medium.com

Document Detection in Python

Shakleen Ishfar

16-20 minutos

12 min read

Jun 5, 2020

Hello, good people! I hope everything is going well. It's been quite a while since I wrote an article on Medium. So, I decided to take up the pen… errm… keyboard, and write again.

Recently, I had a chance to work on document detection. Learnt some pretty amazing things while doing so. I hope to share what I learnt in this article, **which is mainly about document detection using image processing.** That being said, I'm not going to give a thorough walk-through on building the absolute best document detector ever.

Don't cry tho! We're gonna make a simple one together. It might not work every single time. But it'll be enough to get the main idea through. More sophisticated detectors can be built on top of this one.

I'll discuss the general process of building it and mention what needs to change depending on the situation.

Before proceeding further

Required knowledge

You just need to know Python 3. I'll provide links to helpful articles for literally everything else.

Where to find the code

The code from this article can be found in this GitHub repository.

Environment Details

I'll be using Python3 (3.7.7) and relying heavily on the additional python packages (that aren't be included by default). Check the requirements.txt file and install them using pip.

Gonna be a LONG article! Grab a cup of coffee, get comfortable. Let's do this!

Document Detection

Document detection is basically extracting a document from an image. Machine learning and image processing are two common approaches to do this. This article discusses the latter approach. Here, we use pixel manipulation techniques to detect the document.

Steps Involved

1. Pre-processing image to enhance document.
2. Finding document extraction points in the image. Quadrilaterals are the extraction points in this article.
3. Extracting the document from the points.

4. Post processing to correct distortions and imperfections.

Let's code this up in Python. We'll fill in the details as we go.

Code Snippet #1: High level overview of **PageExtractor** class

PageExtractor Class

1. It's only for performing page extraction, which it does in its **_extract_page** method. It is called in line no 22. (You can find how this method works in step 3, towards the end of the article.)
2. Corner detection, pre and post processing code isn't going to be a part of this class. Separate class objects will perform these actions. They'll be passed as parameters. (We will create the required classes soon.)
   Why are these parameters?
   Because there are many different ways to perform these tasks. This way, I can swap them out and leave the core logic unchanged. (Core logic being the page extraction function.)
   What's *output_process* parameter?
   I use this parameter for debugging. When set to true, intermediate images are saved to output folder. I use these images to check if the filters are working properly.
3. Notice lines 13 to 16. Pre-processing is done by applying a chain of processors. The output of one processor is fed into the next processor. This is repeated until there are no more processors left to call. **All processors thus needs to follow a common structure. It must take only an input image and produce only an output image.**
4. You might have noticed that there is no post-processing

here. I omitted it for the sake of keeping the article short. But they should follow the same structure as the pre-processors.

Step 1: Pre-processing

> The end goal is to get a clear and usable document. Pre-processing is done to help achieve that goal.

Common processing steps

1) Threshold or binarize to make images only black and white.
2) De-noising to reduce noise.
3) Sharpening to make blurry images workable.
4) Morphologial transformations to fill in missing details or erase unnecessary details.
5) Contrast stretching to increase the contrast of an image.
6) Inversion to invert light and dark places of an image.
etc.

Press enter or click to view image in full size

Figure #1: Example image of a document

Some things to note before we begin this section:

1. All the processors are kept in the same file **processors.py**, which can be found here.
2. All processors have a common structure, they take an input image and produce an output image. Necessary settings are passed during class object initialization.
3. I'm keeping things simple and using only 3 pre-processing functions. Depending on your use case, you might need to come up with a list of your own. **Processor ordering matter because they are applied in a chain fashion.**
4. I'll be using the image seen in *Figure #1* as an example. It'll be helpful for visualization.

1.1. Re-scaling filter

> Processing high-resolution images are quite resource intensive.

When an image is above a certain size, I like to scale them

down for faster processing. My implementation can be seen in *Code Snippet #2*.

Code Snippet #2: Resizer for scaling images down to a specific size if they are larger.

> **A bit of caution is advised.**
> 1) Scaling an image down too much will cause it lose details and become pixelated.
> 2) Distortions can arise from scaling images. We want to keep height-width ratio same.

1.2. De-noising filter

> The amount of noise in an image depends on the hardware and the condition it was captured in. Images captured in poor lighting tend to be very noisy.

There are multiple methods for de-noising. And their use case is dependent on the amount of noise present. Here are some functions offered by OpenCV.

> Be careful when applying de-noising. **Because, the more strongly an image is de-noised, the more details is lost.** Hence, it can cause data loss in images.

Code Snippet #3: FastDenoiser

Press enter or click to view image in full size

Figure #2: Denoised image

I used **fastNlMeansDenoising**, as seen in line 9 of *Code Snippet #3*. The **strength** parameter in **FastDenoiser**, controls the amount of de-noising.

Figure #2 shows the output of the de-noiser. Pay particular attention to the surface the page lies on. In the original image, it had a grainy texture. But here, it's smooth. That's the smoothing effect of de-noising.

Another thing to notice is the speed of operation. Typically, median filters are really good for de-noising, but are very slow to run. Blurring filters are faster to run but they tend to smooth out important details as well.

1.3. Thresholding

> In digital image processing, **thresholding** is the simplest method of segmenting images. From a grayscale image, thresholding can be used to create binary images. — Wikipedia

There are multiple ways to perform thresholding. Depending on the use case, one method may perform better than the other. Here are some thresholding functions offered by OpenCV.

Code Snippet #4: OtsuThresholder for binarization.

