Visual Data Storytelling with Python

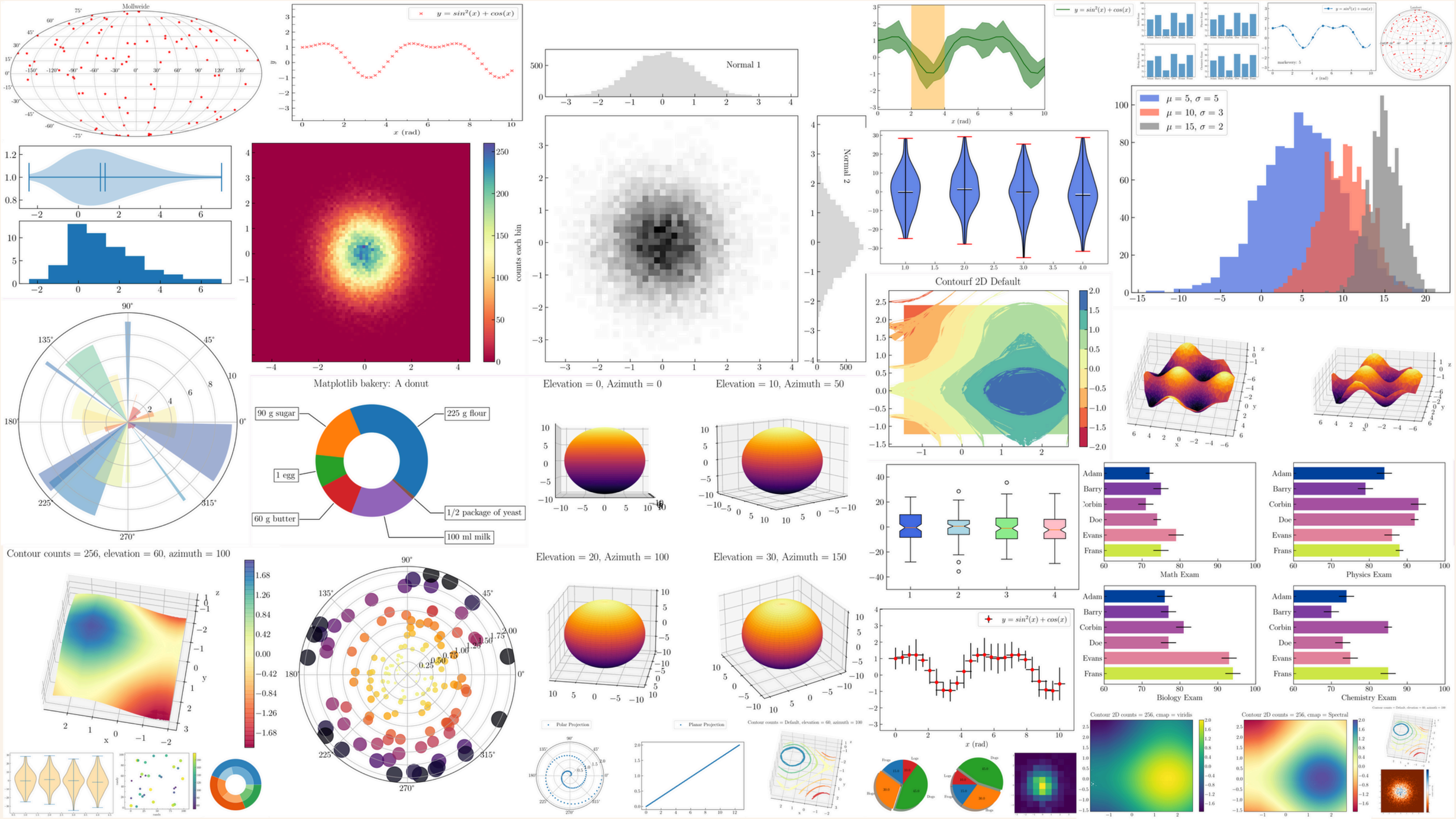
Neeraj Pandey

@n4jp4y

#PyConPL2024

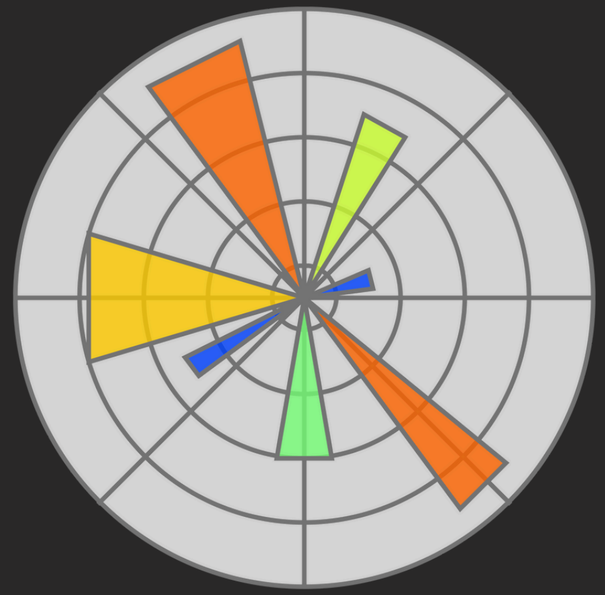
Talk Contents

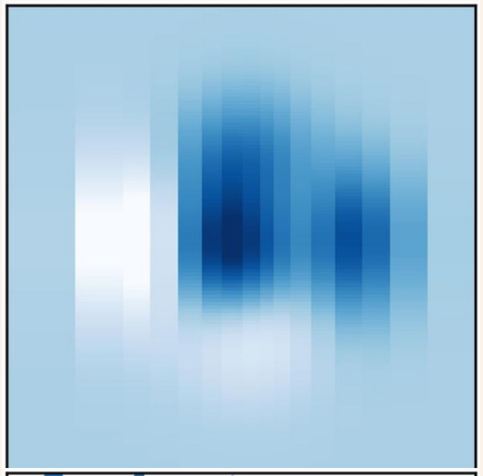
1. Animations in Matplotlib & Manim
2. Blender & it’s Python API
3. 2D & 3D Graphics
4. Lots of graphs & animations

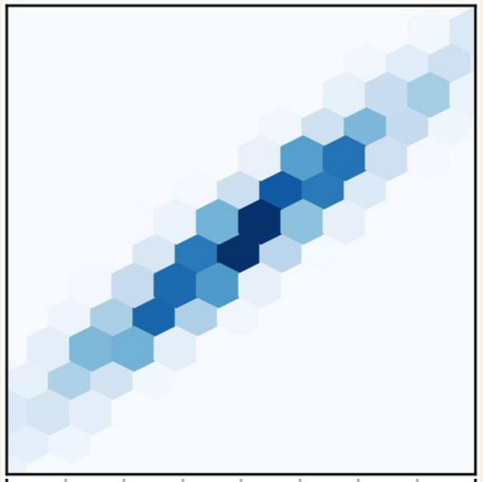


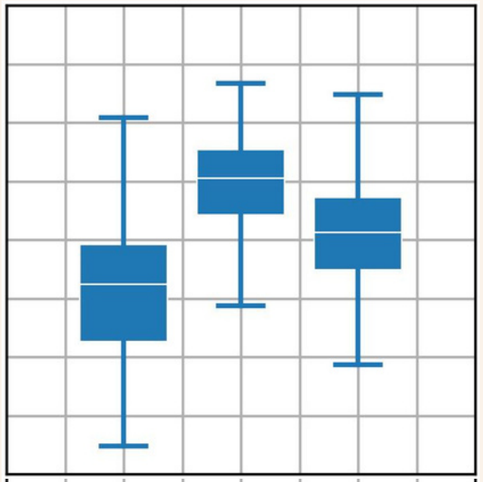
source: towardsdatascience.com/visualizations-with-matplotlib-part-1-c9651008b6b8

