**THE SUPERIOR UNIVERSITY LAHORE** 

**LAB#5**

**Semester: 4th Se~~ctio~~n: AI (B)**

**Faculty of Computer Science and Information Technology Deadline:**

**Subject: PAI LAB Total Marks:**

**Name: M.Hassan Shahid**

**Roll No: 059**

***Instructions:***

* Copying of the assignment willresult in failure.
* Assignment should be submitted in word or pdf.

Below is a comprehensive, humanized report that explains the provided code step by step. This report covers the tools, technologies, and purpose of each code block in a friendly and clear manner.

# 2.1

# Detailed Report

# Image Processing Code

This report walks through a Python script that leverages popular libraries for image processing. The goal is to help you understand each section of the code, what it does, and how it all ties together. We’ll look at the tools used, the operations performed, and the role of each code block in a way that’s easy to follow.

## Introduction to the Tools and Technologies

### OpenCV (cv2)

* **Purpose:** OpenCV (Open Source Computer Vision Library) is widely used for real-time image processing and computer vision tasks. It offers a vast range of functionalities from basic image manipulations to complex machine learning operations.
* **Usage in Code:** We use OpenCV to read, display, save, and perform various arithmetic and bitwise operations on images.

### NumPy

* **Purpose:** NumPy is a powerful library for numerical computing in Python. It handles operations on arrays and matrices efficiently.
* **Usage in Code:** It is used here to create arrays (for instance, generating a brightness-increasing array) and to work with image data.

### Matplotlib

* **Purpose:** Matplotlib is a plotting library that is excellent for visualizing data. In image processing, it helps in displaying images in a more controlled and customizable manner.
* **Usage in Code:** Matplotlib is used to display images in a window, converting color spaces when necessary for proper visualization.

## Code Block Breakdown and Explanations

### 1. Importing Libraries

import cv2

import numpy as np

from matplotlib import pyplot as plt

* **What it does:**  
  This block imports the necessary libraries:
  + cv2 is for all OpenCV functions.
  + numpy (imported as np) is used for numerical operations.
  + pyplot from matplotlib is used for displaying images in a plot-like window.
* **Why it's important:**  
  These libraries are the foundation for our image processing tasks. Without them, we wouldn’t have access to the functions and operations that make the rest of the code work.

### 2. Defining a Function to Display Images

def show\_image(img, title="Image"):

plt.imshow(cv2.cvtColor(img, cv2.COLOR\_BGR2RGB) if len(img.shape) == 3 else img, cmap="gray")

plt.title(title)

plt.axis("off")

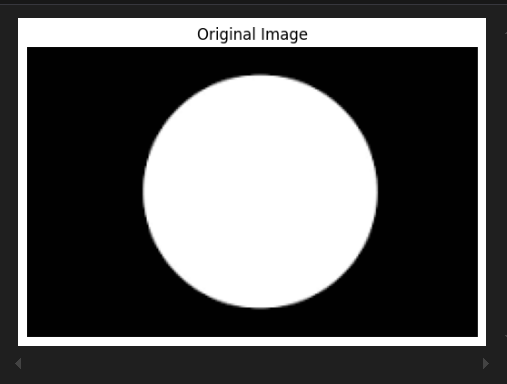
plt.show()

* **What it does:**  
  The show\_image function is designed to display an image using Matplotlib. It:
  + Checks if the image is in color (i.e., has three dimensions) and converts it from BGR (OpenCV’s default) to RGB (Matplotlib’s default) for correct color representation.
  + If the image is grayscale, it directly displays it using a grayscale colormap.
  + Adds a title, turns off axis labels for a cleaner look, and then renders the image.
* **Why it's important:**  
  This utility function encapsulates the display logic in one place, making the rest of the code easier to read and maintain.

### 3. Reading an Image

img = cv2.imread( r"C:\Users\Admin\Pictures\Screenshots\2.1.png)

show\_image(img, "Original Image")



* **What it does:**  
  This block:
  + Reads an image from the specified file path using OpenCV’s imread function.
  + Immediately displays the image using the show\_image function with the title “Original Image.”
* **Why it's important:**  
  Loading the image is the first essential step in any image processing pipeline. It brings your image data into the program so that subsequent operations can be performed on it.

### 4. Displaying an Image Using OpenCV

cv2.imshow("Displayed Image", img)

cv2.waitKey(200)

cv2.destroyAllWindows()

* **What it does:**  
  Here, OpenCV’s GUI functionality is used to:
  + Open a new window and display the image.
  + Pause the execution for 200 milliseconds with cv2.waitKey(200).
  + Close the window automatically using cv2.destroyAllWindows().
* **Why it's important:**  
  While Matplotlib is great for quick visualization, OpenCV’s imshow is another way to view images, particularly when you need a more native window (which can sometimes be more interactive).

### 5. Writing (Saving) an Image

cv2.imwrite("saved\_image.jpg", img)

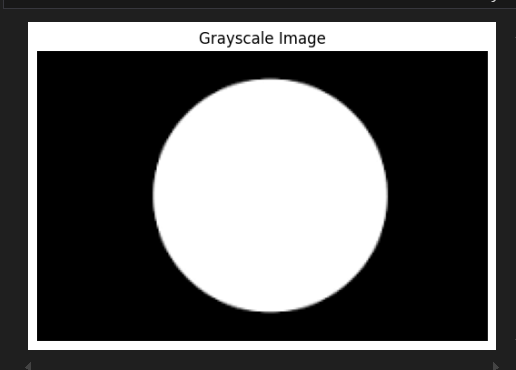
print("Image Saved")

* **What it does:**  
  This block writes the image to disk with the filename “saved\_image.jpg” using OpenCV’s imwrite function.
* **Why it's important:**  
  Saving an image is crucial for tasks where you need to preserve the result of your processing pipeline, such as exporting results for reports or further processing.

### 6. Converting to a Different Color Space (Grayscale)

gray\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

show\_image(gray\_img, "Grayscale Image")

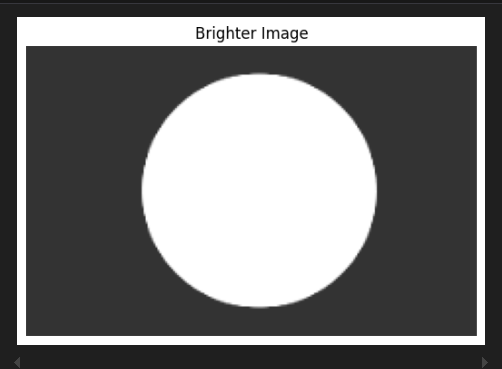


* **What it does:**  
  The image is converted from BGR to grayscale using cv2.cvtColor.
* **Why it's important:**  
  Converting an image to grayscale is a common preprocessing step. It reduces complexity by eliminating color information and is often used for edge detection, thresholding, and other operations where color is not necessary.

### 7. Arithmetic Operations on Images (Brightening)

brighter\_img = cv2.add(img, np.ones(img.shape, dtype=np.uint8) \* 50)

show\_image(brighter\_img, "Brighter Image")



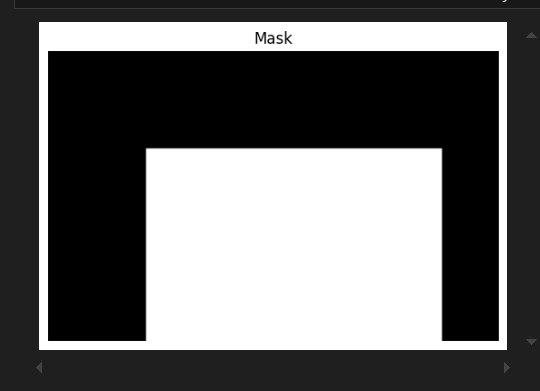
* **What it does:**
  + An array of the same shape as the image is created with every element set to 50.
  + The array is added to the original image using OpenCV’s cv2.add, increasing the brightness.
* **Why it's important:**  
  Brightening an image is a typical enhancement technique that can help in visualizing details better, especially in underexposed images.

