Mobile Image Scanner

Ananya Mukherjee

1 Introduction

In this project I have developed an algorithm to design an image scanner for document images captured by a mobile camera. There are 4 parts in this implementation as below:

1. *Basic Scanning*: Implementation of geometric and appearance correction on document images captured by a mobile camera.
2. *Partial form scanning and stitching*: Combining partial form to produce single complete form.
3. *Reading filled forms*: Getting user's input in a filled form
4. *Optical Character Recognition*: Classifying characters present in the form

I have used Python's OpenCV, Tesseract (For OCR), etc libraries for this project.

2 Basic Scanning

This step involves implementing an algorithm to make geometric and appearance correction on document images. The process of basic scanning can be broadly summarized by the below block diagram.

Figure 1: Basic Scanning Algorithm

Given an image we preprocess the image and apply Canny Edge Detection. Then we clean the edges and find the paper corners. With the corner points available we can compute the perspective transform and apply it on the document image. Finally we apply the computed transform to out image and get the colored transformed image. If we need a grayscale image we then threshold the transformed image to get the desired grayscale image.

Let us discuss the blocks as shown in Fig. 1 involved in basic scanning individually.

2.1 Preprocessing

We first resize the given image to 500×500 and convert it to gray scale. Then we apply Gaussian bilateral filter for removal of noise and unwanted details and textures while preserving edges. Then we apply Canny edge detection algorithm for getting the edge information of the input image. Here we have used a used a low threshold so that no possible document edge is missed. Therefore the canny edges are somewhat noisy, which needs to be cleaned in the next stage. Fig. 2 outlines the preprocessing technique used and Fig. 3 shows the outputs of this stage.

2.2 Cleaning of Edges

Now we do a connected component analysis on the edges we got from preprocessing. We find connected components on the edges (using the connectedComponentsWithStats function in OpenCV)

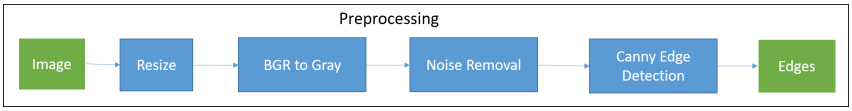


Figure 2: Preprocessing Input Image

(a) Grayscale, resized Image (b) Smoothened Image (c) Edges of input image

Figure 3: Sample outputs of preprocessing stage

and only retain those with large number of pixels and large length/width. Fig. 5 shows the outputs of this stage

1. Large no. of pixels $\rightarrow$ small noise edges are killed
2. Large length/width $\rightarrow$ the document edges are longer compared to other edges

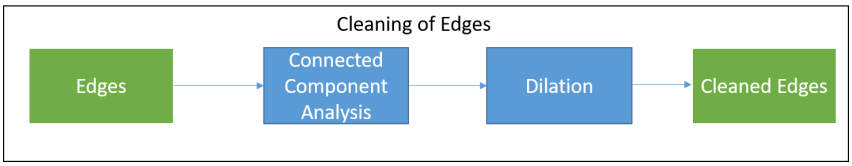


Figure 4: Cleaning Edges

2.3 Finding Paper Corners

Once we have the cleaned edges we now need to find the corner of the paper. For this we first find the largest contour and approximate the contour by a polynomial using the OpenCV approxPolyDP function. If the polynmial is a quadrilateral then we have found our corners, if its not a quadrilateral we then try to find the corners using either of the below two approaches. Fig. 6 broadly summarizes this stage. Fig. 7 shows the output for the corners detected by polynomial approximation

1. *Hough Lines Clustering*: First we detect straight lines on the largest contour using Hough transform. It is expected that there will be 4 dominant lines (corresponding to the 4 edges of the paper). Then all the lines are clustered into 2 clusters based on slope, and then each slope cluster is clustered based on their intercepts. This gives us 4 lines, whose intersections are found. These are potential paper corner points. Fig. 8 shows the output of this stage.
2. *Minimum Area Rectangle*: The smallest rectangle that encloses the largest contour is considered the paper boundary. Once we find corners using these 2 methods, we perform sanity checks, such as if the corners lie in the image and if they enclose a convex area, and choose one of them.

