

# Experiment 1

## Part 1: To solve the system of equations for

$$\begin{aligned} 2x+3y &= 6 \\ -4x+y &= -8 \end{aligned}$$

using matrix method

```
In [3]: import numpy as np

# Coefficient matrix A
A = np.array([
    [2, 3],
    [-4, 1]
])

# Constant matrix B
B = np.array([6, -8])

# Solve the system AX = B
X = np.linalg.solve(A, B)

print("Solution:")
print(f"x = {X[0]}")
print(f"y = {X[1]}")
```

Solution:

```
x = 2.142857142857143
y = 0.5714285714285714
```

## Part 2 : Write python code and visualize the method

```
In [4]: import numpy as np
import matplotlib.pyplot as plt

# System of equations:
# 2x + 3y = 6
# -4x + y = -8

# Coefficient matrix A
A = np.array([[2, 3],
              [-4, 1]])

# Constant matrix B
B = np.array([6, -8])

# Solve the matrix equation AX = B
solution = np.linalg.solve(A, B)
x_sol, y_sol = solution

print("Solution:")
```

```
print("x =", x_sol)
print("y =", y_sol)

# ----- Visualization -----
# Create x values for plotting
x = np.linspace(-5, 5, 400)

# Convert each equation to y = mx + c form
y1 = (6 - 2*x) / 3           # From 2x + 3y = 6
y2 = -8 + 4*x                # From -4x + y = -8

plt.figure(figsize=(7, 6))

# Plot the two lines
plt.plot(x, y1, label="2x + 3y = 6")
plt.plot(x, y2, label="-4x + y = -8")

# Mark the solution point
plt.scatter(x_sol, y_sol, color='red', s=80, label=f"Solution ({x_sol}, {y_sol})")

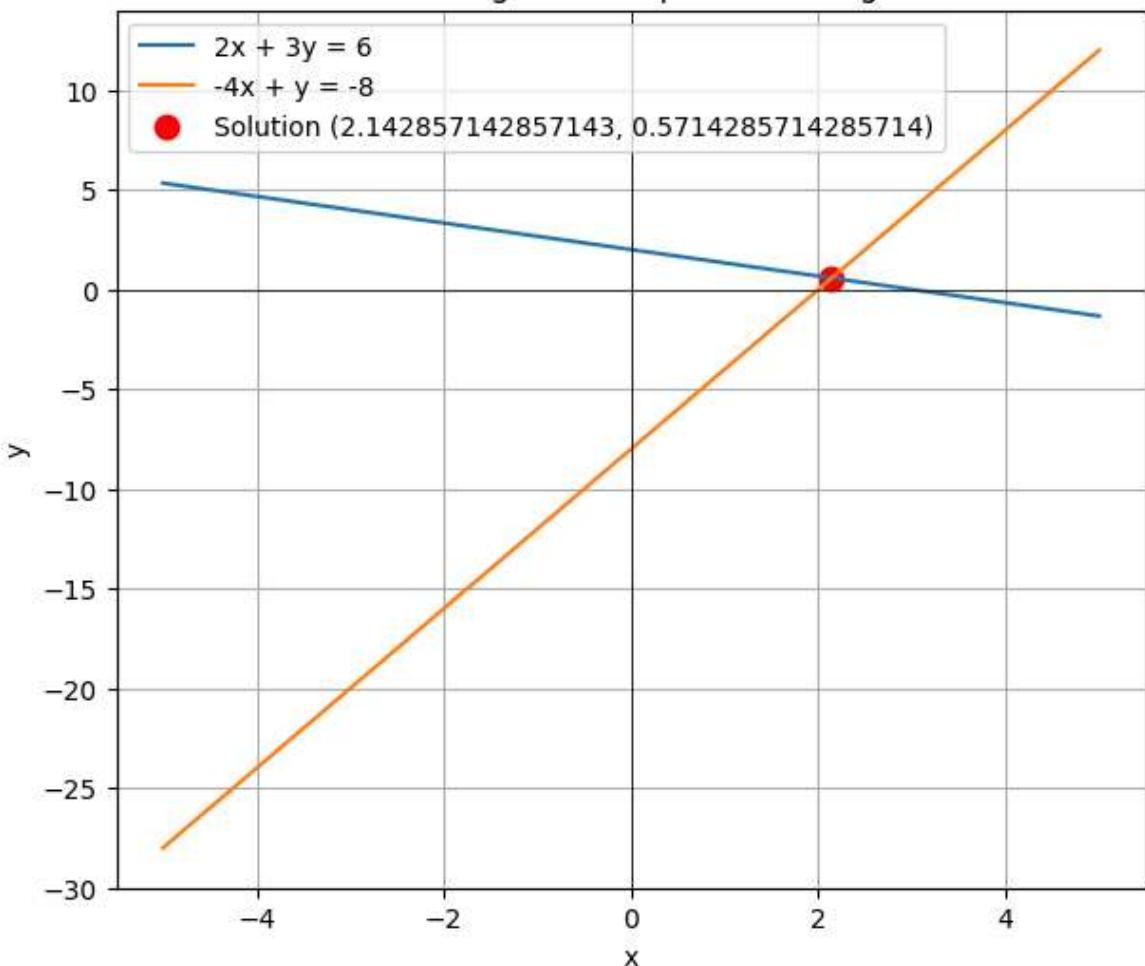
# Add labels and grid
plt.xlabel("x")
plt.ylabel("y")
plt.title("Visualization of Solving Linear Equations using Matrix Method")
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.legend()

plt.show()
```

Solution:

```
x = 2.142857142857143
y = 0.5714285714285714
```

### Visualization of Solving Linear Equations using Matrix Method



### Part 3 :

```
In [5]: import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D # noqa: F401

def plane_from_coeffs(coeffs):
    """Return (a,b,c,d) from [a,b,c] and d (so plane: a x + b y + c z = d)."""
    a, b, c = coeffs[0]
    d = coeffs[1]
    return a, b, c, d

def rank_and_solution(A, b, tol=1e-10):
    """Return ranks and classification and solution or paramization."""
    # ranks
    rank_A = np.linalg.matrix_rank(A, tol)
    aug = np.hstack([A, b.reshape(-1,1)])
    rank_aug = np.linalg.matrix_rank(aug, tol)

    result = {"rank_A": rank_A, "rank_aug": rank_aug}

    if rank_A == rank_aug == 3:
        # unique
        x = np.linalg.solve(A, b)
        result.update({"type": "unique", "solution": x})
    elif rank_A != rank_aug:
        result.update({"type": "inconsistent", "solution": None})

    return result
```

```

else:
    # rank_A == rank_aug < 3 => infinitely many solutions
    # find particular solution (Least squares) and nullspace basis
    x0, *_ = np.linalg.lstsq(A, b, rcond=None)
    # nullspace using SVD
    U, S, Vt = np.linalg.svd(A)
    # nullspace dimension
    null_mask = (S <= tol)
    # but S contains only min(m,n) singular values; better compute nullspace
    # Determine nullspace vectors as rows of Vt corresponding to zero singular values
    nullspace = Vt.T[:, np.where(S <= tol)[0]] if np.any(S <= tol) else Vt.T
    # if above fails, compute nullspace via SVD rank
    if nullspace.size == 0:
        # compute via SVD more robustly
        nullspace = Vt.T[:, rank_A:]
    result.update({"type": "infinite", "particular": x0, "nullspace": nullspace})
return result

def plot_planes_and_solution(case_name, planes, analysis):
    """
    planes: list of tuples (a,b,c,d) representing a x + b y + c z = d
    analysis: output from rank_and_solution
    """
    # Create grid for plotting planes
    xx, yy = np.meshgrid(np.linspace(-5,5,40), np.linspace(-5,5,40))
    fig = plt.figure(figsize=(8,6))
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title(case_name + " - " + analysis["type"])
    ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('z')
    ax.set_xlim(-5,5); ax.set_ylim(-5,5); ax.set_zlim(-5,5)

    # Plot planes
    for i, (a,b,c,d) in enumerate(planes):
        # avoid division by zero for c; if c is zero, solve for y instead
        if abs(c) > 1e-8:
            z = (d - a*xx - b*yy) / c
            ax.plot_surface(xx, yy, z, alpha=0.45, linewidth=0, rstride=1, cstride=1)
        elif abs(b) > 1e-8:
            # solve for y: y = (d - a x - c z)/b -> treat z values from mesh as constants
            zz, xx2 = np.meshgrid(np.linspace(-5,5,40), np.linspace(-5,5,40))
            yy2 = (d - a*xx2 - c*zz) / b
            ax.plot_surface(xx2, yy2, zz, alpha=0.45, linewidth=0)
        else:
            # solve for x: x = d / a (vertical plane)
            x_const = d / a
            yy2, zz2 = np.meshgrid(np.linspace(-5,5,40), np.linspace(-5,5,40))
            xx2 = np.full_like(yy2, x_const)
            ax.plot_surface(xx2, yy2, zz2, alpha=0.45, linewidth=0)

    # Plot according to type
    if analysis["type"] == "unique":
        x, y, z = analysis["solution"]
        ax.scatter([x], [y], [z], s=120)
        ax.text(x, y, z, f" ({x:.2f},{y:.2f},{z:.2f})", size=10)
    elif analysis["type"] == "inconsistent":
        # No common intersection; nothing extra to plot
        pass
    elif analysis["type"] == "infinite":
        # Determine nullspace dimension
        nullspace = analysis["nullspace"]