### 8. Creating and Displaying a Mask

mask = np.zeros(img.shape[:2], dtype=np.uint8)

cv2.rectangle(mask, (50, 50), (200, 200), 255, -1)

show\_image(mask, "Mask")



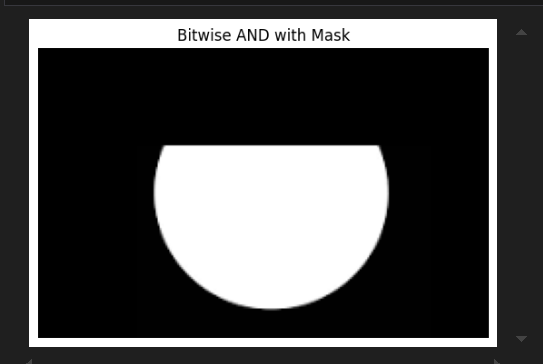
* **What it does:**
  + A blank (black) mask is created with the same height and width as the image.
  + A white rectangle is drawn on this mask using cv2.rectangle. The white area (pixel value 255) indicates the region of interest.
* **Why it's important:**  
  Masks are used to isolate parts of an image. Here, the mask will later be used to perform bitwise operations on a specific region of the image.

### 9. Bitwise Operations Using the Mask

#### Bitwise AND

bitwise\_and = cv2.bitwise\_and(img, img, mask=mask)

show\_image(bitwise\_and, "Bitwise AND with Mask")

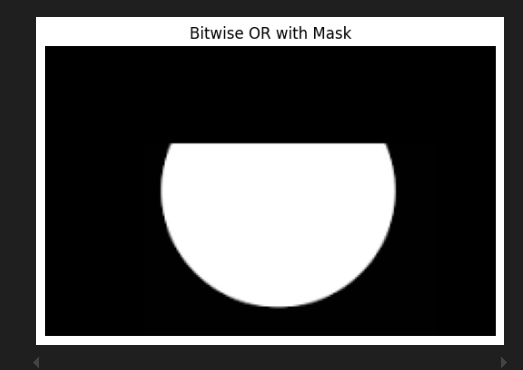


* **What it does:**  
  This operation retains only the parts of the image where the mask is white (the region defined by the rectangle). It’s like “cutting out” that part of the image.
* **Why it's important:**  
  Bitwise AND is often used in image segmentation and in highlighting regions of interest.

#### Bitwise OR

bitwise\_or = cv2.bitwise\_or(img, img, mask=mask)

show\_image(bitwise\_or, "Bitwise OR with Mask")

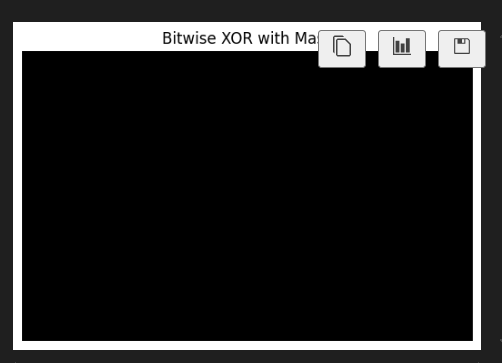


* **What it does:**  
  Similar to the AND operation, but in this case, it’s used for combining two images based on the mask. Since we’re applying it on the same image, the effect may seem subtle.
* **Why it's important:**  
  The OR operation is useful when you want to merge information from multiple images or create overlays.

#### Bitwise XOR

bitwise\_xor = cv2.bitwise\_xor(img, img, mask=mask)

show\_image(bitwise\_xor, "Bitwise XOR with Mask")

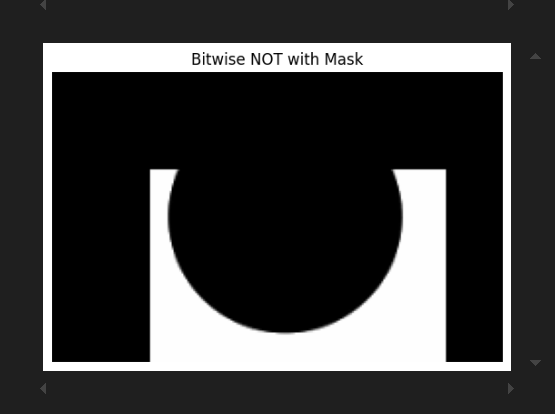


* **What it does:**  
  XOR highlights differences between the two images. With the mask applied on the same image, it creates an effect that may isolate the masked area in a unique way.
* **Why it's important:**  
  XOR is useful in scenarios where detecting differences or changes is necessary.

#### Bitwise NOT

bitwise\_not = cv2.bitwise\_not(img, mask=mask)

show\_image(bitwise\_not, "Bitwise NOT with Mask")



* **What it does:**  
  The NOT operation inverts the pixel values of the image, but only where the mask is white. This produces a negative-like effect on the masked region.
* **Why it's important:**  
  Inverting images is a common technique in image analysis and can help in emphasizing certain features or patterns.

## Conclusion

This Python script provides a hands-on introduction to basic image processing techniques using OpenCV, NumPy, and Matplotlib. Starting with reading and displaying an image, the code demonstrates essential tasks such as saving images, converting color spaces, performing arithmetic operations for brightness adjustment, and applying bitwise operations with a mask.

By understanding each code block and its purpose, you now have a solid foundation to explore more advanced image processing techniques. The friendly breakdown above should serve as a useful guide for both beginners and those looking to refresh their understanding of these fundamental concepts.

# ****2.2****

# ****Report on Image Processing Operations Using OpenCV****

This report provides an easy-to-understand breakdown of the Python script, which demonstrates various image processing techniques using **OpenCV**. We’ll explore each function, its role, and how it manipulates images to achieve specific results.

## ****Introduction to the Tools and Technologies****

### ****OpenCV (cv2)****

* **Purpose:** OpenCV (Open Source Computer Vision Library) is a powerful library for computer vision and image processing. It provides functionalities like reading, modifying, and analyzing images efficiently.
* **Usage in Code:** OpenCV is used for tasks like resizing images, blurring, edge detection, thresholding, morphological transformations, and background subtraction.

### ****NumPy****

* **Purpose:** NumPy is a numerical computing library that allows easy manipulation of arrays.
* **Usage in Code:** It is used to create kernels (for erosion, dilation, and morphological operations) and handle masks in certain transformations.

### ****Matplotlib****

* **Purpose:** Matplotlib is a visualization library that helps display images in Python.
* **Usage in Code:** Matplotlib is used to display the processed images in a user-friendly format.

## ****Code Block Breakdown and Explanation****

### ****1. Importing Required Libraries****

import cv2

import numpy as np

import matplotlib.pyplot as plt

* **What it does:**  
  This imports OpenCV (cv2), NumPy (np), and Matplotlib (plt) for image processing and visualization.
* **Why it's important:**  
  These libraries are essential for handling images, performing transformations, and displaying results.

