Applying image processing techniques(scaling, rotation, blurring, edge detection) using OpenCV

➤ **Step 1: Install OpenCV**

```
!pip install opencv-python-headless
```

```
Requirement already satisfied: opencv-python-headless in /usr/local/lib/python3.10/dist-packages
(4.10.0.84)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/dist-packages (from
opencv-python-headless) (1.26.4)
```

OpenCV is a powerful library for image processing. The opencv-python-headless package is installed using the pip command in Google Colab. This version of OpenCV is designed for use in environments where GUI operations are not necessary, like in Colab.

➤ **Step 2: Import Necessary Libraries**

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Function to display an image using matplotlib
def display_image(img, title="Image"):
 plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.title(title)
plt.axis('off')
plt.show()

# Function to display two images side by side
def display_images(img1, img2, title1="Image 1", title2="Image 2"):
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
plt.title(title1)
plt.axis('off')

 plt.subplot(1, 2, 2)
plt.imshow(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB))
 plt.title(title2)
plt.axis('off')
plt.show()
```

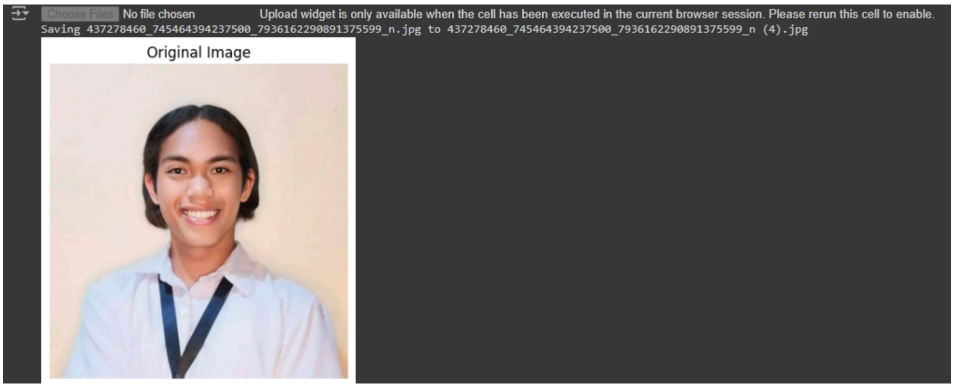
cv2: This imports OpenCV, which provides functions for image processing.
numpy (np): This library is used for handling arrays and matrices, which images are represented as.
matplotlib.pyplot (plt): This is used to display images in a Jupyter notebook or Google Colab environment.

➤ **Step 3: Load an Image**

```
from google.colab import files
from io import BytesIO
from PIL import Image

# Upload an image
uploaded = files.upload()

# Convert to OpenCV format
image_path = next(iter(uploaded)) # Get the image file name
 image = Image.open(BytesIO(uploaded[image_path]))
 image = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2BGR)
display_image(image, "Original Image")
```



display_image(): Converts the image from BGR (OpenCV's default color format) to RGB (the format expected by matplotlib) and displays it using imshow().
display_images(): This function allows two images to be displayed side by side for comparison. We use subplot to create a grid of plots (here, 1 row and 2 columns).

➢Exercise 1: Scaling and Rotation

```
# Scaling
def scale_image(img, scale_factor):
height, width = img.shape[:2]
scaled_img = cv2.resize(img,
 (int(width * scale_factor), int(height * scale_factor)), interpolation=cv2.INTER_LINEAR)
return scaled_img
```

"""
scale_image(): This function scales the image by a given factor. The cv2.resize() function takes the original dimensions of the image, multiplies them by the scale_factor, and resizes the image accordingly. INTER_LINEAR is a common interpolation method for resizing.
"""

```
# Rotate
 def rotate_image(img, angle):
height, width = img.shape[:2]
center = (width // 2, height // 2)
 matrix = cv2.getRotationMatrix2D(center, angle, 1.0)
 rotated_img = cv2.warpAffine(img, matrix, (width, height))
return rotated_img
```

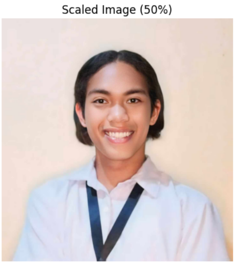
"""
rotate_image(): Rotates the image around its center. cv2.getRotationMatrix2D() creates a transformation matrix for rotation, and cv2.warpAffine() applies this transformation. The angle parameter controls the degree of rotation.
"""

