```
def bit_plane_slice(image):
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

this function bit_plane_slice takes an input image, converts it to grayscale using OpenCV's cvtColor function, and returns the grayscale version of the image.

```
    bit_planes = []
    for i in range(8):
        mask = 2 ** i
        bit_plane = np.bitwise_and(gray_image, mask)
        bit_planes.append(bit_plane)
    return bit_planes
```

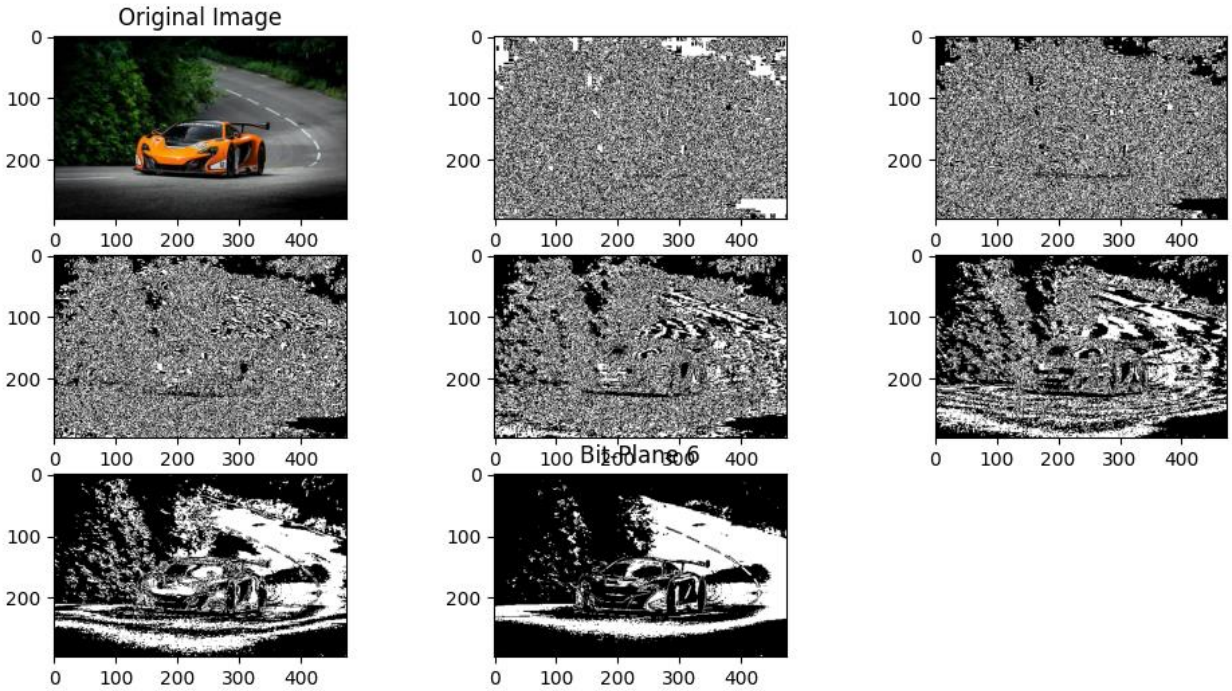
This list will store the bit-plane images generated in the following loop, this part of the code iterates over each bit-plane from 0 to 7, generates a mask for each bit-plane, applies the mask to the grayscale image to extract the corresponding bit-plane, and stores each bit-plane image in a list.

```
image_path = "op.jpg"
image = cv2.imread(image_path)
bit_planes = bit_plane_slice(image)
```

this code segment loads an image from a file specified by image_path using OpenCV's imread function and then applies the bit_plane_slice function to perform bit-plane slicing on the loaded imag