### ****2. Defining a Function to Display Images****

def show\_image(image, title):

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

plt.title(title)

plt.axis("off")

plt.show()

* **What it does:**
  + Converts OpenCV's default BGR image format to RGB for correct display in Matplotlib.
  + Displays the image with a title.
* **Why it's important:**  
  Since OpenCV uses BGR color format while Matplotlib expects RGB, this function ensures proper visualization.

### ****3. Loading an Image****

IMG\_PATH = r"C:\Users\Admin\Pictures\new wallpapers\jinx-smile-lol-2k-wallpaper-uhdpaper.com-569@2@a.jpg"

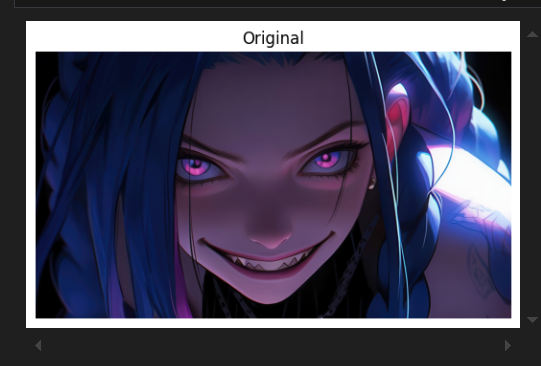
img = cv2.imread(IMG\_PATH)

if img is None:

print("Image not found")

exit()

show\_image(img, "Original")

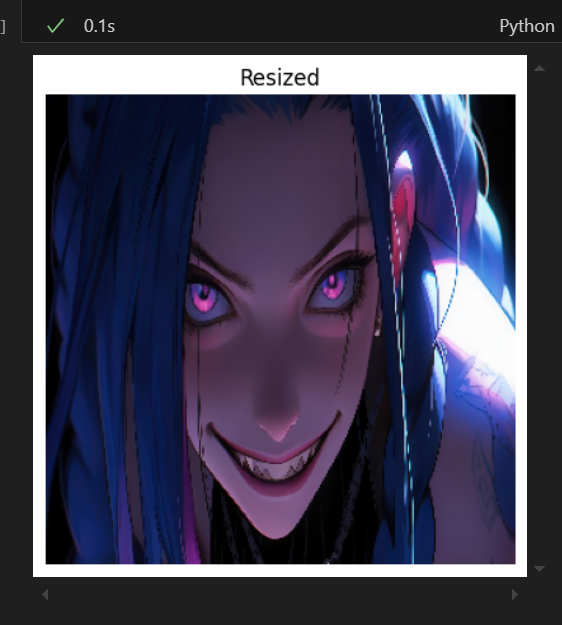


* **What it does:**
  + Reads the image from the given file path.
  + If the image is not found, it prints an error message and exits the program.
  + Displays the original image.
* **Why it's important:**  
  Loading an image is the first step before any processing can be done.

### ****4. Resizing the Image****

img\_resized = cv2.resize(img, (300, 300))

show\_image(img\_resized, "Resized")

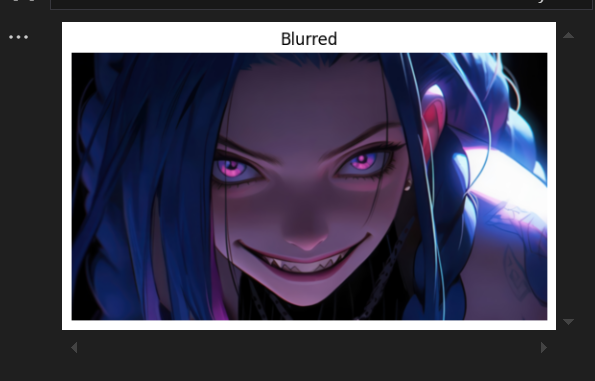


* **What it does:**
  + Resizes the image to 300x300 pixels.
* **Why it's important:**  
  Resizing is commonly used to standardize images for further processing or reduce memory usage.

### ****5. Applying Gaussian Blur****

img\_blurred = cv2.GaussianBlur(img, (15, 15), 0)

show\_image(img\_blurred, "Blurred")

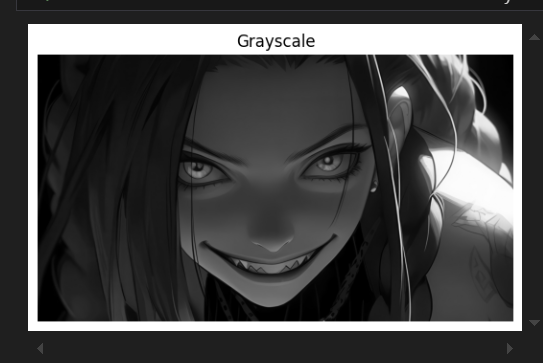


* **What it does:**
  + Applies a Gaussian blur with a 15x15 kernel.
* **Why it's important:**  
  Blurring helps in noise reduction and pre-processing for tasks like edge detection.

### ****6. Converting to Grayscale****

img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

show\_image(img\_gray, "Grayscale")

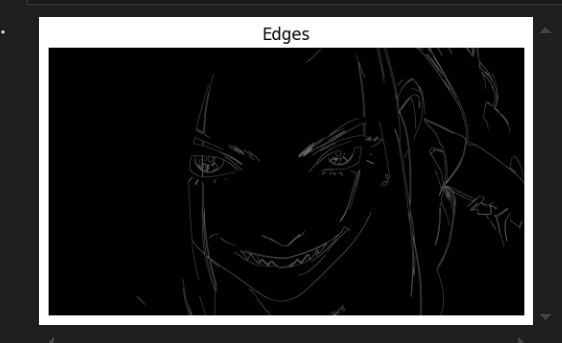


* **What it does:**
  + Converts the image from color to grayscale.
* **Why it's important:**  
  Many image processing techniques work better on grayscale images.

### ****7. Edge Detection****

img\_edges = cv2.Canny(img\_gray, 100, 200)

show\_image(img\_edges, "Edges")



* **What it does:**
  + Detects edges using the Canny algorithm with thresholds 100 and 200.
* **Why it's important:**  
  Edge detection is useful in object detection and shape analysis.

### ****8. Erosion and Dilation****

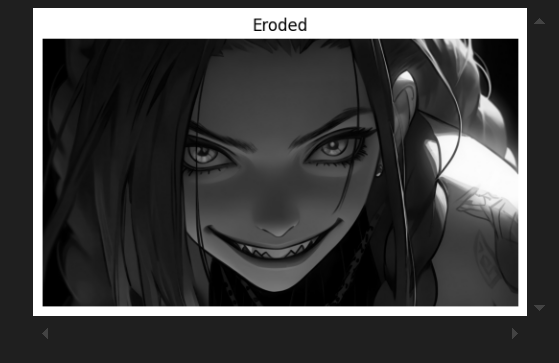
kernel = np.ones((5, 5), np.uint8)

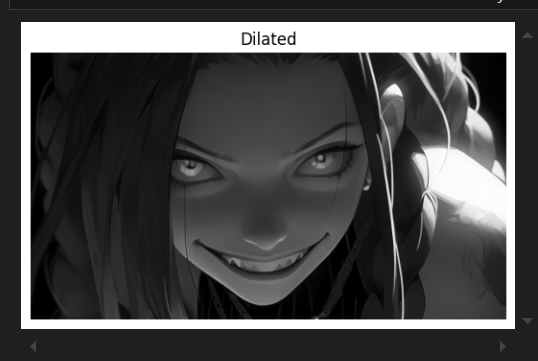
img\_eroded = cv2.erode(img\_gray, kernel, iterations=1)

show\_image(img\_eroded, "Eroded")

img\_dilated = cv2.dilate(img\_gray, kernel, iterations=1)

show\_image(img\_dilated, "Dilated")





* **What it does:**
  + **Erosion:** Shrinks bright regions (removes small white noise).
  + **Dilation:** Expands bright regions (fills small gaps).
* **Why it's important:**  
  These operations are used in noise reduction, shape manipulation, and feature extraction.

