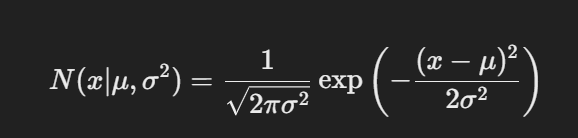
Experiment-1

Aim: Write a Python function to compute the value of the Gaussian distribution N(m,s)N(m, s)N(m,s) and plot varying mean and variance.

Theory: The Gaussian distribution (also known as the normal distribution) is a continuous probability distribution characterized by two parameters: mean μ and variance σ^2. The formula for the Gaussian distribution is:



The mean μ determines the central location of the curve, while the variance σ2 controls its spread. In this practical, we compute the value of the Gaussian distribution for different means and variances, plotting the results to visualize how these parameters affect the distribution. The distribution is symmetric around the mean, with most values concentrated around it, and the curve's width increases with higher variance.

Source Code:

import numpy as np

import matplotlib.pyplot as plt

# Gaussian distribution function

def gaussian\_distribution(x, mean, variance):

    return (1 / (np.sqrt(2 \* np.pi \* variance))) \* np.exp(-(x - mean)\*\*2 / (2 \* variance))

# Plotting function

def plot\_gaussian(mean\_values, variance\_values, x\_range):

    x = np.linspace(\*x\_range, 1000)

    for mean in mean\_values:

        for variance in variance\_values:

            y = gaussian\_distribution(x, mean, variance)

            plt.plot(x, y, label=f'Mean: {mean}, Variance: {variance}')

    plt.title('Gaussian Distribution with Varying Mean and Variance')

    plt.xlabel('X')

    plt.ylabel('Probability Density')

    plt.legend()

    plt.grid(True)

    plt.show()

# Parameters

mean\_values = [0, 1, -1]

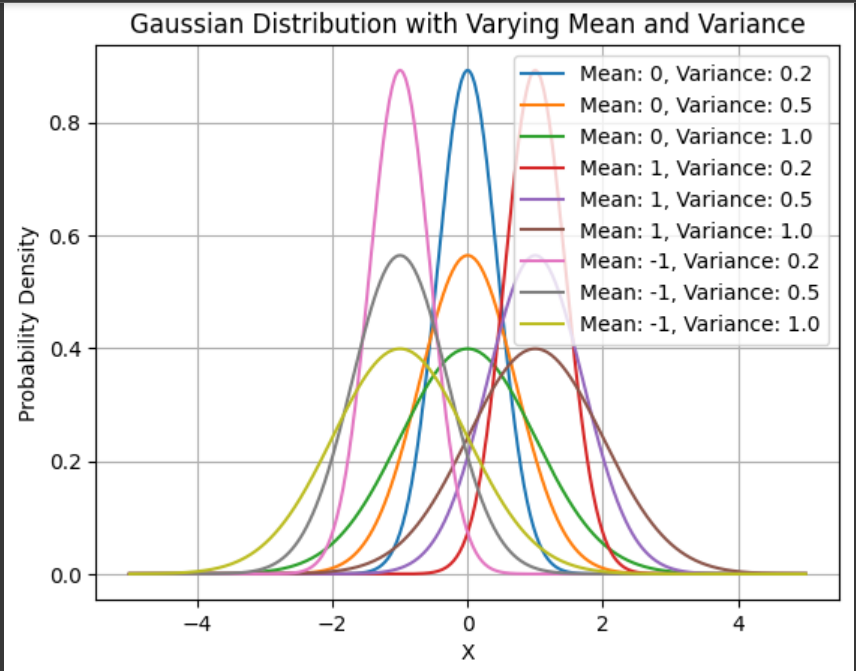
variance\_values = [0.2, 0.5, 1.0]

x\_range = (-5, 5)

# Plot Gaussian distributions

plot\_gaussian(mean\_values, variance\_values, x\_range)

Output:



**Viva Questions**

**Q1)** What is a Gaussian distribution?  
**A1)** A Gaussian distribution, also known as a normal distribution, is a bell-shaped probability distribution characterized by its mean μ\muμ and standard deviation σ\sigmaσ.

**Q2)** What is the effect of varying the mean on the Gaussian distribution?  
**A2)** Varying the mean shifts the entire distribution left or right without changing its shape. The mean represents the central point of the distribution.

**Q3)** How does changing the variance affect the Gaussian distribution?  
**A3)** Changing the variance affects the spread or width of the distribution. A larger variance leads to a wider and flatter distribution, while a smaller variance leads to a narrower and taller distribution.

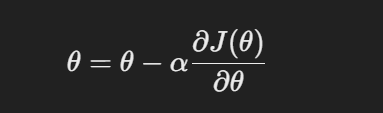
**Q4)** What is the probability density function (PDF) of a Gaussian distribution?  
**A4)** The PDF of a Gaussian distribution is given by f(x∣μ,σ)=1σ2πe−(x−μ)22σ2f(x | \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}f(x∣μ,σ)=σ2π​1​e−2σ2(x−μ)2​, where μ\muμ is the mean and σ\sigmaσ is the standard deviation.

