DSA5203 Assignment 2 Technical Report

Name: Ong Jian Ying Gary

Student ID: A0155664X

This assignment 2 for DSA5203 requires us to write 1 main function within main.py, which warps an input image by a transform and returns the rectified image. The input image is always assumed to contain an object with rectangular boundary which is prominently present with a monochrome background. The result expected is to have rectified images with a boundary that is horizontal or along the vertical direction of the image.

To tackle this problem, we first define the workflow of how we might accomplish the task:

1. Take the input image, convert the input image to grayscale, so that the Canny edge detection algorithm used later can work on a single-channel image. We retain the intensity information while simplifying the image and reducing its size, making it easier for the algorithm to work.

```
    # convert input image to gray scale
    gray = cv2.cvtColor(im1, cv2.COLOR_BGR2GRAY)
```

2. As the input image given in the original code will give a scaled numpy array of the image, we will first convert the floating-point grayscale image back to 8-bit unsigned integer for cv2 operations.

```
    # Convert the floating-point grayscale image back to 8-bit unsigned
integer
    gray_8bit = (gray * 255).astype(np.uint8)
```

3. Next, we apply common pre-processing step on the grayscale image by applying a gaussian blur. Image noise can often create false edges, which negatively impacts the performance of Canny edge detection. By smoothing, we reduce high-frequency noise while preserving low-frequency features such as edges. This results in more accurate & reliable representation of edges in both test images.

```
    # apply a gaussian blur to reduce noise
    blurred = cv2.GaussianBlur(gray_8bit, (5, 5), 0)
```

4. Apply Canny edge detection on blurred image, with minVal = 75, maxVal=200 after testing on both images for best results.

```
    # apply canny edge detection on blurred image
    edged = cv2.Canny(blurred, 75, 200)
```

5. For test1.jpg image, we observe that it is a tricky image with many broken lines when using cv2.findContours() function, as it includes a picture within a frame, the true outer edge of the frame is not detected as the largest contour area. Hence, we will use a morphological

operation know **'closing'** to aid with broken lines being detected for the outer frame of test1.jpg when using findContours. Closing morphological operation will help to close small gaps and join broken edge lines. As the main goal is to connect the broken lines, applying a image dilation followed by erosion will help.

a. Dilation will help to increase the object area of the frame and can accentuate features.
b. Erosion wil erode the boundaries of the foreground object and is used to diminish the features of an image.

A kernel of 3x3 of ones is convolved with the image for both dilation and erosion with iterations set at 3, and the results are satisfactory for the contour lines of test1.jpg for the outer image frame as the broken edge lines have been detected successfully after the closing operation.

```
    # since test1.jpg is a tricky image to detect with many broken lines in
the contours, we need to do some morphological operations
    # perform morphological operations (dilation followed by erosion -
closing) This operation helps to close small gaps and join broken edge lines.
    kernel = np.ones((3,3), np.uint8)
    dilated = cv2.dilate(edged, kernel, iterations=3)
    closed = cv2.erode(dilated, kernel, iterations=3)
```

6. Perform findContours with cv2 package to return a list of contours in the image

```
    # contours are found using findContours, returning a list of contours in
the image
    contours, _ = cv2.findContours(closed, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

7. As mentioned in the hw2.pdf, we will use the largest contour out of all contours found by taking the maximum area.

```
    # the largest contour can be found using the MAX area, since the homework
says the input image will always
    # have a prominent rectangular boundary in a monochrome background.
    largest_contour = max(contours, key=cv2.contourArea)
```

8. Next, we use cv2.arcLength to find the perimeter of the largest contour. The epsilon parameter is calculated by multiplying the perimeter of the largest contour by 2%, which is being used to control the approximation accuracy in approxPolyDP.

```
    # arcLength is used to find the perimeter of the largest contour found.
Multiply by 0.02 to obtain epsilon,
    # which is used to control the approximation accuracy in approxPolyDP.
    epsilon = 0.02 * cv2.arcLength(largest_contour, True)
```

9. We extract the 4 corner points of the largest quadrilateral contour within the image. By simplifying the contour using cv2.approxPolyDP(), it is easier to isolate the 4 corner points, which will be used for perspective transformation.

