DSA5203 Assignment 2 Report

Lin Xueshun A0274618W

The purpose of this project is to implement image rectification algorithm with a rectangular shape image input against a monochrome background, the aim is to adjust the image such that the edges of the object align horizontally or vertically within the image frame. The output should be an image where the rectangular object is properly oriented, with its boundaries parallel to the image's edges.

This report records all walkthrough against rectifying the test image.

Firstly, the test image need to be loaded, here test1.jpg was took as the example. The image was converted to RGB format from BGR temporarily for the review.

```
# import the necessary packages
import cv2
import matplotlib.pyplot as plt
import numpy as np

# load the input image from disk
input_image = cv2.imread("./data/test1.jpg")
#input_image = cv2.imread("./data/test2.jpg")

# convert the image to RGB (OpenCV uses BGR), then display the image
input_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2RGB)
plt.imshow(input_image)
```

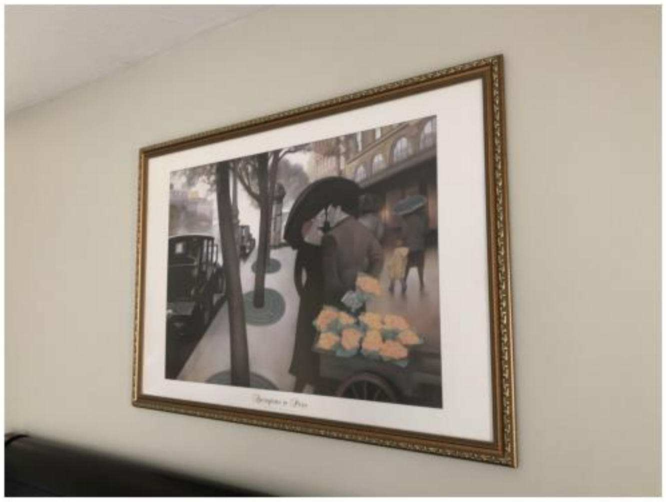

Figure 1 Example test image

Then the image is changed to grayscale to simplify the data, reducing computational complexity and focusing on luminance rather than color, which is beneficial for subsequent processing steps like edge detection or contour analysis.

```
# Convert the image to grayscale
img_gray = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)
plt.imshow(img_gray, cmap='gray')
plt.axis('off')
```

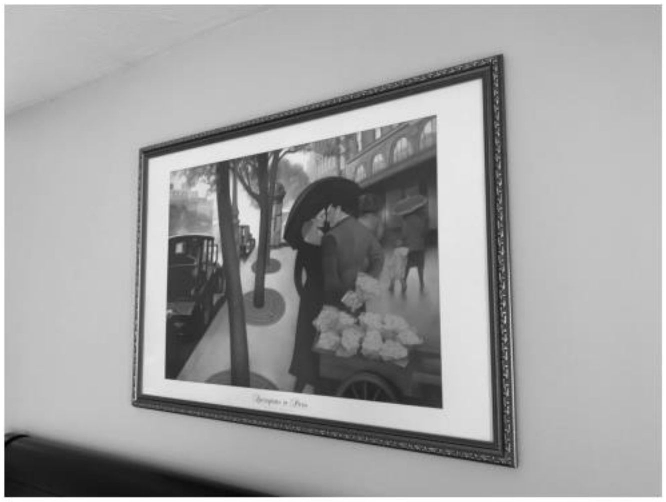

Figure 2 Test image in gray scale

An averaging filter was applied to blur the image, effectively smoothing out variations to reduce noise and help in highlighting the significant structures in the image.

```
# Applying a average blur to the image
img_blur = cv2.blur(img_gray, (10,10))
plt.imshow(img_blur, cmap='gray')
plt.axis('off')
```