Press enter or click to view image in full size

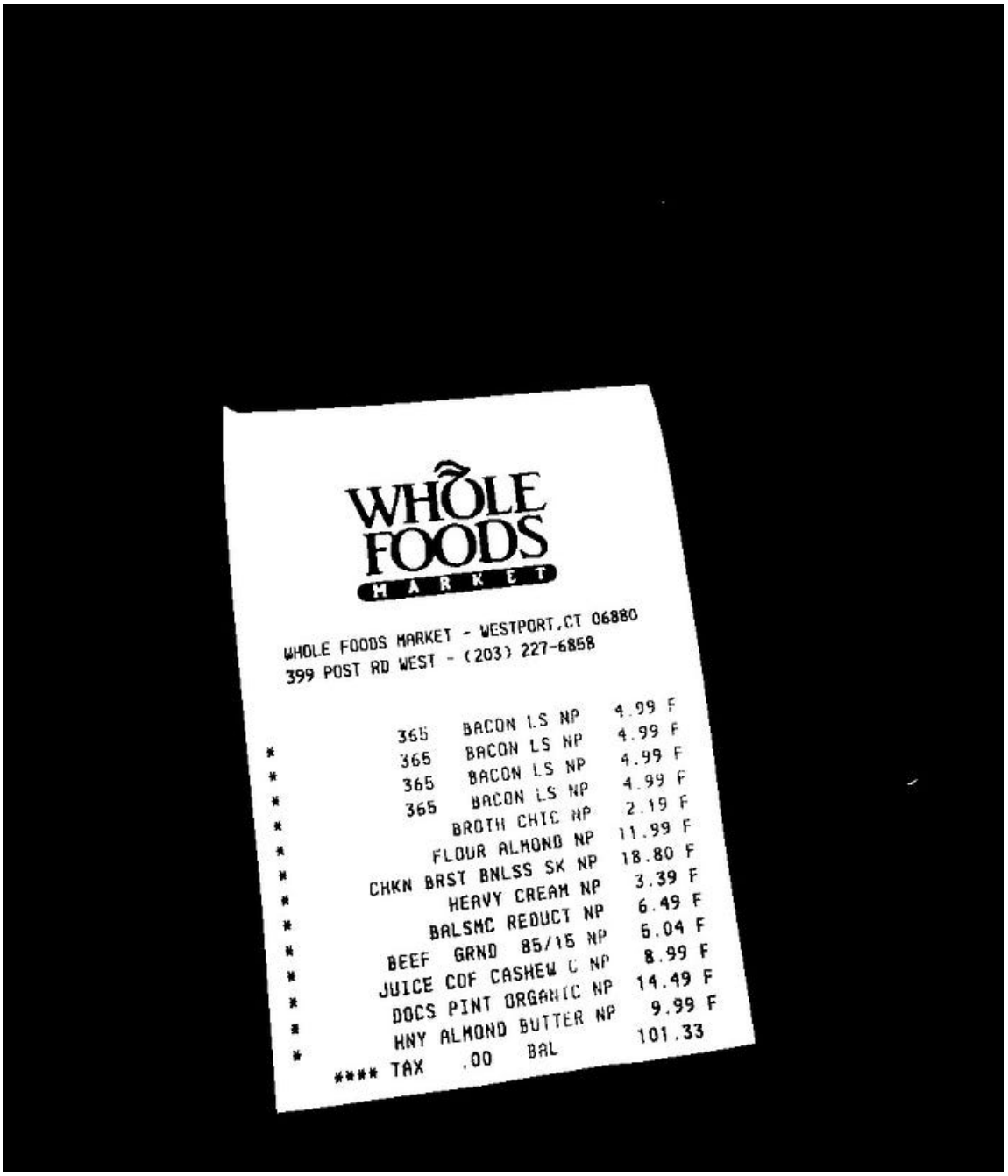


Figure #3: Thresholded image

I decided to go with Otsu's method for my needs (*Code Snippet #4*).

The result of thresholding is illustrated in *figure #3*. Compare this with *figure #2*. The page doesn't have wrinkles. It is pure white and the writings are more easily visible. The background is completely black.

However, thresholding might not always be desirable. When color is an important factor for example.

Step 2: Getting the extraction points

> Extraction points are points on the perimeter of the document which will be used to extract it.

The simplest approach is to use quadrilaterals or the four corners of the document. There are two popular ways to do find quadrilaterals:
1. Hough lines
2. Contours

My implementation is based on the hough lines approach. [If you're interested in a contour based implementation, here's a great article by Adrian in PyImageSearch.] Which ever way you decide to perform this, simply create a class that takes the processed image and spits out the quadrilaterals. Then pass the object of this class to **PageExtractor**.

My Corner Detection Process

1. **Processing step:** Process the input image so that hough

transform goes smoothly.

2. **Getting the lines:** Run hough transform to get hough lines.
3. **Getting the intersections:** Calculate the intersections between hough lines.
4. **Finding quadrilaterals:** Group the intersections into 4 clusters and take the mean of each cluster. These mean points are our quadrilaterals or corners.

   Code Snippet #5: High level overview of corner detection using hough transform

2.1. Processing for corner detection

It's all about processing in image processing.

Hold on a minute, mister! Why are we processing again? Didn't we just do this?
Well, yes. We did. But there is a reason to this madness. Corner detection will work best if we can omit unnecessary details in the image. For example the writings on the page are not necessary to detect it's boundary. To prevent them from affecting out detection process it is desirable to erase them.
But we want the writings to exist in the final detected document. Which is why, we don't want to include these processing in the previous pre-processing step. This is just for corner detection!

