指针，而不是对函数本身的调用。 换而言之，因为我想使用func_to_graph（）函数调用功能，所以我应该使用self.get_graph（func_to_graph），注意，在func_to_graph之后是没有任何括号的。

除了定义独立的绘图功能外，我们还可以使用lambda函数来实现。 例如，如果定义

self.func = lambda x：np.cos（x），然后使用self.get_graph（self.func），我将会得到相同的结果。

使用get_graph（）函数需要显式的传递参数，而不是使用CONFIG {}配置。 除绘图功能外，可能传递的参数是color，x_min和x_max。 如果你未指定颜色，则GraphScene（）函数会循环以蓝色、蓝色和黄色进行着色。 由于我没有为第二张绘图指定特定颜色，所以系统自动为其分配了第一种颜色：BLUE。

有一种简便的绘图方法，可以通过get_vertical_line_to_graph（）函数来进行从x轴到目标图形的垂直线绘制。 我喜欢具有足够描述性方法的命名约定，以使你可以直观地看到每种方法的功能。 做的不错！ get_vertical_line_to_graph（）的参数：是你要在其中绘制线条的x值，以及该线条要绘制到的特定图形。 请注意，get_vertical_line_to_graph（）是GraphScene的方法，而不是图形或轴，因此需要采用self.get_vertical_line_to_graph（）的方式进行调用。

你可以使用get_graph_label（）标记函数来设置与绘图关联的文本。 这类似于Braces（）类的get_text（）方法，因为它在特定位置创建了一个texmobject对象，但没有在屏幕上绘制它。 你需要通过添加或播放动作才能在屏幕上显示该标签。 get_graph_label（）的参数包括你要为其添加标签的特定目标图形，以及为其设定的标签文本。 如果你未指定x值和/或方向，则标签将放置在图形的末尾。方向设定相对于要放置标签目标对象的x_value的位置。

还有几个与GraphScene（）相关的其他方法值得研究，我发现input_to_graph_point

（）函数尤为有帮助。 其通过在图形上指定x值，此方法将返回该图形点所在的屏幕上的坐标，如果你想在图形上特定点的位置放置一些文本或其他对象，这将会很方便。

7.1 CONFIG{}字典

每当创建场景或mobject时，就会调用一个称为digest_config（）的方法。 该方法从你定义的类开始，查找一个名为self.CONFIG的字典，并编译字典中所有条目的列表。 然后转到父类在其中查找self.CONFIG并添加这些条目。 如果遇到已经找到的键，则它将忽略父类的值。 digest_config（）使用Container（）继续上溯到顶级父类，然后，根据键和值为该词典中的每个条目分配一个类变量。 因此，字典条目“ x_min”： -1变为self.x_min = -1，依此类推。 每个字典条目都变为一个类变量，该类变量可由类中的方法访问。 了解类的所有CONFIG {}条目对于充分利用manim 是至关重要的。例如，GraphScene（）具有以下CONFIG {}条目：

```
class GraphScene(Scene):
    CONFIG = {
        "x_min": -1,
        "x_max": 10,
        "x_axis_width": 9,
        "x_tick_frequency": 1,
        "x_leftmost_tick": None, # Change if different from x_min
        "x_labeled_nums": None,
        "x_axis_label": "$x$",
```

```
        "y_min": -1,
        "y_max": 10,
        "y_axis_height": 6,
        "y_tick_frequency": 1,
        "y_bottom_tick": None, # Change if different from y_min
        "y_labeled_nums": None,
        "y_axis_label": "$y$",
        "axes_color": GREY,
        "graph_origin": 2.5 * DOWN + 4 * LEFT,
        "exclude_zero_label": True,
        "num_graph_anchor_points": 25,
        "default_graph_colors": [BLUE, GREEN, YELLOW],
        "default_derivative_color": GREEN,
        "default_input_color": YELLOW,
        "default_riemann_start_color": BLUE,
        "default_riemann_end_color": GREEN,
        "area_opacity": 0.8,
        "num_rects": 50,
    }
```

对GraphScene() 而言（可以在 scene.py 文件中找到），其父类具有如下字典：

```
class Scene(Container):
    CONFIG = {
        "camera_class": Camera,
        "camera_config": {},
        "frame_duration": LOW_QUALITY_FRAME_DURATION,
        "construct_args": [],
        "skip_animations": False,
        "ignore_waits": False,
        "write_to_movie": False,
        "save_frames": False,
        "save_pngs": False,
        "pngs_mode": "RGBA",
        "movie_file_extension": ".mp4",
        "name": None,
        "always_continually_update": False,
        "random_seed": 0,
        "start_at_animation_number": None,
        "end_at_animation_number": None,
        "livestreaming": False,
        "to_twitch": False,
        "twitch_key": None,
    }
```

对于Container（）容器，Scene的父类以及Mobject对象，在CONFIG {}中是没有条目入口的。

在讨论mobjects时，CONFIG {}配置的列表可能会有点长。 我现在不会去讨论这些内容，但是值得你花一些时间去看一下某些mobject子类的层次结构，以了解所有可以控制的属性会有哪些。

8.0 更多绘图

让我们深入研究一下 Manim 中绘图函数的特性。

```
class ExampleApproximation(GraphScene):
    CONFIG = {
        "function" : lambda x : np.cos(x),
        "function_color" : BLUE,
        "taylor" : [lambda x: 1, lambda x: 1-x**2/2, lambda x: 1-
x**2/math.factorial(2)+x**4/math.factorial(4), lambda x: 1-
x**2/2+x**4/math.factorial(4)-x**6/math.factorial(6),
        lambda x: 1-x**2/math.factorial(2)+x**4/math.factorial(4)-
x**6/math.factorial(6)+x**8/math.factorial(8), lambda x: 1-
x**2/math.factorial(2)+x**4/math.factorial(4)-
x**6/math.factorial(6)+x**8/math.factorial(8) - x**10/math.factorial(10)],
        "center_point" : 0,
        "approximation_color" : GREEN,
        "x_min" : -10,
        "x_max" : 10,
        "y_min" : -1,
        "y_max" : 1,
        "graph_origin" : ORIGIN ,
        "x_labeled_nums" :range(-10,12,2),
```

```
}

def construct(self):

    self.setup_axes(animate=True)

    func_graph = self.get_graph(

        self.function,

        self.function_color,

    )

    approx_graphs = [

        self.get_graph(

            f,

            self.approximation_color

        )

        for f in self.taylor

    ]



    term_num = [

        TexMobject("n = " + str(n),aligned_edge=TOP)

        for n in range(0,8)]

    #[t.to_edge(BOTTOM,buff=SMALL_BUFF) for t in term_num]
```

```
#term = TexMobject("")
#term.to_edge(BOTTOM,buff=SMALL_BUFF)
term = VectorizedPoint(3*DOWN)

approx_graph = VectorizedPoint(
    self.input_to_graph_point(self.center_point, func_graph)
)

self.play(
    ShowCreation(func_graph),
)
for n,graph in enumerate(approx_graphs):
    self.play(
        Transform(approx_graph, graph, run_time = 2),
        Transform(term,term_num[n])
    )
    self.wait()
```

【视频：】Example Approximation - manim Series 8.1
https://youtu.be/86tJtr5TrWI

我想演示一下，在一个泰勒展开式中，如何更好地添加高次项并能够更好地达成功能一致性。 这类似于shinigamiphoenix发表在这里

（https://www.youtube.com/watch?v=ytFMmAZ5Jzg）的视频演示。

可以将要绘制的函数在CONFIG {}词典中定义为lambda函数。 如前所述，manim 处理CONFIG {}中的所有元素，并将字典条目转换为类的变量，并以键作为变量名。 因此，可以在我的calss中通过调用self.function来访问“ function”，也可以使用self.taylor方式来调用“ taylor”。 如果你还不熟悉lambda函数，请查阅“Python Conquers the Universe”上的这篇（https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/）文章。

我们使用get_graph（）创建一个图形列表以及一些列表解析。你可以在datacamp.com上找到有关针对列表解析的不错的教程。只是在阅读了本教程之后，我才将列表解析与数学符号联系起来以定义集合（例如，正实数的集合为{R和}或偶数的集合为{I和}），列表解析为我所用。而对于列表self.taylor中的每个项目，都会使用self.approximation_color来创建一个图形。我们还创建了一个名TexMobjects列表，以使用列表解析来指明泰勒展开式中哪个术语顺序应该被包括。

由于我们将从列表中进行连续性转换，因此在屏幕上保留空白占位符会对我们有所帮助。term和roximate_graph是VectorizedPoint的实例，这些mobject它们是在屏幕上不会显示任何内容的。这样，我们就可以将占位符放在屏幕上而不做任何显示，然后可以将这些mobject转换为图形或TexMobjects。

enumerate（）函数是一个有用的工具集，它可以遍历列表并返回指定项的索引。 因此，

对于 n，enumerate（approx_graphs）中的 graph 返回 0 至 4 之间的索引为 n，并将索引 n 在列表中对应的元素返回为 graph，这将应用于在每个图形中显示 term_num 的项目。

9.0 矢量场

在进入矢量场绘制之前，我们应该使用NumberPlane（）来设置下笛卡尔坐标系。这将为你提供两个轴及一个基础网格。 NumberPlane（）的CONFIG {}字典（位于ordinate_systems.py文件中）如下：

```
class NumberPlane(VMobject):
    CONFIG = {
        "color": BLUE_D,
        "secondary_color": BLUE_E,
        "axes_color": WHITE,
        "secondary_stroke_width": 1,
        # TODO: Allow coordinate center of NumberPlane to not be at (0, 0)
        "x_radius": None,
        "y_radius": None,
        "x_unit_size": 1,
        "y_unit_size": 1,
        "center_point": ORIGIN,
        "x_line_frequency": 1,
        "y_line_frequency": 1,
        "secondary_line_ratio": 1,
        "written_coordinate_height": 0.2,
        "propagate_style_to_family": False,
```

```
        "make_smooth_after_applying_functions": True,
    }
```

【视频：】DrawAnAxis - manim Series 9.0

https://youtu.be/TUDn6mUJNeI

你可以通过把任何作为关键字参数的新值传递给字典来更改这些默认值。例如，如果要更改网格线的间距，可以通过使用这些变量重新定义字典，然后将字典传递给NumberPlane（）达到更改x_line_frequency和y_line_frequency值的目的。 如果要查看x轴和y轴的指向，可使用get_axis_labels（）在相应轴的旁边绘制x和y轴标签。 请参见下面的代码。

```
class DrawAnAxis(Scene):
    CONFIG = { "plane_kwargs" : {
        "x_line_frequency" : 2,
        "y_line_frequency" :2
        }
    }

    def construct(self):
        my_plane = NumberPlane(**self.plane_kwargs)
        my_plane.add(my_plane.get_axis_labels())
        self.add(my_plane)
```

参数self.plane_kwargs前面的双星号，可以让类知道这是一个需要解压缩的字典。

我建议去尝试更改各种属性值，以便可以查看它们对轴和网格各有什么影响，这将是我们学习事情的最佳方法。

9.1 简单的矢量场

让我们从一个简单的矢量场开始：一个常量场。我们首先需要为每个网格点定义一组矢量点，在每个网格点定义字段，然后为每个点的场创建Vector（）。最后，我们将所有Vector（）实例组合到一个VGroup中，以使我们能够使用单个命令来绘制所有矢量线。

```
class SimpleField(Scene):
    CONFIG = {
    "plane_kwargs" : {
        "color" : RED
        },
    }
    def construct(self):
        plane = NumberPlane(**self.plane_kwargs)
        plane.add(plane.get_axis_labels())
        self.add(plane)

        points = [x*RIGHT+y*UP
```

```
        for x in np.arange(-5,5,1)
        for y in np.arange(-5,5,1)
        ]

    vec_field = []
    for point in points:
        field = 0.5*RIGHT + 0.5*UP
        result = Vector(field).shift(point)
        vec_field.append(result)

    draw_field = VGroup(*vec_field)
    self.play(ShowCreation(draw_field))
```

【视频：】SimpleField - manim series 9.1
https://youtu.be/vUhryTFfB_c

在创建NumberPlane（）之后，我们使用列表解析来创建所有网格点位置的列表。请记住，RIGHT = np.array（1,0,0）和UP = np.array（0,1,0），因此，此列表以单位步长为单位，包含了从（5,5,0）到（-5，-5，0）的所有点。 arange（）中的最后一个参数指定了步长。接下来，我们创建一个空列表vec_field来保存我们将要创建的所有向量。 for循环遍历每个以点为单位网格位置，并创建一个其长度和方向由场定义好的向量。每次在循环中进行场的定义其效率很低，但我们需要为以后做好准备。 shift（point）命令将矢量移动到point定义的网格位置。然后将这些结果添加到列表中。经过for循环后，所有向

量都被组合在一个称为draw_field的VGroup中。这样做的唯一原因是，你可以使用单个添加或播放命令添加draw_field，你也可以在for循环的每次迭代中都包含self.add（result），然后到了draw_field时进行创建，但是使用VGroup感觉会更为简洁。

9.2 变化的矢量场

对于稍微有趣的场，我们将研究由于正电荷而产生的电场。 电场为：

$$\vec{E} = \frac{1}{4\pi\epsilon_0}\frac{q}{r^3}\vec{r}$$

其中，q_1是点电荷上的电荷，$\frac{1}{4\pi\epsilon_0} = 9 \times 10^9 Nm^2/C^2$是电荷与观察点之间的距离矢量，并且是该矢量的大小。 前面的常量本质上是一个转换因子。 为了实现我们的目的，我们将所有常量设置为零，然后看一下

$$\vec{E} = \frac{1}{r^3}\vec{r}.$$

```
class FieldWithAxes(Scene):
    CONFIG = {
    "plane_kwargs" : {
        "color" : RED_B
        },
    "point_charge_loc" : 0.5*RIGHT-1.5*UP,
    }
```

```
    def construct(self):
        plane = NumberPlane(**self.plane_kwargs)
        plane.add(plane.get_axis_labels())
        self.add(plane)


        field = VGroup(*[self.calc_field(x*RIGHT+y*UP)
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
            ])


        self.play(ShowCreation(field))



    def calc_field(self,point):
        #This calculates the field at a single point.
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        #efield = np.array((-y,x,0))/math.sqrt(x**2+y**2)  #Try one of these two
fields
        #efield = np.array(( -2*(y%2)+1 , -2*(x%2)+1 , 0 ))/3  #Try one of these
```

two fields

return Vector(efield).shift(point)

