Project3 Report

Wenbo Du

Readme

All input images are in the folder named “samples”.
The generated images will be saved in the same folder as the code.
The image named ‘output image.jpeg’ is the final result.
After the calculation, press keyboard 0 to process the next image.

Method

The algorithm we proposed contains three parts: pre-processing module, processing module and post-processing module.

a. Pre-processing module

We first convert the original image into RGB format:

```
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
print(frame.shape)
cv2.imshow("RGB image", frame)
cv2.waitKey(0)
```

The third step is to call function named detect_corner to process the image:

```
detections_in_frame = detect_corner(frame, height, width)
```

```
def detect_corner(frame, height, width):
```

b. Processing module

First, we convert RGB image into HLS format and separate s channel:

```
frame_0 = cv2.cvtColor(frame, cv2.COLOR_RGB2HLS)
h_channel = frame_0[:, :, 0]
cv2.imshow("h_channel image", h_channel)
cv2.waitKey(0)
l_channel = frame_0[:, :, 1]
cv2.imshow("l_channel image", l_channel)
cv2.waitKey(0)
s_channel = frame_0[:, :, 2]
cv2.imshow("s_channel image", s_channel)
cv2.waitKey(0)
```

Next, we use functions named cv2.kmeans and cv2.threshold to do the Optimal Thresholding to the s channel:

```
Z = s_channel.reshape((-1, 1))
# convert to np.float32
Z = np.float32(Z)
# define criteria, number of clusters(K) and apply kmeans()
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
K = 3
ret, label, center = cv2.kmeans(Z, K, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
# Now convert back into uint8, and make original image
center = np.uint8(center)
res = center[label.flatten()]
res2 = res.reshape((s_channel.shape))
cv2.imshow("threshold after s_channel image", res2)
cv2.waitKey(0)
print(center)
max_num = max(center[0][0], max(center[1][0], center[2][0]))
min_num = min(center[0][0], min(center[1][0], center[2][0]))
print(max_num)
print(min_num)
for i in range(K):
    if center[i][0] != max_num and center[i][0] != min_num:
        mid_num = center[i][0]
ret, binary_0 = cv2.threshold(res2, min_num, 255, cv2.THRESH_BINARY)
print(binary_0)
cv2.imshow("threshold after s_channel image", binary_0)
cv2.waitKey(0)
```

After that, we convert RGB image into gray level image:

```
frame_0 = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
```

Now, we introduce Sobel detection by using three different types of cv2.Sobel functions:

```
edges = cv2.Sobel(frame_0, cv2.CV_16S, 1, 1)
edges = cv2.convertScaleAbs(edges)
cv2.imshow("sobel image", edges)
cv2.waitKey(0)

edgesh = cv2.Sobel(frame_0, cv2.CV_16S, 1, 0)
edgesh = cv2.convertScaleAbs(edgesh)
cv2.imshow("sobel horizontal image", edgesh)
cv2.waitKey(0)

edgesv = cv2.Sobel(frame_0, cv2.CV_16S, 0, 1)
edgesv = cv2.convertScaleAbs(edgesv)
cv2.imshow("sobel vertical image", edgesv)
cv2.waitKey(0)
```

The we first combine last two images:

```
grad = cv2.addWeighted(edgesh, 0.5, edgesv, 0.5, 0)
cv2.imshow("vertical + horizontal image", grad)
cv2.waitKey(0)
```

Then we combine the first image and the new image:

```
gradd = cv2.addWeighted(grad, 0.7, edges, 0.3, 0)
cv2.imshow("sobel + vertical + horizontal image", gradd)
cv2.waitKey(0)
```

