Visual Code Marker Detection

Daniel Blatnik, Abheek Banerjee EE 368, Spring 2006

***Abstract*—This report discusses the algorithm we implemented to identify and read data from visual code markers. In addition to outlining the algorithm steps, we summarize the results our algorithm achieved on a set of training images.**

I. Introduction

The problem we address is as follows. Given a JPEG image scattered with visual code markers (Fig. 1), we wish to determine the coordinates of the center of the upper-left square of each marker, as well as the bits encoded by each marker. Our strategy was to identify possible guide bars through region labeling followed by a series of checks on each pair of regions. To keep the execution time as short as possible while limiting false positives and negatives, we tried to strike a balance between robustness and efficiency. After finding a pair of guide bars, we identify the four corners of the code marker using data extracted from the Radon transform of the guide bar regions. Finally, we apply a projective transform to map the four points, which form an arbitrary quadrilateral, to a square in the conventional x-y grid. The data bits can then be read by simple thresholding.

Figure 1: Input image containing three markers.

I. Algorithm Description

A. Filter/Thresholding

We start by converting the image to grayscale by retaining only the luminance of the original image. Color is not an immediate concern since our first goal is to label the dark regions in the image, among which will be the guide bars. However, color information will be useful later for region elimination.

Next, we filter the grayscale image with a 9x9 Laplacian of Gaussian filter with $\sigma = 1.4$ to sharpen edges. The filtered image is then thresholded near its mean (which is roughly 0), giving us an image of all the prominent dark regions, such as that shown in Fig. 2a.

B. Region Labeling and Region Removal

The next step is to detect all of the dark regions in the image. We use 4-connectivity rather than 8-connectivity because we expect all pixels in a guide bar to be well connected to others. Next, before entering our pairwise guidebar search, we attempt to remove as many regions as possible that clearly do not fit the characteristics of a guide bar. The pairwise search will consume almost all of the execution time of our program and will have $O(n^2)$ complexity, so removing roughly 29% of the regions, for example, would make our algorithm twice as fast. We choose to eliminate based on size any region smaller than 50 pixels or larger than 8000 pixels. The lower bound is based on the size of guide bars in the 12 training images that we were provided, in which we found no guide bar regions containing less than 100 pixels. The upper bound is based on the largest possible code marker that would fit in a 640x480 image, which would contain roughly 480x480 pixels. Its short guide bar could consume no more than 5/121 of these pixels, leaving 8000 as a reasonable upper bound.

The next characteristic of guide bars that we take advantage of is the fact that they are black regions on white backgrounds. Looking now at the red, green, and blue color components of the pixels in the region, we remove a region from consideration if the mean squared distance of its color components from the average of its color components exceeds a threshold. This works since we expect the RGB components of a black pixel to be very near each other.

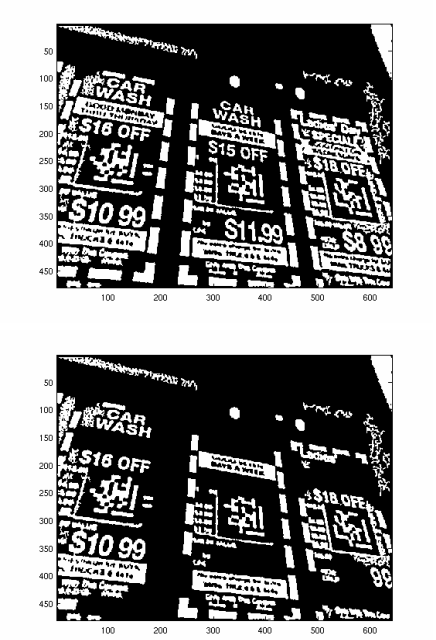

Figure 2: Demonstration of color removal, for training image #5. Thirty regions were rejected on the basis of color (bottom) *that we could not reject on the basis of size* (top)*. This reduced the total number of regions from 149 to 119, for an expected reduction in execution time of 1-(119*117/2)/(149*147/2) = 36%.*