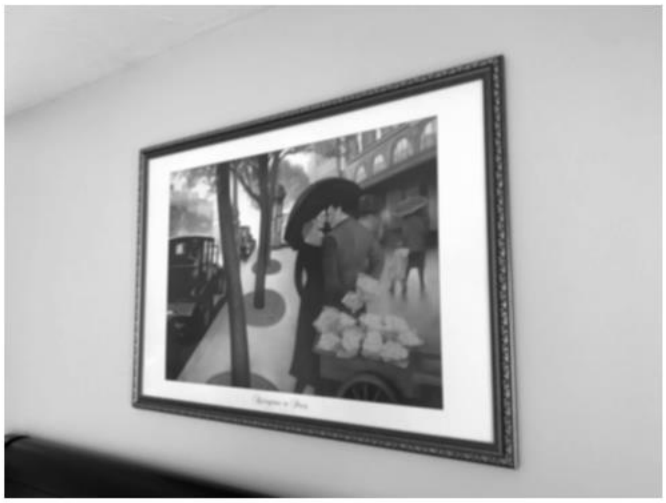

Figure 3 Blurred image with average filter

After blurring the image to minimize noise, an adaptive thresholding technique was employed to convert the image into a binary format, i.e., black and white for further denoising and reducing computational complexity. Specifically, cv2.adaptiveThreshold was utilized with a maximum value of 255 and the cv2.ADAPTIVE_THRESH_MEAN_C method, which calculates the mean of a 21x21 pixel neighborhood as the threshold value. A constant of 10 was subtracted from this mean to fine-tune the thresholding process, which was tried to have great effects. This approach is particularly effective in compensating for different illumination environments, thereby making the overall algorithm more adaptable to each unique image.

```
# used adaptive thresholding to create a binary image
# used the mean thresholding method
# in case to deal with different lighting conditions
```

```
img_thresh = cv2.adaptiveThreshold(img_blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 21,
10)
plt.imshow(img_thresh, cmap='gray')
plt.axis('off')
```

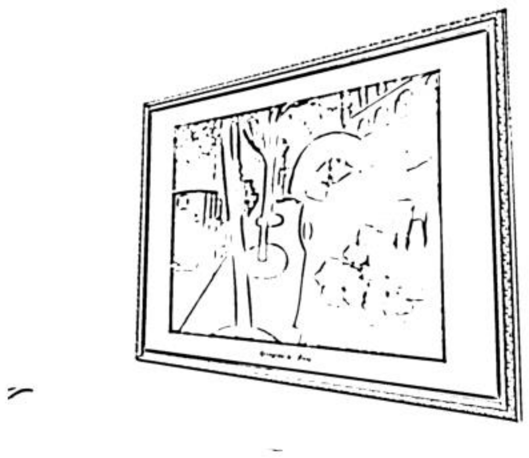

Figure 4 Test image after adaptive thresholding

Following the adaptive thresholding, the Canny edge detector was employed to identify the edges within the image. The function was applied to the thresholded image with a lower threshold of 50 and an upper threshold of 200. This step is crucial for delineating the boundaries and contours of objects within the image, facilitating further analysis or processing.

```
# Detecting edges in the imagel, using the Canny edge detector
img_edges = cv2.Canny(img_thresh, 50, 200)
plt.imshow(img_edges, cmap='gray')
plt.axis('off')
```

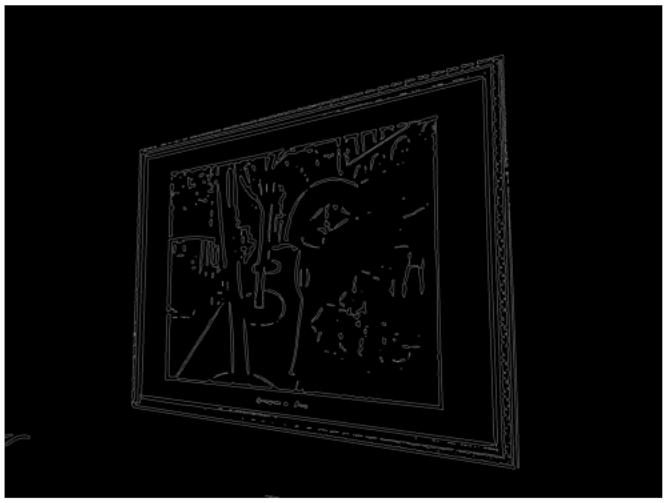

Figure 5: Detected edges of the test image

To refine the edge detection further and close gaps between object edges, a morphological closing operation was applied. The algorithm dilates the edges, thereby bridging small gaps and breaks. Then, erosion is applied, which refines the dilated edges, removing the excess width added during dilation and preserving the original shape and size of objects as much as possible.

