PYTHON CODES

A Beginner's Guide for Data Analysis & Plotting

Manually Creating Data and Plotting

```
import numpy as np
import matplotlib.pyplot as plt
# Create your own data array
x_manual = np.array([1, 2, 3, 4, 5])        # X data
y_manual = np.array([10, 25, 35, 50, 60])   # Y data
# Using 'o' tells the computer to draw points instead of connecting them with
#lines
plt.plot(x_manual, y_manual, 'o')
plt.xlabel("X Data")
plt.ylabel("Y Data")
```

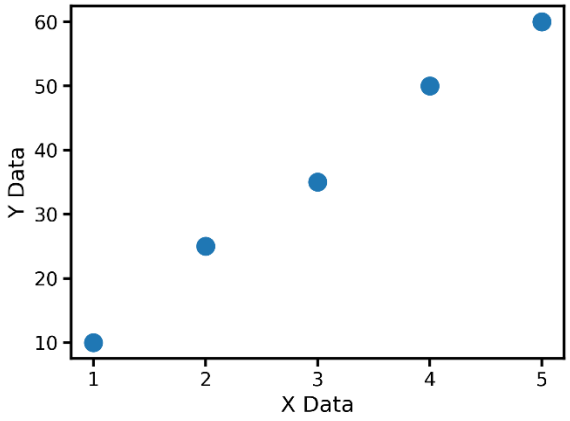

Figure 1. Plot created using manually defined arrays.

Uploading Files to the Session

Click the Folder Icon 📁 on the left sidebar.
Click the Upload Icon (page with up arrow).
Select your file and click OK.

Load CSV

```
import pandas as pd
```

```
df = pd.read_csv('filename.csv')
df.head()    # Displays the first few rows of your DataFrame
```

Load Excel

```
import pandas as pd

df = pd.read_excel('filename.xlsx')
df.head()    # Displays the first few rows of your DataFrame
```

Plotting the Data

```
import matplotlib.pyplot as plt

x = df.iloc[:, 0]    # Select first column as x
y = df.iloc[:, 1]    # Select second column as y
plt.plot(x, y, 'o')
plt.xlabel('x')
plt.ylabel('y')
```

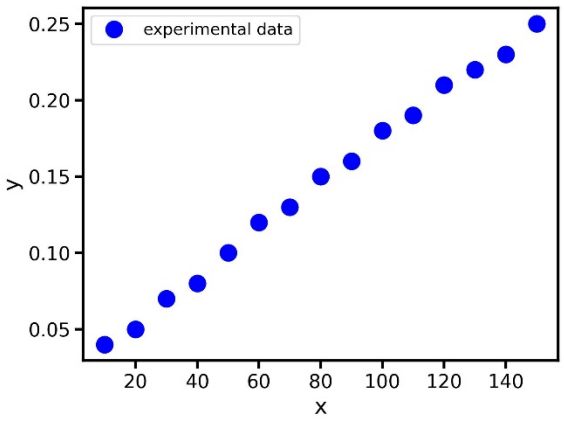

Figure 2. Scatter plot of experimental data.

Curve Fitting (Linear)

```
# polyfit finds the slope and the y-intercept
slope, intercept = np.polyfit(x, y, 1)
best_fit_line = slope * x + intercept
# Draw everything
plt.plot(x, y, 'o', color='blue', label='experimental data')
plt.plot(x, best_fit_line, color='red', label='Best Fit Line')
# Add labels and show
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.legend()     # Shows the labels for dots and the line
```

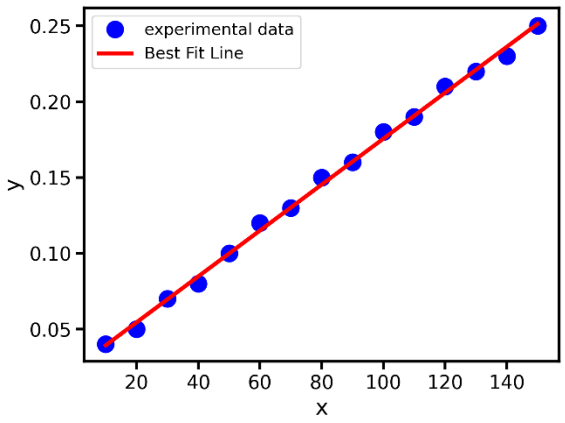

Figure 3. Linear regression analysis of experimental data.