**Q5)** What are some applications of the Gaussian distribution?  
**A5)** The Gaussian distribution is widely used in statistics, machine learning, signal processing, and financial modeling. It is often used to model random variables with a natural central tendency.

Experiment-2

Aim: Implement Gradient Descent Function.

Theory: Gradient descent is an optimization algorithm used to minimize the cost function by iteratively adjusting parameters in the direction of the steepest descent (negative gradient). It’s widely used in machine learning for finding the best parameters in regression and classification tasks. The algorithm starts with random parameter values, computes the gradient (partial derivatives) of the cost function, and updates the parameters using the formula:



Where α is the learning rate, and J(θ) is the cost function.

Source Code:

 import numpy as np

# Example cost function: J(theta) = (theta - 3)^2

def cost\_function(theta):

return (theta - 3) \*\* 2

# Derivative of the cost function: dJ/dtheta = 2 \* (theta - 3)

def gradient(theta):

return 2 \* (theta - 3)

# Gradient Descent Algorithm

def gradient\_descent(initial\_theta, learning\_rate, iterations):

theta = initial\_theta

history = []

for \_ in range(iterations):

cost = cost\_function(theta)

history.append((theta, cost))

theta = theta - learning\_rate \* gradient(theta)

return theta, history

# Parameters

initial\_theta = 0.0

learning\_rate = 0.1

iterations = 20

# Perform gradient descent

final\_theta, history = gradient\_descent(initial\_theta, learning\_rate, iterations)

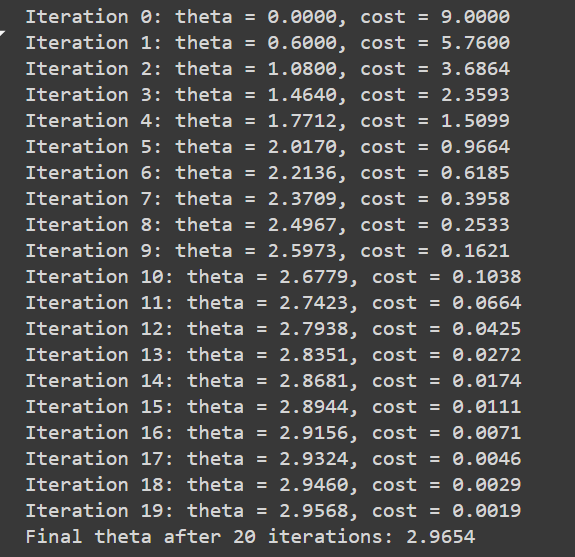
# Display results

for step, (theta, cost) in enumerate(history):

print(f"Iteration {step}: theta = {theta:.4f}, cost = {cost:.4f}")

print(f"Final theta after {iterations} iterations: {final\_theta:.4f}")

Output:



**Viva Questions**

**Q1)** What is gradient descent?  
**A1)** Gradient descent is an optimization algorithm used to minimize the cost function by iteratively adjusting parameters in the opposite direction of the gradient of the cost function.

**Q2)** What is the learning rate in gradient descent, and how does it affect the convergence?  
**A2)** The learning rate is a hyperparameter that determines the step size during parameter updates. A small learning rate results in slow convergence, while a large learning rate may cause divergence.

**Q3)** What is the difference between batch and stochastic gradient descent?  
**A3)** In batch gradient descent, the gradient is computed using the entire dataset, while in stochastic gradient descent (SGD), it is computed for each individual data point, leading to faster but noisier updates.

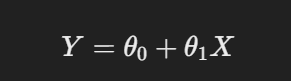
**Q4)** How do you know when to stop the gradient descent algorithm?  
**A4)** Gradient descent is stopped when the cost function reaches a minimum, indicated by a small change in its value between iterations, or after a fixed number of iterations.

**Q5)** Can gradient descent be used for non-convex functions?  
**A5)** Yes, gradient descent can be used for non-convex functions, but it may get stuck in local minima or saddle points depending on the shape of the cost function.

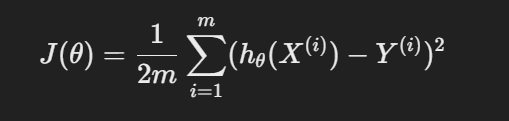
Experiment-3

Aim: Implement Linear Regression using Gradient Descent

Theory: Linear regression is a supervised learning algorithm used to predict a continuous target variable based on one or more input features. The relationship between the input (X) and output (Y) is modelled by a straight line:



Here, θ0​ is the intercept and θ1​ is the slope. Gradient descent is used to minimize the cost function, typically the Mean Squared Error (MSE), which measures how far the predicted values are from the actual values. The cost function for linear regression is:



where hθ(X) is the predicted value, and m is the number of training examples.