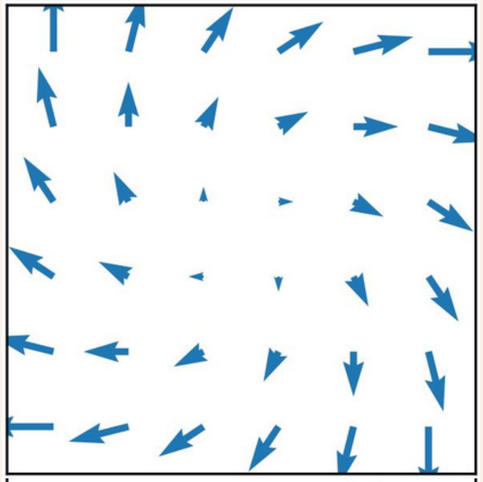
Matplotlib

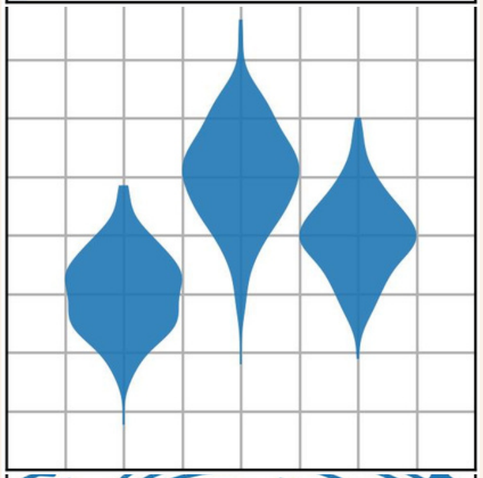


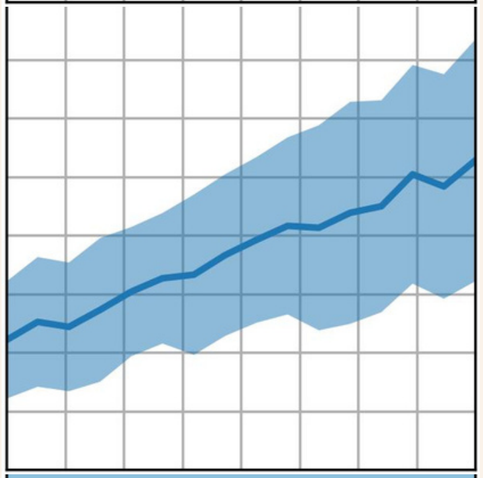


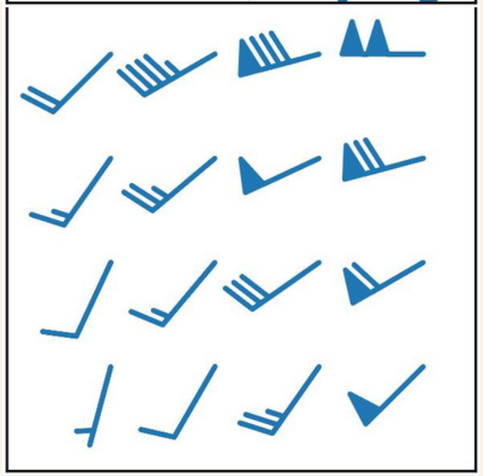


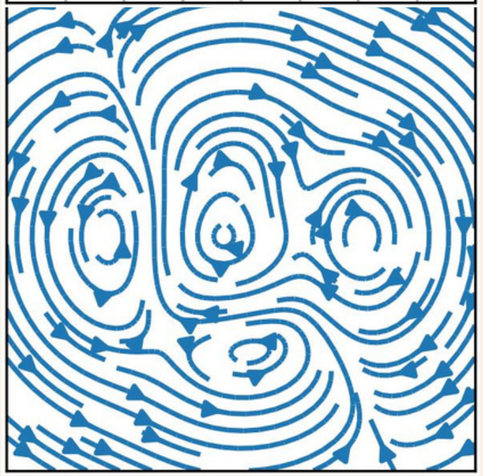


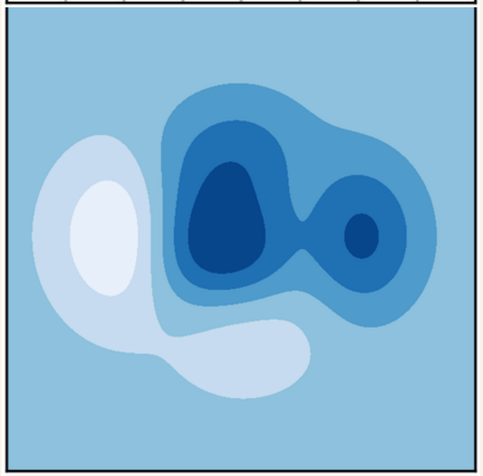


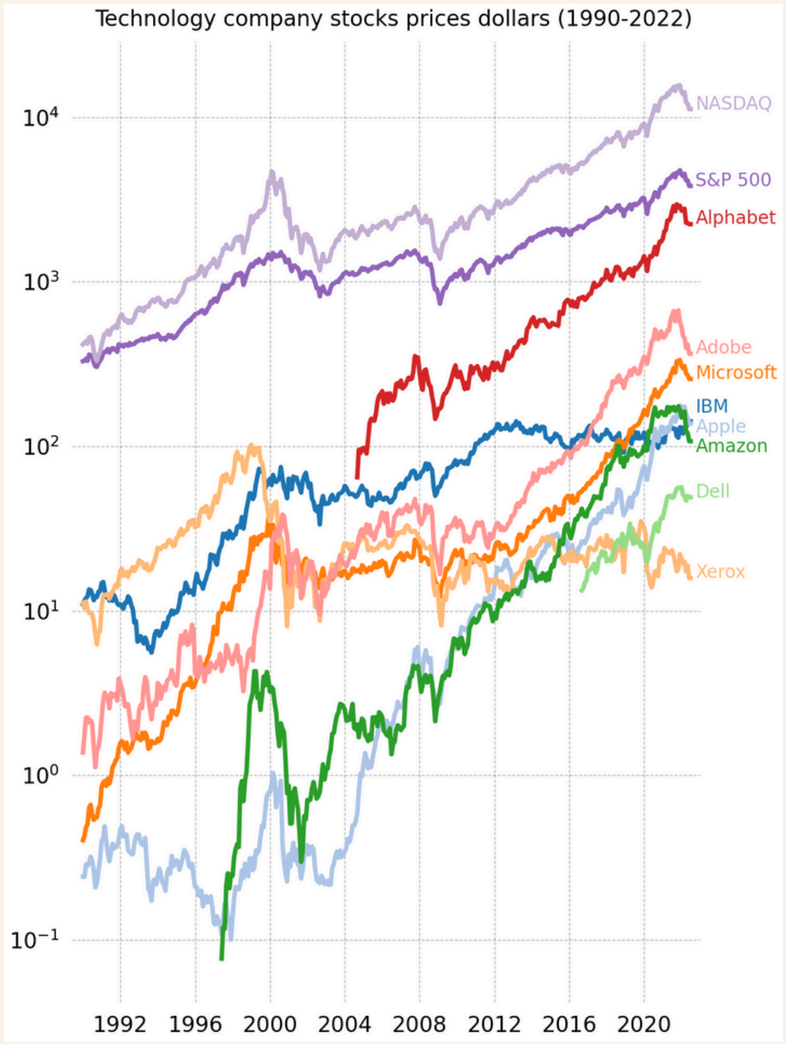






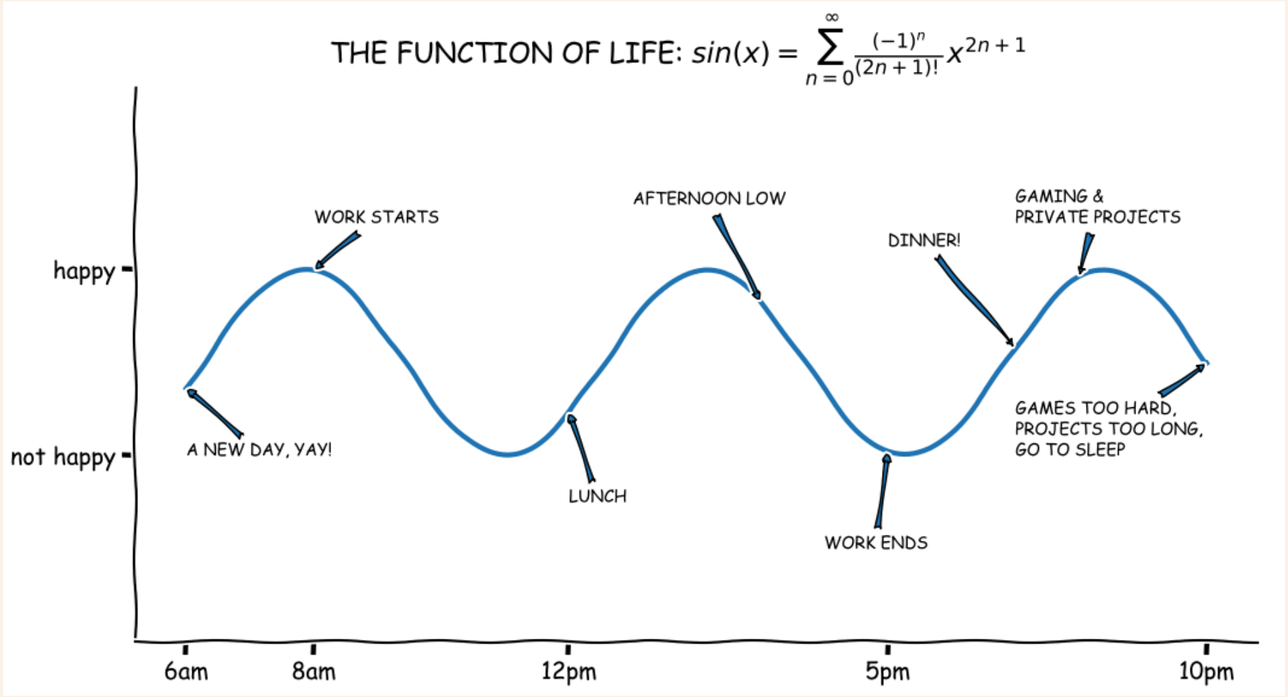


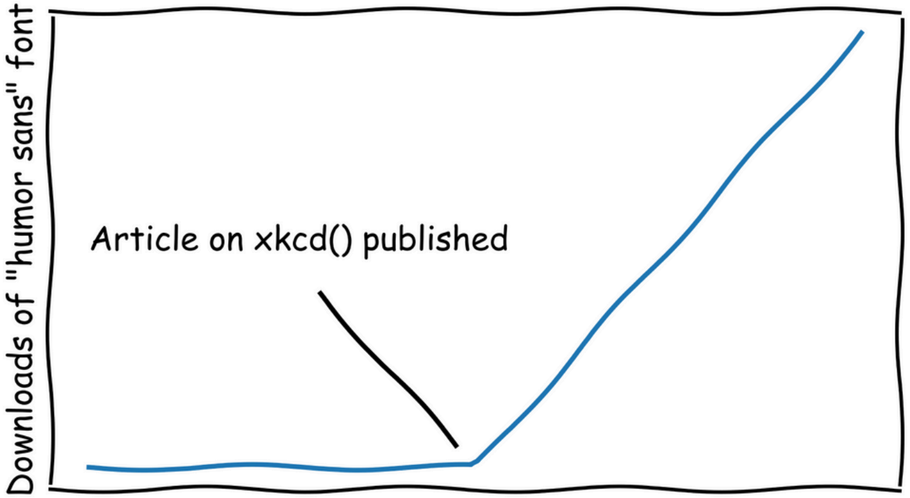




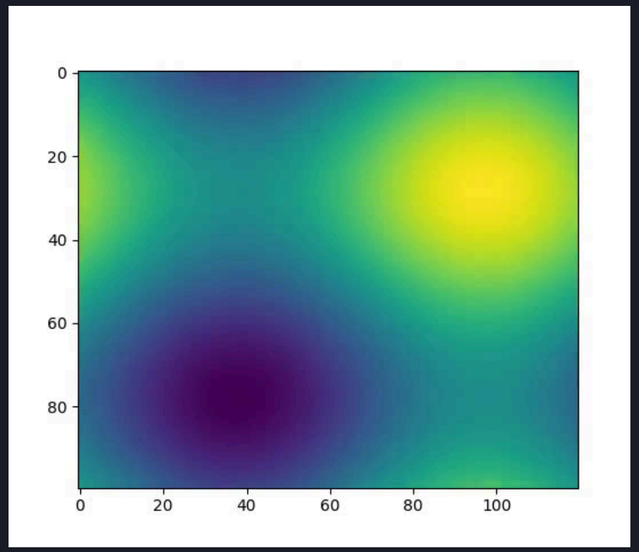
source: https://matplotlib.org/stable/gallery/index.html

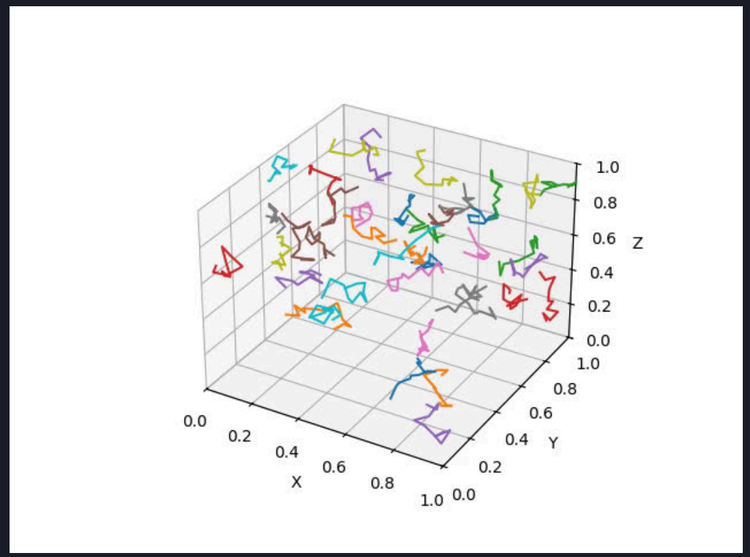
```python
with plt.xkcd():
    # This figure will be in XKCD-style
    fig1 = plt.figure()
    # ...

# This figure will be in regular style
fig2 = plt.figure()
```

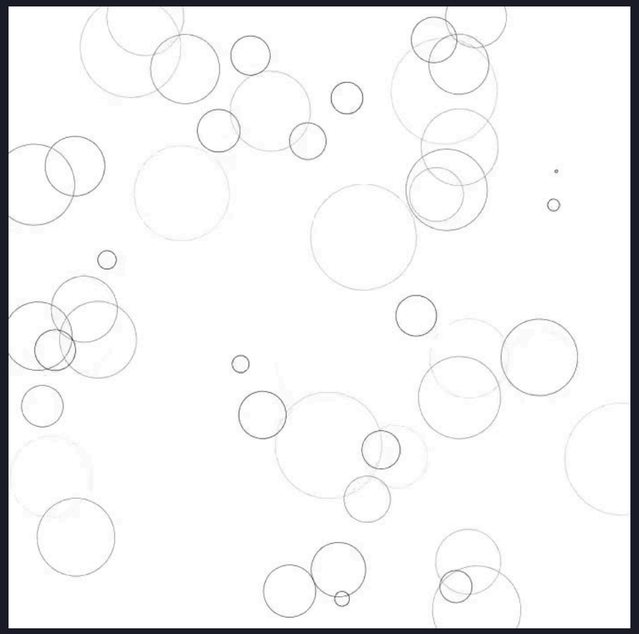


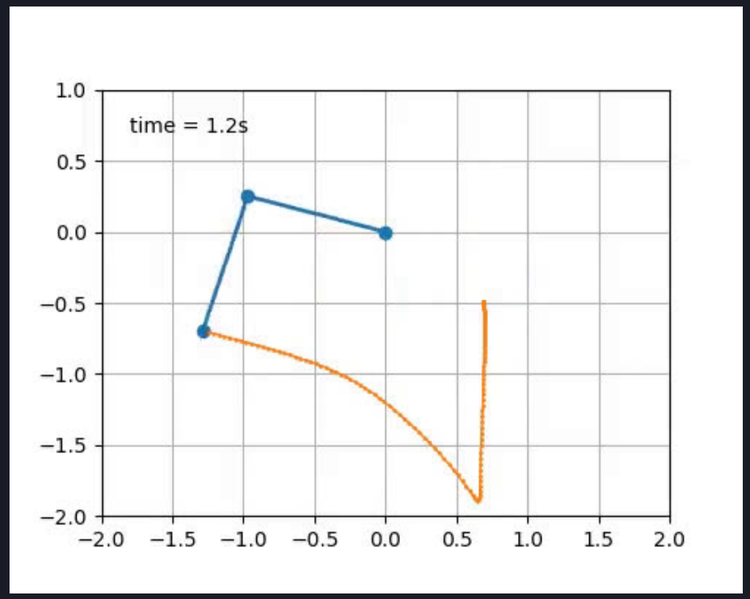


source: https://matplotlib.org/stable/gallery/index.html









source: https://matplotlib.org/stable/gallery/index.html

Matplotlib Animations

(behind the scenes)

MPL Artist Objects -> External Rendering Engines

1. **MPL Core**: Handles data visualization logic, and creates `Artist` objects (lines, markers, text) for each frame.

2. **Animation Module**: Manages frame generation (e.g., *`FuncAnimation`*).

3. **External Libraries** (FFmpeg, ImageMagick):
   - Leverage Matplotlib's Artist objects for frame rendering.
   - Encode individual frames into an animation movie (FFmpeg) or image sequence (ImageMagick).

Matplotlib focuses on declarative animation logic, while external libraries handle rendering and playback.

Matplotlib Animation

matplotlib.animation

FuncAnimation

- Dynamic Updates
- Function-Based: core concept of FuncAnimation is to repeatedly call a user-defined function to update the plot's elements.
- + Initialization (optional)

ArtistAnimation

- Fixed Artists (e.g. Line2D, Patch, etc.)
- Pre-defined Frames
- Simplicity: Easier to use when the visual elements are known and fixed in advance.

- **FuncAnimation**: Ideal for dynamic animations with continuously updating data.
- **ArtistAnimation**: Suitable for simpler animations with a fixed set of visual elements.

Basics of Matplotlib Animation

```python
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np

fig, ax = plt.subplots()
line, = ax.plot([])
```

Animation Module: Utilize `*matplotlib.animation*` *class* for creating animations.

Figure and Axes Setup: Initialize the figure and plot with `plt.subplots()` and define an initial empty plot.