### ****9. Thresholding Techniques****

\_, img\_thresh = cv2.threshold(img\_gray, 127, 255, cv2.THRESH\_BINARY)

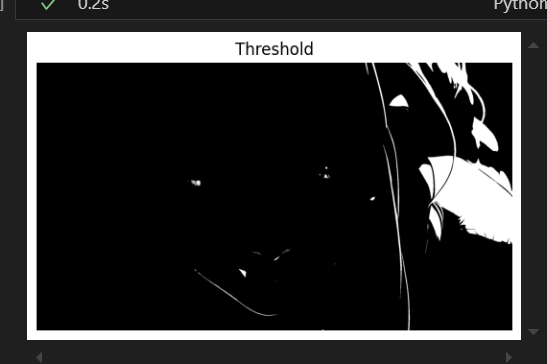
show\_image(img\_thresh, "Threshold")

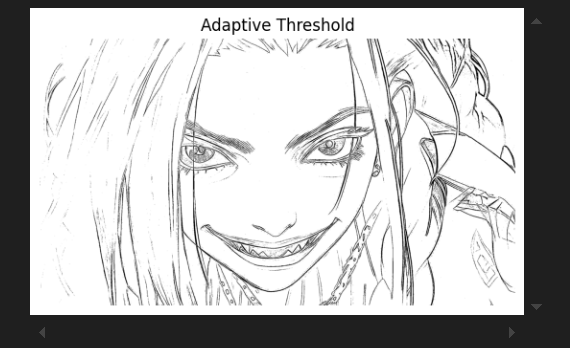
img\_adaptive = cv2.adaptiveThreshold(img\_gray, 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 11, 2)

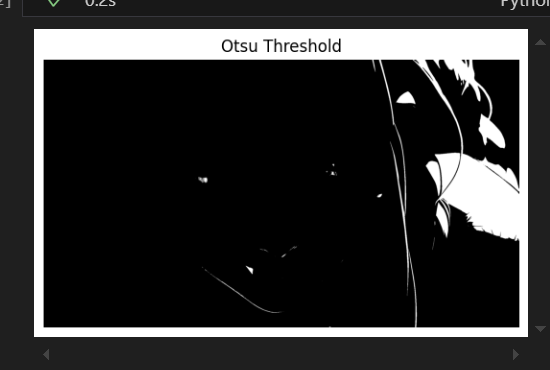
show\_image(img\_adaptive, "Adaptive Threshold")

\_, img\_otsu = cv2.threshold(img\_gray, 0, 255, cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)

show\_image(img\_otsu, "Otsu Threshold")





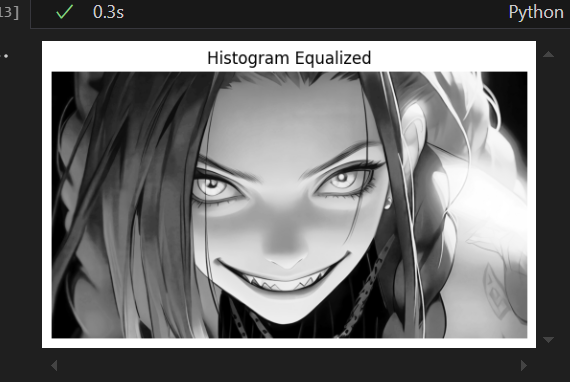


* **What it does:**
  + **Thresholding:** Converts pixels above 127 to white and below to black.
  + **Adaptive Thresholding:** Dynamically determines the threshold for different regions.
  + **Otsu’s Thresholding:** Automatically finds the optimal threshold value.
* **Why it's important:**  
  Used for object segmentation and binarization of images.

### ****10. Histogram Equalization****

img\_hist\_eq = cv2.equalizeHist(img\_gray)

show\_image(img\_hist\_eq, "Histogram Equalized")



* **What it does:**
  + Enhances contrast by spreading out intensity values.
* **Why it's important:**  
  Useful for improving visibility in low-contrast images.

### ****11. Morphological Operations****

img\_open = cv2.morphologyEx(img\_gray, cv2.MORPH\_OPEN, kernel)

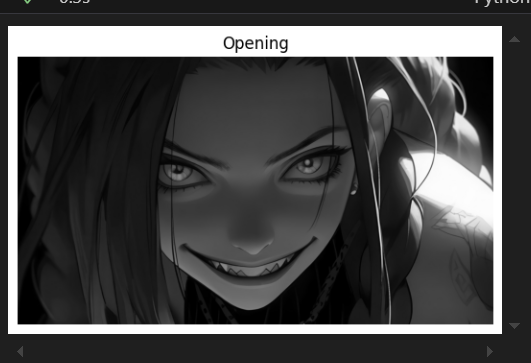
show\_image(img\_open, "Opening")

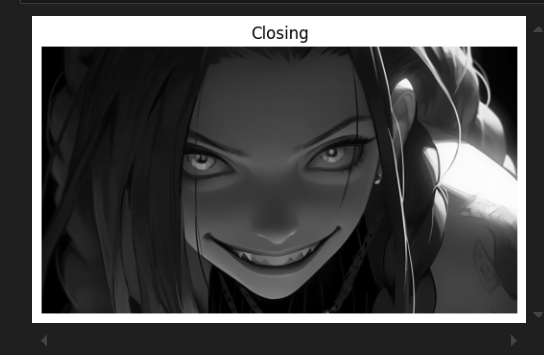
img\_close = cv2.morphologyEx(img\_gray, cv2.MORPH\_CLOSE, kernel)

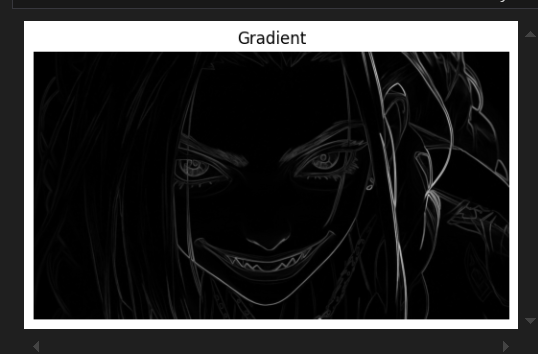
show\_image(img\_close, "Closing")

img\_grad = cv2.morphologyEx(img\_gray, cv2.MORPH\_GRADIENT, kernel)

show\_image(img\_grad, "Gradient")







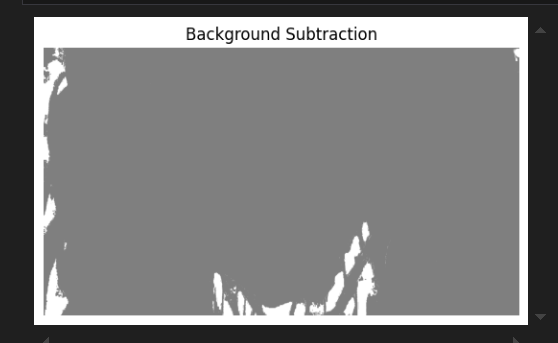
* **What it does:**
  + **Opening:** Removes small noise (erosion followed by dilation).
  + **Closing:** Fills small holes (dilation followed by erosion).
  + **Gradient:** Highlights edges by subtracting eroded image from dilated image.
* **Why it's important:**  
  These techniques are used for refining shapes and noise removal.

