

Matplotlib for Data Visualization - Part 3

▼ Type

Data science masterclass

VII. Interactive and 3D Plotting in Matplotlib

Matplotlib allows for **interactive plotting** and **3D visualization**, making it easier to explore data dynamically. This section covers enabling interactive mode, using widgets, and creating 3D plots.

7.1 Enabling Interactive Mode (`plt.ion`)

Matplotlib has two modes:

- **Static Mode (default)**: Requires `plt.show()` to display plots.
- **Interactive Mode (`plt.ion`)**: Allows real-time updates without blocking execution.

Enabling Interactive Mode

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

plt.ion() # Enable interactive mode

x = np.linspace(0, 10, 100)
y = np.sin(x)

fig, ax = plt.subplots()
line, = ax.plot(x, y, label="Sine Wave")

for phase in np.linspace(0, 2*np.pi, 50):
```

```
line.set_ydata(np.sin(x + phase)) # Update data
plt.draw()
plt.pause(0.1) # Pause to update the plot

plt.ioff() # Disable interactive mode
plt.show()
```

◆ **Use case: Live updates** for streaming data visualization.

7.2 Creating Interactive Plots with Widgets (**matplotlib.widgets**)

Matplotlib provides built-in **interactive widgets** such as sliders, buttons, and checkboxes.

Using a Slider to Adjust a Sine Wave

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.widgets import Slider

x = np.linspace(0, 10, 100)
y = np.sin(x)

fig, ax = plt.subplots()
plt.subplots_adjust(bottom=0.25)

line, = ax.plot(x, y, label="Sine Wave")

# Add a slider
ax_slider = plt.axes([0.2, 0.1, 0.65, 0.03])
slider = Slider(ax_slider, "Frequency", 0.1, 5.0, valinit=1.0)

# Update function
def update(val):
    freq = slider.val
```

```
line.set_ydata(np.sin(freq * x))  
fig.canvas.draw_idle()
```

```
slider.on_changed(update)  
plt.legend()  
plt.show()
```

◆ **Use case: Exploring parameter changes dynamically.**

7.3 Introduction to **mplot3d** for 3D Plots

Matplotlib's **mplot3d** module enables **3D plotting** by adding an **Axes3D** object.

Creating a 3D Axes Object

```
import matplotlib.pyplot as plt  
from mpl_toolkits.mplot3d import Axes3D  
  
fig = plt.figure()  
ax = fig.add_subplot(111, projection='3d')  
  
plt.show()
```

◆ **Use case: Visualizing high-dimensional data.**

7.4 3D Line, Scatter, and Surface Plots

3D Line Plot

```
import numpy as np  
import matplotlib.pyplot as plt  
from mpl_toolkits.mplot3d import Axes3D  
  
fig = plt.figure()  
ax = fig.add_subplot(111, projection='3d')
```

```
t = np.linspace(0, 10, 100)
x = np.sin(t)
y = np.cos(t)
z = t

ax.plot(x, y, z, label="3D Line Plot", color="blue")
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
ax.legend()
plt.show()
```

3D Scatter Plot

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

x = np.random.rand(50)
y = np.random.rand(50)
z = np.random.rand(50)

ax.scatter(x, y, z, c=z, cmap="viridis", marker="o")
plt.show()
```

3D Surface Plot

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

X = np.linspace(-5, 5, 50)
Y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(X, Y)
Z = np.sin(np.sqrt(X**2 + Y**2))
```

```
ax.plot_surface(X, Y, Z, cmap="coolwarm")

plt.show()
```

◆ **Use case:** Visualizing 3D relationships and surfaces.

7.5 Interactive Plots with `plotly` & `mpl_interactions`

Matplotlib's interactivity is limited compared to libraries like **Plotly** and **mpl_interactions**.

Using Plotly for Interactive 3D Plots

```
import plotly.graph_objects as go
import numpy as np

t = np.linspace(0, 10, 100)
x, y, z = np.sin(t), np.cos(t), t

fig = go.Figure(data=[go.Scatter3d(x=x, y=y, z=z, mode='lines', marker=dict(c
olor=z, colorscale='Viridis'))])
fig.show()
```

◆ **Advantages:** Fully interactive with zoom, pan, and hover effects.

VIII. Handling Large Data with Matplotlib

Working with large datasets in Matplotlib can be slow due to memory and rendering limitations. Optimizing performance is essential for smooth visualization.

8.1 Performance Optimization Tips

Matplotlib can become slow when plotting millions of points. Here are key **optimization techniques**:

1. Reduce the Number of Points

- Instead of plotting **every data point**, use **downsampling**.
- Use `pandas.DataFrame.sample()` or `numpy.linspace()` to take fewer representative points.

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

# Generate large dataset
x = np.linspace(0, 100, 1000000) # 1 million points
y = np.sin(x)

# Downsample to 10,000 points
x_sampled = x[::100]
y_sampled = y[::100]

plt.plot(x_sampled, y_sampled)
plt.title("Optimized Plot with Downsampling")
plt.show()
```

2. Use `plot()` Instead of `scatter()`

- `plt.plot()` is **faster** than `plt.scatter()` for large datasets.
- **Reason:** `scatter()` processes each point individually.

```
# Faster Line Plot
plt.plot(x, y, linewidth=0.5)
```

```
# Slower Scatter Plot
plt.scatter(x, y, s=1) # s=1 reduces marker size to improve performance
```

3. Enable Rasterization

- Converts vectors into **bitmaps** for better rendering performance.