This operation utilizes an elliptical-shaped kernel, created with cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3)), which is especially effective for smoothing the contours and closing small holes or gaps within the detected edges. While other kernel types were also tried, e.g. MORPH_CROSS and MORPH_RECT, the MORPH_ELLIPSE showed the best performance.

Then cv2.morphologyEx function was used on the edges obtained from the Canny edge detector, specifying the cv2.MORPH_CLOSE operation to perform the closing process. To ensure thorough closure of gaps, the operation was iterated 10 times.

```
# Applying a morphological operation to close gaps in between object edges
# used a elliptical shaped kernel
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3))
img_closed = cv2.morphologyEx(img_edges, cv2.MORPH_CLOSE, kernel, iterations=10)
plt.imshow(img_closed, cmap='gray')
plt.axis('off')
```

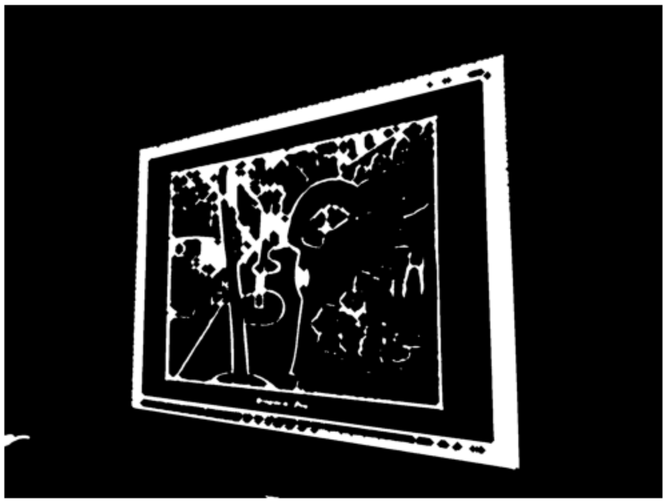

Figure 6: Test image after closing operation

Upon completing the morphological closing operation to enhance edge definition, the next step involved the identification of contours within the image. Utilizing the cv2.findContours method, contours were extracted from the closed image.

Among the detected contours, they were sorted by area, and the largest contour was selected for further processing. This approach ensures that the primary object of interest is highlighted, particularly useful in scenarios where the object occupies a significant portion of the image.

To visualize the largest contour, a new image was created with the same dimensions as the original input image but initialized to black (using np.zeros_like(input_image)). The largest contour was then drawn onto this black canvas with white color ((255, 255, 255)) and a thickness of 3 pixels, making the contour distinctly visible against the black background.

```
# Finding the contours in the image using the cv2.findContours method
contours, _ = cv2.findContours(img_closed, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
# Sorting the contours by area, and keeping only the largest one
maxcontour = max(contours, key=cv2.contourArea)
# Creating a black image with the same dimensions as the input image, and drawing the largest contour
black = np.zeros_like(input_image)
cv2.drawContours(black, [maxcontour], -1, (255, 255, 255), 3)
plt.imshow(black, cmap='gray')
plt.axis('off')
```