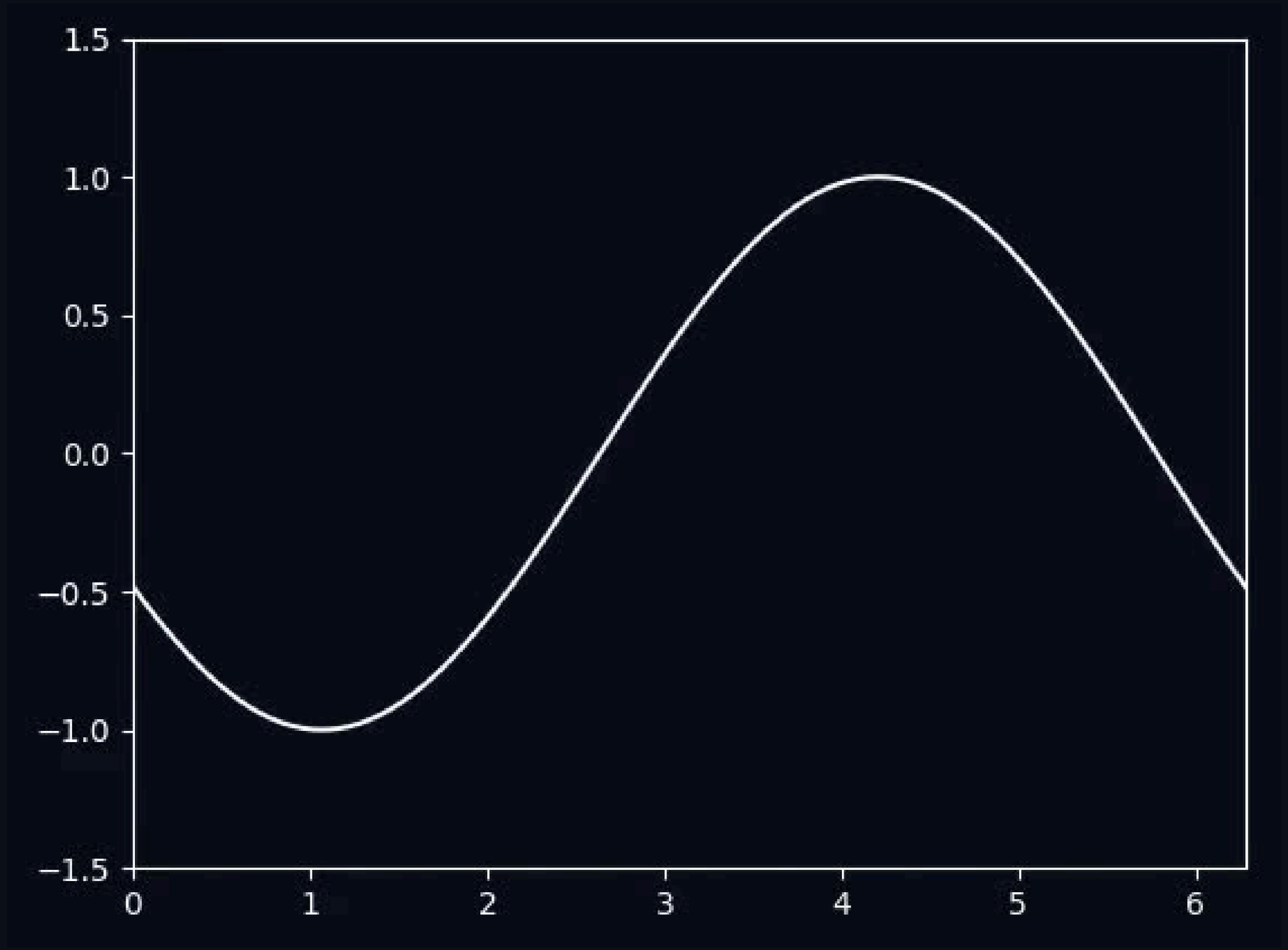
```python
def init():
    ax.set_xlim(0, 2*np.pi)
    ax.set_ylim(-1.5, 1.5)
    return line,
```

```python
def update(frame):
    x = np.linspace(0, 2*np.pi, 100)
    y = np.sin(x + 2 * np.pi * frame / 100)
    line.set_data(x, y)
    return line,
```

Initialization and Update Functions: Define `*init*` for setting up the plot limits & update for modifying plot data per frame

```python
ani = animation.FuncAnimation(
  fig, update, frames=100, init_func=init, blit=True
)
ani.save('sine_wave.mp4', writer='ffmpeg')
plt.show()
```

Creating and Displaying Animation: Use `*FuncAnimation*` to animate the plot, save it if needed, and display with `plt.show()`.



Example: animating a sine wave in matplotlib

Example: Animations using Line Collection

(matplotlib.collections)

Efficient way to handle large collections of lines in a plot.

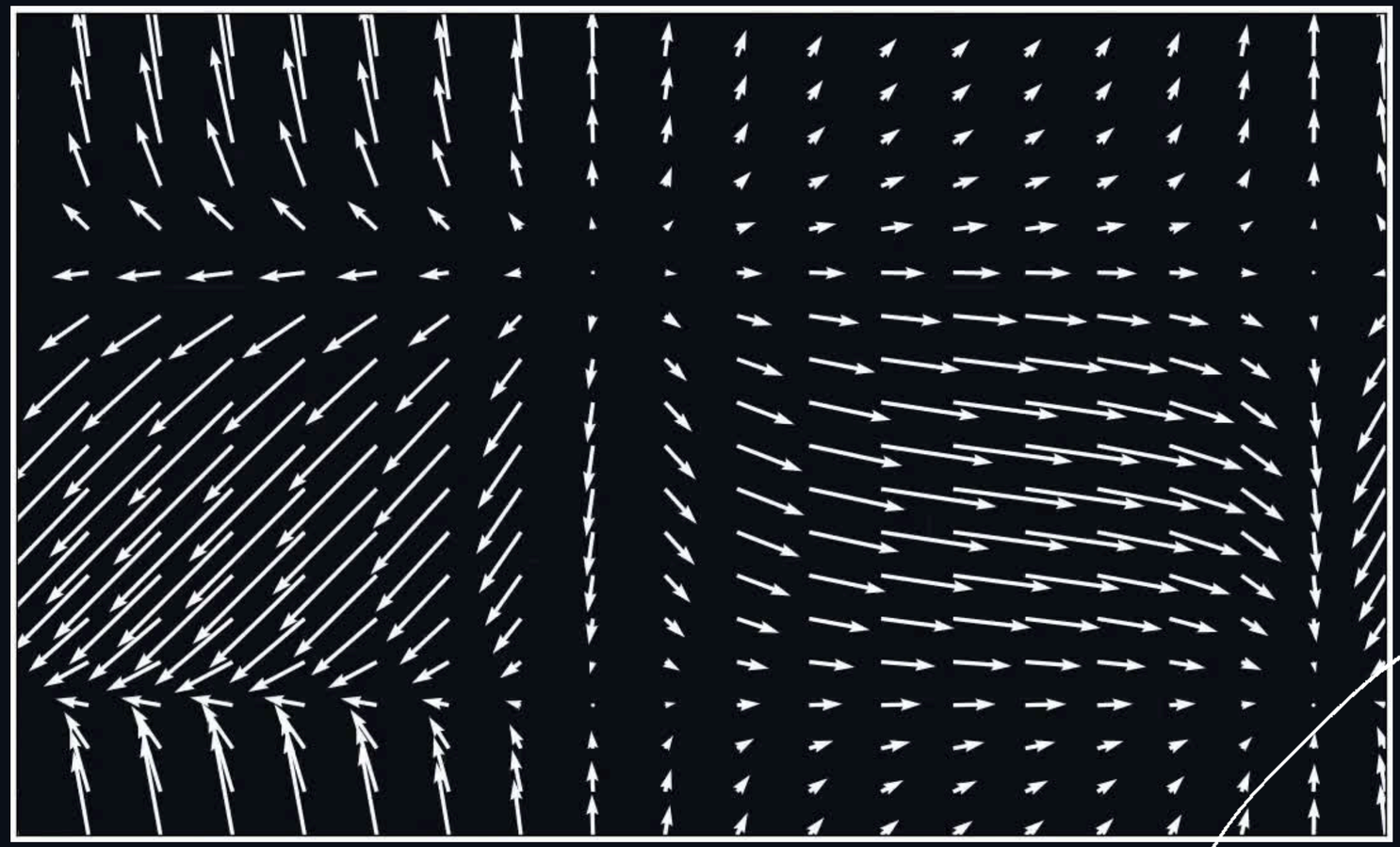
```python
def plot_linecollection(
  ax, x, y, colors="white", linewidths=4
):
    segments = make_segments(x, y)
    lc = LineCollection(
      segments, colors=colors, linewidths=linewidths
    )
    ax.add_collection(lc)
    return lc

# Complex wave generation: combination of sine and cosine
y = np.sin(time) + np.cos(time * 2)

lc = plot_linecollection(ax, time, y)

# Animation
animation = animate_line_collections(lc, fig, frames)
```

```python
def animate_line_collections(lc, fig, frames):
    def init():
        lc.set_segments([])
        return lc,

    def anim_func(frame):
        lc.set_segments(
          make_segments(time[:frame], y[:frame])
        )
        return lc,

    return animation.FuncAnimation(
      fig, anim_func, init_func=init,
      frames=frames, interval=10
    )
```

Example: Quiver Plot using a meshgrid

A type of plot that shows vector fields using arrows.

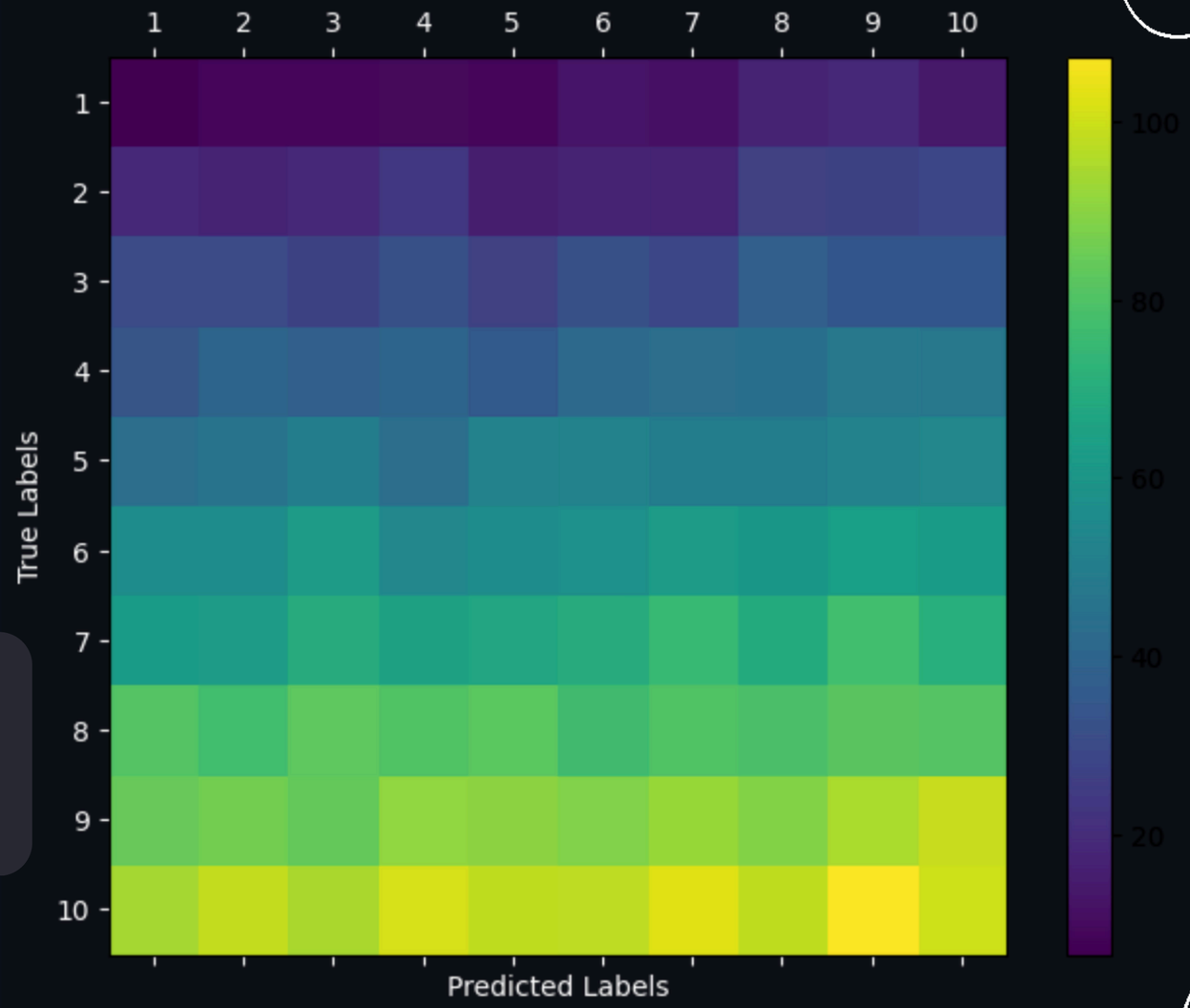
```
# Create quiver plot
Q = ax.quiver(
  X, Y, U, V, color='white',
  edgecolor='none', scale=30
)

def update_quiver(num, Q, X, Y):
    U = np.cos(X + num * np.pi / 30) * np.exp(
      -np.sin(Y + num * np.pi / 30)
    )
    V = np.sin(Y + num * np.pi / 30) * np.exp(
      -np.cos(X + num * np.pi / 30)
    )

    Q.set_UVC(U, V)
    return Q,

# Create animation
ani = animation.FuncAnimation(
  fig, update_quiver, fargs=(Q, X, Y), frames=180,
  interval=100, blit=True
)
```