```

```

x0 = analysis["particular"].flatten()
ns_dim = nullspace.shape[1]
if ns_dim == 1:
    # intersection is a line: param t along nullspace vector
    v = nullspace[:,0].flatten()
    t = np.linspace(-10,10,200)
    line = x0.reshape(3,1) + np.outer(v, t)
    ax.plot(line[0,:], line[1,:], line[2,:], linewidth=3)
    ax.scatter([x0[0]], [x0[1]], [x0[2]], s=60)
    ax.text(x0[0], x0[1], x0[2], " particular", size=9)
elif ns_dim == 2:
    # infinite plane of solutions: plot the plane of solutions spanning
    v1 = nullspace[:,0].flatten()
    v2 = nullspace[:,1].flatten()
    s = np.linspace(-5,5,30)
    t = np.linspace(-5,5,30)
    S, T = np.meshgrid(s,t)
    P = x0.reshape(3,1,1) + v1.reshape(3,1,1)*S + v2.reshape(3,1,1)*T
    ax.plot_surface(P[0,:,:], P[1,:,:], P[2,:,:], alpha=0.6)
    ax.scatter([x0[0]], [x0[1]], [x0[2]], s=60)
    ax.text(x0[0], x0[1], x0[2], " particular", size=9)

plt.tight_layout()
plt.show()

# --- Example cases ---
cases = {
    "Unique intersection (single point)": {
        "# planes defined as [(a,b,c), d]"
        "planes": [
            (np.array([1, 1, 1]), 3),
            (np.array([2, -1, 1]), 0),
            (np.array([1, 2, -1]), 1)
        ]
    },
    "No common intersection (inconsistent)": {
        "# Make two planes parallel but inconsistent, and a third plane that intersects them"
        "planes": [
            (np.array([1, 1, 1]), 1),
            (np.array([2, 2, 2]), 3), # inconsistent multiple of first but different
            (np.array([1, -1, 0]), 0)
        ]
    },
    "Infinitely many solutions - line (two same planes)": {
        "# First two are multiples (same geometric plane), third intersects them"
        "planes": [
            (np.array([1, 1, 1]), 4),
            (np.array([2, 2, 2]), 8), # same plane as first
            (np.array([1, -1, 0]), 0)
        ]
    },
    "Infinitely many solutions - plane (all same plane)": {
        "planes": [
            (np.array([1, 2, -1]), 3),
            (np.array([2, 4, -2]), 6), # same as first
            (np.array([3, 6, -3]), 9) # same plane
        ]
    }
}

```

```
# Run through examples, analyze and plot
for name, data in cases.items():
    planes = [plane_from_coeffs(p) for p in data["planes"]]
    A = np.vstack(([p[0] for p in data["planes"]])).astype(float)
    b = np.array([p[1] for p in data["planes"]], dtype=float)
    analysis = rank_and_solution(A, b)
    print("Case:", name)
    print("Coefficient matrix A:\n", A)
    print("Right-hand side b:", b)
    print("Rank(A):", analysis["rank_A"], "Rank([A|b]):", analysis["rank_aug"])
    if analysis["type"] == "unique":
        sol = analysis["solution"]
        print("Unique solution:", sol)
    elif analysis["type"] == "inconsistent":
        print("No solution (inconsistent system).")
    else:
        x0 = analysis["particular"].flatten()
        ns = analysis["nullspace"]
        print("Infinite solutions. Particular solution (one):", x0)
        print("Nullspace basis vectors (columns):\n", ns)
        print("Nullspace dimension:", ns.shape[1])
    print("\n--- Visualizing ---\n")
    plot_planes_and_solution(name, planes, analysis)

print("Done. Replace the plane coefficients in the 'cases' dict to test your own")
```

Case: Unique intersection (single point)

Coefficient matrix A:

```
[[ 1.  1.  1.]
 [ 2. -1.  1.]
 [ 1.  2. -1.]]
```

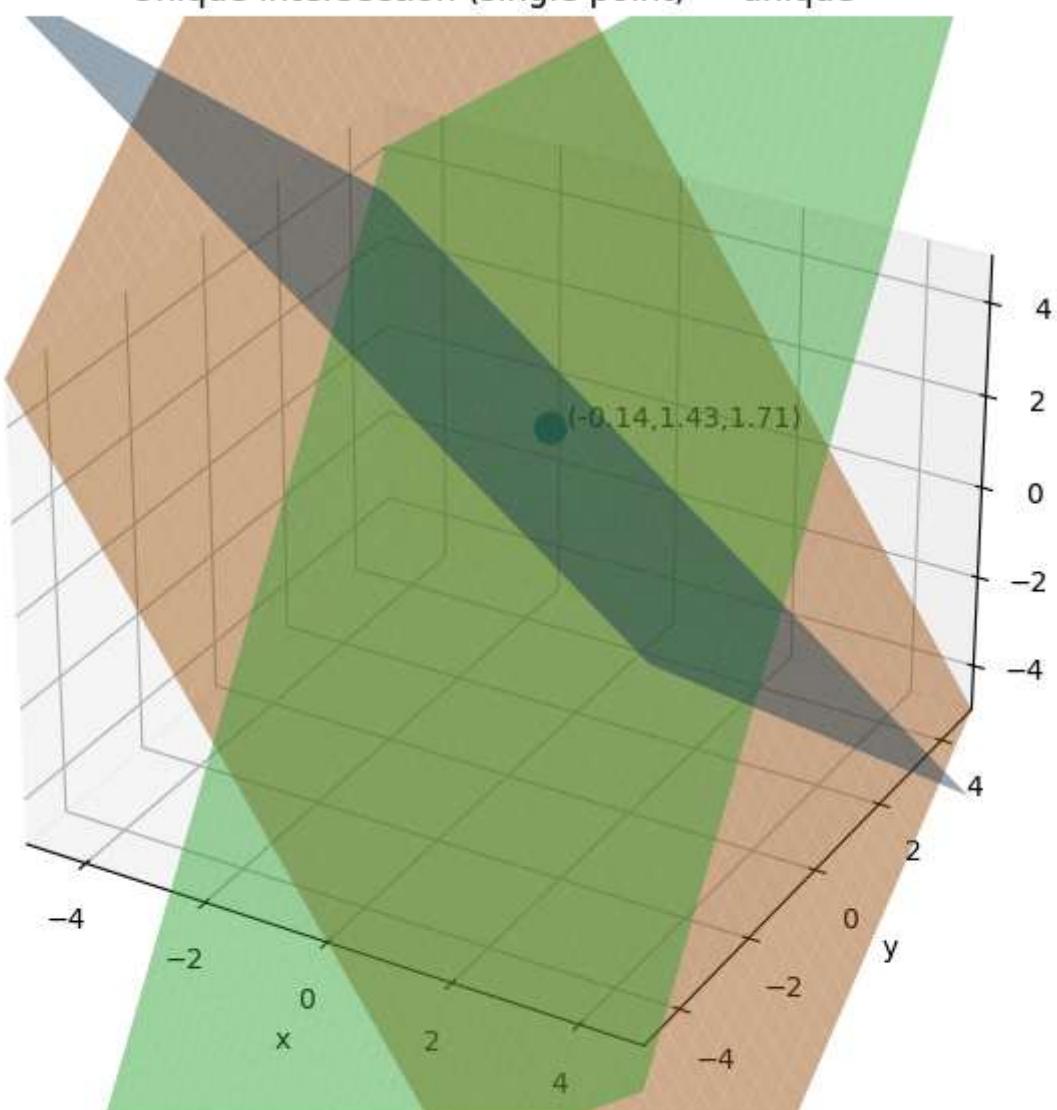
Right-hand side b: [3. 0. 1.]

Rank(A): 3 Rank([A|b]): 3

Unique solution: [-0.14285714 1.42857143 1.71428571]

--- Visualizing ---

### Unique intersection (single point) — unique



Case: No common intersection (inconsistent)