Curve Fitting (Non-linear)

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
# Define the function (Exponential Decay: a.e^(-bx))
def model_func(x, a, b):
    return a * np.exp(-b * x)
# Load your data
df = pd.read_excel('your_file.xlsx')
x = df.iloc[:, 0]
y = df.iloc[:, 1]
# The 'Math Magic' (Finding the curve)
# This finds the best 'a' and 'b' to fit your dots
params, covariance = curve_fit(model_func, x, y)
a, b = params
# Draw the results
plt.plot(x, y, 'o', label='experimental data')
plt.plot(x, model_func(x, a, b), color='red', label='Exponential Fit')
# The curvy line
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

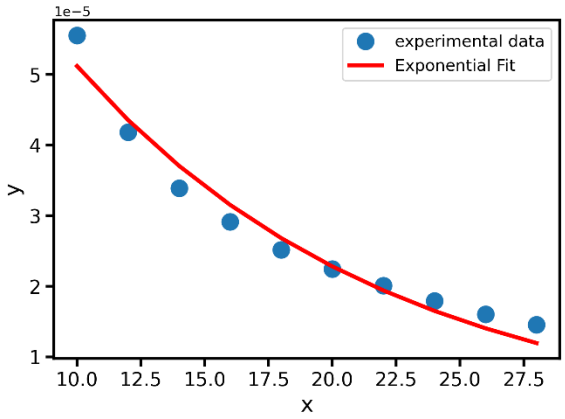

Figure 4. Best-fit exponential curve for experimental results.

Plotting Experimental Results with Error Bars

```
import matplotlib.pyplot as plt

# Create simple data points
x = [1, 2, 3, 4, 5]
y = [2, 4, 5, 7, 9]
# Define the amount of uncertainty
x_err = 0.2    # Horizontal error
y_err = 0.5    # Vertical error
# Plot using errorbar instead of plot
plt.errorbar(x, y, xerr=x_err, yerr=y_err, fmt='o',
             capsize=3,
             markersize=10,      # Makes the dots bigger
             elinewidth=2)       # Makes the error bars thicker
# Add basic labels
plt.xlabel("Time (s)")
plt.ylabel("Distance (m)")
```

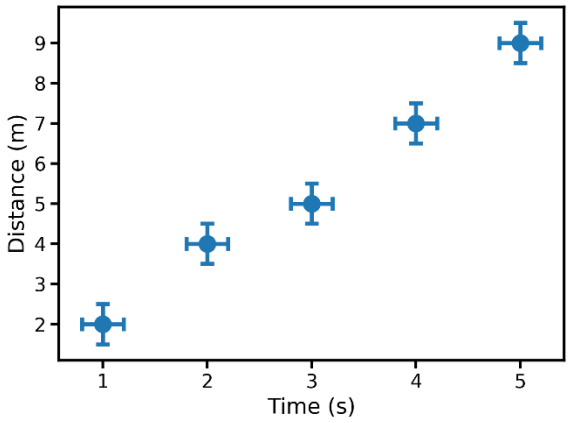

Figure 5. Data Visualization with Measurement Uncertainty.

Linear Spline Interpolation (Connecting the Dots)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
# 1. Define the data points for interpolation
x = np.array([0, 1, 2, 3, 4, 5])    # X coordinates of data
y = np.array([0, 2, 1, 3, 2, 5])    # Y coordinates of data
# 2. Create a linear interpolation function (connects points with straight
#lines)
linear_interp = interp1d(x, y, kind='linear')    # kind='linear' for
                                                 # piecewise linear
# 3. Generate new X values between the original points for a smooth line
x_new = np.linspace(x.min(), x.max(), 100)    # 100 points from X=0 to X=5
y_new = linear_interp(x_new)                  # Calculate interpolated Y
                                              #  values
# 4. Plot the original data and the linear interpolation line
plt.plot(x, y, 'o', label='Data Points')      # 'o' makes circular markers
plt.plot(x_new, y_new, label='Linear Spline') # Draw lines between points
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.show()    # Display the plot
```

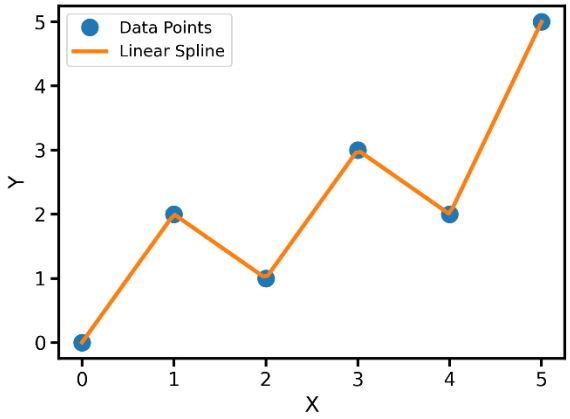

Figure 6. Linear Spline Interpolation.

Cubic Spline Interpolation (Smooth Curve Through Points)

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d
# 1. Define the data points
x = np.array([0, 1, 2, 3, 4, 5])
y = np.array([0, 2, 1, 3, 2, 5])
# 2. Create a cubic interpolation function (smooth curve through the points)
cubic_interp = interp1d(x, y, kind='cubic')     # kind='cubic' gives a cubic
                                                #  spline
# 3. Generate new X values and get interpolated Y values from the cubic
# spline
x_new = np.linspace(x.min(), x.max(), 100)     # 100 points from 0 to 5
y_new = cubic_interp(x_new)                    # Cubic-interpolated Y values
# 4. Plot the original data and the cubic spline interpolation
plt.plot(x, y, 'o', label='Data Points')       # Original data points
plt.plot(x_new, y_new, label='Cubic Spline')   # Smooth cubic spline through
                                               # points
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.show()
```

