HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Document Scanner

Leopold Gaube and Florian Betz

May 12, 2022

Contents

1 Task Overview

Not everybody has a physical document scanner in their own household. However, there are plenty of smartphone apps available that can transform any photograph of a document into a "scanned version" without any distracting background or perspective distortions.

In the following report we will present our own approach for scanning a document in an image. We do not want to implement everything from scratch, so we will be using OpenCV, an open-source computer vision library that fits our purpose. It is available for C, C++, Python and Java [Ope22b]. We will be using Python 3.8. Our code and all test images are available in this GitLab repository [GB22].

The algorithm consists of three main parts, each with its own challenges. First, we have to localize the document in the image. Then we use a perspective transform from the corner points of the document to the entire span of the image in order to obtain a top-down view without perspective distortions. Finally, we use a binary filter to distinguish text from background and save the resulting image as our scanned document.

We also want our algorithm to work under bad lighting conditions, when the document is rotated at 45° and even when the image was taken from an extreme angle. Naturally, the final image will suffer in quality compared to a document taken under near optimal conditions, but our goal is to still obtain a readable document.

2 Document Localization

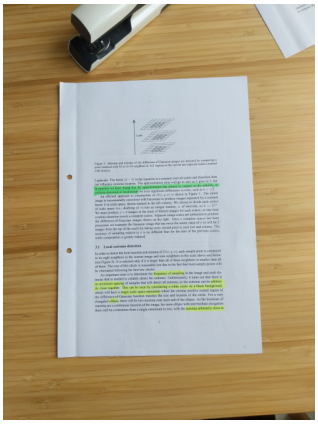
Some smartphone scanner apps require the user to mark the four corner points of the document manually, however, we want to automate this process by detecting these points using Computer Vision. Detecting the corner points consistently is arguably the hardest and most crucial step of the entire scanner program.

Our first goal is to find the document outline in the input image, but before that we will do some preprocessing. We do not need a high resolution image, so we will reduce the information content by first downsizing the input image to something more manageable. We resize the image to 400 pixels on the larger side (width or height) and adjust the smaller side according to the original aspect ratio. In the process we will also get rid of noise which could potentially reduce the algorithm's performance. In addition to downsizing the image, we also use a Gaussian Blur (with a 5x5 kernel) in order to smooth the image even further.

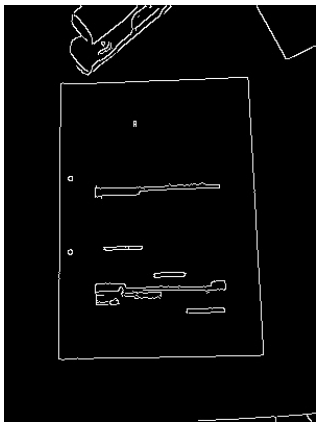
Now we can use a Canny Edge Detector to find locally sudden changes in brightness or color. The resulting binary image distinguishes between edges (depicted as white)

and non-edges (black) for each pixel location in the original image (see Fig. 1b). The Canny Edge Detector uses two thresholds. All potential edges above the maxValue-parameter will always become an edge and all potential edges below the lowValue will all be rejected. Any value in between both thresholds will only be accepted as an edge if the pixel is adjacent to an already existing edge. [Ope22a]

We are only concerned that the document outline shows up in our edge image, so we chose an upper threshold of 150 which works consistently even for documents on medium contrast backgrounds. However, we encountered a problem that on some images the document outline was missing a single pixel in the edge image and therefore resulting in an unclosed contour. That is why we chose a low value of 50 for the lower threshold that helps to increase the chances of obtaining a closed contour.



(a) Input image



(b) Canny Edge Detection

Figure 1: The Canny Edge Detector finds local regions of sudden brightness change

From this point on, we have to assume that there exists a closed contour in our edge image which matches the outline of our document. When trying to find the location of the document corners, it is ok if we miss them by a couple of pixels but detecting the wrong contour would make the resulting document unreadable.

