Visual Code Marker Detection

Ryan Coutts, Jessica Faruque, and Bragi Sveinsson

Abstract—**This report describes and analyzes the performance of an algorithm for detecting visual barcode markers from cell phone camera images. Topics include feature detection, adaptive thresholding, interpolation, and testing methods.**

I. Introduction

Visual codes can be used to label and identify numerous objects. One method of reading visual codes involves taking pictures of the code markers with a low-resolution camera to identify and perhaps find out relevant information about an object. However, recognizing a bar code from surrounding objects and image artifacts can be a challenge. To successfully implement a visual code detection system, image processing algorithms are required. This report discusses one algorithm for bar code detection.

The project code performs two main functions: finding the marker locations and finding the pixel values from the marker locations. Implementation of these functions requires feature detection, adaptive thresholding, and linear interpolation.

The results obtained from implementation of the algorithm were excellent for the training images provided on the course website, and were very good—though not perfect—for other sets of images. These results, and considerations about the time required to perform the algorithm, are discussed.

II. Finding The Marker Locations

The function to detect the markers is called findmarkers.m. It takes the following steps:

- Produce a black and white version of the given image. The built-in Matlab functions for doing this can't be used, since they use a global threshold for determining if a point should be black or white. Instead, adaptive thresholding is used, where a small window is moved over the image and the threshold calculated for each such window. This is done using the function adaptivethreshold.m. By trial and error, the best window size for a typical image was determined to be 11x11 pixels. However, if an image is taken very close up to a code marker, this image size will fail. Therefore, findmarkers.m increases the image size to 22x22 pixels if it doesn't detect anything and tries again. If nothing comes up from that either, it tries a 33x33 pixel window, and finally a 44x44 pixel window before it finally gives up.

- Label the regions of the black and white image using bwlabel (the image is actually "flipped" first, i.e. black regions turned into white regions and vice versa, because the black regions should be labeled, not the white ones).

- Then find the regions which are possibly guidebars, using the function findguidebars.m. This function looks at the size and shape of each region and determines if it could possibly be a guidebar.

-The criterion for the size is that the region is not bigger than 7/121 of the image (this upper bound is reached if the marker completely fills out the image) and not smaller than 10 pixels (if it was any smaller, the marker would be impossible to read anyway).

- The criteria for the shape are that the ratio of the maximum and minimum axis of the region is no bigger than 11.5, but no smaller than 3.5. These values were determined empirically. Of course, if the image was perfect and the camera held at a 90 degree angle from the marker, the ratio should be 5 and 7 for the two guidebars. However, this is seldom the case, since the image is of low quality and often taken at a more obtuse angle.

-The major and minor axes of each region are found using the function ellipse_param.m, which returns the ratio of the axes, the values of the axes and a vector parallel to the major axis. These values are found using the second values of the x- and y-values of the pixels of the regions [1]. The following image shows possible guidebars for training image number 3.

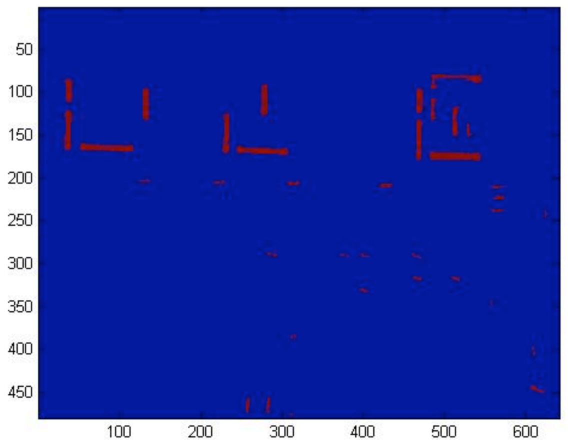


Figure 1: Possible Guidebars for Training_3.jpg

- Then try to use these guidebar candidates to detect the corners of code markers using the function findmarkercorners.m, which does the following things:

- Find the corners of all the possible guidebars found before using the function findregioncorners.m, which calls the function findcorners.m for each possible guidebar.