C. Iterative Guide Bar Search

After filtering, thresholding, and region deletion, we are now ready to search for pairs of guide bars. This involves iterating over each pair of the remaining regions, putting each pair through a series of tests to eliminate any false positives. To eliminate the most obvious non-guidebar regions from consideration with the least computational expense, we start by checking the ratio of the number of pixels in the larger region to the number of pixels in the smaller regions. In a code marker's original grid, the long guide bar consumes seven squares while the short guide bar consumes five squares. Based on their lengths, we expect a ratio of the larger bar over the smaller bar of about 1.4. If the ratio lies too far away from this value, then we can eliminate the pair from consideration.

We also reject from consideration any two regions that lie too far apart from one another. We use two measures of closeness: midpoint-to-midpoint distance, and the distance between the closest pair of points from each region.

The next characteristic of guide bars we use is that each bar should possess a single dominant edge orientation angle. Regions containing a varied mix of edge orientation angles, can thus be eliminated. To determine the approximate distribution of edge orientation angles in a region, we take a rectangular subset that closely surrounds the region from the thresholded image. We perform edge detection on this black and white subset using the Canny detector. We chose this edge detector because it is computationally fast, and because we are not worried about edge connectivity; we only need to know the rough distribution of edge orientation angles in a connected region. To estimate this distribution, we apply the Radon transform to the edge-detected subset, thus giving the strength of lines in the subset at each (ρ, θ). Because we again are only interested in the dominant angle, so we sum the Radon matrix over all ρ for each θ, giving a histogram of edge strength vs. θ. In a true guide bar, the peak angle in this histogram will have much more energy than the rest of the spectrum of angles. A region is eliminated if its distribution of edge strength contains too much energy at angles other than the peak angle. Examples of this histogram that suggest both rejection and non-rejection are shown in Figure 3.

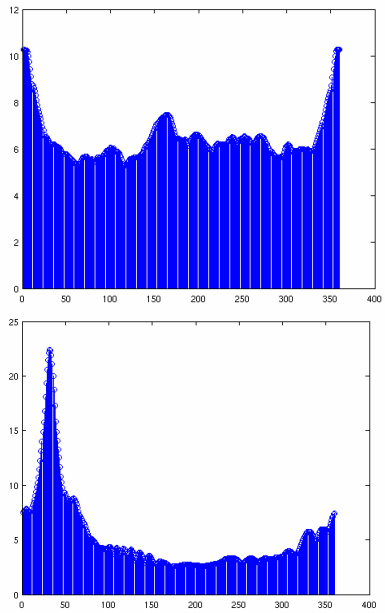

Figure 3: Demonstration of our edge orientation angle criterion. The top image corresponds to the letter 'O' in the word 'OFF' on the left of Figure 2, which we did not reject on the basis of region size, but were able to reject because it contained too much energy at edge orientation angles other than the peak angle. The bottom image corresponds to a long guide bar. Note the strong peak.

Furthermore, a pair of regions is eliminated if the difference between the two peak angles lies too far away from 90 degrees. We do not enforce a particularly strict threshold here either, allowing any pair of regions whose dominant angles are 60 to 120 degrees apart to pass through. This tolerance is necessary because depending on the camera perspective, a square-shaped code marker will be transformed to an arbitrary quadrilateral with two or more non-right angles.

Our final check measures how rectangular each region is. This is accomplished by rotating the region-labeled image by the peak angle determined as above, and then identifying the smallest possible rectangle in the natural grid of the rotated image that fully contains that same region. If the region is a

true guide bar, it has now been rotated back to its natural orientation, and should occupy most (at least two thirds) of the rectangle. If it occupies less, it presumably is too convoluted in shape to be a guide bar.