The gradient descent updates the parameters iteratively until it finds the optimal values that minimize the cost function.

Source Code:

import numpy as np

import matplotlib.pyplot as plt

# Hypothesis (Prediction function)

def predict(X, theta):

return np.dot(X, theta)

# Cost function (Mean Squared Error)

def compute\_cost(X, Y, theta):

m = len(Y)

return (1/(2\*m)) \* np.sum((predict(X, theta) - Y)\*\*2)

# Gradient Descent Algorithm

def gradient\_descent(X, Y, theta, learning\_rate, iterations):

m = len(Y)

cost\_history = []

for \_ in range(iterations):

theta = theta - (learning\_rate/m) \* np.dot(X.T, predict(X, theta) - Y)

cost\_history.append(compute\_cost(X, Y, theta))

return theta, cost\_history

# Data (example)

X = np.array([1, 2, 3, 4, 5])

Y = np.array([5, 7, 9, 11, 13])

# Add a bias (intercept) term to X (X\_0 = 1)

X\_b = np.c\_[np.ones((len(X), 1)), X] # Add a column of 1s for theta\_0

# Initialize theta (parameters)

theta = np.random.randn(2)

# Parameters

learning\_rate = 0.01

iterations = 1000

# Run gradient descent

final\_theta, cost\_history = gradient\_descent(X\_b, Y, theta, learning\_rate, iterations)

# Display final parameters

print(f"Final theta: {final\_theta}")

# Plot the cost function history

plt.plot(cost\_history)

plt.title('Cost Function over Iterations')

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.show()

# Plot the fitted line

plt.scatter(X, Y, color='blue', label='Training data')

plt.plot(X, predict(X\_b, final\_theta), color='red', label='Linear regression fit')

plt.title('Linear Regression Fit')

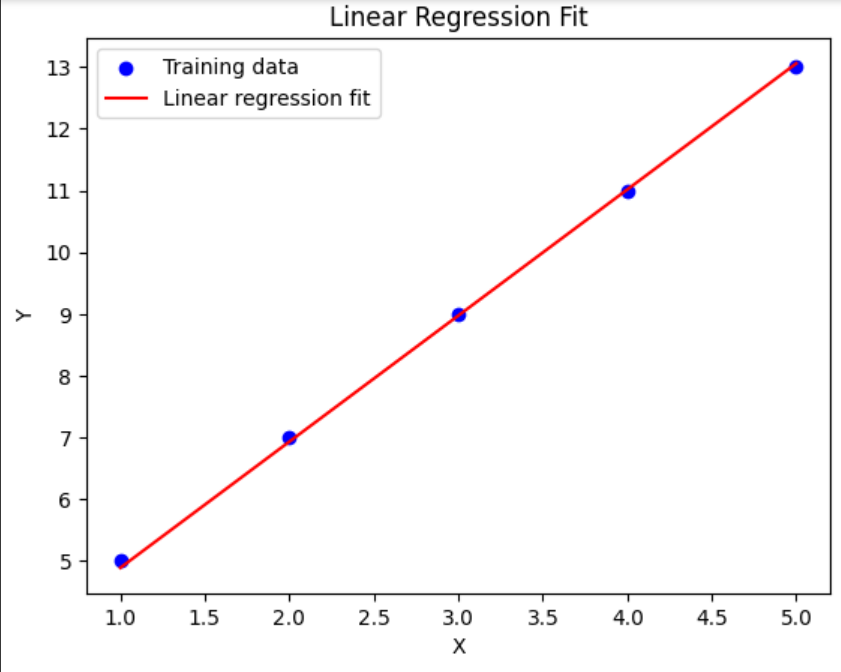
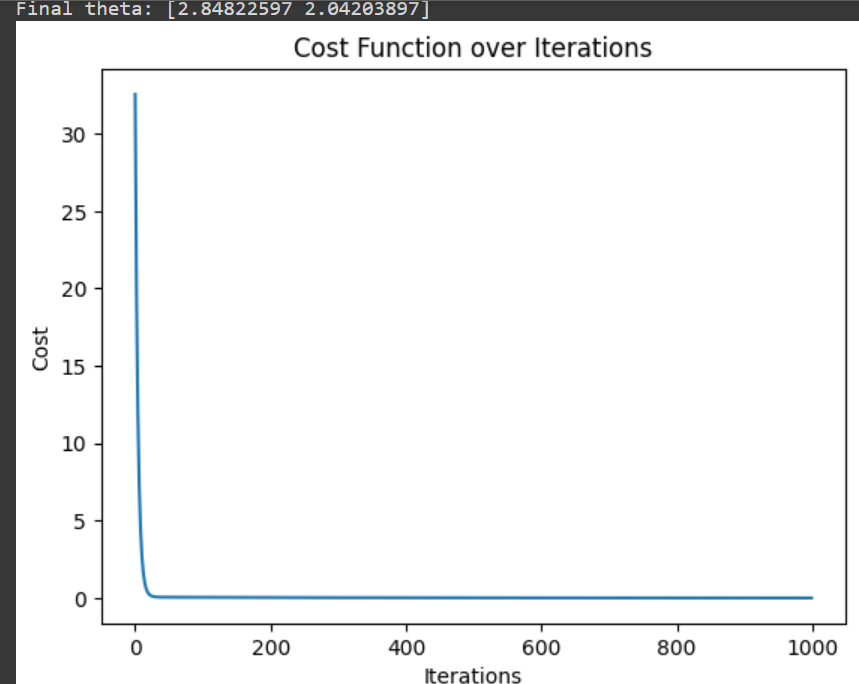
plt.xlabel('X')

plt.ylabel('Y')

plt.legend()

plt.show()

