# **Program 1: Play with Colors - Image Segmentation**

## **1. Introduction**

In this program, we perform **image segmentation** to extract specific colors (green and red) from an image using **color-based thresholding** techniques in the **HSV (Hue, Saturation, Value)** color space. The goal is to isolate regions of interest (ROI) where the **green** and **red** colors are prominent, and display these regions separately on **white backgrounds** for better visualization.

## **2. Problem Solving Approach**

### **2.1 Image Loading and Preprocessing**

The program begins by loading the input image using OpenCV's cv2.imread() function. If the provided path is incorrect or the image cannot be loaded, an error message will be displayed. The image is resized to **500x500 pixels** to maintain consistency in processing for different image sizes.

### **2.2 Color Space Conversion**

Once the image is loaded, it is converted from the **BGR (Blue, Green, Red)** color space to the **HSV (Hue, Saturation, Value)** color space using OpenCV’s cv2.cvtColor() function. The **HSV color space** is more effective for color segmentation because it decouples the intensity (brightness) of the image from the color, allowing easier isolation of specific colors such as green and red.

### **2.3 Green Color Extraction (Output 1)**

A mask is created using the function cv2.inRange(), which checks for pixels within the specified **HSV range for green**. The green regions of the image are then extracted, while all other regions are turned **white** using NumPy array manipulation.

The HSV range used for **green** is:

* **Lower Bound for Green**: [40, 40, 40]
* **Upper Bound for Green**: [80, 255, 255]

This range isolates the pixels corresponding to green in the image.

### **2.4 Red Color Extraction (Output 2)**

Since **red** spans across two ranges in the Hue spectrum (0-10 and 170-180 degrees), two separate ranges are defined for red. The function cv2.inRange() is used to create two masks for these two red ranges. These masks are combined using the **bitwise OR** operation to capture all red areas.

The HSV ranges used for **red** are:

* **First Range for Red**:
  + **Lower Bound for Red**: [0, 50, 50]
  + **Upper Bound for Red**: [10, 255, 255]
* **Second Range for Red**:
  + **Lower Bound for Red**: [170, 50, 50]
  + **Upper Bound for Red**: [180, 255, 255]

The red regions are extracted, and the rest of the image is set to **white**.

### **2.5 Combined Extraction (Output 3)**

In the third output, we combine the green and red masks using the **bitwise AND** operation. This ensures that only the regions where both green and red overlap are extracted. This output highlights both green and red areas on a **white background**.

### **2.6 Final Result Display**

After segmentation, the processed images are saved using OpenCV’s cv2.imwrite() function. The results are also displayed using cv2.imshow() to visualize how well the green, red, and combined extractions have been performed.

## **3. Code Explanation**

The following Python code implements the segmentation process described above:

import cv2

import numpy as np

# Load the original input image

image = cv2.imread('peppers.jpg') # Replace with your image path

if image is None:

print("Error: Image not found!")

exit()

image = cv2.resize(image, (500, 500)) # Resize for consistency (optional)

# Convert to HSV color space for better color segmentation

hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)

# Output 1: Extract green areas with the original color on white background

lower\_green = np.array([40, 40, 40]) # Lower HSV bound for green

upper\_green = np.array([80, 255, 255]) # Upper HSV bound for green

green\_mask = cv2.inRange(hsv, lower\_green, upper\_green)

# Create green output: Keep green areas, make the rest white

green\_output = image.copy()

green\_output[np.where(green\_mask == 0)] = [255, 255, 255] # Set background to white

# Output 2: Extract red areas with the original color on white background

lower\_red1 = np.array([0, 50, 50]) # Lower HSV bound for red (first range)

upper\_red1 = np.array([10, 255, 255]) # Upper HSV bound for red (first range)

lower\_red2 = np.array([170, 50, 50]) # Lower HSV bound for red (second range)

upper\_red2 = np.array([180, 255, 255]) # Upper HSV bound for red (second range)

red\_mask1 = cv2.inRange(hsv, lower\_red1, upper\_red1)

red\_mask2 = cv2.inRange(hsv, lower\_red2, upper\_red2)

red\_mask = cv2.bitwise\_or(red\_mask1, red\_mask2)

# Create red output: Keep red areas, make the rest white

red\_output = image.copy()

red\_output[np.where(red\_mask == 0)] = [255, 255, 255] # Set background to white

# Output 3: Highlight all segmented areas (green + red) on white background

combined\_mask = cv2.bitwise\_and(green\_mask, red\_mask) # Use AND instead of OR

combined\_output = image.copy()

combined\_output[np.where(combined\_mask == 0)] = [255, 255, 255] # Set background to white

# Save and display the outputs

cv2.imwrite('output1\_green.jpg', green\_output)

cv2.imwrite('output2\_red.jpg', red\_output)

cv2.imwrite('output3\_combined.jpg', combined\_output)

cv2.imshow('Original Image', image)

cv2.imshow('Output 1 - Green ROI (White Background)', green\_output)

cv2.imshow('Output 2 - Red ROI (White Background)', red\_output)

cv2.imshow('Output 3 - Combined ROI (White Background)', combined\_output)

cv2.waitKey(0)

cv2.destroyAllWindows()

## **4. Libraries Used**

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

* **OpenCV** is a powerful library for computer vision tasks. It is used here for various image processing operations like loading the image, converting color spaces, creating masks, and displaying results.
* Functions used:
  + cv2.imread(): Load an image.
  + cv2.cvtColor(): Convert between different color spaces (BGR to HSV).
  + cv2.inRange(): Create a binary mask for a specified color range.
  + cv2.bitwise\_or(), cv2.bitwise\_and(): Perform bitwise operations on the masks.
  + cv2.imshow(), cv2.imwrite(): Display and save the processed images.

### **4.2 NumPy (np)**

* **NumPy** is used for array manipulations, such as setting pixel values to create a white background in the extracted regions.
* Functions used:
  + np.where(): Replace pixel values where a condition is met.

## **5. Results and Discussion**

### **5.1 Output 1 - Green Color Segmentation**

The first output displays only the **green regions** of the image. These regions are highlighted in their original colors, while all other areas are turned **white**.

### **5.2 Output 2 - Red Color Segmentation**

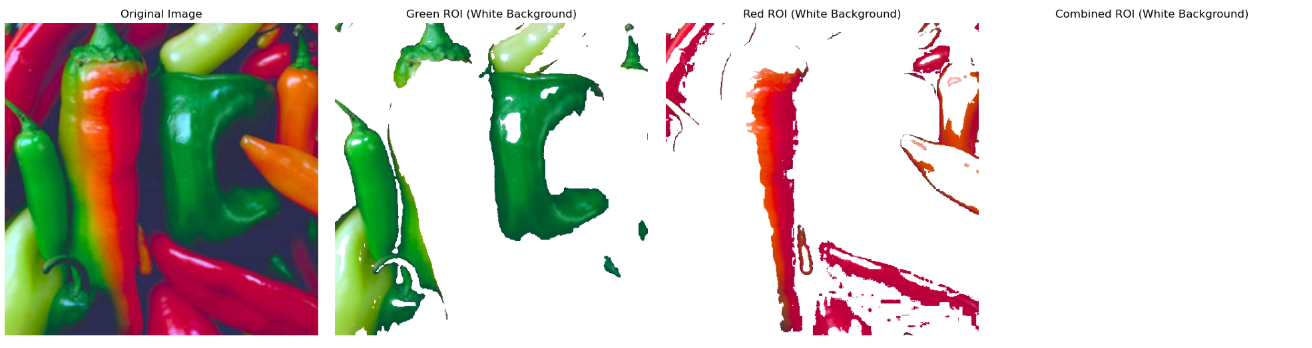
The second output displays only the **red regions** of the image. These areas are highlighted in red, with all other areas turned **white**.

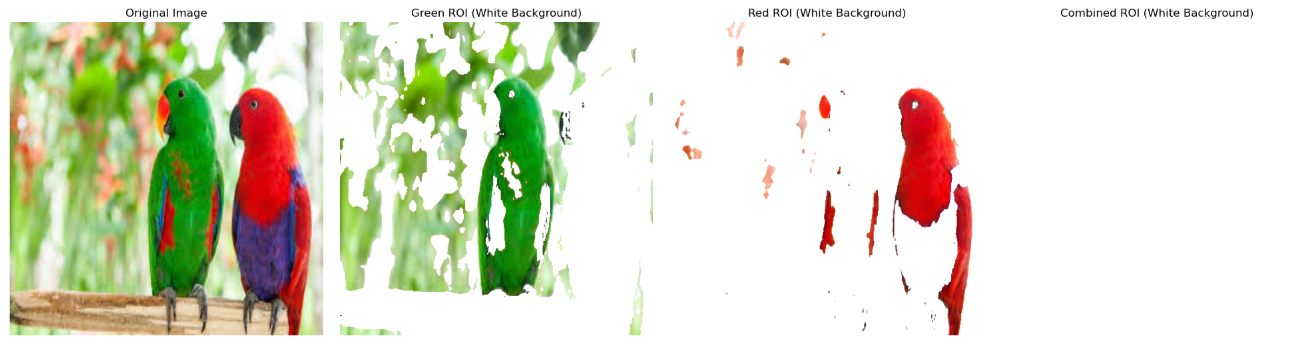
### **5.3 Output 3 - Combined Green and Red Segmentation**

The third output combines the previous two extractions, highlighting both **green and red areas**. This combined output is also displayed with a white background for clarity.

**6. Conclusion**

This program demonstrates how color-based segmentation using the **HSV color space** can be applied to extract specific regions from an image. The method works effectively for simple images where the green and red regions are distinct. However, it may face challenges in complex images with overlapping colors or poor lighting.







# **Program 3: Color Dominance and Background Conversion**

This document provides a detailed explanation of a program that processes an image to highlight a dominant color in the **Green**, **Blue**, and **Red** channels, while converting the rest of the image to grayscale. The result is a visually striking effect, where each color dominance is isolated, and the remaining parts of the image are converted into grayscale.

## **1. Overview**

The program manipulates the color channels of an image to emphasize a specific color—**Green**, **Blue**, or **Red**—and converts the other colors into grayscale. By using color dominance, we create images where only one color stands out, while others fade into the background. This technique can be particularly useful for visual analysis or enhancing specific features in an image.

## **2. Steps and Code Explanation**

### **Step 1: Load and Resize the Image**

The program starts by loading the image (grid.png) from the local directory. The image is resized to a smaller resolution for better visualization.

# Load the original image

image = cv2.imread('grid.png')

# Resize the image (optional)

image = cv2.resize(image, (1000, 500))

* **cv2.imread('grid.png')**: Loads the image named grid.png into the program.
* **cv2.resize(image, (1000, 500))**: Resizes the image to a width of 1000 pixels and height of 500 pixels for better display.