D. Orientation Identification and Corner Search

The lower right corner of the code marker always lies on one of the four edges of the smallest possible rectangular subset fully containing the shorter guide bar. It also lies on one of the four edges of the smallest possible rectangular subset fully containing both guide bars, unless the marker was greatly skewed due to the camera perspective. We identify the lower right corner by essentially looking for the point in the smaller guide bar region that lies on an edge of both the smaller marker's surrounding rectangular subset, and the rectangular subset surrounding both guide bars.

Knowing the coordinates of the lower right corner, we traverse to the other three corners as follows. We use the length and orientation of the long guide bar to estimate the distance and direction that we must travel, starting from the lower right corner, to get to the upper right corner of the code marker. Similarly, we use the length and orientation of the short guide bar to estimate the distance and direction that we must travel to get to the lower left corner. Addition of these two vectors gives the coordinates of the upper left corner, assuming that perspective skew has transformed the shape of the marker to a parallelogram.

This traversal gives rough estimates of the corner locations. Because code markers have isolated black squares at their upper left, upper right, and lower left corners, these estimates can be made more accurate by taking a 10x10 subset around the estimated corner location and choosing the location from that subset with the minimum pixel value, which generally should correspond to the center of a black square.

E. Mapping Back to a Square

Once we have the coordinates of the four corners of a marker, the next step is to undo any transformation of coordinates by mapping all the points from the found quadrilateral to a square. To do this, we need a set of equations that describes a projective transformation from one plane to another:

$$x' = \frac{a_1 + a_2 x + a_3 y}{1 + c_1 x + c_2 y} \qquad y' = \frac{b_1 + b_2 x + b_3 y}{1 + c_1 x + c_2 y}$$

Next, since we have four sets of original points (the warped corners in our perspective image) and four sets of destination points (the unwarped corners in our square image), we have 8 equations and 8 unknowns—solvable as a system of equations using matrices. Once we have the 8 unknowns that govern the transformation back to a flat plane, we apply it to every set of coordinates in the region surrounding the projected code marker. Each of these pixels is mapped back to a 110x110 square as seen below, with each 10x10-pixel box corresponding to one square in the code marker's natural grid. Notice in the unwarped image, there is some distortion in the form of a grid, which is due to the fact that the inverse transform is not one-to-one because of roundoff error. However, this is acceptable distortion because it occurs on very few pixels and has virtually no effect on the bit extraction.

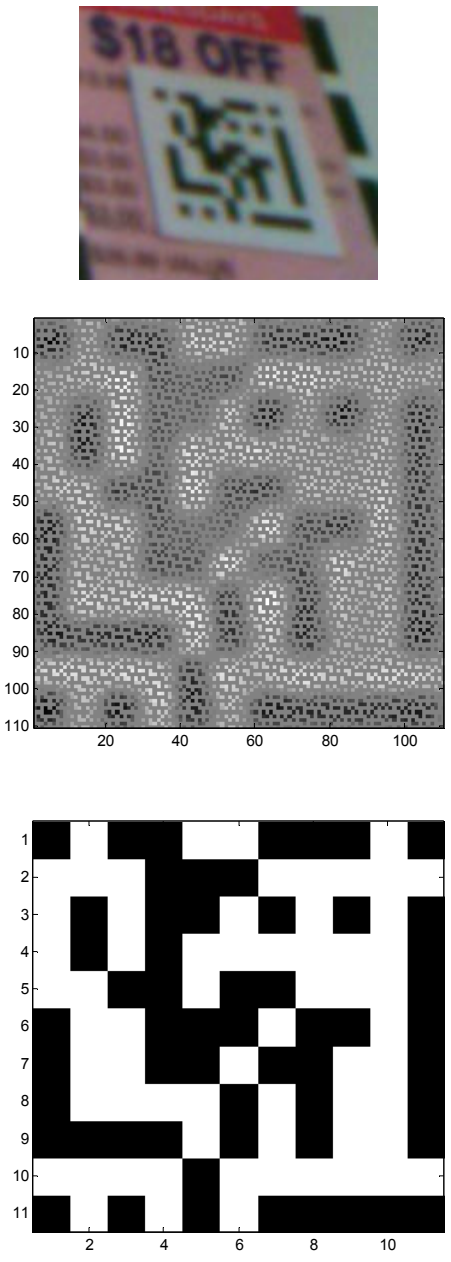

Figure 4: Results of mapping the rightmost code marker (top) *back to a square in its natural grid* (center) *and thresholding* (bottom).