2.4 Apply Transformation

After finding the paper corners we compute the perpespective transform and apply the same to the image to get the transformed image. Fig. 9 shows the block diagram for this stage.

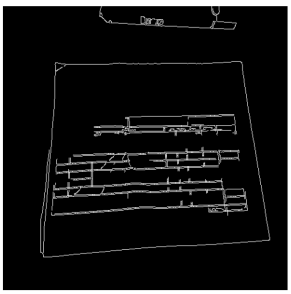


Figure 5: Edges cleaned by connected component analysis

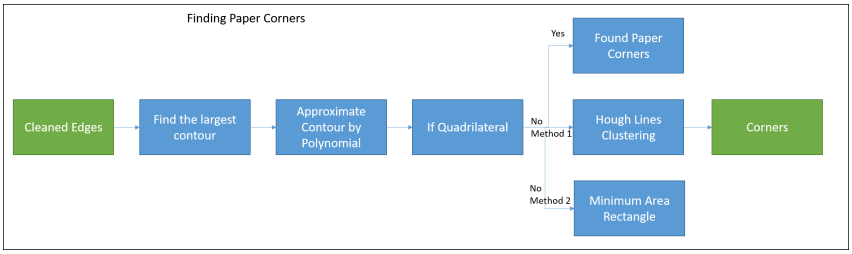


Figure 6: Corner Detection

2.5 Postprocessing

This is the last step in basic scanning. Postprocessing involves thresholding if the user wants a transformed image in grayscale. I have used histogram equalization and Contrast Limited Adaptive Histogram Equalization (Clahe) to make the transformed image visually pleasant. Else the persective transformed image is the final output. The basic block diagram for this stage is shown in Fig 10. Sample results are shown in Fig. 11.

3 Partial Form Scanning & Stitching

Here we combine overlapping partial document images to produce a single complete document image. The process of document scanning and stitching that has been implemented can be summarized broadly by the following block diagram Fig. 12. Sample results are shown in Fig. 13.

Given two partial document images we apply a key point detection and feature extraction algorithm like SIFT [1] on the images, which is a very popular algorithm in computer vision to detect and describe local features in images. With the keypoints we get we perform a matching using brute force (matching each point in one image to each point in the other). RANSAC can also be used, which would be faster. Using the best match we get, the best transformation function is computed. Finally the two images are joined together. The code for this stitching is adapted from *https://compvisionlab.wordpress.com/2013/05/13/image-blending-using-pyramid/*.

The stitched image has a strong edge at the point of stitching, and also black regions due to rotating the image while stitching. Therefore when normalizing a stitched image, these edges show up and cause problems in the paper detection. Since we know the location of the stitched edge and the black regions, we consider edges in those regions as false edges, when processing stitched images. However this process of checking for false edges is somewhat time consuming as we have to check that each edge point in the image does not match any false edge points. Sample results are shown in Fig. 14.

4 Reading Filled Forms

Here 2 images are input, one an empty form, the other a filled form. Using the steps described in 2, we convert both images into their normalized forms. Our target is to identify regions of text

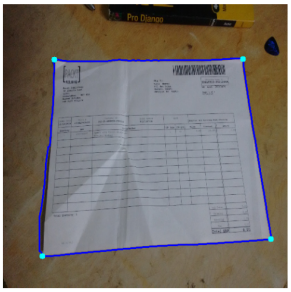
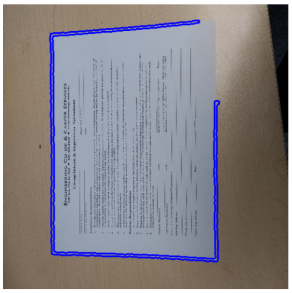
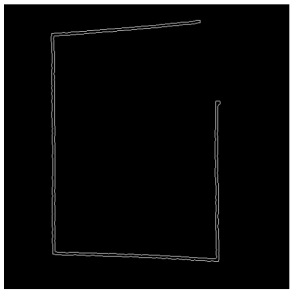