Example: Confusion Matrix 2D

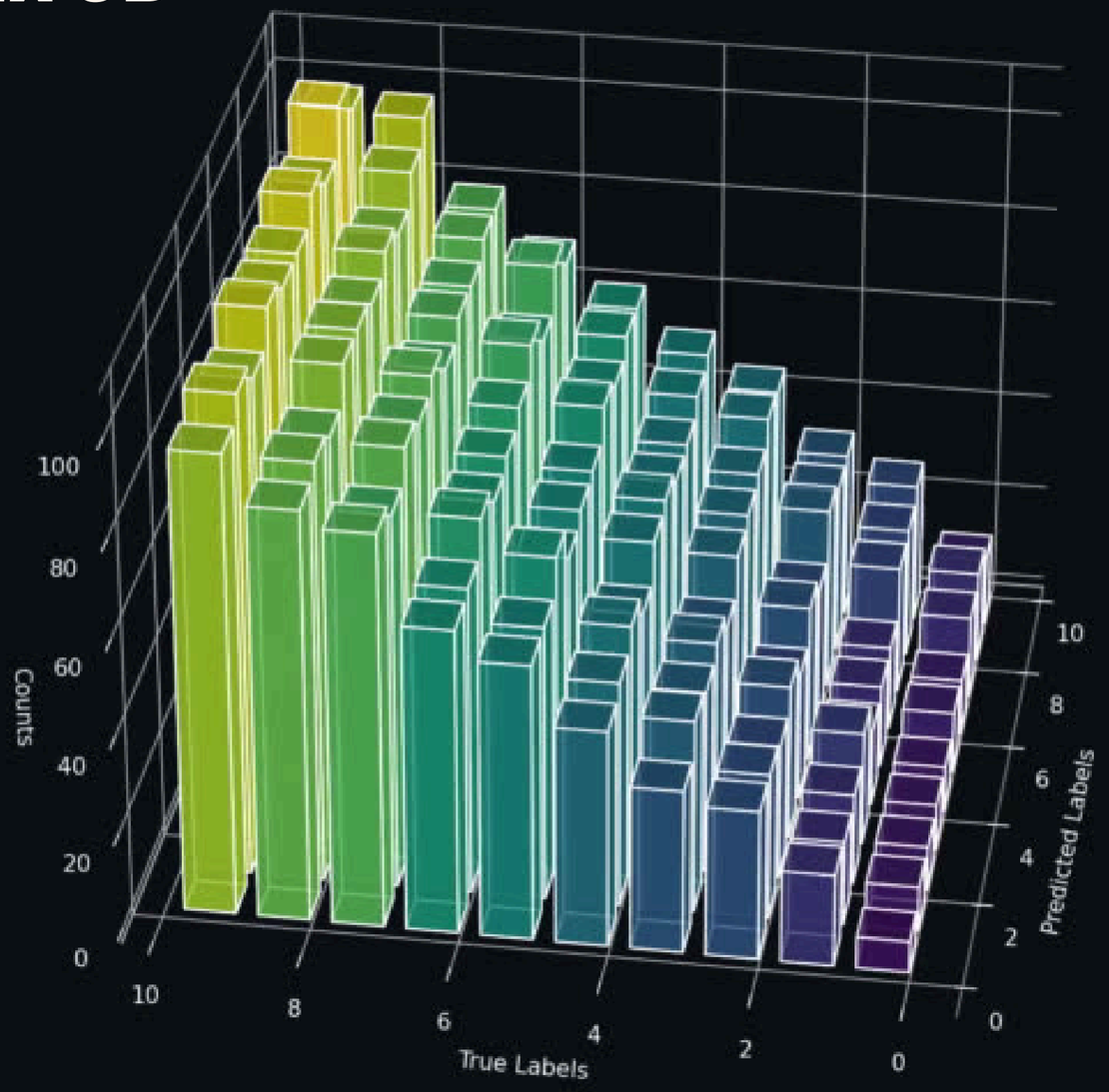
```
np.random.seed(0)
# 10x10 matrix with increasing values
data = np.linspace(1, 100, 100).reshape(10, 10)
# Add slight random noise
data += np.random.rand(10, 10) * 10
```

```
fig, ax = plt.subplots(figsize=(8, 6))

cax = ax.matshow(data, cmap='viridis')
fig.colorbar(cax)
```



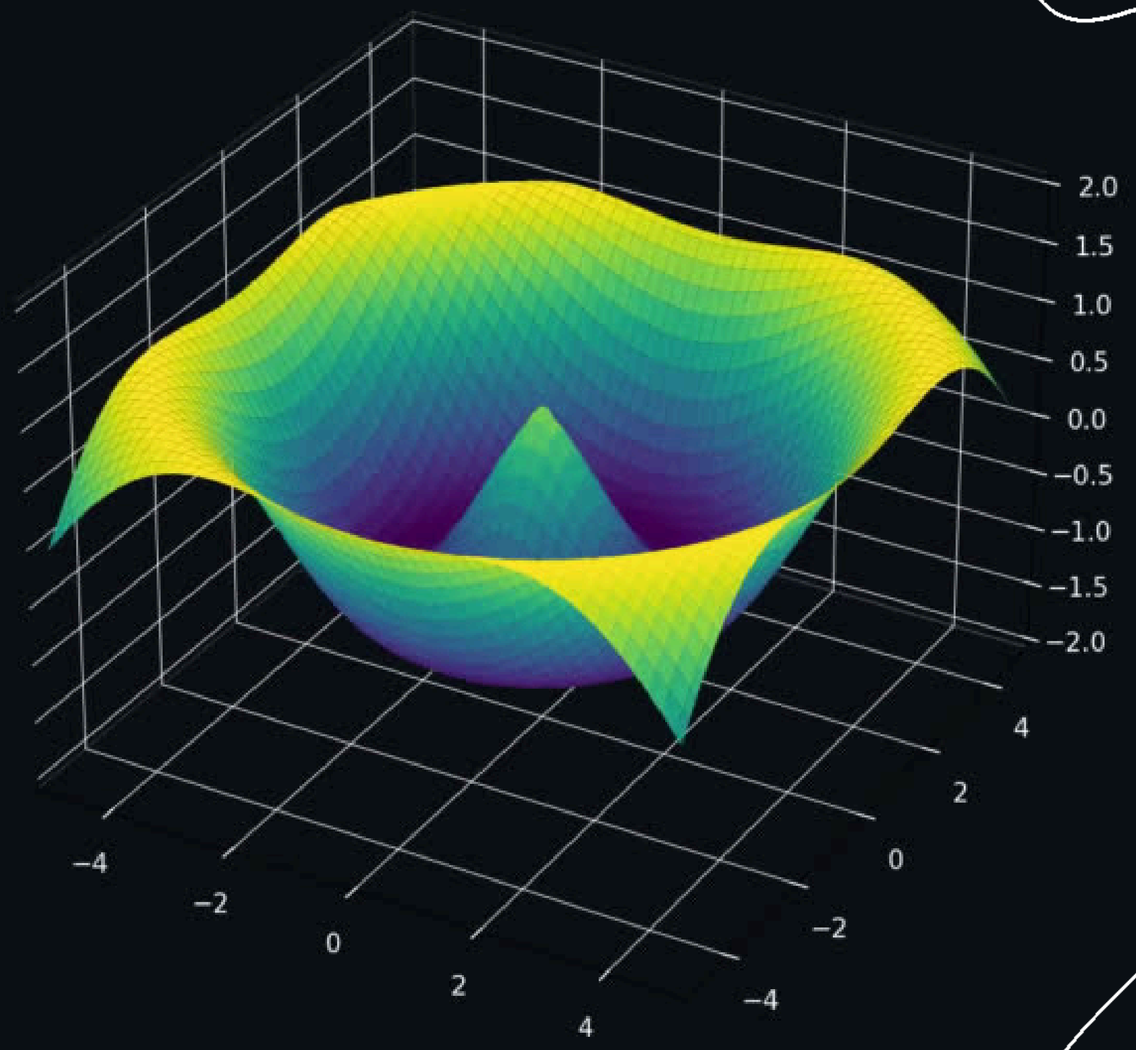
Example: Confusion Matrix 3D

```
def animate(i):
    ax.view_init(elev=30, azim=3 * i)  # Smooth rotation

ani = FuncAnimation(
  fig, animate, init_func=init, frames=360,
  interval=100, blit=False, repeat=True
)
```



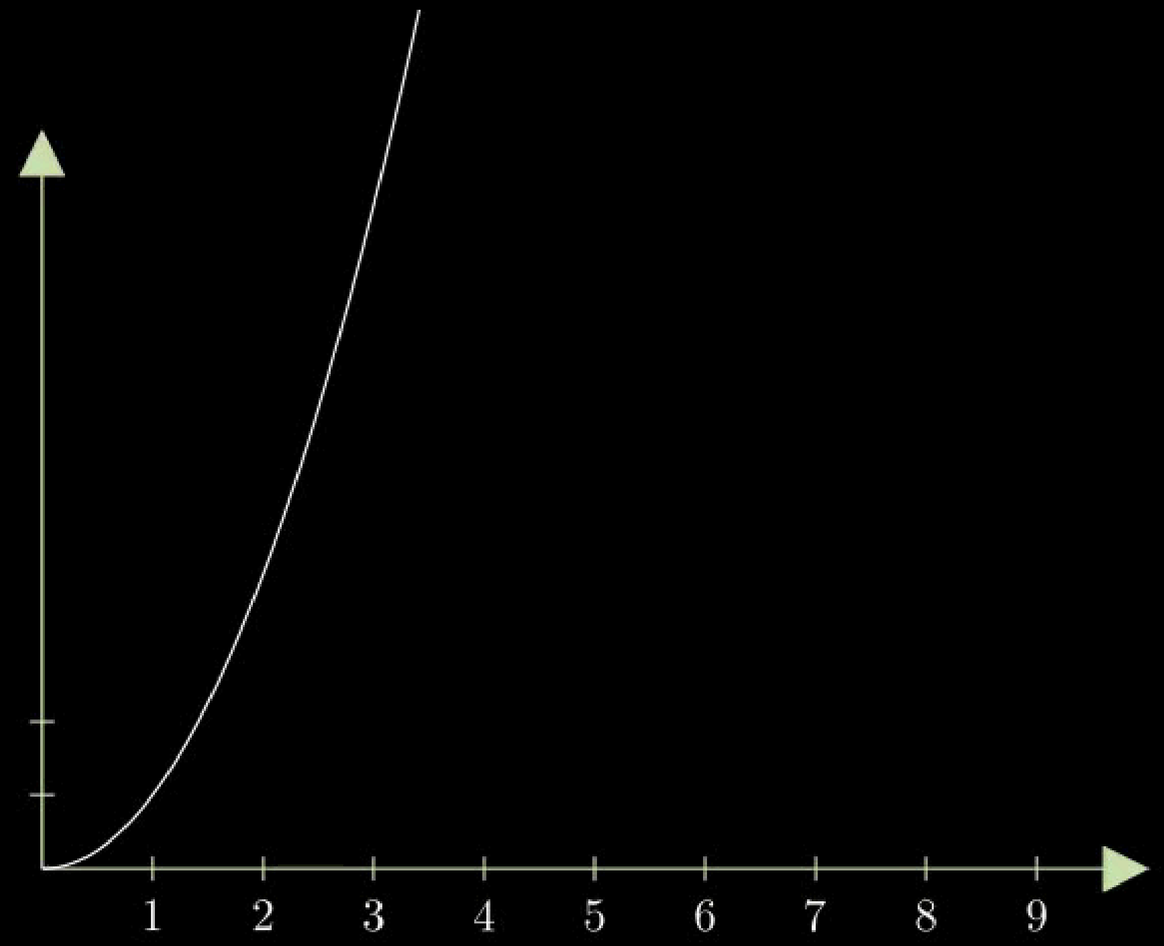
Example: Air Flow Simulation over surface

```
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

def animate(i):
    ax.clear()  # Clear the previous surface
    # Update the z-values to simulate flowing air
    z = np.sin(np.sqrt(x**2 + y**2) - 0.1*i)
    ax.plot_surface(x, y, z, cmap='viridis', edgecolor='none')
    ax.set_xlim([-5, 5])
    ax.set_ylim([-5, 5])
    ax.set_zlim([-2, 2])
    return fig,

# Create the animation
ani = FuncAnimation(
  fig, animate, frames=360, interval=50, blit=False
)
```



01

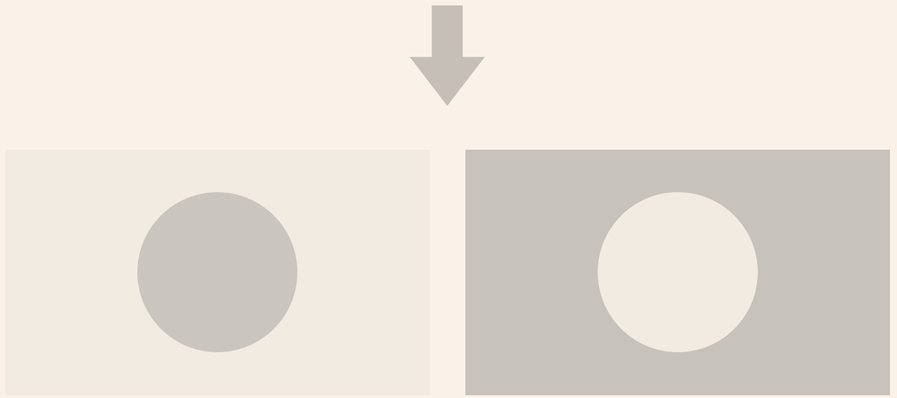
Manim



Manim “*Mathematical Animations Engine*” is an engine for precise programmatic animations, designed for creating explanatory math videos.

