Pattern Recognition and Computer Vision Lab

Faculty Name: Mr. Anupam Kumar

Student Name: Prathya Thakur

Roll No.: 03614802721

Semester: 7th

Group: AIML 1 B

Maharaja Agrasen Institute of Technology, PSP Area, Sector - 22, New Delhi – 110085

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

COMPUTER SCIENCE & ENGINEERING DEPARTMENT

VISION

"To be centre of excellence in education, research and technology transfer in the field of computer engineering and promote entrepreneurship and ethical values."

MISSION

"To foster an open, multidisciplinary and highly collaborative research environment to produce world-class engineers capable of providing innovative solutions to real life problems and fulfill societal needs."

PRACTICAL RECORD

Name of the student : Prathya Thakur

University Roll No. : 03614802721

Branch : CSE-1

Section/ Group : 7 AIML 1 B

Exp. No.	Experiment Name	Date of performance	Date of checking	R1 (3)	R2 (3)	R3 (3)	R4 (3)	R5 (3)	Total Marks (15)	Signature
1.										
2.										
3.										
4.										
5.										
6.										
7.										
8.										
9.										

EXPERIMENT 1

AIM: To write a Python function to compute the value of the Gaussian distribution at a given vector x and plot the effect of varying the mean and variance on the normal distribution.

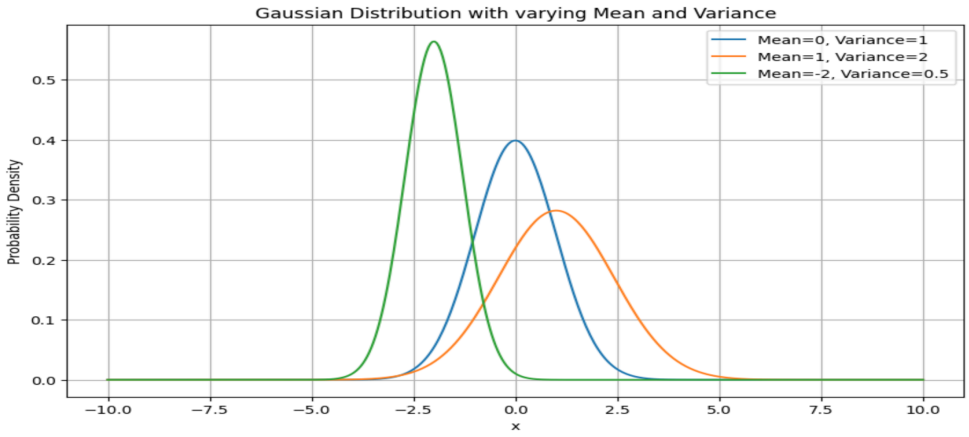
THEORY:

The Gaussian or normal distribution is widely used in statistics and probability theory. It is defined by two parameters: the mean (μ) and the variance (σ^2). The Gaussian function is bell-shaped, and it expresses the probability distribution of continuous variables. The function peaks at the mean and the spread is controlled by the variance. Applications include modeling natural phenomena and noise in images.

SOURCE CODE:

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

def gaussian_distribution(x, mean, variance):
    return (1/np.sqrt(2*np.pi*variance)) * np.exp(-0.5 * ((x - mean)**2 / variance))