【视频：】FieldWithAxes - manim series 9.2

https://youtu.be/TG4PXtijtm0

点电荷的位置在CONFIG {}中进行设置。为了创建矢量场，我们压缩了之前的代码。 我们使用列表推导和函数calc_field（）作为VGroup（）的参数。 calc_field（）函数定义要计算的字段。 为了使公式更易于阅读，我们从点向量和self.point_charge_loc向量对x和y坐标进行了解压缩，代码x，y = point [：2]等效于x = point [0]和y = point [1]。

fade（0.9）函数作用：将线条的不透明度设置为1减去淡入度（所以在这种情况下不透明度设置为0.1）， 这样做的目的是为了让你能够更容易看到远离充电位置的细小箭头。

更多尝试：

–更改CONFIG {}中NumberPlane（）的每个元素，以检查它们对轴和网格线的影响。

–计算不同的场

–试试efield = np.array（（-y，x，0））/ math.sqrt（x ** 2 + y ** 2）

–试试efield = np.array（（-2 *（y%2）+1，-2 *（x%2）+1，0））/ 3

–提出自己的方程式

10.0 移动电场

在Reddit上有一个关于创建电场中电荷如何移动的问题，由于这也是我想要做的事情，所以我觉得尝试一下会很有趣。

在创建变化的场之前，我认为我应该先要从场的转换开始。我知道我在其中一个视频中看到了这一点，因此从代码开始工作，并根据需要予以修改。 如下是我得出的思路：

```
class MovingCharges(Scene):
    CONFIG = {
    "plane_kwargs" : {
        "color" : RED_B
        },
    "point_charge_loc" : 0.5*RIGHT-1.5*UP,
    }
    def construct(self):
        plane = NumberPlane(**self.plane_kwargs)
        plane.add(plane.get_axis_labels())
        self.add(plane)

        field = VGroup(*[self.calc_field(x*RIGHT+y*UP)
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
```

```
            ])
        self.field=field
        source_charge = self.Positron().move_to(self.point_charge_loc)
        self.play(FadeIn(source_charge))
        self.play(ShowCreation(field))
        self.moving_charge()


    def calc_field(self,point):
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        return Vector(efield).shift(point)


    def moving_charge(self):
        numb_charges=4
        possible_points = [v.get_start() for v in self.field]
        points = random.sample(possible_points, numb_charges)
        particles = VGroup(*[
            self.Positron().move_to(point)
            for point in points
        ])
```

```
        for particle in particles:
            particle.velocity = np.array((0,0,0))

        self.play(FadeIn(particles))
        self.moving_particles = particles
        self.add_foreground_mobjects(self.moving_particles )
        self.always_continually_update = True
        self.wait(10)

    def field_at_point(self,point):
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        return efield

    def continual_update(self, *args, **kwargs):
        if hasattr(self, "moving_particles"):
            dt = self.frame_duration
            for p in self.moving_particles:
                accel = self.field_at_point(p.get_center())
                p.velocity = p.velocity + accel*dt
```

```
                p.shift(p.velocity*dt)


class Positron(Circle):
    CONFIG = {
    "radius" : 0.2,
    "stroke_width" : 3,
    "color" : RED,
    "fill_color" : RED,
    "fill_opacity" : 0.5,
    }
    def __init__(self, **kwargs):
        Circle.__init__(self, **kwargs)
        plus = TexMobject("+")
        plus.scale(0.7)
        plus.move_to(self)
        self.add(plus)
```

【视频：】MovingCharges - manim Series Part 11.1

https://youtu.be/ZbDjQbl5mzg

这里最重要的方法是continuation_update（），此方法会更新整个场景中每一帧的显示。 这与依赖play（）方法的各种转换的不同之处在于，这些转换会发生在很短的时间间隔内，通常在几秒钟的数量级上，而连续方法会在整个场景中持续运行。 如果我们想让粒

子在屏幕上移动，我们可能会想使用诸如self.play（ApplyMethod（particle1.shift，5 * LEFT））之类的东西，但是在同一时间，你要同时也控制其他进行的转换，那就会相当困难了，而 Continuousal_update（）则允许你在后台对事物进行动画处理，同时仍控制其他同一时间段的转换。

由于我知道我将在视频中使用带电粒子，所以我编写了Positron class 来创建带正电的粒子。 正电子是带正电的粒子。我为什么不创建一个质子呢？ 因为质子的质量大约是电子的约2000倍，所以我需要一个类似大小的粒子来实现它。

我们重用了上一篇有关电场的文章中的代码，但是我们新添加了一些方法用来创建带电粒子并将其移动。 moving_charge（）是通过随机选择一个场点（possible_points = [v.get_start() for v in self.field] is a list of the locations of the tails of all field vectors）并且通过选择 numb_charges 点来创建 粒子的。 请注意，随机产生的电荷是不会互相互反应的，对于观看这里的视频你可能会有点不解，因为它不完全是在物理层面呈现的。

粒子是向量化的mobject组，其包含了所有移动电荷，且初始速度设置为零（particle.velocity = np.array（（0,0,0））。我们可以通过仅在一个固定位置使用一个粒子来简化代码，但以后需要多次充电。然后将这些粒子添加到屏幕上（self.play（FadeIn（particles））并将其分配给continuous_update（self.moving_particles =particles）中所需的类变量。Mobject按照它们添加到屏幕的顺序进行绘制，但是你也可以使用add_foreground_mobjects（）将某些mobject放置在前景中，以确保它们始终被绘制在其他对象的顶层，这与Photoshop或类似软件中的图层类似，不同之处在于每个

mobject都位于其自己的图层中。 manim 会将所有绘制在屏幕上的对象都依序保存在其列表中，虽然 manim 中没有与之等效的背景mobject方法，但你也可以使用Brive_to_front（）/Bring_to_back（ ）来调整目标对象的向前/向后的“图层”移动。

接下来，我们告诉manim 在后台不断更新内容（self.always_continually_update = True），然后等待十秒钟。 设置wait（）命令很重要，因为持续更新仅在它们是动画队列中的动画元素（play（）命令）或wait（）命令时才会运行。

field_at_point（）方法复制了一些先前的代码，但用于返回数值类型的矢量（一个由numpy组成的三元组），而不是返回一个calc_field（）调用后产生的mobject Vector。为了了解为什么在使用 calc_field() 后能够找到force vector，为此我花费了大量的时间。

当场景被和合成时，每帧都会调用continuousal_update（）方法。 第一行，if hasattr(self, "moving_particles"): 如果尚未创建self.moving_particles，则余下代码不会运行并将抛出错误。 帧持续时间为1 / 15、1 / 30或1/60秒，具体将取决于你渲染的视频是低、中还是生产质量级的品质（即，是否包含了-l，-m或不包含这些命令行参数）。

我们遍历所有运动粒子的列表（for p in self.moving_particles:），然后计算由于每个粒子所在位置的电场而引起的加速度（vect = self.field_at_point(p.get_center())）。p.get_center（）会返回每个粒子p的矢量点位。粒子使用更新后的速率，然后移动相应的距离。

10.1 更新运动电荷的电场

现在，我们已拥有了在屏幕上随意移动对象的经验，因此，我们来继续计算由于粒子而引起的场变。我们将继续重用先前程序中的许多代码，并进行一些更改。

```
class FieldOfMovingCharge(Scene):
    CONFIG = {
    "plane_kwargs" : {
        "color" : RED_B
        },
    "point_charge_start_loc" : 5.5*LEFT-1.5*UP,
    }
    def construct(self):
        plane = NumberPlane(**self.plane_kwargs)
        #plane.main_lines.fade(.9)
        plane.add(plane.get_axis_labels())
        self.add(plane)

        field =
VGroup(*[self.create_vect_field(self.point_charge_start_loc,x*RIGHT+y*UP)
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
            ])
        self.field=field
```

```
        self.source_charge = self.Positron().move_to(self.point_charge_start_loc)
        self.source_charge.velocity = np.array((1,0,0))
        self.play(FadeIn(self.source_charge))
        self.play(ShowCreation(field))
        self.moving_charge()

    def create_vect_field(self,source_charge,observation_point):
        return
Vector(self.calc_field(source_charge,observation_point)).shift(observation_point)

    def calc_field(self,source_point,observation_point):
        x,y,z = observation_point
        Rx,Ry,Rz = source_point
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2 + (z-Rz)**2)
        if r<0.0000001:     #Prevent divide by zero    ##Note:    This won't work -
fix this
            efield = np.array((0,0,0))
        else:
            efield = (observation_point - source_point)/r**3
        return efield
```

```
def moving_charge(self):
        numb_charges=3
        possible_points = [v.get_start() for v in self.field]
        points = random.sample(possible_points, numb_charges)
        particles = VGroup(self.source_charge, *[
            self.Positron().move_to(point)
            for point in points
        ])
        for particle in particles[1:]:
            particle.velocity = np.array((0,0,0))
        self.play(FadeIn(particles[1:]))
        self.moving_particles = particles
        self.add_foreground_mobjects(self.moving_particles )
        self.always_continually_update = True
        self.wait(10)


def continual_update(self, *args, **kwargs):
        Scene.continual_update(self, *args, **kwargs)
        if hasattr(self, "moving_particles"):
            dt = self.frame_duration
```

```
                for v in self.field:
                    field_vect=np.zeros(3)
                    for p in self.moving_particles:
                        field_vect = field_vect + self.calc_field(p.get_center(),
v.get_start())
                    v.put_start_and_end_on(v.get_start(), field_vect+v.get_start())


                for p in self.moving_particles:
                    accel = np.zeros(3)
                    p.velocity = p.velocity + accel*dt
                    p.shift(p.velocity*dt)


        class Positron(Circle):
            CONFIG = {
            "radius" : 0.2,
            "stroke_width" : 3,
            "color" : RED,
            "fill_color" : RED,
            "fill_opacity" : 0.5,
            }
            def __init__(self, **kwargs):
                Circle.__init__(self, **kwargs)
```

```
plus = TexMobject("+")
plus.scale(0.7)
plus.move_to(self)
self.add(plus)
```

【视频：】FieldOfMovingCharge - manim Series Part 11.2
https://youtu.be/h4mHMJXFwRU

我们所做的一项更改是让calc_field（）返回一个numpy vector，而不是一个mobject Vector。 这的确意味着要添加create_vect_field（），以便从numpy vectors来创建mobject。

因为我们希望源电荷能够移动，所以我们必须将该源电荷添加到 particles 列表中。 因此，我们创建的Vgroup包括该源电荷加上使用particles = VGroup(self.source_charge, *[self.Positron().move_to(point) for point in points])。 请记住，列表前面的星号让Python知道列表中的每个元素都应被解压，并作为VGroup（）类分离的参数来对待。为了帮助理解这一行代码，我们可以将其分解为一种不太雅致的形式：

```
list_of_random_charges=[]
for point in points:
    new_charge = self.Positron().move_toe(point)
    list_of_random_charges.append(new_charge)
    particles = VGroup(self.source_charge, list_of_random_charges[0],
    list_of_random_charges[1], list_of_random_charges[2])
```

因此，用一行代码代替了几行代码。 我们在代码中使用particles[1:]来定义速度及粒子衰落的原因是：源电荷已经具有速度并且已在屏幕上了，因此，我们不想重新定义速度并使其再次消失（ 让其闪烁）。

在continuous_update（）中，现在我们需要在每个网格点处依据每个时间步长来计算场变情况。 首先，我们遍历每个场点（for v in self.field:）。 由于我们要从多次充电中累加场变，因此将场向量设置为零（field_vect = np.zeros（3））， 然后由于每次充电而在该点处累加场向量（field_vect = field_vect + self.calc_field(p.get_center(), v.get_start()））。 我们需要通过指定向量的起点和终点来重绘场向量。 起点是矢量开始的初始网格点（v.get_start()），箭头的尖端等于字段矢量加上起点的距离（field_vect+v.get_start()）。

我并没有实现粒子对其他粒子的场是如何作出反应的，对我而言，这看起来非常不真实，但是应该很容易实现，我仅仅是为了发布一篇介绍如何使场线正常工作的基础知识文章。

11.0 三维场景

这是本教程与以前的系列有所不同的第一处，基于以下事实：在切换 Python 3.7 后，manim 的许多变化（至少我已经看到）似乎都集中在了改善 manim 的 3D 功能上。

这可能给了一个解释我是如何弄明白Manim 工作的一个好时机。 我要做的第一件事是转到active_projects目录，找到一个我感兴趣的相关视频文件，然后将其从中进行分理，你要确保选择的活动项目是保编译不会出现问题的。 然后，我将单个场景复制并粘贴到另一个文件中，并开始从代码组件部分进行剥离，直至获得一个我感兴趣的、可简单工作的示例，为此我开始研究一些CONFIG条目并开始添加一些功能。 我发现在github站点上搜索其他场景是非常有用的，那些场景使用了类似的命令来确保可用选项次序的有效性。 格兰特·桑德森（Grant Sanderson）使用的命名规范已经足够清晰了，你通常只需要经历很少的尝试和错误就可以弄清楚所做的内容了。