```
plt.plot(x, y, rasterized=True)
plt.savefig("large_plot.pdf", dpi=300) # Rasterized output
```

4. Adjust Figure Size and DPI

- Reducing **DPI (dots per inch)** and figure size reduces processing load.


```
plt.figure(figsize=(8, 4), dpi=72) # Lower DPI speeds up rendering
plt.plot(x, y)
plt.show()
```

8.2 Using Agg for Faster Rendering

Matplotlib supports different backends for rendering. The **Agg (Anti-Grain Geometry)** backend is **faster for large datasets**.

```
import matplotlib
matplotlib.use("Agg") # Set Agg backend (non-GUI)
import matplotlib.pyplot as plt

plt.plot(x, y)
plt.savefig("output.png") # Faster saving using Agg
```

 **Best Use Case:** When rendering plots in **scripts, Flask apps, and notebooks** without GUI.

8.3 Downsampling Large Data for Plotting

When dealing with millions of data points, **downsampling** reduces the number of points plotted while maintaining trends.

Using Pandas Downsampling

```
import pandas as pd

df = pd.DataFrame({"x": x, "y": y})
df_sampled = df.sample(frac=0.01) # Take 1% of data

plt.plot(df_sampled["x"], df_sampled["y"])
plt.show()
```

8.4 Using **datashader** & **mpl-scatter-density** for Big Data Visualization

Matplotlib **struggles** with millions of points. **Datashader** and **mpl-scatter-density** help visualize big data.

Datashader for Large-Scale Visualization

```
import datashader as ds
import datashader.transfer_functions as tf
from datashader.mpl_ext import dsshow

fig, ax = plt.subplots()
dsshow(df, ds.Point("x", "y"), norm="eq_hist", cmap="Blues", ax=ax)
plt.show()
```

◆ **Efficient** for **millions of points** without performance issues.

mpl-scatter-density for Large Scatter Plots

```
from mpl_scatter_density import ScatterDensityArtist

fig, ax = plt.subplots()
```



```
ax.add_artist(ScatterDensityArtist(ax, x, y))  
plt.show()
```

◆ Works like `scatter()`, but handles **millions of points smoothly**.

IX. Matplotlib with Pandas & Seaborn

9.1 Plotting Directly from Pandas DataFrames

Pandas **integrates directly** with Matplotlib for simple plots.

```
import pandas as pd  
  
df = pd.DataFrame({"A": np.random.randn(100), "B": np.random.randn(100)})  
df.plot(x="A", y="B", kind="scatter")  
plt.show()
```

9.2 Customizing Pandas Plots

```
df.plot(kind="bar", figsize=(10, 5), colormap="coolwarm", title="Custom Bar Chart")  
plt.show()
```

◆ **Customization:** Adjust figure size, colors, and add titles.

9.3 Using Matplotlib with Seaborn

Seaborn enhances Matplotlib for **statistical visualization**.

```
import seaborn as sns  
  
sns.scatterplot(data=df, x="A", y="B", hue="B", palette="coolwarm")  
plt.show()
```

9.4 Creating Pair Plots, Heatmaps, and Joint Plots

Pair Plot

```
sns.pairplot(df)
```

- ◆ **Useful for** checking variable relationships.

Heatmap

```
sns.heatmap(df.corr(), annot=True, cmap="coolwarm")
```

- ◆ **Best for** correlation analysis.

X. Automating and Embedding Matplotlib Plots

10.1 Automating Plot Generation

Generate plots in **loops** for different datasets.

```
for i in range(5):  
    plt.figure()  
    plt.plot(np.random.randn(100))  
    plt.title(f"Plot {i+1}")  
    plt.savefig(f"plot_{i+1}.png")
```

- ◆ **Useful for dashboards** or batch processing.

10.2 Embedding Matplotlib in Dashboards (Flask/Django)

Matplotlib plots can be embedded into **Flask/Django** web apps.

```
from flask import Flask, Response
import io

app = Flask(__name__)

@app.route("/plot.png")
def plot():
    fig, ax = plt.subplots()
    ax.plot([0, 1, 2, 3], [0, 1, 4, 9])

    buf = io.BytesIO()
    fig.savefig(buf, format="png")
    buf.seek(0)
    return Response(buf.read(), mimetype="image/png")

app.run(debug=True)
```

10.3 Exporting Plots for Reports

Save as PDF

```
plt.savefig("report.pdf")
```

◆ **Best for:** Research papers, presentations.

Save as SVG (Vector Format)

```
plt.savefig("vector_graph.svg")
```

◆ **Scalable** for high-quality **printing**.

Export for LaTeX Reports

```
plt.savefig("plot.pgf") # PGF format for LaTeX
```

XI. Advanced Topics in Matplotlib

11.1 Customizing 3D & Geographic Plots

- Use `mplot3d` for **3D plots**.
- Use `Basemap` for **geospatial data**.

11.2 Time-Series Data Visualization

```
import pandas as pd
```

```
df["date"] = pd.date_range(start="1/1/2022", periods=len(df))  
df.set_index("date").plot()
```

◆ **Perfect for:** Stock prices, weather trends.

11.3 Creating Publication-Quality Plots

- **Increase DPI** for clarity.
- Use **LaTeX labels** for professional output.

```
plt.rcParams["text.usetex"] = True  
plt.title(r"$E = mc^2$")
```