Output:



**Viva Questions**

**Q1)** What is the goal of linear regression?  
**A1)** The goal of linear regression is to model the relationship between a dependent variable and one or more independent variables by fitting a straight line to the data.

**Q2)** How is gradient descent used in linear regression?  
**A2)** Gradient descent is used to minimize the cost function (mean squared error) by adjusting the parameters (slope and intercept) in each iteration to find the line of best fit.

**Q3)** What is the cost function in linear regression?  
**A3)** The cost function in linear regression is typically the mean squared error (MSE), which measures the average squared difference between the predicted and actual values.

**Q4)** What are the parameters in linear regression that are updated during gradient descent?  
**A4)** The parameters updated during gradient descent are the slope (coefficient) and intercept (bias) of the regression line.

**Q5)** What is the difference between gradient descent and the normal equation for linear regression?  
**A5)** Gradient descent is an iterative optimization method, while the normal equation is an analytical method that solves for the parameters directly without iteration.

Experiment-4

Aim: Comparison of Classification Accuracy: SVM vs CNN

Theory: Support Vector Machines (SVMs) and Convolutional Neural Networks (CNNs) are both popular algorithms for classification tasks. SVM is a discriminative classifier that finds a hyperplane separating classes in high-dimensional space. It works well on smaller datasets and structured data. On the other hand, CNNs are deep learning models that are particularly suited for image classification due to their ability to capture spatial hierarchies through convolutional layers.

For this experiment, we'll use a common dataset (e.g., MNIST) and compare the accuracy of SVM and CNN classifiers.

Source Code:

* SVM

 from sklearn import datasets

from sklearn.model\_selection import train\_test\_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

# Load dataset (e.g., digits dataset)

digits = datasets.load\_digits()

X = digits.images.reshape((len(digits.images), -1)) # Flatten the images

Y = digits.target

# Split dataset into training and testing sets

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, Y, test\_size=0.3, random\_state=42)

# Train SVM model

svm\_model = SVC(kernel='linear')

svm\_model.fit(X\_train, Y\_train)

# Test SVM model

Y\_pred\_svm = svm\_model.predict(X\_test)

# SVM Accuracy

svm\_accuracy = accuracy\_score(Y\_test, Y\_pred\_svm)

print(f"SVM Accuracy: {svm\_accuracy \* 100:.2f}%")

* CNN

 import tensorflow as tf

from tensorflow.keras import layers, models

from sklearn.model\_selection import train\_test\_split

# Load dataset (e.g., MNIST dataset)

(X\_train, Y\_train), (X\_test, Y\_test) = tf.keras.datasets.mnist.load\_data()

# Normalize the data

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

# Build CNN model

cnn\_model = models.Sequential([

layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Flatten(),

layers.Dense(64, activation='relu'),

layers.Dense(10, activation='softmax')

])

cnn\_model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Reshape the data for the CNN

X\_train = X\_train.reshape(-1, 28, 28, 1)

X\_test = X\_test.reshape(-1, 28, 28, 1)

# Train CNN model

cnn\_model.fit(X\_train, Y\_train, epochs=5)

# Test CNN model

cnn\_accuracy = cnn\_model.evaluate(X\_test, Y\_test, verbose=0)

print(f"CNN Accuracy: {cnn\_accuracy[1] \* 100:.2f}%")

Output:





**Viva Questions**

**Q1)** What is SVM (Support Vector Machine)?  
**A1)** SVM is a supervised learning algorithm used for classification and regression. It finds the hyperplane that maximally separates data points of different classes.

**Q2)** What is CNN (Convolutional Neural Network)?  
**A2)** CNN is a type of deep neural network commonly used for image classification tasks. It uses convolutional layers to extract features from images.

**Q3)** How does SVM differ from CNN in terms of classification?  
**A3)** SVM works well for linear and some non-linear problems, while CNN excels in image classification and tasks requiring hierarchical feature extraction from data.

**Q4)** What metrics can be used to compare the classification accuracy of SVM and CNN?  
**A4)** Metrics such as accuracy, precision, recall, F1-score, and confusion matrix can be used to compare the performance of SVM and CNN on a given dataset.