### ****12. Background Subtraction****

fgbg = cv2.createBackgroundSubtractorMOG2()

fgmask = fgbg.apply(img)

show\_image(fgmask, "Background Subtraction")



* **What it does:**
  + Detects moving objects by removing the background.
* **Why it's important:**  
  Used in video surveillance and object tracking.

### ****13. GrabCut Foreground Extraction****

mask = np.zeros(img.shape[:2], np.uint8)

rect = (50, 50, 300, 500)

bgd = np.zeros((1, 65), np.float64)

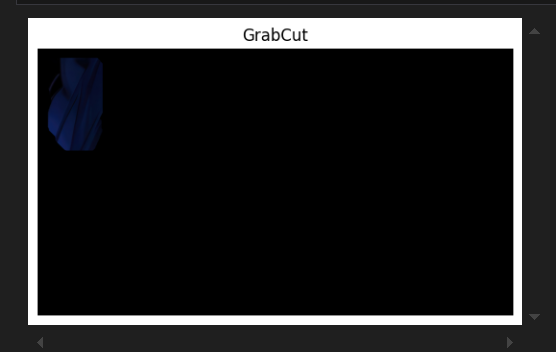
fgd = np.zeros((1, 65), np.float64)

cv2.grabCut(img, mask, rect, bgd, fgd, 5, cv2.GC\_INIT\_WITH\_RECT)

mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')

img\_grabcut = img \* mask2[:, :, np.newaxis]

show\_image(img\_grabcut, "GrabCut")

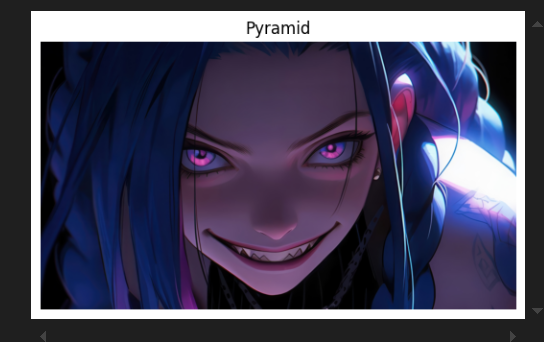


* **What it does:**
  + Extracts the foreground from the background using GrabCut.
* **Why it's important:**  
  Useful for background removal and object segmentation.

### ****14. Image Pyramid (Downsampling)****

img\_pyramid = cv2.pyrDown(img)

show\_image(img\_pyramid, "Pyramid")



* **What it does:**
  + Reduces the image size while retaining important features.
* **Why it's important:**  
  Used in image scaling and object detection.

## ****Conclusion****

This script provides a hands-on demonstration of essential image processing techniques, covering transformations, filters, segmentation, and more. Mastering these techniques will help in tasks like object detection, pattern recognition, and computer vision applications.

# 2.3

# Report on Feature Detection Using OpenCV

## Introduction

In this script, we dive into a variety of feature detection techniques using Python’s OpenCV library. Whether you're interested in detecting lines, circles, or faces with smiles, this code covers a broad range of capabilities that can be applied to many computer vision tasks. With a humanized explanation for every step, you'll understand how each block of code contributes to extracting meaningful information from images.

## Code Block Explanations

### 1. Displaying Images with a Custom Function

def show(img, title):

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

plt.imshow(img)

plt.title(title)

plt.axis('off')

plt.show()

* **What It Does:**  
  This helper function converts images from OpenCV’s default BGR color space to RGB so that they display correctly using Matplotlib. It also sets the title and removes the axis for a cleaner look.
* **Why It’s Important:**  
  Having a reusable function for showing images makes the code more organized and ensures that every image is displayed consistently.

### 2. Line Detection

img1 = cv2.imread('images.jpeg')

if img1 is not None:

gray = cv2.cvtColor(img1, cv2.COLOR\_BGR2GRAY)

edges = cv2.Canny(gray, 50, 150)

lines = cv2.HoughLinesP(edges, 1, np.pi/180, threshold=100, minLineLength=50, maxLineGap=10)

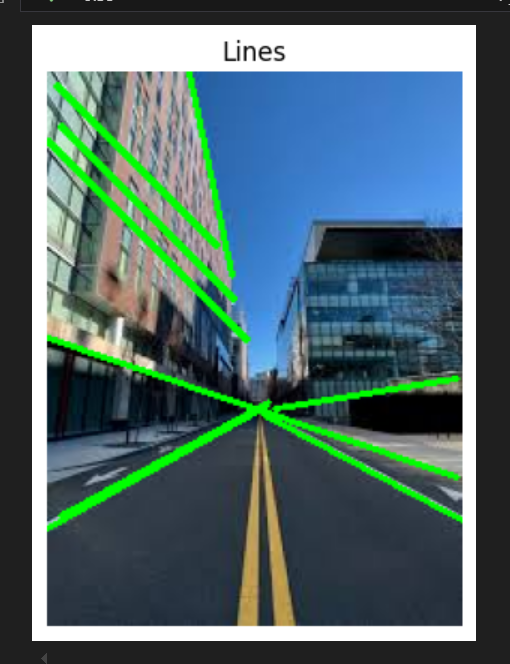
if lines is not None:

for points in lines:

x1, y1, x2, y2 = points[0]

cv2.line(img1, (x1, y1), (x2, y2), (0, 255, 0), 2)

show(img1, 'Lines')



* **What It Does:**
  + Loads an image and converts it to grayscale.
  + Applies the Canny edge detector to highlight the edges.
  + Uses the probabilistic Hough Transform (HoughLinesP) to detect lines within the edge image.
  + Draws the detected lines in green on the original image.
* **Why It’s Important:**  
  Line detection is fundamental in many applications, from lane detection in self-driving cars to identifying structural elements in architectural images.

### 3. Circle Detection with Adjusted Parameters

img2 = cv2.imread('circle.webp')

if img2 is not None:

gray = cv2.cvtColor(img2, cv2.COLOR\_BGR2GRAY)

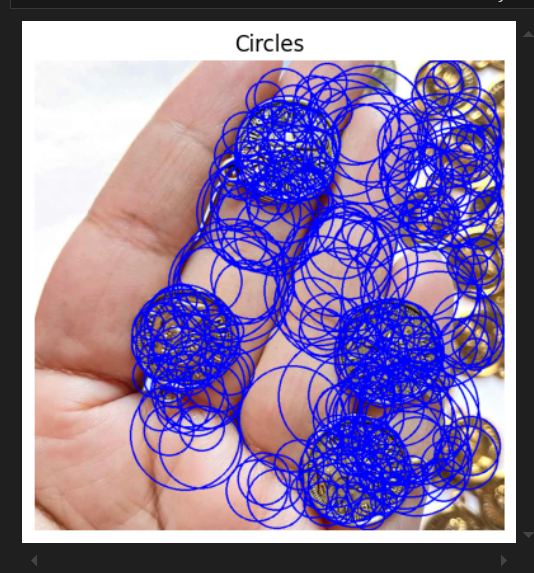
circles = cv2.HoughCircles(gray, cv2.HOUGH\_GRADIENT, 1, 20, param1=50, param2=60, minRadius=10, maxRadius=100)

if circles is not None:

for c in np.uint16(np.around(circles))[0, :]:

cv2.circle(img2, (c[0], c[1]), c[2], (255, 0, 0), 2)

show(img2, 'Circles')



* **What It Does:**
  + Loads an image and converts it to grayscale.
  + Uses the Hough Circle Transform (HoughCircles) to detect circles.
  + Draws the detected circles with a blue outline.
* **Why It’s Important:**  
  Circle detection can be used in a range of applications such as detecting round objects, coins, or even biological cells.