### **Step 2: Convert the Image to Grayscale**

In order to create a neutral background, we convert the image to grayscale. This grayscale image will serve as the backdrop for non-dominant colors, allowing us to keep the dominant color in full intensity while fading out others.

# Convert the image to grayscale for background (this will help with gray shading)

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

* **cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)**: Converts the image from BGR (Blue-Green-Red) color space to grayscale. This results in a single intensity channel that represents the brightness of each pixel.

### **Step 3: Separate the Color Channels**

The image is split into its individual color channels: **Blue**, **Green**, and **Red**. These separate channels will be used to evaluate the dominance of each color.

# Separate the color channels

blue, green, red = cv2.split(image)

* **cv2.split(image)**: Splits the original image into three separate channels: Blue (blue), Green (green), and Red (red).

### **Step 4: Output 1 - Green Dominant**

The first output isolates areas where **Green** is dominant over both **Red** and **Blue**. These regions are retained in full color, while the rest of the image is converted to grayscale.

# Output 1: Green Dominant - Retain green, others in grayscale

output1 = np.zeros\_like(image) # Create a black image with the same shape

mask = (green > red + threshold) & (green > blue + threshold) # Green is dominant

output1[:, :, 1] = np.where(mask, green, 0) # Retain green where it's dominant

output1 = cv2.add(output1, cv2.merge([gray, gray, gray])) # Add grayscale to non-green regions

* **mask = (green > red + threshold) & (green > blue + threshold)**: Identifies pixels where **Green** is more dominant than **Red** and **Blue**. The **threshold** parameter can be adjusted to make the green dominance stronger or weaker.
* **output1[:, :, 1] = np.where(mask, green, 0)**: Retains **Green** values in areas where **Green** is dominant, and sets other regions to 0 (black).
* **cv2.add(output1, cv2.merge([gray, gray, gray]))**: Adds grayscale values to non-dominant regions by merging the grayscale image with the color channels.

### **Step 5: Output 2 - Blue Dominant**

The second output isolates the **Blue** channel, retaining blue regions while converting the rest to grayscale.

# Output 2: Blue Dominant - Retain blue, others in grayscale

output2 = np.zeros\_like(image) # Create a black image with the same shape

mask = (blue > red + threshold) & (blue > green + threshold) # Blue is dominant

output2[:, :, 0] = np.where(mask, blue, 0) # Retain blue where it's dominant

output2 = cv2.add(output2, cv2.merge([gray, gray, gray])) # Add grayscale to non-blue regions

* **mask = (blue > red + threshold) & (blue > green + threshold)**: Identifies pixels where **Blue** is dominant over both **Red** and **Green**.
* **output2[:, :, 0] = np.where(mask, blue, 0)**: Retains **Blue** values where **Blue** is dominant, and sets other areas to 0.
* **cv2.add(output2, cv2.merge([gray, gray, gray]))**: Adds grayscale to non-blue areas.

### **Step 6: Output 3 - Red Dominant**

The third output isolates the **Red** channel, keeping red areas and turning others into grayscale.

# Output 3: Red Dominant - Retain red, others in grayscale

output3 = np.zeros\_like(image) # Create a black image with the same shape

mask = (red > green + threshold) & (red > blue + threshold) # Red is dominant

output3[:, :, 2] = np.where(mask, red, 0) # Retain red where it's dominant

output3 = cv2.add(output3, cv2.merge([gray, gray, gray])) # Add grayscale to non-red regions

* **mask = (red > green + threshold) & (red > blue + threshold)**: Identifies pixels where **Red** is dominant over both **Green** and **Blue**.
* **output3[:, :, 2] = np.where(mask, red, 0)**: Retains **Red** values where **Red** is dominant, and sets other regions to 0.
* **cv2.add(output3, cv2.merge([gray, gray, gray]))**: Converts non-red areas to grayscale.

### **Step 7: Display the Results**

Finally, the program displays the original image alongside the three processed outputs (Green Dominant, Blue Dominant, Red Dominant) using Matplotlib. The results are displayed in a 2x2 grid layout.

# Plot results

outputs = [image, output1, output2, output3]

titles = ["Original", "Output 1 (Green Dominant)", "Output 2 (Blue Dominant)", "Output 3 (Red Dominant)"]

plt.figure(figsize=(10, 8))

for i in range(4):

plt.subplot(2, 2, i + 1)

plt.imshow(cv2.cvtColor(outputs[i], cv2.COLOR\_BGR2RGB)) # Convert to RGB for display

plt.title(titles[i])

plt.axis('off')

plt.tight\_layout()

plt.show()

* **outputs**: A list that holds the original image and the three color-dominant outputs.
* **plt.imshow(cv2.cvtColor(outputs[i], cv2.COLOR\_BGR2RGB))**: Converts the image from BGR (OpenCV format) to RGB (Matplotlib format) for correct display.
* **plt.title(titles[i])**: Sets the title for each subplot.
* **plt.axis('off')**: Hides the axes for a cleaner visual.

## **3. Summary of Outputs**

1. **Output 1 (Green Dominant)**: Focuses on the **Green** color, with other colors in grayscale. This is ideal for images that contain a lot of green areas (e.g., nature, forests).
2. **Output 2 (Blue Dominant)**: Highlights the **Blue** color, with other colors converted to grayscale. This is suitable for images with a lot of sky or water bodies.
3. **Output 3 (Red Dominant)**: Emphasizes the **Red** color while other regions are in grayscale. This is perfect for images with red objects, such as flowers or cars.

## **4. Key Libraries Used**

* **OpenCV (cv2)**: For image manipulation, including loading, splitting color channels, and applying transformations.
* **NumPy**: For creating masks and manipulating image arrays.
* **Matplotlib (plt)**: For displaying images in a grid layout for comparison.

By using these color dominance techniques, you can focus on specific features of an image based on color intensity, allowing for targeted image processing tasks.

## **5. For What Types of Images Does This Work?**

### **Images That Work Well**

* **Nature and Landscape Images**: Images with clear dominance of certain colors (e.g., green forests, blue skies, red flowers) work best with this method.
* **Colorful Objects**: Objects like cars, fruits, and

flowers where one color stands out can be effectively processed.

* **Colorful Grid or Patterned Images**: If the image has a high contrast between the primary colors, this technique will produce visually appealing outputs.

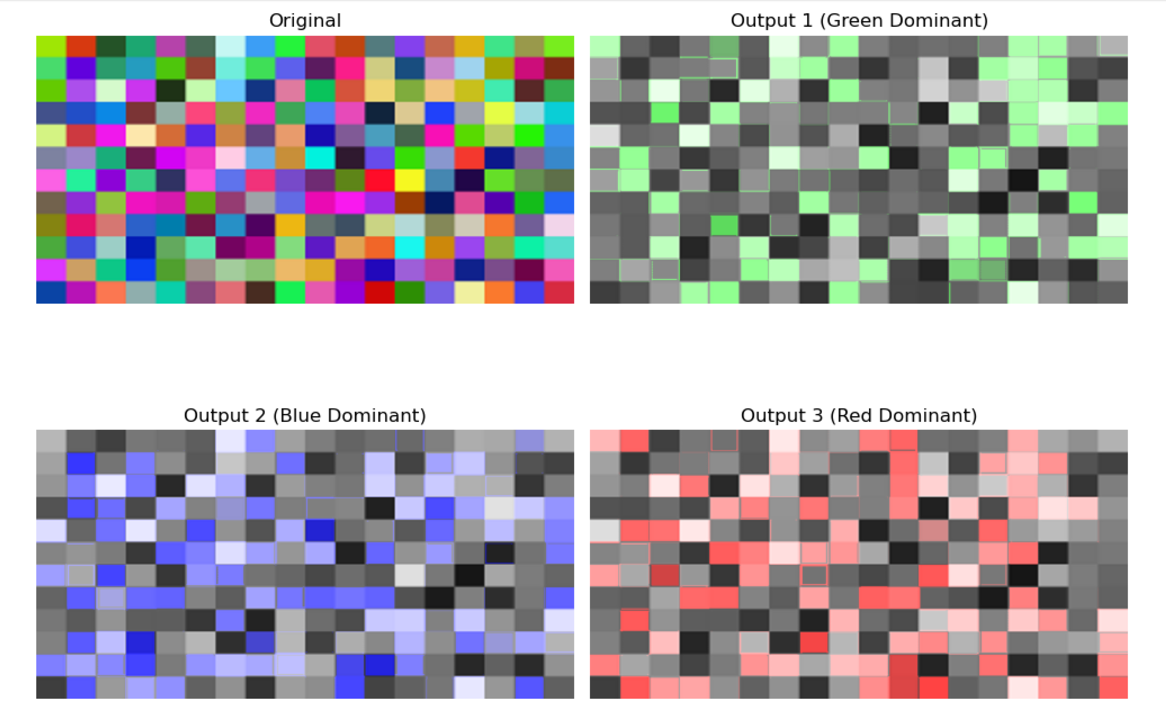
### **Images That May Not Work Well**

* **Monochrome or Black-and-White Images**: These will not show significant color dominance, and the grayscale regions will be redundant.
* **Images with Muted or Equal Color Presence**: If there is no clear dominant color (e.g., images with subtle gradients or balanced color tones), the results may be less striking or visually informative.
* **Low-Resolution Images**: The effect may not be as visually pleasing in low-resolution images due to pixelation.

## 

## **6. Conclusion**

This program allows for an effective visualization of color dominance in images by isolating one color and converting the rest into grayscale. It is useful for enhancing specific features based on their color characteristics.



### **Program 3 - Traffic Light Detection using OpenCV and Python**

#### **Introduction**

Traffic light detection is a crucial task in modern autonomous vehicles, traffic monitoring systems, and robotics. In this program, we utilize the power of OpenCV, an open-source computer vision library, to classify images based on the color of the traffic light. The main objective is to detect red, yellow, and green traffic lights and classify them accordingly to take specific actions:

* **Stop** (Red Light)
* **Wait** (Yellow Light)
* **Go** (Green Light)

This is accomplished by using color filtering and contour detection techniques on images of traffic lights.

#### **Technologies Used**

1. **Python** Python is the programming language employed for this task. It provides a rich set of libraries for computer vision, including OpenCV and NumPy, which are used in this program.
2. **OpenCV** OpenCV (Open Source Computer Vision Library) is a highly efficient and widely used library for computer vision tasks, such as image reading, manipulation, and color filtering. We use OpenCV to perform the following:  
   * Read images from disk
   * Process and manipulate images
   * Detect and classify the color of traffic lights
3. **NumPy** NumPy is used for numerical operations, such as defining color ranges in the HSV color space for filtering purposes. It helps to efficiently handle array-based operations.
4. **Matplotlib** Matplotlib is used for visualizing the results of the image processing. It helps in displaying the processed image with overlaid traffic light information (Stop, Wait, Go).
5. **HSV Color Space** The HSV (Hue, Saturation, Value) color space is particularly useful for detecting specific colors in images. In this program, we use the HSV color space to filter out red, yellow, and green lights in traffic light images.

