Assignment-1 Report

1. Introduction

In this assignment, a rectangle detector is implemented for a set of sudoku puzzle images using OpenCV and python.

2. Problem Statement

To detect rectangles in a sudoku puzzle image following steps are used:

1. Parsing sudoku images
2. Doing preprocessing
    i. Convert to grayscale
    ii. Use gaussian blur
    iii. Take bitwise not to reverse black and white
    iv. Dilate
3. Get the corners of the sudoku image
    i. Use find contours method to get contours use the largest one. (assume it is the largest one)
    ii. Return the index of the corner points
4. Crop the image using corners and original image
    i. Use perspective transform methods to eliminate rotation
    ii. Return cropped image
5. Do the same preprocessing on cropped image with different kernels
6. Use canny edge detection
7. Use a modified Hough transform
    i. Use Hough lines method to find the lines
    ii. Discard the similar lines using methods
    iii. Use the remaining lines to draw on the image
8. Write the new image to the output file

3. Proposed Solution

In this section each operation is going to be explained first with its technical details. And the implemented code will be shown.

1. Gaussian Blur: Uses gaussian filter. A gaussian filter is an N*N convolution filter that weights of the pixels inside of its footprint based on the gaussian function:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The pixels of the filter footprint are weighted using the values got from the gaussian function therefore provides a blurring effect. To reduce computational complexity of a 2d gaussian function, two 1d gaussian functions are multiplied. Gaussian blur is a low-pass filter, it erases high frequency signals. In our case it is used to reduce noise

2. Bitwise not: In the case used in my implementation by using same image twice it made a 1 valued pixel to 0 valued pixel (1 not 1 = 0) thus inverting colors.

3. Dilate: This operation consists of convoluting an image A with some kernel (B), which can have any shape or size. The kernel B has a defined anchor point, usually being the center of the kernel. As the kernel B is scanned over the image, we compute the maximal pixel value overlapped by B and replace the image pixel in the anchor point position with that maximal value. This maximizing operation causes bright regions of an image to grow

4. Find contours: Contours are continuous points, having same color or intensity. Returns same colored points as a list as first parameter.

5. Perspective Transformation: Uses a 3x3 transformation matrix and a complex method.

6. Canny edge detection: Done In 4 steps
    i. Noise reduction: Done with gaussian filter in this case
    ii. Finding intensity gradient of the image: Sobel filter is used in both horizontal and vertical direction.

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

    iii. Non-maximum Supression: A full scan of the image is done to only keep edge pixels as 1. For this at every pixel, pixel is checked if it is a local maximum in its neighborhood in the direction of gradient.
    iv. Hysteresis Thresholding: This stage decides which edges are really edges and which are not. For this 2 values are used to determine edges. Min threshold and max threshold. If above max value for sure an edge. Below min is discarded. Between min and max values if it is connected to a sure edge it is concidired as an edge.

7. Hough Transform: The linear hough transform uses a 2d array, to detect a line described by $r = x\cos\theta + y\,sin\theta$. The dimension of the accumulator equals the number of unknown parameters. For each pixel at (x, y) and its neighborhood, the hough transform determines if there is enough evidince of a straight line at that pixel. If a line candidate is found its vote is incremented. The locations with high votes are determined as a line.

Implementation

1. Preprocessor:

```
def preprocessor(image):
    # reduce noise
    preprocess = cv.GaussianBlur(image, (9, 9), 0)

    # regional threshold yields a better value
    preprocess = cv.adaptiveThreshold(preprocess, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C,
cv.THRESH_BINARY, 11, 2)

    # reverse the 1`s
    preprocess = cv.bitwise_not(preprocess, preprocess)
    kernel = np.ones((5, 5), np.uint8)
    # get larger 1`s
    preprocess = cv.dilate(preprocess, kernel)
    return preprocess
```

2. Get_largest_corners:

```
def get_largest_corners(image):
    # return the pixels with same value
    contours, _ = cv.findContours(image, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)

    # sort the contours so the largest one is in in the first index
    contours = sorted(contours, key=cv.contourArea, reverse=True)

    # assume the biggest rectangle is the corners of the sudoku
    polygon = contours[0]

    # get the corners
    bottom_right, _ = max(enumerate([pt[0][0] + pt[0][1] for pt in polygon]), key=operator.itemgetter(1))
    top_left, _ = min(enumerate([pt[0][0] + pt[0][1] for pt in polygon]), key=operator.itemgetter(1))
    bottom_left, _ = max(enumerate([pt[0][0] - pt[0][1] for pt in polygon]), key=operator.itemgetter(1))
    top_right, _ = min(enumerate([pt[0][0] - pt[0][1] for pt in polygon]), key=operator.itemgetter(1))

    return [polygon[top_left][0], polygon[top_right][0], polygon[bottom_right][0], polygon[bottom_left][0]]
```