### 4. Shi-Tomasi Corner Detection

img3 = cv2.imread('corner.webp')

if img3 is not None:

gray = cv2.cvtColor(img3, cv2.COLOR\_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(gray, 100, 0.01, 10)

if corners is not None:

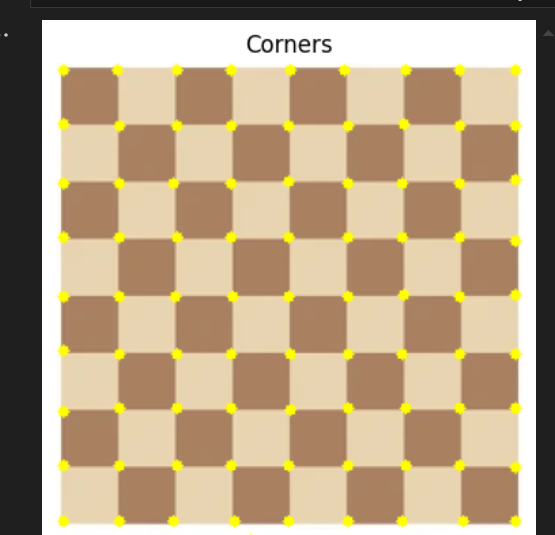
corners = np.int32(corners)

for p in corners:

x, y = p.ravel()

cv2.circle(img3, (x, y), 3, (0, 255, 255), -1)

show(img3, 'Corners')



* **What It Does:**
  + Loads an image and converts it to grayscale.
  + Applies the Shi-Tomasi method to detect up to 100 strong corners.
  + Marks each detected corner with a small circle in a bright yellowish color.
* **Why It’s Important:**  
  Corners are key features in many image analysis tasks, such as object tracking and image stitching. They provide important points for aligning or recognizing structures within images.

### 5. Harris Corner Detection

img4 = cv2.imread('harris.jpeg')

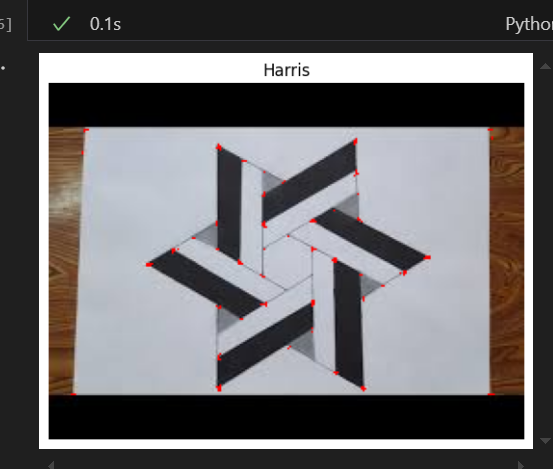
if img4 is not None:

gray = cv2.cvtColor(img4, cv2.COLOR\_BGR2GRAY)

harris = cv2.cornerHarris(np.float32(gray), 2, 3, 0.04)

img4[harris > 0.01 \* harris.max()] = [0, 0, 255]

show(img4, 'Harris')



* **What It Does:**
  + Converts the image to grayscale.
  + Computes the Harris corner response to detect corners.
  + Highlights the detected corners in red by setting those pixel values.
* **Why It’s Important:**  
  Harris corner detection is another popular method for finding corner-like features, which is particularly useful in motion tracking and image matching.

### 6. Document Detection

img5 = cv2.imread('document.png')

if img5 is not None:

gray = cv2.cvtColor(img5, cv2.COLOR\_BGR2GRAY)

edges = cv2.Canny(gray, 50, 150)

contours, \_ = cv2.findContours(edges, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

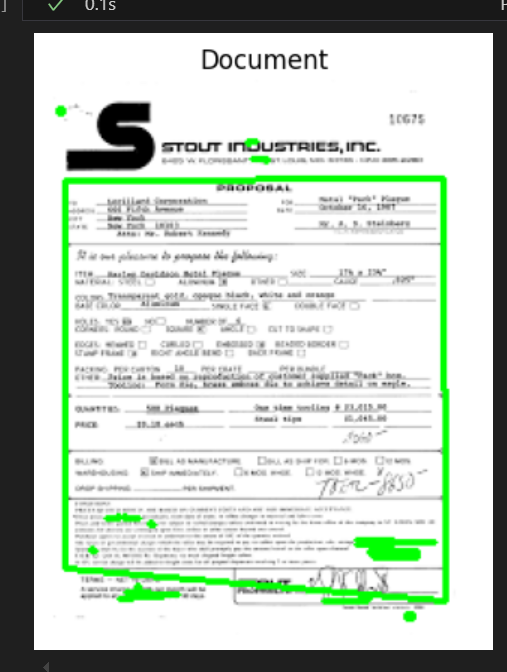
for cnt in contours:

approx = cv2.approxPolyDP(cnt, 0.02 \* cv2.arcLength(cnt, True), True)

if len(approx) == 4:

cv2.drawContours(img5, [approx], -1, (0, 255, 0), 2)

show(img5, 'Document')



* **What It Does:**
  + Loads an image, converts it to grayscale, and detects edges.
  + Finds contours and approximates them to identify shapes with four corners.
  + Highlights these quadrilateral contours in green, which are likely candidates for documents.
* **Why It’s Important:**  
  Document detection is valuable for scanning and digitizing papers, automating the cropping of documents from images.

### 7. Smile Detection

img6 = cv2.imread('smile.jpeg')

if img6 is not None:

gray = cv2.cvtColor(img6, cv2.COLOR\_BGR2GRAY)

faces = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade\_frontalface\_default.xml')

smiles = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade\_smile.xml')

detected = faces.detectMultiScale(gray, 1.3, 5)

for (x, y, w, h) in detected:

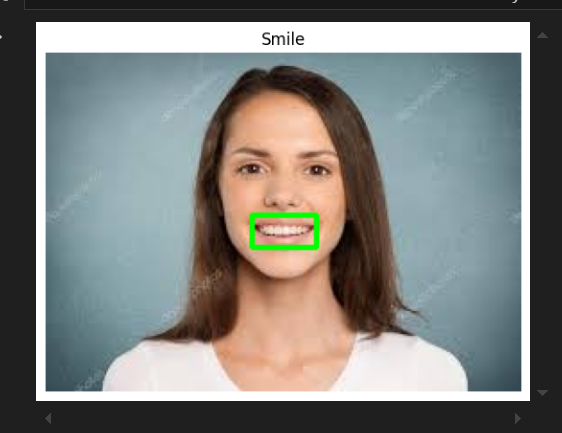
face\_roi = gray[y:y+h, x:x+w]

smiles\_detected = smiles.detectMultiScale(face\_roi, 1.8, 20)

for (sx, sy, sw, sh) in smiles\_detected:

cv2.rectangle(img6, (x+sx, y+sy), (x+sx+sw, y+sy+sh), (0, 255, 0), 2)

show(img6, 'Smile')



* **What It Does:**
  + Reads an image and converts it to grayscale.
  + Uses Haar Cascade classifiers to detect faces and, within each face, detect smiles.
  + Draws green rectangles around detected smiles.
* **Why It’s Important:**  
  Smile detection is often used in interactive applications like photo booths and security systems. It’s also a fun example of how machine learning models can be applied to everyday tasks.