```
# Scale image by 0.5
scaled_image = scale_image(image, 0.2)
 display_image(scaled_image, "Scaled Image (50%)")

 # Rotate image by 45 degrees
rotated_image = rotate_image(image, 45)
 display_image(rotated_image, "Rotated Image (45°)")
```

"""
These lines apply the scaling and rotation functions to the uploaded image and display the results.
"""

Scaled Image (50%)



Rotated Image (150°)

\nThese lines apply the scaling and rotation functions to the uploaded image and display the results.\n

➢ **Exercise 2: Blurring Techniques**

```
# Gaussian Blur
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0)
 display_image(gaussian_blur, "Gaussian Blur (5x5)")
```

"""

cv2.GaussianBlur(): Applies a Gaussian blur to the image, which smooths it by averaging the pixel values in a 5x5 kernel (a small matrix). This is useful for reducing noise in an image.

"""

```
# Median Blur
 median_blur = cv2.medianBlur(image, 5)
display_image(median_blur, "Median Blur (5x5)")
```

"""

cv2.medianBlur(): Applies a median blur, which replaces each pixel's value with the median value of its neighbours in a 5x5 kernel. This method is particularly effective in removing salt-and-pepper noise.

"""

Gaussian Blur (5x5)

Median Blur (5x5)

➢ **Exercise 3: Edge Detection using Canny**

```
# Canny Edge Detection
edges = cv2.Canny(image, 100, 200)
display_image(edges, "Canny Edge Detection (100, 200)")
```

""

cv2.Canny(): Detects edges in the image by calculating the gradient (rate of intensity change) between pixels. The two threshold values (100 and 200) define the edges' sensitivity. Lower thresholds detect more edges, while higher thresholds detect only the most prominent edges.

""

Canny Edge Detection (100, 200)

➢Exercise 4: Basic Image Processor (Interactive)

```
def process_image(img, action):
    if action == 'scale':
        return scale_image(img, 0.5)
    elif action == 'rotate':
        return rotate_image(img, 45)
    elif action == 'gaussian_blur':
        return cv2.GaussianBlur(img, (5, 5), 0)
    elif action == 'median_blur':
        return cv2.medianBlur(img, 5)
    elif action == 'canny':
        return cv2.Canny(img, 100, 200)
    else:
        return img
```

""""

process_image(): This function allows users to specify an image transformation (scaling, rotation, blurring, or edge detection). Depending on the action passed, it will apply the corresponding image processing technique and return the processed image.

""""

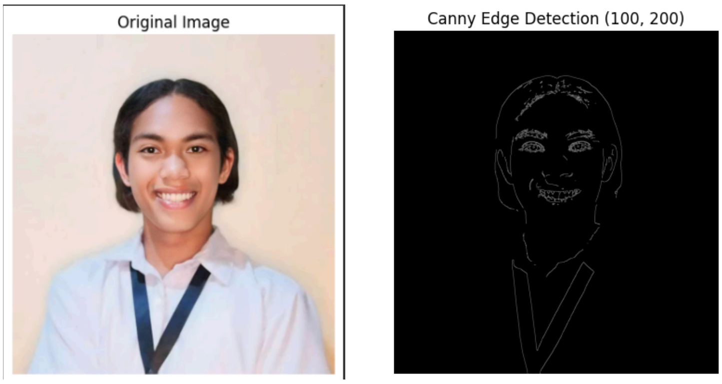
```
action = input("Enter action (scale, rotate, gaussian_blur, median_blur, canny): ")
processed_image = process_image(image, action)
display_images(image, processed_image, "Original Image", f"Processed Image ({action})")
```

""""

This allows users to enter their desired transformation interactively (via the input() function). It processes the image and displays both the original and transformed versions side by side.

""""

```
Requirement already satisfied: opencv-python in /usr/local/lib/python3.10/dist-packages (4.10.0.84)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.10/dist-packages (from opencv-python)
(1.26.4)
Enter action (scale, rotate, gaussian_blur, median_blur, canny): canny
```



➢Exercise 5: Comparison of Filtering Techniques

