CS4053 – Assignment 2

A New Leaf

Adam K. Schmidt – 12312907

Problem

Given scans of 13 pages of a book, identify which page of the book is open in an image. 25 test images of pages on a desk were provided.

Algorithm Summary

1. Read all images into memory, including a blue reference image for back projection
2. Resize the unknown images to speed up processing time (800*600 was found to be a good trade-off between speed and accuracy, any lower and the blue dots on the pages start to be missed)
3. Convert the blue reference image and the unknown images to HSV format so that we can compute a Hue-Saturation 2D histogram to use for back projection
4. Back project blue onto the unknown images
5. Threshold the Value channel of the histogram to isolate light coloured areas of the image.
6. Invert and subtract 5 above from the back projected image to isolate blue dots occurring in a light background
7. Iterate through each pixel of the resulting image searching for the left-, right-, top-, and bottom– most white pixels to take as corner pixels
8. Create a perspective transform matrix with the four points from above and warp the original (RGB) image using the matrix
9. Resize the image to be 100*100 pixels and the templates to be 103*103 to allow for the fact that there is a small white border around the template images.
10. Convert both templates and warped images to greyscale, and then Otsu Threshold them to maximise the difference between pages.
11. Template match each of the 13 templates to each of the warped page images and return the index of the best matching page

Technical Explanation

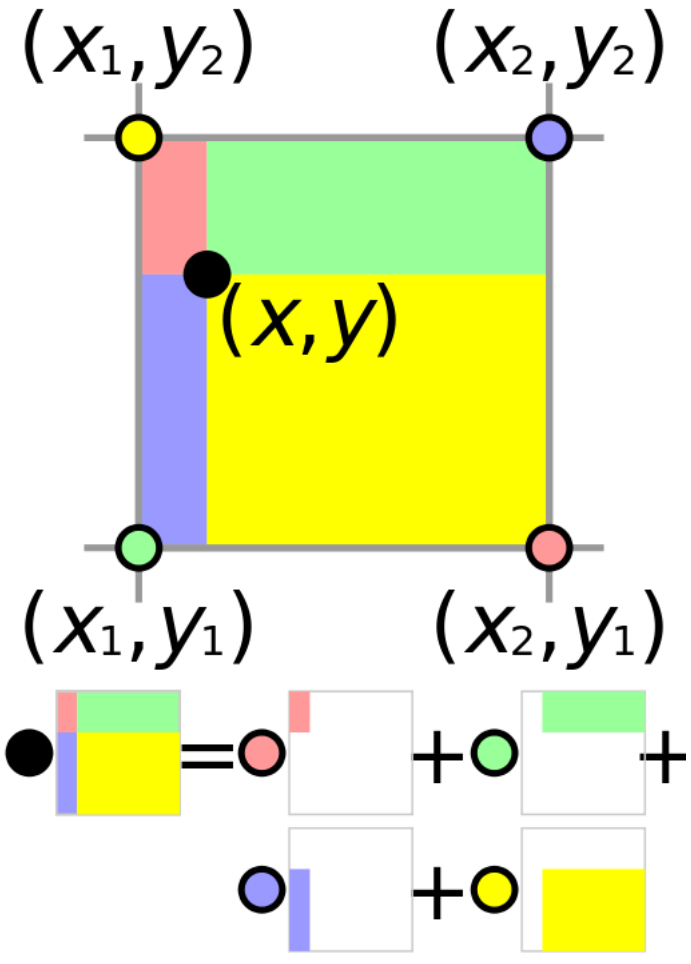
The problem can be split up into two main parts; extracting a page image from the unknown image and recognising that page.

Page Extraction

For this portion of the program, our input is an unknown image and our output should be the four co-ordinates of the page in that image. In this section we return four co-ordinate points even if there is no page in the image. The recognition section deals with images where no page is present.

