CS764: Assignment 2

Report

Siddharth Saha 170100025 **Parth Shettiwar** 170070021 **Parikshit Bansal** 170050040

1 Affine Transformations

Affine transformation is a linear mapping method that preserves points, straight lines, and planes. Sets of parallel lines remain parallel after an affine transformation.

The objective of this exercise is to correct the given distorted image of a chessboard, assuming a planar axial shear distortion. The correspondence of the points is obtained by manually finding pixel coordinates of corner points using `cv2.imshow()` in the distorted image. In both parts, we used the `cv2.warpAffine()` OpenCV API which applied the input transform to the distorted image.

1.1 Manual Method

We first look at the OpenCV documentation of `cv2.warpAffine()` function:

$$\texttt{dst}(x, y) = \texttt{src}(\texttt{M}_{11}x + \texttt{M}_{12}y + \texttt{M}_{13}, \texttt{M}_{21}x + \texttt{M}_{22}y + \texttt{M}_{23})$$

Here `M` represents the 2×3 affine transformation matrix.

In this part, we leverage the definition of **shear transformation**. A broad idea is that it displaces each point in a fixed direction, by an amount proportional to its signed distance from the line parallel to that direction.

The key observation is that at least one point is anchored in a shear transformation. Thus, we can set $\texttt{M}_{13} = 0$ and $\texttt{M}_{13} = 0$ and expect to find a 2×2 linear mapping between input and output images(4 unknowns). We use the convention of Euclidean coordinates for both input and output pixel coordinates.

We only require 2 points of correspondence(smartly chosen*) to obtain the linear mapping. We use the idea of **Direct Linear Transform**. We have the relation

$$x' = H \times x$$

- H: 2×2 Direct Linear Transform matrix
- x': Output image pixel in euclidean coordinates
- x: Input image pixel in euclidean coordinates

In order to solve for H, we use the relation: $X' = H \times X$. Each of X and X' have two columns corresponding to the two chosen pixel coordinates.

$$H = X' \times X^{-1}$$

We finally append a 2×1 zero column to H to obtain the `M` matrix.

* For example, choosing input pixels as $(0, 0)^T$ and $(601, 61)^T$ gives us a singular matrix. We finally choose $(601, 61)^T$ & $(61, 601)^T$ and are able to obtain X^{-1}.

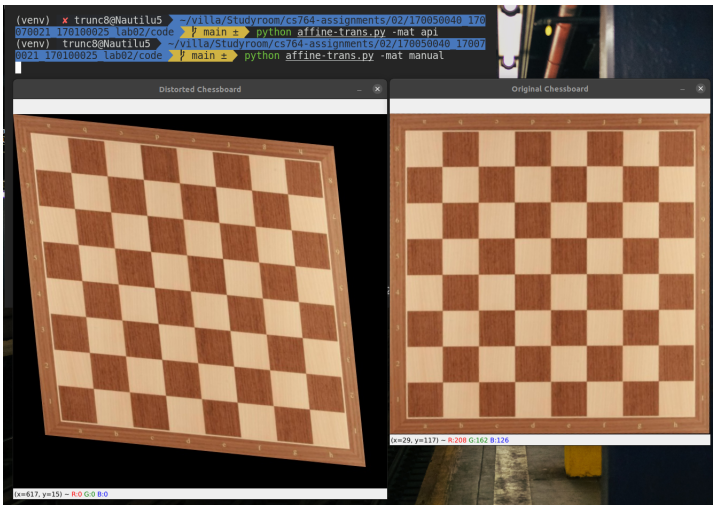


Figure 1: Output of manual method

1.2 API Method

In this part, we use the OpenCV API `cv2.getAffineTranformation()` to obtain the affine transformation matrix. We utilize 3 correspondent points to create the 2×3 transform matrix. The un-distortion results are seen to be identical. The key difference from *Part 1* is the requirement of an additional point as we could not incorporate the shear assumption here.

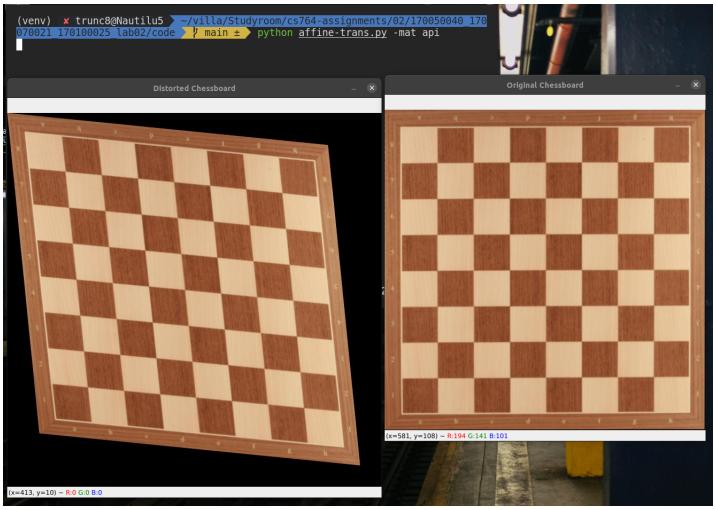


Figure 2: Output of api method

2 Perspective Transformations

The animation exits on pressing the key 'Q'

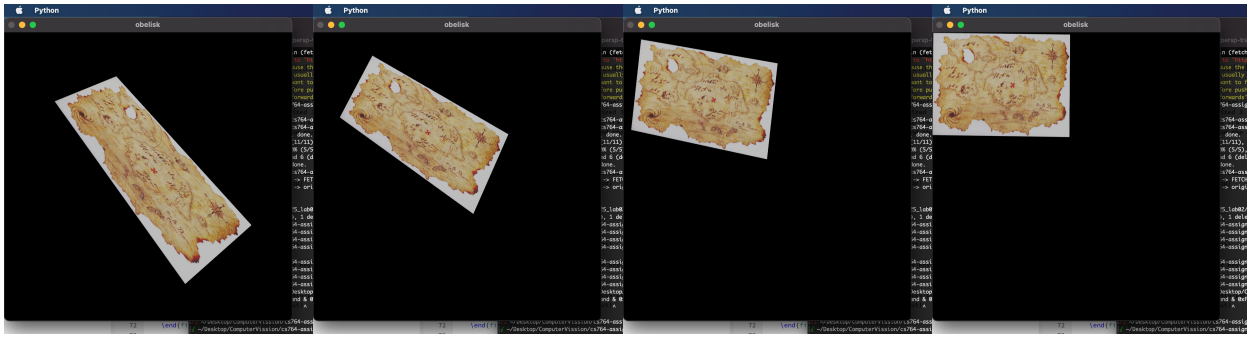


Figure 3: obelisk output

The transformation required here is homography which is inherently 3D and hence introduces that concept. Homography matrices have 8 degrees of freedom and hence we need 4 point correspondeces to uniquely identify the homography matrix, which we do by locating the 4 corner points of the obelisk. Though homography matrices are (3×3), they are ususally scaled leading to DoF of 8

There are multiple ways of going from view1 to view2. For example in our animation of the transformation we can go from the original image to any intermediate view and then finally to the required view. i.e. a homography matrix (3×3) can be decomposed as a product of 2 different homography matrices i.e. (3×3).

3 Document Scanner

In this part, the aim was to automate the process of document scanner. Given an input image captured from some camera angle, we had to produce the output image with the contents of document shown from top view. The following was the algorithm used for this process.