**Q5)** Which method typically performs better on image data: SVM or CNN?  
**A5)** CNN usually performs better on image data due to its ability to automatically learn spatial hierarchies of features from images.

Experiment-5

Aim: Implement Basic Image Handling and Processing operations on the Image

Theory: Image handling and processing involve tasks such as loading, displaying, and manipulating images, which are crucial in computer vision. These tasks can be performed using libraries like OpenCV or PIL (Pillow). Common operations include resizing, rotating, flipping, converting between color spaces (e.g., RGB to grayscale), and drawing shapes on images. These basic techniques form the foundation for more advanced image processing tasks, such as filtering, edge detection, and segmentation, which are often used in object detection and recognition systems.

Source Code:

 import cv2

from matplotlib import pyplot as plt

# Load an image

image = cv2.imread(' /content/a.jpg’)

# Convert the image from BGR to RGB (since OpenCV loads in BGR)

image\_rgb = cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)

# Display the original image

plt.imshow(image\_rgb)

plt.title('Original Image')

plt.axis('off') # Turn off axis numbers

plt.show()

# Convert to grayscale

gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

plt.imshow(gray\_image, cmap='gray')

plt.title('Grayscale Image')

plt.axis('off')

plt.show()

# Resize the image

resized\_image = cv2.resize(image\_rgb, (200, 200))

plt.imshow(resized\_image)

plt.title('Resized Image (200x200)')

plt.axis('off')

plt.show()

# Rotate the image

(h, w) = image.shape[:2]

center = (w // 2, h // 2)

matrix = cv2.getRotationMatrix2D(center, 45, 1.0) # Rotate by 45 degrees

rotated\_image = cv2.warpAffine(image\_rgb, matrix, (w, h))

plt.imshow(rotated\_image)

plt.title('Rotated Image (45 degrees)')

plt.axis('off')

plt.show()

# Flip the image (horizontal flip)

flipped\_image = cv2.flip(image\_rgb, 1)

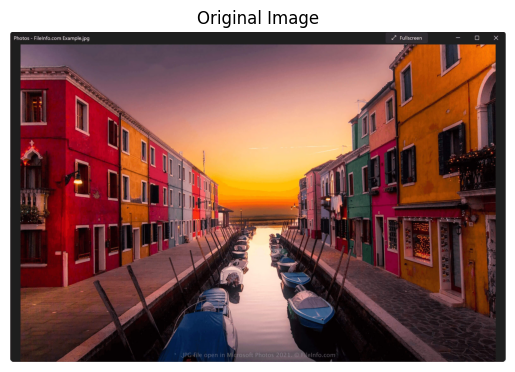
plt.imshow(flipped\_image)

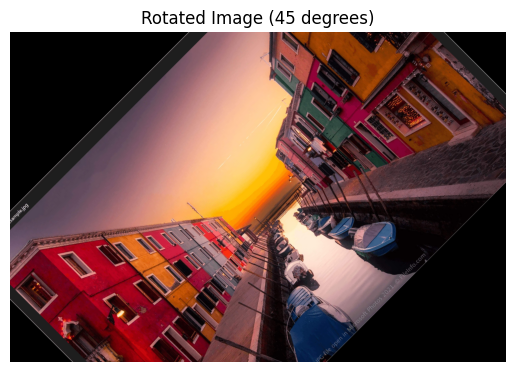
plt.title('Horizontally Flipped Image')

plt.axis('off')

plt.show()

Output:







**Viva Questions**

**Q1)** What are the basic image processing operations?  
**A1)** Basic image processing operations include reading, displaying, resizing, cropping, filtering, and transforming images.

**Q2)** What is the purpose of image filtering in image processing?  
**A2)** Image filtering is used to enhance an image, reduce noise, and highlight certain features by applying various filters like Gaussian, median, and edge detection filters.

**Q3)** How is image transformation performed?  
**A3)** Image transformation is performed using techniques like rotation, translation, scaling, and warping to manipulate the appearance of the image.

**Q4)** What is the difference between a grayscale image and an RGB image?  
**A4)** A grayscale image contains intensity values ranging from black to white, whereas an RGB image contains three color channels: Red, Green, and Blue, each with intensity values.

**Q5)** How do you handle image data in Python?  
**A5)** Image data in Python can be handled using libraries like OpenCV, PIL, and scikit-image for tasks such as reading, modifying, and saving images.

Experiment-6

Aim: Implementation of Geometric Transformation

Theory: Geometric transformations involve altering the spatial arrangement of pixels in an image through operations like scaling, translation, rotation, and affine transformations. These transformations are essential in image manipulation tasks such as resizing or aligning images. Each transformation can be represented by a matrix, which when applied to the image’s coordinates, alters its structure.

* **Scaling** changes the size of the image.
* **Translation** shifts the image along the x or y axis.
* **Rotation** rotates the image by a specified angle.
* **Affine transformations** preserve parallelism but not distances and angles, useful for shearing or skewing images.