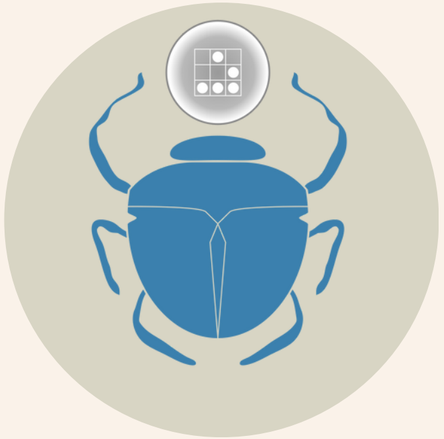
```python
class PyConText(Scene):
    def construct(self):
        text = Text("PyCon 2024").scale(2)
        self.play(Write(text))
        self.wait(1)
```

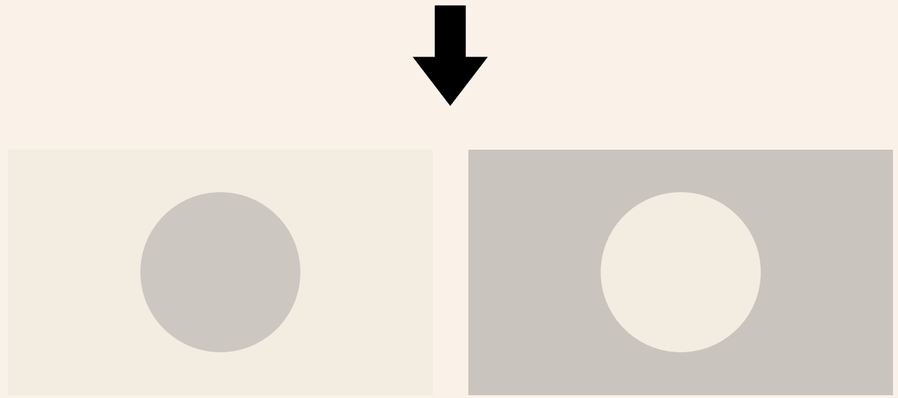
PyCon 20



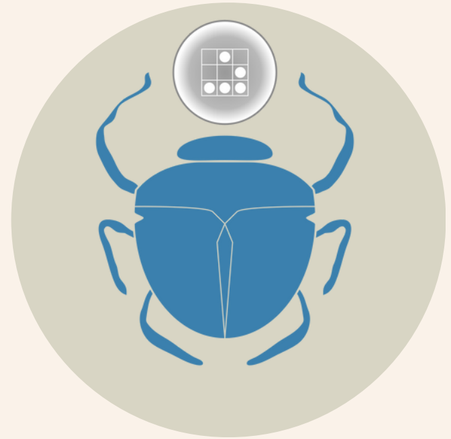
```python
class PyConText(Scene):
    def construct(self):
        text = Text("PyCon 2024").scale(2)
        self.play(Write(text))
        self.wait(1)
```

PyCon 20



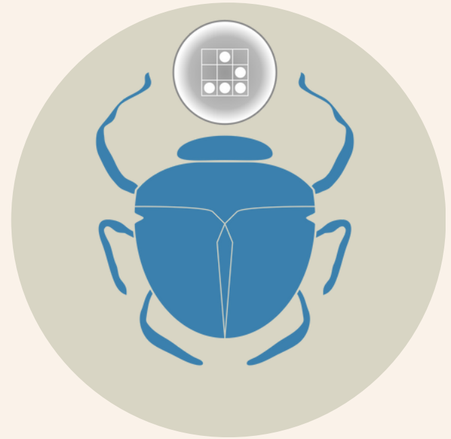


```python
class PyConText(Scene):
    def construct(self):
        text = Text("PyCon 2024").scale(2)
        self.play(Write(text))
        self.wait(1)
```

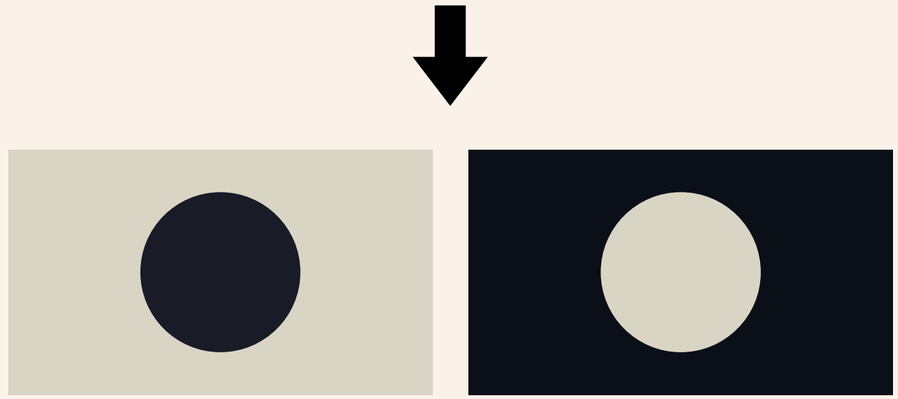


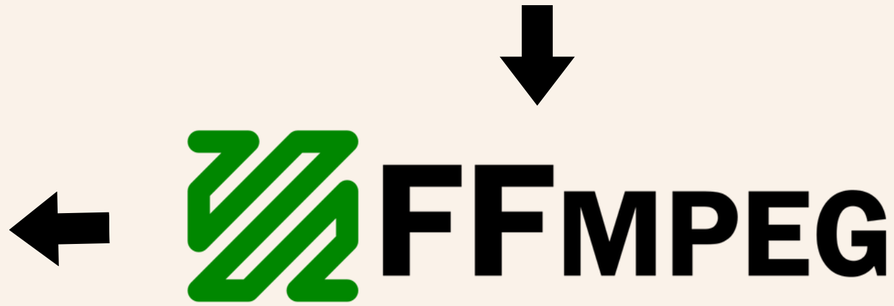
PyCon 20

```python
class PyConText(Scene):
    def construct(self):
        text = Text("PyCon 2024").scale(2)
        self.play(Write(text))
        self.wait(1)
```



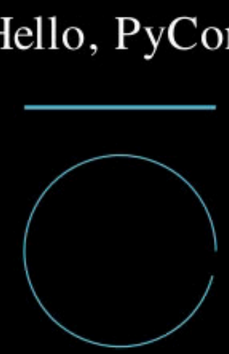
PyCon 20





Basic Objects in Manim: Text, Lines, and Shapes

- **Text**: TextMobject("Hello, World!") creates text.
- **Lines**: Line(LEFT, RIGHT) draws a simple line.
- **Shapes**: Circle(), Square(), and Polygon(*vertices) for geometric figures.
- MObjects, VMObject



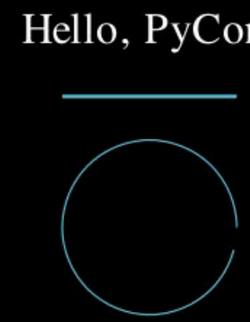
```python
from manim import *

class BasicObjectsScene(Scene):
    def construct(self):
        # Create a text object
        text = Text("Hello, PyCon!", font_size=36)
        text.to_edge(UP)

        # Create a line object
        line = Line(LEFT, RIGHT)
        line.set_color(BLUE)

        # Create a circle object
        circle = Circle()
        circle.set_fill(BLUE, opacity=0.5)
        circle.set_stroke(color=BLUE)

        # Arrange objects on the screen
        group = VGroup(text, line, circle)
        group.arrange(DOWN, buff=0.5)

        # Animate the objects
        self.play(Write(text))
        self.play(Create(line))
        self.play(DrawBorderThenFill(circle))

        # Hold the final scene
        self.wait(2)
```

Creates a **text object** that says "Hello, PyCon!" with a font size of 36.

Creates a **line** that stretches from the left to the right side of the scene

Creates a **circle** object.
And adds a **blue** fill.

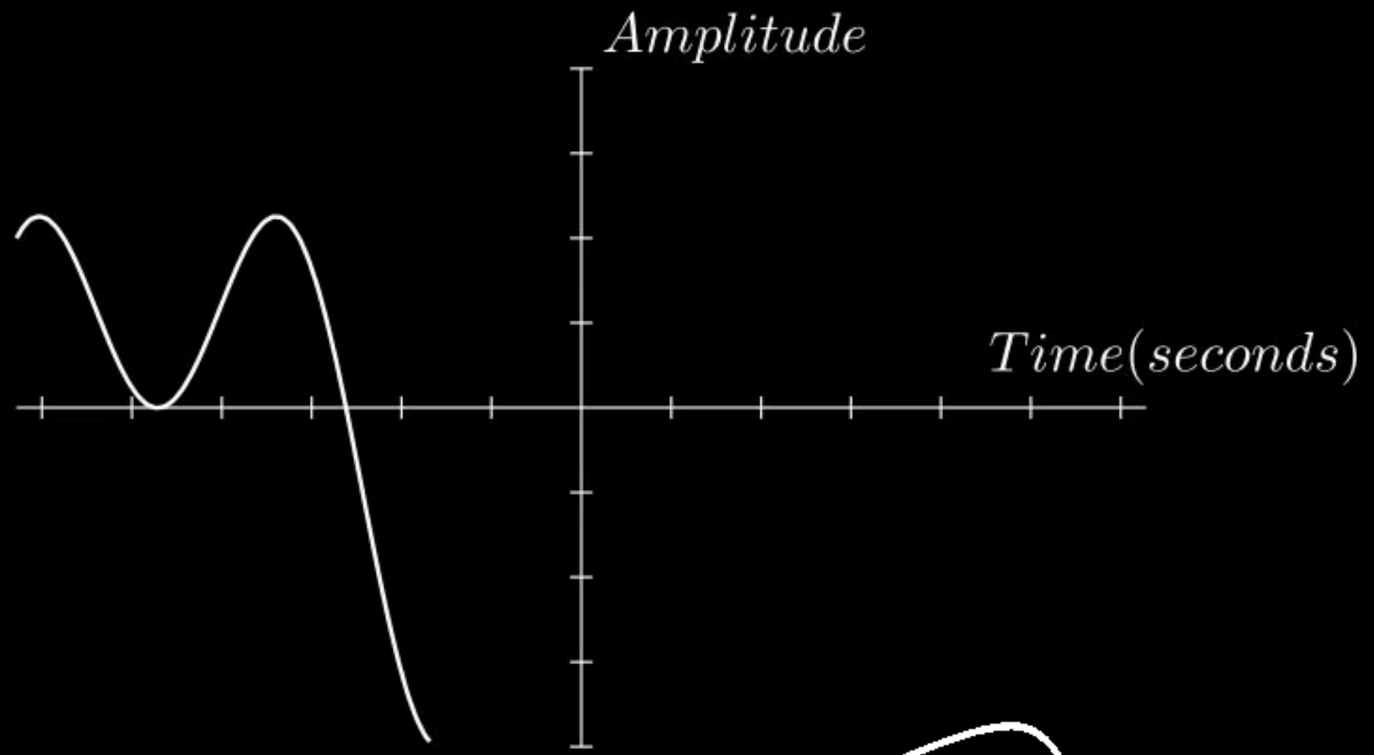
Groups all the created objects (text, line, circle) together. Imagine putting them in a container.



Wave functions

The wave is defined by the function `sin(x)+cos(2x)`*, which combines a sine wave and a cosine wave of double the frequency, resulting in a more complex pattern.*

This function is plotted over the specified range `[-2π,2π][-2π,2π]`*.*



Wave Functions

Importing Libraries and Setting Up the Scene

```
class WaveAnimation(Scene):
    def construct(self):
        # Set up the axes
        axes = Axes(
            x_range=[-2 * PI, 2 * PI, 1],
            y_range=[-2, 2, 0.5],
            x_length=10,
            y_length=6,
            axis_config={"color": WHITE},
            tips=False,
        )

        # Label axes
        axes_labels = axes.get_axis_labels(
            x_label="Time (seconds)", y_label="Amplitude"
        )
```

Defines the **WaveAnimation** class, inheriting from Scene.

Sets up the axes for the graph, specifying the range and length for both x and y-axes.

Wave Functions

Creating the Wave Function

```
# Create the wave function
wave = axes.plot(
    lambda x: np.sin(x) + np.cos(2 * x),
    color=WHITE,
    x_range=[-2 * PI, 2 * PI],
)
```

Defines a **wave function using a lambda function** that combines sine and cosine waves.

```
# Initial setup of the wave line (starting at the first point of the wave)
first_point = wave.get_points()[0]
wave_line = VMobject()
wave_line.set_points_as_corners([first_point, first_point])
wave_line.set_color(WHITE)
```

*Initializes the **wave line**, starting at the first point of the wave.*

*It creates a **VMobject** for the wave line and sets its initial points and color.*