- Find all regions that are possibly dots using the function finddots.m. It works basically the same as findguidebars.m, except here the maximum/minimum axis ratio is required to be less than 1.8 (a value found empirically) and the minimum allowed size is lowered to 5 pixels. The following image shows possible dots for training image number 3.

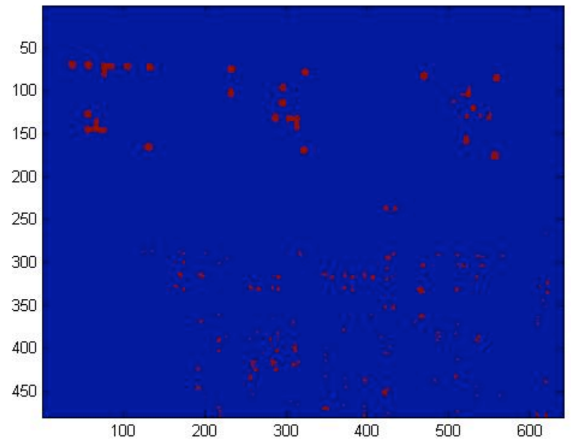


Figure 2: Possible Dots for Training Image 3

- Find all guidebar candidates which could possibly be a pair of guidebars belonging to a single marker using the function findbarpairs.m. This is done by comparing guidebar candidate no. i to guidebar candidates no. i+1,…,N for i between 1 and N-1, where N is the number of guidebar candidates. If the two guidebar candidates being compared have sort of the right axis ratio (the ratio of the longer bar length to the shorter one should be less than 3), the right size ratio (one bar should not have more than twice the number of pixels of the other bar), are sort of correctly aligned (the angle between the bars should not be less than thirty degrees) and are close to each other (in the sense that a corner of one bar is within length L of a corner of the other bar, where L is the length of the bar), the two bars are labeled as a possible pair. All of these threshold values for determining whether we have a pair or not were determined empirically. This grossly overestimates the number of guidebar pairs, but later checks ensure that bad pairs are thrown away. The important thing is that no true guidebar pairs are left out.

- Make an estimate of where the corners of the code marker should be for every guidebar pair. This is done by starting at the point on the vertical bar (looking at the code marker the normal way, as shown on the class website) which should be closest to the upper-right corner, then travelling the distance H in the direction of v_v, where H is an estimate of the "height" of each square of the marker and v_v is the orientation vector of the marker, found using the function ellipse_param.m (it is made sure that v_v points in the right direction). This gives an estimate of the upper-right corner of the marker. Likewise, the distance L is traveled in the direction of v_h from the point on the horizontal bar which should be closest to the lower-left corner, where L is an estimate of the "width" of each square of the marker and v_h is the orientation vector of the horizontal guidebar. The lower-right corner should simply be the lower-right corner of the horizontal guidebar. The last corner point is estimated by using the three other estimated points and assuming the marker to be a parallelogram.

- If the estimated corners are not out of the image boundaries, the algorithm tries to adjust the points to their right values using the function fixpoints.m, which does the following:

- The true value of the upper-right corner should be in the center of a dot. So an ellipse is drawn around the estimate of the upper-right corner, with the major axis parallel to the vertical guidebar. This is done with the function alignwithdot1.m. The ellipse is made big enough to include the true right dot, unless the estimate is very far off. If multiple dots are found within the ellipse, the dot closest to the estimate of the lower-right corner is chosen. If we can't find a dot, or the dot chosen results in an error (see below), that pair of guidebar candidates does not give a code marker and is thrown away.

The same is done for the lower-left corner, i.e. an ellipse is drawn with its major axis parallel to the horizontal guidebar and any dots within that ellipse detected. This is done with the function alignwithdot3.m. The difference is that if multiple dots are detected, then the dot furthest away from the estimate of the lower-right corner is chosen first (since there can be dots between the lower-left corner dot and the horizontal guidebar). If the choice of this dot doesn't result in an error it is as a corner, otherwise the same procedure is repeated for the dot to its left. If all the dots give errors or no dot is found, the guidebar pair is thrown away.