```
plt.figure(figsize=(12, 6))
plt.subplot(3, 3, 1), plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB)),
plt.title('Original Image')
for i in range(7):
    plt.subplot(3, 3, i+2), plt.imshow(bit_planes[i], cmap='gray'),
plt.title('Bit-Plane ' + str(i))
plt.show()
```

This code provides a comprehensive visualization of both the original image and its bit-plane decomposition, aiding in better understanding the contribution of individual bits to the overall image.


Original Image
Bit Plane 6

Method: adaptive thresholding

Parameters:

- **`image`**: The input image for which adaptive thresholding is performed.
- **`new_mean`**: The initial threshold value used to segment the image.
- **`error`**: The error tolerance to stop the iterative process.

Description: This method implements adaptive thresholding, a technique used in image processing to segment an image into foreground and background regions based on local intensity characteristics. It iteratively updates the threshold value until convergence, aiming to minimize the difference between the mean intensities of the segmented regions.

Method: thresholding

Parameters:

- **`image`**: The input image to be segmented.
- **`L`**: The threshold value used to separate foreground and background regions.

Description: The **`thresholding`** method performs a basic thresholding operation on the input image, segmenting it into binary regions based on a given threshold value **`L`**. Pixels with intensities below **`L`** are assigned to the background (0), while pixels with intensities above **`L`** are assigned to the foreground