2.1.1. Morph functions for eliminating unnecessary details

Now, that we're in the same page, how do we erase

unnecessary details? It should be noted that the image being fed into the corner detector is assumed to be a black and white image. (If it isn't it needs to be binarized before anything) We can use morphological transformations for erasing unnecessary details. Here's a great article to get you up to speed on morph functions offered by OpenCV.

Code Snippet #6: Closer for performing morphological close function

Press enter or click to view image in full size

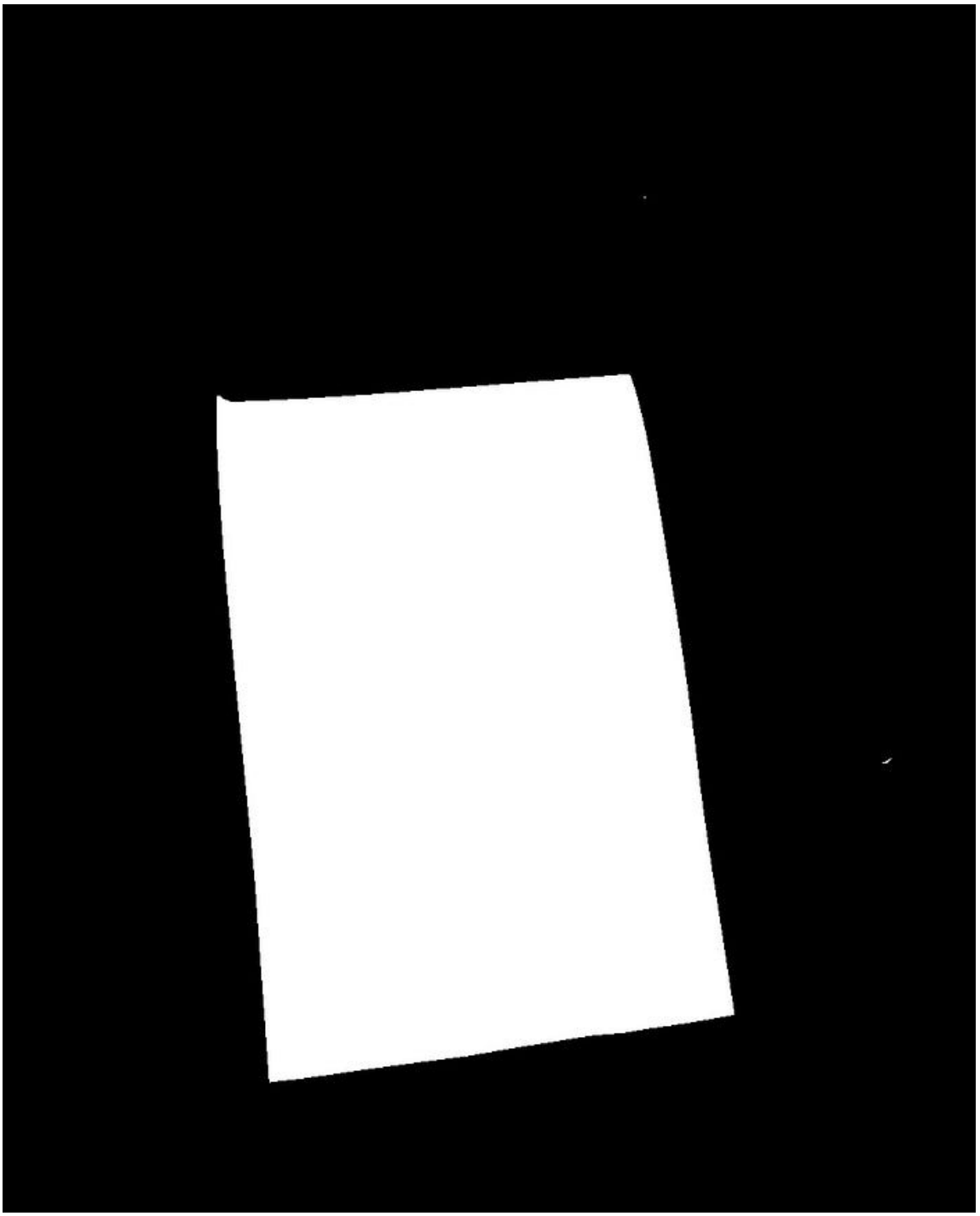


Figure #4: Output of closing. Writings eliminated. Only page visible.

I created **Closer** class (*Code Snippet #6*) to perform morphological close function. You can control the effect of the morphological function by tuning the parameters **kernel_size** and **iterations**.

The output should look something like *Figure #4*. Notice how only the page is left and the writings are gone. Perfect!

Depending on the use case, you might need other morph functions. For example, there might be dark regions in the page or white noise in the back ground. These can be fixed using morph functions!

> Be cautious when playing with the parameters. You might distort the actual shape of the page if you aren't careful.

2.1.2. Canny Edge Detection

Canny is an edge detection algorithm. A quick guide of canny and how it works can be found here.

So, what purpose does this serve?
Mainly, to detect the page perimeter. To perform document detection, we need only the perimeter and nothing else. The inner white portion is unnecessary. Canny gives us exactly that. It gives us the perimeter outlines or the edges. Moreover, without this, hough trabsform will not behave the way we want it to!