CONFIG 字典中的 ThreeDScene 类定义如下：

```
class ThreeDScene(Scene):
    CONFIG = {
        "camera_class": ThreeDCamera,
        "ambient_camera_rotation": None,
        "default_angled_camera_orientation_kwargs": {
        "phi": 70 * DEGREES,
        "theta": -135 * DEGREES,
        }
    }
```

你在 ThreeDScene （位于 three_d_scene.py 文件中）函数中可以调用的方法如下：

- set_camera_orientation
- begin_ambient_camera_rotation
- stop_ambient_camera_rotation
- move_camera

还有另外的其他几种方法，但现在我们将重点介绍这些方法。 首先，我们将创建一个普通的2D场景，但是使用3D相机进行旋转，我们将重用上一节中的代码：

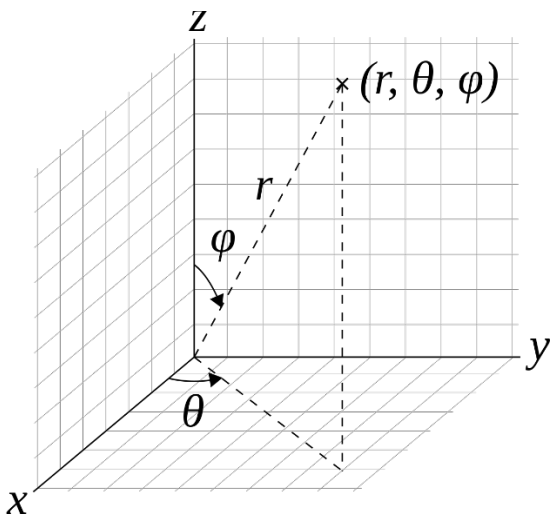
```
class ExampleThreeD(ThreeDScene):
    CONFIG = {
    "plane_kwargs" : {
        "color" : RED_B
        },
    "point_charge_loc" : 0.5*RIGHT-1.5*UP,
    }
    def construct(self):
        plane = NumberPlane(**self.plane_kwargs)
        plane.add(plane.get_axis_labels())
        self.add(plane)

        field2D = VGroup(*[self.calc_field2D(x*RIGHT+y*UP)
```

```
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
            ])

        self.set_camera_orientation(phi=PI/3,gamma=PI/5)
        self.play(ShowCreation(field2D))
        self.wait()
        #self.move_camera(gamma=0,run_time=1)  #currently broken in manim
        self.move_camera(phi=3/4*PI, theta=-PI/2)
        self.begin_ambient_camera_rotation(rate=0.1)
        self.wait(6)

    def calc_field2D(self,point):
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        return Vector(efield).shift(point)
```

【视频：】ExampleThreeD

https://youtu.be/byeXslxXIR4

通过将场景定义为ThreeDScene的子类，我们可以访问3D摄像机选项。 然后，只需移动摄像机即可。



By Dmcq – Own work, CC BY-SA 3.0, Link

使用set_camera_orientation（）设置相机的原始方向并使之作用，你也可以通过distance变量来设置从摄像机到原点之间的距离，还有一个伽玛（希腊字母）选项（也即欧拉角）可以设定。通过改变伽玛值会让相机绕着穿过镜头中心的轴进行旋转，从而使我们可以改变屏幕上的水平方向。请注意，如果我们在场景中间使用了set_camera_orientation，则相机将跳至新的方向。

设置相机方向时要记住的一件事是，尽管相机本自身是指向原点的，但角度却是从相机设置的中心轴上测量得到的。出于某种原因，并且对应于正y轴（在右侧）与正x轴（在下方）的坐标关系，这就解决了如何与正常的2D方向（其中x轴指向右，y轴指向屏幕）的关系对应。如果一旦设定，我们将让屏幕予以翻转。

为使摄像机平稳移动，我们使用move_camera（）函数，它具有与set_camera_orientation（）相同的参数。 编注：例如，你可以指定run_time = 4，来设定move_camera（）操作花费4秒时间。

我们可以通过调用begin_ambient_camera_rotation（）来设置摄像机绕z轴旋转，并指定摄像机旋转的速率，我想速率应是以弧度/秒为单位的。

如果你对度数比较熟悉，则可以将你的角度乘以DEGREES常量来得到度数。因此，默认的摄影机方向设定为：

be self.set_camera_orientation(phi=0*DEGREES, theta=-90*DEGREES)。

更多尝试：

–通过angles phi, theta，以及 gamma 的设定，来了解相机的方向

–创建一系列2D mobject，并使用3D摄像机在其周围进行缩放操作

12.0 协同 SVG 文件工作

3B1B中的 PiCreatures 是可缩放的矢量图形（svg）文件。 manim 有一个SVGMobject类，可以导入svg文件。 为了在manim 中使用svg图像，我使用了开源矢量图形包（Inkscape）[https://inkscape.org/en/]创建了一些图形。 我想在manim 中尝试创建火柴人挥舞的简笔画，所以我创建了两个人物，一个是正常人物，另一个是挥手的。 你可以获取本文结尾处使用的svg文件。 将它们放在 \media\designs\svg_images\文件夹中。

我用来导入简笔画的代码基于 \manim lib\for_3b1b_videos\pi_creatures.py 中的PiCreature代码。 我的代码如下：

```
HEAD_INDEX = 0
BODY_INDEX = 1
ARMS_INDEX = 2
LEGS_INDEX = 3


class StickMan(SVGMobject):
    CONFIG = {
        "color" : BLUE_E,
        "file_name_prefix": "stick_man",
        "stroke_width" : 2,
        "stroke_color" : WHITE,
```

```
        "fill_opacity" : 1.0,
        "height" : 3,
    }
    def __init__(self, mode = "plain", **kwargs):
        digest_config(self, kwargs)
        self.mode = mode
        self.parts_named = False
        try:
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
                "%s_%s.svg" % (self.file_name_prefix, mode)
            )
            SVGMobject.__init__(self, file_name=svg_file, **kwargs)
        except:
            warnings.warn("Noesign with mode %s" %
                          (self.file_name_prefix, mode))
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
                "stick_man_plain.svg",
            )
            SVGMobject.__init__(self, mode="plain", file_name=svg_file, **kwargs)
```

```
    def name_parts(self):
        self.head = self.submobjects[HEAD_INDEX]
        self.body = self.submobjects[BODY_INDEX]
        self.arms = self.submobjects[ARMS_INDEX]
        self.legs = self.submobjects[LEGS_INDEX]
        self.parts_named = True

    def init_colors(self):
        SVGMobject.init_colors(self)
        if not self.parts_named:
            self.name_parts()
        self.head.set_fill(self.color, opacity = 1)
        self.body.set_fill(RED, opacity = 1)
        self.arms.set_fill(YELLOW, opacity = 1)
        self.legs.set_fill(BLUE, opacity = 1)
        return self
```

我确定我可以进一步对代码进行优化，但我只想一些无需过多调试即可工作的代码。 我已将火柴人的两个文件命名为stick_man_plain.svg和stick_man_wave.svg。 如果你在实例化StickMan类时未指定模式，它将尝试加载具有CONFIG词典中指定的前缀（在本示例中为stick_man）和后缀为_plain的文件。 模式变量可用于指定相关文件。 例如，挥舞着棍

棒的svg文件称为stick_man_wave.svg，因此，如果我指定wave的模式，此类将加载该文件。 我可以使用StickMan("wave")使用挥舞的人物来创建火柴人的实例。 以下是引起火柴人风潮的场景代码，你可以在这里（https://github.com/zimmermant/manim_tutorial）找到我在Github存储库中使用的svg文件：

```
class Waving(Scene):
    def construct(self):
        start_man = StickMan()
        plain_man = StickMan()
        waving_man = StickMan("wave")

        self.add(start_man)
        self.wait()
        self.play(Transform(start_man,waving_man))
        self.play(Transform(start_man,plain_man))

        self.wait()
```

【视频：】Waving1
https://youtu.be/5AP2zS2hxAE

之所以创建StickMan（）的两个实例，是因为我正在转换start_man，但希望图像最终看起来与原始图一样。

有两件事需要注意：

（1）PiCreatures的stroke_width和stroke_color设置为不绘制对象的轮廓。 如果要查看线条或形状的轮廓，则需要将这些值设置为可见的值（即，非零stroke_width和不同于背景的stroke_color）。

（2）svg中的行被标记为路径。 manim 处理路径的方式将其视为封闭形状。 这意味着如果我不将线条的不透明度设置为零，则会看到一个封闭的形状。 请参见下面的视频，其中我已将init_colors方法中的fill_opacity设置为1。

【视频：】Waving2

https://youtu.be/b4F7ltUo5x4

尽管我还没有深入研究Manim 代码，但我认为所有Manim 所关注的只是形状的轮廓，而不是填充。

我创建了第二个场景，只是为了确保我对可缩放矢量图形导入有一个了解。 创建自己的图像时，需要在文本编辑器中打开.svg文件，以确定每个子对象的索引。 manim 将每个svg实体（例如，路径、椭圆形、方框或其他形状）作为单个子对象导入，你将需要确定父SVGMobject类中这些项目的顺序。 我创建了几个形状（通过线连接了一对正方形与圆）。 此处显示了svg文件的相关部分（文件中还有很多元数据被我遗漏了）。

该文件包含一个圆，两个路径和两个矩形。 因此，当导入manim 时，圆将是第一个子对象（索引为0），两条路径或直线将是第二和第三子对象（索引1和2），而两个正方形将是第四和第五子对象（ 索引3和4）。 我用于定义此圆形和正方形绘图的类是

```
class CirclesAndSquares(SVGMobject):
    CONFIG = {
        "color" : BLUE_E,
        "file_name_prefix": "circles_and_squares",
        "stroke_width" : 2,
        "stroke_color" : WHITE,
        "fill_opacity" : 1.0,
        "height" : 3,
        "start_corner" : None,
        "circle_index" : 0,
        "line1_index" :1,
        "line2_index" : 2,
        "square1_index" : 3,
        "square2_index" : 4,
    }
    def __init__(self, mode = "plain", **kwargs):
        digest_config(self, kwargs)
        self.mode = mode
        self.parts_named = False
        try:
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
```

```
                "%s_%s.svg" % (self.file_name_prefix, mode)
            )
            SVGMobject.__init__(self, file_name=svg_file, **kwargs)
        except:
            warnings.warn("Noesign with mode %s" %
                          (self.file_name_prefix, mode))
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
                "circles_and_squares_plain.svg",
            )
            SVGMobject.__init__(self, mode="plain", file_name=svg_file, **kwargs)


def name_parts(self):
    self.circle = self.submobjects[self.circle_index]
    self.line1 = self.submobjects[self.line1_index]
    self.line2 = self.submobjects[self.line2_index]
    self.square1 = self.submobjects[self.square1_index]
    self.square2 = self.submobjects[self.square2_index]
    self.parts_named = True


def init_colors(self):
```

```
        SVGMobject.init_colors(self)
        self.name_parts()
        self.circle.set_fill(RED, opacity = 1)
        self.line1.set_fill(self.color, opacity = 0)
        self.line2.set_fill(self.color, opacity = 0)
        self.square1.set_fill(GREEN, opacity = 1)
        self.square2.set_fill(BLUE, opacity = 1)
        return self
```

在类开始时，我使用了svg文件中不同元素的顺序，那些已在我的CONFIG词典中已经进行了标记索引。 在屏幕上显示此代码为

```
class SVGCircleAndSquare(Scene):
    def construct(self):
        thingy = CirclesAndSquares()
        self.add(thingy)
        self.wait()
```

【视频:】SVGCircleAndSquare

https://youtu.be/PeajCLr4-ZQ

我知道我可以减少svg类的代码，但我仍将再保留一些时日。

svg文件和教程文件都可以从此处（https://github.com/zimmermant/manim_tutorial）下载获得。

三、附录：英文原址及内容

Getting Started Animating with manim and Python 3.7

Posted on January 8, 2019by talkingphysics

I previously wrote a series of blog posts detailing how to use manim, the **m**athematical **anim**ation package created by Grant Sanderson of 3Blue1Brown. Since I've written those posts there have been many changes to manim, including switching to Python 3.7. I will go through and update my information for the version of manim as of December, 2018. Much of the information will be a repeat of earlier posts in situations where there have been no changes to the manim code. The primary changes from my previous series of posts are related to changes to manim, primarily in dealing with 3D scenes. Note that future versions may break some of these commands, but hunting down the problems is the best way to learn the inner workings of manim.

1.0 Installing manim

Brian Howell has put together a really nice post on how to install the necessary components of manim at http://bhowell4.com/manic-install-tutorial-for-mac/. One of the most useful tips for making sure everything works is to use virtual environments. If you have trouble getting manim working I suggest asking for help on the github page for manim since that has an active group of users who can typically help out.
The Readme docs on github also have instructions on installing manim.
To make sure you installation is working you can run the example file that comes with manim. Type `python -m manim example_scenes.py -pl`. If this produces errors you should check out the Issues tab on the github site since frequently someone else has had the same issue.

2.0 Creating Your First Scene

You can copy and paste the code below into a new text file and save it as `manim_tutorial_P37.py` in the top-level manim directory or you can download all of the tutorials at https://github.com/zimmermant/manim_tutorial/blob/master/manim_tutorial_P37.py. The `.py` extension tells your operating system that this is a Python file.
Open up a command line window and go to the top-level manim directory, and type `python -m manim pymanim_tutorial_P37.py Shapes -pl`

We are calling the Python interpreter with the `python` command. If you have multiple versions of Python installed you may need to call `python3` rather than just `python` (I use Anaconda virtual environments to keep all manim-related Python code in one handy place).