3. Crop_sudoku_puzzle

```
def crop_sudoku_puzzle(image, crop_rectangle):

#X = (ax + by + c) / (gx + hy + 1)        X, Y -> new cords|| x,y -> old coords || a..h -> constants
#Y = (dx + ey + f) / (gx + hy + 1)



    img = image
    crop_rect = crop_rectangle

    def distance_between(a, b):

        return np.sqrt(((b[0] - a[0]) ** 2) + ((b[1] - a[1]) ** 2))
```

```
#crops rectangular portion from image and wraps it into a square of similar size

    def crop_img():
        top_left, top_right, bottom_right, bottom_left = crop_rect[0],
crop_rect[1], crop_rect[2], crop_rect[3]

        source_rect = np.array(np.array([top_left, bottom_left, bottom_right,
top_right],
                                        dtype='float32'))

#get longest side in rectangle

        sides = max([
            distance_between(bottom_right, top_right),
            distance_between(top_left, bottom_left),
            distance_between(bottom_right, bottom_left),
            distance_between(top_left, top_right)
        ])

        dest_square = np.array([[0, 0], [sides - 1, 0], [sides - 1, sides - 1],
[0, sides - 1]], dtype='float32')

        #Skew the image by comparing 4 before and after points -- return matrix
        modified = cv.getPerspectiveTransform(source_rect, dest_square)

        #Perspective Transformation on original image
        return cv.warpPerspective(img, modified, (int(sides), int(sides)))

    return crop_img()
```

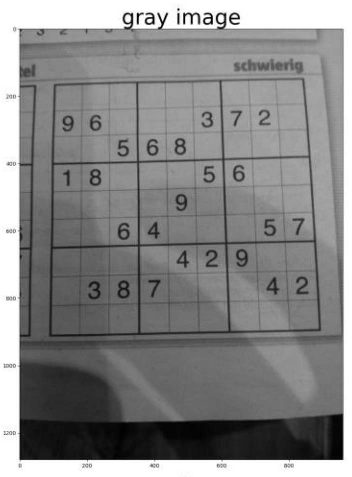
4. Hough Transform

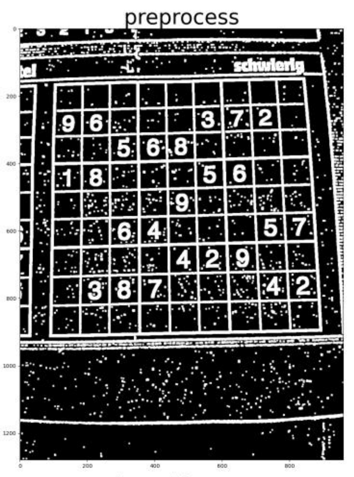
```
def hough_transform(image, hough_image):

    # get candidate lines
    lines = cv.HoughLines(hough_image, 1, np.pi / 180, 150)

    if lines is None:
        return image

    if filter:
        rho_threshold = 15
        theta_threshold = 0.1

        # how many lines are similar to a given one
        similar_lines = {i: [] for i in range(len(lines))}
        for i in range(len(lines)):
            for j in range(len(lines)):
                if i == j:
                    continue

                rho_i, theta_i = lines[i][0]
                rho_j, theta_j = lines[j][0]
                if abs(rho_i - rho_j) < rho_threshold and abs(theta_i - theta_j) <
theta_threshold:
                    similar_lines[i].append(j)

        # ordering the INDICES of the lines by how many are similar to them
        indices = [i for i in range(len(lines))]
        indices.sort(key=lambda x: len(similar_lines[x]))

        # line flags is the base for the filtering
        line_flags = len(lines) * [True]
        for i in range(len(lines) - 1):
            # if we already disregarded the ith element in the ordered list then
we don't care
            if not line_flags[indices[i]]:
                continue
            # we are only considering those elements that had less similar line
            for j in range(i + 1, len(lines)):
                # and only if we have not disregarded them already
                if not line_flags[indices[j]]:
                    continue

                rho_i, theta_i = lines[indices[i]][0]
                rho_j, theta_j = lines[indices[j]][0]
                if abs(rho_i - rho_j) < rho_threshold and abs(theta_i - theta_j) <
theta_threshold:
                    # if it is similar and have not been disregarded yet then drop
it now
                    line_flags[indices[j]] = False

    filtered_lines = []

    if filter:
        # filtering
        for i in range(len(lines)):
            if line_flags[i]:
                filtered_lines.append(lines[i])
    else:
        filtered_lines = lines

    # hough transform step
    for line in filtered_lines:
        rho, theta = line[0]
        a = np.cos(theta)
        b = np.sin(theta)
        x0 = a * rho
        y0 = b * rho
        x1 = int(x0 + 1000 * (-b))
        y1 = int(y0 + 1000 * a)
        x2 = int(x0 - 1000 * (-b))
        y2 = int(y0 - 1000 * a)

        cv.line(image, (x1, y1), (x2, y2), (0, 0, 255), 3)

    return image
```

