Matplotlib Notes for ProgSD Exam

Contents

Chapter 1 Matplotlib

1.1 Imports

- To use Matplotlib effectively, it is essential to import the library with standard conventions.
- The most common imports for Matplotlib are:
  - `import matplotlib.pyplot as plt`: This is the standard convention for accessing the Pyplot module, which provides a simple interface for creating charts and plots.
  - `import numpy as np`: Often used alongside Matplotlib for numerical operations and generating data for visualizations.

Example 1: Standard Matplotlib Import.

```python
# Import Matplotlib's Pyplot module
import matplotlib.pyplot as plt

# Create a simple line plot
plt.plot([1, 2, 3, 4])
plt.title("Simple Line Chart")

plt.show()
```

Example 2: Using NumPy with Matplotlib.

```python
# Import Matplotlib and NumPy
import matplotlib.pyplot as plt
import numpy as np

# Generate data using NumPy
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Create a sine wave plot
plt.plot(x, y)
plt.title("Sine Wave")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")

plt.show()
```

Example 3: Importing 3D Toolkit.

```python
# Import Matplotlib's 3D plotting toolkit
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

# Generate data for 3D surface plot
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create a 3D surface plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')

plt.show()
```

1.2 Introduction to Matplotlib

- **Matplotlib Overview:**
  - Used for visualising datasets in **Jupyter Notebook** with `Pandas` and `Matplotlib`.
  - A Python library specialising in the creation of two-dimensional charts (including 3D charts).
  - One of the most widely used tools for graphical representation of data.
- **Pyplot Module:**
  - Provides a classic Python interface for the `matplotlib` library.
  - Requires the import of the `Numpy` package separately.
- **Code for Importing Pyplot:**

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

1.3 Customising Charts

1.3.1 Adding Titles and Text

- Use the `text()` function to add custom text annotations anywhere on the chart.
- Use the `title()` function to set the chart's title.
- Example code:

```
plt.axis([0, 5, 0, 20])
plt.title('My First plot', fontsize=20,
           fontname='Times New Roman')
plt.xlabel('Counting', color='gray')
plt.ylabel('Square values', color='gray')
plt.text(1, 1.5, 'First')
plt.text(2, 4.5, 'Second')
plt.text(3, 9.5, 'Third')
plt.text(4, 16.5, 'Fourth')
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.show()
```

1.3.2 Adding Grids and Legends

- Use the `grid()` function to add gridlines to your chart.
- Use the `legend()` function to add a legend to the chart.
- By default, the legend is added to the upper-right corner of the chart.
- The `loc` keyword argument in the `legend()` function allows you to change the location of the legend.
- The table below lists the location codes and their corresponding positions:

Location Code	**Location String**
0	best
1	upper right
2	upper left
3	lower left
4	lower right
5	right
6	center left
7	center right
8	lower center
9	upper center
10	center

Table 1.1: Legend Location Codes

Listing 1.1: Adding a Grid and a Legend

```
plt.axis([0, 5, 0, 20])
plt.title('My First plot', fontsize=20,
          fontname='Times New Roman')
plt.xlabel('Counting', color='gray')
plt.ylabel('Square values', color='gray')
plt.text(1, 1.5, 'First')
plt.text(2, 4.5, 'Second')
plt.text(3, 9.5, 'Third')
plt.text(4, 16.5, 'Fourth')
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.grid(True)
plt.legend(['First Series'], loc=2) # Upper-left corner
```

1.3.3 Handling Date Values

- Use `matplotlib.dates` to handle date values in plots.
- Set major and minor locators and formatters for the x-axis to display dates.
- Example code:

```
import datetime
import matplotlib.dates as mdates

# Define data
events = [datetime.date(2017, 1, 23), datetime.date(2017, 2, 3),
         datetime.date(2017, 2, 24), datetime.date(2017, 3, 15),
         datetime.date(2017, 3, 24), datetime.date(2017, 4, 8),
         datetime.date(2017, 4, 24)]

readings = [12, 22, 25, 20, 18, 15, 17, 14]

# Define locators and formatters
months = mdates.MonthLocator()
days = mdates.DayLocator()
timeFmt = mdates.DateFormatter('%Y-%m')

# Create the plot
fig, ax = plt.subplots()
plt.plot(events, readings)
ax.xaxis.set_major_locator(months)
ax.xaxis.set_major_formatter(timeFmt)
ax.xaxis.set_minor_locator(days)
plt.show()
```

1.4 Line Charts

Introduction to Line Charts

- The simplest chart is a line chart.
- A line chart represents a sequence of data points connected by a line.
- Each data point consists of a pair of values (x, y).
- Use the `color` and `linestyle` keyword arguments (`kwargs`) to define the appearance of the stroke.