#### **Approach**

The following steps outline the process of traffic light detection:

1. **Load and Preprocess the Image** The image is first loaded into memory using OpenCV’s cv2.imread() function. It is then resized to a standard size (300x300 pixels) to ensure consistent processing.
2. **Convert to HSV Color Space** To improve the accuracy of color detection, the image is converted from the default BGR (Blue, Green, Red) color space to the HSV color space using OpenCV's cv2.cvtColor() function. The HSV color space allows easier manipulation of color components, which is essential for detecting traffic light colors.
3. **Define Color Ranges for Red, Yellow, and Green** For detecting red, yellow, and green traffic lights, we define color ranges in the HSV space. These ranges are used to isolate pixels belonging to each color:  
   * **Red**: We use two ranges to capture the two different shades of red in the HSV color space.
   * **Yellow**: We define the range for detecting yellow light.
   * **Green**: We define the range for detecting green light.
4. **Create Masks for Each Color** Using OpenCV's cv2.inRange() function, we create masks that highlight the pixels within each defined color range. Each mask is a binary image where pixels that fall within the color range are white (1), and others are black (0).
5. **Find Contours for Each Mask** Contours are then detected in each of the color masks using cv2.findContours(). Contours are outlines or boundaries of the shapes detected in the binary image. If a contour is found for a given color, we can assume that the traffic light of that color is present.
6. **Classify the Traffic Light** Based on the number of contours found for each color mask, we classify the image into one of three categories:  
   * **Red Light (Stop)**: If red contours are detected.
   * **Yellow Light (Wait)**: If yellow contours are detected.
   * **Green Light (Go)**: If green contours are detected.
7. If no contours are detected, we assume that no traffic light is present in the image.
8. **Display the Detected Traffic Light** The final step involves visualizing the detection result. If a traffic light is detected, the program overlays the corresponding command ("Stop", "Wait", "Go") on the image. The processed image is then displayed using Matplotlib.

### **Code Implementation**

|  |
| --- |
| import cv2  import numpy as np  import matplotlib.pyplot as plt  def detect\_traffic\_light(image\_path):  # Load the input image  image = cv2.imread(image\_path)  if image is None:  print(f"Error: Image at {image\_path} not found.")  return None, None    # Resize for consistent processing  image = cv2.resize(image, (300, 300))    # Convert to HSV color space  hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)  # Define color ranges for red, yellow, and green lights  red\_lower1 = np.array([0, 120, 100])  red\_upper1 = np.array([10, 255, 255])  red\_lower2 = np.array([160, 120, 100])  red\_upper2 = np.array([180, 255, 255])  yellow\_lower = np.array([20, 120, 100])  yellow\_upper = np.array([30, 255, 255])  green\_lower = np.array([40, 120, 100])  green\_upper = np.array([70, 255, 255])  # Create masks for each color  red\_mask1 = cv2.inRange(hsv, red\_lower1, red\_upper1)  red\_mask2 = cv2.inRange(hsv, red\_lower2, red\_upper2)  red\_mask = cv2.bitwise\_or(red\_mask1, red\_mask2)  yellow\_mask = cv2.inRange(hsv, yellow\_lower, yellow\_upper)  green\_mask = cv2.inRange(hsv, green\_lower, green\_upper)  # Find contours for each mask  contours\_red, \_ = cv2.findContours(red\_mask, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)  contours\_yellow, \_ = cv2.findContours(yellow\_mask, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)  contours\_green, \_ = cv2.findContours(green\_mask, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)  # Detect the light color and determine the command  command = "No traffic light detected"  if len(contours\_red) > 0:  command = "Stop"  elif len(contours\_yellow) > 0:  command = "Wait"  elif len(contours\_green) > 0:  command = "Go"  return image, command  # List of image paths  image\_paths = ['red.png', 'yellow.jpg', 'green.jpg']  # Set up subplots: 3 rows, 2 columns  fig, axes = plt.subplots(3, 2, figsize=(12, 15))  for i, ax in enumerate(axes.flat):  if i < len(image\_paths):  image, command = detect\_traffic\_light(image\_paths[i])  if image is not None:  ax.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB))  ax.set\_title(f"Image {i + 1} - {command}")  ax.axis('off')  else:  ax.set\_title(f"Error: Image {i + 1}")  ax.axis('off')  plt.tight\_layout()  plt.show() |

### **Testing the Code**

The program is tested on several images containing red, yellow, and green traffic lights. The results are displayed in a grid with each image showing the corresponding command: Stop, Wait, or Go.

#### **Test Images:**

1. **red.png**: Contains a red traffic light.
2. **yellow.jpg**: Contains a yellow traffic light.
3. **green.jpg**: Contains a green traffic light.

#### **Expected Output:**

Images with detected traffic lights will show the correct label ("Stop", "Wait", or "Go"). If no traffic light is detected, the program will return "No traffic light detected".

### **For What Images Does It Work?**

The traffic light detection program works well for images that contain clearly visible traffic lights in the colors red, yellow, or green. Specifically, the program performs well under the following conditions:

* **Clear, well-lit images** where the traffic light is easily distinguishable.
* **Images with distinct color ranges** where the red, yellow, and green lights are visible and separated.
* **Images with minimal background clutter** to avoid confusion with other objects in the scene.
* **Close-up images of traffic lights** where the lights are centered and clear.

#### **Examples of suitable images:**

* A close-up image of a traffic light showing the red color clearly.
* A traffic light at an intersection with good lighting and contrast.

### **For What Kind of Images Does It Not Work?**

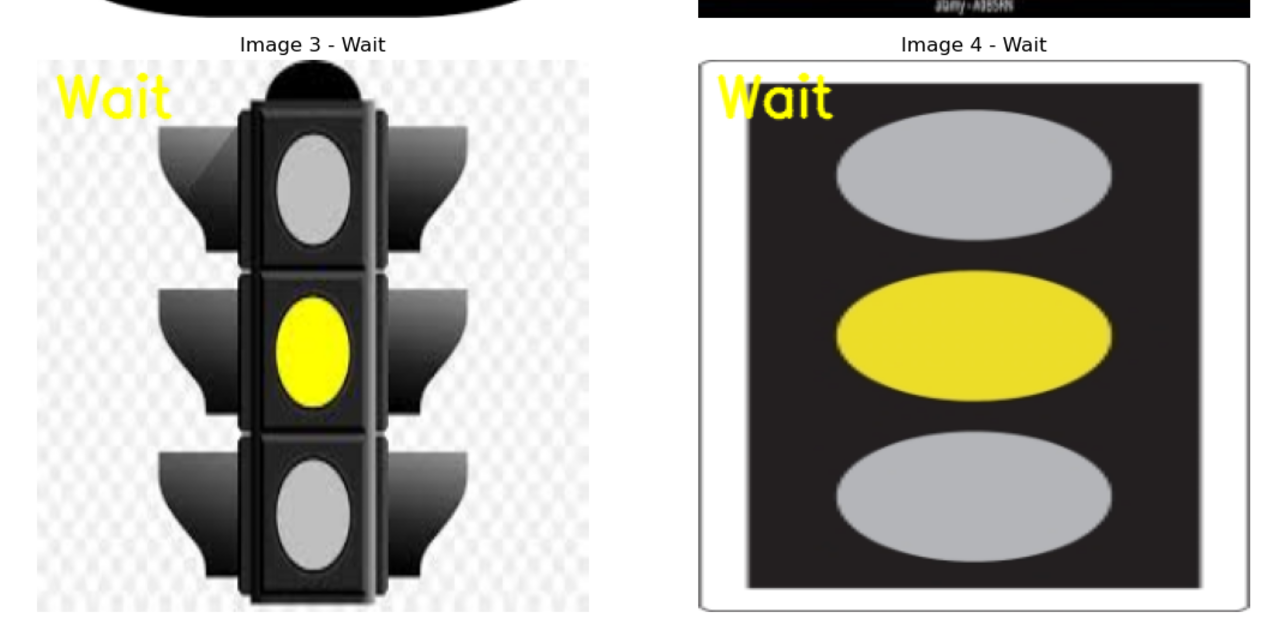
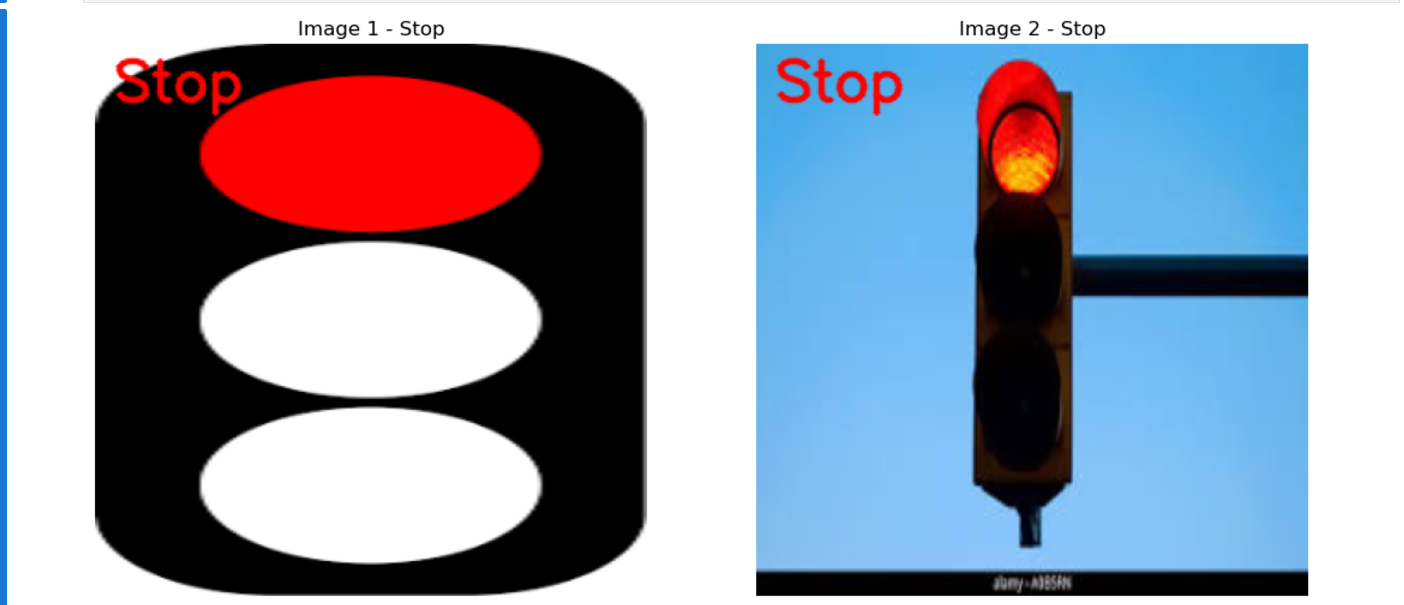
The program may not work effectively in the following scenarios:

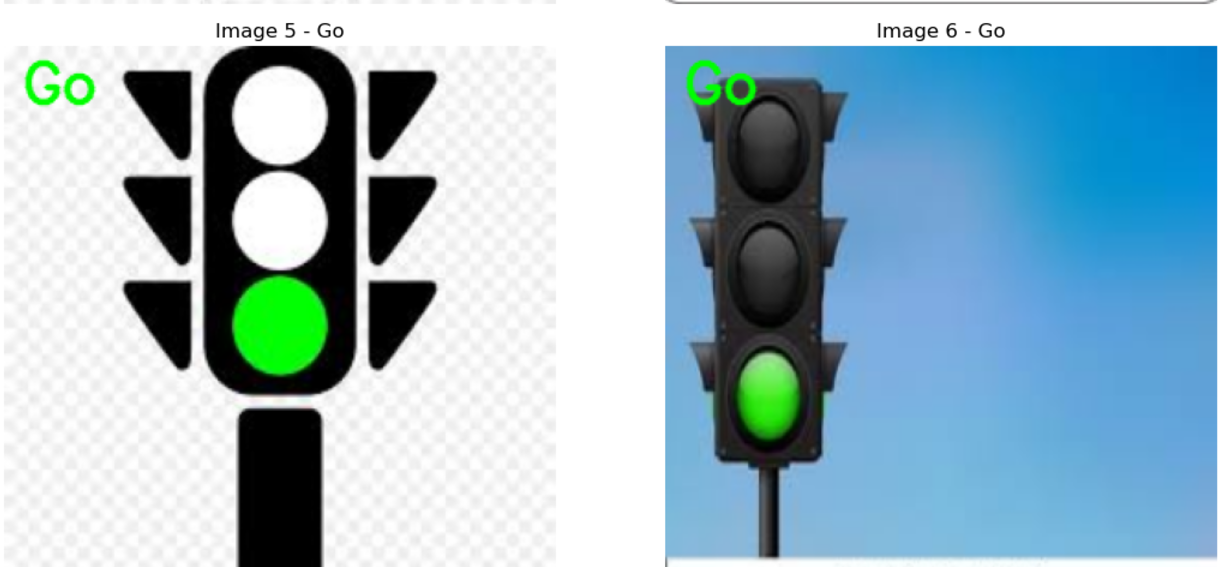
* **Low lighting or poor contrast**: If the traffic light is not well lit or the image is too dark, the colors may not be detected properly.
* **Images with overlapping lights**: If the traffic light is part of a multi-light system (e.g., red, yellow, and green lights appear in the same frame), the program may confuse the detection.
* **Images with significant background clutter**: If there are other objects with similar colors to the traffic light in the background, the program might mistakenly classify them.
* **Blurry or out-of-focus images**: If the traffic light is not sharp

or is blurry, contours may not be detected correctly.

#### **Examples of unsuitable images:**

* An image of a traffic light taken in low-light conditions.
* An image where the traffic light is partially obscured or out of focus.





# **Shape Coordinates Detection - Documentation**

## **1. Introduction**

This document provides a detailed explanation of the shape coordinates detection process, implemented using OpenCV and Python. The objective of the program is to detect the centroids (center points) of shapes present in an image. The centroids are calculated based on the contours of the shapes, and red circles are drawn to visually highlight these points.

## **2. Features**

* **Detecting Shape Coordinates**: The program calculates the center (centroid) of each shape detected in the image.
* **Visualizing Results**: Red circles are drawn on the image at the coordinates of the detected centroids, which can help with object tracking and shape recognition.
* **Input**: The program accepts an image as input, processes it to find contours, and calculates the centroids of detected shapes.

## **3. Requirements**

Before running the code, ensure you have the following libraries installed:

* **OpenCV**: For image processing.
* **NumPy**: For numerical operations.
* **Matplotlib**: For displaying the images in a grid.

To install these libraries, run the following command in your terminal or command prompt:

pip install opencv-python numpy matplotlib

## **4. Code Explanation**

### **4.1. Image Preprocessing**

The first step in the program is to convert the input image to grayscale. This is done using the cv2.cvtColor() function:

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

Next, a binary threshold is applied to the grayscale image using the cv2.threshold() function. This helps in segmenting the image into distinct shapes and contours.

\_, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH\_BINARY)

### **4.2. Contour Detection**

Contours are detected using the cv2.findContours() function. This function finds the boundaries of all shapes in the image.

contours, \_ = cv2.findContours(thresh, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

### **4.3. Centroid Calculation**

The centroids of the shapes are calculated using image moments. The moment is a weighted average of pixel intensities in the shape. The center of mass (centroid) can be calculated from the moments as follows:

moments = cv2.moments(cnt)

if moments['m00'] != 0:

cx = int(moments['m10'] / moments['m00']) # X coordinate

cy = int(moments['m01'] / moments['m00']) # Y coordinate

The centroid coordinates are stored in a list for later use.

### **4.4. Marking Centroids**

Red circles are drawn at the calculated centroids on the image using the cv2.circle() function:

cv2.circle(output\_image, (cx, cy), 5, (0, 0, 255), -1) # Draw centroid in red

### **4.5. Displaying the Image**

The original image and the processed image (with centroids) are displayed side-by-side using **Matplotlib**. This allows for easy comparison between the original and the modified images.

plt.figure(figsize=(10, 5))

# Original Image

plt.subplot(1, 2, 1)

plt.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Original Image")

plt.axis('off')

# Output Image with Centroids

plt.subplot(1, 2, 2)

plt.imshow(cv2.cvtColor(output\_image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Output Image with Centroids")

plt.axis('off')

plt.tight\_layout()

plt.show()

The **Original Image** and the **Output Image with Centroids** are displayed in a grid layout.

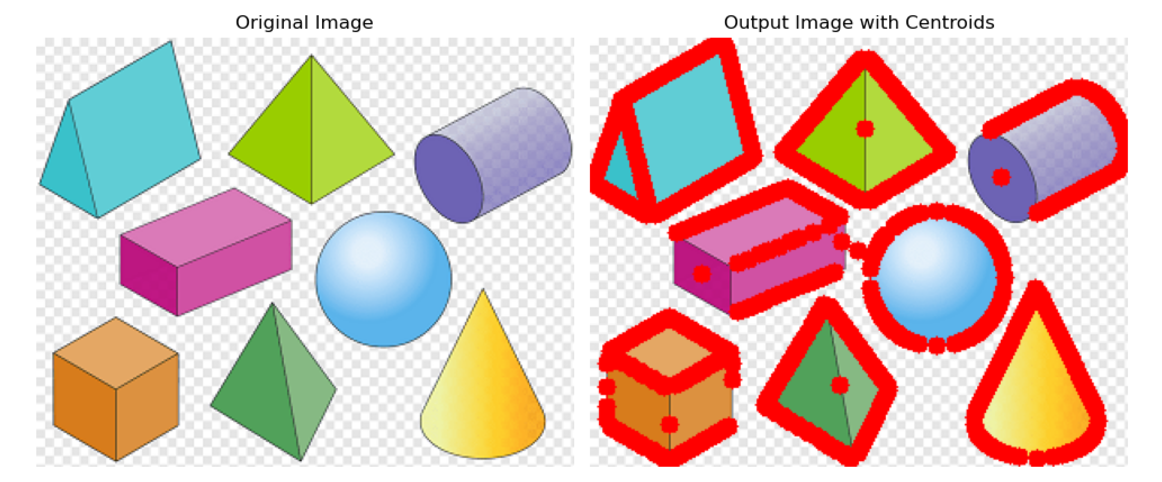
## **5. Example Output**

### **5.1. Original Image**

This is the input image before any processing is applied.

### **5.2. Output Image with Centroids**

This is the processed image with red circles marking the centroids of the detected shapes.



## **6. For What Images Does It Work?**

This program is designed to detect and mark the centroids of shapes in images where clear contours are present. It works well with:

* **Images with distinct, well-defined shapes**: For example, simple geometric shapes such as squares, circles, and triangles that are clearly distinguishable from the background.
* **High contrast images**: When the objects and the background have contrasting colors, it enhances the detection of contours and centroids.

**Example:**

* Simple images with geometric shapes like circles, squares, and triangles.

## **7. For What Kind of Images Does It Not Work?**

The program may not perform well on images where:

* **Shapes are blurred or merged**: If the shapes are not clearly separated or are blurred, the contours might not be detected properly, leading to incorrect centroid calculations.
* **Low-contrast or noisy images**: If the image has little contrast between objects and background, or if the image is noisy, it becomes difficult to detect contours and calculate the centroids accurately.
* **Complex shapes or irregular objects**: The program is best suited for detecting simple and regular shapes. It may struggle with complex or irregular shapes.

**Example:**

* Images with complex scenes or objects that are overlapping or blurry.

## **8. Conclusion**

This shape coordinates detection program demonstrates the use of **OpenCV** for detecting shapes in images and calculating their centroids. The results are visualized by marking the centroids with red circles on the image. This approach is widely applicable in computer vision and image processing tasks that involve shape detection and object tracking.

# **Subprogram 2: Shape Detection and Area Calculation**

## **1. Introduction 📚**

The project aims to implement basic image processing techniques for shape detection and area computation. This involves detecting shapes in an image, calculating their areas, and displaying the result. The project uses OpenCV to analyze an image, detect shapes, and then compute their areas based on contours.

## **2. Problem Statement 🧐**

The problem being solved is detecting various shapes (circles, squares, triangles, etc.) in an image and calculating their areas. The method applies image preprocessing (grayscale conversion, thresholding), contour detection, and area calculation to achieve this task.

## **3. Assumptions 🤔**

* The images contain simple shapes like circles, squares, or triangles with well-defined contours.
* The shapes are distinguishable from the background.
* The image is static, meaning no dynamic interaction is required.

## **4. Approach**

The solution is broken down into the following steps:

1. **Convert the image to grayscale**:   
   * The original image is converted to grayscale to simplify the processing and reduce computational complexity.
2. **Thresholding**:   
   * A binary threshold is applied to the grayscale image, turning it into a black-and-white image, where shapes are represented by white pixels, and the background is black.
3. **Contour Detection**:   
   * The contours of the shapes in the binary image are detected using OpenCV’s contour detection functions.
4. **Area Calculation**:   
   * The area of each detected contour is computed using the cv2.contourArea method, which provides the area of the shape enclosed by each contour.
5. **Overlay Areas on Image**:   
   * The calculated areas are overlayed on the original image for visualization.

## **5. Subprogram 2: Compute Area ✨**

This subprogram focuses on calculating the area of detected shapes in the image. The steps involve converting the image to grayscale, applying thresholding, finding contours, and computing the area for each shape.