Interpolation

We resize the images using a simple bilinear interpolation for speed. By reducing the number of pixels we have to deal with, we can greatly speed up processing time. This diagram, from wikipedia [1], presents a geometric explanation for what is happening:



In effect, we multiply the value of each pixel by the fractional distance along each axis to the interpolated pixel and sum the computed values together to get the value for the interpolated pixel.

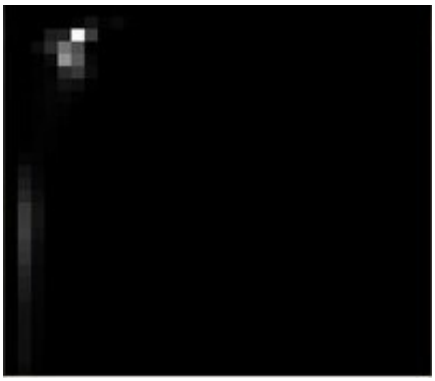
The actual formula, using the notation from above:

$$f'(x,y) = (x - x_1)(y - y_1)(f(x1, y_1)) + (x_2 - x)(y - y_1)(f(x_{2,} y_1)) + (x - x_1)(y_2 - y)(f(x_{1,} y_2)) + (x_2 - x)(y_2 - y)(f(x_{2,} y_2))$$

where f() denotes the value of a pixel and f'() denotes the interpolated value of a point.

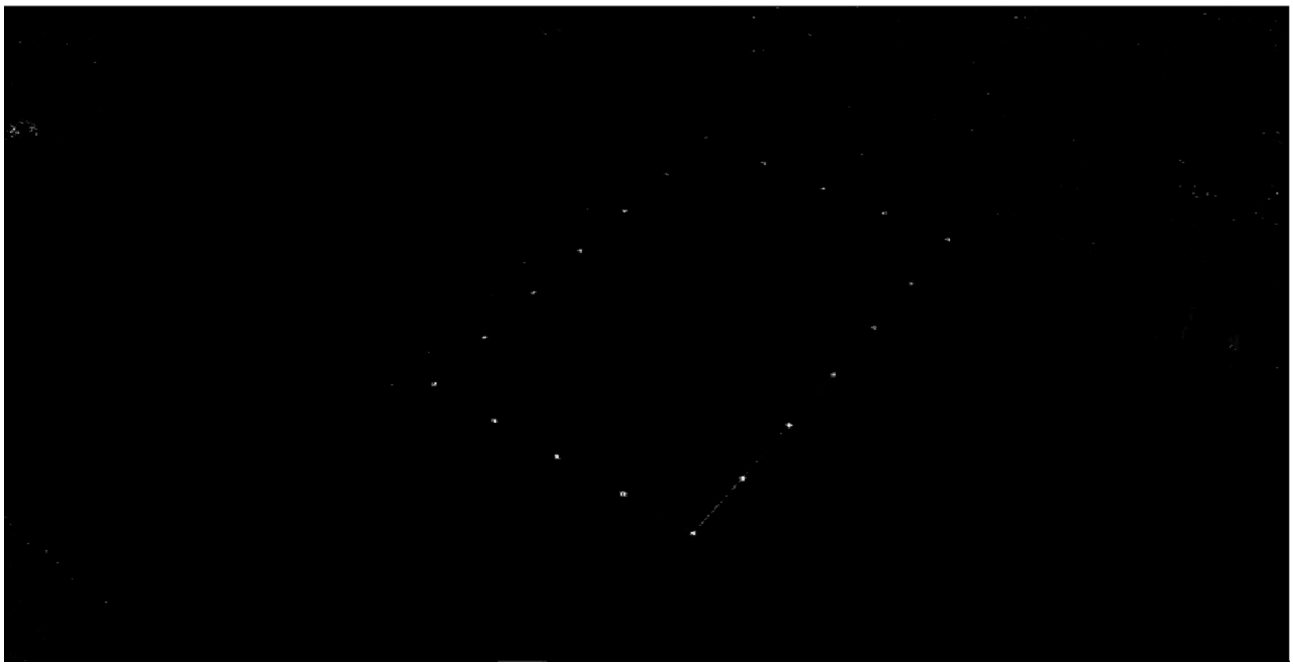
Back Projection

Next, we compute a Hue-Saturation 2D histogram of the blue reference image



An example 2D histogram [2]