Code Snippet #7: Edge detection using Canny

Press enter or click to view image in full size

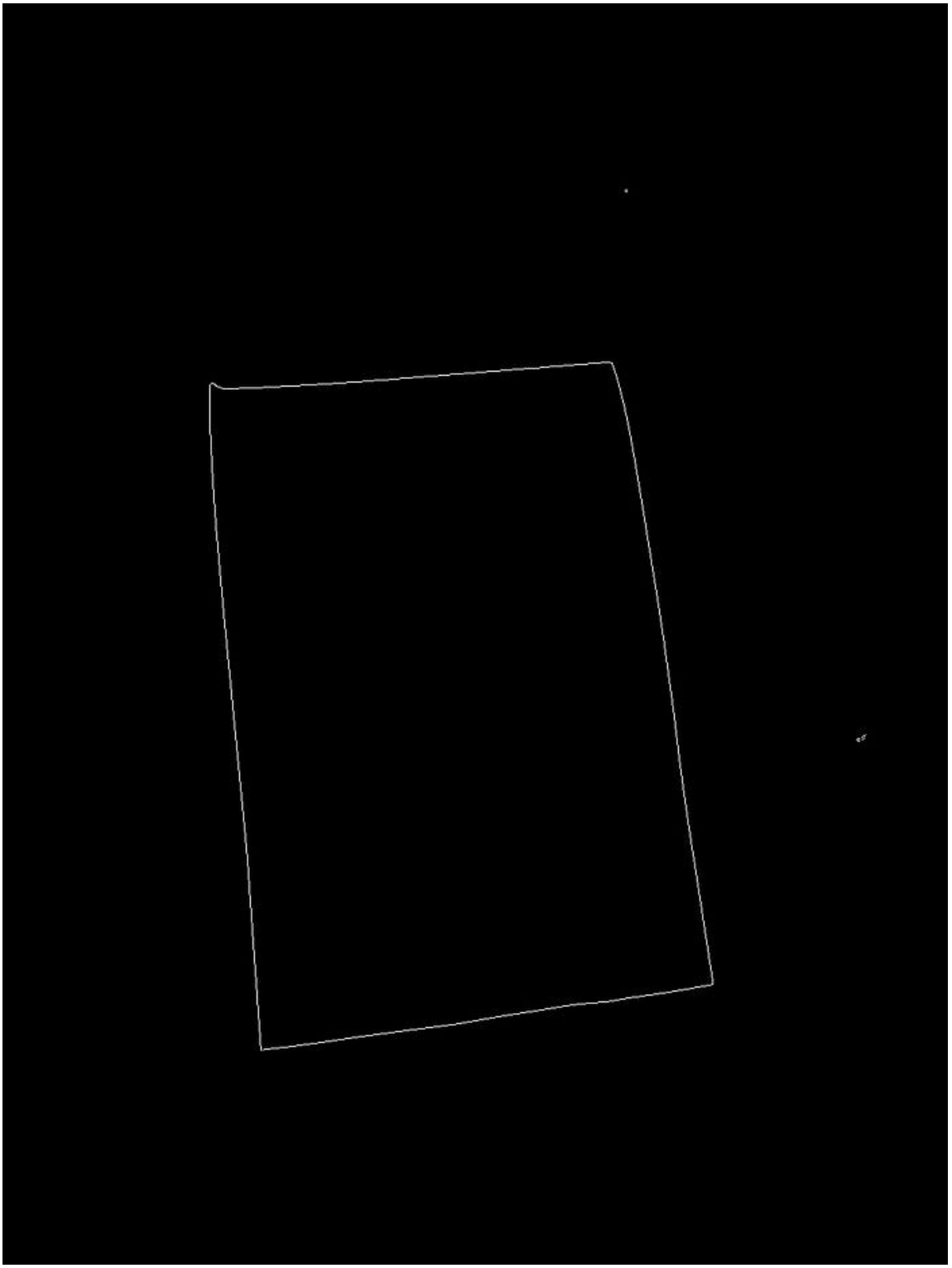


Figure #5: EdgeDetector output

I created a processor for edge detection using canny called **EdgeDetector** (*Code Snippet #7*).

The output looks like *figure #5*. Usually, this will be the image that we feed into hough transform function. But it's

possible for this image to have way too many unnecessary edge information (Yes, even after all that processing). Hence, an edge cleaning filter might be required to clean the output before proceeding with hough transform.