Source Code:

 # Scaling the image

scaled\_image = cv2.resize(image\_rgb, None, fx=1.5, fy=1.5, interpolation=cv2.INTER\_LINEAR)

plt.imshow(scaled\_image)

plt.title('Scaled Image (150%)')

plt.axis('off')

plt.show()

# Translating the image (shift by 50 pixels right and 30 pixels down)

translation\_matrix = np.float32([[1, 0, 50], [0, 1, 30]])

translated\_image = cv2.warpAffine(image\_rgb, translation\_matrix, (w, h))

plt.imshow(translated\_image)

plt.title('Translated Image (Right by 50, Down by 30)')

plt.axis('off')

plt.show()

# Rotating the image (already covered earlier)

# Affine Transformation (Shearing effect)

points\_before = np.float32([[50, 50], [200, 50], [50, 200]])

points\_after = np.float32([[10, 100], [200, 50], [100, 250]])

affine\_matrix = cv2.getAffineTransform(points\_before, points\_after)

affine\_image = cv2.warpAffine(image\_rgb, affine\_matrix, (w, h))

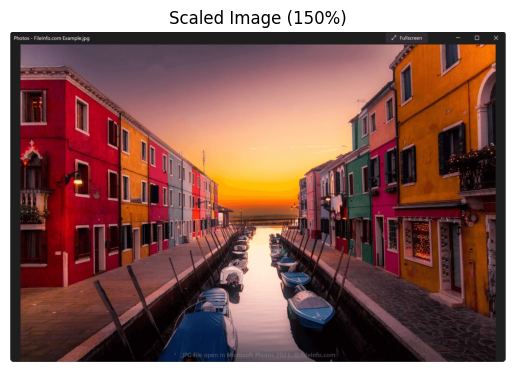
plt.imshow(affine\_image)

plt.title('Affine Transformed Image (Shearing)')

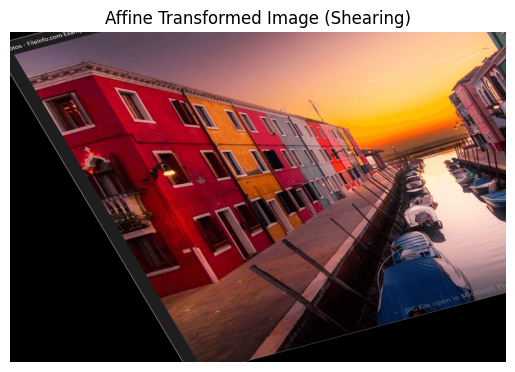
plt.axis('off')

plt.show()

Output:







**Viva Questions**

**Q1)** What is a geometric transformation in image processing?  
**A1)** Geometric transformation refers to the process of altering the geometric properties of an image, such as rotating, scaling, translating, or skewing the image.

**Q2)** What are the common types of geometric transformations?  
**A2)** Common types include translation, rotation, scaling, and shearing, which modify the position, size, and shape of an image.

**Q3)** What is affine transformation?  
**A3)** Affine transformation is a geometric transformation that preserves lines and parallelism, but not necessarily angles and distances. It includes translation, rotation, scaling, and shearing.

**Q4)** How does interpolation affect geometric transformations?  
**A4)** Interpolation is used to estimate pixel values when an image is transformed. Methods like nearest-neighbor, bilinear, and bicubic interpolation affect the quality of the transformed image.

**Q5)** Can geometric transformations be reversed?  
**A5)** Yes, geometric transformations can be reversed if the transformation matrix is invertible. Reversing involves applying the inverse of the transformation matrix.

Experiment-7

Aim: Implementation of Perspective Transformation

Theory: Perspective transformation is a geometric operation that maps a 2D image from one plane to another, simulating the effect of viewing an image from a different angle. It is commonly used in tasks like image rectification (e.g., correcting the perspective of a document image taken at an angle) and creating panoramic views. The transformation is defined by four pairs of points: one set from the source image and one set from the destination image. The operation finds a transformation matrix, which is then applied to warp the source image to the new perspective.

Source Code:

 # Define four points in the original image

pts\_before = np.float32([[50, 50], [200, 50], [50, 200], [200, 200]])

# Define where these points should map in the output image (change of perspective)

pts\_after = np.float32([[10, 100], [180, 50], [100, 250], [220, 220]])

# Get the perspective transformation matrix

perspective\_matrix = cv2.getPerspectiveTransform(pts\_before, pts\_after)

# Apply the perspective transformation

warped\_image = cv2.warpPerspective(image\_rgb, perspective\_matrix, (w, h))

# Display the warped image

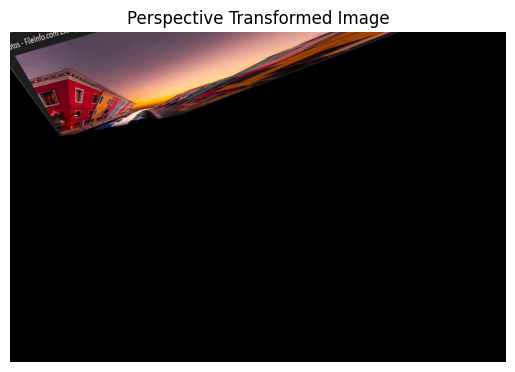
plt.imshow(warped\_image)

plt.title('Perspective Transformed Image')

plt.axis('off')

plt.show()

Output:



**Viva Questions**

**Q1)** What is perspective transformation in image processing?  
**A1)** Perspective transformation changes the viewpoint of an image, giving it a 3D-like appearance by mapping the points of an image to a new perspective.