Color Codes in Matplotlib

Code	Colour
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Table 1.2: Matplotlib color codes.

Examples of Line Charts

Listing 1.2: Basic Line Chart

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

x = np.arange(-2 * np.pi, 2 * np.pi, 0.01)
y = np.sin(3 * x) / x
plt.plot(x, y)
plt.show()
```

Listing 1.3: Customised Line Chart

```
x = np.arange(-2 * np.pi, 2 * np.pi, 0.01)
y = np.sin(3 * x) / x
plt.plot(x, y, color='r', linestyle='--', linewidth=5)
plt.show()
```

Multiple Line Charts in a Single Plot

Listing 1.4: Multiple Line Charts

```
x = np.arange(-2 * np.pi, 2 * np.pi, 0.01)
y1 = np.sin(3 * x) / x
y2 = np.sin(2 * x) / x
y3 = np.sin(x) / x

plt.plot(x, y1, 'k-', linewidth=3) # Black solid line
plt.plot(x, y2, 'g-', linewidth=2) # Green solid line
plt.plot(x, y3, 'r--', linewidth=5) # Red dashed line
plt.show()
```

Line Charts with DataFrames

- Data visualisation is straightforward using Pandas DataFrames.
- Pass a DataFrame as an argument to the `plot()` function to create multi-series line charts.

Listing 1.5: Line Chart with Pandas DataFrame

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = {'series1': [1, 3, 4, 3, 5],
        'series2': [2, 4, 5, 2, 4],
        'series3': [3, 2, 3, 1, 3]}
df = pd.DataFrame(data)
x = np.arange(5)

plt.axis([0, 5, 0, 7])
plt.plot(x, df)
plt.legend(data, loc=2) # Display legend in upper-left
    corner
plt.show()
```

1.5 Working with Multiple Figures and Axes

- The `subplot()` function is used to represent various subplots in a single figure by subdividing the figure into different drawing areas.
- It can also be used to focus the commands on a specific subplot.
- The argument passed to the `subplot()` function determines the mode of subdivision:
  - The first integer specifies how many parts the figure is split into vertically.
  - The second integer specifies how many parts the figure is split into horizontally.
  - The third integer selects the current subplot on which the commands will act.
- Below are examples of sinusoidal trends divided into subplots.

Example 1: Two horizontal subplots (211 and 212).

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

t = np.arange(0, 5, 0.1)
y1 = np.sin(2 * np.pi * t)
y2 = np.cos(2 * np.pi * t)

plt.subplot(211) # First subplot
plt.plot(t, y1, 'b-.') # Plot sine in blue dash-dot

plt.subplot(212) # Second subplot
plt.plot(t, y2, 'r--') # Plot cosine in red dashed line

plt.show()
```

Example 2: Two vertical subplots (121 and 122).

```
t = np.arange(0.1, 1, 0.05)
y1 = np.sin(2 * np.pi * t)
y2 = np.cos(2 * np.pi * t)

plt.subplot(121) # First vertical subplot
plt.plot(t, y1, 'b-.') # Plot sine in blue dash-dot

plt.subplot(122) # Second vertical subplot
plt.plot(t, y2, 'r--') # Plot cosine in red dashed line

plt.show()
```

1.6 Histograms

- A histogram consists of adjacent rectangles erected on the *x-axis*, split into discrete intervals called **bins**.
- The `hist()` function allows you to represent a histogram.
- **Practical example**: Let's generate a population of 100 random values from 0 to 100 using the `random.randint()` function.

Listing 1.6: Generating Random Population

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

pop = np.random.randint(0, 100, 100)
```

The generated population is then used to create a histogram.