Coefficient matrix A:

```
[[ 1.  1.  1.]
 [ 2.  2.  2.]
 [ 1. -1.  0.]]
```

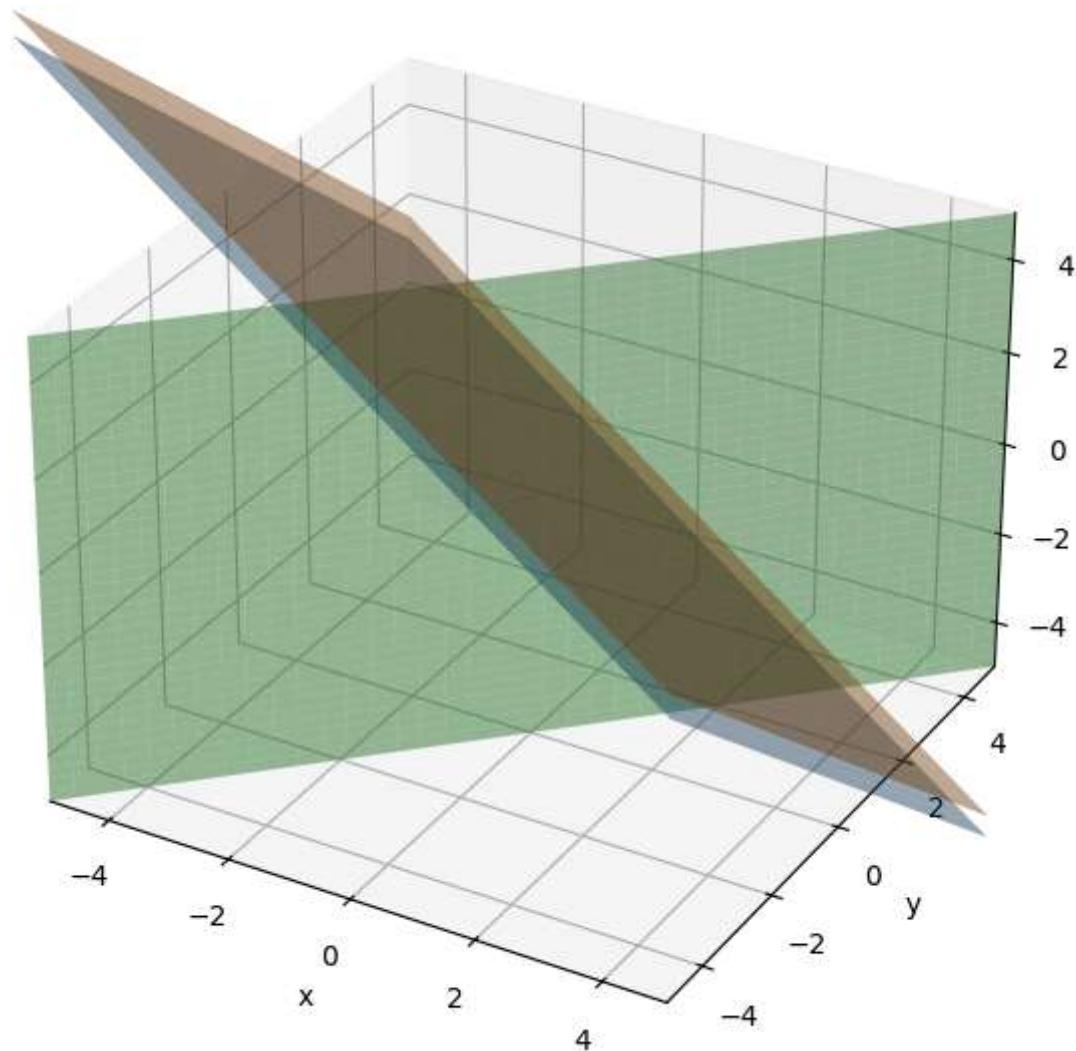
Right-hand side b: [1. 3. 0.]

Rank(A): 2 Rank([A|b]): 3

No solution (inconsistent system).

--- Visualizing ---

## No common intersection (inconsistent) — inconsistent



Case: Infinitely many solutions - line (two same planes)

Coefficient matrix A:

```
[[ 1.  1.  1.]
 [ 2.  2.  2.]
 [ 1. -1.  0.]]
```

Right-hand side b: [4. 8. 0.]

Rank(A): 2 Rank([A|b]): 2

Infinite solutions. Particular solution (one): [1.33333333 1.33333333 1.33333333]

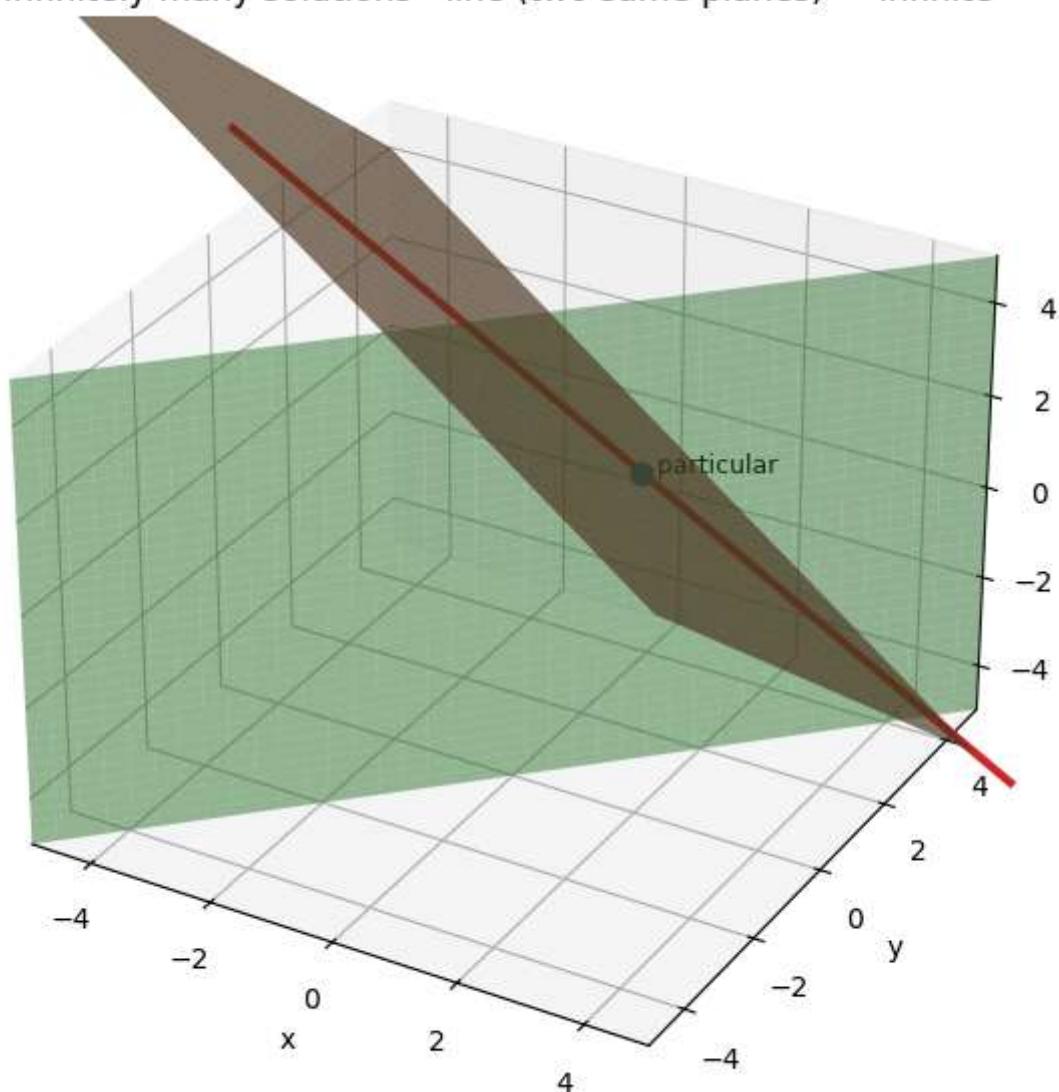
Nullspace basis vectors (columns):

```
[[ 0.40824829]
 [ 0.40824829]
 [-0.81649658]]
```

Nullspace dimension: 1

--- Visualizing ---

## Infinitely many solutions - line (two same planes) — infinite



Case: Infinitely many solutions - plane (all same plane)

Coefficient matrix A:

```
[[ 1.  2. -1.]
 [ 2.  4. -2.]
 [ 3.  6. -3.]]
```

Right-hand side b: [3. 6. 9.]

Rank(A): 1 Rank([A|b]): 1

Infinite solutions. Particular solution (one): [ 0.5 1. -0.5]

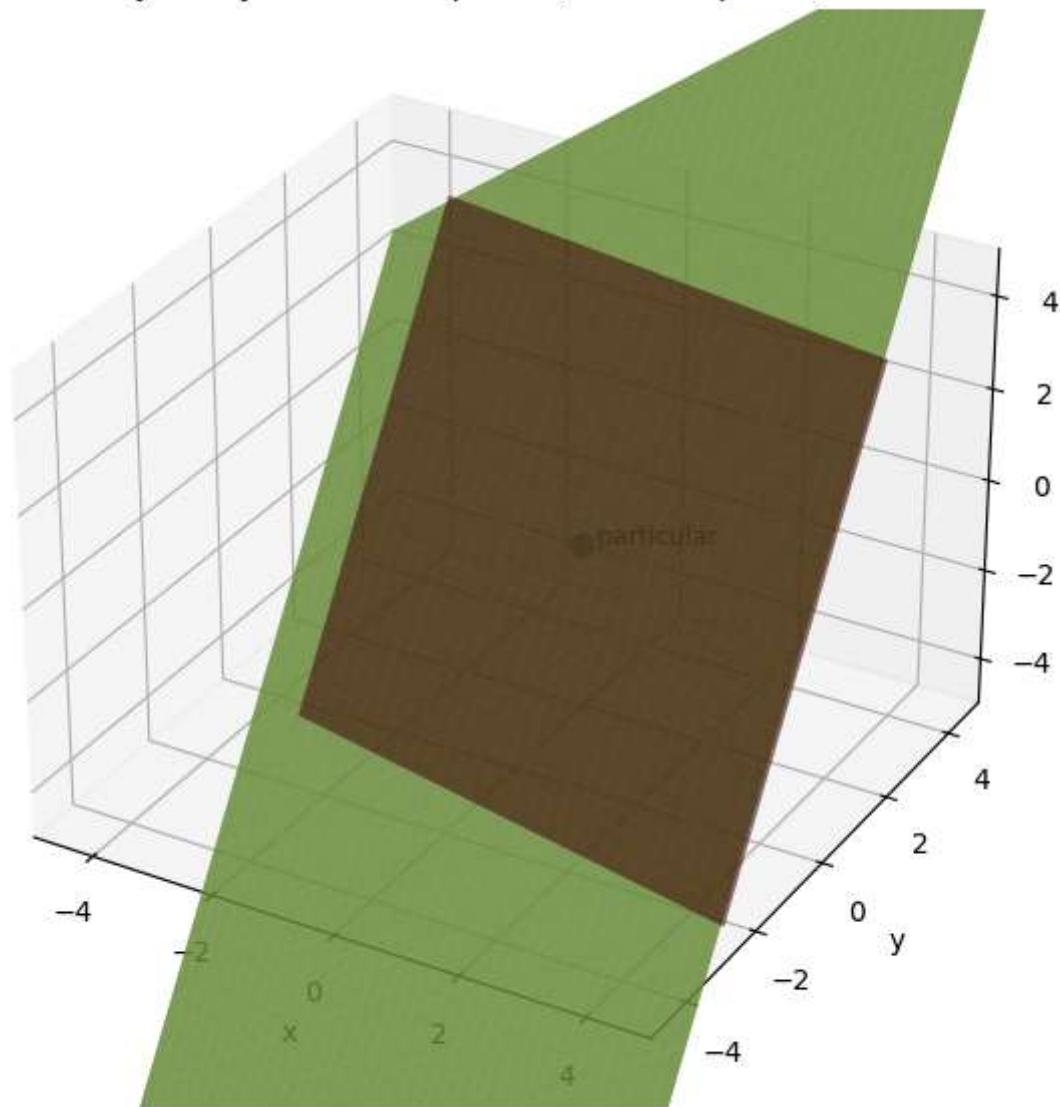
Nullspace basis vectors (columns):

```
[[-0.91287093  0.        ]
 [ 0.36514837  0.4472136 ]
 [-0.18257419  0.89442719]]
```

Nullspace dimension: 2

--- Visualizing ---

Infinitely many solutions - plane (all same plane) — infinite



Done. Replace the plane coefficients in the 'cases' dict to test your own examples.