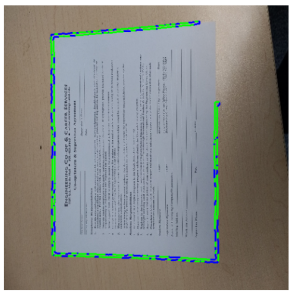
Figure 7: Corners detected by polynomial approximation



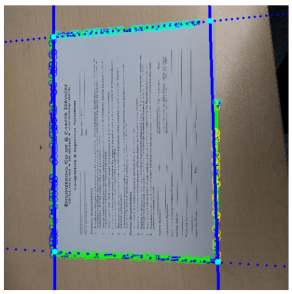
(a) Incomplete Contour Image



(b) Incomplete Contour Edges



(c) Hough Lines



(d) Detected Corners

Figure 8: Sample when polygon approximation does not work. Hough lines detection is used

http://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/ as the pipeline shown Fig. 15. On normalized images, we assume that text appears horizontally. Therefore if a dilation on the edges (fig. 16c for empty form and similarly fig. 17c for filled form) with an horizontal structuring kernel is performed, then characters in a single line get linked up. After that we find connected components, as we expect after the dilation a single line will condense into a single connected component as shown in green Fig. 16d for empty forms and Fig. 17d for filled forms. We find minimum bounding rectangles on the connected components and classify each region as a text or non-text region. For text regions we check for the following criteria:

1. The width of the region must be more than its height.
2. The region must be sufficiently wide.
3. The region must be sufficiently 'full' that is after dilation and linking, the ratio of number of white to black pixels in the region should be high

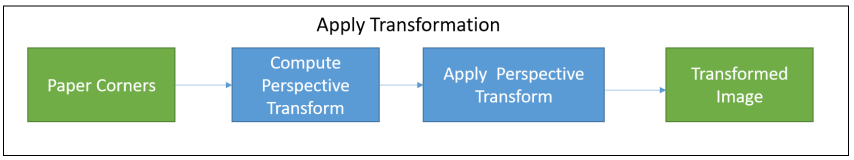


Figure 9: Perspective Transformation

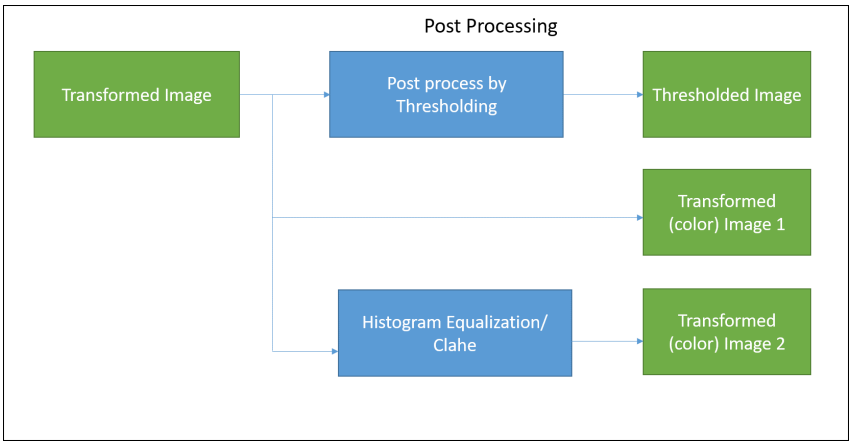


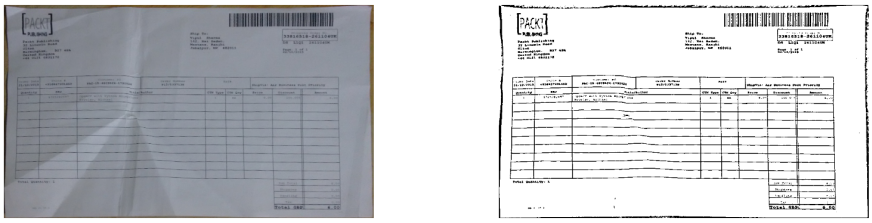
Figure 10: Postprocessing

4. The region should not be sticking to the boundary. This condition eliminates some boxes that form due to strong edges at the boundaries.