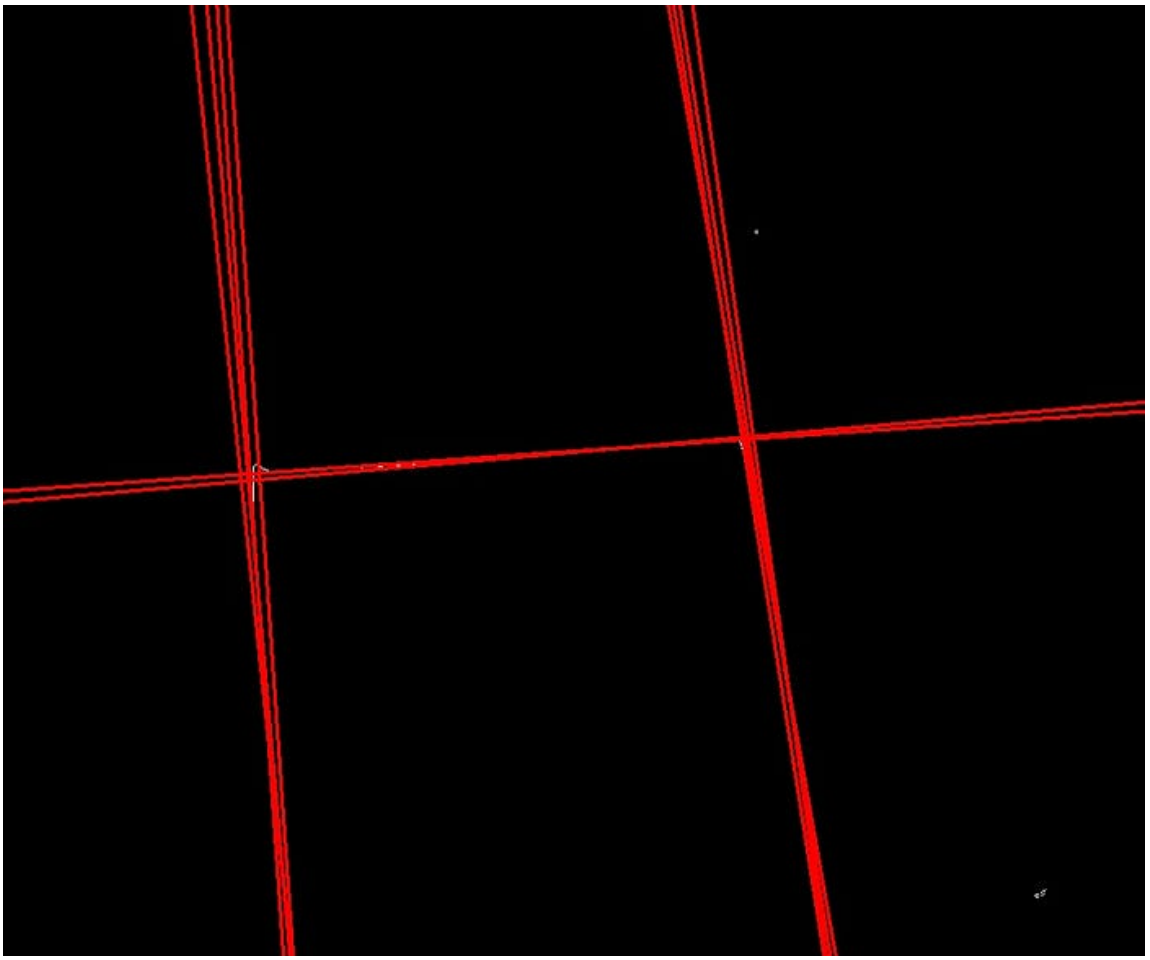
2.2. Hough lines

If you're not sure what hough transform is, here's an excellent explanation to get you up-to speed. OpenCV has **HoughLines** and **HoughLinesP** functions for this purpose. I'll be using the former. It has 3 parameters which are of concern:

1. Rho Accuracy
2. Theta Accuracy
3. Line threshold

Code Snippet #8: Getting hough lines

Press enter or click to view image in full size



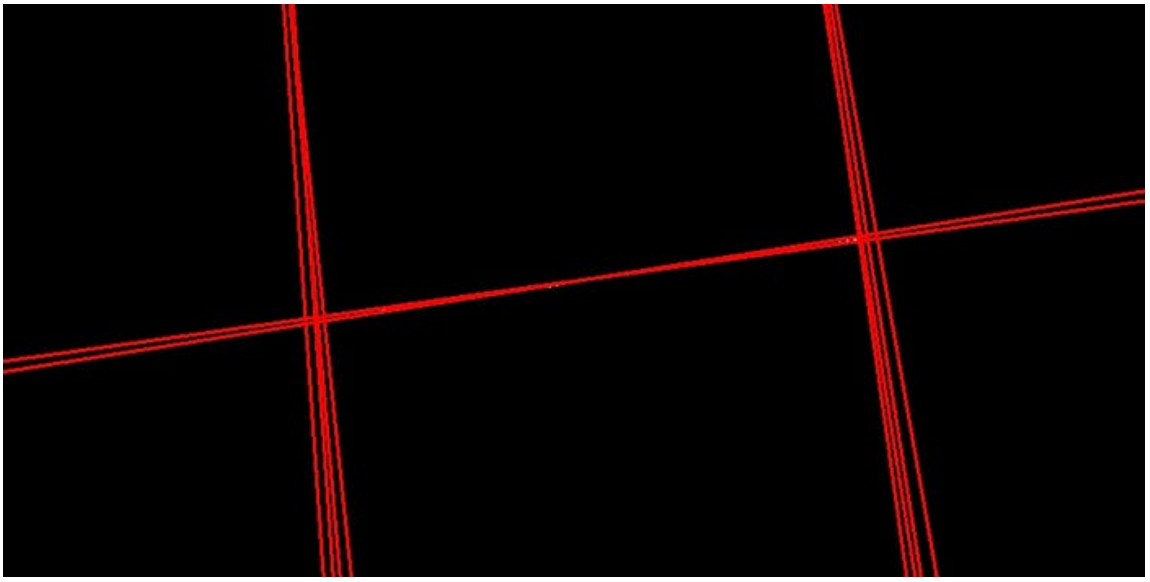


Figure #6: Hough lines from processed image

_get_hough_lines is our function of interest (illustrated in *Code Snippet #8*). The lines variable holds our desired hough lines (Line 5). Here, lines is an array of rho and theta value pairs. These two values describe each of the hough lines.

I got the result shown in *figure #6*. The hough lines are colored in red. You can clearly see that it properly encompasses the page in the processed image. There might be stray hough lines (although there aren't any here). These are lines that don't align well with the page border. As long there aren't too many of them, they are of no concern.

2.3. Getting intersections

So, we need to calculate the intersections between the hough lines and we're all good right?

Not so fast! As you can see from *figure #6*, there are multiple lines on each side. Lines on the same side are almost parallel to each other. If we calculate intersections among all lines, then we'd be left with

1) way too many intersections

2) and some won't be even close to the quadrilateral points.