## Experiment 1.2

```
In [6]: import numpy as np
import matplotlib.pyplot as plt

# Helper function to compute and print stats
def stats(name, data):
    mean = np.mean(data, axis=0)
    median = np.median(data, axis=0)
    print(f"{name} Mean: {mean}, Median: {median}")
    return mean, median

# ===== A) 2D Uniform Distribution =====
np.random.seed(42)
uniform_points = np.random.uniform(low=0, high=1, size=(10000, 2))

mean_u, median_u = stats("Uniform (original)", uniform_points)

plt.figure(figsize=(6,5))
plt.scatter(uniform_points[:,0], uniform_points[:,1], s=5)
```

```
plt.title("Uniform Distribution (10,000 points)")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

# B) Adding outliers
outliers = np.array([[3,3], [4,4], [5,5]])
uniform_with_outliers = np.vstack([uniform_points, outliers])

mean_u_o, median_u_o = stats("Uniform (with outliers)", uniform_with_outliers)

plt.figure(figsize=(6,5))
plt.scatter(uniform_with_outliers[:,0], uniform_with_outliers[:,1], s=5)
plt.title("Uniform Distribution with Outliers")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

# ===== Gaussian Distribution =====
gaussian_points = np.random.normal(loc=0, scale=1, size=(10000, 2))

mean_g, median_g = stats("Gaussian (original)", gaussian_points)

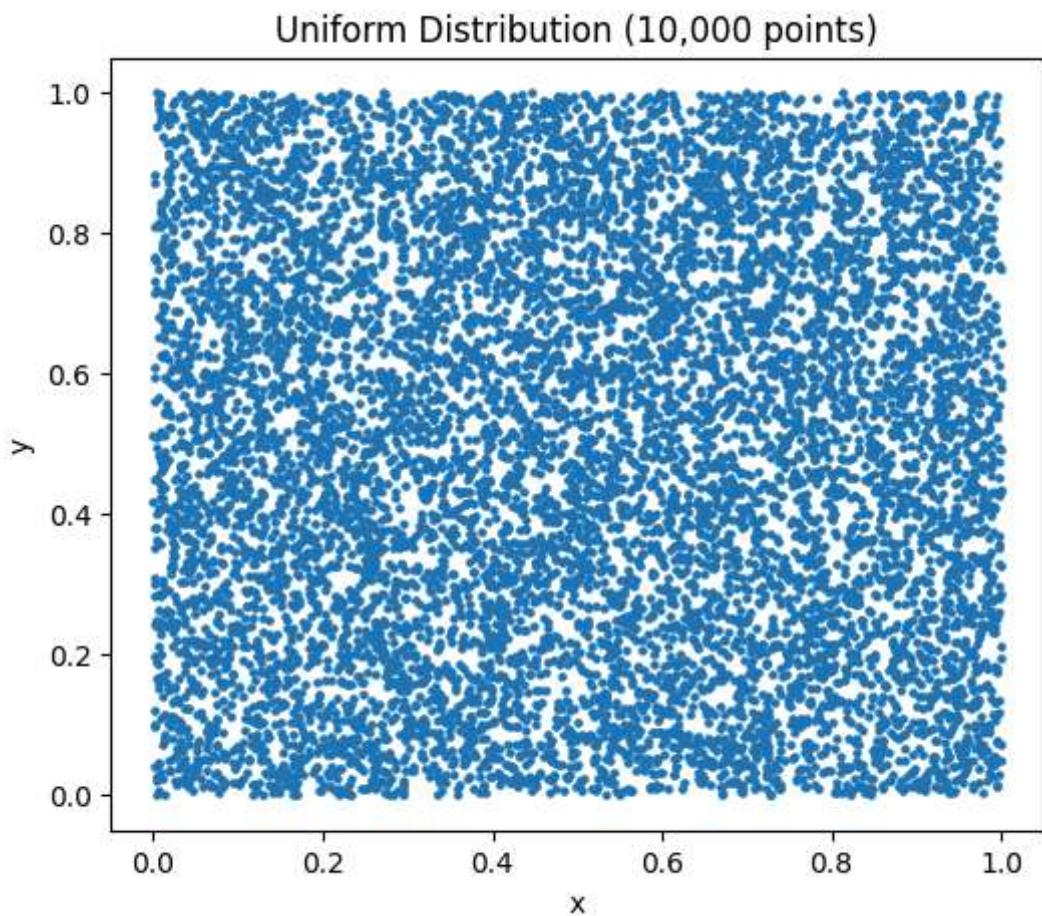
plt.figure(figsize=(6,5))
plt.scatter(gaussian_points[:,0], gaussian_points[:,1], s=5)