Fig. 16, Fig. 17 shows text region detection in empty and filled form respectively, and Fig. 18 shows the region of interest in red in filled form.

5 Optical Character Recognition

Here I have used a Tesseract OCR engine [2]. It expects a single line text as input so we must ensure that our text input is single line. This is done as described in 4. The OCR performs really well for printed text in high resolution images, but does not do very well for handwritten characters.



(a) Output image (b) Output, thresholded image

Figure 11: Basic Scanning output of input image

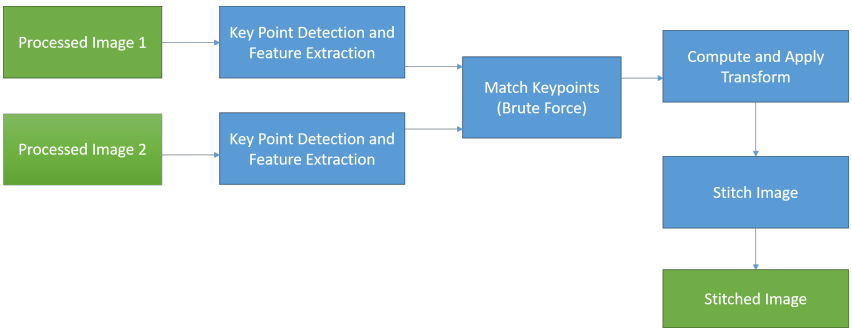


Figure 12: Partial Form Scanning and Stitching Algorithm

6 Sample Form

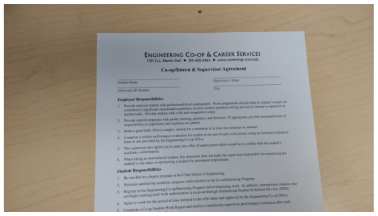
For the sample form given in class, the algorithm is able to correctly detect the paper's corners and align the image. In detecting regions of interest, it detects most of the correct hand-filled regions. But there are a few mistakes, and the reason for their failure is given below:

1. The number '1' is tall and thin compared to the aspect ratio of lines of text, which are usually wide. Therefore it is not detected.
2. The number '3' touches the line, which is part of a much larger connected component. The algorithm is able to detect text sitting on simple lines, but it fails here as the '3' is sitting inside a table structure. Once dilated, the '3' joins the connected component of the table and is impossible to retrieve using the current algorithm.
3. The paragraph starting with 'Comments' is returned as a region of interest. That is because in the empty form, each line in that paragraph formed its own connected component. However the hand-drawn circle around the word 'comment' joins 2 lines and creates a bigger connected component. This bigger connected component finds no match in the empty form, hence is returned as a region of interest. Outputs of the sample form is shown Fig. 19.

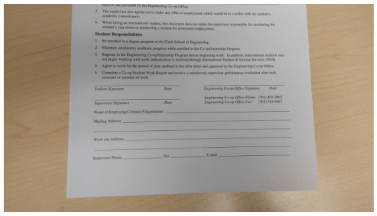
References

[1] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

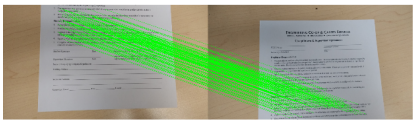
[2] R. Smith. An overview of the tesseract ocr engine. In *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*, ICDAR '07, pages 629–633, Washington, DC, USA, 2007. IEEE Computer Society.