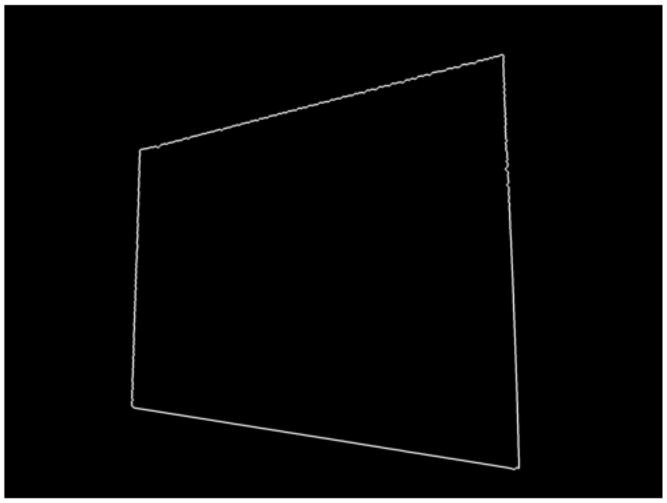

Figure 7: The outmost contour of the test image

Following the isolation of the largest contour against a black background, the next phase involved refining the analysis to pinpoint the corners of this primary object. To approximate the polygonal curves of the contour and thus identify the corners, the cv2.approxPolyDP function was employed. An epsilon value, which dictates the approximation accuracy, was set to 1% of the contour's arc length (cv2.arcLength(maxcontour, True)). This epsilon value strikes a balance between accuracy and the level of approximation, ensuring that the resulting polygon closely resembles the original contour while simplifying its complexity. The True parameter specifies that the polygon is closed.

With the corners identified, they were then highlighted directly on the grayscale image of the contour. Each corner was marked by drawing a white circle ((255, 255, 255)) with a radius of 20 pixels around the corner's coordinates, achieved by iterating through the corners array.

```
img_corner = cv2.cvtColor(black, cv2.COLOR_BGR2GRAY)
# Approximating the polygonal curves of the contour, then finding the corners of the polygon
epsilon = 0.01 * cv2.arcLength(maxcontour, True)
corners = cv2.approxPolyDP(maxcontour, epsilon, True)


# Drawing the corners on the contour image
```

```
for i in corners:
    x, y = i.ravel()
    cv2.circle(img_corner, (x, y), 20, (255,255,255), -1)

plt.imshow(img_corner, cmap='gray')
plt.axis('off')
```

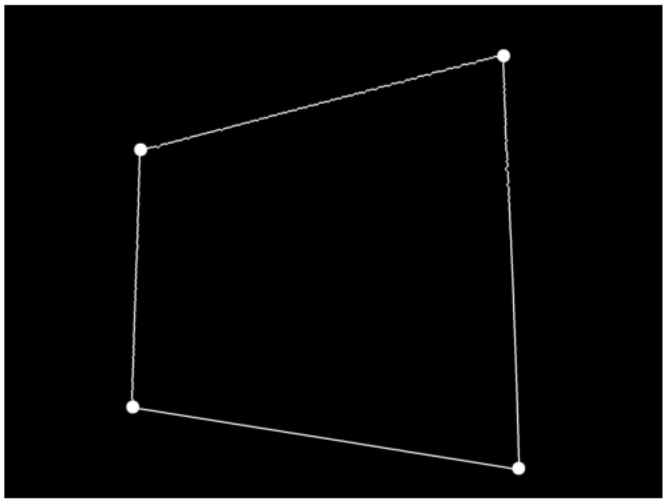

Figure 8: The identified image plotted on the contour

After identifying and marking the corners of the primary object within the image, the next step involved organizing these corner points in a specific order to facilitate further processing, such as perspective transformation or geometric analysis. The initial task was to sort the corner points based on their x-coordinates. This sorting enabled the separation of points into left and right groups, corresponding to their position relative to the object's geometry.

To determine the top-left and bottom-left points among the left subset, a secondary sorting was performed based on the y-coordinates. Since the coordinate system of images has its origin at the top-left corner, the point with the lesser y-coordinate value (after sorting) was identified as the top-left point, and the one with the greater y-coordinate value was the bottom-left point.

For the right subset of points, a Euclidean distance calculation was employed to differentiate between the top-right and bottom-right points. The distance between each of the right side points and the top-left point was computed. The point farther from the top-left point was designated as the bottom-right point due to the geometric property that in a rectangular object, diagonal points are at the maximum distance from each other. Consequently, the other point was identified as the top-right.

```
# sort the points based on their x-coordinates
horizontally_sorted = corners[np.argsort(corners[:, 0, 0])].squeeze()
# grab the left side points and right side points from the sorted x-coordinates
left = horizontally_sorted[:2]
right = horizontally_sorted[2:]

# sort the points in the left side so that the top-left point is the one with higher y-coordinate
# and the bottom-left point is the one with lower y-coordinate
top_left, bottom_left = left[np.argsort(left[:, 1])]

# compute the Euclidean distance between the top-left and the two right side points
# the point with the maximum distance will be the bottom-right point
distance = np.linalg.norm(top_left - right, axis=1)
top_right, bottom_right = right[np.argsort(distance)]

# re-arrange the points in the order
corners_in_order = np.float32([top_left, top_right, bottom_right, bottom_left])
```