# Generate x values
x_values = np.linspace(-10, 10, 400)
means = [0, 1, -2]
variances = [1, 2, 0.5]

# Plot Gaussian distributions with varying means and variances
plt.figure(figsize=(10, 6))
for mean, variance in zip(means, variances):
    y_values = gaussian_distribution(x_values, mean, variance)
    plt.plot(x_values, y_values, 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()
```

OUTPUT:



EXPERIMENT 2

AIM: To implement the gradient descent algorithm in Python and observe how it optimizes the parameters to fit a line through given data points.

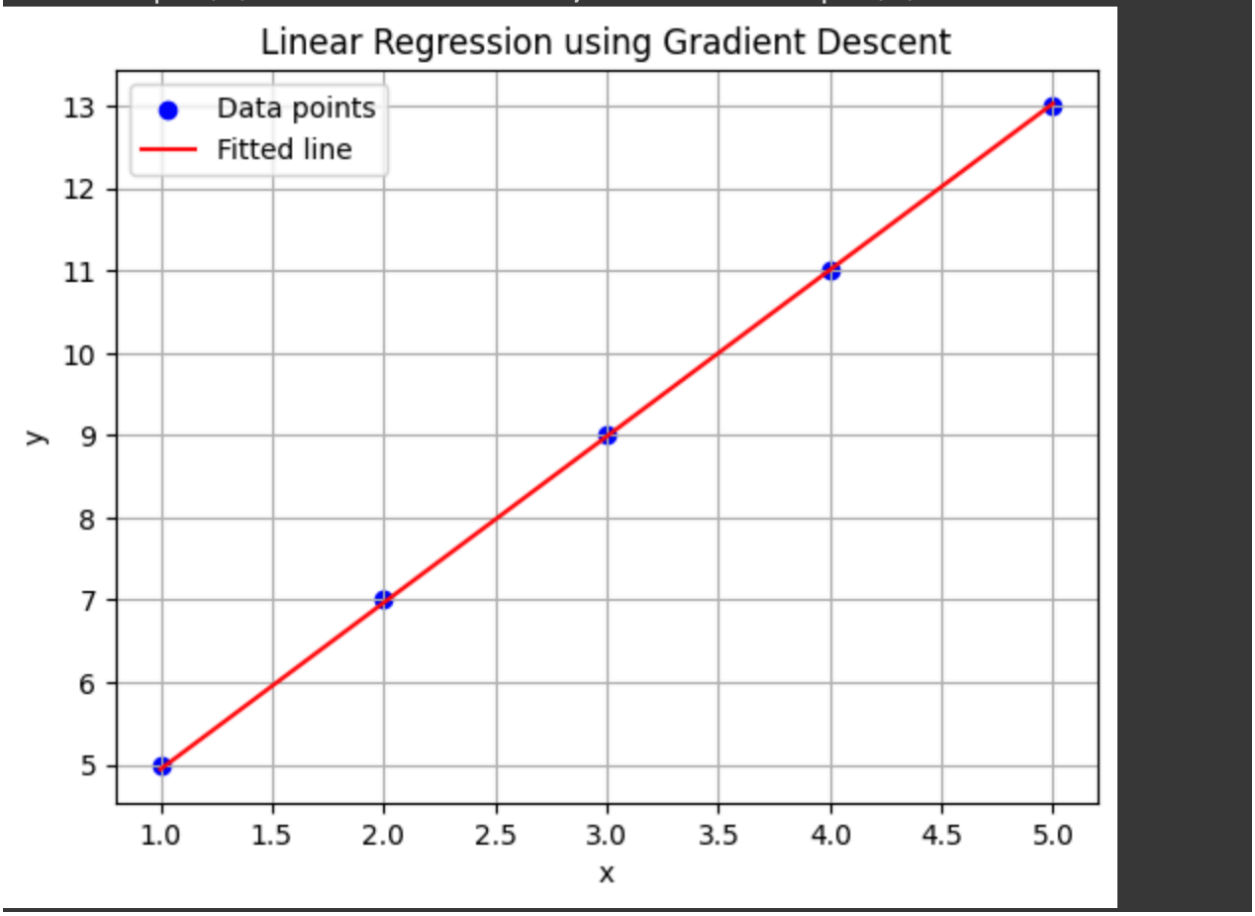
THEORY:
Gradient descent is an optimization algorithm used to minimize a function by iteratively moving toward the steepest descent as defined by the negative gradient. It is widely used for optimizing parameters in machine learning algorithms such as linear regression and neural networks.

SOURCE CODE:

```
import numpy as np

# Gradient descent function
def gradient_descent(x, y, learning_rate=0.01, iterations=1000):
    m, c = 0, 0  # Initial values of slope and intercept
    n = float(len(x))  # Number of data points

    for i in range(iterations):
        y_pred = m * x + c
        # Calculate gradients
        D_m = (-2/n) * sum(x * (y - y_pred))  # Gradient of m
        D_c = (-2/n) * sum(y - y_pred)        # Gradient of c
        # Update the slope and intercept
        m = m - learning_rate * D_m
        c = c - learning_rate * D_c
    return m, c

# Example data points
x = np.array([1, 2, 3, 4, 5])
y = np.array([5, 7, 9, 11, 13])

# Perform gradient descent
m, c = gradient_descent(x, y)

# Output the final slope and intercept
print(f"Final Slope (m): {m}, Final Intercept (c): {c}")

# Plot the result
import matplotlib.pyplot as plt
plt.scatter(x, y, color='blue', label='Data points')
plt.plot(x, m*x + c, color='red', label='Fitted line')
plt.title('Linear Regression using Gradient Descent')
plt.xlabel('x')
plt.ylabel('y')
```

OUTPUT:

```
Final Slope (m): 2.021281045682893, Final Intercept (c): 2.923168672645527
```



EXPERIMENT 3

AIM: To implement linear regression using the gradient descent algorithm in Python and find the best fit for a set of data points.

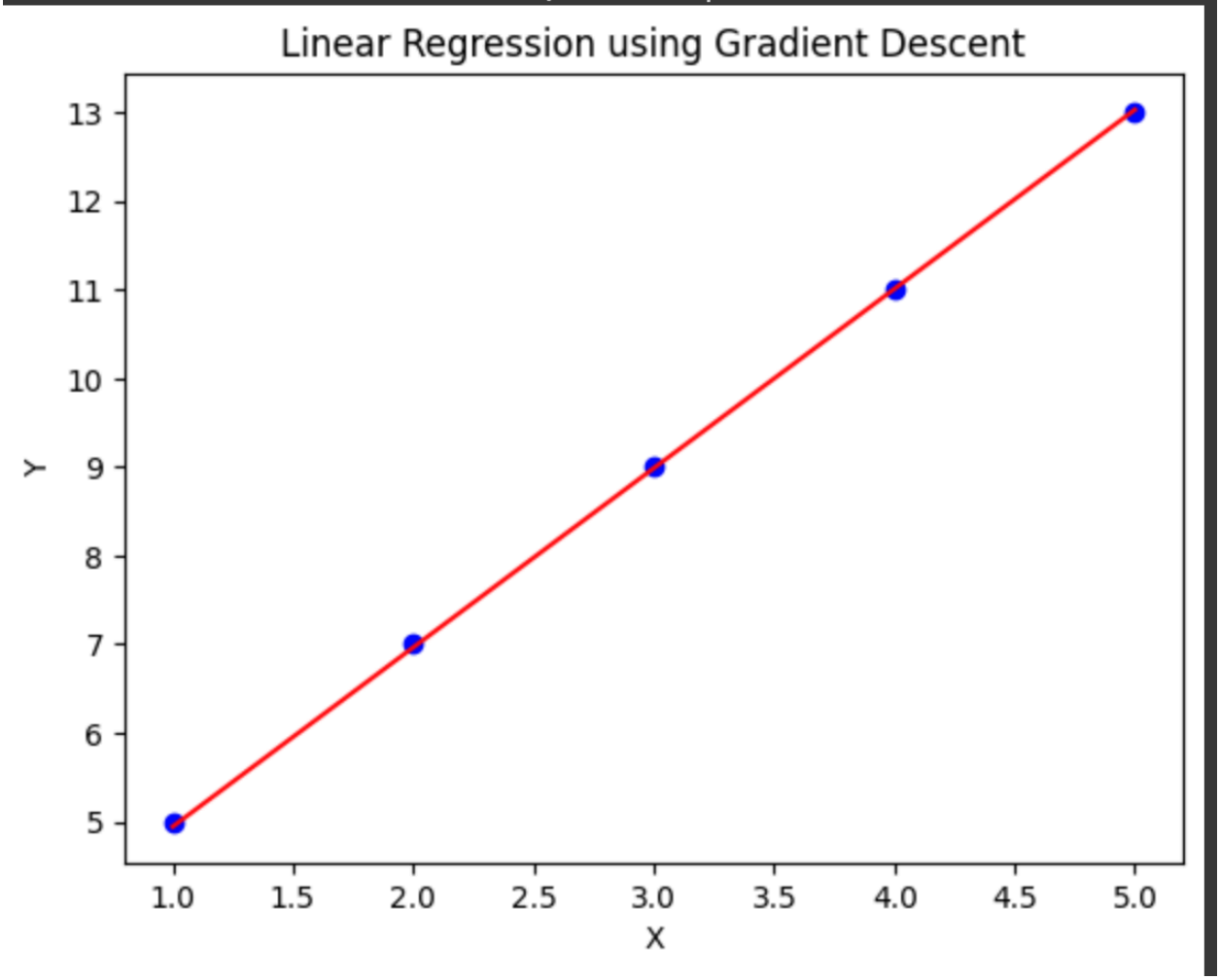
THEORY:
Linear regression is used to predict the dependent variable based on the independent variable(s). Gradient descent helps in finding the optimal values of the parameters (slope and intercept) by iteratively minimizing the cost function.

SOURCE CODE:

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

# Generate some data
X = np.array([1, 2, 3, 4, 5])
Y = np.array([5, 7, 9, 11, 13])

# Function for Linear Regression using Gradient Descent
def gradient_descent_linear(X, Y, learning_rate=0.01, iterations=1000):
    m, c = 0, 0
    n = len(X)

    for i in range(iterations):
        Y_pred = m * X + c
        D_m = (-2/n) * sum(X * (Y - Y_pred))
        D_c = (-2/n) * sum(Y - Y_pred)
        m = m - learning_rate * D_m
        c = c - learning_rate * D_c
    return m, c

# Perform gradient descent
m, c = gradient_descent_linear(X, Y)
print(f"Coefficient: {m}, Intercept: {c}")

# Plot
plt.scatter(X, Y, color='blue')
plt.plot(X, m*X + c, color='red')
plt.title('Linear Regression using Gradient Descent')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

OUTPUT:

```
Coefficient: 2.021281045682893, Intercept: 2.923168672645527
```



EXPERIMENT 4

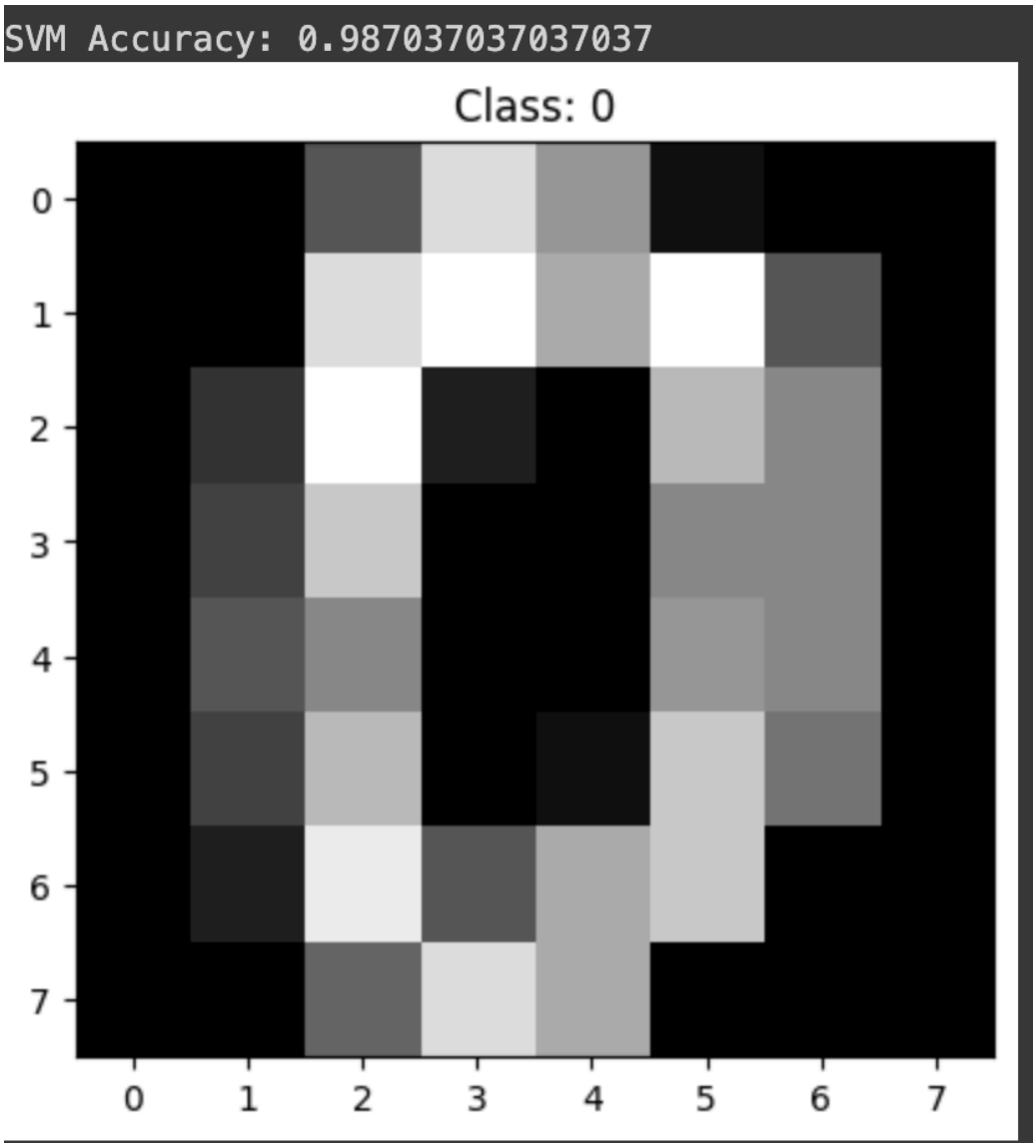
AIM: To implement and compare the classification accuracy of Support Vector Machines (SVM) and Convolutional Neural Networks (CNN) on a dataset.

THEORY:
Support Vector Machines (SVM) are supervised learning models used for classification and regression analysis. CNN is a class of deep learning models primarily used for image classification and processing tasks. CNNs are specialized for extracting features from image data, while SVM focuses on separating data points using hyperplanes.

SOURCE CODE:

```
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 (digits)
digits = datasets.load_digits()
X = digits.data
y = digits.target

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Train an SVM model
svm_model = SVC()
svm_model.fit(X_train, y_train)