### **Code Implementation for Subprogram 2:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def compute\_area(image):

"""

This function calculates the area of the shapes detected in the input image.

It works by finding the contours of the shapes and calculating the area

using the contourArea method provided by OpenCV.

Args:

- image (numpy.ndarray): The input image in which the shapes' areas need to be computed.

Returns:

- areas (list): A list of areas of each detected shape.

"""

# Convert the image to grayscale

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

# Apply a binary threshold to get a binary image

\_, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH\_BINARY)

# Find contours in the binary image

contours, \_ = cv2.findContours(thresh, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

# List to hold the areas of detected shapes

areas = []

# Loop over all contours and compute the area

for cnt in contours:

area = cv2.contourArea(cnt)

if area > 0: # Ignore very small contours or noise

areas.append(area)

return areas

def display\_image\_with\_areas(image, areas):

"""

Display the input image and overlay the areas of detected shapes.

Args:

- image (numpy.ndarray): The input image.

- areas (list): The list of areas to overlay on the image.

"""

output\_image = image.copy()

# Convert the image to grayscale for contour finding

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

\_, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH\_BINARY)

contours, \_ = cv2.findContours(thresh, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

# Draw the contours and overlay the area on the image

for i, cnt in enumerate(contours):

area = cv2.contourArea(cnt)

if area > 0:

x, y, w, h = cv2.boundingRect(cnt)

cv2.putText(output\_image, f"Area: {int(area)}", (x, y - 10),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 255, 0), 2)

cv2.drawContours(output\_image, [cnt], -1, (0, 255, 0), 2) # Green contour

# Display the original image and the output image with areas

plt.figure(figsize=(10, 5))

# Original Image

plt.subplot(1, 2, 1)

plt.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Original Image")

plt.axis('off')

# Output Image with Areas

plt.subplot(1, 2, 2)

plt.imshow(cv2.cvtColor(output\_image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Image with Areas")

plt.axis('off')

plt.tight\_layout()

plt.show()

# Main function to demonstrate area calculation

if \_\_name\_\_ == "\_\_main\_\_":

image\_path = 'shapes\_and\_colors.jpg' # Replace with your image path

image = cv2.imread(image\_path)

# 1. Compute areas

areas = compute\_area(image)

print("Shape Areas:", areas)

# 2. Display image with areas

display\_image\_with\_areas(image, areas)

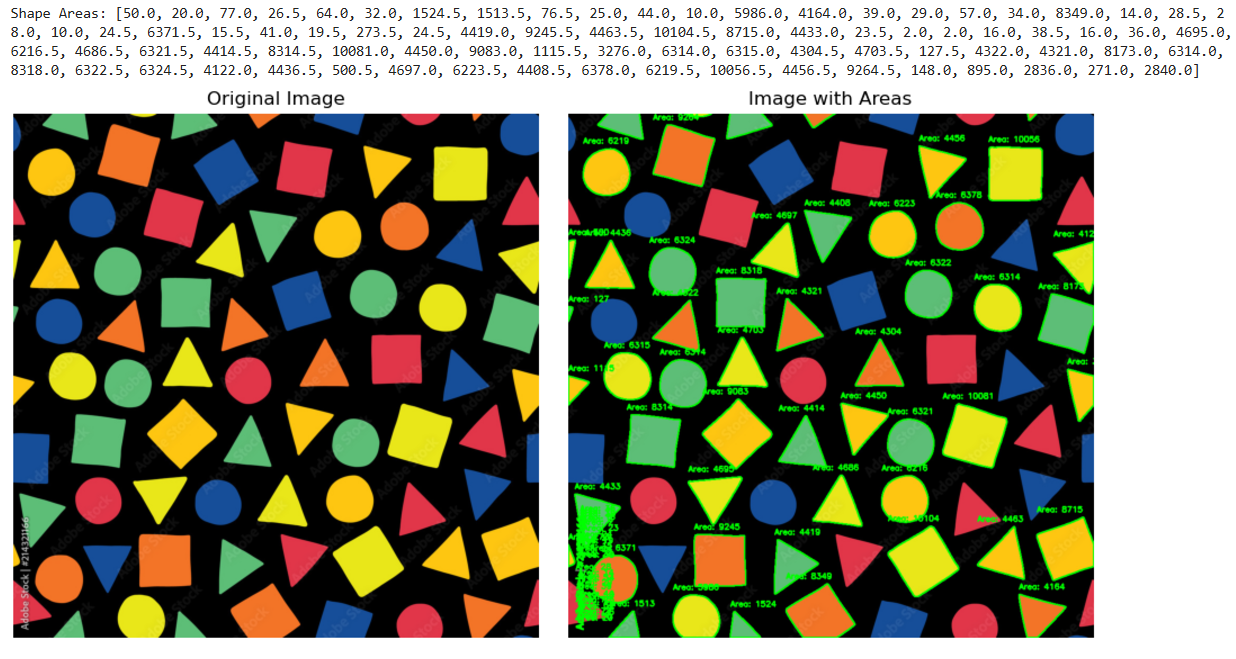
## **6. Results and Visualizations**

When you run the above code, it will calculate the areas of the detected shapes in the image and display the original image alongside the processed image with the areas overlaid.

### **Expected Output:**

* **Original Image**: Shows the original image with shapes.
* **Image with Areas**: Displays the original image with calculated areas shown on each detected shape.

### **Example Output:**

** *Image with areas displayed on each detected shape*

## **7. Use Cases**

* **Shape Detection in Manufacturing**: To analyze the size of objects in automated manufacturing processes.
* **Medical Imaging**: Detect and measure areas of certain regions in medical images such as tumors or organs.
* **Geometric Analysis**: Used in applications requiring the measurement of geometric objects from images.

## **8. Limitations and Future Work ⚠**

### **Limitations:**

* **Blurred Images**: The algorithm may not work effectively with images that are blurred or have unclear shapes.
* **Complex Backgrounds**: Shapes with complex or cluttered backgrounds may not be detected correctly.

## **9. Conclusion**

This approach effectively solves the problem of detecting shapes and calculating their areas in a given image using simple image processing techniques in Python with OpenCV. The method is efficient and can be extended to more complex applications involving shape recognition and analysis.

Here’s the **document version** of Subprogram 3: Detect Only Red Circles, including detailed explanations, limitations, and the code:

# **Subprogram 3: Detect Only Red Circles**

## **Introduction**

This subprogram is designed to **detect only red circles** in an image. The method utilizes color filtering and the **Hough Circle Transform** to find circular shapes that are specifically red in color. The process includes isolating the red color, detecting circles, and visualizing the results by overlaying the detected red circles on the original image.

## **How It Works**

1. **Convert to HSV Color Space** The first step in the detection process is to convert the input image from **BGR** (Blue-Green-Red) color space to **HSV** (Hue-Saturation-Value) color space. The reason for this is that HSV allows for easier detection of specific colors compared to the BGR space, as it separates the chromatic content (hue) from intensity (saturation and value).
2. **Red Color Masking** After converting the image to HSV, two ranges of red color values are defined to cover the full spectrum of red:  
   * **Lower red shades** (range close to 0 degrees on the hue scale).
   * **Upper red shades** (range close to 179 degrees on the hue scale).
3. These two ranges are then used to create **binary masks**, where all pixels falling within the defined red hue are marked. By combining both masks, we isolate all the red regions in the image.
4. **Circle Detection with Hough Transform** The next step involves detecting circles within the red regions. We convert the red-filtered image into grayscale, which simplifies the image and prepares it for circle detection. The **Hough Circle Transform** is then applied to find circular shapes by analyzing the gradients and edges in the image. This technique detects circles by looking for patterns in pixel intensities that correspond to circles.
5. **Overlaying Detected Circles** Once the circles are detected, they are drawn onto the original image. The detected circles are highlighted with **green circles** (with boundaries), allowing us to visualize the location and size of each red circle that was detected.

## **Code Implementation**

Here’s the code to detect red circles in an image:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def detect\_red\_circles(image):

# Convert the image from BGR to HSV

hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)

# Define lower and upper ranges for the red color

lower\_red1 = np.array([0, 100, 100])

upper\_red1 = np.array([10, 255, 255])

lower\_red2 = np.array([160, 100, 100])

upper\_red2 = np.array([179, 255, 255])

# Create masks for both red ranges

mask1 = cv2.inRange(hsv, lower\_red1, upper\_red1)

mask2 = cv2.inRange(hsv, lower\_red2, upper\_red2)

# Combine the two masks to isolate red regions

red\_mask = mask1 + mask2

# Use the red mask to extract the red regions from the image

red\_region = cv2.bitwise\_and(image, image, mask=red\_mask)

# Convert the red region to grayscale for circle detection

gray = cv2.cvtColor(red\_region, cv2.COLOR\_BGR2GRAY)

# Detect circles using Hough Circle Transform

circles = cv2.HoughCircles(

gray, cv2.HOUGH\_GRADIENT, dp=1, minDist=20, param1=50, param2=30, minRadius=10, maxRadius=100

)

return circles

def display\_detected\_circles(image, circles):

# Create a copy of the original image

output\_image = image.copy()

# If circles are detected, draw them on the output image

if circles is not None:

circles = np.round(circles[0, :]).astype("int")

for (x, y, r) in circles:

# Draw the circle center

cv2.circle(output\_image, (x, y), 2, (0, 255, 0), 3)

# Draw the circle perimeter

cv2.circle(output\_image, (x, y), r, (0, 255, 0), 3)

# Display the original image and the image with detected red circles

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

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

plt.title("Original Image")

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(cv2.cvtColor(output\_image, cv2.COLOR\_BGR2RGB))

plt.title("Red Circles Detected")

plt.axis('off')

plt.tight\_layout()

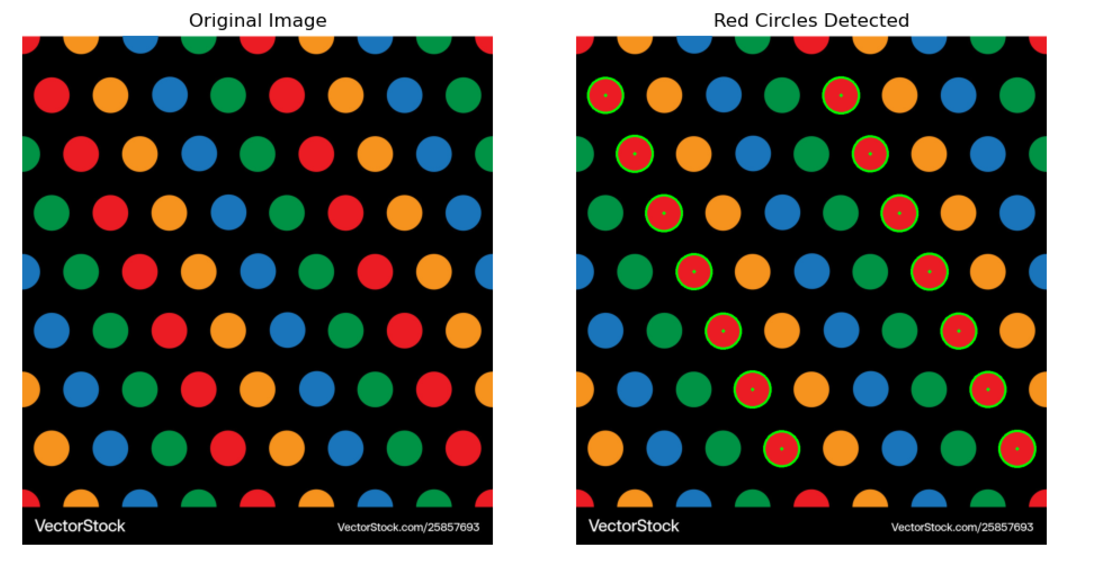
plt.show()