We use OpenCV's built-in method for finding coherent contours. The challenge is to distinguish the one corresponding to our document from all the other contours. Other contours may include text, graphs and pictures on the document itself or even objects e.g. a stapler on the desk. The contour we are looking for should have one of the largest perimeters and it should be able to be approximated it with only four points. Both of these assumptions will help us to find the right one, as it is highly improbable that a different contour meets both criteria. We will pick the largest contour that can be approximated using four points as shown by the following code:

```
# use OpenCV to extract all contours from an edge image
contours, _ = cv.findContours(
        edge_img, mode=cv.RETR_EXTERNAL, method=cv.CHAIN_APPROX_SIMPLE)

largest_contour = None
largest_area = -1
for contour in contours:
        perimeter = cv.arcLength(contour, closed=True)

        # approximate the contour using the least amount of vertices
        # that still satisfy a distance precision of 3%
        low_poly_contour = cv.approxPolyDP(contour, 0.03 * perimeter, closed=True)

        if len(low_poly_contour) == 4:
                # approximate area with a roated rectangle
                ((_cx, _cy), (w, h), _angle) = cv.minAreaRect(contour)
                area = w * h
                if area > largest_area:
                        largest_area = area
                        largest_contour = low_poly_contour
```

3 Perspective Transformation

For the document localization part, we have been working with a low resolution image, making the text unreadable. Now we want to create a top-down view of the document with readable text, so we need to project the detected corner points back onto the original resolution.

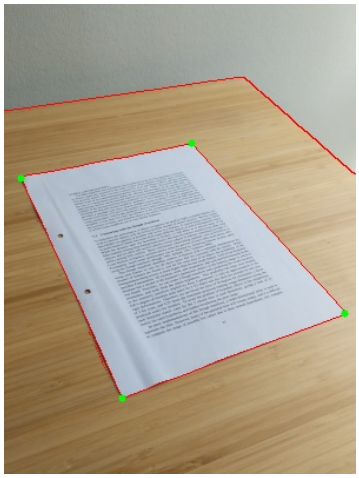
When detecting contours, OpenCV does not care about the order of the points which may result in a rotated image when applying the perspective transformation. Therefore, we must first sort our corner points in a consistent manner.

OpenCV provides a function for automatically calculating a perspective transformation matrix. It maps our detected (sorted) corner points to the corresponding image corners. This matrix can in turn be used to apply this transformation to our input image in order to obtain a top-down view of our document (see Fig. 2b).

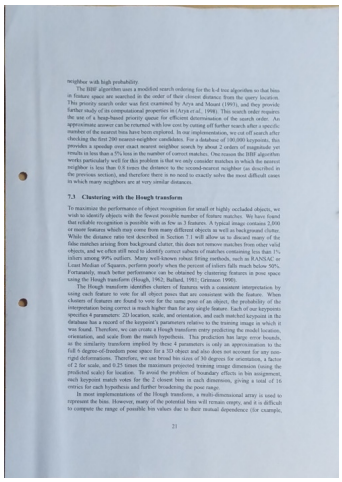
```
# sort corners on their relative position [top left, bottom left, top right, bottom right]
sorted_corners = sort_corners(doc_corners)

src = np.array(sorted_corners, dtype="float32")
dst = np.array([[0, 0], [0, HEIGHT], [WIDTH, 0], [WIDTH, HEIGHT]], dtype="float32")

M = cv.getPerspectiveTransform(src, dst)
top_down_img = cv.warpPerspective(img, M, (WIDTH, HEIGHT))
```



(a) Detected contours (red) and document corner points (green) overlayed onto the original image



(b) Document in top-down view

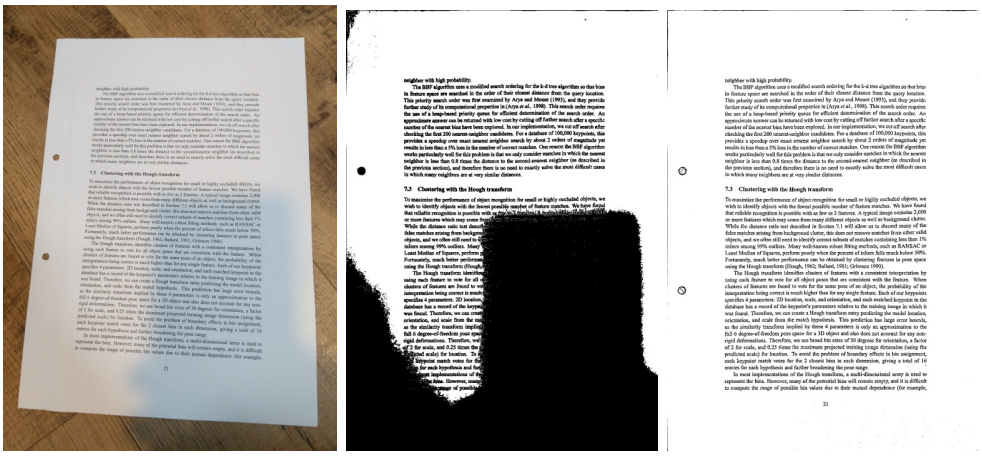
Figure 2: By using a perspective tranformation we obtain a readable document without any background.