F. Reading Data Bits

Now that we have performed a transformation back to square dimensions and a level plane, bit extraction is fairly straightforward. Once we have mapped the pixels back to a 110x110 square, we take the average value of each 10x10 box and compare it to the threshold given by Otsu's method to determine whether to detect a 0 (white) or a 1 (black) for the corresponding data bit. We do not check the parity bits, because we have already used the guide bars and the corner squares, and would have ended the current iteration of the guide bar search already if problems had been found in identifying these.

II. Performance Results

After running our visual code marker dectection algorithm on the 12 training images, we observe positive results. Our algorithm correctly extracts 1904 out of a possible 1909 total bits (23 total code markers x 83 bits per code marker). This comes out to roughly 0.2 missed bits per code marker. Once we have found all true code markers, the accuracy of our algorithm depends primarily on how accurately we can estimate the coordinates of the four corners. The corners are most easily detected in images without any perspective or skewness, in which case we get near perfect bit extraction. The error of our transformation coefficients increases as the perspective of the code marker increases, increasing the bit error of our algorithm. The following are the number of missed bits for each of the 12 training images:

As far as runtime is concerned, our algorithm appears to be well within the one minute upper limit as defined in the project instructions. Most of the images--the ones containing a moderate amount of mid-sized dark regions--take on the order of 8-20 seconds. As we increase the number of black regions about the size of a typical guide bar, such as text, the computation time increases as well. The following are processing times for the 12 training images:

In addition to testing our algorithm on the original 12 training images, we created 3 more images per original, rotating each one 90, 180, and 270 degrees. This was helpful in the debugging process to ensure that we tested as many guide bar orientations as possible. After some code modification, the algorithm was successful in decoding the 36 rotated images as well.

Training Image Number	Number of Markers	Execution Time (seconds)	Total Bits Missed
1	1	7.4705	0
2	2	17.1778	0
3	3	25.7949	2
4	1	14.1999	0
5	3	19.6949	0
6	1	5.9940	0
7	2	16.8511	0
8	1	10.0729	0
9	3	17.9349	0
10	3	10.9590	0
11	1	47.9530	3
12	2	4.7052	0

Table 1: Performance results for the training image set.

III. Conclusion

Overall, we feel that our algorithm performs the task of visual code marker bit detection both accurately and efficiently. Probably one of the more time consuming and subjective components of the algorithm was finding matching sets of guide bars. To start, we attempted to eliminate false positives using fairly stringent methods, which we felt would be necessary to find all of the code markers in the time allotted. However, soon we realized that using only one or two checks was insufficient to catch all of the correct guide bars or to safely eliminate the false positives. To correct this, we introduced more criteria, each taking advantage of a different property of true guide bars. However, we were also especially careful not to make the thresholds too lenient, which would increase computation time too much. This allowed us to safely pinpoint the location of each code marker while staying under the time constraint. After finishing the rest of the code, we realized that once we found a correct code marker, the time until bit extraction was only about 2 seconds. This allowed us to use the vast majority of the time on region detection.

References

[1] R. Fisher, S. Perkins, A. Walker and E. Wolfart. "Laplacian of Gaussian"[Online]. Available: http://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm

Appendix: Work Distribution

Initial filtering -- Daniel

Small region removal – Daniel and Abheek

Region removal based on orientation angle – Daniel and Abheek

Mapping arbitrary quadrilateral to square -- Daniel

Identification of corner coordinates -- Abheek

Reading bits -- Abheek