Thirdly, we do the Gaussian blur by using cv2.GaussianBlur function:

```
kernel_size_temp = round(((height * width) / (80 * 80)) ** 0.5)
if kernel_size_temp % 2 == 0:
    kernel_size_temp = kernel_size_temp + 1
kernel_size = max(9, kernel_size_temp)
print(kernel_size)
frame_1 = cv2.GaussianBlur(gradd, (kernel_size, kernel_size), 1)
cv2.imshow("GaussianBlur image", frame_1)
cv2.imwrite('GaussianBlur image.jpeg', frame_1)
```

The kernel size is self-adaptive to the image size. If the image size is large, then the kernel size will also be relatively large.
After doing the Gaussian blur, we continue do the median blur to the image by using cv2. medianBlur function:

```
kernel_size_temp = round(((height * width) / (180 * 180)) ** 0.5)
if kernel_size_temp % 2 == 0:
    kernel_size_temp = kernel_size_temp + 1
kernel_size = max(5, kernel_size_temp)
print(kernel_size)
frame_2 = cv2.medianBlur(frame_1, kernel_size)
cv2.imshow("medianBlur image", frame_2)
cv2.waitKey(0)
print('frame_2', frame_2)
```

The kernel size is also self-adaptive.
Then, we use cv2.Canny to do the Canny detection:

```
low_bound = 100
upper_bound = 150
frame_3 = cv2.Canny(frame_2, low_bound, upper_bound)# + cv2.Canny(s_channel, low_bound, upper_bound)
cv2.imshow("Canny image", frame_3)
cv2.imwrite('Canny image.jpeg', frame_3)
```

And we use cv2. HoughLinesP to detect lines:

```
lines_0 = cv2.HoughLinesP(frame_3, rho=1, theta=np.pi/180, threshold=round((height + width) / 20),
                          minLineLength=round((height + width) / 20), maxLineGap=round((height + width) / 20))
```

Now, we add detected lines to the blank image whose size is equal to the original image:

```
lines = []
if lines_0 is not None:
    lines.extend(lines_0)
#if lines_1 is not None:
#    lines.extend(lines_1)
print(type(lines_0))
frame_blank = np.zeros((height,width,3), np.uint8)
if lines is not []:
    for line in lines:
        for x1, y1, x2, y2 in line:
            cv2.line(frame_blank, (x1, y1), (x2, y2), color=(255, 255, 255), thickness=2)
cv2.imshow("Result image", frame_blank)
cv2.imwrite('Result image.jpg', frame_blank)
```

After that, we convert previous image into gray level image and detect Harris corners. Besides, we also add a threshold to filter some noise points:

```
gray = cv2.cvtColor(frame_blank, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (7, 7), 0)
cv2.imshow('Gaussian', gray)
gray = np.float32(gray)
# harris detection
dst = cv2.cornerHarris(gray, 2, 3, 0.02)
# dst = cv2.dilate(dst, None)
# dst = cv2.erode(dst, None)
# threshold
frame_blank[dst > 0.0001 * dst.max()] = [225, 0, 0]
cv2.imshow('Harris', frame_blank)
```

Then we create a mask to filter Harris corners that are outside the check:

```
gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (7, 7), 0)
ret, mask = cv2.threshold(gray, 100, 255, cv2.THRESH_BINARY)
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (9, 9))
# mask = cv2.erode(mask, kernel)
mask = cv2.dilate(mask, kernel)
cv2.imshow('mask', mask)
mask_inv = cv2.bitwise_not(mask)
img1_bg = cv2.bitwise_and(frame, frame, mask=mask_inv)
img2_fg = cv2.bitwise_and(frame_blank, frame_blank, mask=mask)
dst = cv2.add(img1_bg, img2_fg)
cv2.imshow('mask_result', dst)
```