4 Thresholding

For the last step of our project we convert the scanned document into a binary image, thus each pixel is supposed to be either black for text and graphs or white for the background. The easiest way to achieve this is by converting the color image into grayscale and setting a threshold value. Any pixel value below this threshold would become black, pixel values above would be set to white.

However, there is a problem with this simple thresholding approach, as it performs poorly under bad lighting conditions as can be seen in Figure 3b. When taking pictures of documents, a user may cast a shadow with their camera or mobile phone onto the document. This can be problematic if a region in the shadows has lower pixel values for the background than the text in a well-lit region. In such a scenario, it would be impossible to find a static threshold that works well for the entire document. So instead, we need an adaptive approach that looks at a set of neighboring pixels and chooses a threshold for this local region dynamically (see Fig. 3c).

The corresponding OpenCV function takes multiple parameters and after some experimenting we settled on a 11 x 11 neighbourhood and a constant C of 5. Furthermore, we preferred the adaptive Gaussian method over the adaptive mean method, making the text look smoother. We determined these parameters empirically by trying to maximaize document readability in most test images.



(a) input image (b) simple threshold (c) adaptive threshold

Figure 3: A cast shadow in the input image (a) may result in unreadable text when using a fixed threshold value (b) and therefore an adaptive thresholding method (c) should be used instead.

It should be noted that an input image with a lot of pixel noise will result in a noisy scanned document no matter the thresholding method (see Fig. 4). In order to reduce the black artifacts we could smooth the image before the thresholding. Unfortunately it would also compromise the text readability and is therefore inadvisable.

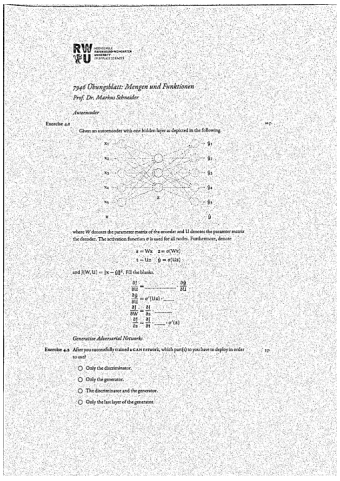
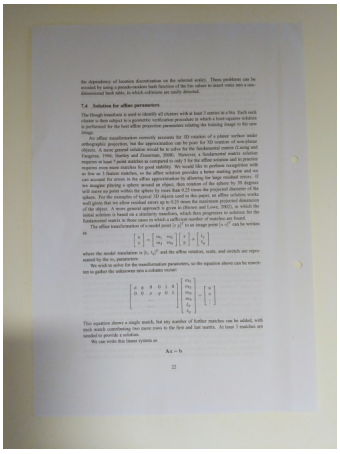


Figure 4: A high noise images produce small black artefacts in the final scanned document. This is a minor inconviniance, because the text is still readable. This image was taken with a DSLR camera using an ISO setting of 3200.

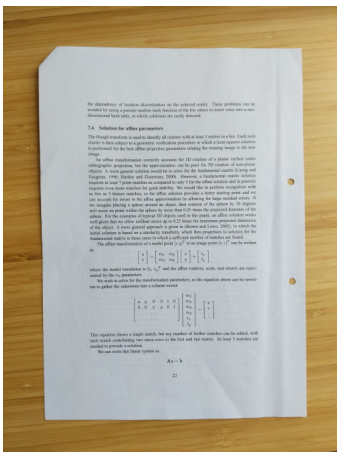
5 Known problems and possible solutions

Overall, our algorithm works consistently even under difficult lighting conditions or when the picture of the document was taken from an unusual angle. However, we also observed some use cases where it fails. If the document is placed on a low-contrast background, for instance a white desk (see Fig. 5a), our algorithm has problems detecting an edge where the document outline should be. In some cases it may help to do more preprocessing e.g. contrast enhancement. In its current state, our program uses the Canny Edge Detector only once with a fixed lower and upper threshold, because this works fine for the majority of images. Whenever the document localization fails, we could try to repeat the process with lower thresholds in order to increase our chance of success. This will introduce more edge artifacts in the edge image, but it might also be enough to detect the document outline.