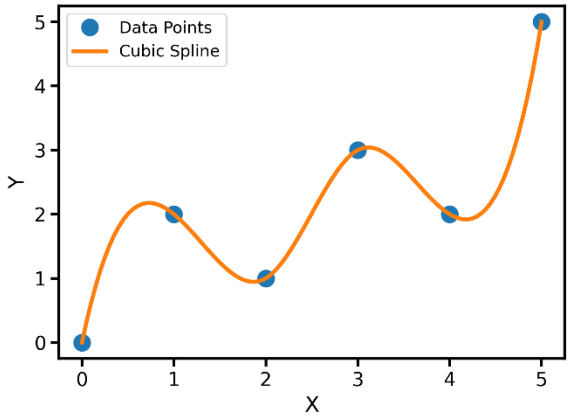

Figure 7. Cubic Spline Interpolation.

Data Slicing

```
import pandas as pd

df = pd.read_csv('filename.csv')
df.head()    # Displays the first few rows of your DataFrame

# Column Slicing (Vertical)

x = df.iloc[:, 0]    # All rows, 1st column
y = df.iloc[:, 1]    # All rows, 2nd column


# Picking Multiple Columns

first_three = df.iloc[:, 0:3]    # Columns 0, 1, and 2

# Picking Different Columns

specific_cols = df.iloc[:, [0, 3]]    # Only columns 0 and 3

# The Last Index Shortcut

last_row = df.iloc[-1, :]    # The final row
last_col = df.iloc[:, -1]    # The final column

# Row Slicing (Horizontal)

start_rows = df.iloc[0:10, :]    # Get the first 10 rows
row_five = df.iloc[4, :]         # Get a specific row (e.g., the 5th row)
middle_row = df.iloc[20:31, :]               # Grabs rows 20 through 30
```

Derivative of data (1-point derivative)

$$\frac{dx_i}{dt} = \frac{x_{i+1} - x_i}{t_{i+1} - t_i}$$

Method 1: Using the 'for' loop

```
import numpy as np

t = np.array([0.0, 1.0, 3.0, 3.5, 5.0])
x = np.array([0.0, 2.0, 8.0, 10.0, 20.0])

v_loop = []

for i in range(len(t) - 1):          # i from 0 to n-2
    dt = t[i+1] - t[i]
    dx = x[i+1] - x[i]
    v = dx / dt
    v_loop.append(v)

# Convert list to numpy array for nicer display
v_loop = np.array(v_loop)
```

Method 2: Using vectorization

```
import numpy as np

t = np.array([0.0, 1.0, 3.0, 3.5, 5.0])
x = np.array([0.0, 2.0, 8.0, 10.0, 20.0])

x_vec  = x[1:] - x[:-1]   # array of differences: [2, 6, 2, 10]
dt_vec = t[1:] - t[:-1]   # array of differences: [1, 2, 0.5, 1.5]

v_vec = dx_vec / dt_vec    # element-wise division

print("Velocities (vectorized):", v_vec)
```

Method 3: Using list comprehension

```
import numpy as np
# Step 1: create list of position differences
t = np.array([0.0, 1.0, 3.0, 3.5, 5.0])
x = np.array([0.0, 2.0, 8.0, 10.0, 20.0])

dx_list = [x[i+1] - x[i] for i in range(len(x)-1)]

# Step 2: create list of time differences
dt_list = [t[i+1] - t[i] for i in range(len(t)-1)]

# Step 3: pair them with zip and compute velocity
v_zip = [dx / dt for dx, dt in zip(dx_list, dt_list)]

print(v_zip)
```

Method 4: Using built in 'diff' function in Numpy Library

```
import numpy as np

t = np.array([0.0, 1.0, 3.0, 3.5, 5.0])
x = np.array([0.0, 2.0, 8.0, 10.0, 20.0])

dx_diff = np.diff(x)    # array([ 2.,  6.,  2., 10.])
dt_diff = np.diff(t)    # array([1. , 2. , 0.5, 1.5])

v_diff = dx_diff / dt_diff
```

Method 5: Solving coupled differential equations

$$F = ma = -kx$$

Since, $a = \frac{dv}{dt}$ and $v = \frac{dx}{dt}$, we split into two first-order equations:

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = -\frac{kx}{m}$$

```
import numpy as np

from scipy.integrate import solve_ivp
```

```
m = 1    #mass
k = 1    #constant
def system(t, y):
    x, v = y
    dxdt = v
    dvdt = (-k * x) / m
    return [dxdt, dvdt]
y0      = [1.0, 0.0]    #initial conditions>> y0[x,v]
t_eval = np.linspace(0, 20, 1000)
sol = solve_ivp(system, (0, 20), y0, t_eval=t_eval)
x, v, t = sol.y[0], sol.y[1], sol.t
```