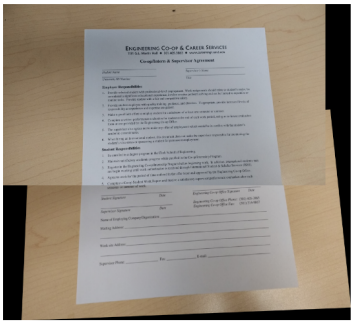
(a) Partial Form



(b) Output, thresholded image

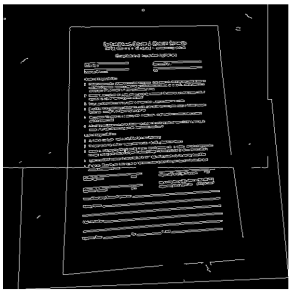


(c) Key Points and Descriptors

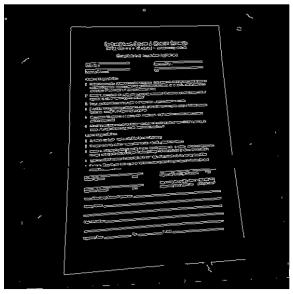


(d) Output, Stitched Image

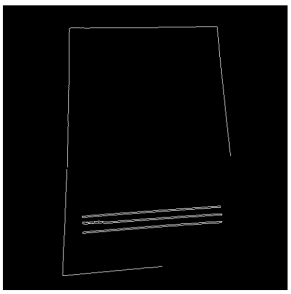
Figure 13: Partial Form Scanning and Stitching Output



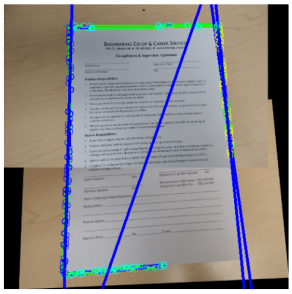
(a) Stitched Partial Form edges



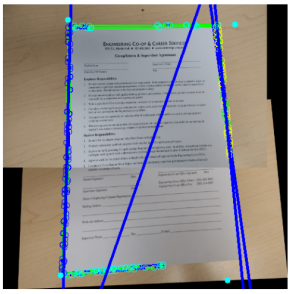
(b) Cleaned Edges using mask



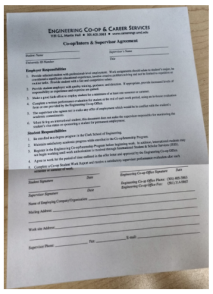
(c) Cleaned Edges further



(d) Hough Line Clustering of Stitched Image



(e) Corner Detection of Stitched Image



(f) Output: Stitched Image

Figure 14: Partial Form Scanning and Stitching Output

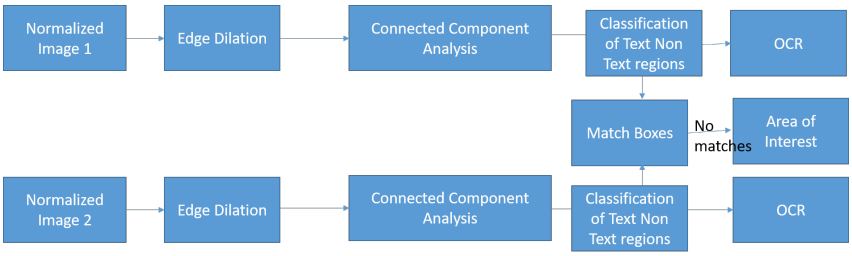
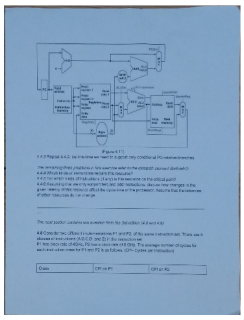
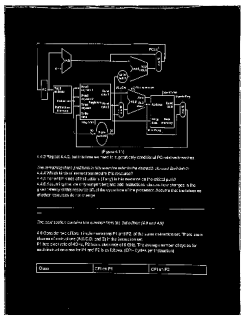