The first argument passed to Python, `manim` is running manim.py in the main manim directory (we'll ignore the `-m` switch, which you should include). It looks like you can live-stream the output to Twitch but I'm not using that feature so I'll focus on extract_scene.py, which is called from manim.py and which is the code that runs your script and creates a video file. The second argument, `manim_tutorial_P37.py` is the name of the file (i.e. the module) where your script is stored. The third argument, `Shapes` is the name of the class (i.e. the scene name) defined within your file that gives instructions on how to construct a scene. The last arguments, `-pl` tell the `extract_scene` script to **p**review the animation by playing it once it is done and to render the animation in **l**ow quality, which speeds up the time to create the animation. Typing `python -m manim --help` will pull up a list of the different arguments you can use when calling `python -m manim`.

```
from big_ol_pile_of_manim_imports import *



class Shapes(Scene):

    #A few simple shapes

    def construct(self):

        circle = Circle()

        square = Square()

        line=Line(np.array([3,0,0]),np.array([5,0,0]))

triangle=Polygon(np.array([0,0,0]),np.array([1,1,0]),np.arra
y([1,-1,0]))

        self.add(line)
        self.play(ShowCreation(circle))
        self.play(FadeOut(circle))
        self.play(GrowFromCenter(square))
        self.play(Transform(square,triangle))
```

If everything works you should see the following messages (or something similar) in your terminal:

The first line in the command terminal tells you where the video file is being saved. The next several lines list the name of the animation commands that you called, along with some information about how long it took for each animation and other info I don't understand. The last line just lets you know how many animations were called in your script.

The video should look like this:

All of the various manim modules are contained in `big_ol_pile_of_manim_imports.py` so importing this gives you all of the basic features of manim. This doesn't include every single module from manim but it contains the core modules. You can look at the modules included here. It is worth your time to dive into some of the modules to see how things are put together. I've learned a surprising amount of Python by trying to figure out how things work. Incidentally I find using the search box at https://github.com/3b1b/manim very helpful for finding different classes and figuring out what arguments they take and how they work. Documentation is also being put together for manim here, although it is still a work in progress.

2.1 Scenes and Animation

The `Scene` is the script that tells manim how to place and animate your objects on the screen. I read that each 3blue1brown video is created as individual scenes which are stitched together using video editing software. You must define each scene as a separate class that is a subclass of `Scene`. This class must have a `construct()` method, along with any other code required for creating objects, adding them to the screen, and animating them.
The `construct()` method is essentially the main method in the class that gets called when run through `extract_scene.py` (which is called by the manim.py script). It is similar to `__init__`; it is the method that is automatically called when you create an instance of any subclass of `Scene`. Within this method you should define all of your objects, any code needed to control the objects, and code to place the objects onscreen and animate them. For this first scene we've created a circle, a square, a line, and a triangle. Note that the coordinates are specified using numpy arrays `np.array()`. You can

pass a 3-tuple like `(3,0,0)`, which works sometimes, but some of the transformation methods expect the coordinates to be a numpy array.
One of the more important methods from the `Scene()` class is the `play()` method. `play()` is what processes the various animations you ask manim to perform. My favorite animation is `Transform`, which does a spectacular job of morphing one math object (a **mobject**) into another. This scene shows a square changing into a triangle, but you can use the transform to morph any two objects together. To have objects appear on the screen without any animation you can use `add()` to place them. The line has been added and shows up in the very first frame, while the other objects either fade in or grow. The naming of the transformations is pretty straight forward so it's usually obvious what each one does.

Things to try

– Change the order of the `add()` and `play()` commands. How does changing the order affect when they appear on the screen.
– Try using the `Transform()` method on other shapes.
– Check out the shapes defined in geometry.py which is located in the `/manim/manimlib/mobject/` folder.

3.0 More Shapes

You can create almost any geometric shape using manim. You can create circles, squares, rectangles, ellipses, lines, and arrows. Let's take a look at how to draw some of those shapes.

You can download the completed code here: manim_tutorial_P37.py. After downloading the tutorial file to your top level manim directory you can type the following into the command line to run this scene: `python -m manim manim_tutorial_P37.py MoreShapes -pl`.

```
class MoreShapes(Scene):

    def construct(self):

        circle = Circle(color=PURPLE_A)

        square = Square(fill_color=GOLD_B, fill_opacity=1,
color=GOLD_A)

        square.move_to(UP+LEFT)

        circle.surround(square)
```

```
        rectangle = Rectangle(height=2, width=3)

        ellipse=Ellipse(width=3, height=1, color=RED)

        ellipse.shift(2*DOWN+2*RIGHT)
        pointer = CurvedArrow(2*RIGHT,5*RIGHT,color=MAROON_C)
        arrow = Arrow(LEFT,UP)
        arrow.next_to(circle,DOWN+LEFT)
        rectangle.next_to(arrow,DOWN+LEFT)
        ring=Annulus(inner_radius=.5, outer_radius=1,
color=BLUE)
        ring.next_to(ellipse, RIGHT)

        self.add(pointer)
        self.play(FadeIn(square))
        self.play(Rotating(square),FadeIn(circle))
        self.play(GrowArrow(arrow))
        self.play(GrowFromCenter(rectangle),
GrowFromCenter(ellipse), GrowFromCenter(ring))
```

You'll notice we have a few new shapes and we are using a couple of new commands. Previously we saw the `Circle`, `Square`, `Line`, and `Polygon` classes. Now we've added `Rectangle`, `Ellipse`, `Annulus`, `Arrow`, and `CurvedArrow`. All shapes, with the exception of lines and arrows, are created at the origin (center of the screen, which is (0,0,0)). For the lines and arrows you need to specify the location of the two ends.

For starters, we've specified a color for the square using the keyword argument `color=`. Most of the shapes are subclasses of VMobject, which stands for a **v**ectorized **m**ath **object**. `VMobject` is itself a subclass of the **m**ath **object** class `Mobject`. The best way to determine the keyword arguments you can pass to the classes are to take a look at the allowed arguments for the VMobject and Mobject class. Some possible keywords include `radius`, `height`, `width`, `color`, `fill_color`, and `fill_opacity`. For the `Annulus` class we have `inner_radius` and `outer_radius` for keyword arguments.

A list of the named colors can be found in the `COLOR_MAP` dictionary located in the constant.py file which is located in the `/manim/manimlib/` directory. The named colors are keys to the `COLOR_MAP` dictionary which yield the hex color code. You can create your own colors using a hex color code picker and adding entries to `COLOR_MAP`.

3.1 Direction Vectors

The constants.py file contains other useful defintions, such as direction vectors that can be used to place objects in the scene. For example, `UP` is a numpy array (0,1,0), which corresponds to 1 unit of distance. To honor the naming convention used in manim I've decided to call the units of distance the MUnit or **m**ath **unit** (this is my own term, not a manim term). Thus the default screen height is 8 MUnits (as defined in constants.py). The default screen width is 14.2 MUnits.
If we are thinking in terms of x-, y-, and z-coordinates, `UP` is a vector pointing along the positive y-axis. `RIGHT` is the array (1,0,0) or a vector pointing along the positive x-axis. The other direction vectors are `LEFT`, `DOWN`, `IN`, and `OUT`. Each vector has a length of 1 MUnit. After creating an instance of an object you can use the `.move_to()` method to move the object to a specific location on the screen. Notice that the direction vectors can be added together (such as `UP+LEFT`) or multiplied by a scalar to scale it up (like `2*RIGHT`). In other words, the direction vectors act like you would expect mathematical vectors to behave. If you want to specify your own vectors, they will need to be numpy arrays with three components. The center edge of each screen side is also defined by vectors `TOP`, `BOTTOM`, `LEFT_SIDE`, and `RIGHT_SIDE`.
The overall scale of the vectors (the relationship between pixels and MUnits) is set by the `FRAME_HEIGHT` variable defined in constants.py. The default value for this is 8. This means you would have to move an object `8*UP` to go from the bottom of the screen to the top of the screen. At this time I don't see a way to change it other than by changing it in `constants.py`.
Mobjects can also be located relative to another object using the `next_to()` method. The command `arrow.next_to(circle,DOWN+LEFT)` places the arrow one MUnit down and one to the left of the circle. The rectangle is then located one MUnit down and one left of the arrow.
The `Circle` class has a `surround()` method that allows you to create a circle that completely encloses another mobject. The size of the circle will be determined by the largest dimension of the mobject surrounded.

3.2 Making Simple Animations

As previously mentioned, the `.add()` method places a mobject on screen at the start of the scene. The `.play()` method can be used animate things in your scene.
The names of the animations, such as `FadeIn` or `GrowFromCenter`, are pretty self-explanatory. What you should notice is that animations play sequentially in the order listed and that if you want multiple animations to occur simultaneously, you should include all those animations in the argument of a

single `.play()` command separated by commas. I'll show you how to use lists of animations to play multiple animations at the same time later.
Things to try:
– Use the `Polygon` class to create other shapes
– Try placing multiple objects on the screen at various locations using `next_to()` and `move_to()`
– Use `surround()` to draw a circle around objects on the screen
– Take a look at the different types of transformations available in `/manim/manimlib/animation/transforms.py`

4.0 Creating Text

There is a special subclass of Mobject called a TextMobject (a **text m**ath **object**) that can be found in tex_mobject.py. Type `python -m manim manim_tutorial_P37.py AddingText -pl` at the command line. Note that the text looks really fuzzy because we are rending the animations at low quality to speed things up. With a small file like this you could render it at full resolution without taking too much time. To do this, replace `-pl` with `-p` (leaving off the **l**ow resolution tag).

```
class AddingText(Scene):

    #Adding text on the screen

    def construct(self):

        my_first_text=TextMobject("Writing with manim is
fun")

        second_line=TextMobject("and easy to do!")

        second_line.next_to(my_first_text,DOWN)

        third_line=TextMobject("for me and you!")

        third_line.next_to(my_first_text,DOWN)



        self.add(my_first_text, second_line)
        self.wait(2)
        self.play(Transform(second_line,third_line))
        self.wait(2)
```

```
        second_line.shift(3*DOWN)
        self.play(ApplyMethod(my_first_text.shift,3*UP))
```

To create a textmobject you must pass it a valid string as an argument. Text rendering is based on Latex so you can use many Latex typesetting features; I'll get into that later. As a subclass of Mobjects, any method such as `move_to()`, `shift()`, and `next_to()` can be used with textmobjects.
The `wait()` method will prevent the next command for the scene from being executed for the desired number of seconds. The default time is 1 second so calling `self.wait()` will wait 1 second before executing the next command in your script.
You should notice that, during the animation, the second line jumps down while the top line gently glides up. This has to do with the fact that we applied the `shift()` method to the second line but we created an animation of the shift to the first line. When animating a `mobject()` method (like `shift()`, `next_to()` or `move_to()`), the `ApplyMethod()` animation is needed inside of a `play()` command. The `shift()` method by itself moves the mobject while using `ApplyMethod()` will animate the motion between the starting and ending points. Notice the arguments of `ApplyMethod()` is a pointer to the method (in this case `my_first_text.shift` without any parentheses) followed by a comma and then the what you would normally include as the argument to the `shift()` method. In other words, `ApplyMethod(my_first_text.shift,3*UP)` will create an animation of shifting `my_first_text` three MUnits up.

4.1 Changing Text

Try running the `AddMoreText` scene.

```
class AddingMoreText(Scene):

    #Playing around with text properties

    def construct(self):

        quote = TextMobject("Imagination is more important
than knowledge")

        quote.set_color(RED)

        quote.to_edge(UP)

        quote2 = TextMobject("A person who never made a
mistake never tried anything new")
```

```
        quote2.set_color(YELLOW)

        author=TextMobject("-Albert Einstein")

        author.scale(0.75)
        author.next_to(quote.get_corner(DOWN+RIGHT),DOWN)

        self.add(quote)
        self.add(author)
        self.wait(2)
        self.play(Transform(quote,quote2),

ApplyMethod(author.move_to,quote2.get_corner(DOWN+RIGHT)+DOW
N+2*LEFT))

        self.play(ApplyMethod(author.scale,1.5))
        author.match_color(quote2)
        self.play(FadeOut(quote))
```

Here we see how to change the color of text using `set_color()`. This uses the same colors discussed in relation to drawing geometric shapes, many of which are defined in the `COLOR_MAP` dictionary in `constants.py`. In addition to setting the color, you can also match the color to another object. In the second to last line of code above we use `match_color()` to change the color of the `author` to match `quote2`.

You can change the size of text using `scale()`. This method scales the mobject up by the numerical factor given. Thus `scale(2)` will double the size of a mobject while `scale(0.3)` will shrink the mobject down to 30% of its current size.

You can align mobjects with the center of the edge of the screen by telling `to_edge()` whether you want the object to be `UP`, `DOWN`, `LEFT`, or `RIGHT`. You can also use `to_corner()`, in which case you need to combine two directions such as `UP+LEFT` to indicate the corner.

Each mobject has a bounding box that indicates the outermost edges of the mobject and you can get the coordinates of the corners of this bounding box using `get_corner()` and specifying a direction. Thus `get_corner(DOWN+LEFT)` will return the location of the lower left corner of a mobject. In our example we find the lower right corner of `quote` and place the author one unit down from that point. Later we move the `author` down and slightly left of `quote2`.

An important thing to note is that the `Transform()` animation still leaves the mobject `quote` on the screen but has just changed its display text and properties to be those of `quote2`. This is why `FadeOut()` refers to `quote` and not `quote2`. However, the corner of `quote` is where it was originally, which is why we have to find the corner of `quote2` to move `author` to the correct location.