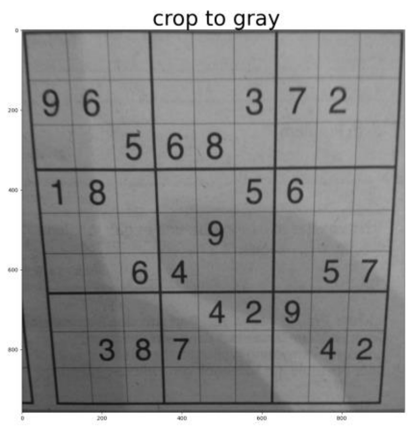
4. Results

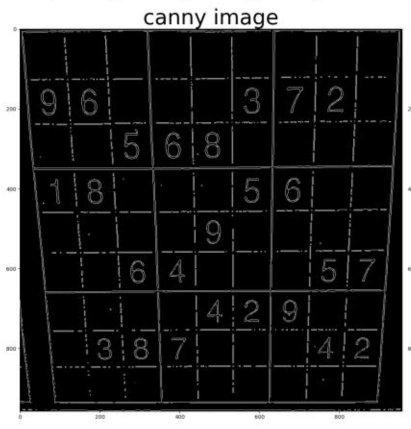
Here are some results I got using this algorithm

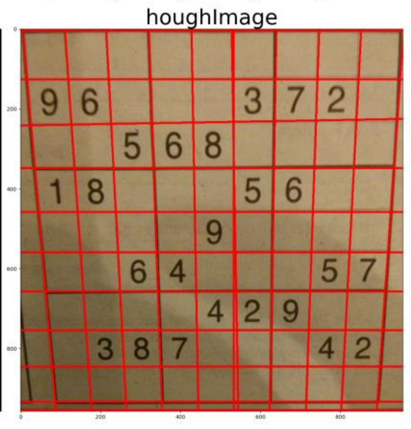
These are the good results:

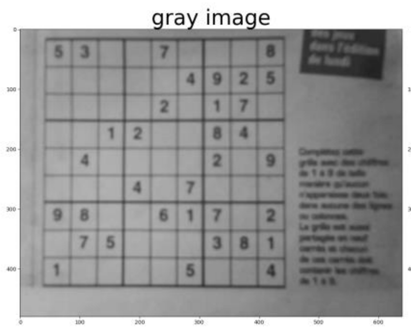


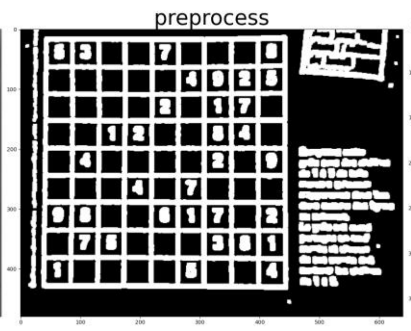


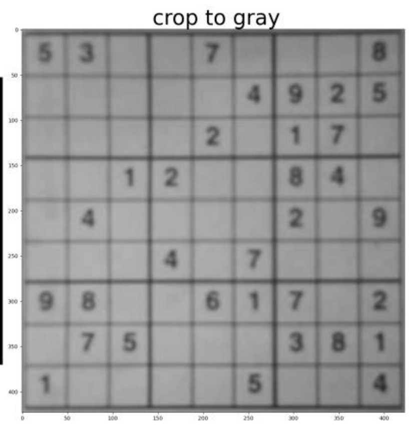


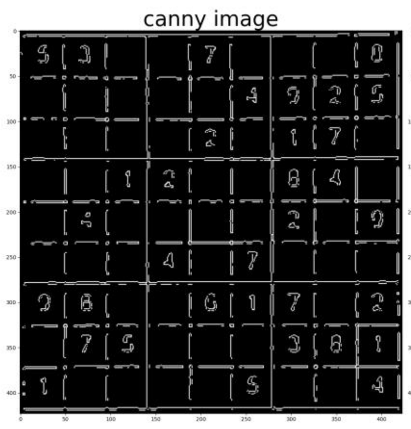


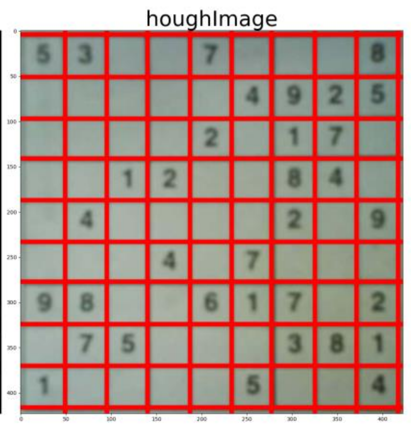












And some bad results: