Report of Computer Vision with OpenCV

(Project 2, part 1)[1]

Author: ZHUO Xu (卓旭); Email: zhuoxu@seu.edu.cn

1. Read Image

Read the image using OpenCV and convert it into grayscale. The result is

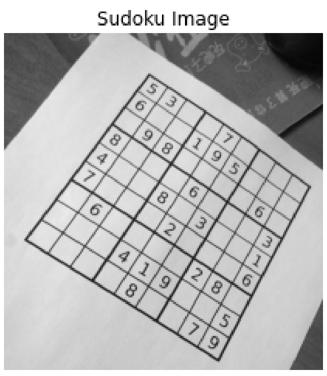


Figure 1 – Grayscale Sudoku Image.

2. Sudoku Puzzle Detection

We first reduce the noise in the image using a median blur with 5×5 window.

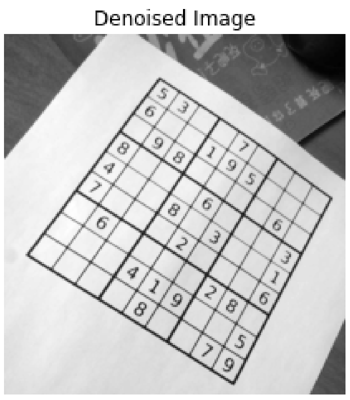


Figure 2 – Denoised Image.

Then, using Adaptive Threshold to convert it into binary image. Compared to *cv2.threshold* which requires a hard preset thresh, *cv2.adaptiveThreshold* can adaptively select a thresh for each patch in the image automatically. This is usually better, especially when the brightness condition is not uniform. After that, another 7×7 median blur is applied to further reduce the noise. The binary image is as follow.

[1] Corresponding code: *sudoku_part1.ipynb*; Opensource at github.com/z0gSh1u/rennes1-seu-cv .

Threshed Image

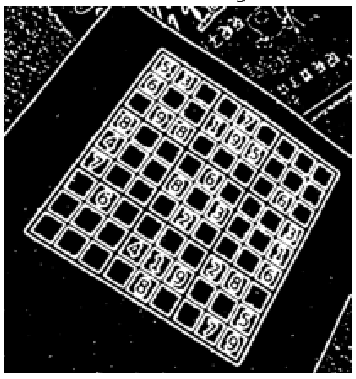


Figure 3 – Threshed Image.

Now we need to extract the 9×9 puzzle region. We can use *cv2.findContours* to extract the external contour of the puzzle by setting *mode* to *cv2.RETR_EXTERNAL*. And set method to *cv2.CHAIN_APPROX_SIMPLE* to reduce unnecessary contour points. From all the contours detected, we can use the prior knowledge that the puzzle region has largest area (this might not be very robust). So that we sort the contours by *cv2.contourArea* and use *cv2.drawContours* to draw the one with largest area. The result is as follow.

Contour of Puzzle

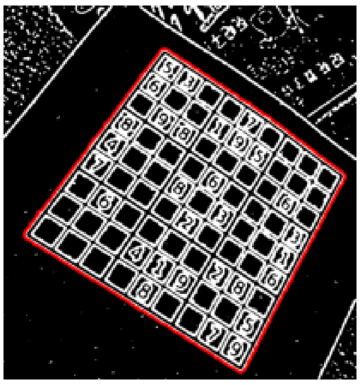


Figure 4 – Extracted Contour of the Puzzle (red line).

3. De-skew the Puzzle

We need to first identify the coordinates of four corners of the puzzle. To do this, we can use *cv2.approxPolyDP* to fit the contour. This function approximates a contour with a polygon. Note that it cannot be forced to return exactly four points (i.e., using a quadrilateral to fit). So, I choose to iterate the *eps* parameter in $0.01i \times p$, where $0.01i \in [0,1], i \in \mathbb{Z}$, and p is the perimeter of the contour. This parameter controls the accuracy (or tolerance) of winding the edges of the polygon. We use the result with exactly four points as corners of the puzzle (this might not be very robust).

Mark the corners, we get the following image:

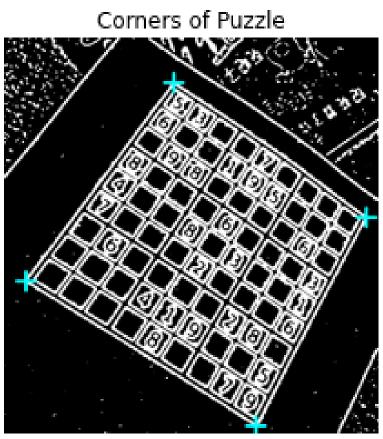


Figure 5 – Corners of the Puzzle (cyan crosshairs).

Now, we can finally de-skew the image. The main idea is to calibrate connecting lines of corners to be horizontal or vertical. To do this, we need to determine a perspective transform matrix using *cv2.getPerspectiveTransform* to map four corners into (0, 0), (0, *h-1*), (*w-1*, 0) and (*w-1*, *h-1*) (expressed in OpenCV's (*x*, *y*) formation). Since the puzzle is square, we simply choose the longest edge in the skewed puzzle as *w* and *h*. A problem is that how to correspond the skewed corners with de-skewed ones. A simple but effective and robust way introduced in [1] is the rule that

The bottom-right point has largest *col+row* ;
The top-left point has smallest *col+row* ;
The bottom-left point has smallest *col-row* ;
The top-right point has largest *col-row* .

Note that this rule can also be used to extract four corners from the contour, instead of using *cv2.approxPolyDP*. By applying this rule, we can map every two points.

```
Skewed
 [[353  94]
 [754 375]
 [524 810]
 [ 45 508]]
Deskewed
 [[  0   0]
 [720   0]
 [720 720]
 [  0 720]]
```

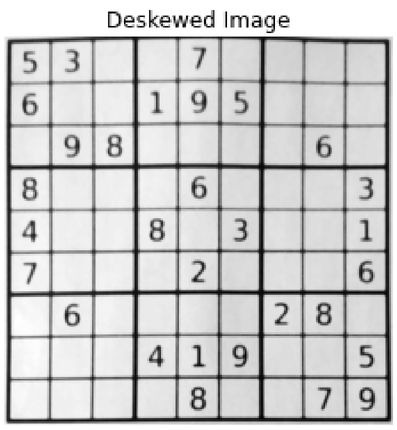
Figure 6 – The coordinates mapping.

Then we can generate the matrix of perspective transform. Since the image is 2-D, the perspective matrix is then 3×3.

```
array([[ 1.05034206e+00,  7.81413901e-01, -4.44223655e+02],
       [-8.68748406e-01,  1.23974417e+00,  1.90132236e+02],
       [-2.39205110e-04,  1.87496828e-04,  1.00000000e+00]])
```

Figure 7 – The perspective transform matrix.

Then, we can use *cv2.warpPerspective* to apply this transform onto the (denoised) skewed image to get a calibrated one.


Deskewed Image

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 8 – De-skewed Puzzle.

References

[1] "Sudoku Solver using OpenCV and DL—Part 1", *Medium*, 2019. [Online]. Available: https://aakashjhawar.medium.com/sudoku-solver-using-opencv-and-dl-part-1-490f08701179.
[Accessed: 03- Jul- 2022].