plt.title("Gaussian Distribution (10,000 points)")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

# Adding outliers for Gaussian
gaussian_with_outliers = np.vstack([gaussian_points, outliers])

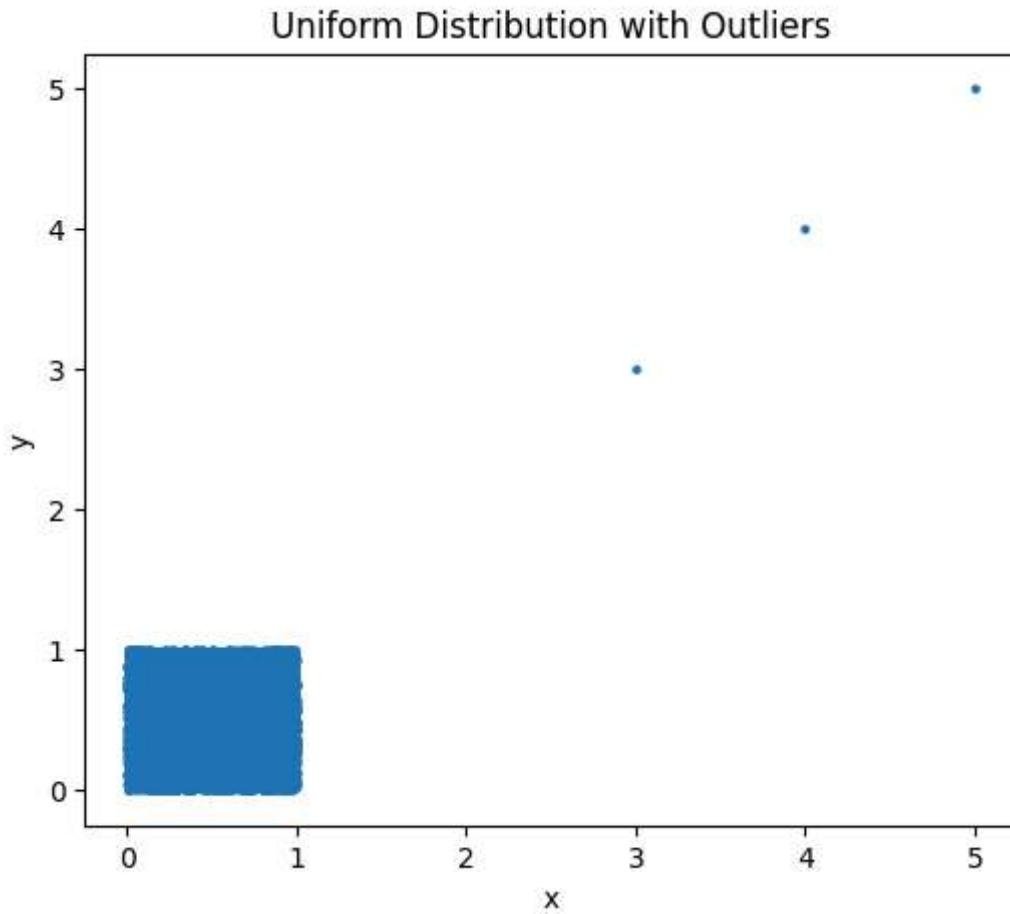
mean_g_o, median_g_o = stats("Gaussian (with outliers)", gaussian_with_outliers)

plt.figure(figsize=(6,5))
plt.scatter(gaussian_with_outliers[:,0], gaussian_with_outliers[:,1], s=5)
plt.title("Gaussian Distribution with Outliers")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

Uniform (original) Mean: [0.49847358 0.50021585], Median: [0.49835983 0.50027152]

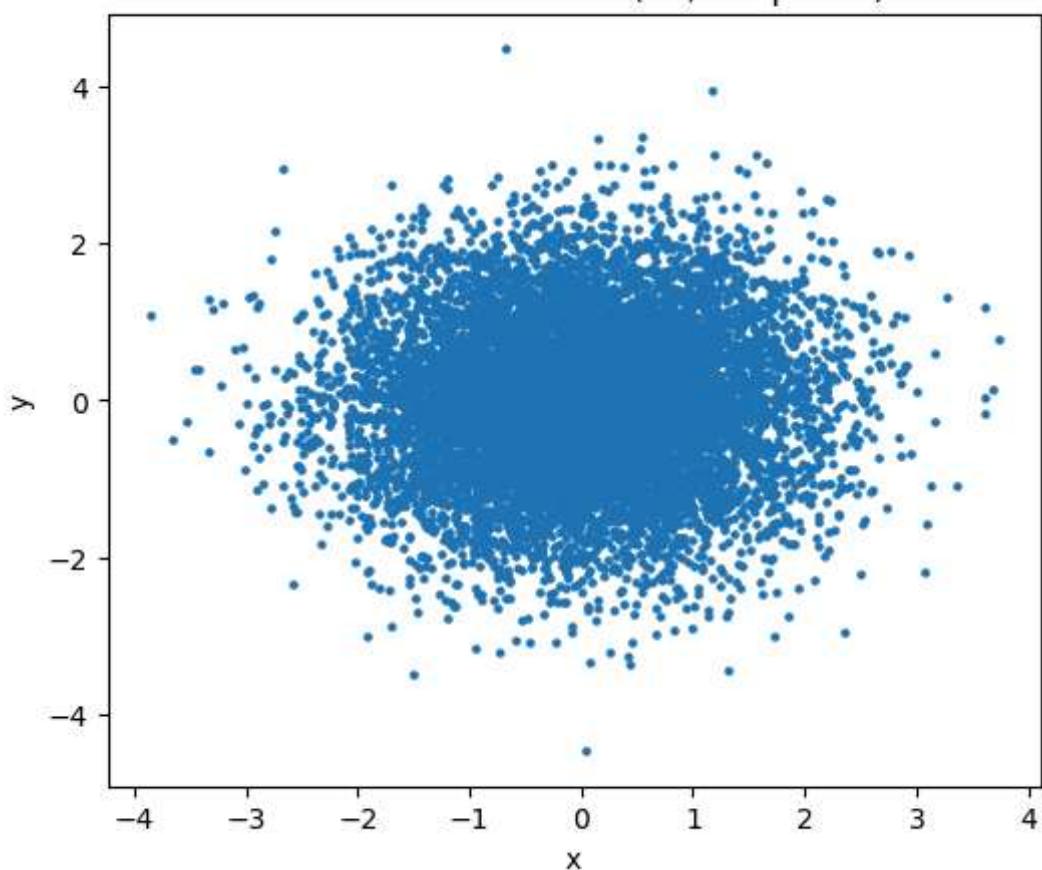


Uniform (with outliers) Mean: [0.49952372 0.50126547], Median: [0.49840754 0.50035836]



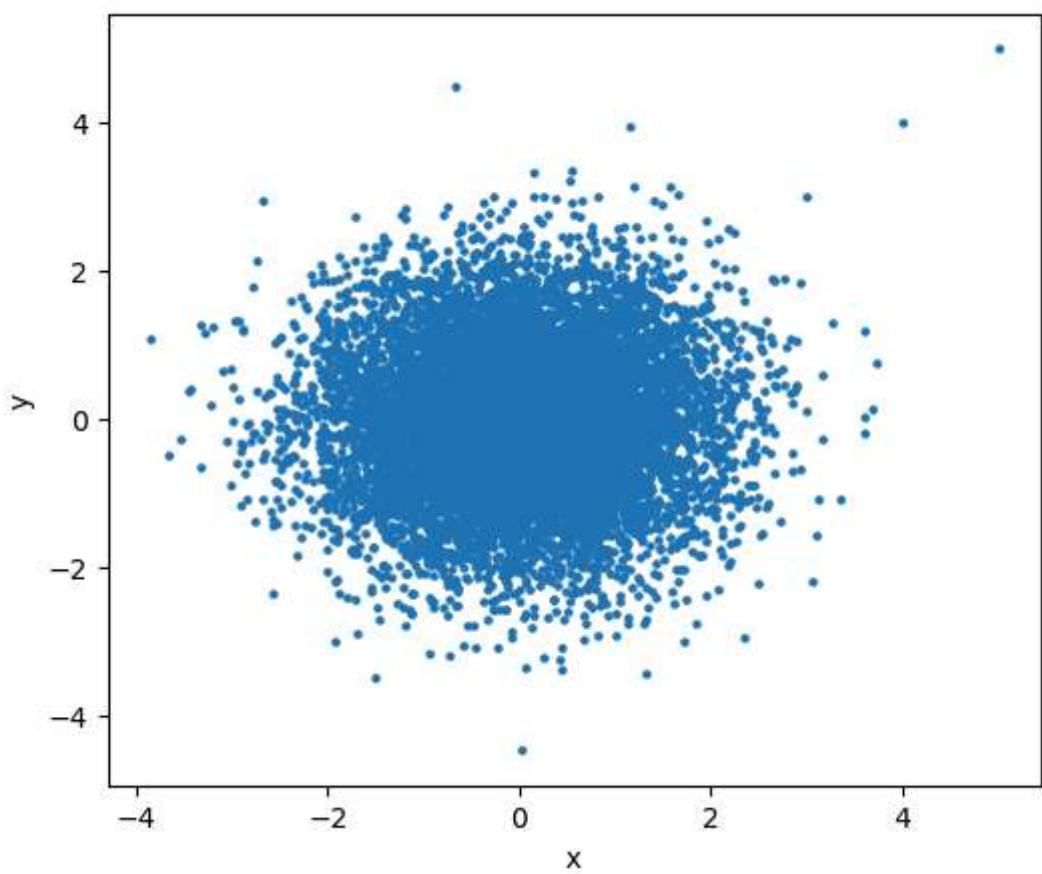
Gaussian (original) Mean: [-0.00449359 -0.00775959], Median: [ 0.00150458 -0.00592736]

Gaussian Distribution (10,000 points)



Gaussian (with outliers) Mean: [-0.00329261 -0.00655763], Median: [ 0.00173494 0.00572284]

Gaussian Distribution with Outliers



# Experiment 2

Greenfield town is seeing rapid real estate development. A real estate company wants to predict house prices based solely on the size of the house (in sq ft) they have collected data from 2 recently sold houses.

Size (sqft)	Price (\$)
1200	300000
1500	320000
2000	450000
1800	400000
2500	550000
3000	600000
2200	480000
2700	700000