```
# Applying Gaussian, Median, and Bilateral filters
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0)
median_blur = cv2.medianBlur(image, 5)
bilateral_filter = cv2.bilateralFilter(image, 9, 75, 75)
```

""""

cv2.bilateralFilter(): This filter smooths the image while keeping edges sharp, unlike Gaussian or median filters. It's useful for reducing noise while preserving details.

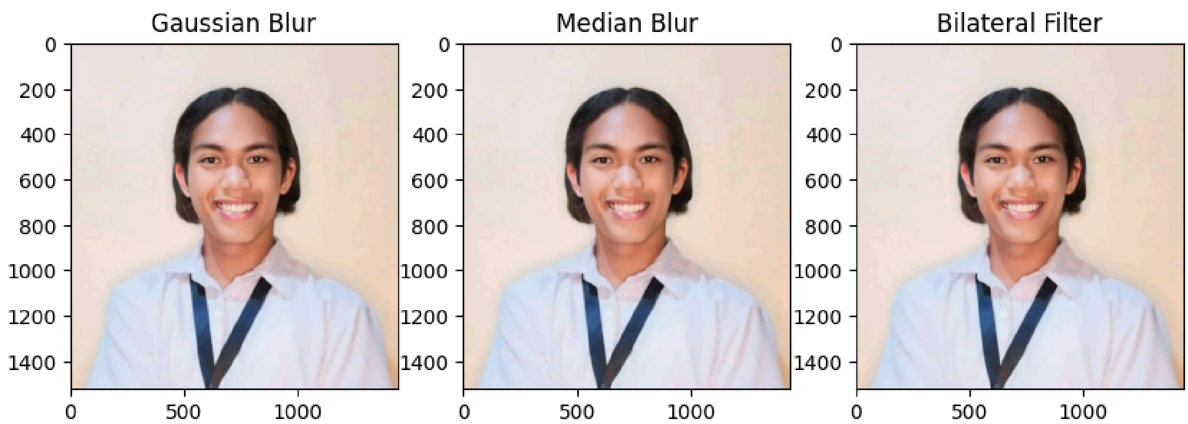
""""

```
# Display the results for comparison
plt.figure(figsize=(10, 5))
 plt.subplot(1, 3, 1)
plt.imshow(cv2.cvtColor(gaussian_blur, cv2.COLOR_BGR2RGB))
 plt.title("Gaussian Blur")

plt.subplot(1, 3, 2)
plt.imshow(cv2.cvtColor(median_blur, cv2.COLOR_BGR2RGB))
 plt.title("Median Blur")
plt.subplot(1, 3, 3)
plt.imshow(cv2.cvtColor(bilateral_filter, cv2.COLOR_BGR2RGB))
plt.title("Bilateral Filter")

 plt.show()

""""
Explanation: This displays the images processed by different filtering techniques (Gaussian, Median, and Bilateral) side by side for comparison.
""""
```




```
\nExplanation: This displays the images processed by different filtering techniques
(Gaussian,\nMedian, and Bilateral) side by side for comparison.\n
```

➢Exercise 6: Sobel Edge Detection

```
# Sobel Edge Detection
def sobel_edge_detection(img):
# Convert to grayscale
 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Sobel edge detection in the x direction
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=5)
# Sobel edge detection in the y direction
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=5)
# Combine the two gradients
sobel_combined = cv2.magnitude(sobelx, sobely)
return sobel_combined
# Apply Sobel edge detection to the uploaded image
sobel_edges = sobel_edge_detection(image)
plt.imshow(sobel_edges, cmap='gray')
plt.title("Sobel Edge Detection")
plt.axis('off')
plt.show()
```

Conclusion:

This code demonstrates various image processing techniques using OpenCV in Google Colab.
It covers image loading, displaying, scaling, rotation, blurring, and edge detection.
We explored Gaussian Blur, Median Blur, Bilateral Filter, Canny Edge Detection, and Sobel Edge Detection.
These techniques are fundamental for tasks like image enhancement, noise reduction, object detection, and feature extraction. By comparing different filtering methods, we can understand their impact on image quality and edge preservation. Furthermore, the code showcases how to interact with Google Colab's file system for uploading and processing images. The combination of OpenCV and Colab offers a powerful environment for exploring and experimenting with image processing.