Wave Functions

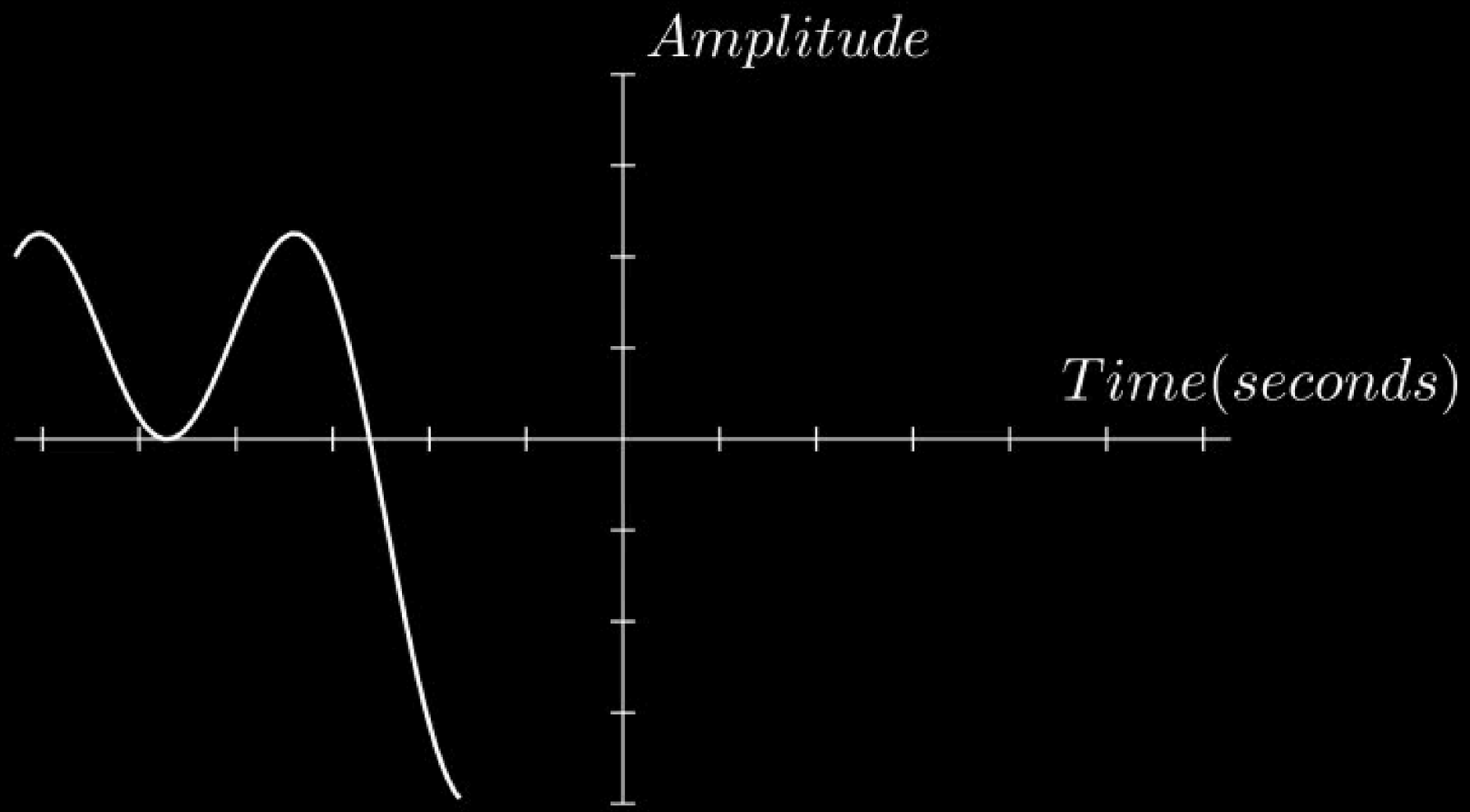
Animating the Wave Line

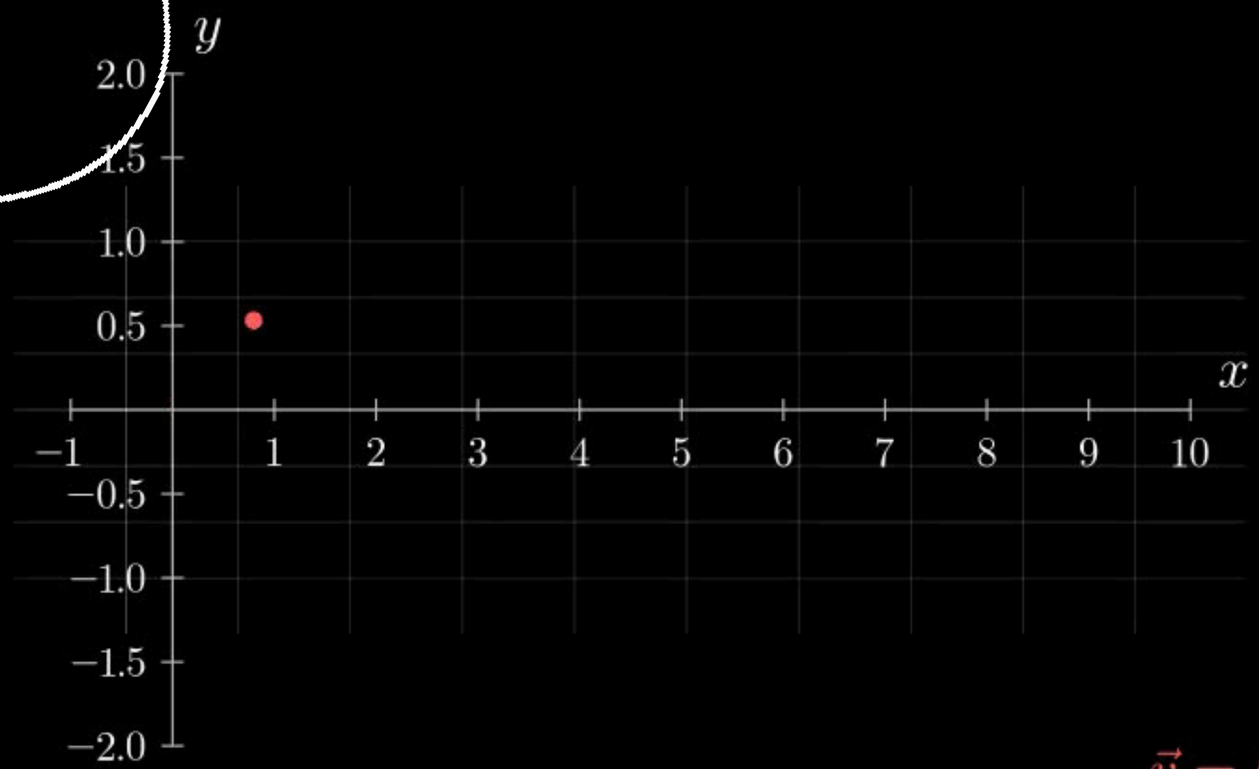
Adds the **axes, labels, and initial wave line** to the scene.

```
# Animate drawing the wave
self.add(axes, axes_labels, wave_line)
self.play(Create(axes), Write(axes_labels))
self.play(
    UpdateFromAlphaFunc(
        wave_line,
        lambda mob, alpha: mob.set_points_as_corners(
            [first_point, *wave.get_points()[:int(alpha * len(wave.get_points()))]]
         )
     ),
        run_time=4,  # Animation run time in seconds
        rate_func=linear
)

self.wait(1)  # Hold the final frame
```

*Uses the **UpdateFromAlphaFunc** to animate the drawing of the wave line from the first point to the last point over a specified run time.*

*Holds the **final frame** for a moment to allow the audience to observe the completed animation.*


Amplitude
Time(seconds)



$\vec{v} = (0, 0)$

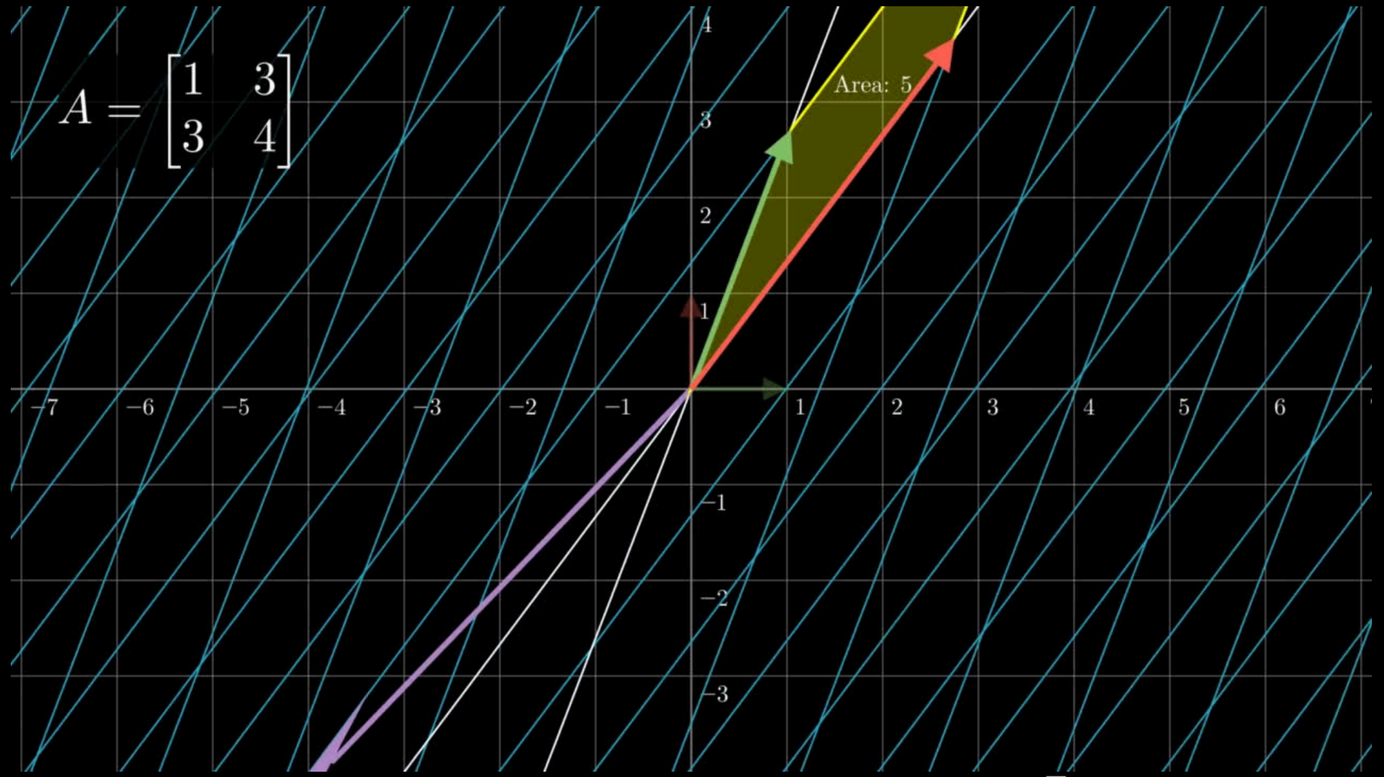
Coordinate Systems

This scene showcases an enhanced coordinate system with dynamic elements to explore vector movements and transformations in a 2D space.

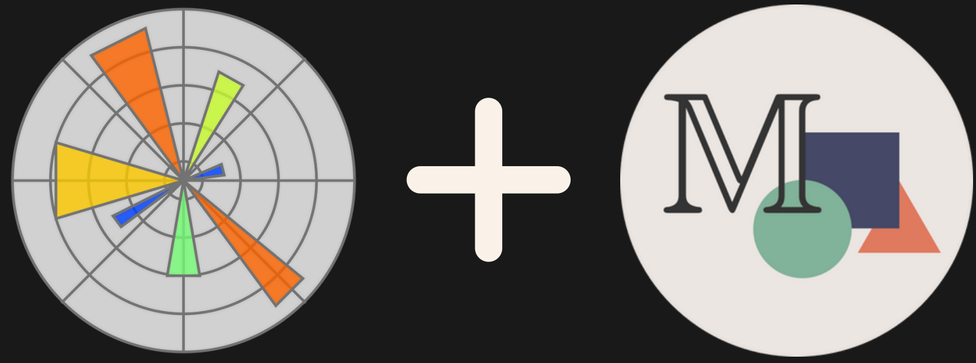
Linear Transformations

This demonstrates the effect of a linear transformation, specified by a matrix, on various geometric objects in a 2D space.

By applying the matrix [[1, 3], [3, 4]], the script visually transforms a unit square, a vector, an ellipse, and a triangle, illustrating how these objects rotate, scale, and skew according to the rules defined by the matrix.



Combining Matplotlib with Manim

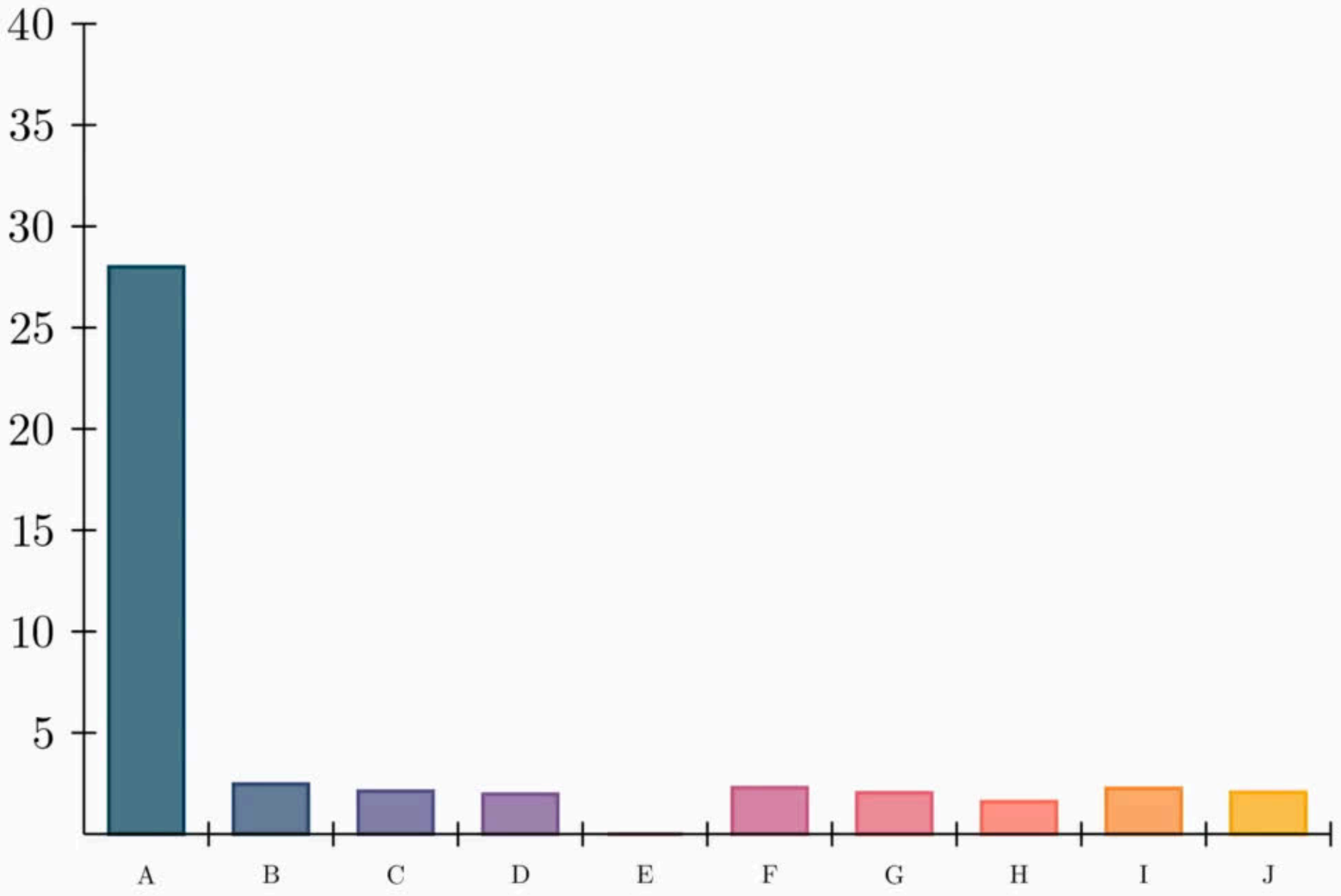


```python
class BarChart(Scene):
    def construct(self):
        # Increase the figure size
        fig, ax = plt.subplots(figsize=(10, 6))

        # Create a bar chart with a darker theme
        ax.bar([1, 2, 3], [3, 1, 2], color='white')

        # Set a dark background and light gridlines
        fig.patch.set_facecolor('#0e1117')
        ax.set_facecolor('#0e1117')
        ax.grid(color='white', linestyle='-', linewidth=0.5)

        # Set the color of the spines, ticks, and labels to white
        ax.spines['bottom'].set_color('white')
        ax.spines['top'].set_color('white')
        ax.spines['right'].set_color('white')
        ax.spines['left'].set_color('white')
        ax.xaxis.label.set_color('white')
        ax.yaxis.label.set_color('white')
        ax.tick_params(axis='x', colors='white')
        ax.tick_params(axis='y', colors='white')

        # Save the plot
        plt.savefig("bar_chart.png", facecolor=fig.get_facecolor(), edgecolor='none')

        # Load and display the image in Manim
        image = ImageMobject("bar_chart.png")
        self.play(FadeIn(image, run_time=3))  # Increase run_time to slow down the animation
```

Matplotlib figure and axis are created with a specified size to increase the figure's size. A bar chart is plotted with **three bars**.