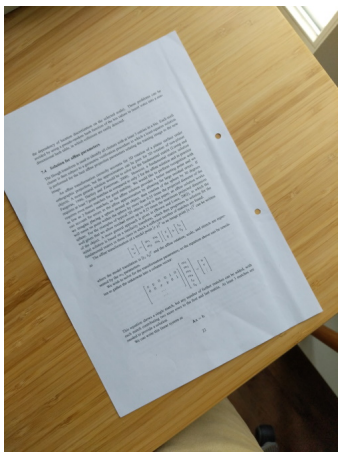
Another situation in which our program works poorly is with a bent document corner (see Fig. 5b). This may break the localization algorithm, because the detected contour would have to be approximated using five instead of four corner points. The same problem arises if the user accidentally cuts off a single corner from the image composition (see Fig. 5c).



(a) low contrast background



(b) a bent corner



(c) one missing corner

Figure 5: These are example images on which our algorithm does not work properly on.

For the last two problematic use cases, we have a possible solution in mind: Whenever the document localization fails, we could try a completely different approach using a Hough Transformation. Hough Transformations can be used to detect straight lines in edge images. Detecting the four most prevalent lines (with maximum-supression), corresponds to detecting the most likely document edges. The intersections of any two lines give us the location of each document corner. A bent corner or even a corner outside

of the image composition should be less of a problem when using the Hough Transformation approach. Nevertheless, it has its own downside, because the most prevalent lines have to be document edges and not for instance the edge of a desk. Therefore, the Hough Transformation would not replace our approach using contours, but rather act as a backup solution whenever our first document localization fails.

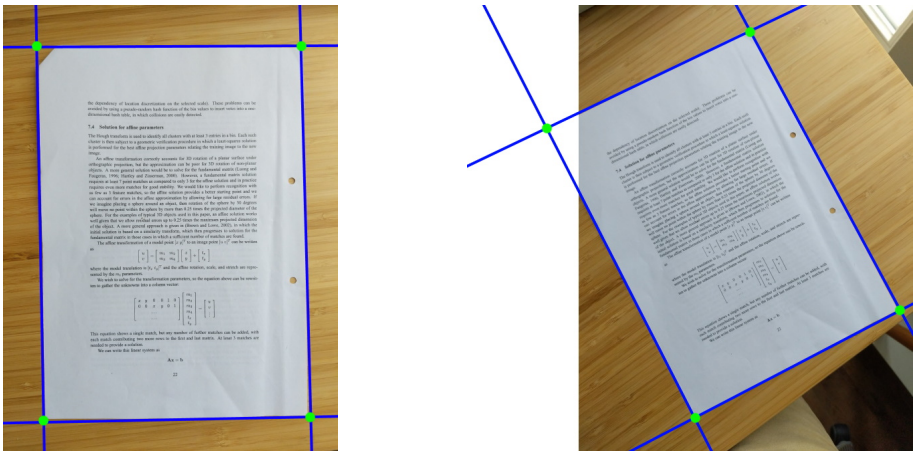


Figure 6: A Hough Transformation for detecting the four most prevalent lines (in blue). Using the intersections of these lines could potentially fix problems with missing corners. Both figures were created manually and are for illustrative purposes only.

List of Acronyms

OpenCV Open Source Computer Vision Library

DSLR Digital Single Lens Reflex

References

[GB22] Gaube and Betz. Document scanner. `https://gitlab.com/gaubeleo/document-scanner`, 11.05.2022.

[Ope22a] OpenCV. Canny edge detector. `https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html`, 11.05.2022.

[Ope22b] OpenCV. Opencv - about. `https://opencv.org/about/`, 21.04.2022.