## **Limitations**

1. **Lighting Conditions** The method relies on the proper lighting to highlight red colors. In dim lighting or overexposed images, the red regions may be indistinguishable from other colors, leading to poor detection results.
2. **Overlapping Circles** If multiple red circles overlap or are very close to each other, the **Hough Circle Transform** may not detect them properly. It might group them as a single circle or miss smaller ones altogether.
3. **Noise** The detection is highly sensitive to noise, especially in areas where the red color is faint or where there are non-circular objects. Noise may result in false positives or missed detections.
4. **Circle Radius Range** The program currently detects circles within a specific range of radii (from 10 to 100 pixels). This may not be suitable for detecting very large or very small circles, requiring adjustments in the **minRadius** and **maxRadius** parameters of the **HoughCircles** function.

## **Conclusion**

This subprogram effectively detects **red circles** in an image by combining **color masking** and **Hough Circle Transform**. The output is a visualization of the **red circles** detected in the image, which are highlighted with green borders for clarity. Despite its effectiveness, the method does have certain limitations, such as dependency on lighting conditions and sensitivity to noise. Adjustments to the circle detection parameters or pre-processing techniques can help improve the robustness of the system.



# **Detecting All Green Shapes in an Image**

## **Overview:**

This program detects all **green shapes** present in a given image. The process involves color filtering to isolate the green shapes and **contour detection** to highlight the contours of these shapes. It uses the **HSV color space** for more accurate color detection compared to the RGB space. The contours of the detected shapes are drawn on the original image for visualization.

## **Code Implementation:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def detect\_green\_shapes(image):

"""

Detects all green shapes in the given image.

Uses color masking in HSV space to isolate green shapes and contours to identify the shapes.

Args:

- image (numpy.ndarray): The input image to detect green shapes in.

Returns:

- output\_image (numpy.ndarray): The image with detected green shapes highlighted.

"""

# Convert the image to HSV color space

hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)

# Define the range for green color in HSV space

lower\_green = np.array([35, 50, 50]) # Lower bound for green

upper\_green = np.array([85, 255, 255]) # Upper bound for green

# Create a mask to isolate green regions

green\_mask = cv2.inRange(hsv, lower\_green, upper\_green)

# Apply the mask to the image

green\_region = cv2.bitwise\_and(image, image, mask=green\_mask)

# Convert the green region to grayscale for contour detection

gray = cv2.cvtColor(green\_region, cv2.COLOR\_BGR2GRAY)

# Apply binary thresholding to get a clean binary image

\_, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH\_BINARY)

# Find contours of the green shapes

contours, \_ = cv2.findContours(thresh, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

# Create a copy of the original image to draw contours

output\_image = image.copy()

# Loop through each contour and draw it on the output image

for cnt in contours:

cv2.drawContours(output\_image, [cnt], -1, (0, 255, 0), 3) # Draw green contours

return output\_image

def display\_detected\_shapes(image, output\_image):

"""

Display the original image and the output image with detected green shapes.

Args:

- image (numpy.ndarray): The input image.

- output\_image (numpy.ndarray): The image with detected green shapes.

"""

plt.figure(figsize=(10, 5))

# Original Image

plt.subplot(1, 2, 1)

plt.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Original Image")

plt.axis('off')

# Output Image with Detected Green Shapes

plt.subplot(1, 2, 2)

plt.imshow(cv2.cvtColor(output\_image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Detected Green Shapes")

plt.axis('off')

plt.tight\_layout()

plt.show()

# Main function to demonstrate the detection of green shapes

if \_\_name\_\_ == "\_\_main\_\_":

image\_path = 'shapes\_and\_colors.jpg' # Replace with the path to your image

image = cv2.imread(image\_path)

# Detect green shapes

output\_image = detect\_green\_shapes(image)

# Display original image and image with detected green shapes

display\_detected\_shapes(image, output\_image)

## **Explanation:**

### **1. Convert to HSV Color Space:**

* The image is converted from **BGR** to **HSV** (Hue, Saturation, and Value). The **HSV color space** is more suitable for color-based detection as it separates chromatic content (Hue) from intensity (Value). 🌈

### **2. Green Color Masking:**

* We define the **lower and upper bounds for the green color** in HSV space, which allows us to isolate green regions in the image using a **mask**. The mask will filter out everything except green areas. 🟩

### **3. Apply Mask to Image:**

* The mask is applied to the image using cv2.bitwise\_and, which keeps only the green regions, making them easier to detect.

### **4. Contour Detection:**

* After isolating the green areas, we convert the masked image to **grayscale** and apply **binary thresholding** to obtain a clear binary image.
* We then use cv2.findContours to detect the boundaries of the green shapes. 🔍

### **5. Draw Contours:**

* Once the contours are detected, we use cv2.drawContours to draw green outlines around each detected green shape on a copy of the original image. 🖊️

### **6. Display Results:**

* The **original image** and the **image with detected green shapes** are displayed side by side using **Matplotlib**, enabling a visual comparison. 📸

## **Limitations:**

### **1. Lighting Conditions:**

* The program’s accuracy heavily depends on the **lighting** in the image. If the image is poorly lit, it might be challenging to isolate green shapes accurately due to **shadows** or **overexposed areas**. 💡

### **2. Predefined Green Range:**

* The green color detection is based on a predefined **HSV range**. If the green color in the image differs (e.g., darker or lighter shades), it might not be detected unless the range is adjusted. 🎨

### **3. Noise:**

* **Noise** in the image, such as pixel-level inconsistencies or background clutter, may result in **false positives**, where unwanted contours are also detected as green shapes. 🧳

### **4. Overlapping Shapes:**

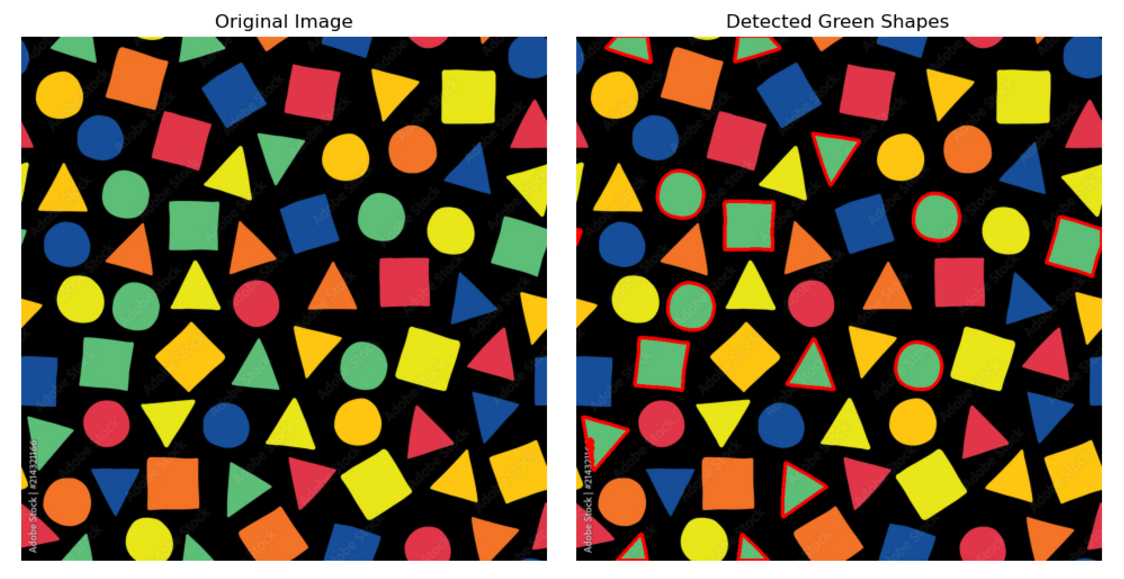
* The program may struggle to correctly detect **overlapping green shapes** or shapes that are too close together. In such cases, contours may merge, leading to inaccurate or incomplete detection. 🔴🟢

### **5. Contour Accuracy:**

* The accuracy of contour detection can vary based on the **complexity** of the shapes. Simple, well-defined shapes are detected more accurately, while more complex or irregular shapes might cause issues with contour extraction. 🔄

## **Conclusion:**

This program efficiently detects and highlights **green shapes** in images using **HSV-based color masking** and **contour detection**. By isolating the green color and identifying contours, it can be applied to various use cases such as **image analysis** and **object recognition**. However, its performance may be affected by **lighting conditions**, **noise**, and **overlapping shapes**, and improvements can be made for more accurate and adaptable results.



# **Subprogram 5: Detecting Large Circles**

## **1. Overview**

In this subprogram, we aim to detect large circles within an image using the **Hough Circle Transform**. This method allows us to identify circular shapes in an image, and we can filter the circles by size using adjustable **radius parameters**.

### **What the Program Does:**

* **Preprocessing**: The image is first converted to grayscale to simplify the detection process.
* **Noise Reduction**: A **Gaussian Blur** is applied to smooth the image and reduce noise, ensuring more accurate circle detection.
* **Circle Detection**: The **Hough Circle Transform** is used to detect circles of all sizes, but the detection range is filtered based on **minimum** and **maximum radius** values.
* **Customization**: By adjusting the **min\_radius** and **max\_radius**, we can fine-tune the program to detect **only large circles** while ignoring smaller ones.

## **2. Steps in the Program**

### **2.1 Image Preprocessing**

Before detecting circles, we need to prepare the image:

* **Convert to Grayscale**: This step simplifies the process of circle detection, as color information is not necessary for identifying shapes.
* **Apply Gaussian Blur**: The blur reduces noise and helps the algorithm to focus on the essential features in the image, making the circle detection process more accurate.

### **2.2 Circle Detection with Hough Circle Transform**

Using **OpenCV’s** Hough Circle Transform, we detect circles. The algorithm is based on identifying edges and gradients in the image that represent circular shapes. We adjust the **min\_radius** and **max\_radius** to focus on large circles by setting the minimum and maximum values of the radius that the program should detect.