```
In [48]: import numpy as np
import matplotlib.pyplot as plt
# Data
sizes = np.array([1200, 1500, 2000, 1800, 2500, 3000, 2200, 2700])
prices = np.array([300000, 320000, 450000, 400000, 550000, 600000, 480000, 700000])
```

```
# Means
x_mean = sizes.mean()
y_mean = prices.mean()

# Compute slope m manually
numerator = np.sum((sizes - x_mean) * (prices - y_mean))
denominator = np.sum((sizes - x_mean)**2)
m = numerator / denominator

# Compute intercept b manually
b = y_mean - m * x_mean

# Predictions
y_pred = m * sizes + b

print(f"Slope (Coefficient): {m:.2f}")
print(f"Intercept: {b:.2f}")
```

Slope (Coefficient): 212.17  
 Intercept: 26789.65

```
In [49]: import numpy as np

# Data
sizes = np.array([1200, 1500, 2000, 1800, 2500, 3000, 2200, 2700])
prices = np.array([300000, 320000, 450000, 400000, 550000, 600000, 480000, 700000])
```

```
# Construct X matrix with a column of 1s
X = np.column_stack((np.ones(len(sizes)), sizes))

# y vector
y = prices.reshape(-1, 1)

# Closed-form solution  $\theta = (X^T X)^{-1} X^T y$ 
theta = np.linalg.inv(X.T @ X) @ X.T @ y

theta0 = theta[0, 0] # intercept
theta1 = theta[1, 0] # slope

print(f"Theta0 (Intercept): {theta0:.2f}")
print(f"Theta1 (Slope): {theta1:.2f}")
```

Theta0 (Intercept): 26789.65

Theta1 (Slope): 212.17

In [50]:

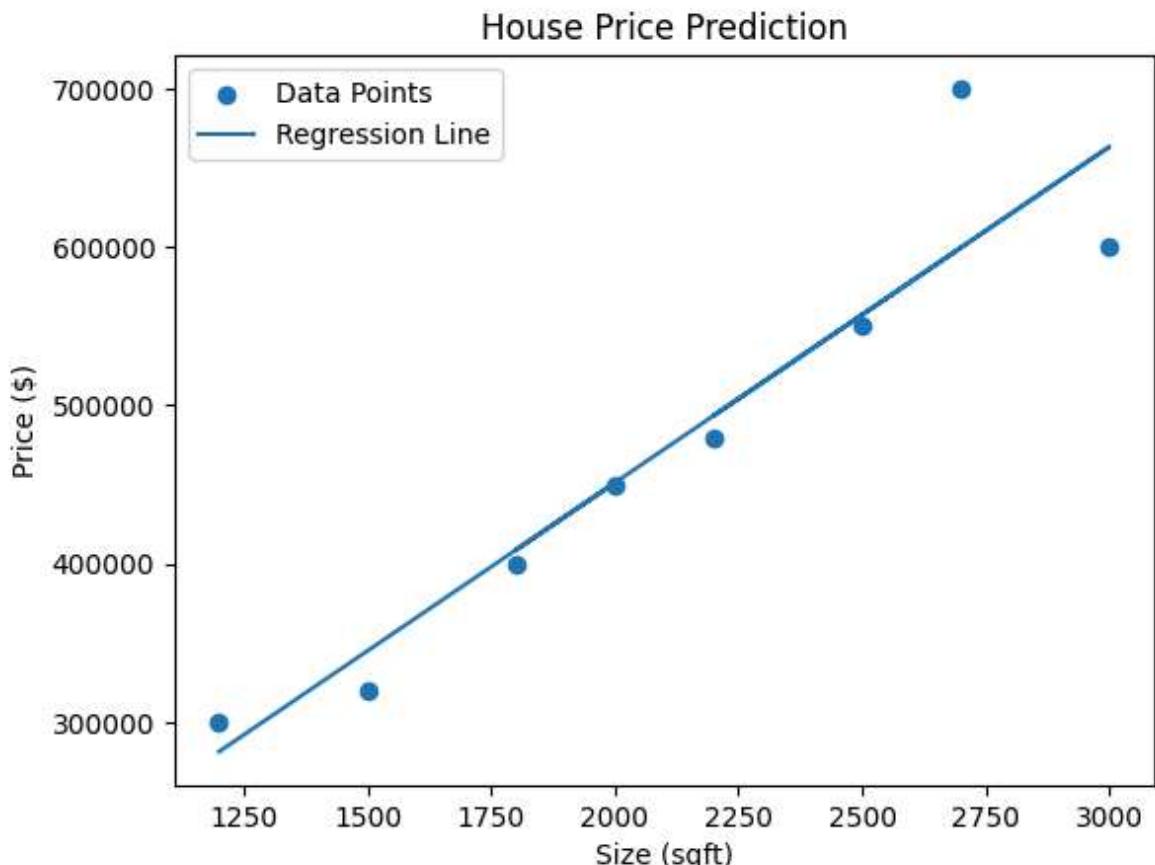
```
xpred = 1940
y_pred = m * xpred + b
print(f"Predicted price for size {xpred} sqft: {y_pred:.2f}")
```

Predicted price for size 1940 sqft: 438400.57

In [51]:

```
y_line = m * sizes + b

plt.scatter(sizes, prices, label='Data Points')
plt.plot(sizes, y_line, label='Regression Line')
plt.xlabel('Size (sqft)')
plt.ylabel('Price ($)')
plt.title('House Price Prediction')
plt.legend()
plt.show()
```



## Experiment 2.2

```
In [52]: import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.linear_model import LinearRegression

data=pd.read_csv('../assets/exp2_data.csv')
```

```
In [53]: data.head(5)
```

Out[53]:

	YearsExperience	Salary
<b>0</b>	1.1	39343.0
<b>1</b>	1.3	46205.0
<b>2</b>	1.5	37731.0
<b>3</b>	2.0	43525.0
<b>4</b>	2.2	39891.0

```
In [54]: from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Prepare X and y
X = data['YearsExperience'].values.reshape(-1, 1)
y = data['Salary'].values

# Split into train/test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Train the model on training data
model = LinearRegression()
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Get slope and intercept
m = model.coef_[0]
b = model.intercept_

print(f"Slope (Coefficient): {m:.2f}")
print(f"Intercept: {b:.2f}")
```

Slope (Coefficient): 9423.82  
 Intercept: 25321.58

```
In [55]: y_line = model.predict(X) # prediction for the full dataset

plt.scatter(data['YearsExperience'], data['Salary'], color='blue', label='Data P
plt.plot(data['YearsExperience'], y_line, color='red', label='Regression Line')
```

```
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.title('Salary vs Experience')
plt.legend()
plt.show()
```



In [56]: # Predict test values  
y\_pred = model.predict(X\_test)  
print(y\_pred)

```
[115790.21011287 71498.27809463 102596.86866063 75267.80422384  
55477.79204548 60189.69970699]
```

In [57]: # Calculate R^2 score  
r2\_score = model.score(X\_test, y\_test)  
print(f"R^2 Score: {r2\_score:.4f}")

```
R^2 Score: 0.9024
```

In [58]: import pickle  
import os  
  
# Ensure the folder exists (optional but recommended)  
os.makedirs("../models", exist\_ok=True)  
  
filename = '../models/experiment2-2.sav'  
  
with open(filename, 'wb') as f:  
 pickle.dump(model, f)

In [59]: loaded\_model = pickle.load(open(filename, 'rb'))  
loaded\_model

Out[59]:

LinearRegression		
Parameters		
fit_intercept	True	
copy_X	True	
tol	1e-06	
n_jobs	None	
positive	False	

```
In [60]: result=loaded_model.predict([[15]])
print(result)
```

```
[166678.81285724]
```

```
In [61]: prediction_input=int(input("Enter years of experience: "))
print(loaded_model.predict([[prediction_input]]))
```

```
[336307.4886718]
```

# Experiment 3

For a dataset given

- $X=[1,2,4,6,7]$
- $Y=[2,3,5,2,3]$

write the code for linear regression using gradient descent without using fit function and visualize the result

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

# initialize data
X = np.array([1,2,4,6,7])
Y = np.array([2,3,5,2,3])

# initialize parameters
m=0
c=0
alpha=0.01
epochs=1000
n=len(X)

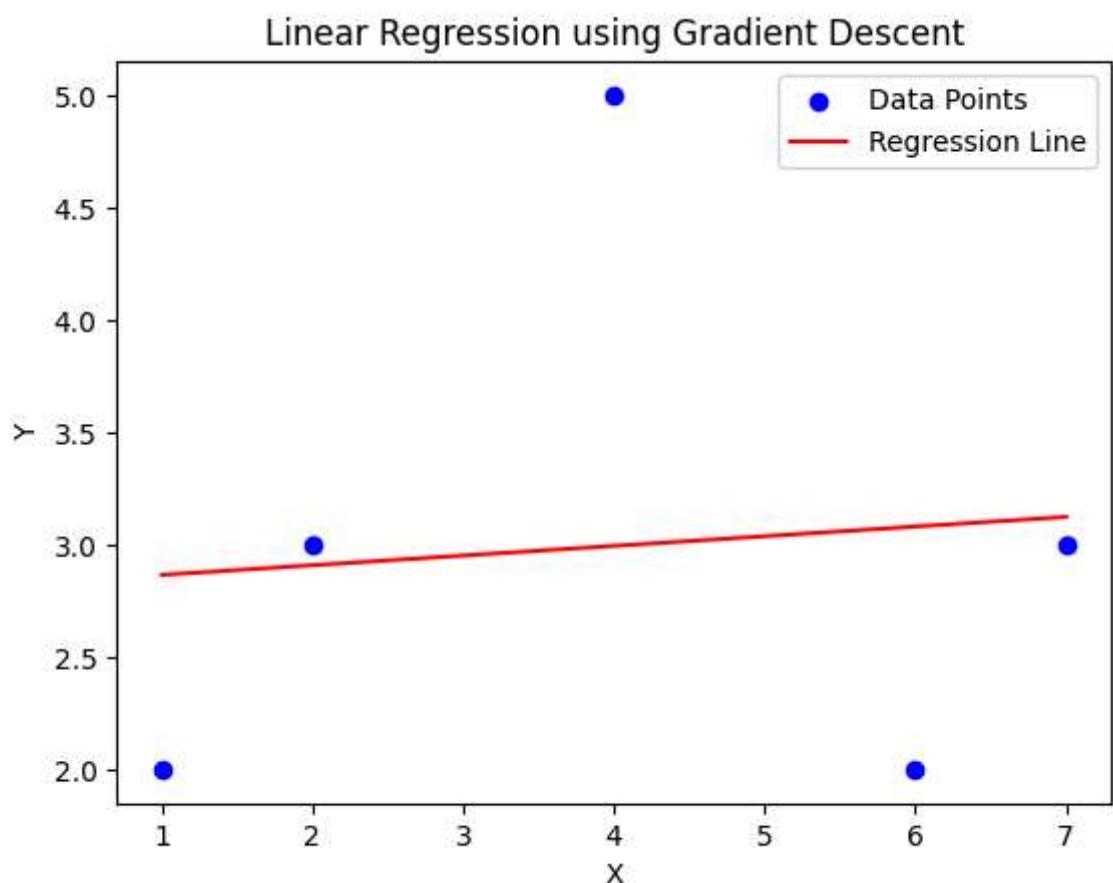
for _ in range(epochs):
    Y_pred = m*X + c
    D_m = (-2/n) * sum(X * (Y - Y_pred))
    D_c = (-2/n) * sum(Y - Y_pred)
    m = m - alpha * D_m
    c = c - alpha * D_c

print(f"Optimized parameters: m = {m:.4f}, c = {c:.4f}")
```

Optimized parameters:  $m = 0.0430$ ,  $c = 2.8224$

```
In [2]: # Predict Y values using the model
Y_pred = m*X + c

# Plotting the results
plt.scatter(X, Y, color='blue', label='Data Points')
plt.plot(X, Y_pred, color='red', label='Regression Line')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Linear Regression using Gradient Descent')
plt.legend()
plt.show()
```



# Objective:

To apply Multiple Linear Regression and Polynomial Regression on the California Housing dataset, handle missing values and categorical variables, and compare the performance of both models using appropriate evaluation metrics.

## Tasks:

### Task 1: Dataset Loading and Preprocessing

- Load the California Housing dataset using Pandas.
- Inspect the dataset to understand feature types and structure.
- Handle missing (NaN) values by replacing numerical missing values with column means.
- Convert categorical features into numerical form using One-Hot Encoding.

### Task 2: Feature and Target Preparation

- Identify the target variable (`median_house_value`).
- Separate input features and the target variable.
- Split the dataset into training and testing sets (80% training, 20% testing).