# Test and print accuracy
y_pred = svm_model.predict(X_test)
print(f"SVM Accuracy: {accuracy_score(y_test, y_pred)}")

# Plot a sample from the dataset
plt.imshow(digits.images[0], cmap='gray')
plt.title(f'Class: {digits.target[0]}')
plt.show()
```

OUTPUT:



EXPERIMENT 5

AIM: To implement basic image handling and processing operations, such as reading, displaying, and applying filters to an image using Python.

THEORY:
Image processing involves manipulating images for various purposes, including noise reduction, enhancement, and feature extraction. Common operations include reading images, converting to grayscale, and applying filters like Gaussian blur for smoothing.

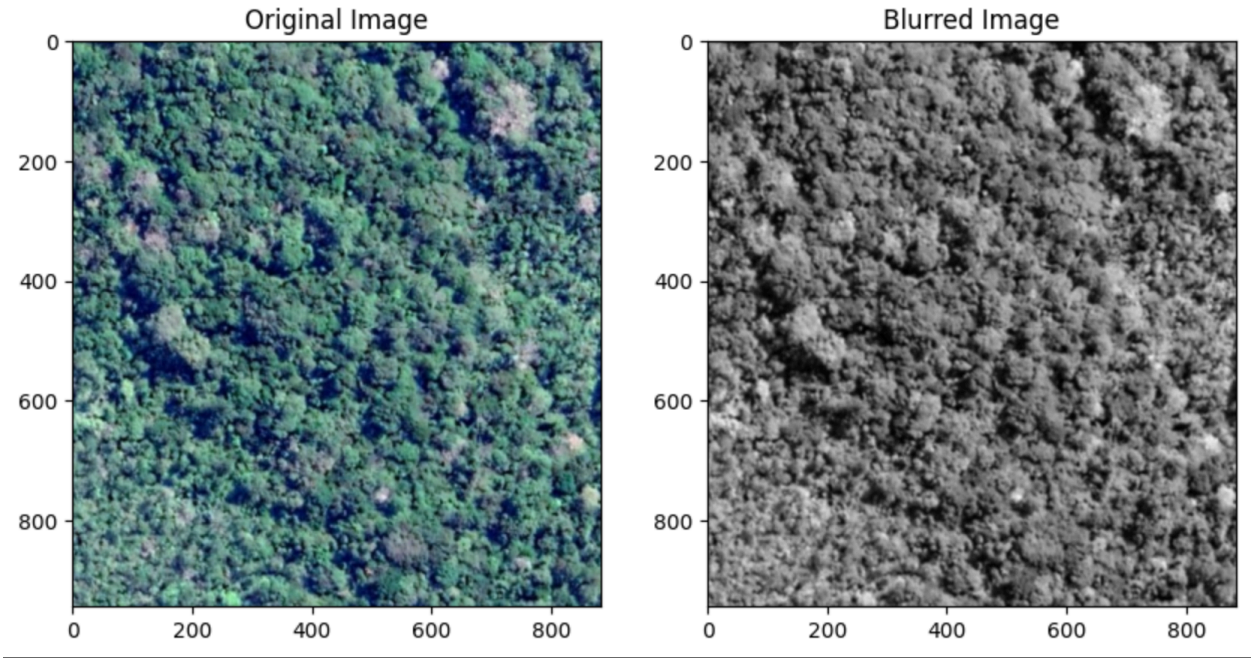
SOURCE CODE:

```
import cv2
from matplotlib import pyplot as plt
from google.colab import files

# Step 1: Upload the image from your local system to Google Colab
uploaded = files.upload()

# Step 2: Access the image filename from the uploaded dictionary
image_filename = next(iter(uploaded.keys()))

# Step 3: Load the image using OpenCV
img = cv2.imread(image_filename)

# Step 4: Check if the image was loaded successfully
if img is None:
    print("Error: Unable to load the image. Please check the file path.")
else:
    # Convert image to grayscale
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian Blur to the grayscale image
    blur_img = cv2.GaussianBlur(gray_img, (5, 5), 0)

    # Display the original and processed images
    plt.figure(figsize=(10, 5))

    # Original image
    plt.subplot(1, 2, 1)
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.title('Original Image')

    # Blurred image
    plt.subplot(1, 2, 2)
    plt.imshow(blur_img, cmap='gray')
```

OUTPUT:

- **Screenshot 2024-10-15 at 17.21.50.jpg**(image/jpeg) - 201793 bytes, last modified: 15/10/2024 - 100% done