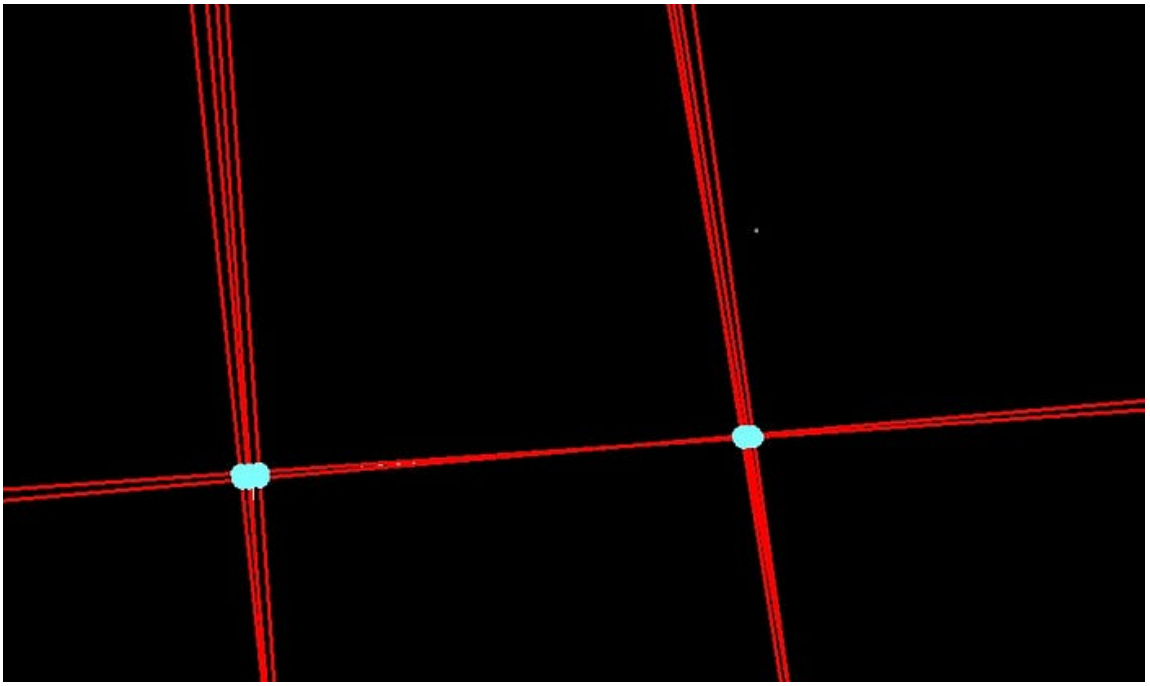
So, we need a way to check if the intersections are proper. We're going to do the following for each and every pair of hough lines:

1. We will calculate the intersection angle between them. That this angle will be close to 90 degrees. (Because the angle created by each side of the page at the quadrilaterals should be 90 degree). Let's assume the angle will be between 80 and 100 degrees.
2. If the angle is within bounds for two lines, we will calculate the intersection them.
3. Finally, we check if the intersection is within the boundary of the image and if so, add it our list of intersections calculated so far (which will empty to start with).

You might employ more tactics to carefully choose the intersections. The better you perform here, the better the detection will be.

Code Snippet #9: Calculating intersections

Press enter or click to view image in full size



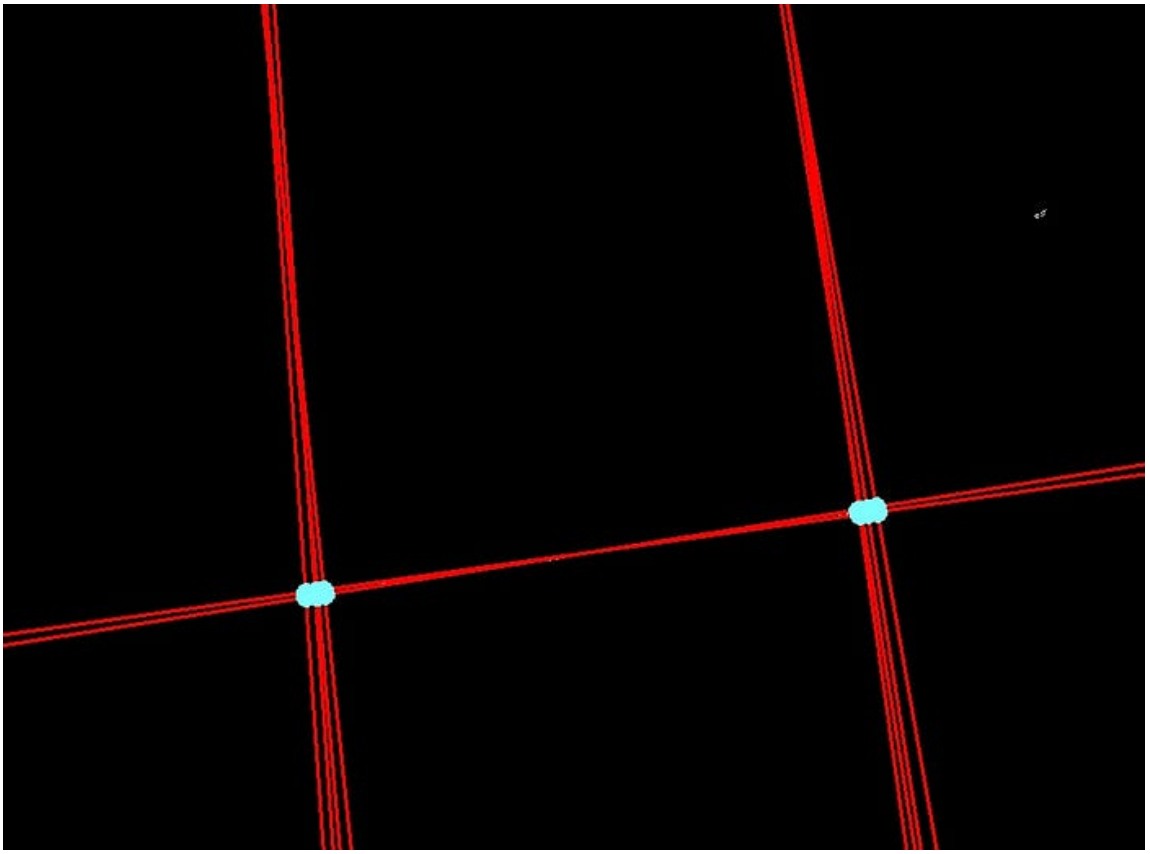


Figure #7: Intersections between hough lines

Code Snipper #9 produces the image in *Figure #7*. The blue dots are the intersections we get. Also notice that the parallel lines didn't produce any intersection points.