Styling the Chart, setting the backgrounds, customising the grid linescontrast, and updating the colors of labels, ticks etc.

- The saved image is loaded into Manim as an ImageMobject which is then animated to appear on the screen using a fade-in effect that lasts for 3 seconds.


40
35
30
25
20
15
10
5
A
B
C
D
E
F
G
H
I
J

Blender

bpy - Blender's Python API

`bpy` extends its 3D modeling capabilities to automate and script model creation and animation.

This tool transforms data into dynamic 3D visual narratives, enhancing data visualization with programmable elements from charts to complex simulations.

`bpy` melds Blender's graphical power with Python's flexibility, making it an essential tool for data analysts and visual artists, redefining traditional data visualization methods.

Scene Management

- *Encompasses all elements in a 3D world: cameras, lights, objects, and world settings.*

```python
import bpy
# Create a new scene
new_scene = bpy.data.scenes.new("MyScene")
bpy.context.window.scene = new_scene
```

- **Create:** Instantiate new scenes directly via script.

Scene Management

- *Encompasses all elements in a 3D world: cameras, lights, objects, and world settings.*

```python
import bpy

# Set the camera position and orientation
camera = bpy.context.scene.camera
camera.location = (3, 3, 3)
camera.rotation_euler = (0.785, 0, 1.047)  # Approx. 45 and 60 degrees in radians

# Add and configure a point light
bpy.ops.object.light_add(type='POINT', location=(2, 2, 5))
light = bpy.context.object
light.data.energy = 500  # Set light intensity
```

- **Configure:** Modify properties of scenes to adjust settings such as lighting or camera angles.

Scene Management

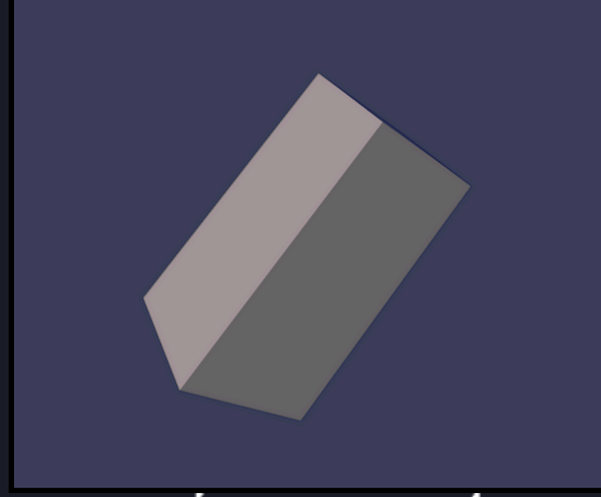
```python
import bpy

# Create two new scenes
scene1 = bpy.data.scenes.new("Scene1")
scene2 = bpy.data.scenes.new("Scene2")

# Set scene1 as the active scene
bpy.context.window.scene = scene1

# Perform operations in scene1
# Example: Adding a cube in scene1
bpy.ops.mesh.primitive_cube_add(size=1, location=(0, 0, 0))

# Switch to scene2
bpy.context.window.scene = scene2

# Perform different operations in scene2
# Example: Adding a sphere in scene2
bpy.ops.mesh.primitive_uv_sphere_add(radius=1, location=(2, 2, 2))
```

- **Switch:** Programmatically change between multiple scenes within a project for efficient workflow management.

Object Manipulation

- **Add Objects:** Use `bpy.ops.mesh.primitive_*_add` functions to create various types of geometric primitives (e.g., cubes, spheres, circles).
- **Delete Objects:** Remove objects from scenes using `bpy.ops.object.delete()` or manage data directly via `bpy.data.objects.remove()`.
- **Transform and Modify Properties:** Programmatically adjust object locations, rotations, scales, and other properties.

Object Manipulation



```python
import bpy

# Add a cube to the scene
bpy.ops.mesh.primitive_cube_add(size=2, location=(0, 0, 0))
cube = bpy.context.object  # Get the newly created object

# Move the cube
cube.location = (2, 3, 5)

# Rotate the cube
cube.rotation_euler = (0.785, 0, 0.785)  # Approx. 45 degrees in radians on the X and Z axes

# Scale the cube
cube.scale = (1, 2, 1)  # Scale to double the height
```

- **Translation:** Move objects in 3D space.
- **Rotation:** Rotate objects around axes.
- **Scaling:** Resize objects proportionally or non-uniformly.

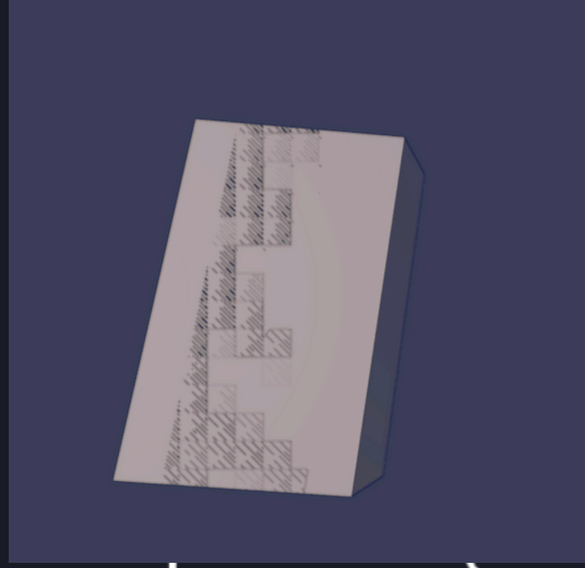
Data Handling

- ***Data Types:*** *Blender allows manipulation of various data types including meshes, curves, armatures, lamps, cameras, and materials. Each type represents different elements within a 3D scene.*

- ***Reading and Writing Data:*** *Access and modify data blocks to change properties like object geometries, apply materials, or adjust visual effects. This includes editing vertices in a mesh, changing material settings, or adjusting pose positions in armatures.*

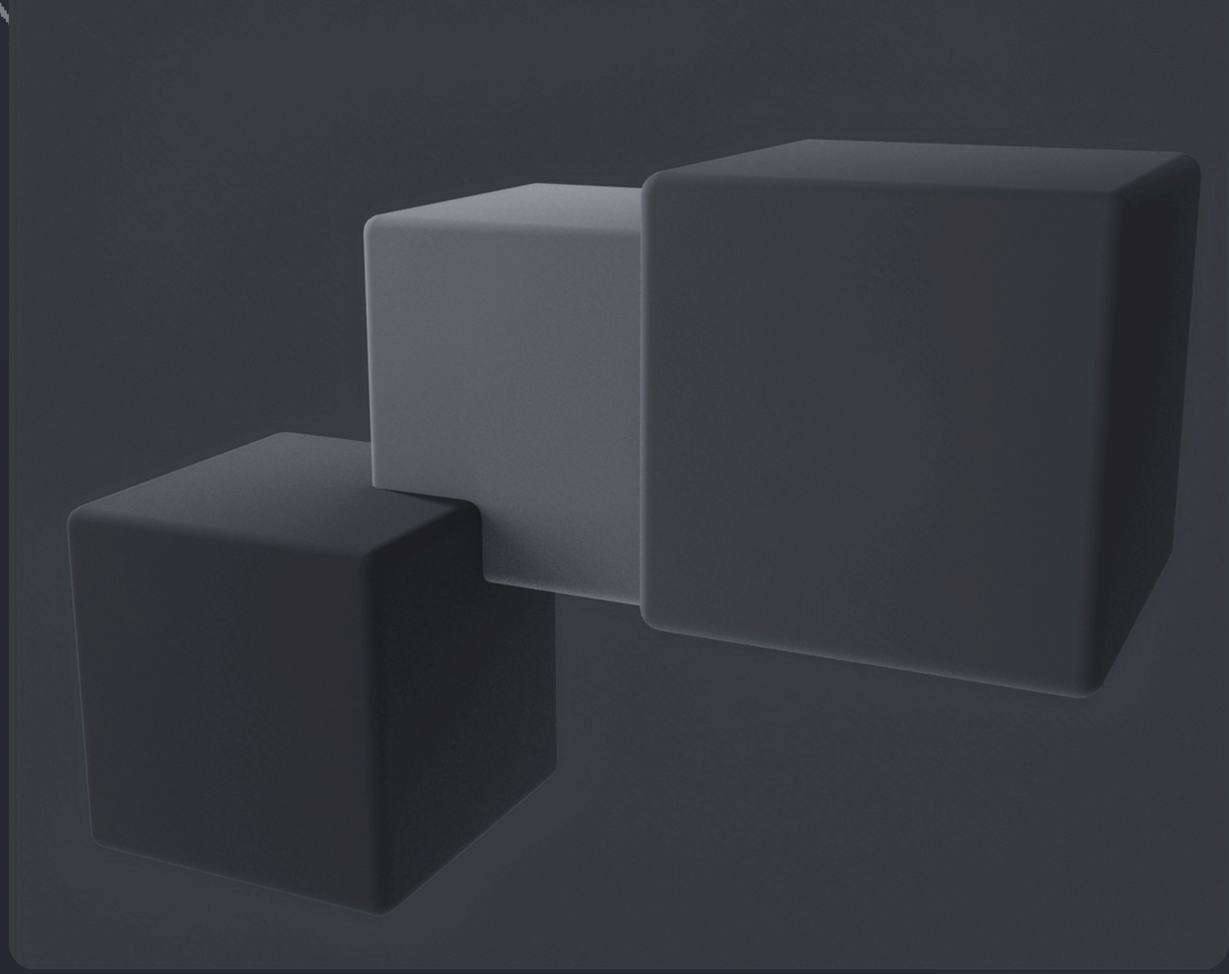
Data Handling

```python
import bpy

# Get the active object
obj = bpy.context.object

# Check if the active object is a mesh
if obj.type == 'MESH':
    mesh = obj.data  # Access the mesh data linked to the object

    # Modify vertices
    for vertex in mesh.vertices:
        vertex.co.x += 0.5  # Move each vertex along the X axis by 0.5
```



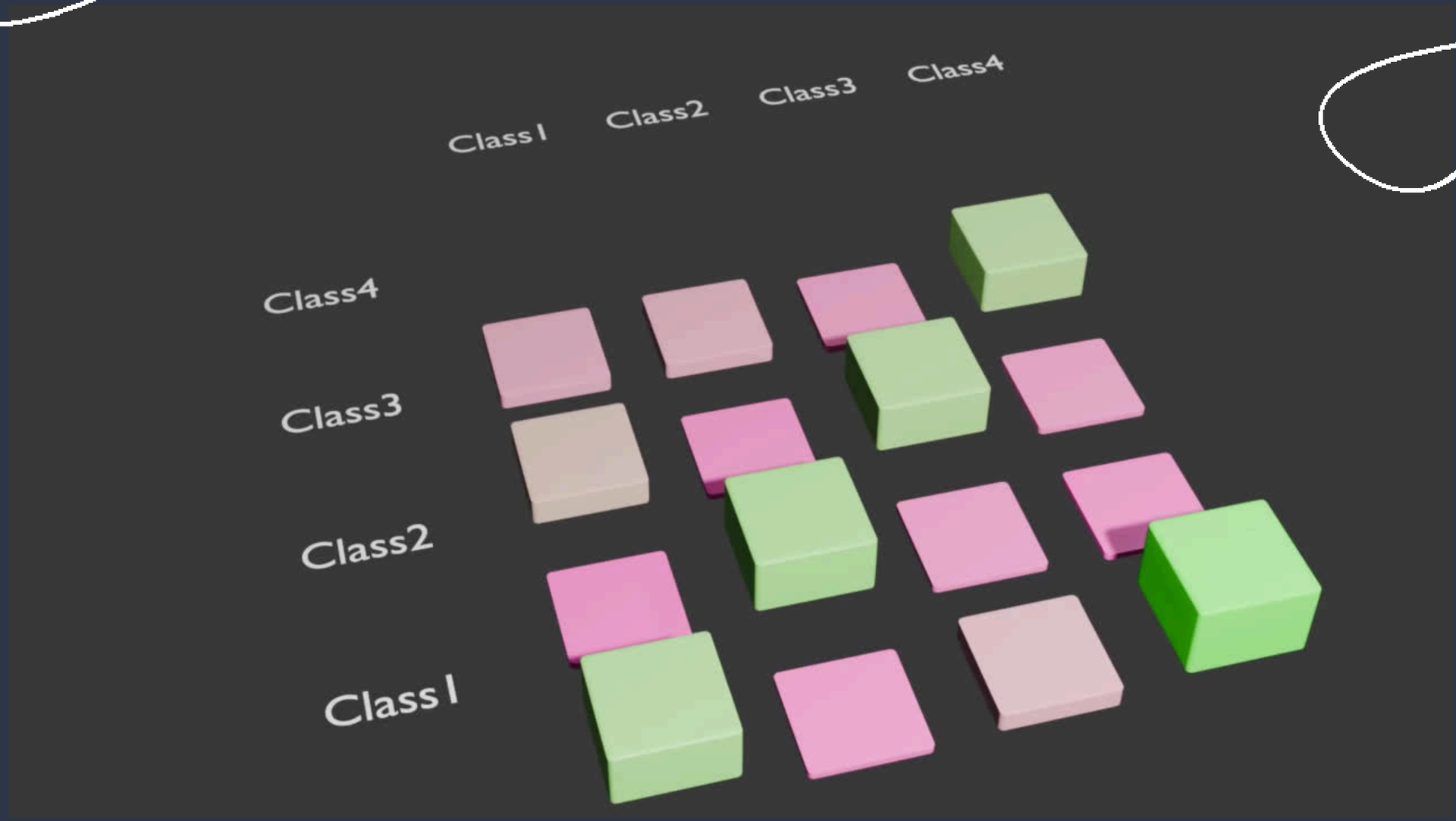
- **Active Object Access:** The script retrieves the currently selected object in Blender's context to ensure it's working with the right element.
- **Type Checking:** Confirms that the object is of type 'MESH', which is necessary to access and modify its vertex data.
- **Vertex Manipulation:** Adjusts the x-coordinate of each vertex in the mesh, shifting the entire mesh along the X-axis.