## Conclusion

This script is a rich demonstration of OpenCV’s feature detection capabilities. From detecting lines and circles to recognizing corners, documents, and even smiles, each block of code shows how computer vision can extract specific features from images. Whether you're just starting with image processing or looking to expand your toolkit, these techniques provide a strong foundation for a wide range of practical applications. Enjoy experimenting with these methods and exploring further possibilities in the exciting field of computer vision!

# 2.4

# Report on OpenCV Drawing Functions

## Introduction

This script demonstrates how to create a blank canvas and draw multiple shapes using OpenCV. By leveraging functions such as lines, arrows, ellipses, circles, rectangles, polylines, and text, you can design graphics and overlays for various applications. Whether you’re working on image annotation, user interface design, or artistic visualizations, these techniques are a great foundation.

## Tools and Technologies

* **OpenCV (cv2):** A versatile library for computer vision tasks. Here, it’s used to draw geometric shapes and text on an image.
* **NumPy:** Provides support for array manipulations. In this case, it’s used to create a blank canvas and process coordinate data.
* **Matplotlib:** Utilized to display the resulting image in a user-friendly manner after converting color spaces.

## Code Block Breakdown

### 1. Importing Libraries and Setting Up the Canvas

import cv2

import numpy as np

import matplotlib.pyplot as plt

canvas = np.zeros((500, 500, 3), dtype=np.uint8) + 255

* **What it Does:**
  + Imports OpenCV, NumPy, and Matplotlib.
  + Creates a 500x500 pixel canvas with three color channels (BGR). The canvas is initially filled with zeros (black), and adding 255 sets it to white.
* **Why it’s Important:**  
  This sets the stage for drawing. A white canvas provides a clean background where all drawn elements stand out clearly.

### 2. Drawing a Blue Line

cv2.line(canvas, (50, 50), (450, 50), (255, 0, 0), 2)

* **What it Does:**  
  Draws a blue line (color: BGR (255, 0, 0)) from the point (50, 50) to (450, 50) with a thickness of 2 pixels.
* **Why it’s Important:**  
  Lines are a fundamental element in drawing and can be used for borders, separators, or design elements.

### 3. Drawing an Arrowed Line in Green

cv2.arrowedLine(canvas, (50, 100), (450, 100), (0, 255, 0), 2)

* **What it Does:**  
  Creates an arrowed line in green from (50, 100) to (450, 100). The arrowhead indicates direction, adding a dynamic element to the drawing.
* **Why it’s Important:**  
  Arrowed lines are particularly useful for indicating movement, direction, or pointing out areas of interest in images.

### 4. Drawing an Ellipse in Red

cv2.ellipse(canvas, (250, 250), (150, 100), 0, 0, 360, (0, 0, 255), 2)

* **What it Does:**  
  Draws an ellipse centered at (250, 250) with a horizontal axis of 150 pixels and a vertical axis of 100 pixels. The ellipse is drawn in red with a thickness of 2 pixels.
* **Why it’s Important:**  
  Ellipses can be used to highlight or enclose areas, and they are common in both artistic designs and technical illustrations.

### 5. Drawing a Filled Circle in Cyan

cv2.circle(canvas, (250, 250), 50, (255, 255, 0), -1)

* **What it Does:**  
  Places a filled circle (the fill is indicated by the thickness -1) at (250, 250) with a radius of 50 pixels. The color used is cyan (BGR: (255, 255, 0)).
* **Why it’s Important:**  
  Filled shapes like circles can be used as markers, buttons, or decorative elements in a design.

### 6. Drawing a Purple Rectangle

cv2.rectangle(canvas, (50, 400), (450, 450), (128, 0, 128), 3)

* **What it Does:**  
  Draws a rectangle with its top-left corner at (50, 400) and bottom-right corner at (450, 450). The rectangle’s border is purple (BGR: (128, 0, 128)) with a thickness of 3 pixels.
* **Why it’s Important:**  
  Rectangles are essential for creating structured layouts, bounding boxes, or highlighting specific regions.

### 7. Adding Text to the Canvas

cv2.putText(canvas, 'OpenCV Drawing', (100, 480), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 0), 2)

* **What it Does:**  
  Renders the text “OpenCV Drawing” at position (100, 480) using the Hershey Simplex font. The text is black and has a thickness of 2 pixels.
* **Why it’s Important:**  
  Text can provide labels, annotations, or decorative flair. It makes the graphic more informative and visually appealing.

### 8. Drawing a Closed Polygon (Triangle) and Its Centroid

pts = np.array([[100, 300], [400, 300], [250, 100]], np.int32).reshape((-1, 1, 2))

cv2.polylines(canvas, [pts], True, (0, 255, 255), 2)

centroid = np.mean(pts, axis=0).astype(int).flatten()

cv2.circle(canvas, (centroid[0], centroid[1]), 5, (255, 0, 0), -1)

* **What it Does:**
  + Creates an array of points that define a triangle.
  + Draws the triangle with yellow lines (BGR: (0, 255, 255)), and the True parameter ensures that the shape is closed.
  + Calculates the centroid (average position) of the triangle’s vertices.
  + Draws a small blue filled circle at the centroid.
* **Why it’s Important:**  
  Drawing polygons and computing their centroids are common in computer vision tasks like shape analysis, object recognition, and geometric transformations.

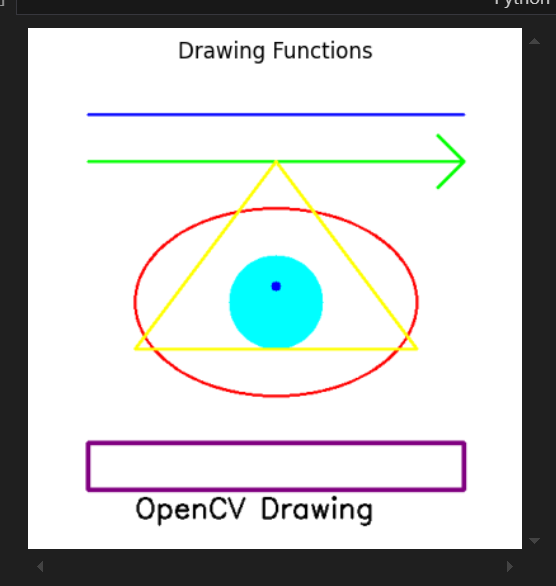
### 9. Displaying the Final Canvas

plt.imshow(cv2.cvtColor(canvas, cv2.COLOR\_BGR2RGB))

plt.title('Drawing Functions')

plt.axis('off')

plt.show()



* **What it Does:**
  + Converts the canvas from BGR (OpenCV’s format) to RGB (Matplotlib’s format) for accurate color display.
  + Uses Matplotlib to show the final image with the title “Drawing Functions” and without axis ticks.
* **Why it’s Important:**  
  This final step ensures that all your drawn shapes and text are displayed correctly, letting you see the complete, polished result of your drawing operations.

## Conclusion

This script is an excellent example of how to use OpenCV for drawing and annotating images. From creating a blank canvas to drawing lines, shapes, and text, each operation is a building block for more advanced graphics and computer vision applications. By understanding and experimenting with these drawing functions, you can create visually compelling designs and practical interfaces, making your projects more interactive and informative. Enjoy exploring and expanding these techniques in your creative endeavors!