Keep in mind that when you use `Tranform`, properties of the mobects involved might not be what you think they are so user beware.
Another useful piece of information is that the `scale()` method changes the size of the objects as it currently is. In other words, using `scale(.5)` followed by `scale(.25)` results in an object that is $0.5 * 0.25 = 0.125$ times the original size and not 0.25 as you might think.
Things to try:
– Compare using `.shift()`, `next_to()`, and `move_to()` to applying them with the `ApplyMethod()` method
– Try using the `to_corner()` method
– Check out `COLOR_MAP` in the constants.py file and change the color of the text

4.2 Rotating and Highlighting Text

The following code will demonstrate how to rotate text and give it some pizzazz. Go ahead and run `python -m manim manim_tutorial_P37.py RotateAndHighlight -p`

```
class RotateAndHighlight(Scene):

    #Rotation of text and highlighting with surrounding
geometries

    def construct(self):

        square=Square(side_length=5,fill_color=YELLOW,
fill_opacity=1)

        label=TextMobject("Text at an angle")

        label.bg=BackgroundRectangle(label,fill_opacity=1)
        label_group=VGroup(label.bg,label)  #Order matters
        label_group.rotate(TAU/8)
        label2=TextMobject("Boxed text",color=BLACK)

label2.bg=SurroundingRectangle(label2,color=BLUE,fill_color=
RED, fill_opacity=.5)
        label2_group=VGroup(label2,label2.bg)
        label2_group.next_to(label_group,DOWN)
        label3=TextMobject("Rainbow")
        label3.scale(2)
        label3.set_color_by_gradient(RED, ORANGE, YELLOW,
GREEN, BLUE, PURPLE)
        label3.to_edge(DOWN)
```

```
        self.add(square)
        self.play(FadeIn(label_group))
        self.play(FadeIn(label2_group))
        self.play(FadeIn(label3))
```

We've added a square in the background to show what `BackgroundRectangle` does. Note that the opacity of the fill color defaults to zero so if you don't define the `fill_opacity` you only see the edges of the square. To create a background rectangle you need to specify the textmobject to apply this method to, as well as the opacity. You can't change the color background to anything but black.

The `VGroup` class allows you to combine multiple mobjects into a single vectorized math object. This allows you to apply any `VMobject` methods to the all elements of the group. You are still able change properties of the original mobjects after they are added to a group. In other words, the original mobjects are not destroyed, the vmobject is just a higher level grouping of the mobjects.

By grouping the text and the background rectangle we can then use `rotate()` to change the orientation of both objects together. Note that `TAU` is equal to 2π (see the Tau Manifesto, which makes some interesting points).

The `next_to()` method can be thought of as a shift relative to some other object so `label2_group.next_to(label_group,DOWN)` places `label2_group` shifted down one unit from `label1_group` (remember that the unit of distance is set by the `FRAME_HEIGHT` variable in `constants.py` and the default screen height is 8 units).

You can create a a color gradient using `set_color_by_gradient()`. Pass the method any number of colors, separated by commas.

Things to play with

– Try changing the fill opacity for both the square and the background rectangle

– Try rotating the background rectangle separately from the the text

– Change the color of label2 to see how it affects the readability of the text

– Change the colors of "Rainbow"

– Place the "Rainbow" text on a different edge of the screen.

5.0 Mathematical Equations

A math animation package wouldn't be much use if you couldn't include nice looking equations. The best way I know of to typeset equations is using LaTeX (LaTeX) , which manim makes use of. If you'd like to learn more about typesetting with LaTeX I'd recommend the tutorials at ShareLaTeX for a basic intro, but you don't need to know much about LaTeX to use manim. You can find a list of commonly used symbols here, which is about all you need to know for manim.

Use manim to run the following scene from the tutorial file to see the following scene:

```
class BasicEquations(Scene):

    #A short script showing how to use Latex commands

    def construct(self):

        eq1=TextMobject("$\\vec{X}_0 \\cdot \\vec{Y}_1 = 3$")

        eq1.shift(2*UP)

        eq2=TexMobject(r"\vec{F}_{net} = \sum_i \vec{F}_i")

        eq2.shift(2*DOWN)



        self.play(Write(eq1))
        self.play(Write(eq2))
```

In LaTeX you normally enclose an equation with dollar signs $$ to denote an equation and that works here as well. The main difference is that, due to how manim parses the text, an extra backslash must be included in front of all LaTeX commands. For instance Greek letters can be created in LaTeX by typing out the name of the letter preceded by a backslash; lower case alpha α would be α, the angle theta θ would be θ. In manim, however, a double backslash is needed so α would be $\\alpha$ and θ would be written as $\\theta$.

David Bieber pointed out in a comment that you can use the raw string literal flag in front of the quote symbol, which removes the need for the double-slashes before Latex symbols. Instead of typing out `eq2=TexMobject("\\vec{F}_{net} = \\sum_i \\vec{F}_i")` you can put the letter `r` in front of the opening quotes and remove the double-slashes so `eq2=TexMobject(r"\vec{F}_{net} = \sum_i \vec{F}_i")`. This makes the code more readable and makes it much easier for those of us who normally type Latex. Going forward I will use the raw tag `r` rather than double-slashes.
You can place a vector arrow over a variable such as $\vec{A}$ using `\vec{A}` (remember you either need to use double-slashes or use the raw string literal tag `r`). Whatever you place inside the brackets will show up on screen with an arrow over it. Subscripts are denoted by the underscore

so $\vec{X}_0$ would be written as $\vec{X}_0$. If the subscript consists of more than a single character you can enclose the subscript in brackets. Thus $\vec{F}_{net}$ in manim would be $\vec{F}_{net}$.
It can get tedious having to always include the dollar signs so the `TexMobject` class (which is different than a `TextMobject` – notice the missing 't' in the middle of the class name) assumes all strings are Latex strings. TEX ($\TeX$) is the typesetting language that LaTeX is based on so I assume `TexMobject` is named for TEX. The main difference between `TextMobject()` and `TexMobject` is the text math object assumes everything is plain text unless you specify an equation with dollar signs while the Tex math object assumes everything is an equation unless you specify something is plain text using `\\text{}`.
When mobjects of any sort are created the default position seems to be the center of the screen. Once created you can use `shift()` or `move_to()` to change the location of the mobjects. For this example above I've moved the equations either two MUnits up or two MUnits down (remember that the MUnit or **m**ath **unit** is what I call the measure of length inside manim). Since the screen height is set to a default of 8 MUnits, a 2 MUnit shift corresponds to about a quarter of the screen height.
The `Write()` method, which is a sublcass of `ShowCreation()`, takes a TextMobject or TexMobject and animates writing the text on the screen. You can also pass a string to `Write()` and it will create the TextMobject for you. `Write()` needs to be inside of `play()` in order to animate it.

5.1 Coloring Equations

```
class ColoringEquations(Scene):

    #Grouping and coloring parts of equations

    def construct(self):

        line1=TexMobject(r"\text{The vector } \vec{F}_{net} \text{ is the net }",r"\text{force }",r"\text{on object of mass }")

        line1.set_color_by_tex("force", BLUE)

        line2=TexMobject("m", "\\text{ and acceleration }", "\\vec{a}", ".  ")
```

```
        line2.set_color_by_tex_to_color_map({

            "m": YELLOW,

            "{a}": RED

        })

        sentence=VGroup(line1,line2)

        sentence.arrange_submobjects(DOWN,
buff=MED_LARGE_BUFF)

        self.play(Write(sentence))
```

For this example we have broken our text into blocks of plain text and equations. This allows us to color parts of the text or equations using either `set_color_by_tex()` or `set_color_by_tex_to_color_map()`. For example, the reason the first sentence is broken up into three parts is so the word `force` can be colored blue. As far as I can tell there isn't an easy way in manim to make changes to part of a string. While you could use slicing of a string, I'm following the convention that Grant Sanderson uses and breaking up text into a list of strings.

The `set_color_by_tex()` method takes the individual string you want colors and the color as arguments. It looks like you only have to specify part of a string to match but the entire string gets colored. For instance, if we type in `line1.set_color_by_tex("F",BLUE)`, the only place a capital F occurs is in the force variable so the first part of this line is blue. If instead we try `line1.set_color_by_tex("e",BLUE)`, the letter `e` appears in several places in `line1` so the entire line ends up blue. If you want to change the color of multiple elements within a list of texmobjects you can use `set_color_by_tex_to_color_map()` and a dictionary. The key for the dictionary should be the text we want colored (or a unique part of the string) and the value should be the desired color.

Notice that, since we are using a texmobject and not a textmobject, we have to enclose plain text in the LaTeX command `\\text{}`. If you don't do this the text is assumed to be part of an equation so the font and spacing are of the text looks funny. Thus "the net force on object of mass" would look like $thenetforceonobjectofmass$. The equation environment doesn't recognize spaces between words, uses a different font, and spaces the letters differently than normal text.

By grouping the two lines together with `VGroup()`, we can use the `arrange_submobjects()` method to space out the two lines. The first argument is the direction you want the objects spaced out and `buff` is the

buffer distance between the mobjects. There are several default buffer distances defined in `constants.py` but you can also a single number. The smallest default buffer is `SMALL_BUFF=0.1` and the largest is `LARGE_BUFF=1`. Although I didn't dive into the code, I think the way the buffers work is as a multiplicative factor of one of the main directional vectors (e.g. `UP`, `DOWN`, `LEFT`, `RIGHT`) so that specifying `SMALL_BUFF` and `LEFT` would be equivalent to $0.1 * (-1, 0, 0) = (-0.1, 0, 0)$.

Things to try:
– Create your own equations using the symbols here.
– Try changing the colors of different parts of the equations
– Use `set_color_by_tex` and match only a part of a full string to see how the entire string is changed
– Write out a sentence as a single string and then use slicing to create texmobjects

6.0 Aligning Text and Using Braces

Let's look at how to use braces to visually group equations or text together but also how to align text elements. We will first write a program to align elements of two equations but in a somewhat clunky fashion; this is not the most elegant way to accomplish this task. After looking at this first version we will rewrite the code in a more concise fashion that lines everything up even better.

You can find the following code in the manim tutorial file.

```
class UsingBraces(Scene):

    #Using braces to group text together

    def construct(self):

        eq1A = TextMobject("4x + 3y")

        eq1B = TextMobject("=")

        eq1C = TextMobject("0")

        eq2A = TextMobject("5x -2y")

        eq2B = TextMobject("=")
```

```
        eq2C = TextMobject("3")

        eq1B.next_to(eq1A,RIGHT)

        eq1C.next_to(eq1B,RIGHT)

        eq2A.shift(DOWN)

        eq2B.shift(DOWN)

        eq2C.shift(DOWN)

        eq2A.align_to(eq1A,LEFT)

        eq2B.align_to(eq1B,LEFT)

        eq2C.align_to(eq1C,LEFT)



        eq_group=VGroup(eq1A,eq2A)

        braces=Brace(eq_group,LEFT)

        eq_text = braces.get_text("A pair of equations")



        self.add(eq1A, eq1B, eq1C)
        self.add(eq2A, eq2B, eq2C)
        self.play(GrowFromCenter(braces),Write(eq_text))
```

To line up parts of the equations on screen we use `next_to()` and `align_to()`. For this example we've broken the equation into smaller parts and then used `next_to()` to place the subparts of each equation next to each other and then `align_to()` to line up the left side of each part of the equation. You can also use `UP`, `DOWN`, and `RIGHT` to align different edges of the mobjects.

We've also added a brace to show how to visually group a set of equations. In order to use the braces we must use `VGroup()` to combine the equations. When we instantiate the braces the first argument is the group and the second argument is where the braces are located relative to the grouping. You can set the text next to the braces using `get_text()` (this is a little confusing naming because you are setting the text, not getting it). This method does not draw the

text on the screen, it is only used to set the location of the text relative to the braces so you will still need to add the text to the screen.

```
class UsingBracesConcise(Scene):

    #A more concise block of code with all columns aligned

    def construct(self):

        eq1_text=["4","x","+","3","y","=","0"]

        eq2_text=["5","x","-","2","y","=","3"]

        eq1_mob=TexMobject(*eq1_text)

        eq2_mob=TexMobject(*eq2_text)

        eq1_mob.set_color_by_tex_to_color_map({

            "x":RED_B,

            "y":GREEN_C

            })

        eq2_mob.set_color_by_tex_to_color_map({

            "x":RED_B,

            "y":GREEN_C

            })

        for i,item in enumerate(eq2_mob):

            item.align_to(eq1_mob[i],LEFT)

        eq1=VGroup(*eq1_mob)

        eq2=VGroup(*eq2_mob)

        eq2.shift(DOWN)
```

```
        eq_group=VGroup(eq1,eq2)

        braces=Brace(eq_group,LEFT)

        eq_text = braces.get_text("A pair of equations")



        self.play(Write(eq1),Write(eq2))
        self.play(GrowFromCenter(braces),Write(eq_text))
```

Here is a (somewhat) more concise version of the previous code. Each equation is written out as a list with each part of the equation as a separate string. This allows more control over the vertical alignment of the parts of the two equations. Inside the `for` loop we use `align_to()` to line up the left edge of the elements in `eq1` and `eq2`.

Notice that when creating the texmobjects that we passed the variable name of the list with an asterisk in front of it `eq1_mob=TexMobject(*eq1_text)`. The asterisk is a Python command to unpack the list and treat the argument as a comma-separated list. Thus `eq1_mob=TexMobject(*eq1_text)` is identical to `eq1_mob=TexMobject("4","x","+","3","y","=","0")`.

Things to try:

– Arrange the equations on the screen

– Add some shapes around your equations.

7.0 Graphing Functions

The easiest way to plot functions is to base your scene class on the `GraphScene()`. The scene creates a set of axes and has methods for creating graphs. One thing that confused me a little at first is that the axes belong to your scene class so you will need to use `self` to access the methods related to the axes. This caused me a few issues when I started out.