```
Saving Screenshot 2024-10-15 at 17.21.50.jpg to Screenshot 2024-10-15 at 17.21.50.jpg
```



EXPERIMENT 6

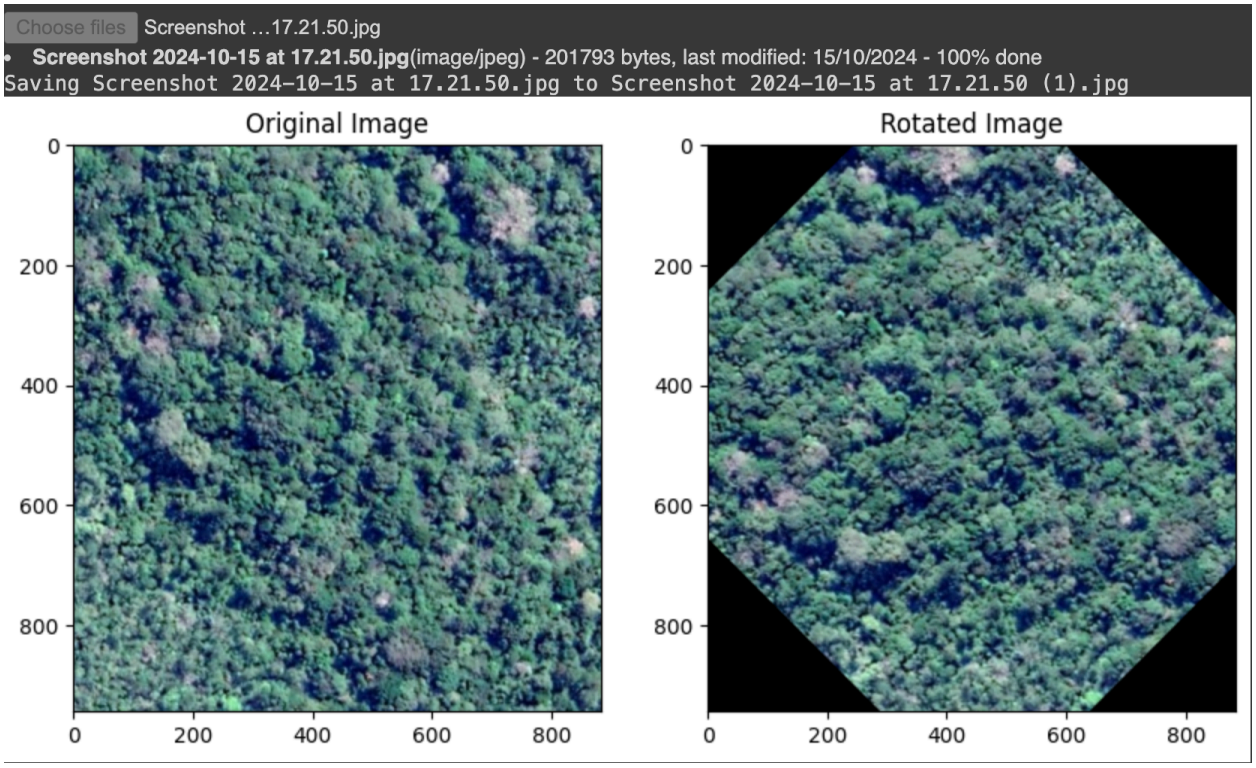
AIM: To implement geometric transformations such as rotation and scaling on an image using Python.

THEORY:
Geometric transformations modify the geometric structure of an image. Common transformations include translation, rotation, scaling, and shearing. These operations are fundamental for various image processing tasks such as object detection, image augmentation, and computer vision applications.

SOURCE CODE:

```
# Step 1: Upload the image
uploaded = files.upload()

# Step 2: Access the image filename from the uploaded dictionary
image_filename = next(iter(uploaded.keys()))

# Step 3: Load the image using OpenCV
img = cv2.imread(image_filename)

# Step 4: Check if the image was loaded successfully
if img is None:
    print("Error: Unable to load the image. Please check the file path.")
else:
    # Get the dimensions of the image
    (h, w) = img.shape[:2]

    # Define the center of the image (for rotation)
    center = (w // 2, h // 2)

    # Perform rotation by 45 degrees
    rotation_matrix = cv2.getRotationMatrix2D(center, 45, 1.0)
    rotated_img = cv2.warpAffine(img, rotation_matrix, (w, h))

    # Display original and rotated images
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.title('Original Image')

    plt.subplot(1, 2, 2)
    plt.imshow(cv2.cvtColor(rotated_img, cv2.COLOR_BGR2RGB))
    plt.title('Rotated Image')

    plt.show()
```

OUTPUT:


Choose files
Screenshot …17.21.50.jpg
Screenshot 2024-10-15 at 17.21.50.jpg(image/jpeg) - 201793 bytes, last modified: 15/10/2024 - 100% done
Saving Screenshot 2024-10-15 at 17.21.50.jpg to Screenshot 2024-10-15 at 17.21.50 (1).jpg
Original Image
Rotated Image
0
200
400
600
800

EXPERIMENT 7

AIM: To implement perspective transformation on an image and observe the change in viewing angles using Python.

THEORY:
Perspective transformation is a technique used to change the viewing perspective of an image. It maps points from one plane to another plane and is often used in tasks such as document scanning to correct skewed perspectives. It requires defining four points in the source image and four corresponding points in the destination image.

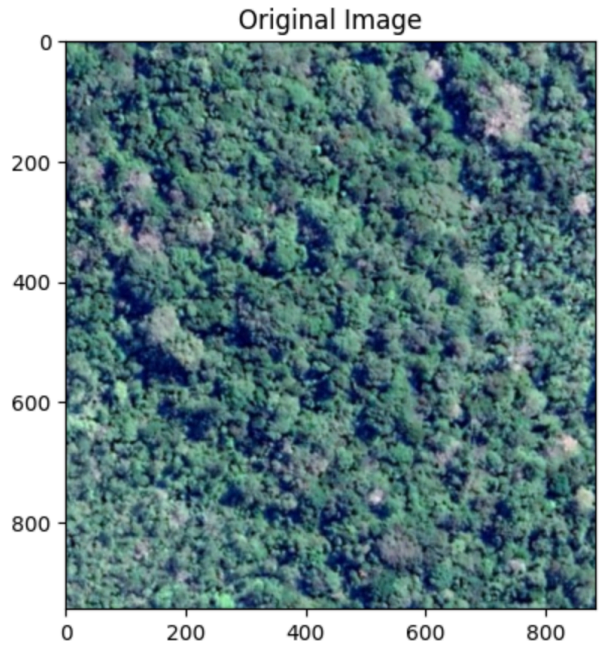
SOURCE CODE:

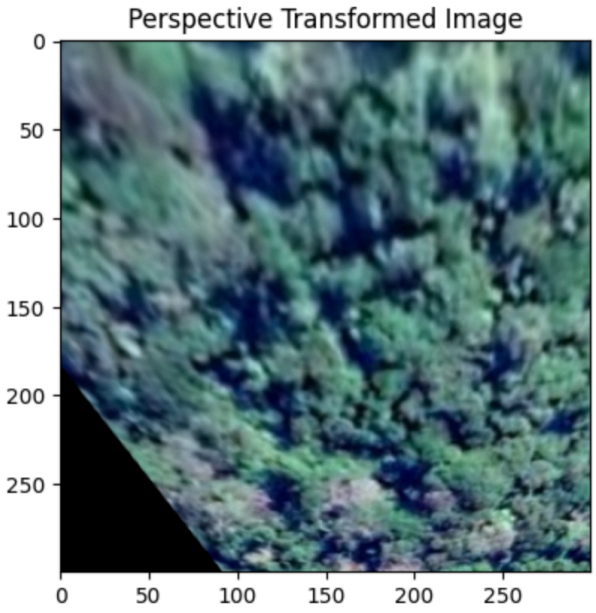
```
import cv2
import numpy as np
from matplotlib import pyplot as plt
from google.colab import files