## **3. Code Implementation**

Here is the Python code implementing **Subprogram 5: Detecting Large Circles**:

import cv2

import numpy as np

import matplotlib.pyplot as plt

def detect\_large\_circles(image, min\_radius=50, max\_radius=200):

"""

Detects large circles in the input image using the Hough Circle Transform.

Circles with radius greater than or equal to the min\_radius and less than or equal to max\_radius are considered large.

Args:

- image (numpy.ndarray): The input image to detect large circles in.

- min\_radius (int): The minimum radius of the circles to detect.

- max\_radius (int): The maximum radius of the circles to detect.

Returns:

- output\_image (numpy.ndarray): The image with large circles highlighted.

"""

# Convert the image to grayscale

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

# Apply Gaussian blur to reduce noise and improve circle detection

blurred = cv2.GaussianBlur(gray, (15, 15), 0)

# Detect circles using Hough Circle Transform

circles = cv2.HoughCircles(blurred, cv2.HOUGH\_GRADIENT, dp=1, minDist=20,

param1=50, param2=30, minRadius=min\_radius, maxRadius=max\_radius)

# If no circles are found, return the original image

if circles is None:

return image

# Convert the circle coordinates to integers

circles = np.round(circles[0, :]).astype("int")

# Create a copy of the original image to draw circles on

output\_image = image.copy()

# Draw each detected circle on the image

for (x, y, r) in circles:

cv2.circle(output\_image, (x, y), r, (0, 255, 0), 4) # Draw the circle in green

cv2.rectangle(output\_image, (x - 5, y - 5), (x + 5, y + 5), (0, 128, 255), -1) # Draw a center point

return output\_image

def display\_detected\_circles(image, output\_image):

"""

Display the original image and the output image with detected large circles.

Args:

- image (numpy.ndarray): The input image.

- output\_image (numpy.ndarray): The image with detected large circles.

"""

plt.figure(figsize=(10, 5))

# Original Image

plt.subplot(1, 2, 1)

plt.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Original Image")

plt.axis('off')

# Output Image with Detected Large Circles

plt.subplot(1, 2, 2)

plt.imshow(cv2.cvtColor(output\_image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct color display

plt.title("Detected Large Circles")

plt.axis('off')

plt.tight\_layout()

plt.show()

# Main function to demonstrate the detection of large circles

if \_\_name\_\_ == "\_\_main\_\_":

image\_path = 'shapes\_and\_colors.jpg' # Replace with the path to your image

image = cv2.imread(image\_path)

# Detect large circles with updated parameters (larger circles)

output\_image = detect\_large\_circles(image, min\_radius=50, max\_radius=200)

# Display original image and image with detected large circles

display\_detected\_circles(image, output\_image)

## **4. Limitations and Considerations**

### **4.1 Parameter Tuning**

The **min\_radius** and **max\_radius** parameters need to be adjusted based on the size of the circles in the image. Incorrect values may:

* Miss detecting large circles if the min\_radius is too large.
* Detect small circles if the max\_radius is set too large, which might not be desired.

### **4.2 Image Quality**

The accuracy of circle detection can be heavily influenced by the **quality of the input image**:

* **Noise**: High noise levels can lead to false detections. To mitigate this, **Gaussian blur** is used, but excessive blurring might distort the circles.
* **Complex Backgrounds**: In images with complex backgrounds or overlapping shapes, circle detection may not perform well without additional preprocessing, such as edge detection or thresholding.

### **4.3 Accuracy of Circle Detection**

While the **Hough Circle Transform** is a powerful algorithm, its accuracy is not perfect. In particular:

* The program may fail to detect circles if the image has poor contrast or blurred edges.
* Very small or irregularly shaped circles might not be detected accurately even with adjusted parameters.

## **5. Conclusion**

This subprogram effectively detects **large circles** by adjusting the **radius** parameters to filter out smaller circles. While it is robust in controlled conditions, it may require fine-tuning for images with varying lighting, noise, or circle sizes.

This method is useful for applications like **shape analysis**, **object detection**, or **image recognition**, where identifying circular objects is necessary.

# **🟠 Subprogram 6: Detecting Small Circles**

### **Purpose:**

This program aims to detect small circular shapes in an image using **Hough Circle Transform**. The key feature of this program is to find and highlight only the circles within a **specified radius range**, marking them distinctly on the image. It helps in isolating **smaller circles** while ignoring larger ones.

## **Program Overview:**

The **Hough Circle Transform** algorithm is used to detect circles based on their radius. By setting a minimum and maximum radius, we can restrict the detection to small circles, making the program effective in scenarios where detecting specific-sized circles is important.

### **Key Components:**

1. **Image Preprocessing**:  
   * Converts the image to **grayscale** to simplify the detection process.
   * Applies a **Gaussian blur** to reduce image noise and improve accuracy.
2. **Circle Detection**:  
   * Uses the **Hough Circle Transform** to detect circles within a defined **radius range**.
   * Highlights detected circles with **green outlines** and marks their centers with **orange squares**.
3. **Output**:  
   * Displays the original image alongside the modified image showing the detected small circles.

## **Code Explanation:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def detect\_small\_circles(image, min\_radius=10, max\_radius=30):

# Convert the image to grayscale

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

# Apply Gaussian blur to reduce noise

blurred = cv2.GaussianBlur(gray, (15, 15), 0)

# Detect circles using Hough Circle Transform

circles = cv2.HoughCircles(blurred, cv2.HOUGH\_GRADIENT, dp=1, minDist=20,

param1=50, param2=30, minRadius=min\_radius, maxRadius=max\_radius)

# If no circles are found, return the original image

if circles is None:

return image

# Convert the circle coordinates to integers

circles = np.round(circles[0, :]).astype("int")

# Create a copy of the original image to draw circles on

output\_image = image.copy()

# Draw each detected circle on the image

for (x, y, r) in circles:

cv2.circle(output\_image, (x, y), r, (0, 255, 0), 4) # Draw the circle in green

cv2.rectangle(output\_image, (x - 5, y - 5), (x + 5, y + 5), (0, 128, 255), -1) # Draw a center point

return output\_image

def display\_detected\_circles(image, output\_image):

# Display original image and the output image with detected small circles

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB)) # Convert to RGB for proper display

plt.title("Original Image")

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(cv2.cvtColor(output\_image, cv2.COLOR\_BGR2RGB)) # Convert to RGB for proper display

plt.title("Detected Small Circles")

plt.axis('off')

plt.tight\_layout()

plt.show()

# Main function to demonstrate small circle detection

if \_\_name\_\_ == "\_\_main\_\_":

image\_path = 'shapes\_and\_colors.jpg' # Path to your image

image = cv2.imread(image\_path)

# Detect small circles

output\_image = detect\_small\_circles(image, min\_radius=10, max\_radius=30)

# Display the results

display\_detected\_circles(image, output\_image)

## **Detailed Explanation:**

### **1. Image Preprocessing:**

The program starts by converting the input image into **grayscale**. This step is crucial because color information is not necessary for detecting circular shapes. Grayscale simplifies the image, making the detection process faster and more efficient.

Next, a **Gaussian blur** is applied to reduce noise. Blurring the image ensures that minor variations in pixel intensity don't interfere with the circle detection algorithm.

### **2. Circle Detection Using Hough Circle Transform:**

The **Hough Circle Transform** is an algorithm that detects circles in an image. It works by considering every point in the image as a potential center for a circle. The program allows for two important parameters:

* **minRadius**: This is the smallest radius a circle can have to be detected.
* **maxRadius**: This is the largest radius a circle can have to be detected.

The algorithm will detect all circles that fit within this defined range. In this program, the default values are set to detect circles with a radius between **10 pixels** and **30 pixels**.

### **3. Drawing Circles:**

Once the circles are detected, they are drawn on the **output image**. Each circle is outlined in **green**, and a small **orange square** is placed at the center of each circle to highlight its position.

### **4. Displaying Results:**

The program outputs two images side by side:

* **Original Image**: The original image without any modifications.
* **Detected Small Circles**: The image showing the small circles detected by the program.

## **Limitations:**

1. **Sensitivity to Radius**:  
   * If the min\_radius and max\_radius parameters are not set correctly, some circles may not be detected, especially in images where circles of different sizes appear.
   * Fine-tuning these parameters based on the image content is essential for optimal detection.
2. **Noise in Images**:  
   * The presence of noise or complex backgrounds might interfere with the detection process. In noisy images, circles could be missed or falsely detected.
   * The program may need additional noise reduction techniques or parameter adjustments to work accurately in such cases.
3. **Overlapping Circles**:  
   * If multiple circles are close to each other, the algorithm might incorrectly detect them as one. This can be alleviated by adjusting the minDist parameter.
4. **Complexity of Background**:  
   * The program might struggle in scenarios where there are multiple overlapping shapes or highly textured backgrounds. More advanced image processing methods might be needed for better accuracy in such conditions.

## **Key Features:**

* **Gaussian Blur**: Helps to smooth out the image and reduce noise.
* **Hough Circle Transform**: Detects circles based on radius and center coordinates.
* **Interactive Parameters**: Ability to adjust the min\_radius and max\_radius to target specific circle sizes.

## **Conclusion:**

The program provides a simple yet powerful way to detect **small circles** in an image. By adjusting the parameters and applying preprocessing techniques, the detection accuracy can be enhanced. This method is particularly useful when you need to focus on detecting specific-sized circular objects within images.



# **Subprogram 7: Detect and Count Corners**

### **Overview**

In this program, we use **Harris Corner Detection** to detect **corners** in an image. Corners are crucial features in image processing, often used in applications such as **object recognition**, **motion tracking**, and **image stitching**. The program identifies these corner points and marks them on the image, counting how many corners were detected.

### **How It Works**

1. **Convert the image to grayscale**: The first step is to convert the input image into a **grayscale image**. This is necessary because corner detection is typically performed on single-channel images.
2. **Apply Harris Corner Detection**: Using OpenCV's **cv2.cornerHarris()** function, we compute the corner response for each pixel in the grayscale image. The response tells us how likely each point is to be a corner.
3. **Thresholding the corners**: The detected corner responses are then thresholded. This step helps to filter out weak or insignificant corners. We keep the strongest corners based on the threshold value.
4. **Mark corners on the image**: Once we detect the corners, we mark them with a **red color** to make them visually identifiable.
5. **Display the original and processed image**: We show two images side by side. One is the original image, and the other is the image with the corners marked in red.

### **Code Implementation**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def detect\_corners(image, threshold=0.01):

# Convert to grayscale

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

# Apply Harris corner detection

gray = np.float32(gray)

corners = cv2.cornerHarris(gray, 2, 3, 0.04)

# Dilate to make the corners visible

corners = cv2.dilate(corners, None)

# Mark corners with red color

output\_image = image.copy()

output\_image[corners > threshold \* corners.max()] = [0, 0, 255]

# Get coordinates of corners

corner\_coords = np.column\_stack(np.where(corners > threshold \* corners.max()))

return corner\_coords, output\_image

def display\_corners(image, output\_image, corner\_coords):

# Display the original and output image

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

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

plt.title("Original Image")

plt.axis('off')

plt.subplot(1, 2, 2)

plt.imshow(cv2.cvtColor(output\_image, cv2.COLOR\_BGR2RGB))

plt.title("Detected Corners")

plt.axis('off')

plt.tight\_layout()

plt.show()

print(f"Number of detected corners: {len(corner\_coords)}")

### **Explanation of the Code**

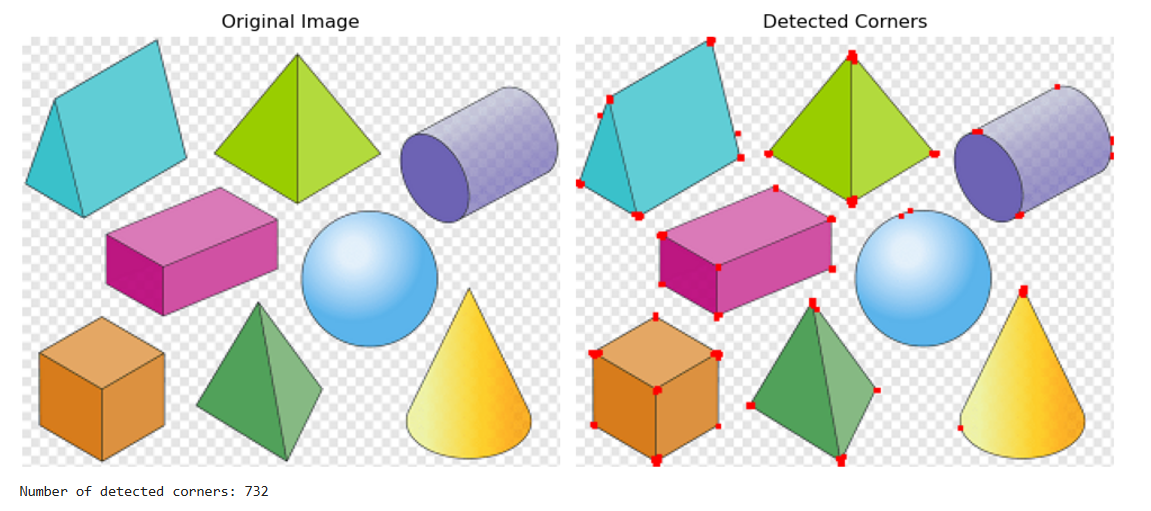
* **Grayscale Conversion**: We convert the input image to grayscale using OpenCV's cv2.cvtColor() function. This simplifies the image by removing color information, making it easier for corner detection to work.
* **Harris Corner Detection**: The cv2.cornerHarris() function calculates the corner response at each pixel. A corner response is a value indicating how much the pixel is a corner based on changes in intensity in different directions.
* **Thresholding**: To isolate the most significant corners, we apply a threshold. The threshold is based on the maximum corner response in the image. Corners with a response below the threshold are discarded.
* **Marking Corners**: Corners are marked with a red color [0, 0, 255] in the final image.
* **Displaying Results**: We use **Matplotlib** to display two images side by side: the original image and the image with marked corners. We also print the number of corners detected.

### **Limitations**

1. **Noise Sensitivity**: Harris corner detection can be sensitive to noise in the image. Noisy images may result in false corners being detected. Applying a **Gaussian blur** to the image before corner detection can help reduce noise.
2. **Threshold Tuning**: The **threshold** for detecting corners is crucial. A low threshold may detect too many corners, including noise, while a high threshold might miss smaller corners. Fine-tuning the threshold based on the image is necessary for accurate detection.
3. **Only Detecting Corners**: This method is specifically designed for detecting **corners**. It may not be suitable for detecting other features, such as edges or flat regions, and may not work well for images where corners are not well-defined.

### **Final Thoughts**

This subprogram provides a reliable method for **detecting and counting corners** in an image using Harris Corner Detection. The detected corners can be used in a variety of image processing applications, such as feature matching and object tracking. However, the method requires tuning and may not perform well on noisy or low-contrast images.



## **Subprogram 8: Shape Segmentation and Display 📸✂️**

### **Objective:**

This subprogram isolates individual shapes in an image and displays them in a grid format. The process involves edge detection, contour finding, and segmentation. It allows users to visually analyze separate shapes and their segments.

### **How It Works:**

#### **1. Image Preprocessing:**

* **What is happening**:
  + The image is first read using cv2.imread() and converted to grayscale using cv2.cvtColor(). Grayscale images are simpler for detecting edges and contours.
  + A **Gaussian Blur** is applied to reduce noise in the image using cv2.GaussianBlur(). This helps in detecting clear edges.
* **Why is this important?**
  + Grayscale conversion simplifies the image. The blur reduces small noises which could otherwise be detected as edges.

#### **2. Edge Detection:**

* **What is happening**:  
  + **Canny Edge Detection** is applied using cv2.Canny(). This detects edges in the image where there is a significant intensity gradient.
  + Edge detection is crucial for isolating the shapes, as contours are derived from edges.
* **Why is this important?**
  + By detecting edges, we can later find the boundaries of each shape. Canny is popular because it detects edges effectively by suppressing weak edges and detecting strong ones.

#### **3. Contour Finding:**

* **What is happening**:  
  + **Contours** are detected using cv2.findContours(). This function finds the boundaries of objects detected by the edge detection algorithm.
  + It retrieves the external contours (cv2.RETR\_EXTERNAL), which helps in isolating individual shapes.
* **Why is this important?**
  + Contours are needed to distinguish the boundaries of the shapes, which will later be segmented and displayed.

#### **4. Shape Segmentation:**

* **What is happening**:  
  + For each detected contour, we calculate its area using cv2.contourArea(). Only contours with an area larger than a threshold (500 pixels in this case) are considered for segmentation.
  + A **mask** is created using np.zeros\_like() (black mask), and the contour is drawn onto this mask using cv2.drawContours().
  + The original image is then segmented by applying this mask using cv2.bitwise\_and().
* **Why is this important?**
  + By isolating contours and applying the mask, we create separate images for each individual shape. This allows the shapes to be displayed clearly.

#### **5. Displaying the Original Image and Segmented Shapes:**

* **What is happening**:  
  + The **original image** is displayed first, followed by each segmented shape. The segmented shapes are arranged in a grid format, with 2 images per row.
  + The display is handled using matplotlib.pyplot, which allows for flexible plotting and display of images.
* **Why is this important?**
  + Displaying the original image first gives context to the segmented shapes. The grid layout helps visualize multiple segmented shapes efficiently.

### **Code Explanation:**

import cv2

import numpy as np

import matplotlib.pyplot as plt

def segment\_shapes(image\_path):

# Read image, convert to grayscale, apply Gaussian blur, and detect edges

image = cv2.imread(image\_path)

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY) # Convert image to grayscale

blurred = cv2.GaussianBlur(gray, (5, 5), 0) # Apply Gaussian blur to reduce noise

edges = cv2.Canny(blurred, 50, 150) # Detect edges using Canny edge detection

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

segmented\_images = []

for cnt in contours:

area = cv2.contourArea(cnt) # Calculate area of each contour

if area > 500: # Only consider contours with an area greater than 500

mask = np.zeros\_like(image) # Create a black mask

cv2.drawContours(mask, [cnt], -1, (255, 255, 255), thickness=cv2.FILLED) # Fill the contour in the mask

segmented\_image = cv2.bitwise\_and(image, mask) # Apply the mask to the original image

segmented\_images.append(segmented\_image) # Add the segmented image to the list

return image, segmented\_images

def display\_segmented\_images(original\_image, segmented\_images):

num\_images = len(segmented\_images) # Get the number of segmented images

# Calculate grid layout with 2 images per row

cols = 2

rows = (num\_images // cols) + (num\_images % cols > 0) # Adjust rows to fit all images

plt.figure(figsize=(10, rows \* 3))

# Display original image first

plt.subplot(rows + 1, cols, 1)

plt.imshow(cv2.cvtColor(original\_image, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct display

plt.title("Original Image")

plt.axis('off')

# Display each segmented image

for i, seg\_img in enumerate(segmented\_images):

plt.subplot(rows + 1, cols, i + 2)

plt.imshow(cv2.cvtColor(seg\_img, cv2.COLOR\_BGR2RGB)) # Convert BGR to RGB for correct display

plt.title(f"Segment {i + 1}")

plt.axis('off')

plt.tight\_layout()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

image\_path = 'shapes.png' # Image path, replace with your own image

original\_image, segmented\_images = segment\_shapes(image\_path)

display\_segmented\_images(original\_image, segmented\_images)

### **Key Functions:**

* **segment\_shapes(image\_path)**: Segments the shapes from the given image and returns both the original image and a list of segmented shapes.
* **display\_segmented\_images(original\_image, segmented\_images)**: Displays the original image and segmented shapes in a grid format.

#### **1. What is happening in the code?**

* The code begins by reading and preprocessing the image (grayscale and blurring). It then detects edges using the Canny method and identifies contours in the image. For each contour, a mask is created, and the shape is isolated and stored. Finally, the original image and segmented shapes are displayed.

#### **2. What images are not suitable for this program?**

* **Images with overlapping shapes**: If shapes are too close together or overlap, the contours might not be detected correctly, leading to inaccurate segmentation.
* **Low-contrast images**: If there isn’t a clear distinction between shapes and background, the Canny edge detection might not identify the edges well, and segmentation might fail.
* **Noisy images**: If there is too much noise in the image, especially small details, it could result in detecting many small, irrelevant contours. Gaussian blur helps reduce noise, but excessive noise could still cause problems.
* **Complex shapes**: Highly irregular shapes or shapes with intricate details might not be detected effectively by contour detection.

### **Limitations:**

1. **Shape Size Threshold**: Only shapes with an area larger than 500 pixels are considered. Smaller shapes will not be detected unless this threshold is lowered.
2. **Edge Detection Sensitivity**: The parameters for **Canny edge detection** (50, 150) are set for general use but may need adjustments for different types of images.
3. **Shape Overlap**: Shapes that are very close or overlapping might not be segmented properly.
4. **Noise**: Excessive noise could lead to incorrect segmentation or missing contours.

### **Conclusion:**

This subprogram successfully segments individual shapes from an image and displays them in a clear grid format. While it performs well with simple and clean images, it might face challenges with complex, noisy, or overlapping shapes. With the ability to adjust parameters like the area threshold and Canny edge detection limits, this program is flexible enough to be adapted for various use cases.