- To create a histogram of these samples, use the `hist()` function.
- The `bins` kwarg divides the occurrences into specific intervals (default value is 10 bins).
- Example with 20 bins and the default 10 bins are shown below.

Listing 1.7: Creating a Histogram

```
n, bins, patches = plt.hist(pop, bins=20)

plt.show()
```

1.7 Bar Charts

1.7.1 Introduction to Bar Charts

- Bar charts are a common type of chart, similar to histograms, but the x-axis is used to reference categories.
- The `bar()` function is used to create a bar chart.

```
index = [0, 1, 2, 3, 4]
values = [5, 7, 3, 4, 6]
plt.bar(index, values)
plt.show()
```

1.7.2 Customizing Bar Charts with Tick Labels

- The tick labels are defined by a list of strings passed to the `xticks()` function.
- The location of ticks is specified using the first argument of `xticks()`.

```
index = np.arange(5)
values = [5, 7, 3, 4, 6]
plt.bar(index, values)
plt.xticks(index + 0.4, ['A', 'B', 'C', 'D', 'E'])
plt.show()
```

1.7.3 Adding Error Bars and Transparency

- Use the `yerr` kwarg to add error bars, combined with `error_kw` for customization.
- The `alpha` kwarg adjusts transparency of the bars.

```
index = np.arange(5)
values = [5, 7, 3, 4, 6]
errors = [0.8, 1.0, 0.4, 0.9, 1.3]
plt.bar(index, values, yerr=errors,
       error_kw={'ecolor': '0.1', 'capsize': 6}, alpha=0.7)

plt.xticks(index + 0.4, ['A', 'B', 'C', 'D', 'E'])
plt.legend(['Series 1'], loc=2)
plt.show()
```

1.7.4 Horizontal Bar Charts

- Horizontal bar charts are created using the `barh()` function.
- The roles of axes are reversed: categories are on the y-axis, and numerical values are on the x-axis.

```
plt.barh(index, values, xerr=errors,
       error_kw={'ecolor': '0.1', 'capsize': 6}, alpha=0.7)

plt.yticks(index + 0.4, ['A', 'B', 'C', 'D', 'E'])
plt.legend(['Series 1'], loc=5)
plt.show()
```

1.7.5 Multiseries Bar Charts

- Multiseries bar charts are created by stacking bars side by side.
- Adjust the bar positions using the `width` parameter.

```
values2 = [6, 4, 5, 2, 8]
width = 0.4

plt.bar(index, values, width=width, label='Series 1')
plt.bar(index + width, values2,
       width=width, label='Series 2')

plt.xticks(index + width / 2, ['A', 'B', 'C', 'D', 'E'])
plt.legend()
plt.show()
```

1.7.6 Multiseries Horizontal Bar Charts

- Similar to multiseries bar charts but oriented horizontally.

```
plt.barh(index, values, height=0.4, label='Series 1')
plt.barh(index + 0.4, values2, height=0.4, label='Series 2')
plt.yticks(index + 0.2, ['A', 'B', 'C', 'D', 'E'])
plt.legend()
plt.show()
```

1.7.7 Stacked Bar Charts with Pandas DataFrame

- Use Pandas DataFrame to create stacked bar charts for better handling of multiseries data.
- Sum the series directly to stack bars vertically.

```
import pandas as pd
data = pd.DataFrame({'Series 1': values, 'Series 2': values2},
                    index=['A', 'B', 'C', 'D', 'E'])

data.plot(kind='bar', stacked=True)
plt.show()
```

1.7.8 Bar Charts with Negative Values

- Represent one series with negative values using the `facecolor` kwarg.
- Add numerical values at the end of each bar using the `text()` function.

```
values3 = [-5, -7, -3, -4, -6]
plt.bar(index, values, width=0.4, label='Positive', color='blue')
plt.bar(index, values3, width=0.4, label='Negative',
        facecolor='red', bottom=values)
for i, v in enumerate(values):
    plt.text(i, v + 0.5, str(v), ha='center')

for i, v in enumerate(values3):
    plt.text(i, v - 0.5, str(v), ha='center')

plt.legend()
plt.show()
```

1.8 Pie Charts

1.8.1 Introduction to Pie Charts

- The `pie()` function is used to create pie charts in Matplotlib.
- A pie chart represents data as slices of a circle, with each slice proportional to the value it represents.
- Various kwargs are available for customization, including:
  - `explode`: Separates slices for emphasis.
  - `startangle`: Rotates the chart (default is `0`).
  - `autopct`: Adds percentage text labels to the center of each slice.
  - `shadow`: Adds a shadow effect to the chart.