We will start off by looking at how to create the axes and graphs but we will come back to look at the `CONFIG{}` dictionary, which is used frequently in manim for initializing many of the class variables.

```
class PlotFunctions(GraphScene):

    CONFIG = {

        "x_min" : -10,

        "x_max" : 10.3,
```

```
        "y_min" : -1.5,

        "y_max" : 1.5,

        "graph_origin" : ORIGIN ,

        "function_color" : RED ,

        "axes_color" : GREEN,

        "x_labeled_nums" :range(-10,12,2),

    }

    def construct(self):

        self.setup_axes(animate=True)


func_graph=self.get_graph(self.func_to_graph,self.function_c
olor)

        func_graph2=self.get_graph(self.func_to_graph2)

        vert_line =
self.get_vertical_line_to_graph(TAU,func_graph,color=YELLOW)

        graph_lab = self.get_graph_label(func_graph, label =
"\\cos(x)")

        graph_lab2=self.get_graph_label(func_graph2,label =
"\\sin(x)", x_val=-10, direction=UP/2)

        two_pi = TexMobject("x = 2 \\pi")

        label_coord =
self.input_to_graph_point(TAU,func_graph)

        two_pi.next_to(label_coord,RIGHT+UP)
```

```
self.play(ShowCreation(func_graph),ShowCreation(func_graph2)
)
       self.play(ShowCreation(vert_line),
ShowCreation(graph_lab),
ShowCreation(graph_lab2),ShowCreation(two_pi))

    def func_to_graph(self,x):
        return np.cos(x)

    def func_to_graph2(self,x):
        return np.sin(x)
```

Under the `construct` method, the first line is `self.setup_axes()` which will create a set of axes on screen. With the exception of whether the creation is animated or not, all other variables for the axes are set using `CONFIG{}`, which I'll explain in a bit. The default values for the `GraphScene()` (which are located in graph_scene.py) are shown below:

```
CONFIG = {

    "x_min": -1,

    "x_max": 10,

    "x_axis_width": 9,

    "x_tick_frequency": 1,

    "x_leftmost_tick": None, # Change if different from
x_min

    "x_labeled_nums": None,

    "x_axis_label": "$x$",

    "y_min": -1,

    "y_max": 10,

    "y_axis_height": 6,

    "y_tick_frequency": 1,
```

```
    "y_bottom_tick": None, # Change if different from y_min

    "y_labeled_nums": None,

    "y_axis_label": "$y$",

    "axes_color": GREY,

    "graph_origin": 2.5 * DOWN + 4 * LEFT,

    "exclude_zero_label": True,

    "num_graph_anchor_points": 25,

    "default_graph_colors": [BLUE, GREEN, YELLOW],

    "default_derivative_color": GREEN,

    "default_input_color": YELLOW,

    "default_riemann_start_color": BLUE,

    "default_riemann_end_color": GREEN,

    "area_opacity": 0.8,

    "num_rects": 50,

}
```

With our example we have changed `x_min`, `x_max`, `y_min`, `y_max`, `graph_origin`, `axes_color`, and `x_labeled_num`. The values assigned in our class take priority over values set by the parent class. Every value that we don't change is automatically assigned the value defined in the parent class. The `x_labeled_num` property takes a list of numbers for labels along the x-axis. We've used `range(-10,12,2)` to generate a list of values from -10 to +10 in steps of 2. One issue I've noted with the y-axis is that setting the min values along either axis to numbers that are not integer multiples of 0.5 results in the tick marks along that axis not being symmetric about zero (e.g. try `y_min` = -1.2). I'm not sure what that is about but it isn't a problem if you stick to integer multiples of 0.5 you don't have any problems.

Once you have the axes set up you can use `self.get_graph()` to graph a function. The argument of `get_graph()` needs to be a pointer to a function, rather than a call to the function itself. In other words, since one of my functions is `func_to_graph()` I should use `self.get_graph(func_to_graph)` without any parentheses after `func_to_graph`.

Rather than defining separate functions for graphing we could use lambda functions. For example, if I define `self.func = lambda x: np.cos(x)` and then use `self.get_graph(self.func)` I will get the same result.

With `get_graph()` you do need to explicitly pass arguments rather than using `CONFIG{}`. The possible arguments, in addition to the function to graph, are `color`, `x_min`, and `x_max`. If you don't specify a color `GraphScene` will cylce through `BLUE`, `GREEN`, and `YELLOW` for successive graphs. Since I didn't specify a color for my second graph it was automatically assigned the first color, `BLUE`.

There is a handy method to draw a vertical line from the x-axis to the graph called `get_vertical_line_to_graph()`. I love that the method naming convention is descriptive enough that you can see what each method does at a glance. Good job, Grant! The arguments for `get_vertical_line_to_graph()` are the x-value where you want the line and the particular graph you want the line drawn to. Note that `get_vertical_line_to_graph()` is a method of the `GraphScene` and not the graph or axes so it is called with `self.get_vertical_line_to_graph()`.

You can label graphs using `get_graph_label()` to set the text associated with the graph. This is similar to the `get_text()` method of the `Braces()` class in that it creates a texmobject at a specific location but does not draw it on the screen; you need to `add` or `play` to show the label. The arguments for `get_graph_label()` are the particular graph you want to add a label to and the text for the label. If you don't specify an x-value and/or direction the label is placed at the end of the graph. The `direction` specifies where, relative to the `x_value` you want the label placed.

There are several other methods associated with the `GraphScene()` that are worth looking at, but I found the `input_to_graph_point()` to be very helpful. By specifying an x-value on the graph, this method will return the coordinate on the screen where that graph point lies. This is handy if you want to place some text or other mobject to call out a particular point on a graph.

7.1 The CONFIG{} Dictionary

Whenever a scene or mobject are created a method called `digest_config()` gets called. This method starts with the class you defined and looks for a dictionary called `self.CONFIG` and compiles a list of all entries in the dictionary. It then goes to the parent class and looks for `self.CONFIG` there and adds those entries. If the method comes across keys that have already been found, it ignores the values from the parent class. `digest_config()` keeps traveling up the hierarchy

to the top parent class, with is `Container()`. Each entry in this dictionary is then assigned a class variable based on the key and value. Thus the dictionary entry `"x_min" : -1` becomes `self.x_min = -1` and so on. Each dictionary entry becomes a class variable that can be accessed by the methods within the class. Understanding all of the `CONFIG{}` entries for a class is crucial to getting the most out of manim. For example, `GraphScene()` has the following `CONFIG{}` entries:

```
class GraphScene(Scene):

    CONFIG = {

        "x_min": -1,

        "x_max": 10,

        "x_axis_width": 9,

        "x_tick_frequency": 1,

        "x_leftmost_tick": None, # Change if different from
x_min

        "x_labeled_nums": None,

        "x_axis_label": "$x$",

        "y_min": -1,

        "y_max": 10,

        "y_axis_height": 6,

        "y_tick_frequency": 1,

        "y_bottom_tick": None, # Change if different from
y_min

        "y_labeled_nums": None,

        "y_axis_label": "$y$",

        "axes_color": GREY,
```

```
        "graph_origin": 2.5 * DOWN + 4 * LEFT,
        "exclude_zero_label": True,
        "num_graph_anchor_points": 25,
        "default_graph_colors": [BLUE, GREEN, YELLOW],
        "default_derivative_color": GREEN,
        "default_input_color": YELLOW,
        "default_riemann_start_color": BLUE,
        "default_riemann_end_color": GREEN,
        "area_opacity": 0.8,
        "num_rects": 50,
    }
```

The parent class for `GraphScene()` (found in the scene.py file) has the following dictionary:

```
class Scene(Container):
    CONFIG = {
        "camera_class": Camera,
        "camera_config": {},
        "frame_duration": LOW_QUALITY_FRAME_DURATION,
        "construct_args": [],
        "skip_animations": False,
        "ignore_waits": False,
        "write_to_movie": False,
```

```
        "save_frames": False,
        "save_pngs": False,
        "pngs_mode": "RGBA",
        "movie_file_extension": ".mp4",
        "name": None,
        "always_continually_update": False,
        "random_seed": 0,
        "start_at_animation_number": None,
        "end_at_animation_number": None,
        "livestreaming": False,
        "to_twitch": False,
        "twitch_key": None,
    }
```

`Container()`, the parent to `Scene` as well as `Mobject`, has no `CONFIG{}` entries. When talking about mobjects, the list of `CONFIG{}` entries can get a little long. I won't go into those right now but it is worth you time to take a look at the hierarchy of some of the mobject subclasses to see what all the properties you can control are.

8.0 More Graphing

Let's take a deeper dive into some of the graphing features in manim.

```
class ExampleApproximation(GraphScene):
    CONFIG = {
        "function" : lambda x : np.cos(x),
```

```
        "function_color" : BLUE,
        "taylor" : [lambda x: 1, lambda x: 1-x**2/2, lambda
x: 1-x**2/math.factorial(2)+x**4/math.factorial(4), lambda
x: 1-x**2/2+x**4/math.factorial(4)-x**6/math.factorial(6),
        lambda x: 1-
x**2/math.factorial(2)+x**4/math.factorial(4)-
x**6/math.factorial(6)+x**8/math.factorial(8), lambda x: 1-
x**2/math.factorial(2)+x**4/math.factorial(4)-
x**6/math.factorial(6)+x**8/math.factorial(8) -
x**10/math.factorial(10)],
        "center_point" : 0,
        "approximation_color" : GREEN,
        "x_min" : -10,
        "x_max" : 10,
        "y_min" : -1,
        "y_max" : 1,
        "graph_origin" : ORIGIN ,
        "x_labeled_nums" :range(-10,12,2),

    }
    def construct(self):
        self.setup_axes(animate=True)
        func_graph = self.get_graph(
            self.function,
            self.function_color,
        )
        approx_graphs = [
            self.get_graph(
                f,
                self.approximation_color
            )
            for f in self.taylor
        ]

        term_num = [
            TexMobject("n = " + str(n),aligned_edge=TOP)
            for n in range(0,8)]
        #[t.to_edge(BOTTOM,buff=SMALL_BUFF) for t in
term_num]


        #term = TexMobject("")
        #term.to_edge(BOTTOM,buff=SMALL_BUFF)
        term = VectorizedPoint(3*DOWN)
```

```
        approx_graph = VectorizedPoint(
            self.input_to_graph_point(self.center_point,
func_graph)
        )

        self.play(
            ShowCreation(func_graph),
        )
        for n,graph in enumerate(approx_graphs):
            self.play(
                Transform(approx_graph, graph, run_time = 2),
                Transform(term,term_num[n])
            )
            self.wait()
```

I wanted to demonstrate how adding higher terms in a Taylor expansion results in better and better agreement with a function. This is similar to what shinigamiphoenix posted here.

The functions to plot are defined as lambda functions in the `CONFIG{}` dictionary. As previously mentioned, manim processes all elements in `CONFIG{}` and turns the dictionary entries into class variables with the key as the variable name. Thus `"function"` can be accessed within my class by calling `self.function` and `"taylor"` can be called with `self.taylor`. If you aren't familiar with lambda functions, check out this post at Python Conquers the Universe.

We create a list of graphs using `get_graph()` and a list comprehension. You can find a nice tutorial on list comprehensions over at datacamp.com. It was only after reading this tutorial that I made the connection between list comprehensions and mathematical notation for definitions of sets (e.g. the set of positive real numbers is $\{x|x \in$ R and $x > 0\}$ or the set of even numbers which is $\{x|x \in$ I and $x \ mod(2) = 0\}$), which made list comprehensions click for me. For each item in the list `self.taylor`, a graph is created with color `self.approximation_color`. We also created a list of TexMobjects to indicate which order of terms are included from the Taylor expansion using a list comprehension.

Since we are going to do successive transformations from a list, it helps to have a blank placeholder on the screen. `term` and `approx_graph` are `VectorizedPoint` instances, which are mobjects that don't display anything on screen. This way we can put the placeholders on the screen without anything appearing, and then transform those mobjects into either the graph or the TexMobjects.

The `enumerate()` command is a useful tool that iterates over a list and also returns the index of the item returned. Thus `for n,graph in enumerate(approx_graphs)` returns the index between 0 and 4 as `n`, and the element within the list as `graph`. This is used to display the corresponding item from `term_num` with each graph.

9.0 Vector Fields

Before diving into draw a vector field, we should set up a Cartesian axes using `NumberPlane()`. This gives you two axes and an underlying grid.
The `CONFIG{}` for the `NumberPlane()` (found in the coordinate_systems.py file) is:

```
class NumberPlane(VMobject):

    CONFIG = {

        "color": BLUE_D,

        "secondary_color": BLUE_E,

        "axes_color": WHITE,

        "secondary_stroke_width": 1,

        # TODO: Allow coordinate center of NumberPlane to not
be at (0, 0)

        "x_radius": None,

        "y_radius": None,

        "x_unit_size": 1,

        "y_unit_size": 1,

        "center_point": ORIGIN,

        "x_line_frequency": 1,

        "y_line_frequency": 1,
```

```
        "secondary_line_ratio": 1,

        "written_coordinate_height": 0.2,

        "propagate_style_to_family": False,

        "make_smooth_after_applying_functions": True,

    }
```

You can change any of these default values by passing a dictionary with new values as keyword arguments. For example, if you want to change the spacing of the grid lines you could change `x_line_frequency` and `y_line_frequency` by defining a dictionary with these variables and then passing the dictionary to `NumberPlane()`. If you want to see the x-axis and y-axis indicated you can use `get_axis_labels()` to draw an `x` and a `y` next to the appropriate axis. See the code below.

```
class DrawAnAxis(Scene):

    CONFIG = { "plane_kwargs" : {

        "x_line_frequency" : 2,

        "y_line_frequency" :2

        }

    }



    def construct(self):

        my_plane = NumberPlane(**self.plane_kwargs)

        my_plane.add(my_plane.get_axis_labels())
        self.add(my_plane)
```

The double asterisk in front of the argument `self.plane_kwargs` lets the class know that this is a dictionary that needs to be unpacked.
I recommend changing the various properties to see what affect they have on the axes and grid. This is the best way to learn what things do.