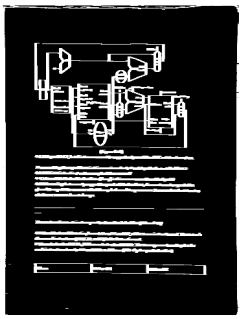
Figure 15: Read Filled Form Pipeline



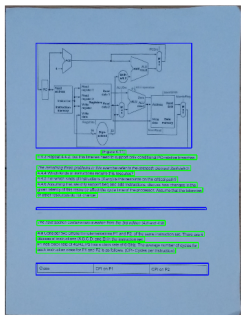
(a) Normalized empty form



(b) Edges

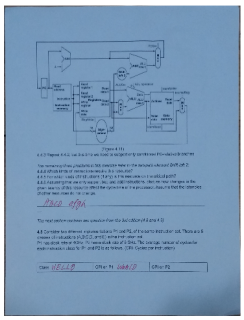


(c) Dilated edges

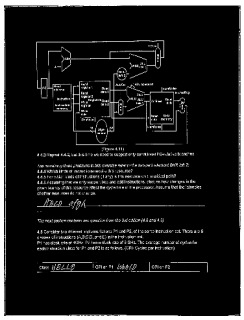


(d) Detecting text and non-text regions

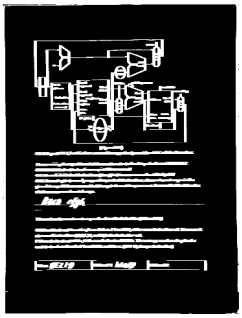
Figure 16: Text region detection in empty form



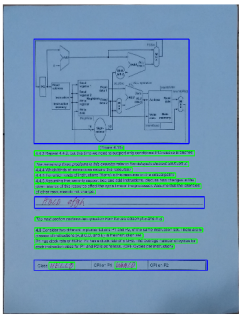
(a) Normalized filled form



(b) Edges



(c) Dilated edges



(d) Detecting text and non-text regions

Figure 17: Text region detection in filled form

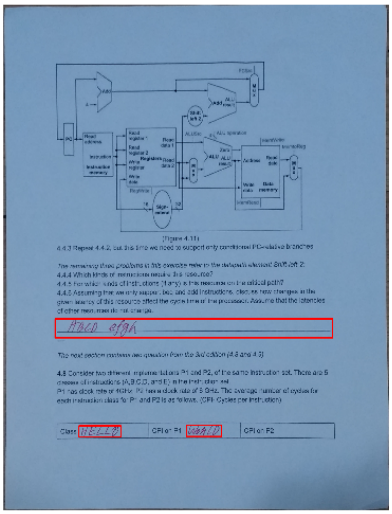
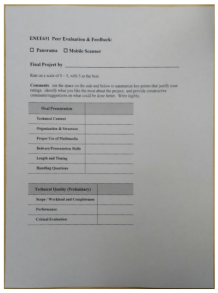
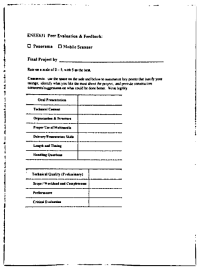


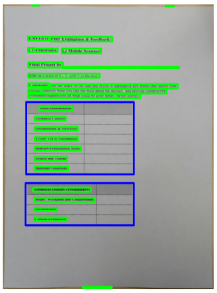
Figure 18: Region of Interest in filled form



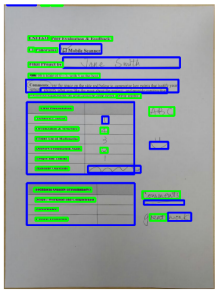
(a) Sample Form Scanned output



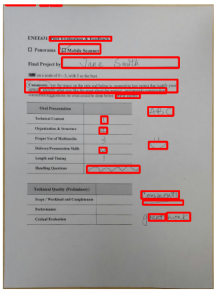
(b) Sample Form Scanned binary output



(c) Regions of Text in empty sample form



(d) Regions of Text in filled sample form



(e) Regions of Interest

Figure 19: Sample form outputs