# Step 1: Upload the image
uploaded = files.upload()

# Step 2: Access the image filename from the uploaded dictionary
image_filename = next(iter(uploaded.keys()))

# Step 3: Load the image using OpenCV
img = cv2.imread(image_filename)

# Step 4: Check if the image was loaded successfully
if img is None:
    print("Error: Unable to load the image. Please check the file path.")
else:
    # Define four points in the original image (manually chosen)
    pts1 = np.float32([[50, 50], [200, 50], [50, 200], [200, 200]])

    # Define four corresponding points in the output image
    pts2 = np.float32([[10, 100], [200, 50], [100, 250], [220, 220]])

    # Perspective transformation matrix
    matrix = cv2.getPerspectiveTransform(pts1, pts2)

    # Apply perspective transformation
    transformed_img = cv2.warpPerspective(img, matrix, (300, 300))

    # Display original and transformed images
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
```

```
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.title('Original Image')

    plt.subplot(1, 2, 2)
    plt.imshow(cv2.cvtColor(transformed_img, cv2.COLOR_BGR2RGB))
    plt.title('Perspective Transformed Image')

    plt.show()
```

OUTPUT:

```
Choose files  Screenshot ...17.21.50.jpg
• Screenshot 2024-10-15 at 17.21.50.jpg(image/jpeg) - 201793 bytes, last modified: 15/10/2024 - 100% done
Saving Screenshot 2024-10-15 at 17.21.50.jpg to Screenshot 2024-10-15 at 17.21.50 (2).jpg
```





EXPERIMENT 8

AIM: To calibrate a camera using Python by estimating the camera parameters such as intrinsic and extrinsic parameters to correct lens distortion.

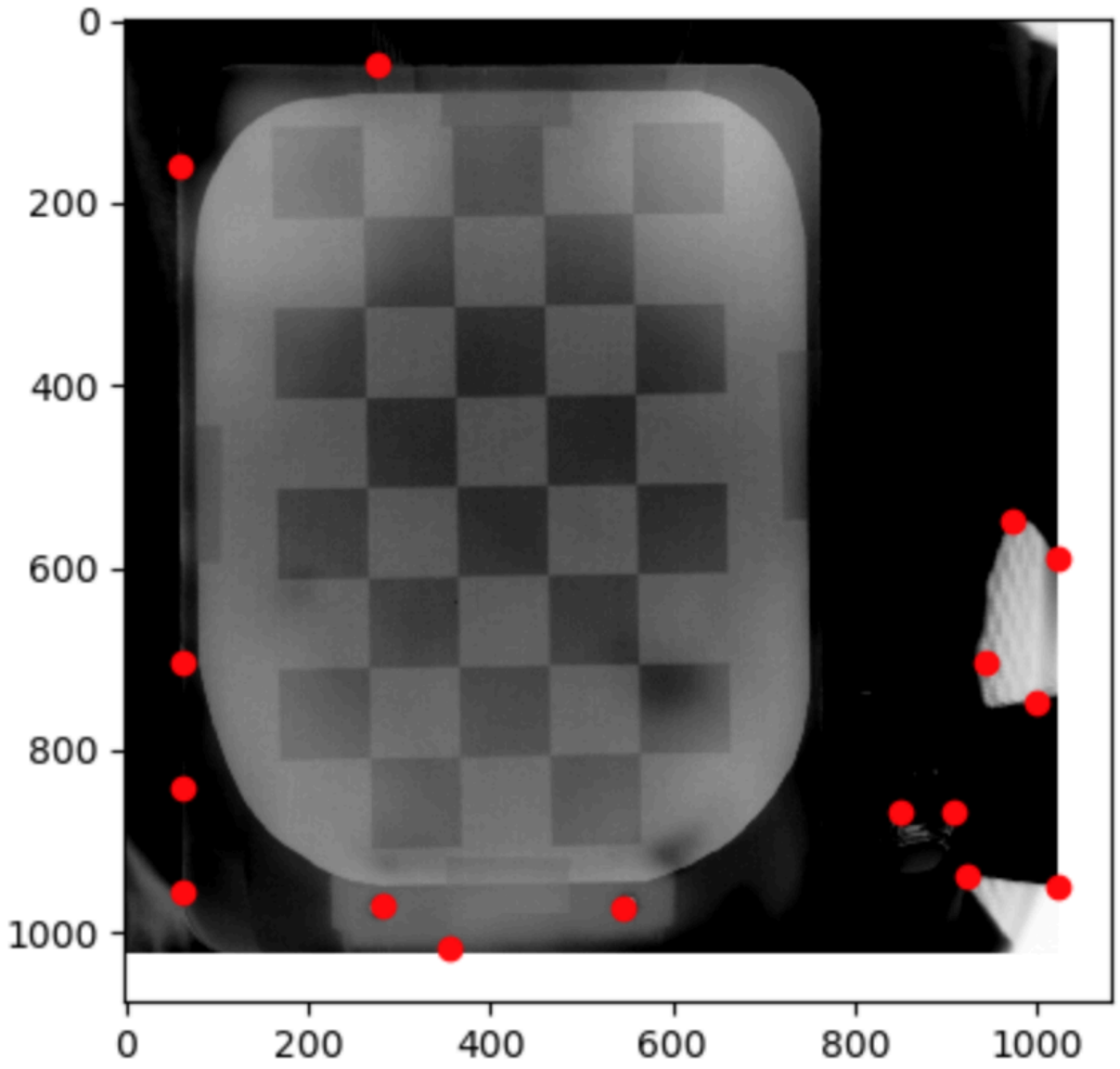
THEORY:
Camera calibration is the process of determining the internal parameters of a camera to correct distortions caused by lenses. Calibration typically involves using images of a known pattern (like a chessboard) to calculate the camera's intrinsic and extrinsic properties. These parameters help in 3D reconstruction, augmented reality, and other vision-based tasks.

SOURCE CODE:

```
import cv2
import numpy as np
import glob
from google.colab import files

# Upload a zip of calibration chessboard images if needed
uploaded = files.upload()

# Step 1: Prepare object points (0,0,0), (1,0,0), (2,0,0), etc.
objp = np.zeros((6*7,3), np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images
objpoints = []  # 3D points in real world space
imgpoints = []  # 2D points in image plane

# Load calibration images
images = glob.glob('*.jpg')  # Assuming you uploaded a zip file of calibration images

for fname in images:
    img = cv2.imread(fname)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Step 2: Find the chessboard corners
    ret, corners = cv2.findChessboardCorners(gray, (7,6), None)

    if ret == True:
        objpoints.append(objp)
        imgpoints.append(corners)

        # Step 3: Draw and display the corners
        img = cv2.drawChessboardCorners(img, (7,6), corners, ret)
        cv2.imshow('img', img)
        cv2.waitKey(500)
```

```python
cv2.destroyAllWindows()

# Step 4: Perform camera calibration
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)

# Print the camera matrix and distortion coefficients
print(f"Camera Matrix: \n{mtx}")
print(f"Distortion Coefficients: \n{dist}")
```

OUTPUT:



EXPERIMENT 9

AIM: To compute the fundamental matrix between two stereo images, which relates corresponding points in the two images using Python.

THEORY:
The fundamental matrix is used in computer vision to relate corresponding points in two stereo images. It encodes the intrinsic projective geometry between two views, making it crucial for tasks such as 3D reconstruction, where two different views of the same scene are used to extract depth information.

SOURCE CODE:

```
import cv2
import numpy as np

# Step 1: Example corresponding points from two images
pts1 = np.float32([[100, 100], [150, 200], [200, 250], [300, 300]])
pts2 = np.float32([[90, 110], [140, 210], [190, 260], [290, 310]])

# Step 2: Compute the fundamental matrix
F, mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)

# Step 3: Output the fundamental matrix
print("Fundamental Matrix:")
print(F)

# Example to visualize corresponding points (for demo purposes)
img1 = np.ones((400, 400, 3), np.uint8) * 255  # Dummy white image
img2 = np.ones((400, 400, 3), np.uint8) * 255  # Dummy white image

for pt1, pt2 in zip(pts1, pts2):
    img1 = cv2.circle(img1, tuple(pt1.astype(int)), 5, (255, 0, 0), -1)
    img2 = cv2.circle(img2, tuple(pt2.astype(int)), 5, (255, 0, 0), -1)

# Display the two images with corresponding points
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(img1)
plt.title('Image 1 with Corresponding Points')

plt.subplot(1, 2, 2)
plt.imshow(img2)
plt.title('Image 2 with Corresponding Points')

plt.show()
```

OUTPUT:

```
Fundamental Matrix:
None
```