1.8.2 Creating a Pie Chart

```
# Sample data
labels = ['A', 'B', 'C', 'D']
sizes = [25, 35, 20, 20]
explode = [0, 0.1, 0, 0] # Emphasize the second slice

plt.pie(sizes, labels=labels, explode=explode,
       autopct='%1.1f%%',
       shadow=True, startangle=90)

plt.title('Pie Chart Example')
plt.axis('equal') # Ensures the pie chart is a perfect circle
plt.show()
```

1.8.3 Pie Chart with Pandas DataFrame

- Pie charts can also be created directly from a Pandas DataFrame using the `plot()` function with the `kind='pie'` kwarg.
- The `figsize` kwarg ensures the pie chart is displayed circular.

```
import pandas as pd

# Sample DataFrame
data = pd.DataFrame({
    'Series 1': [25, 35, 20, 20],
    'Series 2': [15, 25, 35, 25],
    'Series 3': [30, 20, 25, 25]
}, index=['A', 'B', 'C', 'D'])

# Plotting the first series as a pie chart
data['Series 1'].plot(kind='pie', autopct='%1.1f%%',
                    shadow=True,
                    figsize=(6, 6),
                    startangle=90)

plt.title('Pie Chart from Pandas DataFrame')
plt.ylabel('') # Removes the y-axis label
plt.show()
```

1.9 Scatter Plots

1.9.1 Introduction to Scatter Plots

- A scatter plot is used to represent data points in a two-dimensional space, with one variable plotted along the x-axis and another along the y-axis.
- The `scatter()` function in Matplotlib is used to create scatter plots.
- Key customizations for scatter plots include:
  - `c`: Specifies the colors of the points. Accepts an array of values, a single color, or a colormap.
    - `'blue'`: Standard blue shade.
    - `'red'`: Vibrant red.
    - `'green'`: Natural green.
    - `'orange'`: Bright orange.
    - `'purple'`: Rich purple.
  - `marker`: Specifies the shape of the markers. Common options include:
    - `'o'`: Circle (default)
    - `'s'`: Square
    - `'`$^{\prime}: Triangleup$`'v'`$: Triangledown$
    - `'x'`: Cross
    - `'D'`: Diamond
  - `s`: Specifies the size of the markers.
  - `alpha`: Adjusts the transparency of the points (range: 0.0 to 1.0).

1.9.2 Creating a Basic Scatter Plot

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

# Generate random data
N = 100
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N) # Random colors
area = np.pi * (15 * np.random.rand(N))**2 # Marker sizes

# Create scatter plot
plt.scatter(x, y, s=area, c=colors, alpha=0.7, marker='o')
plt.title('Basic Scatter Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Adds a color scale legend
plt.colorbar(label='Color Scale')
plt.show()
```

1.9.3 Scatter Plot with Different Markers and Colors

- To differentiate data points, use the `c` kwarg for colors and `marker` kwarg for shapes.
- Below is an example with multiple marker styles.

```
# Generate additional data for different markers
x1, y1 = np.random.rand(N), np.random.rand(N)
x2, y2 = np.random.rand(N), np.random.rand(N)

# Scatter plots with different markers and colors
plt.scatter(x1, y1, color='blue', marker='^', label='Group
    1')
plt.scatter(x2, y2, color='red', marker='s', label='Group
    2')

plt.title('Scatter Plot with Different Markers')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend()
plt.show()
```

1.10 Advanced Chart - Contour Plot

- Contour plots are used to represent three-dimensional data in two dimensions using contour lines or filled regions.
- The `contour()` function draws contour lines, while the `contourf()` function fills the areas between the lines with colors.
- The data for contour plots is defined as a grid, with values at each grid point determining the height (or Z-value).
- Key arguments for customization:
  - `levels`: Specifies the number or specific values of contour levels.
  - `cmap`: Defines the color map (e.g., `'viridis'`, `'plasma'`, `'coolwarm'`).
  - `linewidths`: Adjusts the thickness of the contour lines.
  - `alpha`: Controls the transparency of the plot.

Example 1: Basic Contour Plot.

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

# Generate grid data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create a contour plot
plt.contour(X, Y, Z, levels=10, cmap='viridis')
plt.title('Contour Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.colorbar(label='Z-value')
plt.show()
```

Example 2: Filled Contour Plot.