# 3.1

# Report on Video Playback Using OpenCV

## Introduction

This Python script demonstrates how to read and display a video using OpenCV. It opens a specified video file, processes each frame, and shows it in a window to create a smooth playback experience. The script includes error checking to ensure the video file opens correctly and a user-interaction feature that lets you quit the playback by pressing 'q'. This approach is widely used in video analysis, editing, and computer vision applications.

## Code Block Breakdown

### 1. Importing OpenCV and Setting the Video Path

import cv2

video\_path = r"C:\Users\Admin\Videos\Screen Recordings\Screen Recording 2024-12-14 232909.mp4"

* **What It Does:**
  + Imports the OpenCV library (cv2) for video processing.
  + Defines video\_path, which holds the path to the video file you want to play.
* **Why It’s Important:**  
  This setup is essential to access the necessary libraries and specify the video file for subsequent processing.

### 2. Opening the Video File

cap = cv2.VideoCapture(video\_path)

if not cap.isOpened():

print("Can't open video.")

exit()

* **What It Does:**
  + Attempts to open the video file using cv2.VideoCapture.
  + Checks if the video file opened successfully with cap.isOpened().
  + If the video cannot be opened (due to an incorrect path or file issue), it prints an error message and terminates the script.
* **Why It’s Important:**  
  Verifying that the video file is accessible ensures that the program doesn’t proceed with an invalid file, preventing runtime errors.

### 3. Reading and Displaying Video Frames

while cap.isOpened():

ret, frame = cap.read()

if not ret:

break

cv2.imshow("Video", frame)

if cv2.waitKey(25) == ord('q'):

break

* **What It Does:**
  + Enters a loop that runs as long as the video file remains open.
  + Uses cap.read() to read each frame. ret is a Boolean that indicates if the frame was successfully read, and frame contains the frame data.
  + If reading a frame fails (e.g., end of the video), the loop breaks.
  + Displays the current frame in a window titled "Video" using cv2.imshow.
  + Waits for 25 milliseconds between frames with cv2.waitKey(25). If the 'q' key is pressed, the loop exits.
* **Why It’s Important:**  
  This loop is at the heart of the video playback, creating a real-time display of each frame. The wait time ensures smooth playback, and the ability to quit early provides user control.

### 4. Releasing Resources and Closing Windows

cap.release()

cv2.destroyAllWindows()

* **What It Does:**
  + Releases the video capture object with cap.release(), freeing up system resources.
  + Closes all OpenCV windows using cv2.destroyAllWindows().
* **Why It’s Important:**  
  Proper cleanup is crucial for resource management and ensures that your application does not leave open windows or locked files after execution.

## Conclusion

This script offers a clear demonstration of how to use OpenCV for video playback. By reading and displaying each frame in a loop, it simulates a basic video player, complete with error handling and an exit option using the 'q' key. This approach is practical not only for simple video display tasks but also forms the basis for more complex video analysis and computer vision applications. Whether you're developing a project or just exploring video processing, this code provides a solid foundation for your learning and experimentation. Enjoy coding and exploring the world of video processing with OpenCV!

# ****3.2****

# ****Generating and Extracting Video Frames Using OpenCV****

## ****Introduction****

In this project, we use OpenCV to create a video from a sequence of images and then extract frames from a video file. This technique is widely used in animation, time-lapse creation, and frame-by-frame video analysis.

* **Part 1:** Convert a series of images into a video.
* **Part 2:** Extract frames from a video file and save them as images.

Both parts use OpenCV’s built-in functions to handle media files efficiently.

## ****Part 1: Creating a Video from Images****

### ****1. Importing Required Libraries & Setting Up the Folder****

import cv2

import os

* **What It Does:**
  + Imports OpenCV (cv2) for image and video processing.
  + Imports os to manage file paths and directories.

### ****2. Selecting Images for the Video****

folder = r"C:\Users\Admin\Pictures\new wallpapers"

images = sorted(os.listdir(folder))

images = [img for img in images if img.endswith(('.jpg', '.png'))]

* **What It Does:**
  + Specifies the folder containing images.
  + Lists and sorts the files in the folder.
  + Filters out only images with .jpg and .png extensions.
* **Why It’s Important:**  
  This ensures only valid images are used for the video creation, preventing file format errors.

### ****3. Setting Up Video Writer****

frame = cv2.imread(f"{folder}/{images[0]}")

h, w, \_ = frame.shape

video = cv2.VideoWriter("output.mp4", cv2.VideoWriter\_fourcc(\*'mp4v'), 10, (w, h))

* **What It Does:**
  + Reads the first image to determine the video’s height and width.
  + Creates a video writer object (cv2.VideoWriter) to store the generated video.
  + Uses cv2.VideoWriter\_fourcc(\*'mp4v') to set the video format (MP4).
  + Sets the frame rate to **10 FPS**.
* **Why It’s Important:**  
  Ensures the video dimensions match the images and creates a smooth playback.

### ****4. Writing Images into Video****

for img in images:

frame = cv2.imread(f"{folder}/{img}")

video.write(frame)

video.release()

print("Video done")

* **What It Does:**
  + Iterates through each image.
  + Reads and writes each frame into the video file.
  + Releases the video file when done.
* **Why It’s Important:**
  + Converts the sequence of images into a playable video file.
  + The release() function prevents file corruption.

## ****Part 2: Extracting Frames from a Video****

### ****1. Importing Libraries & Defining Output Directory****

import cv2

import os

video\_path = r"C:\Users\Admin\Desktop\lab 5 PAI\output.mp4"

output\_folder = "data"

os.makedirs(output\_folder, exist\_ok=True)

* **What It Does:**
  + Specifies the video file path.
  + Defines an output directory (data) where extracted frames will be stored.
  + Uses os.makedirs(output\_folder, exist\_ok=True) to create the directory if it doesn’t exist.
* **Why It’s Important:**
  + Ensures a proper destination folder for saving frames.
  + Prevents errors if the folder already exists.

### ****2. Reading the Video & Extracting Frames****

cam = cv2.VideoCapture(video\_path)

currentframe = 0

while cam.isOpened():

ret, frame = cam.read()

if not ret:

break

cv2.imwrite(f"{output\_folder}/frame\_{currentframe}.jpg", frame)

print(f"Extracting frame {currentframe}")

currentframe += 1

cam.release()

cv2.destroyAllWindows()

print(f"✅ {currentframe} frames extracted.")

* **What It Does:**
  + Opens the video using cv2.VideoCapture().
  + Reads the video frame-by-frame.
  + Saves each frame as an image (frame\_0.jpg, frame\_1.jpg, etc.).
  + Prints progress to track frame extraction.
  + Releases resources and closes any OpenCV windows after extraction.
* **Why It’s Important:**
  + Converts a video into a set of images for analysis or editing.
  + The frame-by-frame extraction is useful in applications like object detection, motion tracking, and training datasets for AI models.

## ****Conclusion****

This project showcases how OpenCV can handle both video generation and frame extraction.

1. **We converted a sequence of images into a smooth video** by ensuring proper frame dimensions and sorting images correctly.
2. **We extracted frames from a video** and saved them as individual images, a key technique for video processing tasks.

By understanding these two fundamental operations, you can now explore advanced video processing applications such as:

* **Time-lapse creation** (from periodic images).
* **Slow-motion analysis** (by extracting and analyzing frames).
* **Machine learning training datasets** (using extracted video frames).