Then, plot the corner points on the original test image in red for better visualization.

```
# plot the corner points on the original test image
for i in corners_in_order:
    x, y = i.ravel()
    cv2.circle(input_image, (int(x), int(y)), 10, (255,0,0), -1)
plt.imshow(input_image, cmap='gray')
plt.axis('off')
```

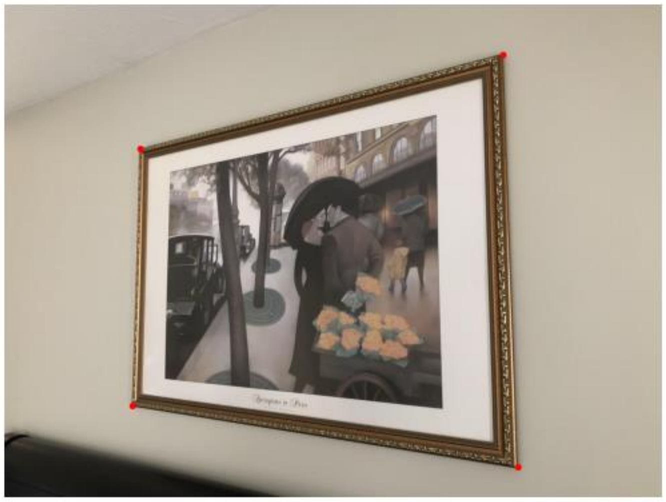

Figure 9: Test image with detected corner points

After determining and organizing the corners of the primary object, the final step was to rectify the image to present the object in a frontal view, standardizing its appearance. The output image's dimensions were set by averaging the distances between opposite corners, defining a proportional and geometrically consistent view.

A perspective transformation was then calculated using cv2.getPerspectiveTransform, mapping the original corners to a new set of points that reflect the desired rectified orientation. This mapped the object onto a standard rectangular frame, accounting for a uniform margin around it, so keep the backgound with the mainbody of the image as well

Using cv2.warpPerspective, the transformation was applied to the original image, resulting in a rectified image where the object appears directly facing the viewer.

```
# Define the width and height of the output image, set to the average of the distances between the corners
width = np.mean([np.linalg.norm(top_left - top_right), np.linalg.norm(bottom_left - bottom_right)])
height = np.mean([np.linalg.norm(top_left - bottom_left), np.linalg.norm(top_right - bottom_right)])

# Define the four corners of the output image
targets = np.float32([[50, 50], [width - 50, 50], [width - 50, height - 50], [50, height - 50]])

# Compute the perspective transform matrix
```

```
M = cv2.getPerspectiveTransform(corners_in_order, targets)


# Apply the perspective transform to the input image
img_rectified = cv2.warpPerspective(input_image, M, (int(width), int(height)))


# Display the rectified image
plt.imshow(img_rectified)
plt.axis('off')
```

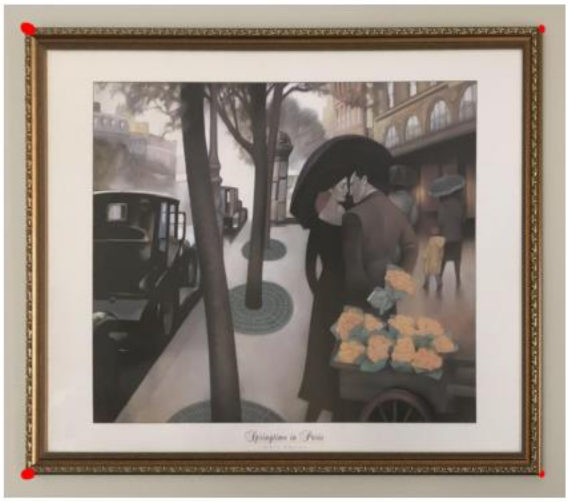

Figure 10: The final rectified image