Finally, the two dots found (the upper-right one and lower-left one) and the lower-right corner estimate, which we don't alter, are used to estimate the location the upper-left corner dot by assuming the marker to be a parallelogram. A circle is drawn around this estimate and the dot inside that circle which is the furthest away from the lower-right dot is chosen. The following figure explains this more graphically.

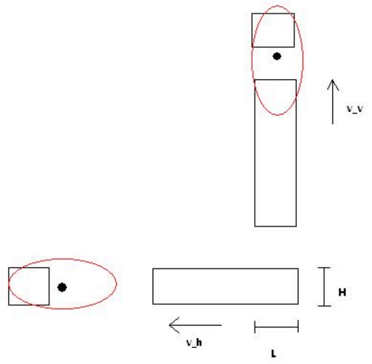


Figure 3: Selection of Corner Points

- After these dots are found, the algorithm checks whether the dots are roughly the same size, and whether the smaller guidebar is at least 2.5 times bigger than the smallest dot. It is also made sure that the same dot hasn't been counted twice. If this seems all to be in order, the four points are returned as the location of a code bar.

- This algorithm throws out almost all things mistakenly detected as a marker. Some things occationally slip through, but they are taken care of when the marker is read, by checking the fixed values of the markers.

III. Validating Detected Markers

The methods and parameters used in Part II caused points to be detected that were not actually markers, or false positives. In these cases, the false positives shared some similar features with the regions containing the actual markers. It is preferred that the feature detection algorithm included false positives rather than miss some actual markers. Nevertheless, including the coordinates of the false positives in the processing for determining the element values causes a large number of erroneous results. To account for this, some further operations were implemented to exclude these false positives. Two different sets of checks were implemented: a set of simple preliminary checks, and a more detailed check comparing fixed element values.

Initially, three preliminary checks were performed using the four coordinates of the supposed marker. Comparing the fixed element values involved dividing the region surrounded by the coordinates into an array of 121 elements and sampling the fixed elements. This involved some pre-processing and refining the coordinates, which will be discussed in more detail in Section IV. In the 121-element array, there are 38 fixed elements if the white spaces are included.

The decision made using the fixed elements is binary: if the elements in the test array match the 38 fixed elements, then the test array is actually a marker; otherwise it is not. When this algorithm was implemented with the requirement that all 38 fixed element values are correct, some of the actual markers were found to be false positives. To prevent or reduce this problem, it was instead required that only a fraction of the elements should match. In the code implemented, the determination that the array was actually a marker required that any 24 of the 38 locations matched the fixed element array. This number was empirically chosen based on the training images and some additional images taken by the authors of this report. This threshold allowed all of the correct markers in the training images to be detected while excluding the erroneously detected marker coordinates.

IV. Determining Pixel Values from Markers

To determine the pixel values and obtain the 83-element results, the following steps were used: refining the coordinates, creating an array of coordinates, sampling the array elements, and comparing the results with the original array elements. The detailed check for false positives mentioned in the previous section is also done in the function call used for this part of the algorithm. This was because the check made use of the 121 element array generated for finding the 83-element results.

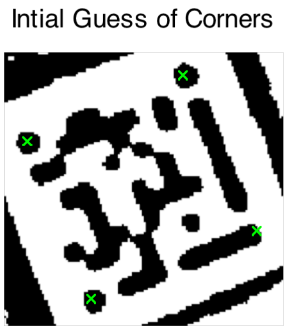
A. Pre-processing

It was found that some pre-processing of the coordinates were required to accurately find the pixel values.