### Task 3: Multiple Linear Regression

- Train a Multiple Linear Regression model using the training data.
- Predict housing prices on the test dataset.
- Evaluate the model using the following metrics:
  - Mean Squared Error (MSE)
  - R<sup>2</sup> Score

### Task 4: Polynomial Regression

- Transform input features using Polynomial Features of degree 2.
- Train a Polynomial Regression model using the transformed features.
- Predict housing prices on the test dataset.
- Evaluate the model using the following metrics:
  - Mean Squared Error (MSE)
  - R<sup>2</sup> Score

### Task 5: Model Comparison

- Compare the performance of Multiple Linear Regression and Polynomial Regression models.
- Analyze the impact of polynomial feature expansion on model accuracy.

In [3]:

```
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error, r2_score
import kagglehub
import os
```

c:\Users\adity\OneDrive\Documents\GitHub\MLLab\.venv\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IPython not found. Please update jupyter and ipywidgets. See [https://ipywidgets.readthedocs.io/en/stable/user\\_install.html](https://ipywidgets.readthedocs.io/en/stable/user_install.html)

In [4]:

```
# Download dataset from Kaggle
dataset_path = kagglehub.dataset_download(
    "camnugent/california-housing-prices"
)

# Load CSV file
csv_path = os.path.join(dataset_path, "housing.csv")
df = pd.read_csv(csv_path)

print("First 5 rows of dataset:")
df.head()
```

Downloading to C:\Users\adity\.cache\kagglehub\datasets\camnugent\california-housing-prices\1.archive...

100%|██████████| 400k/400k [00:01<00:00, 378kB/s]

Extracting files...

First 5 rows of dataset:

Out[4]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	...
0	-122.23	37.88	41.0	880.0	129.0	322.0	
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	
2	-122.24	37.85	52.0	1467.0	190.0	496.0	
3	-122.25	37.85	52.0	1274.0	235.0	558.0	
4	-122.25	37.85	52.0	1627.0	280.0	565.0	



In [5]:

```
# Fill NaN values with column mean
df.fillna(df.mean(numeric_only=True), inplace=True)

print("Missing values after handling:")
df.isnull().sum()
```

Missing values after handling:

```
Out[5]: longitude      0
latitude        0
housing_median_age 0
total_rooms      0
total_bedrooms   0
population       0
households       0
median_income    0
median_house_value 0
ocean_proximity  0
dtype: int64
```

```
In [6]: # Convert categorical column to numerical
df_encoded = pd.get_dummies(df, drop_first=True)

print("Columns after encoding:")
df_encoded.columns
```

Columns after encoding:

```
Out[6]: Index(['longitude', 'latitude', 'housing_median_age', 'total_rooms',
   'total_bedrooms', 'population', 'households', 'median_income',
   'median_house_value', 'ocean_proximity_INLAND',
   'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',
   'ocean_proximity_NEAR OCEAN'],
  dtype='str')
```

```
In [7]: # Target variable
y = df_encoded['median_house_value']

# Feature variables
X = df_encoded.drop('median_house_value', axis=1)
```

```
In [8]: X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42
)

print("Training and testing data split completed.")
```

Training and testing data split completed.

```
In [9]: linear_model = LinearRegression()

# Train the model
linear_model.fit(X_train, y_train)

# Predict on test data
y_pred_linear = linear_model.predict(X_test)

# Evaluate performance
mse_linear = mean_squared_error(y_test, y_pred_linear)
r2_linear = r2_score(y_test, y_pred_linear)

print("--- Multiple Linear Regression Results ---")
print("Mean Squared Error (MSE):", mse_linear)
print("R2 Score:", r2_linear)
```

```
--- Multiple Linear Regression Results ---
Mean Squared Error (MSE): 4904399775.949299
R2 Score: 0.6257351821159687
```

```
In [10]: poly = PolynomialFeatures(degree=2, include_bias=False)

# Transform features
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

# Train polynomial regression model
poly_model = LinearRegression()
poly_model.fit(X_train_poly, y_train)

# Predict on test data
y_pred_poly = poly_model.predict(X_test_poly)

# Evaluate polynomial model
mse_poly = mean_squared_error(y_test, y_pred_poly)
r2_poly = r2_score(y_test, y_pred_poly)

print("--- Polynomial Regression (Degree 2) Results ---")
print("Mean Squared Error (MSE):", mse_poly)
print("R2 Score:", r2_poly)
```

```
--- Polynomial Regression (Degree 2) Results ---
Mean Squared Error (MSE): 4426103431.798885
R2 Score: 0.6622349582997737
```

```
In [11]: results = pd.DataFrame({
    "Model": [
        "Multiple Linear Regression",
        "Polynomial Regression (Degree 2)"
    ],
    "MSE": [
        mse_linear,
        mse_poly
    ],
    "R2 Score": [
        r2_linear,
        r2_poly
    ]
})

results
```

	Model	MSE	R <sup>2</sup> Score
<b>0</b>	Multiple Linear Regression	4.904400e+09	0.625735
<b>1</b>	Polynomial Regression (Degree 2)	4.426103e+09	0.662235

```
In [12]: print("Program executed successfully.")
```

Program executed successfully.

# Objective:

to generate data of degree 3 polynomial and fit models of degree 1, degree 3 and degree 13 polynomial and compare their performances.

## Tasks:

### Task 1: Data generation

- generate 500 samples of the target variable y using the equation ,  $y=0.5x^3-x^2+2$  and add noise
- Visualize the generated data points

### Task 2: Polynomial regression (500 samples)

Fit polynomial regression models of the following degree to the generated data:(use polynomial features library from Scikit-learn)

- Linear Regression model(Degree - 1)
- Polynomial Regression model (Degree - 3)
- Polynomial Regression model (Degree - 13)
- Compare the estimated model co-efficients to the actual ground truth equations
- Evaluate the model performance using metrics: MSE , MAE, R<sup>2</sup>

## Task 1 Code:

```
In [3]: import numpy as np
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)

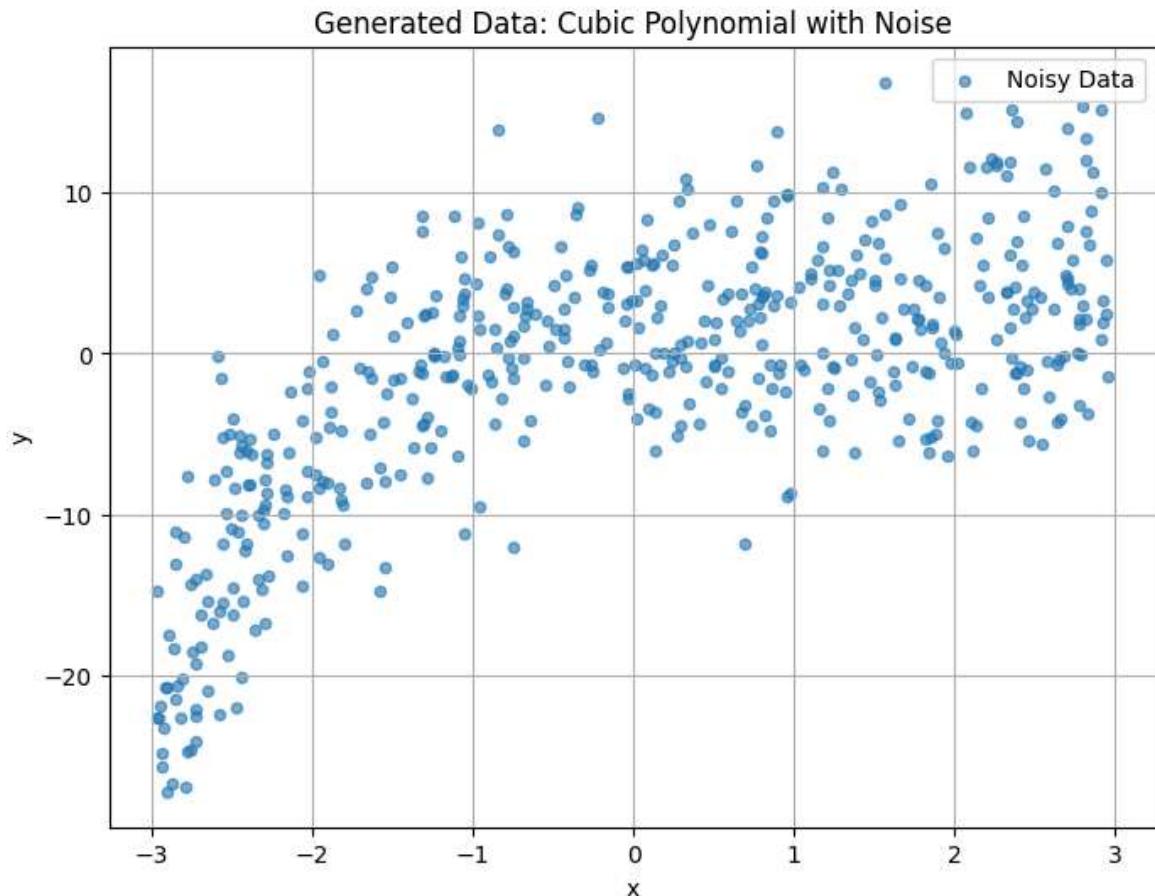
# Number of samples
n_samples = 500

# Generate input data
x = np.random.uniform(-3, 3, n_samples)

# True underlying function
y_true = 0.5 * x**3 - x**2 + 2

# Add Gaussian noise
noise = np.random.normal(0, 5, n_samples)
y = y_true + noise
```

```
# Visualize the generated data
plt.figure(figsize=(8, 6))
plt.scatter(x, y, s=20, alpha=0.6, label="Noisy Data")
plt.xlabel("x")
plt.ylabel("y")
plt.title("Generated Data: Cubic Polynomial with Noise")
plt.legend()
plt.grid(True)
plt.show()
```