3.1 Algorithm

- First convert the RGB image to Grayscale image using `cv2.COLOR_BGR2GRAY`.
- Now we threshold the image and create a mask, so that we will have document separated from background. This step is important in later step of contour detection. It helps to avoid taking unnecessary contours from background. The thresholding is simply done by giving a lower value above which the pixel intensity of gray image should lie.
- Now we use `cv2.findContours` function to find all the contours in our thresholded image. This step will give all possible contours in our image. We pass a paramter `cv2.RETR_EXTERNAL` to this function, which would ensure that when we have contour inside another contour, it would take the outermmost contour. This helps to elimiate contours which might lie inside document since the outermost document contour will be taken.
- Now we extract the biggest contour in our image. This is based on the assumption that document object is biggest object in image and hence its contour would be biggest. This step is simply done by checking length and taking max of all possible contours got from previous step.

- Now having got the biggest contour, we need to determine the 4 corners of the document. This is simply done by iterating through all coordinates of the contour and taking those coordinates from following equation:

$$x1, y1 = \underset{c \in \mathcal{C}}{\text{argmax}}\ c_x + c_y$$
$$x2, y2 = \underset{c \in \mathcal{C}}{\text{argmax}}\ c_x - c_y$$
$$x3, y3 = \underset{c \in \mathcal{C}}{\text{argmax}}\ -c_x + c_y$$
$$x4, y4 = \underset{c \in \mathcal{C}}{\text{argmin}}\ c_x + c_y$$

  Here $\mathcal{C}$ is the list of coordinates of biggest contour got from previous step. This is based on assumption that our document is convex quadrilateral and hence its corners would satify the above property.

- Having received the 4 coordinates, we now use the function `cv2.getPerspectiveTransform` to get the transformation matrix.

- We use the `cv2.warpPerspective` function to do the transoformation based on the matrix got from previous step. We finally resize and show the output image.

Following are the images for some steps listed above.

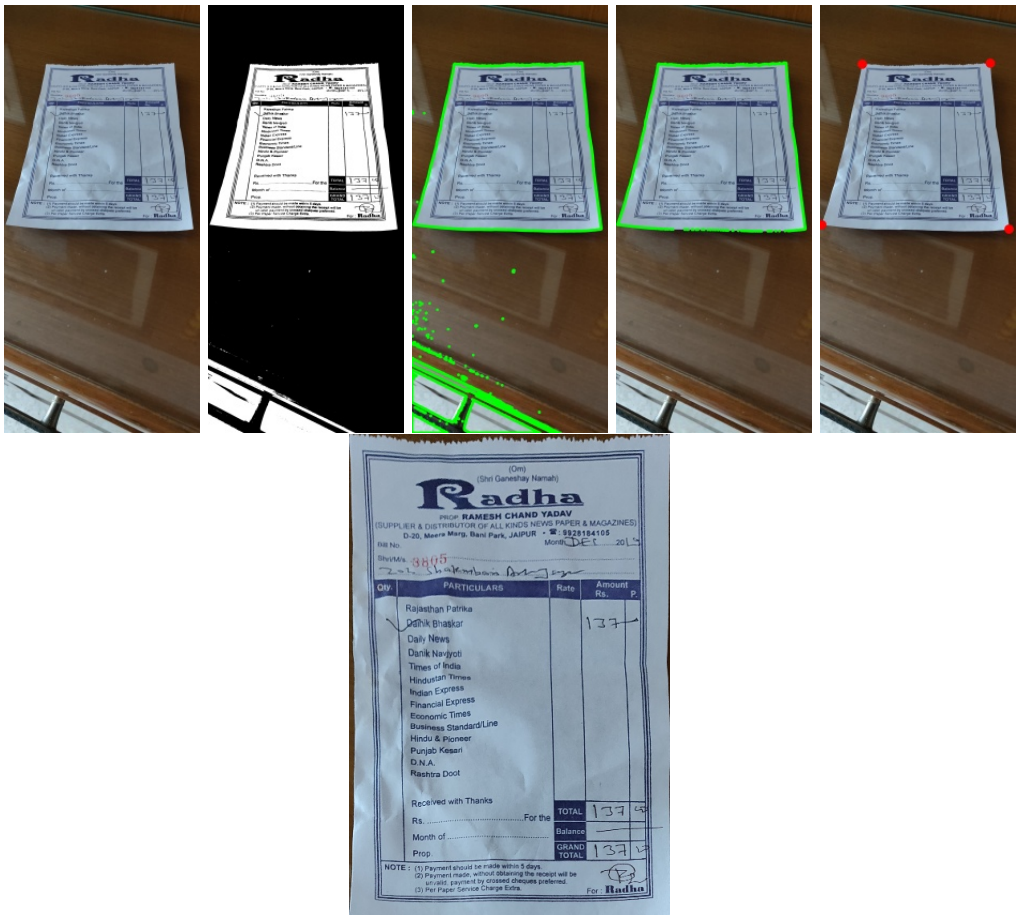
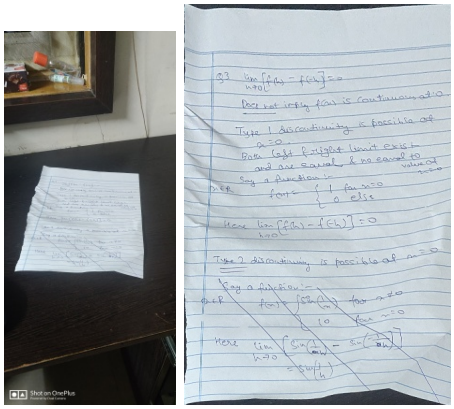