1) Image thresholding: Like Section I, this part of the program used adaptive thresholding to obtain a black-and-white image of the markers. However, the threshold values determined in Part I were not used, since better adaptive thresholding could be performed after the size in pixels of each region containing a marker was determined. In the adaptive thresholding, the local threshold value was determined over 0.3, or approximately one-third of the size of the region containing each marker. This value was determined empirically. See the next section for a detailed discussion of adaptive thresholding.

2) Coordinate Refinement: To ensure that the detection of marker elements is accurate, further refinement of the four coordinates from part I was performed. The coordinates from Part I provided an approximate location of the corners of the image; this code found better estimates of the corners of the image.

Before creating a grid, the corner points required more accurate refinement in order to provide better results. The points that are found from the barcodes are the center points of the square dots around the edges and a dot on the short bar that is somewhere in the corner. As can be seen in the figure, the corners are initially off by a lot. Then the program starts by finding all 5 of the stationary marker points. Then from the center of the square points it moves towards the edge of the dot until it finds the edge. Some morphological operations are used for this process. For the non-square guide bar it is not as easy to find a starting point, such as the center of the squares, so what the program finds the center of the long bar and the center of the short bar. Then it extrapolates a line from the corner dots through the non-square bars such that the intersection of the 2 lines intersect in the bottom right corner of the guide bar which is a pretty good estimate of where the center would lie if it were a square dot. Essentially, the program draws a border around the barcode after connecting the dots that were found. The program then finds the intersection of these dots, creating better corners. In order to improve the accuracy the program, shrink the box by traveling in the direction of the point diagonal to it until it hits a black region. This last step was critical to an improved bit error rate.



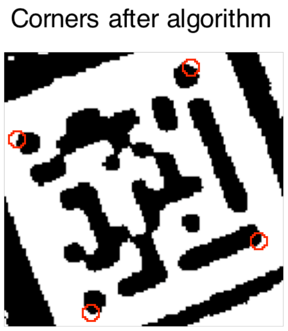


Figure 4: Corners Before and After Corner-Enhancing Algorithm

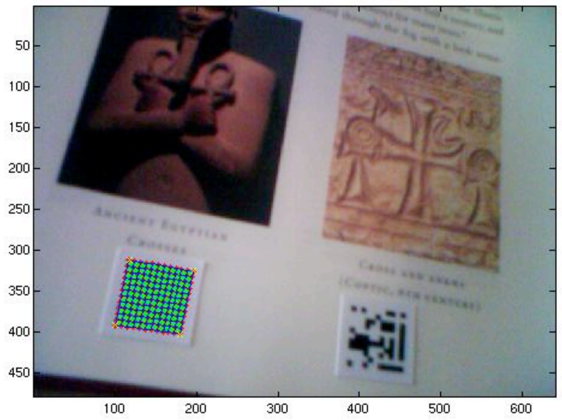
Two cases were possible in the thresholded image for each of the corner elements: either the coordinate is black or white. If the coordinate was white, then the program used the nearest black pixel in the direction towards the center of the marker as a new approximation for the corner. Empirically, this was shown to produce better results for all test images.

B. Creating an Array of Marker Element Values

Once the four corners of the marker were determined to reasonable accuracy, they were interpolated linearly to find the approximate center point and four corner points of each element in the bar code marker. Essentially, a grid of 121 squares was created using the endpoints using linear interpolation.

C. Sampling the Array Elements

To obtain the results, a few different schemes were attempted to find the element value (1 or 0) of each square in the 121-element array. These were: use of the nearest pixel to the center sample, bilinear transformation of the center sample value to the four closest pixels, and averaging over a group of pixels around the sample. All three methods produced approximately the same error rates when comparing with the ground truth values. In the interest of saving processing time, the nearest pixel to the center sample was used. A sampled image is shown below.