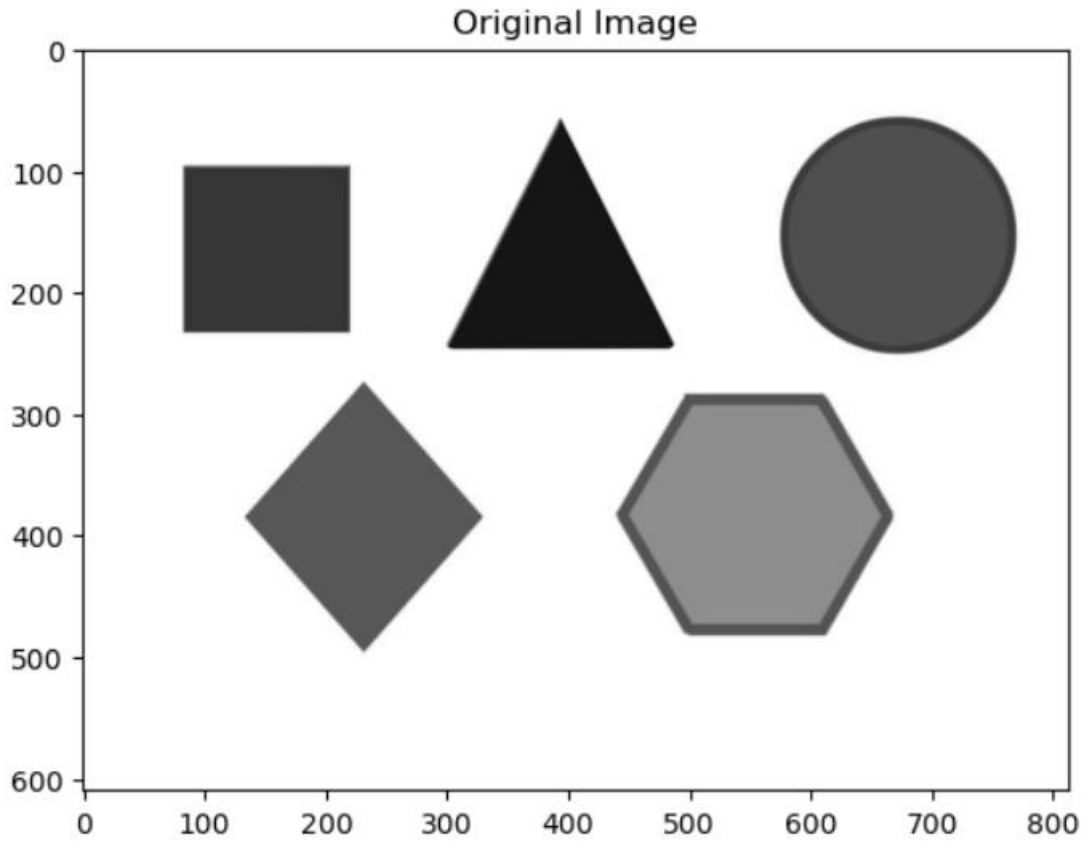
Adaptive Thresholding Loop:

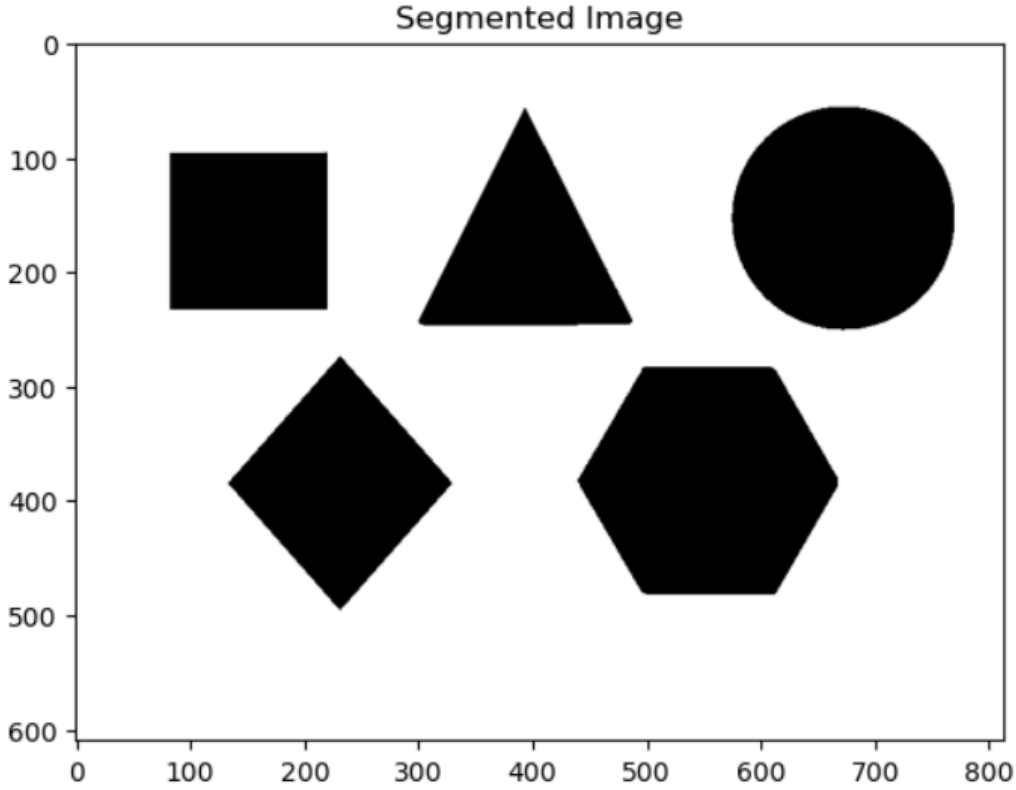
- The method **`adaptive_thresholding`** iteratively updates the threshold value until convergence, where the difference between consecutive threshold values falls below a specified error threshold.
- Within each iteration:
  - It calculates the mean intensities of the pixels below and above the current threshold value (**`new_mean`**).
  - Updates **`new_mean`** to the average of these two means.
  - Segments the image using the updated threshold value via the **`thresholding`** method.
  - Displays the segmented image using Matplotlib.
  - Prints the mean intensities of the two segments and the updated threshold value.
- The loop terminates when the absolute difference between the old and new threshold values is less than or equal to the specified error tolerance.

Thresholding Method:

- The **`thresholding`** method takes an image and a threshold value (**`L`**) as input.

- It creates a binary mask (**`image_segment`**) where pixels below the threshold value are set to 0 (background) and pixels above or equal to the threshold value are set to 1 (foreground).
- The method returns the binary segmented image.




Segmented Image
0
100
200
300
400
500
600
0
100
200
300
400
500
600
700
800

Log transformation

Converting to Grayscale:

gray_image = cv2.cvtColor(np.array(image), cv2.COLOR_BGR2GRAY)

It converts the input image to grayscale using OpenCV's cvtColor function.

Adding a Small Value:

```
    gray_image = gray_image + 1e-10
```

A small value (1e-10) is added to the grayscale image. This is typically done to avoid taking the logarithm of zero, which is undefined.

Logarithmic Transformation:

transformed_image = np.log(gray_image)

The logarithm of the grayscale image is computed element-wise. This is a common technique to enhance the contrast of images.

Scaling:

transformed_image = np.uint8(transformed_image * 255 / np.max(transformed_image))

The transformed image is scaled so that its values are in the range [0, 255]. This is typically done for display purposes.

Converting Back to Image:

```
transformed_image_pil = Image.fromarray(transformed_image)
        return transformed_image_pil
```

The transformed image is converted back to a PIL (Python Imaging Library) image using Image.fromarray.

Input photo :

Output photo :

power_trnsform

- **cv2.cvtColor(img,cv2.COLOR_BGR2RGB)**

By performing this conversion, you're essentially swapping the order of color channels from BGR to RGB. This conversion is often necessary when you want to display an image using libraries like Matplotlib, which expect images in RGB format.

- **imge_normalize = img_converted / 255**

By dividing the pixel values by 255, you scale them down to the range [0, 1]. This normalization ensures that the pixel values are within a consistent range .

- **np.power(imge_normalize, power)**

The np.power function from NumPy is used here to perform element-wise power calculation. It takes two arguments: the base (the normalized pixel values) and the exponent (the power parameter), and returns an array where each element is the result of raising the corresponding element of the base to the power of the exponent .

- **plt.figure(figsize=[15,5])**

  By setting the figure size using figsize, you can control the dimensions of the plot that will be displayed when you subsequently add plots or images to the figure using other matplotlib functions like plt.imshow().

Thresholding Image Report

Abstract

This report outlines the process of thresholding an image using Python and OpenCV. The purpose of this code is to convert a given image into a binary format, separating foreground objects from the background.

Introduction

Thresholding is a fundamental technique in image processing used to segment images by converting grayscale or color images into binary images. This report details the implementation of a thresholding algorithm using Python's OpenCV library.

Problem Statement

The task is to implement a function that takes an image file path as input, performs thresholding on the image, and returns the thresholded image.

Methodology

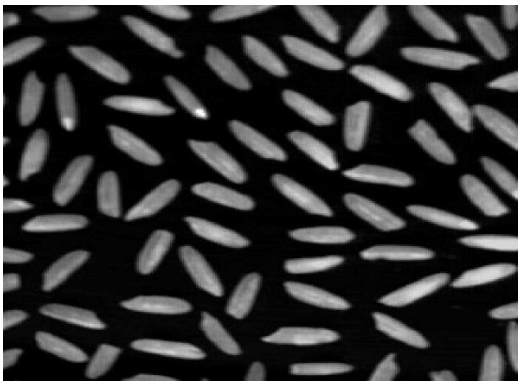
The methodology involves the following steps:

Read the image using OpenCV's `imread` function.

1. Convert the image to grayscale using `cvtColor`.
2. Calculate the optimal threshold value using OpenCV's `threshold` function with the Otsu thresholding method.
3. Apply the calculated threshold to the grayscale image.
4. Display the thresholded image using matplotlib.

Result

ORIGINAL



AFTER TRESHOLDING

Implementation Details

The `Thresholding` function takes an image file path as input. It reads the image using OpenCV, converts it to grayscale, calculates the optimal threshold value using the Otsu method, applies the threshold to the grayscale image, and displays the thresholded image using matplotlib.

Conclusion

This report has presented a simple yet effective implementation of image thresholding using Python and OpenCV. The function outlined here can be used to threshold images for various applications in image processing and computer vision.

References

- OpenCV Documentation: https://opencv.org/
- Matplotlib Documentation: https://matplotlib.org/

Watershed Segmentation

```
dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
ret, sure_fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(), 255, 0)
```

-This code calculates the distance transform of the binary image `opening`, representing the distance of each pixel to the nearest background pixel. Then, it thresholds the distance transform to obtain `sure_fg`, highlighting foreground regions likely to belong to objects, based on a threshold set at 70% of the maximum distance value.

```
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg, sure_fg)
```

-This code converts the thresholded foreground image (`sure_fg`) to an 8-bit unsigned integer format. Then, it computes the regions of uncertainty (`unknown`) by subtracting the certain background (`sure_bg`) from the certain foreground (`sure_fg`). This identifies regions where object presence is uncertain.

```
markers = cv2.watershed(image, markers)
```

-`cv2.watershed` divides an image into different sections, like how water divides a landscape, based on markers. It helps in segmenting objects and defining their boundaries in the image.

```
image[markers == -1] = [0, 0, 255]
```

-This line of code modifies the original image by assigning a specific color (blue) to regions marked as -1 in the watershed segmentation result. This typically highlights the boundaries of segmented objects in the image.

Original

Segmented