Here's a quick run down of what happens where.

1. **_get_intersections** at line 3 is going to produce the intersections we want. It first checks the angle and then computes intersection point. Finally returning a list of all intersections.
2. Angles are being calculated by **_get_angle_between_lines** at lines 25 to 33.
3. Calculated angles are being checked at line 15.
4. If the angle is acceptable, **_intersection** calculates the intersection (lines 36–53).
5. In line 18, I check to make sure that the intersection point is within the boundary of the image and if so it's added to the

list of intersections in line 19.

2.4. Getting the quadrilaterals

All that's left is to cluster the intersections into 4 groups and get the mean point from each group. I did this using KMeans algorithm, [which you can learn more about here](#)!

Code Snippet #10: Finding quadrilaterals

Press enter or click to view image in full size

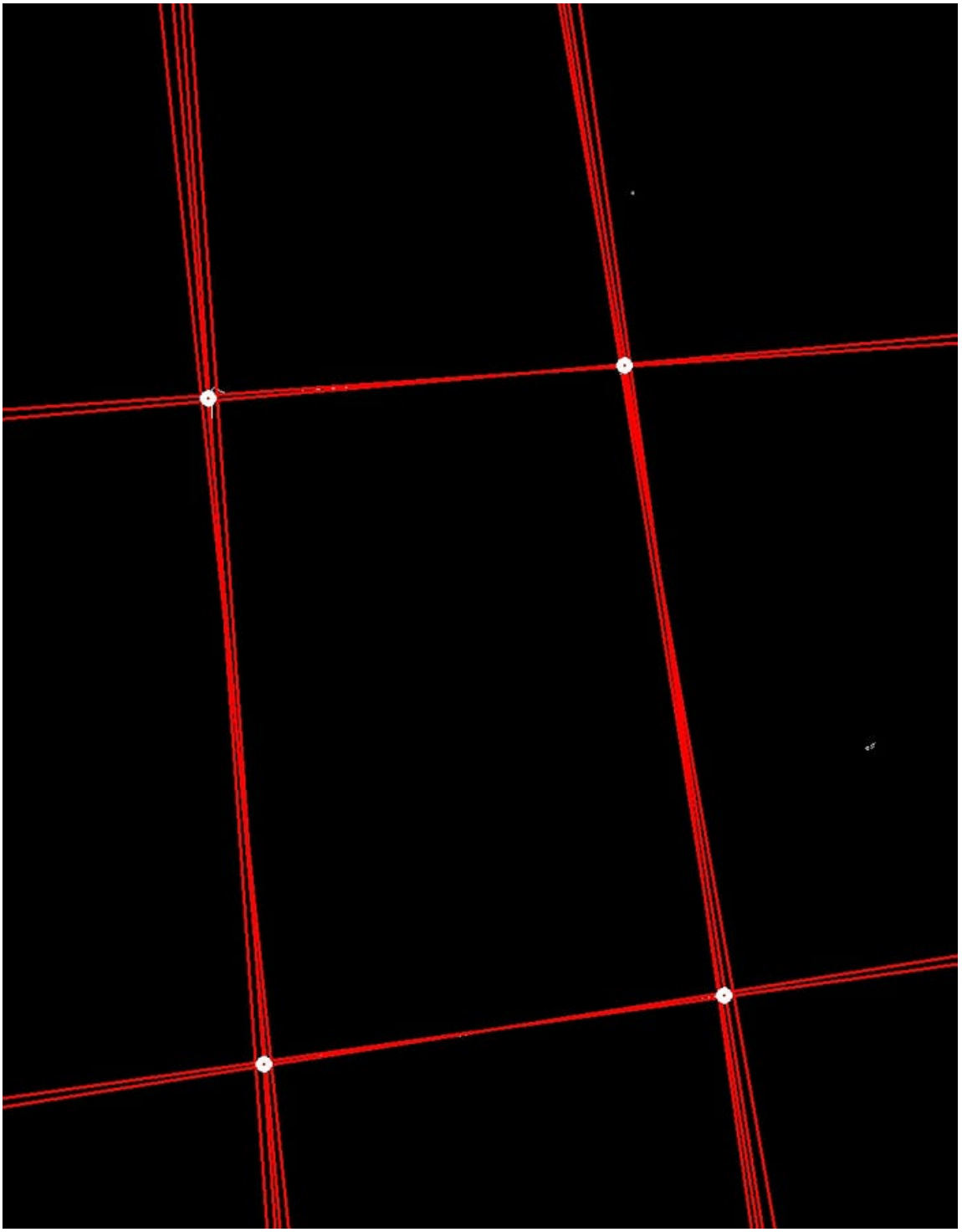


Figure #8: Quadrilaterals

The clustering step is shown in *Code Snippet #10*. This piece of code produces the image seen in *Figure #8*. The white points are our quadrilaterals.