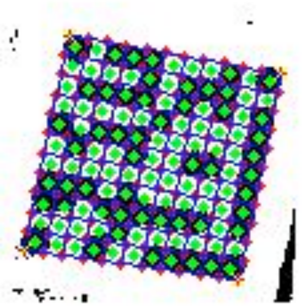


Figure 5: Sampled Image in Original Context; Close-up Thresholded Sampled Image

D. Comparing with the Ground Truth Values

Using the thresholded image and the 121 element grid, a 83-by-1 matrix of binary values was created to represent the ground values in the 83 critical squares. The ordering of the elements was the same as in the ground truth matrix given, and bit-by-bit comparison of the results was performed.

V. Adaptive Thresholding for Gray Scaling

Due to some of the various shadow gradients that are apparent in the camera phone images when global thresholding is applied to the images, if there is a shadow on the marker, it is deleted from the picture. The solution to this problem was to use adaptive thresholding. This method takes a window of a given size and finds the threshold that would be used for that window individually. Then the window snakes around the image either incrementing or decrementing the threshold value based on the previous threshold value and the current threshold value of the window. This method allows the visual code markers to be found more easily and consistently. An example of the usefulness of this application is shown.

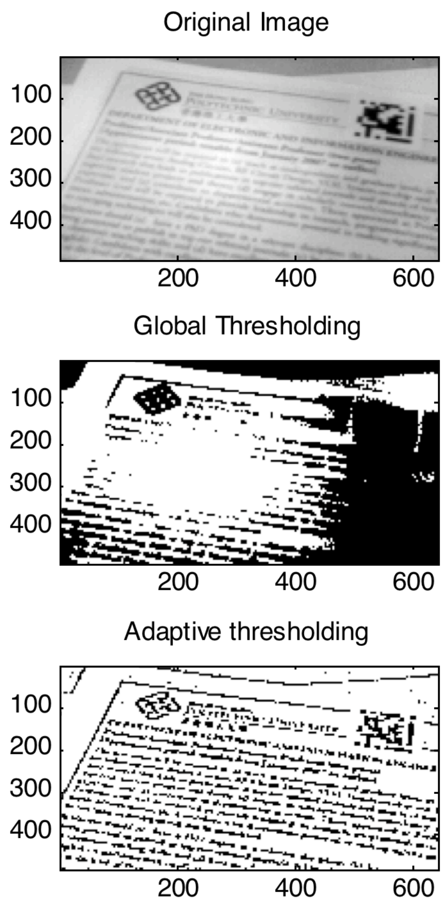


Figure 6: Different Thresholding Methods

VI. Testing Methods

In order to test the algorithm, the training images provided on the class website were first used. After successful detection of these images, the program was tested on 25 additional images taken by the authors of this report. These test images were much more difficult to detect than the images provided on the course website. For instance, none of the provided code markers were upside down. None of the markers were very small or very large compared to the image size. Another consideration was whether the markers resembled other features that were present in the stationary points. A couple of images from the testing stage are shown. The program is relatively successful in detecting only the real barcodes in these images.