Figure 4: From left to right: Pic1 = Input image, Pic2: Image after thresholding, we see that background is blackened and document is shown as white. This helps in contour detection, Pic 3: We display all the contours found in image, marked by green boundary, we see other than document there is some contour outside document too, Pic4: We select the biggest contour, Pic5: We find the corners of document, Bottom Pic: Final Output

3.2 Results

We test the code on some our own self taken images and following are the results obtained:



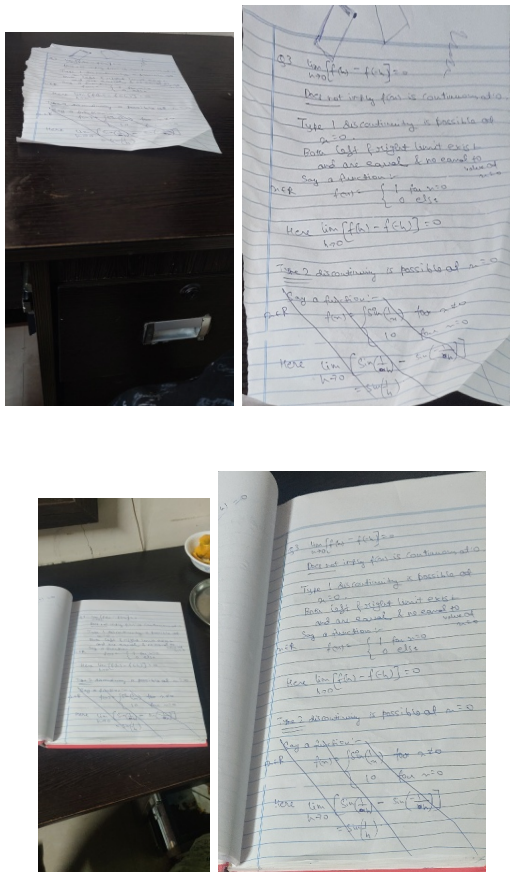


Figure 5: Left: Input image, Right: Output Image. We see that in first 2 cases, it gives correct output, however in third one, couldnt get the coordinates exactly from the image of the doc. This happened because the biggest contour in this case was including the left part of notebook. As in our algorithm, we threshold the image, hence left part of notebook also came and subsequently the biggest contour had this left part.

The algorihtm works on many cases. However whenever there is some background which has some contour bigger than document , it would fail. Again we tried to mitigate this by thresholding and this wont happen if backgorund is dark. Secondly, if we reduce the angle of view and deviate away from top view, like the second row input image in above case, after some time, it would also fail, since this time, even though the contour is found correctly, the 4 corners to find is relatively difficult and some wrong coordinates might be chosen for the transform.