A question might pop into your mind right about now. What if there were hough lines all over the place? What if they weren't as properly positioned as these? Wouldn't that mess up the intersections in in turn mess up the quadrilaterals? Well yes. They would. But remember that we're discarding intersections whose angles aren't close to 90 degrees. And then we're taking the mean of the intersections of a group. The results won't be perfect sure. But it'll be reasonably close.

Step 3: Page Extraction

Now that we have the quadrilaterals, we just need to extract the page. If you remember this was the purpose of **PageExtractor** class. We can use OpenCV's **warpPerspective** method to perform this extraction. Here's an excellent article that discusses this function in depth.

Code Snippet #11: Page extraction at last.

Running the code

We have all that we need. Now we just need to create an instance of PageExtractor and call it with an image path. *Code snippet #12* shows how to do this:

Code Snippet #12: Creating an instance of PageExtractor.

And finally, here's what this code produces.

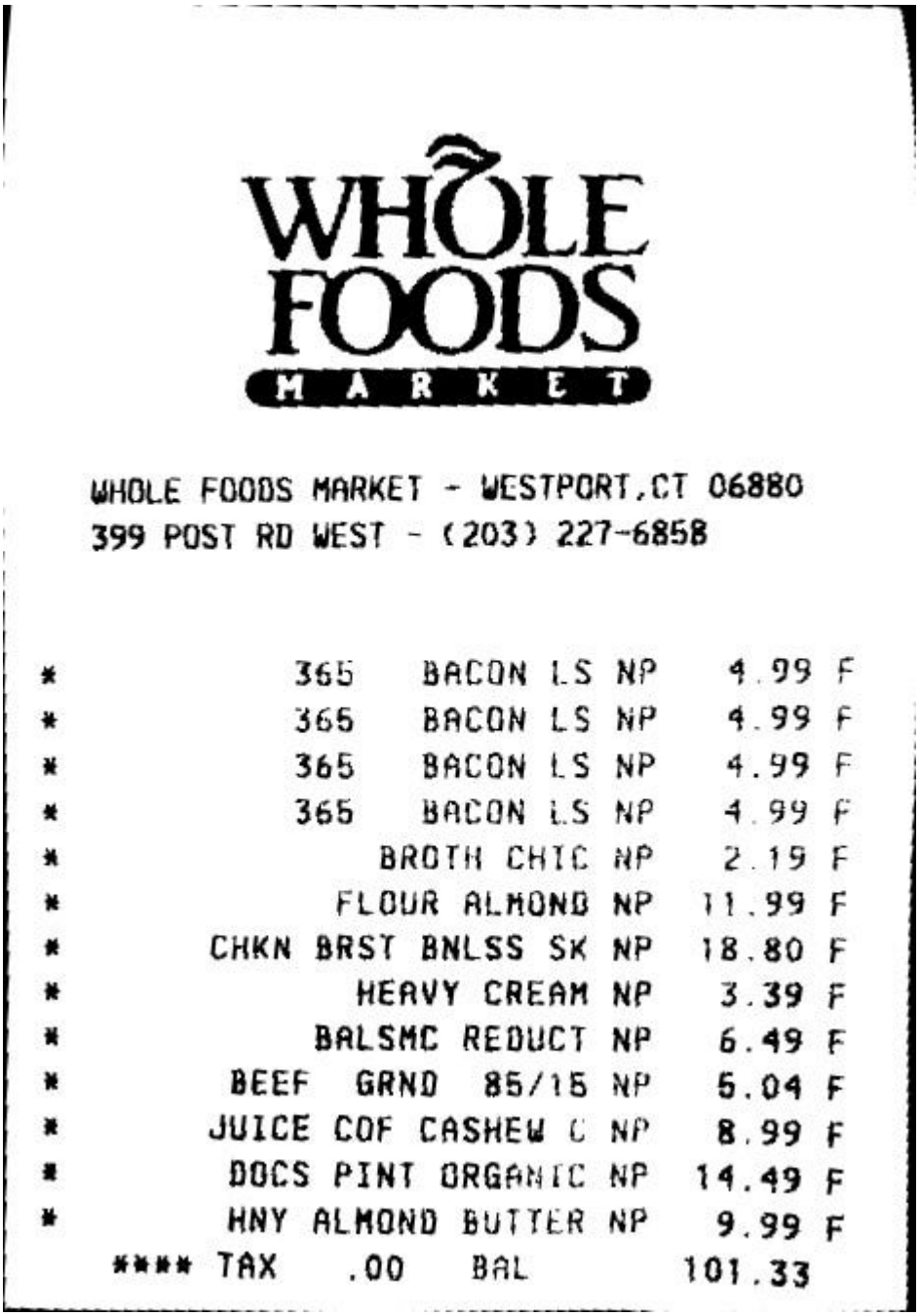


Figure #9: Detected document

Step #4: Post Processing

There aren't too many distortions in the extracted image. However, it might benefit from a little bit of post processing. Here's what could be done:

1. Notice that the image is slightly tilted to one side. The writings aren't completely horizontal. So, a rotation correction filter would be useful here.

2. Some of the characters appear to have lost some details. Morphological transformations can help to reduce this.

   But because these aren't big issues and the article is already humongous enough, I'll leave it to you, the reader, to explore this more on your own.

Summary

That was a long article. Don't worry if the underlying details have escaped you. Here's a quick run down of the entire thing:

1. Pre-process image to enhance the document. Use processors like sharpening, blurring, thresholding, resizing etc.
2. Get the quadrilaterals. This can be done by contours or hough lines.
3. Extract document using the quadrilaterals.
4. Correct unwanted distortions due to the extraction process using post-processing filters.