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

# Generate grid data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create a filled contour plot
plt.contourf(X, Y, Z, levels=15, cmap='plasma', alpha=0.8)
plt.colorbar(label='Z-value')
plt.title('Filled Contour Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
```

1.11 Advanced Chart - 3D Surfaces with mplot3d

- The `mplot3d` toolkit in Matplotlib allows for the creation of 3D plots.
- Import `Axes3D` from `mpl_toolkits.mplot3d` to enable 3D plotting capabilities.
- Common types of 3D plots include:
  - **3D Surface Plots:** Uses `plot_surface()` to represent 3D surfaces based on meshgrid data.
  - **Randomly Elevated Surface Plots:** Applies random variations to surface heights.
  - **3D Scatter Plots:** Plots individual points in 3D space.
- Use `view_init()` to adjust the elevation and azimuthal angle of the 3D view.

Example 1: Basic 3D Surface Plot.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Generate grid data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create a 3D surface plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', edgecolor='none')
ax.set_title('3D Surface Plot')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
plt.show()
```

Example 2: Randomly Elevated 3D Surface Plot.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Generate grid data with random elevations
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2)) + 0.5 *
    np.random.randn(*X.shape)

# Create a 3D surface plot with random elevations
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap='plasma', edgecolor='none',
    alpha=0.8)
ax.set_title('Randomly Elevated 3D Surface Plot')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
plt.show()
```

Example 3: 3D Scatter Plot.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Generate random data points for scatter plot
N = 100

# Random values between -5 and 5
x = np.random.rand(N) * 10 - 5
y = np.random.rand(N) * 10 - 5
z = np.random.rand(N) * 10 - 5

colors = np.random.rand(N)
sizes = 100 * np.random.rand(N)

# Create a 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(x, y, z, c=colors, s=sizes,
                  cmap='cool', alpha=0.7)

ax.set_title('3D Scatter Plot')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_zlabel('Z-axis')
plt.colorbar(scatter, label='Color Scale')
plt.show()
```

1.12 Subplots Within Other Subplots

- A subplot within another subplot can be created using the `add_axes()` function.
- This approach allows you to define a smaller plotting area inside an existing plot by specifying the position and size as a list of relative coordinates.
- Example: Create a main line chart with an inner line chart in the top-right corner.
  - The `add_axes()` argument takes a list of four values:
    * The x-coordinate of the bottom-left corner of the inner subplot.
    * The y-coordinate of the bottom-left corner of the inner subplot.
    * The width of the inner subplot (relative to the main figure).
    * The height of the inner subplot (relative to the main figure).

Example: Line Chart with an Inner Line Chart in the Top Right.

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

# Main plot data
x_main = np.linspace(0, 10, 100)
y_main = np.sin(x_main)

# Inner plot data
x_inner = np.linspace(0, 2 * np.pi, 50)
y_inner = np.cos(x_inner)

# Create the main plot
fig, ax_main = plt.subplots()
ax_main.plot(x_main, y_main, label='Main Plot', color='blue')
ax_main.set_title('Main Plot with Inner Subplot')
ax_main.set_xlabel('X-axis')
ax_main.set_ylabel('Y-axis')
ax_main.legend()

# Add the inner subplot
ax_inner = fig.add_axes([0.65, 0.65, 0.25, 0.25]) # [x, y, width,
    height]
ax_inner.plot(x_inner, y_inner, label='Inner Plot', color='red')
ax_inner.set_title('Inner Plot', fontsize=10)
ax_inner.set_xticks([])
ax_inner.set_yticks([])

plt.show()
```