```
    # approximates the contour shape. Result is a simplified contour that
retains the general shape but with fewer vertices.
    # used to extract the 4 corner points
    approx_corners = cv2.approxPolyDP(largest_contour, epsilon, True)
```

10. We should have 4 corner points from the quadrilateral object, we will proceed to perform ordering of the corner points. Ordering corner points before perspective transform is essential because it determines how the source quadrilateral is mapped to the destination rectangle. They must be mapped properly to ensure the resulting rectified image is correctly transformed and preserves the expected orientation and aspect ratio. For this, order_corners function is defined as a supporting function:

```
def order_corners(corners):
    """
    Define a function to take in a set of 4 corner points (2D coordinates) and
order them clockwise,
    starting from the top-left corner. Ordering is required to ensure correct
perspective transformation
    when using cv2.getPerspectiveTransform() later.
    """
    # initialize an empty 4x2 numpy array to store the ordered corner points
    rect = np.zeros((4, 2), dtype=np.float32)

    # sum the x and y coordinates for each corner point
    s = corners.sum(axis=1)
    rect[0] = corners[np.argmin(s)] # represents corner with smallest sum of x
and y coordinates - top left corner
    rect[2] = corners[np.argmax(s)] # represents corner with largest sum of x
and y coordinates - bottom right corner

    # calculate the diffrence between x and y coordinates for each corner
point
    diff = np.diff(corners, axis=1)
    rect[1] = corners[np.argmin(diff)] # corner with smallest difference - top
right corner
    rect[3] = corners[np.argmax(diff)] # corner with largest difference -
bottom left corner

    return rect
```

11. After we achieve the corner points that are ordered in clockwise direction, we can perform rectification on the object. We define a new function rectify_image, which takes in a image,

the 4 corner points, a predefined rectified size of image desired – set as (1200,1000) by default, and some padding – set as 100 pixels by default. Padding is added to expand the output rectangle, so as to ensure that the rectified image will not only include the object of interest but also some background pixels. It ensures that the image output is not just the object of interest, so that we avoid losing contextual information after performing rectification. We first unpack the rectified_size parameter to get the width and height of the destination rectangle, then define the output rectangle corner points with some padding. Next, we will calculate the perspective transform matrix M by using the input corner points, and the output destination corner points. We apply perspective transform using matrix M on the image to return the rectified image.

```
def rectify_image(image, quadrilateral_points, rectified_size=(1200,1000),
padding=100):
    """
    Takes an input image, the corner points of the found quadrilateral, the
desired output size and default padding
    and returns an output
    """
    # unpack the rectified default size into width and height
    width, height = rectified_size

    # define the output rectangle's corner points, with padding, since we do
not just want to extract the object of interest
    output_rectangle_points = np.float32([[0 + padding, 0 + padding],
                                          [width - padding, 0 + padding],
                                          [width - padding, height -
padding],
                                          [0 + padding, height - padding]])


    # calculate the perspective transform matrix M using the input
quadrilateral corner points and output rectangle corner points
    # This is same as in the lecture notes for matrix H, and we use this to
transform the input image into the rectified image.
    M = cv2.getPerspectiveTransform(np.float32(quadrilateral_points),
output_rectangle_points)

    # Apply the perspective transform matrix M to the input image using
warpPerspective
    rectified_image = cv2.warpPerspective(image, M, (width, height))