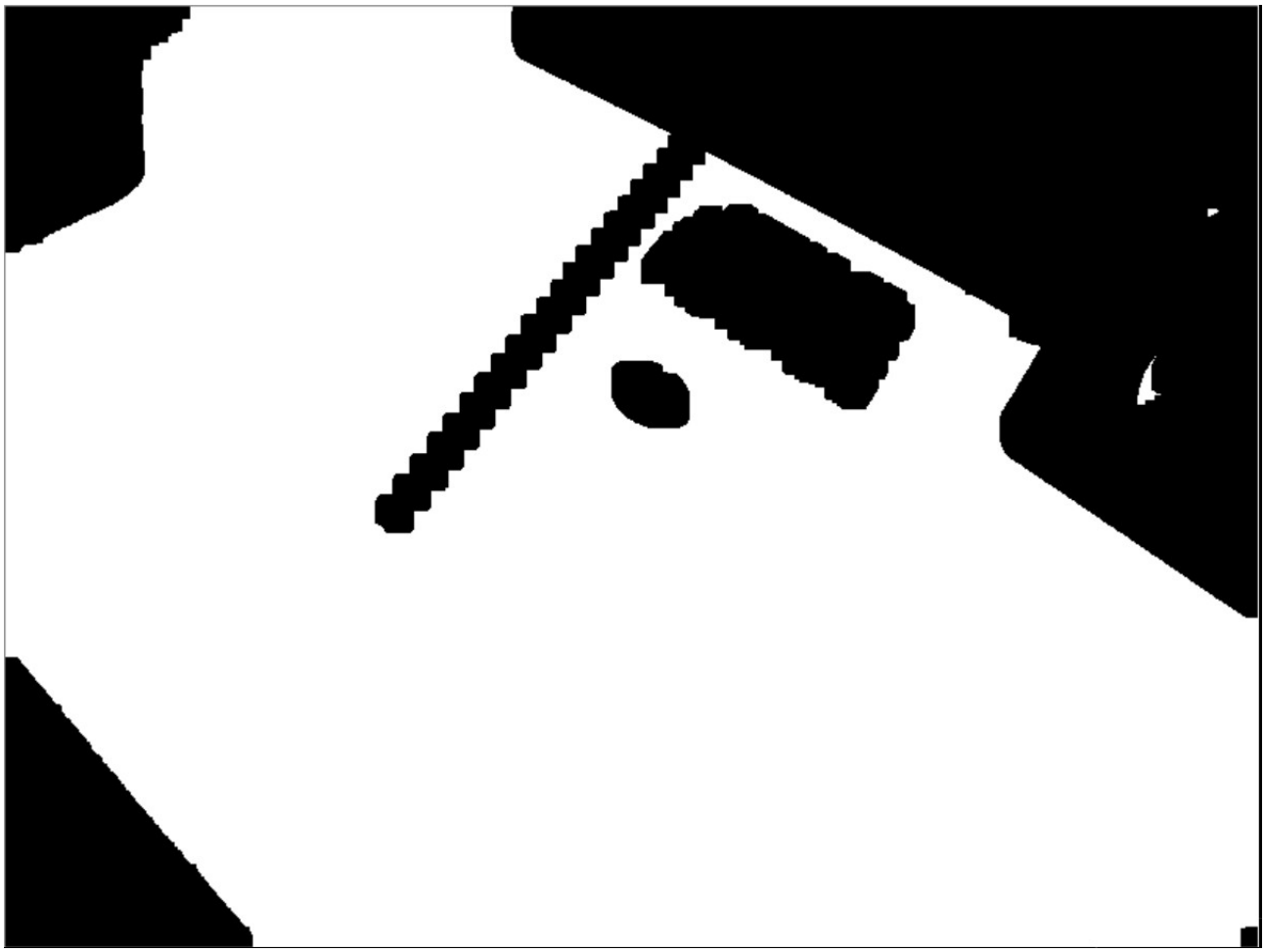
This calculates the number of pixels for each combination of Hue and Saturation values in a number of bins (in this case, we create a 32 * 32 histogram) and then normalises it so that it's highest value is 1. For each pixel in our unknown image we look up its Hue and Saturation value in the histogram and return the value we find there. This creates a “probability” image which is an image where each pixel is given a value based on how likely it is to be in the reference image (more accurately, how high its value was in the reference H-S Histogram). This creates an image like the below:



Note that there are pixels with high values outside of the blue dots on the page.

Masking

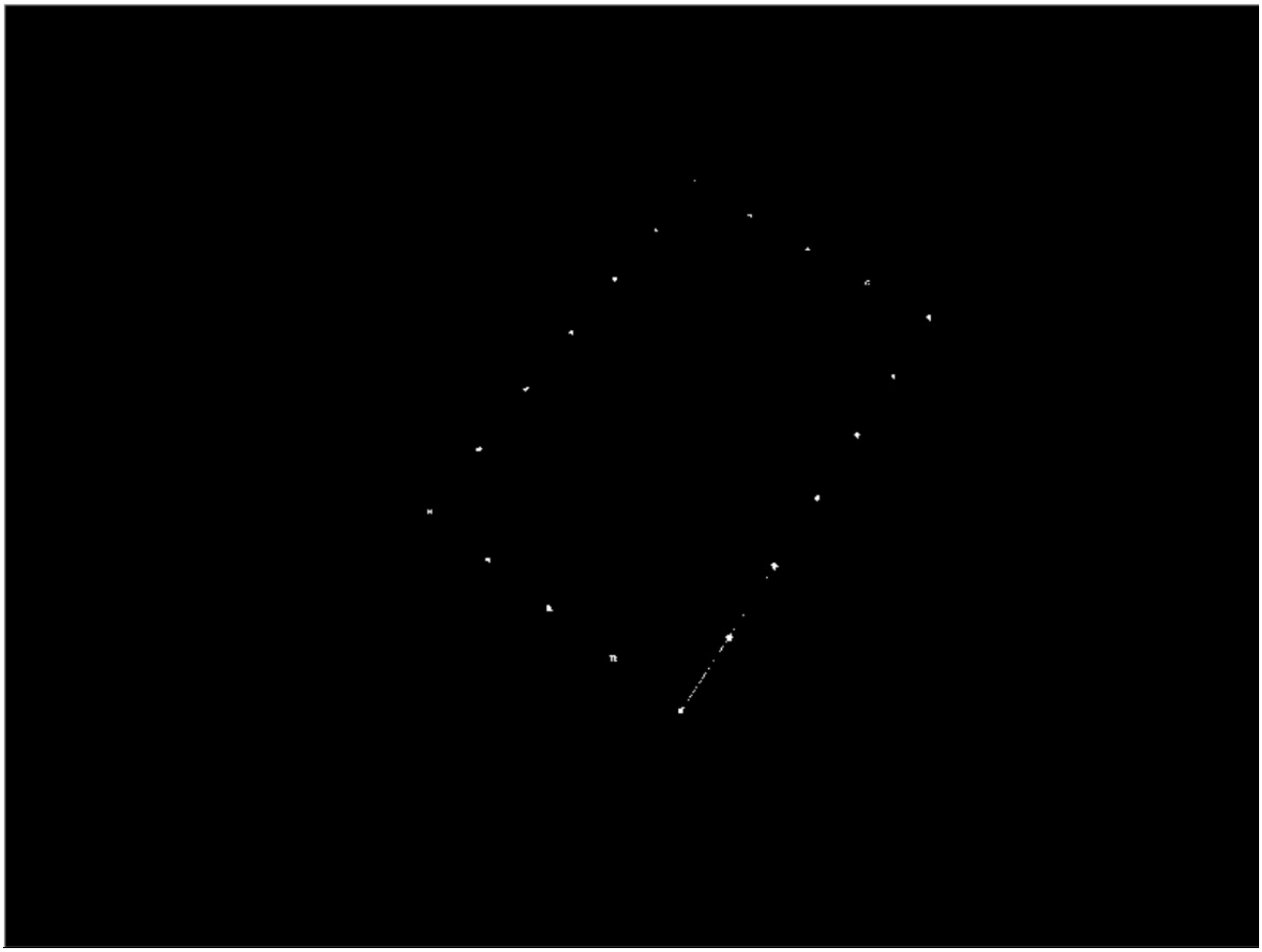
To get rid of these extra pixels, we Otsu threshold the value channel of the image. The value channel of the image should be almost the same as the greyscale of the image because it too measures the brightness of each pixel, rather than its colour. Otsu thresholding seeks to maximise the separation of background from foreground pixels by exhaustively searching for the threshold level which minimises the variance of values in the background and in the foreground. i.e. given a bimodal histogram it attempts to partition the histogram such that the variance of each partition is minimal. This separates our image into light and dark regions. We slightly dilate the image by setting all pixels which have a white pixel in the 3*3 grid of surrounding pixels to white. This makes the blue dots in the image (which are black after thresholding) become white. We then erode the image by setting pixels for which any of their 3*3 grid of neighbours are black to black. We do this multiple times to sufficiently erode away features like the binder clips on the book, which can sometimes be picked up by back projection as being close to blue. This results in something like:



Once we have this mask, we invert it by setting all black pixels to white and vice versa, and then subtract it from the back projected image from the step above. This lets us find blue pixels which are contained within a light region (hopefully the page). This masking makes a number of assumptions, chief among them being that there are no other blue dots within the light region. A more robust solution would be to use back projection to get the white pixels in the image instead, but I was unable to create a suitable reference image which reliably picked up the pages. Even if we could perfectly pick up the pages, the algorithm still assumes that there are no errant marks of, for

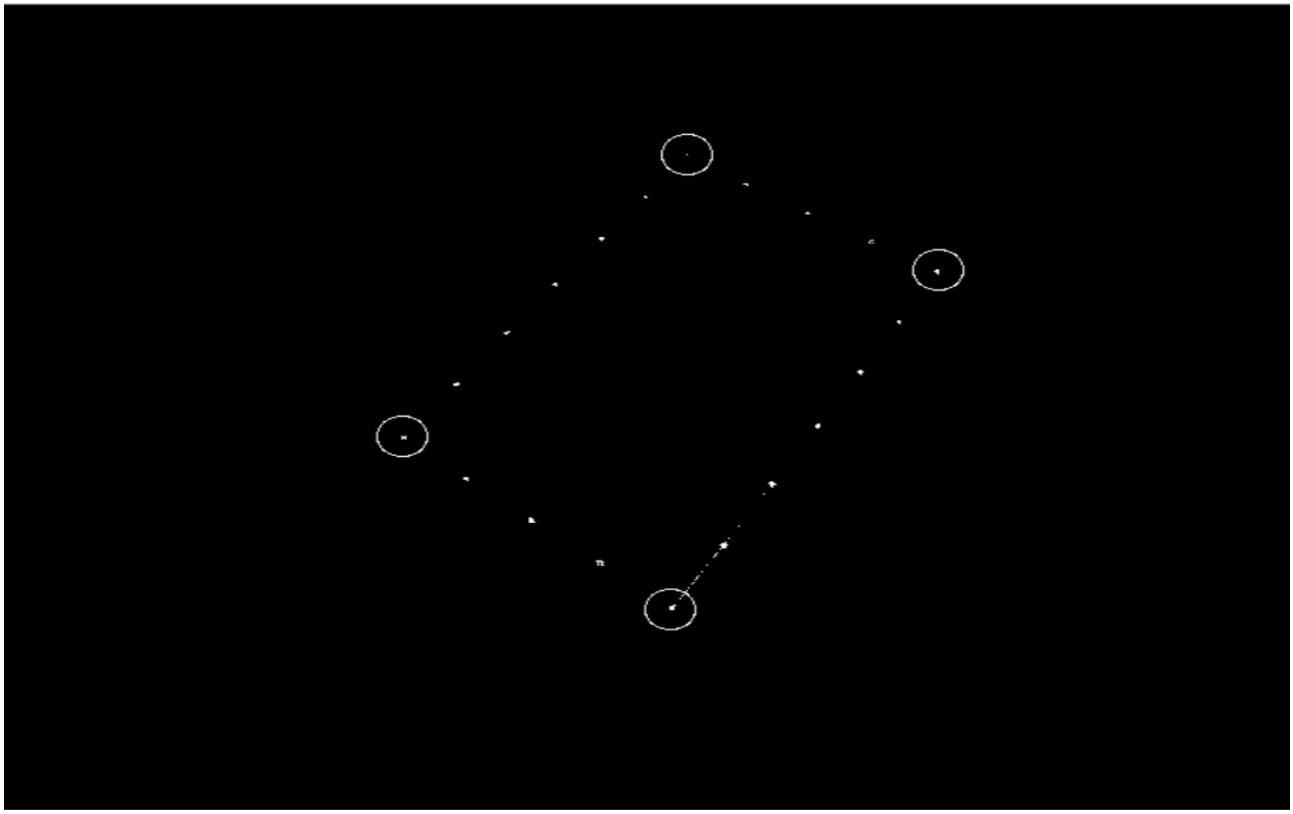