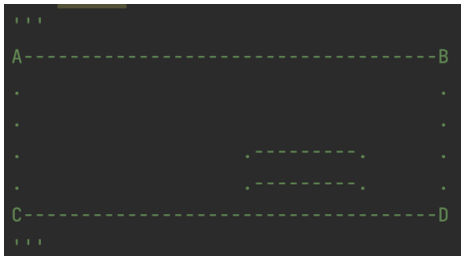
Now, we are able to calculate locations of four corner. The method is just calculate the distance between the each Harris corner to each picture corner and select the point that has the minimal value:

```
for i in range(0, width):
    for j in range(0, height):
        if dst[j][i][0] == 225 and dst[j][i][1] == 0 and dst[j][i][2] == 0:
            distance_leftup = math.pow((j - 0), 2) + math.pow((i - 0), 2)
            distance_leftup = math.sqrt(distance_leftup)
            if distance_leftup < leftup_min:
                leftup_min = distance_leftup
                leftup_coor = [i, j]
            distance_rightup = math.pow((j - 0), 2) + math.pow((i - width), 2)
            distance_rightup = math.sqrt(distance_rightup)
            if distance_rightup < rightup_min:
                rightup_min = distance_rightup
                rightup_coor = [i, j]
            distance_leftdown = math.pow((j - height), 2) + math.pow((i - 0), 2)
            distance_leftdown = math.sqrt(distance_leftdown)
            if distance_leftdown < leftdown_min:
                leftdown_min = distance_leftdown
                leftdown_coor = [i, j]
            distance_rightdown = math.pow((j - height), 2) + math.pow((i - width), 2)
            distance_rightdown = math.sqrt(distance_rightdown)
            if distance_rightdown < rightdown_min:
                rightdown_min = distance_rightdown
                rightdown_coor = [i, j]
```

Then we replace four corners with circles:

```
cv2.circle(dst, leftup_coor, 9, (0, 0, 255), thickness=8)
cv2.circle(dst, rightup_coor, 9, (0, 0, 255), thickness=8)
cv2.circle(dst, leftdown_coor, 9, (0, 0, 255), thickness=8)
cv2.circle(dst, rightdown_coor, 9, (0, 0, 255), thickness=8)
cv2.imshow('corners result', dst)
cv2.imwrite('corners result.jpeg', dst)
```

The final step is to calculate the angel of the check. If AB + CD > AC + BD, then the size of output image is 1000X500. Otherwise, the size will be 500X1000.



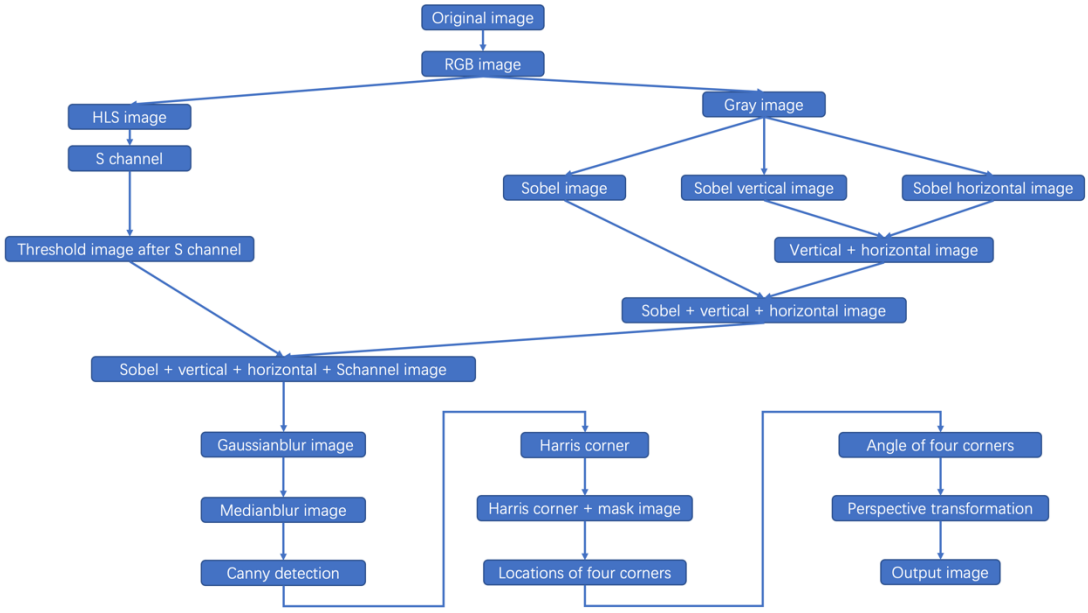
```
distance_AB = math.pow((leftup_coor[0] - rightup_coor[0]), 2) + math.pow((leftup_coor[1] - rightup_coor[1]), 2)
distance_AB = math.sqrt(distance_AB)
distance_CD = math.pow((leftdown_coor[0] - rightdown_coor[0]), 2) + math.pow((leftdown_coor[1] - rightdown_coor[1]), 2)
distance_CD = math.sqrt(distance_CD)

distance_AC = math.pow((leftup_coor[0] - leftdown_coor[0]), 2) + math.pow((leftup_coor[1] - leftdown_coor[1]), 2)
distance_AC = math.sqrt(distance_AC)
distance_BD = math.pow((rightup_coor[0] - rightdown_coor[0]), 2) + math.pow((rightup_coor[1] - rightdown_coor[1]), 2)
distance_BD = math.sqrt(distance_BD)

if (distance_AB + distance_CD) > (distance_AC + distance_BD):
    result_width = 1000
    result_height = 500
else:
    result_width = 500
    result_height = 1000
```