**Q2)** What are the applications of perspective transformation?  
**A2)** Perspective transformation is used in applications such as projective geometry, camera calibration, and image stitching.

**Q3)** How is the transformation matrix for perspective transformation calculated?  
**A3)** The transformation matrix for perspective transformation is a 3x3 matrix that defines how points in the source image are mapped to the destination image.

**Q4)** How do you select points for perspective transformation?  
**A4)** Typically, four points from the original image and their corresponding points in the transformed image are selected to compute the perspective transformation matrix.

**Q5)** What is the difference between affine and perspective transformations?  
**A5)** Affine transformations preserve parallelism, while perspective transformations do not. Perspective transformations map straight lines to curves and can create 3D effects.

Experiment-8

Aim: Implementation of Camera Calibration

Theory: Camera calibration is the process of estimating the internal and external parameters of a camera to correct lens distortion and determine the relationship between the 3D real-world coordinates and the 2D coordinates of the image. This is important for applications such as 3D reconstruction, robot navigation, and augmented reality. Calibration typically involves capturing images of a known reference object (such as a chessboard) and finding correspondences between the object’s 3D points and the image’s 2D points.

**Steps in camera calibration**:

1. Capture multiple images of a calibration pattern (like a chessboard).
2. Detect the pattern in the images and extract corner points.
3. Use these correspondences to compute the camera matrix and distortion coefficients.

Source Code:

 import cv2

import numpy as np

# Load the image of the chessboard (for example)

image = cv2.imread(/content.chess.jpg)

# Convert to grayscale

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# Define the size of the chessboard pattern (number of internal corners)

pattern\_size = (9, 6) # Example for 9x6 chessboard

# Find the chessboard corners

ret, corners = cv2.findChessboardCorners(gray, pattern\_size, None)

# If corners are found, proceed with calibration

if ret:

# Draw the corners on the image

image\_with\_corners = cv2.drawChessboardCorners(image, pattern\_size, corners, ret)

# Display the image with detected corners

plt.imshow(cv2.cvtColor(image\_with\_corners, cv2.COLOR\_BGR2RGB))

plt.title('Chessboard Corners Detected')

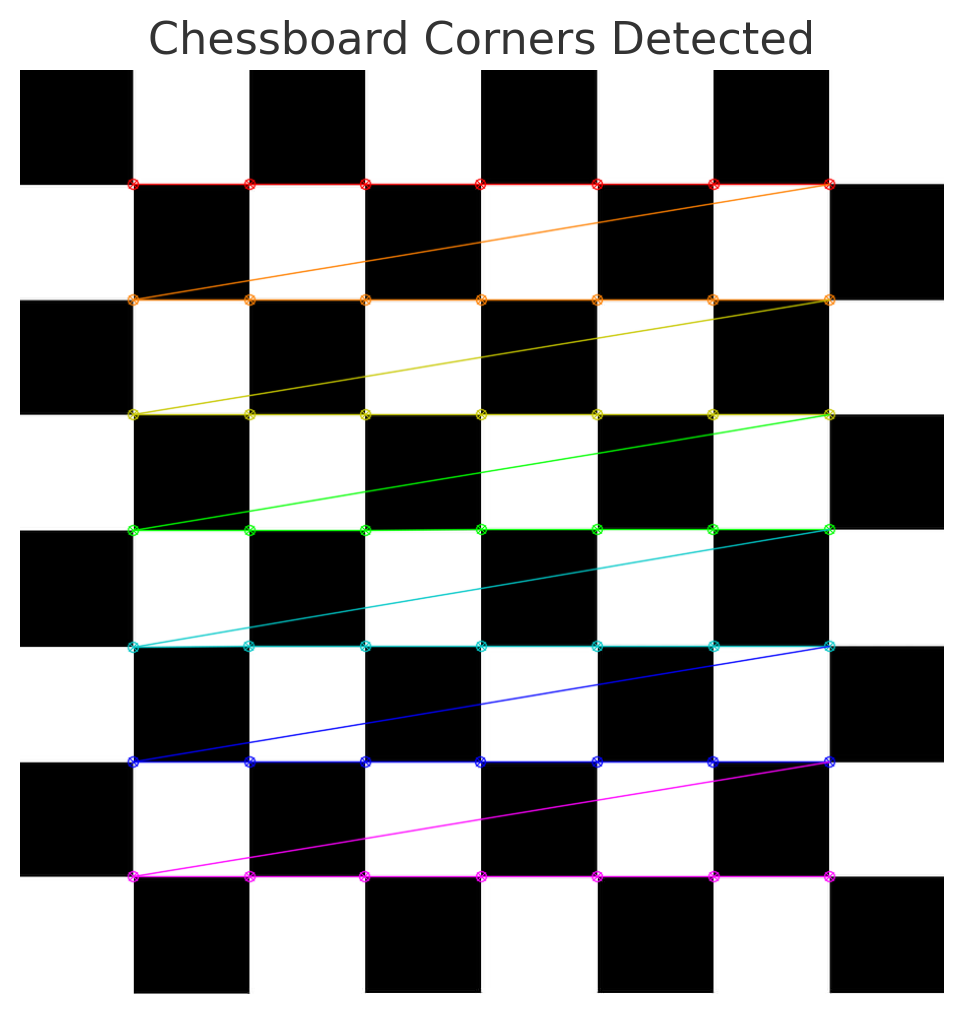
plt.axis('off')

plt.show()

# In a full calibration process, multiple images would be used to compute

# the camera matrix and distortion coefficients.

Output:



**Viva Questions**

**Q1)** What is camera calibration?  
**A1)** Camera calibration is the process of estimating the camera parameters to correct distortions and accurately reconstruct 3D scenes from 2D images.

**Q2)** Why is camera calibration important?  
**A2)** Calibration is crucial for applications like 3D reconstruction, augmented reality, and computer vision, as it ensures that the measurements and positions derived from images are accurate.

**Q3)** What techniques are commonly used for camera calibration?  
**A3)** Techniques include using checkerboard patterns, point correspondences, and methods like Zhang's calibration method, which utilizes multiple images from different views.

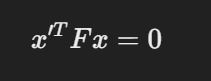
**Q4)** What are intrinsic and extrinsic camera parameters?  
**A4)** Intrinsic parameters define the internal characteristics of the camera (focal length, optical center), while extrinsic parameters describe the camera's position and orientation in the world coordinate system.

**Q5)** How can the accuracy of camera calibration be evaluated?  
**A5)** Accuracy can be evaluated by measuring the reprojection error, which quantifies the difference between the observed image points and the projected points obtained from the calibrated model.

Experiment-9

Aim: Compute Fundamental Matrix

Theory: The fundamental matrix FFF relates corresponding points between two stereo images. It encapsulates the epipolar geometry, ensuring that if a point is visible in one image, its corresponding point in the other image lies along a specific line called the epipolar line. The fundamental matrix is critical in applications like 3D reconstruction, depth estimation, and stereo vision. It is a 3x3 matrix derived from corresponding points in both images and is used to calculate the epipolar constraint:



where x and x′ are corresponding points in the two images.

Source Code:

 import cv2

import numpy as np

pts\_image1 = np.array([[100, 150], [120, 200], [130, 220], [150, 180],

                        [160, 140], [170, 190], [180, 210], [190, 160]], dtype=np.float32)

pts\_image2 = np.array([[90, 140], [115, 190], [125, 215], [145, 175],

                        [155, 135], [165, 185], [175, 205], [185, 155]], dtype=np.float32)

# Compute the fundamental matrix

F, mask = cv2.findFundamentalMat(pts\_image1, pts\_image2, cv2.FM\_LMEDS)

if F is not None:

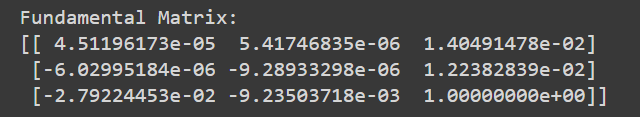
    print("Fundamental Matrix:")

    print(F)

else:

    print("Fundamental Matrix computation failed.")

Output:



**Viva Questions**

**Q1)** What is the fundamental matrix in computer vision?  
**A1)** The fundamental matrix is a 3x3 matrix that encapsulates the intrinsic projective geometry between two views of a scene, relating corresponding points in stereo images.

**Q2)** How is the fundamental matrix computed?  
**A2)** The fundamental matrix can be computed using point correspondences between two images, often through techniques like the eight-point algorithm or the RANSAC algorithm to handle outliers.

**Q3)** What role does the fundamental matrix play in stereo vision?  
**A3)** It enables the estimation of epipolar lines, which constrain the search for corresponding points in stereo images, facilitating depth perception and 3D reconstruction.

**Q4)** What are some challenges in computing the fundamental matrix?  
**A4)** Challenges include handling noise in the point correspondences, dealing with outliers, and ensuring that the computed matrix adheres to the constraints of epipolar geometry.

**Q5)** How can the quality of the fundamental matrix be evaluated?  
**A5)** The quality can be evaluated by analyzing the epipolar constraints' satisfaction and measuring the reprojection error for the corresponding points.