example, blue pen outside the blue border of the page. To get rid of those one might have to identify the rectangle of dots as a rectangle by identifying the lines they make and be able to extrapolate the corners from that. This would also allow for the corners to be covered while still correctly warping the page.

In any case, once we have masked the image we threshold it to remove low probability blue pixels and obtain something like the following:



Corner Detection

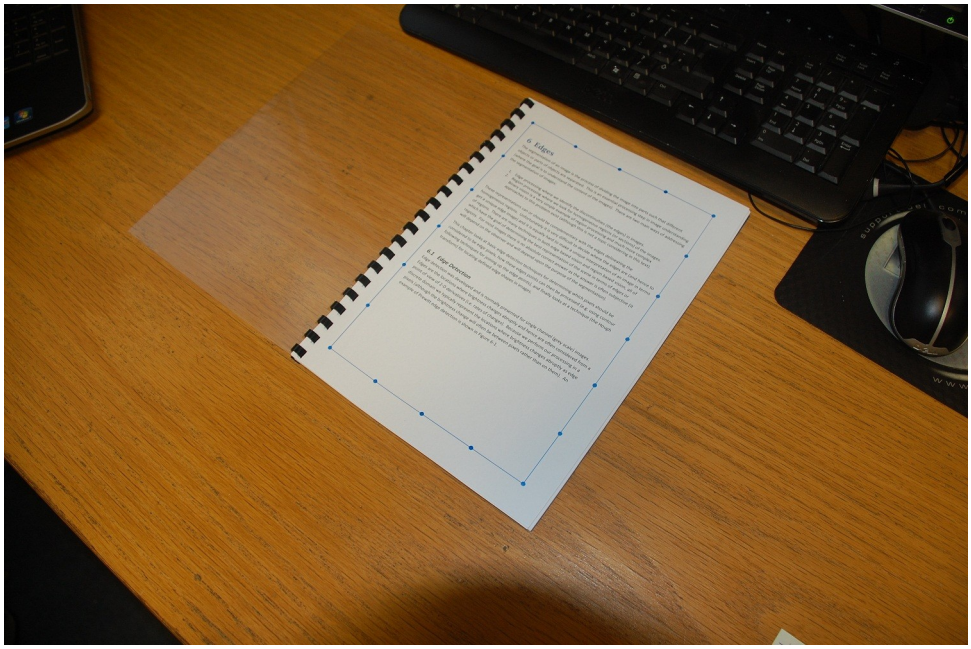
To detect the corners we simply iterate through each pixel of the image, recording the location of each white pixel if it is further left than our previously furthest left pixel (initialised to the right side of the image) and so on for the other three corners. Drawing ellipses around these points for illustrative purposes gives us:



This algorithm breaks down if there are errant pixels outside the blue border which managed to get through the mask. It also ceases to work correctly when the page is aligned with the image, i.e. when there is little rotation of the page. One would need to implement a different algorithm to cope with this case, for instance, calculating the distance from each white point to each of the corner points and picking the closest ones to each corner. This algorithm covers the cases where the first breaks down, and where it breaks down the first algorithm works, so together they cover the range of possibilities of rotation.

Perspective Transformation

We must transform the page in such a way that we can compare it to the scans of the known pages. By observing the four corner points above, we can create a perspective transformation matrix which will map each pixel on our output image to a point (not necessarily on a pixel) in the input image (the original image with the unknown page). The value of these points will be bi-linearly interpolated as above. This allows us to go from our original image to an interpolated version of the page present in that image.

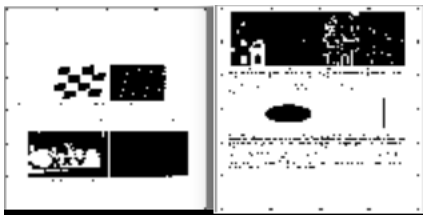


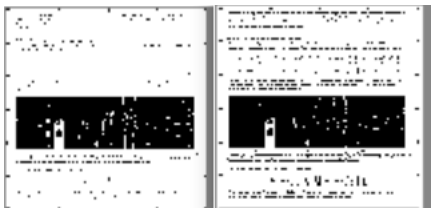
This image has been scaled down to speed up the template matching process and has been made square so that it is symmetric under rotation. This allows us to rotate the image by 90, 180, and 270 degrees to recognise images which are in different orientations.

Recognition

Preprocessing

As described above, the identified pages have been scaled down and made square. Each identified page is now a 100*100 square. We scale the templates down to be 103*103 pixel squares to allow for some small variation in where the corner pixels were detected and for the small white borders around the template images. We then convert both templates and identified pages to greyscale and Otsu threshold them to maximise the difference between pages. Since the pages are mostly text or images on a white background this step helps to differentiate pages which otherwise are quite similar, especially with the blurring introduced by scaling the images down so much.





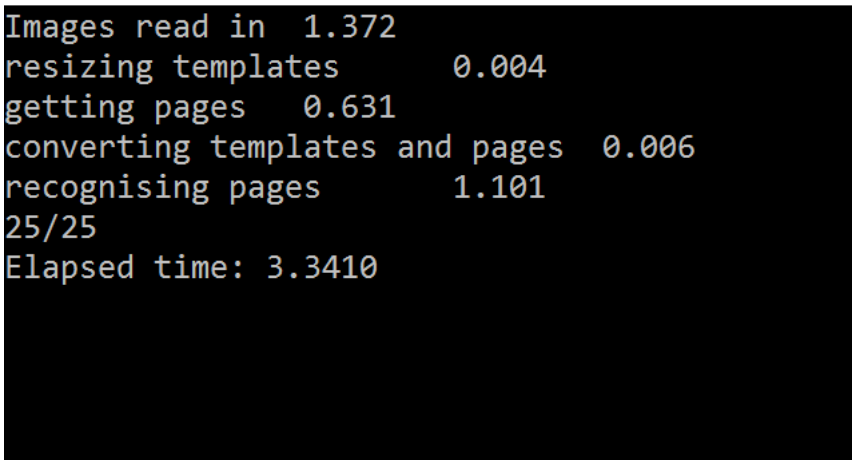
Examples of the images used for template matching showing two non-matching pages and two matching pages. Note how in the thresholding some points in the two matching images have been treated differently. This is a result of varying brightness levels etc in the original image and is to be expected. Matching pages are still much more similar than non-matching ones.