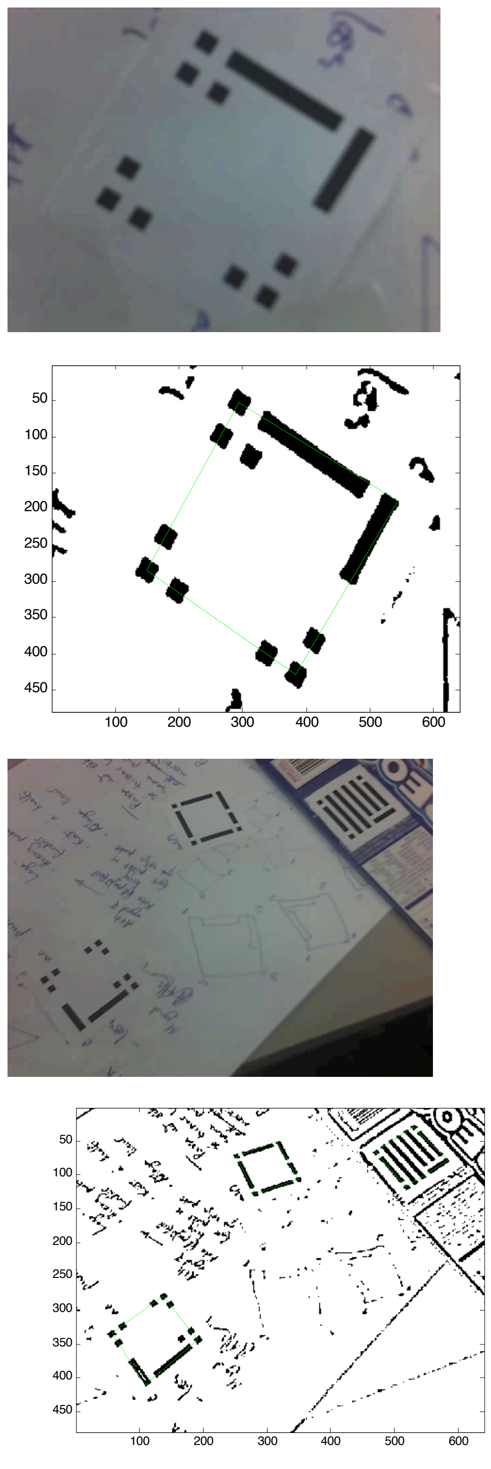


Figure 7: Two Test Images and Thresholded Results

VII. Timing Considerations

The time taken to process was not considered as critical as the accuracy of the detection. Once the code of the plots and data used for error checking was removed, the time taken to process each image was reduced. The final times for each image are shown in the plot below.

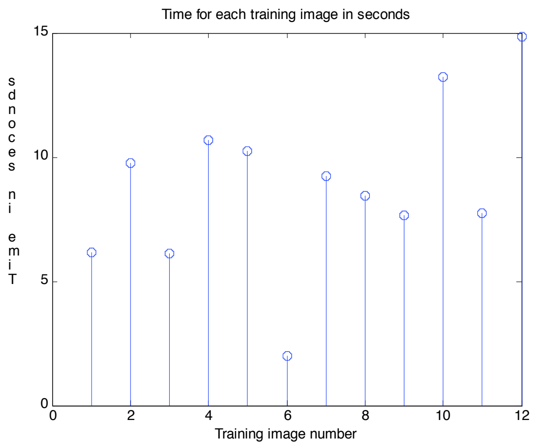


Figure 8: Timing Used for Training Images

VIII. Discussion

The results from our finished program with the test images are excellent. First of all, the score from the evaluate function is 1909, which is 100 percent correct. When tested on the 25 additional images, the results were relatively satisfying, though not perfect. Specifically, there were some barcodes that were not on flat paper, so the assumption that the barcode was a parallelogram was not satisfied. Also, some of the barcode images were very small so the program did not detect them.

IX. Conclusion

Finding the bar code markers in a random image is not a trivial task. There are many considerations that must be accounted for in order to have a successful detection and reading of the marker. Such considerations were how to convert the image to a black and white image to find the barcodes, how to find the barcodes, reading the barcodes, and checking to make sure it really is a barcode. Implementing this algorithm required making sure that the program does not have any errors no matter what the input image is, as long as it is the correct format. The algorithm was designed to not miss anything so there are many checks in place that do take up some CPU time, which may slow down the algorithm. However, these checks are essential in successful implementation of the program.

Appendix: Division of Labor

Ryan Coutts: Boundary box reshaping for marker, integration, image quality enhancing, adaptive thresholding, creating additional test images, haralick corner detector, finding marker center.

Jessica Faruque: Finding the pixel values from markers, image quality enhancing, integration and testing, checking if marker locations are actually images, bilinear interpolation.

Bragi Sveinsson: Finding the marker locations.

References

[1] Rohs, Michael: Real-World Interaction with Camera-Phones, 2nd International Symposium on Ubiquitous Computing Systems (UCS 2004), Tokyo, Japan, November 2004.