9.1 A Simple Vector Field

Let's start with a simple vector field; a constant field. We first need to define a set of vector points for each grid point, define the field at each grid point, then create the `Vector()` for the field at each point. Finally we combine all the `Vector()` instances into a `VGroup` to allow us to draw all vector lines with a single command.

```
class SimpleField(Scene):

    CONFIG = {

    "plane_kwargs" : {

        "color" : RED

        },

    }

    def construct(self):

        plane = NumberPlane(**self.plane_kwargs)

        plane.add(plane.get_axis_labels())
        self.add(plane)

        points = [x*RIGHT+y*UP
            for x in np.arange(-5,5,1)
            for y in np.arange(-5,5,1)
            ]

        vec_field = []
        for point in points:
            field = 0.5*RIGHT + 0.5*UP
            result = Vector(field).shift(point)
            vec_field.append(result)

        draw_field = VGroup(*vec_field)


        self.play(ShowCreation(draw_field))
```

After creating the `NumberPlane()` we use a list comprehension to create a list of the location of all grid points. Remember that `RIGHT=np.array(1,0,0)` and `UP=np.array(0,1,0)` so this list comprehension covers all points from (5,5,0) down to (-5,-5,0) in unit step sizes. The last number in `arange()` specifies the step size. Next we create an empty list `vec_field` to hold all of the vectors we are going to create. The `for` loop goes through each grid location in `points` and creates a vector whose length and direction are defined by `field`. It is inefficient to keep defining `field` each time through the loop but we are setting things up for later.
The `shift(point)` command moves the vector to the grid location defined by `point`. These results are then appended to a list. After going through the `for` loop, all of the vectors are grouped together in a single `VGroup` called `draw_field`. The only reason for doing this is that you can then add `draw_field` using a single `add` or `play` command. You could have included `self.add(result)` inside each iteration of the `for` loop instead of showing the creation of `draw_field`, but using the `VGroup` feels cleaner.

9.2 A Variable Vector Field

For a slightly more interesting field we will look at the electric field due to a postive point charge. The electric field is:

$$\vec{E} = \frac{1}{4\pi\epsilon_0}\frac{q}{r^3}\vec{r}$$

where q_1 is the charge on the point charge, $\vec{r}$ is the distance vector between the charge and the observation point, and r is the magnitude of that vector. The constant out front $\frac{1}{4\pi\epsilon_0} = 9 \times 10^9 Nm^2/C^2$ is essentially a conversion factor. For our purposes we will set all constants equal to zero and just look at

$$\vec{E} = \frac{1}{r^3}\vec{r}.$$

```
class FieldWithAxes(Scene):

    CONFIG = {

    "plane_kwargs" : {

        "color" : RED_B
```

```
        },

    "point_charge_loc" : 0.5*RIGHT-1.5*UP,

    }

    def construct(self):

        plane = NumberPlane(**self.plane_kwargs)

        plane.add(plane.get_axis_labels())
        self.add(plane)

        field = VGroup(*[self.calc_field(x*RIGHT+y*UP)
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
            ])

        self.play(ShowCreation(field))


    def calc_field(self,point):
        #This calculates the field at a single point.
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        #efield = np.array((-y,x,0))/math.sqrt(x**2+y**2)
#Try one of these two fields
        #efield = np.array(( -2*(y%2)+1 , -2*(x%2)+1 , 0 ))/3
#Try one of these two fields
        return Vector(efield).shift(point)
```

The location of the point charge is set in `CONFIG{}`. To create the vector field we've condensed the previous code. We use a list comprehension and the function `calc_field()` as the argument of `VGroup()`. The `calc_field()` function defines the field to calculate. To make the formulas a little easier to read we unpack the x- and y-coordinates from the `point` vector and the `self.point_charge_loc` vector. The code `x,y=point[:2]` is equivalent to `x=point[0]` and `y=point[1]`.

The `fade(0.9)` method sets the opacity of the lines to be one minus the fade level (so in this case the opacity is set to 0.1). This was done to make it easier to see the tiny field arrows farther from the charge location.

Things to try:
– Change each of the elements in `CONFIG{}` for `NumberPlane()` to see what affect they have on the axes and grid lines.
– Calculate different fields
– Try `efield = np.array((-y,x,0))/math.sqrt(x**2+y**2)`
– Try `efield = np.array(( -2*(y%2)+1 , -2*(x%2)+1 , 0 ))/3`
– Come up with your own equation

10.0 Field of a Moving Charge

There was a question over on Reddit about how to create the electric field of a moving charge. Since that is something I will want to do at some point I figured it would be fun to give it a try.
Before creating a changing field, I thought I'd start with moving charges around. I know I saw this in one of the videos so I can start with working code and modify it to my needs. Here is what I came up with:

```
class MovingCharges(Scene):

    CONFIG = {

    "plane_kwargs" : {

        "color" : RED_B

        },

    "point_charge_loc" : 0.5*RIGHT-1.5*UP,

    }

    def construct(self):

        plane = NumberPlane(**self.plane_kwargs)

        plane.add(plane.get_axis_labels())
        self.add(plane)

        field = VGroup(*[self.calc_field(x*RIGHT+y*UP)
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
            ])
```

```
        self.field=field
        source_charge =
self.Positron().move_to(self.point_charge_loc)
        self.play(FadeIn(source_charge))
        self.play(ShowCreation(field))
        self.moving_charge()

    def calc_field(self,point):
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        return Vector(efield).shift(point)

    def moving_charge(self):
        numb_charges=4
        possible_points = [v.get_start() for v in self.field]
        points = random.sample(possible_points, numb_charges)
        particles = VGroup(*[
            self.Positron().move_to(point)
            for point in points
        ])
        for particle in particles:
            particle.velocity = np.array((0,0,0))

        self.play(FadeIn(particles))
        self.moving_particles = particles
        self.add_foreground_mobjects(self.moving_particles )
        self.always_continually_update = True
        self.wait(10)

    def field_at_point(self,point):
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        return efield

    def continual_update(self, *args, **kwargs):
        if hasattr(self, "moving_particles"):
            dt = self.frame_duration
            for p in self.moving_particles:
                accel = self.field_at_point(p.get_center())
                p.velocity = p.velocity + accel*dt
```

```
            p.shift(p.velocity*dt)


class Positron(Circle):
    CONFIG = {
    "radius" : 0.2,
    "stroke_width" : 3,
    "color" : RED,
    "fill_color" : RED,
    "fill_opacity" : 0.5,
    }
    def __init__(self, **kwargs):
        Circle.__init__(self, **kwargs)
        plus = TexMobject("+")
        plus.scale(0.7)
        plus.move_to(self)
        self.add(plus)
```

The most important method here is `continual_update()`. This method updates the screen for each frame during the entire scene. This differs from the various transformations that rely on the `play()` method in that the transformations occur over a short time interval, usually on the order of a few seconds while the continual methods continue to run for the entire scene. If we want a particle to move across the screen we might be tempted to use something like `self.play(ApplyMethod(particle1.shift,5*LEFT))` but it would be challenging to control the timing of other transformations going on at the same time. The `continual_update()` allows you to animate things in the background while still controlling the timing of other transformations.
Since I know I will be using charged particles in my videos I've written a `Positron` class to create positively charged particles. The positron is the positive antiparticle of the electron. Why didn't I make it a proton? Because the proton is roughly 2000 times more massive and I want similarly sized particles for what I want to do.
We've reused the code from a previous post about electric fields but we've added methods to create the charged particles and move them around. `moving_charge()` is what creates positrons by randomly selecting a field point (`possible_points = [v.get_start() for v in self.field]` is a list of the locations of the tails of all field vectors) and then selects `numb_charges` points to create particles at. Note that the randomly generated charges don't react to one another, which I find disturbing to watch because it isn't physical. `particles` is a vectorized mobject group that contains all of the moving charges with initial velocities set to zero (`particle.velocity = np.array((0,0,0))`). We could have simplified the code by only using one particle at a set location, but we'll need multiple charges later on. The charges are then added to the screen

(`self.play(FadeIn(particles))`) and assigned to a class variable that is needed in `continual_update` (`self.moving_particles = particles`). Mobjects are drawn in the order they are added to the screen but you can place certain mobjects in the foreground to insure they always remain drawn on top of other objects by using `add_foreground_mobjects()`. It is kind of like layers in Photoshop or similar software except each mobject is in its own layer. This has to do with the fact that manim keeps all mobjects drawn on the screen in a list and draws them in the order they are listed. There is no equivalent background mobject method, but you can send mobjects to the front or back layers with `bring_to_front()` and `bring_to_back()`.
Next we tell manim to continually update things in the background (`self.always_continually_update = True`) and then wait ten seconds. It is important to set the `wait()` command because the continual update only runs as long as their are animation elements (`play()` commands) or `wait()` commands in the animation queue.
The `field_at_point()` method duplicates some of the earlier code but is used to return a numerical vector (a numpy 3-element array) rather than a mobject `Vector`, which is what `calc_field()` returns. It took me an embarassing amount of time to figure out why I couldn't just use `calc_field()` to find the force vector.
The `continual_update()` method is called each frame when the scene is being composed. The first line, `if hasattr(self, "moving_particles"):` prevents the rest of the code running and throwing and error if you haven't created `self.moving_particles`. The frame duration is either 1/15, 1/30, or 1/60 of a second, depending on whether your video is low, medium, or production quality (i.e. whether you include -l, -m, or no command line argument when extracting the scene). We run through the list of all moving particles (`for p in self.moving_particles:`) and then calculate the acceleration due to the electric field at the location of each particle (`vect = self.field_at_point(p.get_center())`). `p.get_center()` returns the vector location of each particle `p`. The velocity is updated using $\vec{v}_f = \vec{v}_i + a\Delta t$ and then the particle is shifted over the distance $\vec{v}_f \Delta t$.

10.1 Updating the Electric Field of a Moving Charge

Now we've got some experience moving things around on the screen so we can move on to calculating the field due to the particle. We will reuse much of the code from our previous program, with a few changes.

```
class FieldOfMovingCharge(Scene):
```

```
    CONFIG = {

    "plane_kwargs" : {

        "color" : RED_B

        },

    "point_charge_start_loc" : 5.5*LEFT-1.5*UP,

    }

    def construct(self):

        plane = NumberPlane(**self.plane_kwargs)

        #plane.main_lines.fade(.9)
        plane.add(plane.get_axis_labels())
        self.add(plane)

        field =
VGroup(*[self.create_vect_field(self.point_charge_start_loc,
x*RIGHT+y*UP)
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
            ])
        self.field=field
        self.source_charge =
self.Positron().move_to(self.point_charge_start_loc)
        self.source_charge.velocity = np.array((1,0,0))
        self.play(FadeIn(self.source_charge))
        self.play(ShowCreation(field))
        self.moving_charge()

    def
create_vect_field(self,source_charge,observation_point):
        return
Vector(self.calc_field(source_charge,observation_point)).shi
ft(observation_point)

    def calc_field(self,source_point,observation_point):
        x,y,z = observation_point
        Rx,Ry,Rz = source_point
```

```
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2 + (z-Rz)**2)
        if r<0.0000001:   #Prevent divide by zero  ##Note:
This won't work - fix this
            efield = np.array((0,0,0))
        else:
            efield = (observation_point - source_point)/r**3
        return efield



    def moving_charge(self):
        numb_charges=3
        possible_points = [v.get_start() for v in self.field]
        points = random.sample(possible_points, numb_charges)
        particles = VGroup(self.source_charge, *[
            self.Positron().move_to(point)
            for point in points
        ])
        for particle in particles[1:]:
            particle.velocity = np.array((0,0,0))
        self.play(FadeIn(particles[1:]))
        self.moving_particles = particles
        self.add_foreground_mobjects(self.moving_particles )
        self.always_continually_update = True
        self.wait(10)


    def continual_update(self, *args, **kwargs):
        Scene.continual_update(self, *args, **kwargs)
        if hasattr(self, "moving_particles"):
            dt = self.frame_duration

            for v in self.field:
                field_vect=np.zeros(3)
                for p in self.moving_particles:
                    field_vect = field_vect +
self.calc_field(p.get_center(), v.get_start())
                v.put_start_and_end_on(v.get_start(),
field_vect+v.get_start())

            for p in self.moving_particles:
                accel = np.zeros(3)
                p.velocity = p.velocity + accel*dt
                p.shift(p.velocity*dt)
```

```
   class Positron(Circle):
      CONFIG = {
      "radius" : 0.2,
      "stroke_width" : 3,
      "color" : RED,
      "fill_color" : RED,
      "fill_opacity" : 0.5,
      }
      def __init__(self, **kwargs):
         Circle.__init__(self, **kwargs)
         plus = TexMobject("+")
         plus.scale(0.7)
         plus.move_to(self)
         self.add(plus)
```

One change we've made is to let `calc_field()` return a numpy vector rather than a mobject `Vector`. This does mean adding in `create_vect_field()` to create the mobjects from the numpy vectors.

Since we want our source charge to be able to move we have to add that source charge to the `particles` list. Thus the Vgroup we create includes that source charge plus the randomly generated charges using `particles = VGroup(self.source_charge, *[self.Positron().move_to(point) for point in points])`. Remember that the asterisk in front of the list lets Python know that each element in the list should be broken out and treated as a separate argument for the `VGroup()` class. To help make sense of this line of code we can break it out into a less elegant form:

```
list_of_random_charges=[]

for point in points:

   new_charge = self.Positron().move_toe(point)

   list_of_random_charges.append(new_charge)

   particles = VGroup(self.source_charge,
list_of_random_charges[0],

   list_of_random_charges[1], list_of_random_charges[2])
```

Thus one line of code replaces several lines. The reason we use `particles[1:]` in the code defining the velocity and fading in the particles

is that the source charge already has a velocity and is on screen so we don't want to redefine the velocity or have it fade in again (which makes it blink).
In `continual_update()` we now need to calculate the field at each grid point at each time step. First we cycle through each field point (`for v in self.field:`). Since we want to add up the fields from several charges, we set the field vector to zero (`field_vect=np.zeros(3)`) and then add up the fields at that point due to each charge (`field_vect = field_vect + self.calc_field(p.get_center(), v.get_start())`). We need to redraw the field vectors by specifying the start and end points of the vector. The start point is the initial grid point where the vector starts (`v.get_start()`) and the tip of the arrow is a distance equal to the field vector plus the starting point (`field_vect+v.get_start()`).
I don't have the particles react to the fields of the other particles. This looks very unrealistic to me but it should be easy enough to implement. I just wanted to put out a post that lays out the basics of how to get the field lines working.