    # Returns the rectified image as a numpy array
    return rectified_image
```

12. Some visualization results are illustrated below for both test1.jpg and test2.jpg and the resulting output images:

Figure 1: Contour lines and corner points detected in original image (test1.jpg)

Rectified Image Output

Figure 2: Rectified image output for test1.jpg

Figure 3: Contour lines and corner points detected in original image (test2.jpg)

Rectified Image Output

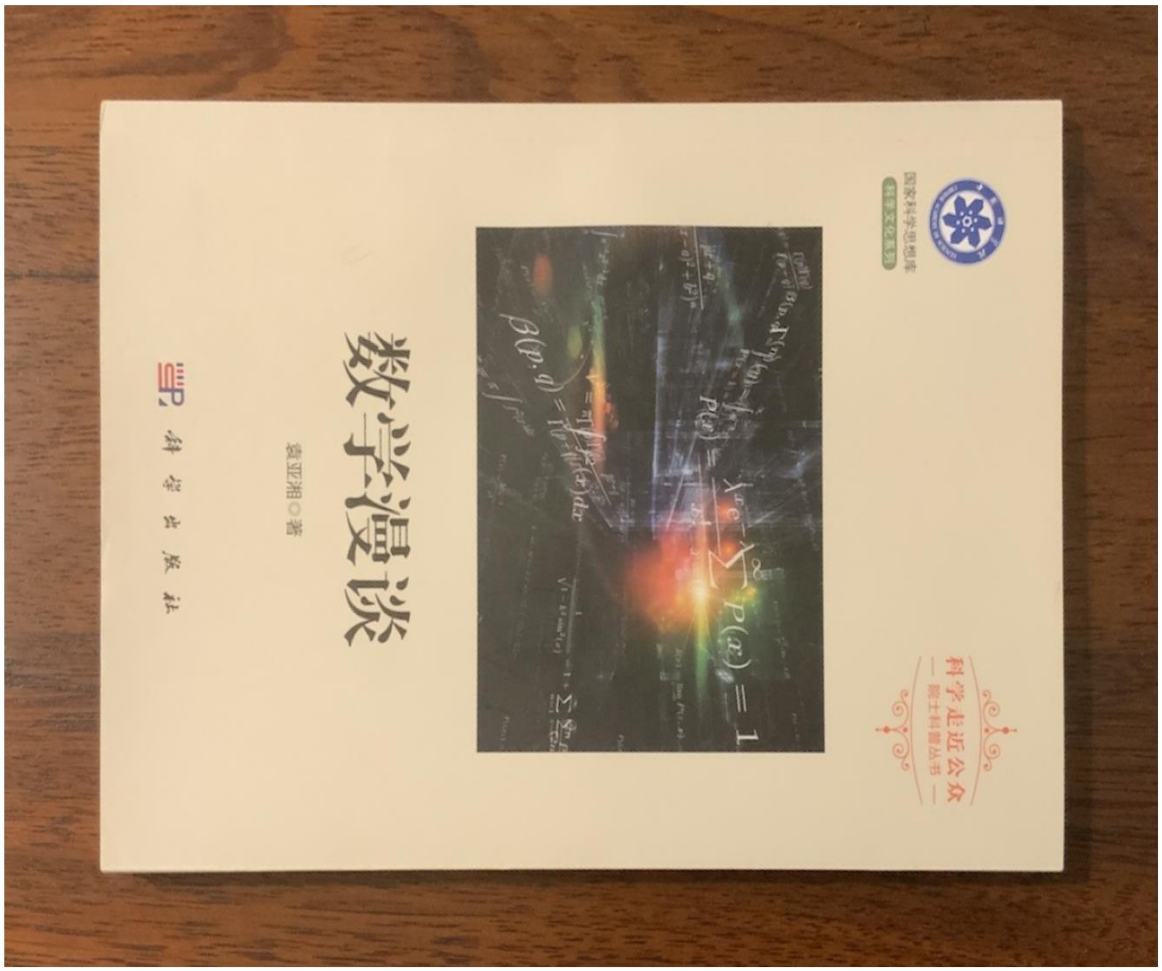

Figure 4: Rectified image output for test2.jpg

13. From the results above, we can see that for the more challenging image test1.jpg, the outer frames are successfully detected as the largest contours, since the broken edge lines have been fixed by morphological operations. Background pixels are also shown here, since padding has been utilized.