Template Matching

Template matching works by first creating a matching space which contains every possible pixel from which the template image can start in the image to be matched. It is important to note here that the “template image” for matching is actually the unknown page image, because it is smaller. In any case, because our canonical page image is 103 * 103 pixels and our to-be-identified page is 100 * 100 pixels, our matching space consists of a 3*3 grid representing the nine positions in the top left of the larger image where the smaller image could start and still be within the bounds of the larger image. This 3*3 grid is then filled with values corresponding to the fraction of pixels which are the same in both images when the smaller image starts at that position. We iterate through this 3*3 matrix to find the highest value and use this as our likelihood measure. Since there is no error checking before this stage in the algorithm, we need to discard those images for which no reliable match can be made before returning the index of the page with the highest likelihood measure. In the code this threshold likelihood value is set as 0.8 (i.e. 80% of the pixels in the template matched the search image).

We repeat this calculation for each of the 4 possible orientations of the identified image. (Note: Rotation is achieved through the use of the function provided at [3].) This allows us to identify images at all orientations.

Discussion



Speed

Console output with the DEBUG flag set to off, compiled in Release mode. The numbers following each line are the time taken in seconds for each of the steps for all relevant images (i.e. all 13 page images and 25 test images were read in in 1.372 seconds). As can be seen, not counting the time to load the images, the algorithm takes roughly 2 seconds to run over the 25 images. This is too slow to recognise the pages in video if we check every frame, but by taking only every second or third frame, we can get good recognition capabilities in very nearly real time. Since we are unlikely to

need to know what page is open 30 or 60 times a second, this is acceptable.

Performance

The algorithm correctly identified every one of the 25 test images, so its metrics are very simple to compute;

True Positive:	25
True Negative:	0
False Positive:	0
False Negative:	0
Precision:	1.0
Recall:	1.0
Accuracy:	1.0
F1:	1.0

Specificity is undefined as there were no true negatives or false positives.

Conclusion

The algorithm described above works very well in all cases of the test imagery, and should continue to work well for a variety of different inputs involving rotation about all 3 axes and differences in scale. It does not work when pages are squared with respect to the image plane, i.e. the edges of the image are parallel with the edges of the page, though an alternative method for detecting corners has been suggested in the Corner Detection section to correct this.

Since the images are scaled down so much before template matching, the algorithm should be relatively scale invariant. It will run into problems at large distances where blue dots may fall between pixel sensors and no longer be detected as blue.

The algorithm assumes that all pages are completely contained within the image, and that only one page is visible at a time.

Given these assumptions and caveats the algorithm works quickly and accurately enough to be used in nearly real time (~10Hz), for instance for educational software to display relevant links / videos when certain pages are open etc.

References

[1] "Bilinear interpolation visualisation" by Cmglee - Own work. Licensed under CC BY-SA 3.0 via Commons - https://commons.wikimedia.org/wiki/File:Bilinear_interpolation_visualisation.svg#/media/File:Bilinear_interpolation_visualisation.svg

[2] “Back Projection — OpenCV 2.4.12.0 documentation” http://docs.opencv.org/2.4/doc/tutorials/imgproc/histograms/back_projection/back_projection.html\

[3] function rot90() by TimZaman -- http://stackoverflow.com/questions/15043152/rotate-opencv-matrix-by-90-180-270-degrees