11 Three Dimensional Scenes

This is the first place where this tutorial diverges from the previous series. This is due to the fact that many of the changes in manim after switching the Python 3.7 (at least that I've seen) seem to be focused on improving the 3D capabilities of manim.

This might be a good time to explain how I have been figuring manim out. The first thing I do is go to the active_projects directory, find a file related to an interesting video, and pick it apart. As long as you pick an active project you know it should compile without any problems. I will then copy and paste a single scene into another file and start stripping out components of the code until I have a simple working example of the thing I'm interested in. I'll start playing around with some of the CONFIG entries and start adding in features. I find it very useful to search the github site for other scenes that have used similar commands to determine what sort of options are available. The naming convention that Grant Sanderson uses is good enough that you can usually figure out what things do with only a little trial and error.
The CONFIG dictionary for the `ThreeDScene` class is:

```
class ThreeDScene(Scene):

    CONFIG = {

        "camera_class": ThreeDCamera,

        "ambient_camera_rotation": None,
```

```
        "default_angled_camera_orientation_kwargs": {

        "phi": 70 * DEGREES,

        "theta": -135 * DEGREES,

        }

    }
```

The methods you can call in a `ThreeDScene` (which can be found in three_d_scene.py) are:

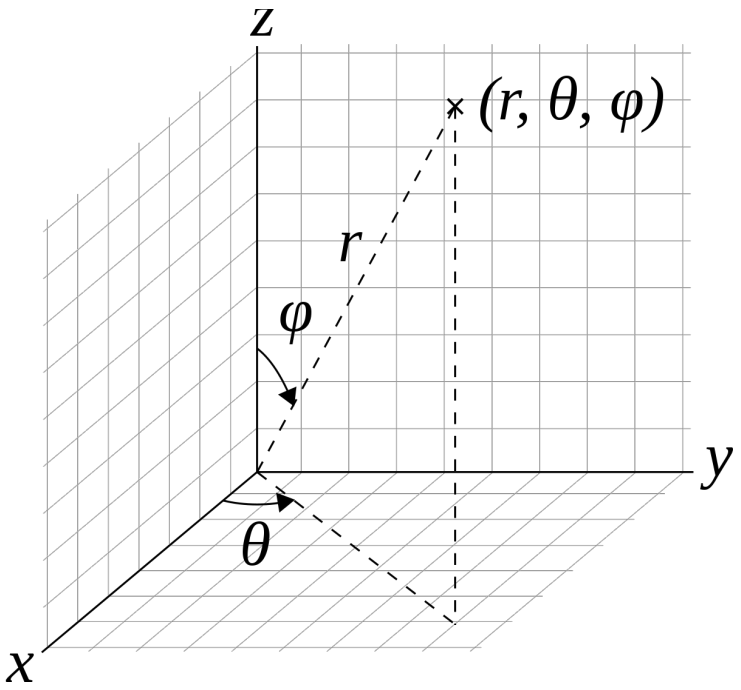
- `set_camera_orientation`
- `begin_ambient_camera_rotation`
- `stop_ambient_camera_rotation`
- `move_camera`

There are a few other methods but we'll focus on these for now. To start with we will create a normal 2D scene but use the 3D camera to rotate around. We'll reuse the code from the previous section:

```
class ExampleThreeD(ThreeDScene):

    CONFIG = {

    "plane_kwargs" : {

        "color" : RED_B

        },

    "point_charge_loc" : 0.5*RIGHT-1.5*UP,

    }

    def construct(self):

        plane = NumberPlane(**self.plane_kwargs)

        plane.add(plane.get_axis_labels())
        self.add(plane)

        field2D = VGroup(*[self.calc_field2D(x*RIGHT+y*UP)
```

```
            for x in np.arange(-9,9,1)
            for y in np.arange(-5,5,1)
            ])

        self.set_camera_orientation(phi=PI/3,gamma=PI/5)
        self.play(ShowCreation(field2D))
        self.wait()
        #self.move_camera(gamma=0,run_time=1)  #currently
broken in manim
        self.move_camera(phi=3/4*PI, theta=-PI/2)
        self.begin_ambient_camera_rotation(rate=0.1)
        self.wait(6)

    def calc_field2D(self,point):
        x,y = point[:2]
        Rx,Ry = self.point_charge_loc[:2]
        r = math.sqrt((x-Rx)**2 + (y-Ry)**2)
        efield = (point - self.point_charge_loc)/r**3
        return Vector(efield).shift(point)
```

By defining our scene as a subclass of `ThreeDScene`, we gain access to the 3D camera options. Then it is just a matter of moving the camera around.



By Dmcq – Own work, CC BY-SA 3.0, Link

The original orientation of the camera is set using `set_camera_orientation()` which takes θ and ϕ. You can also set the distance from the camera to the origin using the keyword argument `distance`. There is also the option to change `gamma` (the greek letter γ), which is one of the Euler angles. Changing `gamma` causes the camera to rotate about an axis

through the center of the lens, allowing us to change which direction is horizontal on the screen. Note that if we use `set_camera_orientation` in the middle of the scene the camera will jump to the new orientation.
One thing to keep in mind when setting the camera orientation is that, although the camera itself is pointing towards the origin, the angles θ, ϕ, and γ are measured from an axis set up at the center of the camera. For some reason $\phi = 0$ and $\theta = 0$ corresponds to the positive y-axis being ot the right and the positive x-axis being down. This is
why $\phi = 0$ and $\theta = -\pi/2$ corresponds to the normal 2D orientation with the x-axis pointing right and the y-axis pointing up on the screen. If you set $\theta = \pi/2$ we flip the screen over.

To get the camera to smoothly pan we use `move_camera()`, which has the same arguments as `set_camera_orientation()`. Edit: It turns out you can specify `run_time=4` , for example, to have the `move_camera()` operation take 4 seconds.
We can set the camera rotating about the z-axis by
calling `begin_ambient_camera_rotation()` and we can specify the rate at which it is rotating. I believe the `rate` is measured in radians per second.
If you are more familiar with degrees you can multiply your angle by `DEGREES`, so the default camera orientation would
be `self.set_camera_orientation(phi=0*DEGREES, theta=-90*DEGREES)`.
Things to try
– Play around with the angles `phi`, `theta`, and `gamma` to get a feel for how the camera is oriented
– Create a series of 2D mobjects and use the 3D camera to zoom around the mobjects.

12.0 Working with SVG Files

The PiCreatures in 3B1B are scalable vector graphics (svg) files. manim has an `SVGMobject` class that can import svg files. To play around with using svg images in manim, I've created a couple of figures using (Inkscape)[https://inkscape.org/en/], an open source vector graphics package. I wanted to try to make a stick figure wave in manim so I created two figures, one normal and one with the hand waving. You can get the svg files I've used at the end of this post. Place them in
the `\media\designs\svg_images\` folder.
The code I used to import the stick figure was based on the PiCreature code located in \manimlib\for_3b1b_videos\pi_creatures.py. My code looks like:

```
HEAD_INDEX = 0

BODY_INDEX = 1
```

```
ARMS_INDEX = 2

LEGS_INDEX = 3



class StickMan(SVGMobject):

    CONFIG = {

        "color" : BLUE_E,

        "file_name_prefix": "stick_man",

        "stroke_width" : 2,

        "stroke_color" : WHITE,

        "fill_opacity" : 1.0,

        "height" : 3,

    }

    def __init__(self, mode = "plain", **kwargs):

        digest_config(self, kwargs)

        self.mode = mode
        self.parts_named = False
        try:
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
                "%s_%s.svg" % (self.file_name_prefix, mode)
            )
            SVGMobject.__init__(self, file_name=svg_file,
**kwargs)
        except:
            warnings.warn("Noesign with mode %s" %
                          (self.file_name_prefix, mode))
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
```

```
            "stick_man_plain.svg",
        )
        SVGMobject.__init__(self, mode="plain",
file_name=svg_file, **kwargs)


    def name_parts(self):
        self.head = self.submobjects[HEAD_INDEX]
        self.body = self.submobjects[BODY_INDEX]
        self.arms = self.submobjects[ARMS_INDEX]
        self.legs = self.submobjects[LEGS_INDEX]
        self.parts_named = True

    def init_colors(self):
        SVGMobject.init_colors(self)
        if not self.parts_named:
            self.name_parts()
        self.head.set_fill(self.color, opacity = 1)
        self.body.set_fill(RED, opacity = 1)
        self.arms.set_fill(YELLOW, opacity = 1)
        self.legs.set_fill(BLUE, opacity = 1)
        return self
```

I'm sure I could trim the code more but I just wanted something that would work without too much debugging. I've named the two files for the stick man as `stick_man_plain.svg` and `stick_man_wave.svg`. If you don't specify a mode when you instantiate the `StickMan` class, it will try load a file with the prefix specified in the CONFIG dictionary (in this example it is `stick_man`) and a suffix `_plain`. The `mode` variable can be used to specify related files. For example, the svg file with the stick man waving is called `stick_man_wave.svg` so if I specify the mode of `wave` this class will load that file. I can create an instance of the stick man using the waving figure using `StickMan("wave")`. Below is the code for the scene to make the stick man wave and you can find the svg files I used in my Github repository here:

```
class Waving(Scene):


    def construct(self):


        start_man = StickMan()


        plain_man = StickMan()


        waving_man = StickMan("wave")
```

```
        self.add(start_man)
        self.wait()
        self.play(Transform(start_man,waving_man))
        self.play(Transform(start_man,plain_man))

        self.wait()
```

The reason I create two instances of the `StickMan()` is because I am transforming `start_man` but want the image to end up back looking like the original figure.

Two things to note. (1) The `stroke_width` and `stroke_color` for PiCreatures are set to not draw the outline of objects. If you want to see lines or the outlines of shapes you will need to set these values to something visible (i.e. non-zero `stroke_width` and a `stroke_color` that is different than the background). (2) Lines in svg are labeled as paths. The way manim deals with paths it to treat them as closed shapes. That means that if I don't set the `opacity` to zero for a line, I will see an enclosed shape. See the video below where I've set the `fill_opacity` in the `init_colors` method for everything to 1.

Although I haven't delved into the manim code, I think all manim looks at is the outlines of the shapes and not the filling.

I created a second scene just to make sure I had a handle on the scalable vector graphics import. When creating your own images, you will need to open the `.svg` file in a text editor to determine the indices for each submobject. manim imports each svg entity (e.g. a path, ellipse, box, or other shape) as a single submobject, and you will need to determine the ordering of those items in the parent SVGMobject class. I created a couple of shapes (a circle connected by lines to a pair of squares). The relevant part of the svg file for this is shown here (there is a lot more metadata in the file I left out):

```
1       <br />
```

The file contains a circle, two paths, and two rectangles. Thus, when imported into manim, the circle will be the first submobject (index of 0), the two paths or lines will be the second and third submobject (indices 1 and 2) and the two squares will be the fourth and fifth submobject (indices 3 and 4). The class I used for this circle and square drawing is

```
class CirclesAndSquares(SVGMobject):

    CONFIG = {

        "color" : BLUE_E,
```

```
        "file_name_prefix": "circles_and_squares",

        "stroke_width" : 2,

        "stroke_color" : WHITE,

        "fill_opacity" : 1.0,

        "height" : 3,

        "start_corner" : None,

        "circle_index" : 0,

        "line1_index" :1,

        "line2_index" : 2,

        "square1_index" : 3,

        "square2_index" : 4,

    }

    def __init__(self, mode = "plain", **kwargs):

        digest_config(self, kwargs)

        self.mode = mode
        self.parts_named = False
        try:
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
                "%s_%s.svg" % (self.file_name_prefix, mode)
            )
            SVGMobject.__init__(self, file_name=svg_file,
**kwargs)
        except:
            warnings.warn("Noesign with mode %s" %
                          (self.file_name_prefix, mode))
            svg_file = os.path.join(
                SVG_IMAGE_DIR,
```

```
            "circles_and_squares_plain.svg",
        )
        SVGMobject.__init__(self, mode="plain",
file_name=svg_file, **kwargs)


    def name_parts(self):
        self.circle = self.submobjects[self.circle_index]
        self.line1 = self.submobjects[self.line1_index]
        self.line2 = self.submobjects[self.line2_index]
        self.square1 = self.submobjects[self.square1_index]
        self.square2 = self.submobjects[self.square2_index]
        self.parts_named = True

    def init_colors(self):
        SVGMobject.init_colors(self)
        self.name_parts()
        self.circle.set_fill(RED, opacity = 1)
        self.line1.set_fill(self.color, opacity = 0)
        self.line2.set_fill(self.color, opacity = 0)
        self.square1.set_fill(GREEN, opacity = 1)
        self.square2.set_fill(BLUE, opacity = 1)
        return self
```

I've used the order of the different elements in the svg file to label the indices in my `CONFIG` dictionary at the start of the class. The code to display this on the screen is

```
class SVGCircleAndSquare(Scene):

    def construct(self):

        thingy = CirclesAndSquares()

        self.add(thingy)
        self.wait()
```

I know I can trim the code down for the svg class but I'll save that for another day.