Basic cubes



```python
# Clear existing mesh objects
bpy.ops.object.select_all(action='DESELECT')
bpy.ops.object.select_by_type(type='MESH')
bpy.ops.object.delete()

# Function to create a cube with bevel and smooth shading
def create_cube(location):
    bpy.ops.mesh.primitive_cube_add(size=1, location=location)
    cube = bpy.context.active_object
    bpy.ops.object.modifier_add(type='BEVEL')
    cube.modifiers['Bevel'].segments = 4
    cube.modifiers['Bevel'].width = 0.05
    bpy.ops.object.shade_smooth()

    # Assign a random color material
    mat = bpy.data.materials.new(name="CubeMaterial")
    mat.diffuse_color = (random.random(), random.random(), random.random(), 1)
    cube.data.materials.append(mat)
    return cube

# Create three cubes with random positions
for _ in range(3):
    create_cube((random.uniform(-1, 1), random.uniform(-1, 1), random.uniform(-1, 1)))
```

Confusion Matrix in Blender



Confusion Matrix

Data Normalization and Scene Preparation

```python
import bpy
import numpy as np

# Confusion Matrix Data (Example)
confusion_matrix = np.array([
    [30, 2, 11, 39],
    [1, 31, 3, 1],
    [15, 2, 28, 4],
    [9, 10, 5, 27]
])

# Normalize the matrix
max_val = np.max(confusion_matrix)
normalized_matrix = confusion_matrix / max_val

# Clear existing objects
bpy.ops.object.select_all(action='DESELECT')
bpy.ops.object.select_by_type(type='MESH')
bpy.ops.object.delete()
bpy.ops.object.select_by_type(type='FONT')
bpy.ops.object.delete()
```

Create a matrix and **normalize** it to scale the values between 0 and 1, ensuring that the cube heights are proportional.

Clear the scene of any previous objects to ensure a clean start for the visualization

Confusion Matrix

Creating Cubes and Text Labels

```python
def create_cube(location, scale, value):
    bpy.ops.mesh.primitive_cube_add(size=1, location=location)
    cube = bpy.context.active_object
    cube.scale = (scale, scale, 0)  # Start with zero height

    # Add bevel and smooth shading
    bpy.ops.object.modifier_add(type='BEVEL')
    cube.modifiers['Bevel'].width = 0.05
    cube.modifiers['Bevel'].segments = 8  # Increased segments
    bpy.ops.object.modifier_add(type='SUBSURF')
    cube.modifiers['Subdivision'].levels = 2
    bpy.ops.object.shade_smooth()

    # Color gradient from red to green
    mat = bpy.data.materials.new(name=f"CubeMaterial_{value}")
    mat.use_nodes = True
    bsdf = mat.node_tree.nodes.get('Principled BSDF')
    color = (1 - value, value, 0.5 * (1 - value))
    bsdf.inputs['Base Color'].default_value = (*color, 1)
    cube.data.materials.append(mat)

    # Animation for cube growth
    animate_cube_growth(cube, scale, value)

    return cube
```

A cube is added to the scene at the specified location with a size of 1 unit, and it's **intial scale is set**.

A **bevel modifier** is applied to soften the edges of the cube, and a **subdivision surface modifier** is added to increase the mesh density.

The material's color is **dynamically** set to represent the data value through a color gradient, shifting from red (low value) to green (high value) which visually encodes the value dimension of the data

Confusion Matrix

Animation & Text Labels

```python
def animate_cube_growth(cube, scale, value):
    frame_start = 1
    frame_end = 60
    cube.scale[2] = 0
    cube.keyframe_insert(data_path="scale", index=2, frame=frame_start)
    cube.scale[2] = scale * max(0.1, value)
    cube.keyframe_insert(data_path="scale", index=2, frame=frame_end)
```

Animates the **vertical growth** of a cube from zero to its designated scale based on the value.

```python
def create_text(label, location, size=1):
    bpy.ops.object.text_add(location=location)
    txt = bpy.context.object
    txt.data.body = label
    txt.data.align_x = 'CENTER'
    txt.data.align_y = 'CENTER'
    txt.scale = (size, size, size)
    return txt
```

Creates a text object at a **specified** location with a given label and scale.

Confusion Matrix

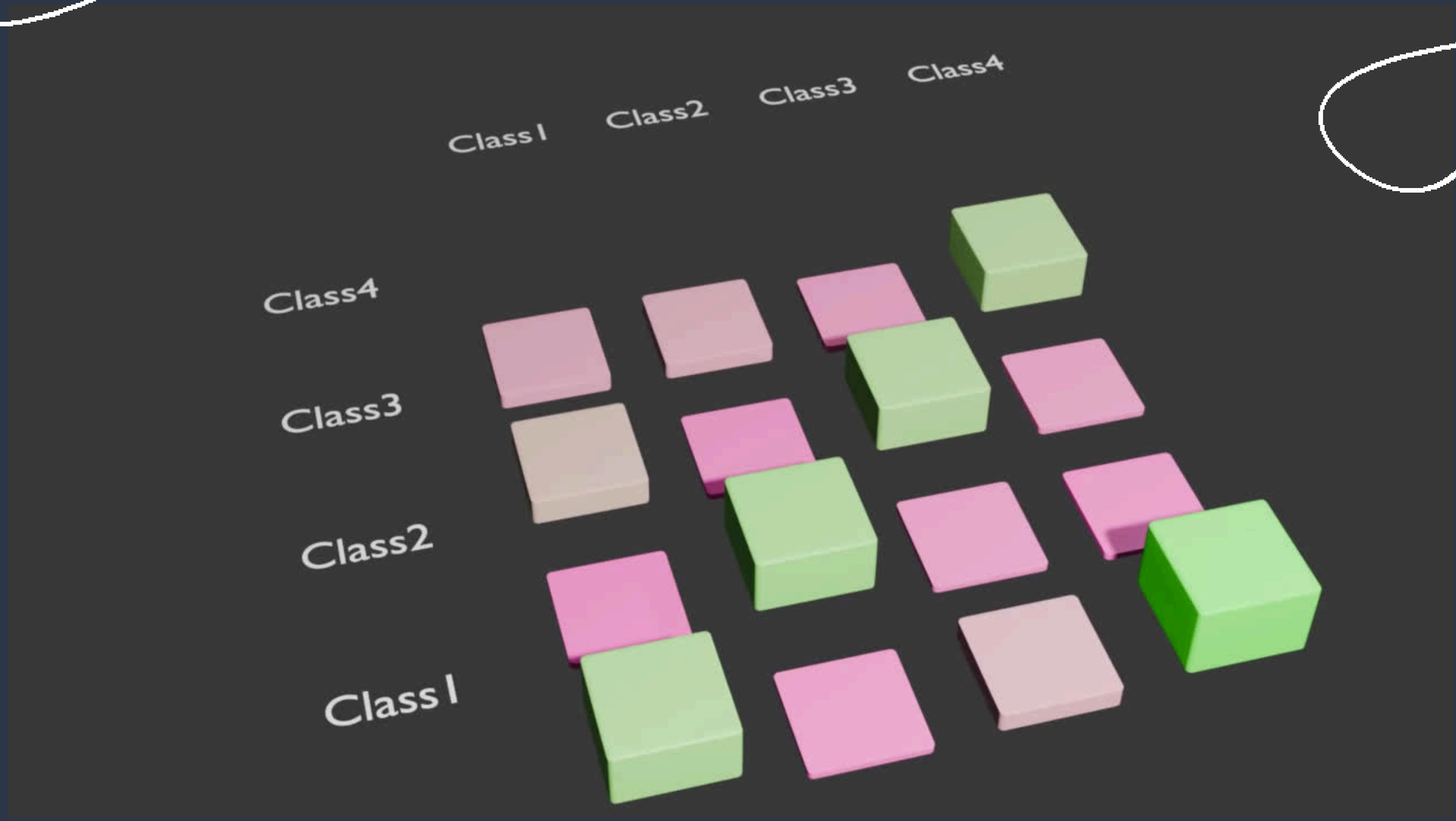
Scene Finalisation

```python
# Create visualization
num_rows, num_cols = confusion_matrix.shape
grid_size = 1.2
labels = ['Class1', 'Class2', 'Class3', 'Class4']
highest_value = np.max(normalized_matrix)

for i in range(num_rows):
    for j in range(num_cols):
        value = normalized_matrix[i, j]
        location = ((j - num_cols / 2) * grid_size, (i - num_rows / 2) * grid_size, value / 2)
        scale = 0.8
        create_cube(location, scale, value)
        if j == 0:
            create_text(labels[i], (location[0] - 1.5, location[1], highest_value), size=0.3)
        if i == num_rows - 1:
            create_text(labels[j], (location[0], location[1] + 1.5, highest_value), size=0.3)

# Setup camera and lighting
bpy.ops.object.camera_add(location=(0, -10, 5))
camera = bpy.context.active_object
camera.rotation_euler = (np.radians(60), 0, 0)

bpy.ops.object.light_add(type='POINT', location=(0, -10, 10))
light = bpy.context.active_object
light.data.energy = 1000
```

The **grid layout** is set up based on the confusion matrix dimensions, cube's position and scale are determined, and text labels are **dynamically** positioned based on their corresponding rows and columns.

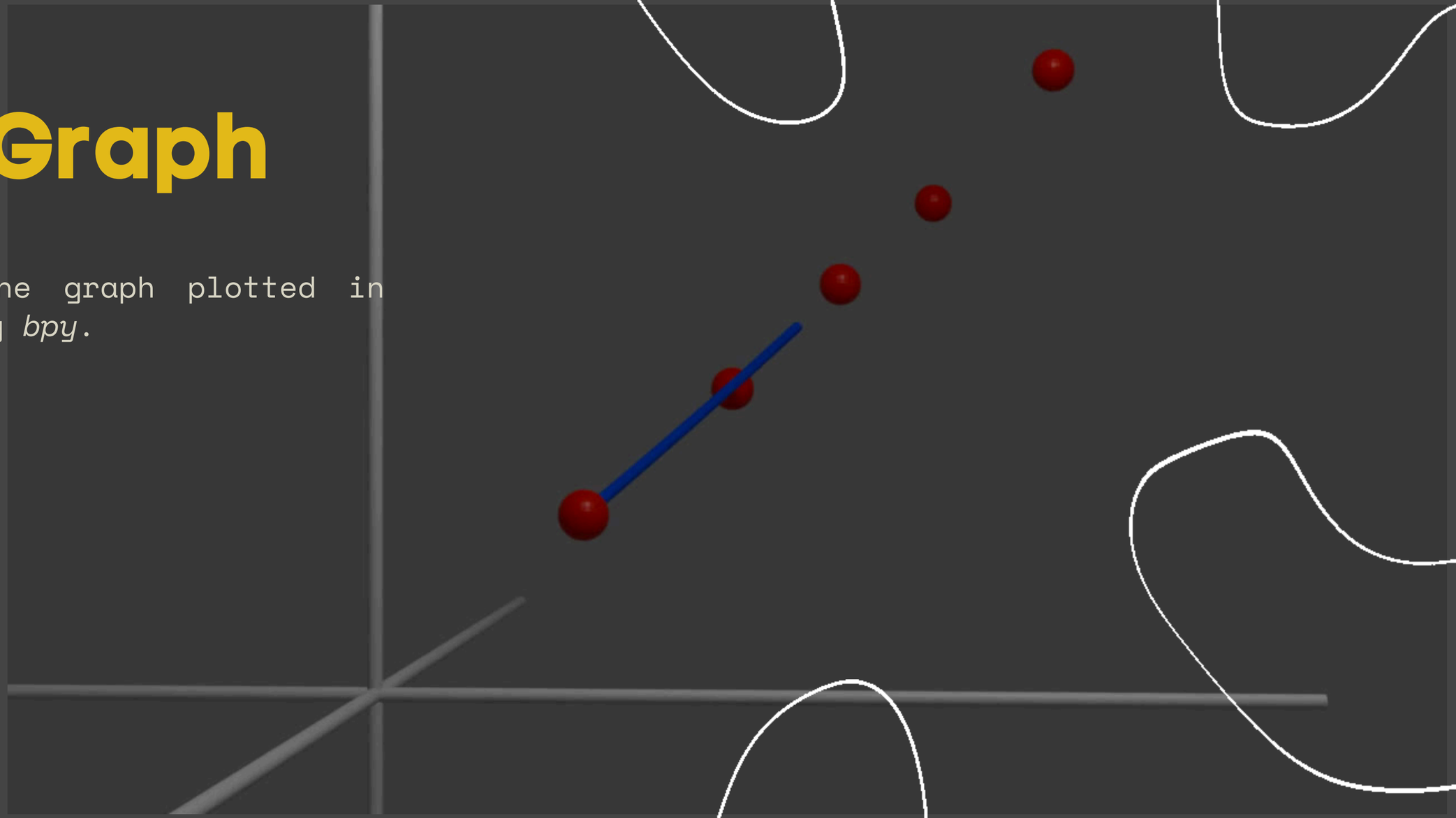
Configure the **camera and lighting** to enhance the visualization's appearance.

Confusion Matrix in Blender



Line Graph

A simple line graph plotted in Blender using *bpy*.



Resources

1. The Python visualization landscape (PyViz)
2. Animations in Matplotlib, docs
3. Made with Manim (community examples)
4. Quickstart with Blender

Notebook, Slides, Contact

Thank You

Neeraj Pandey——————————@n4jp4y

#PyConPL2024