c. Post-processing module

Now, we are able to introduce functions named getPerspectiveTransform and warpPerspective to calculate the final output image:

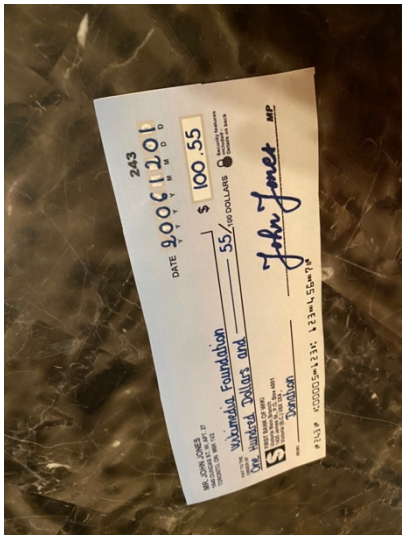
```
src_points = np.array([leftup_coor, rightup_coor, leftdown_coor, rightdown_coor], dtype="float32")
dst_points = np.array([[0., 0.], [result_width, 0.], [0., result_height], [result_width, result_height]], dtype="float32")

M = cv2.getPerspectiveTransform(src_points, dst_points)
perspective = cv2.warpPerspective(frame, M, (result_width, result_height), cv2.INTER_LINEAR)

perspective = cv2.cvtColor(perspective, cv2.COLOR_RGB2BGR)
cv2.imshow("output image", perspective)
cv2.imwrite('output image.jpeg', perspective)
```

After the calculation, press keyboard 0 to process next image.
The algorithm flowchart is shown below:

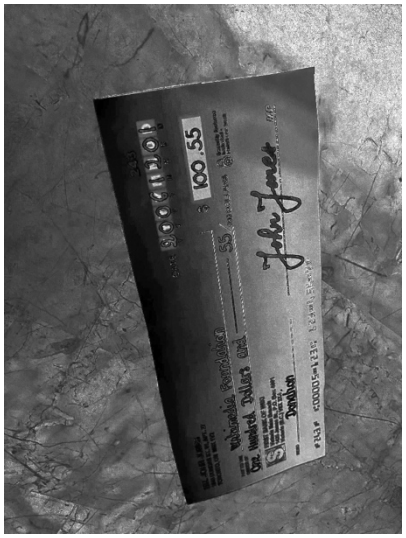


Experiment Results

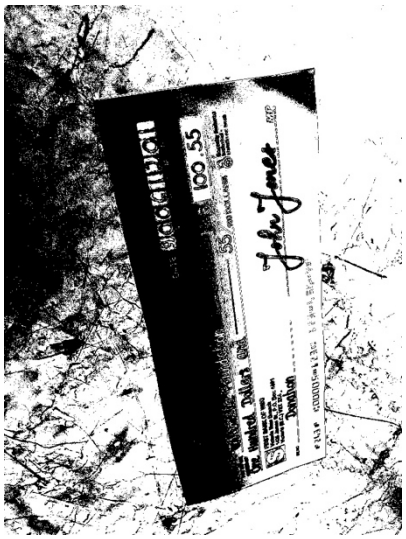
The original image is shown below:



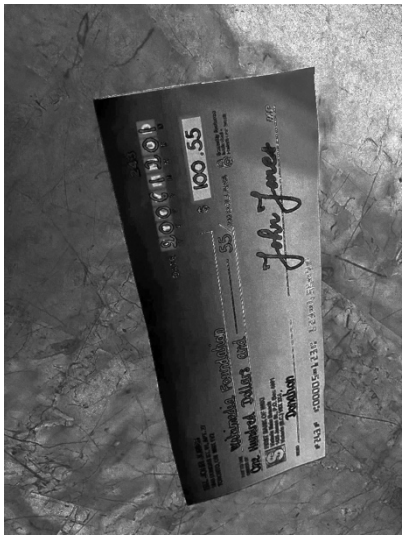
The s channel of HLS image is shown below:



The image after Optimal Thresholding is shown below:



The gray image is shown below:



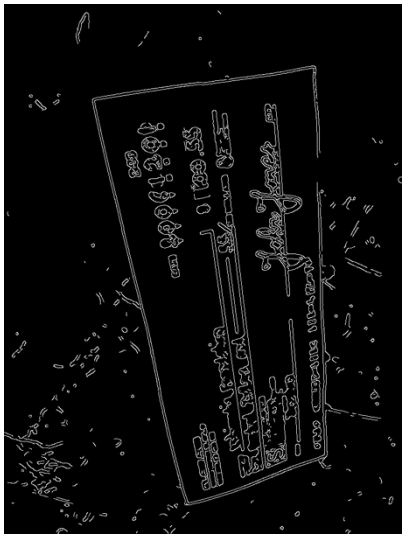
Images after three different type of Sobel detection are shown below:

The combined images are shown below:

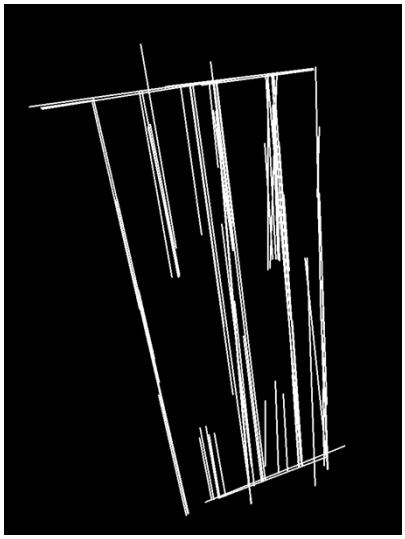
The image after Gaussian blur is shown below:

The image after median blur is shown below:

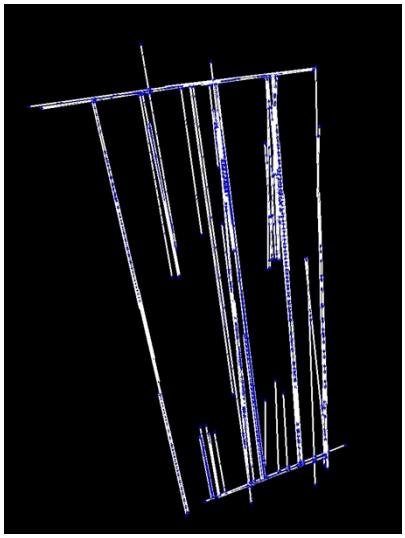
The image after Canny detection is shown below:



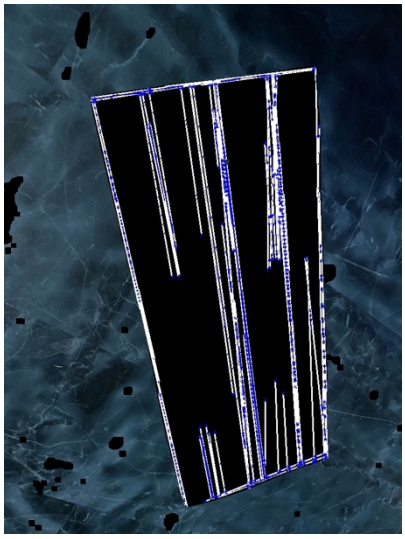
The image after HoughLines detection on a blank image is shown below:



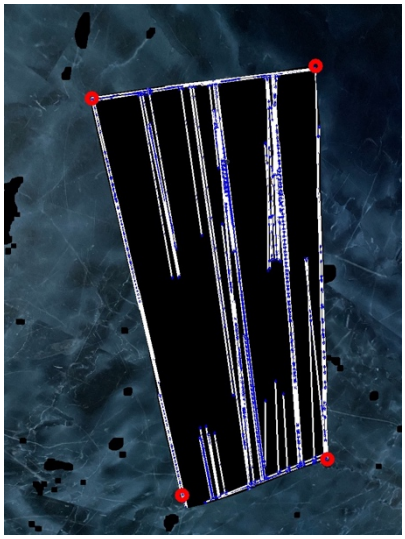
The Harris corners are shown below:



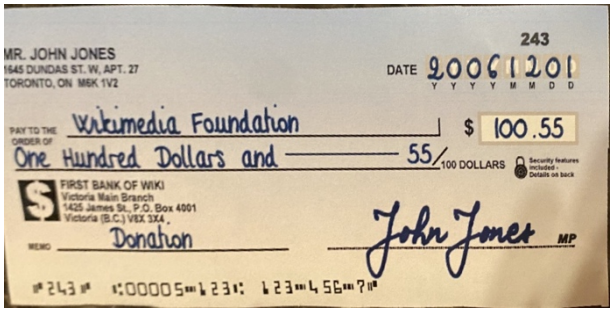
The image after putting a mask is shown below:



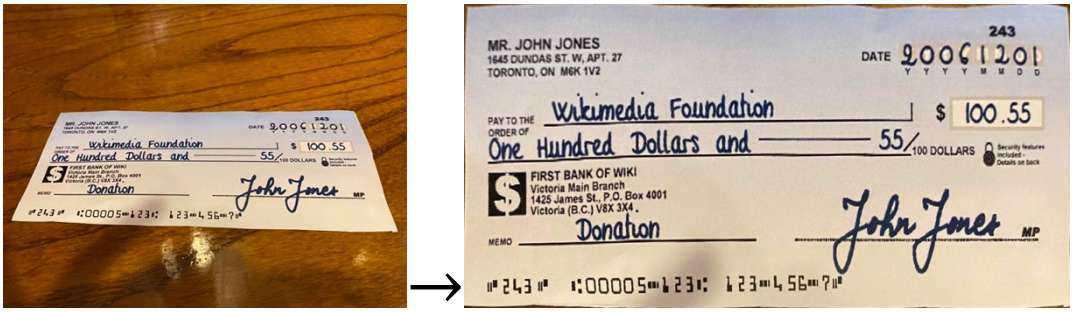
The image after locating four corners is shown below:



The final output image is:



Another example is shown below:



Future Work

In the future, we plan to introduce deep learning module to optimize the performance of the algorithm and realize the function of text extraction.