## Task 2 Code:

In [4]:

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Reshape X for sklearn
X = x.reshape(-1, 1)
y_target = y

# Degrees to test
degrees = [1, 3, 13]

results = []

for degree in degrees:
    # Polynomial feature transformation
    poly = PolynomialFeatures(degree=degree, include_bias=True)
```

```
X_poly = poly.fit_transform(X)

# Linear regression on transformed features
model = LinearRegression()
model.fit(X_poly, y_target)

# Predictions
y_pred = model.predict(X_poly)

# Evaluation metrics
mse = mean_squared_error(y_target, y_pred)
mae = mean_absolute_error(y_target, y_pred)
r2 = r2_score(y_target, y_pred)

results.append({
    "Degree": degree,
    "MSE": mse,
    "MAE": mae,
    "R2": r2,
    "Coefficients": model.coef_,
    "Intercept": model.intercept_
})

# Display coefficients
print(f"\nPolynomial Degree {degree}")
print("Intercept:", model.intercept_)
for i, coef in enumerate(model.coef_):
    print(f"x^{i}: {coef:.4f}")

# Convert results to dataframe
results_df = pd.DataFrame(results)
print("\nModel Performance Summary")
print(results_df[["Degree", "MSE", "MAE", "R2"]])
```

Polynomial Degree 1  
Intercept: -1.2555889536901006  
 $x^0$ : 0.0000  
 $x^1$ : 3.0521

Polynomial Degree 3  
Intercept: 2.5318007492252086  
 $x^0$ : 0.0000  
 $x^1$ : 0.1944  
 $x^2$ : -1.1537  
 $x^3$ : 0.4978

Polynomial Degree 13  
Intercept: 2.034565317696061  
 $x^0$ : 0.0000  
 $x^1$ : -3.5659  
 $x^2$ : 0.8056  
 $x^3$ : 11.4449  
 $x^4$ : -1.5138  
 $x^5$ : -9.8695  
 $x^6$ : 0.3932  
 $x^7$ : 3.9528  
 $x^8$ : -0.0272  
 $x^9$ : -0.7787  
 $x^{10}$ : -0.0021  
 $x^{11}$ : 0.0738  
 $x^{12}$ : 0.0002  
 $x^{13}$ : -0.0027

#### Model Performance Summary

	Degree	MSE	MAE	R2
0	1	39.476836	5.097990	0.430638
1	3	25.005302	3.992331	0.639356
2	13	24.379374	3.912345	0.648384

In [1]:

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("yasserh/breast-cancer-dataset")

print("Path to dataset files:", path)
```

```
c:\Users\adity\OneDrive\Documents\GitHub\MLLab\.venv\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IPython not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
    from .autonotebook import tqdm as notebook_tqdm
Downloading to C:\Users\adity\.cache\kagglehub\datasets\yasserh\breast-cancer-dataset\1.archive...
100%|██████████| 48.6k/48.6k [00:00<00:00, 216kB/s]
Extracting files...
Path to dataset files: C:\Users\adity\.cache\kagglehub\datasets\yasserh\breast-cancer-dataset\versions\1
```

## Perform logistic regression on breast cancer data

- Find the Co-relation matrix
- Accuracy
- Precision
- Recall/sensitivity
- F1 Score
- False positive Rate
- True Positive rate
- ROC curve

In [2]:

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

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
    roc_curve,
    auc
)

# Load dataset
df = pd.read_csv(path + "/breast-cancer.csv")
```

In [3]:

```
df.columns
```

```
Out[3]: Index(['id', 'diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
   'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
   'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
   'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
   'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
   'fractal_dimension_se', 'radius_worst', 'texture_worst',
   'perimeter_worst', 'area_worst', 'smoothness_worst',
   'compactness_worst', 'concavity_worst', 'concave points_worst',
   'symmetry_worst', 'fractal_dimension_worst'],
  dtype='str')
```

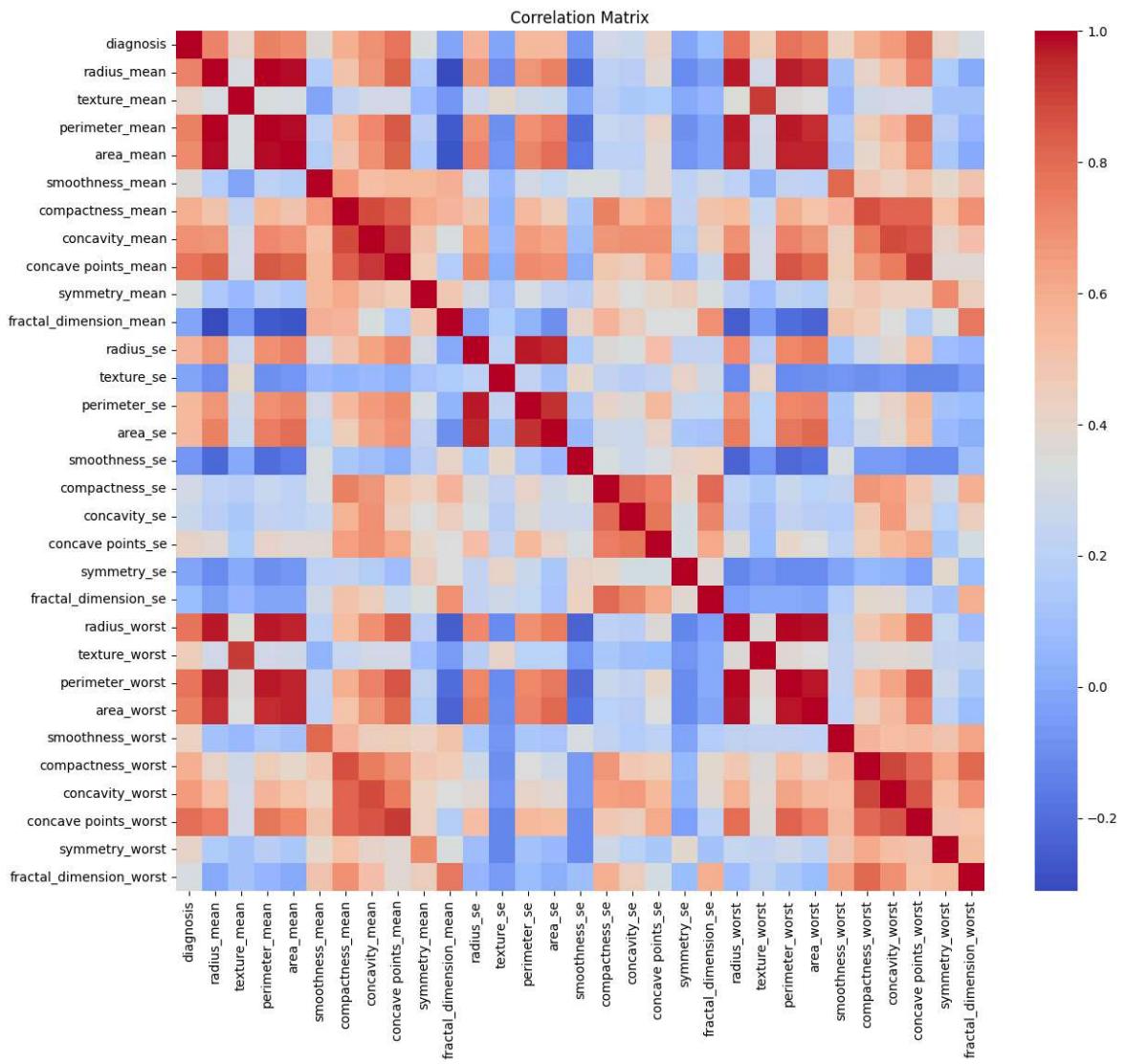
```
In [4]: # Drop ID column (not useful for ML)
df.drop("id", axis=1, inplace=True)

# Encode target variable
df["diagnosis"] = df["diagnosis"].map({"M": 1, "B": 0})

# Split features and target
X = df.drop("diagnosis", axis=1)
y = df["diagnosis"]
```

```
In [5]: # Correlation Matrix

plt.figure(figsize=(14, 12))
sns.heatmap(df.corr(), cmap="coolwarm")
plt.title("Correlation Matrix")
plt.show()
```



In [6]: # Train/Test Split

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [7]: # Logistic Regression

```
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]
```

In [8]: # Metrics

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)      # Sensitivity
```

```
f1 = f1_score(y_test, y_pred)

cm = confusion_matrix(y_test, y_pred)
tn, fp, fn, tp = cm.ravel()

false_positive_rate = fp / (fp + tn)
true_positive_rate = tp / (tp + fn)
```

```
In [9]: # ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
```

```
In [10]: # Output

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall or Sensitivity:", recall)
print("F1 Score:", f1)
print("False Positive Rate:", false_positive_rate)
print("True Positive Rate:", true_positive_rate)
print("AUC:", roc_auc)

# ROC Plot
plt.figure()
plt.plot(fpr, tpr, label=f"AUC = {roc_auc:.2f}")
plt.plot([0,1], [0,1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

Accuracy: 0.9736842105263158  
Precision: 0.9761904761904762  
Recall or Sensitivity: 0.9534883720930233  
F1 Score: 0.9647058823529412  
False Positive Rate: 0.014084507042253521  
True Positive Rate: 0.9534883720930233  
AUC: 0.99737962659679

