EE368 Project Report

Francisco Godoy and Johan Schonning

29 May 2006

Abstract

In this report we explain the approach used to find binary markers in color pictures. Our approach focuses on the elementary properties of the markers. Specifically in the known characteristics of the marker, such as: markers are black and white only, fixed corner elements and fixed guide bars. We use this properties to determine the location of the marker, its orientation and perspective, to finally decode it.

1 Introduction

The main problem consists of finding visual code markers within a picture. The markers are 11x11 2-dimensional arrays in black and white as shown in figure 1. Additionally these markers have 3 fixed corners and 2 fixed guide bars. These are the same elements we use to locate the visual marker. More specifically the guide bars are the first elements we look for in the picture. Our algorithm search for the bars by performing edge detection on the image. After the edge detection, we perform a dilation over the edge-only image in order to create closed regions that are later going to be labeled. Once the labeling is performed we only keep the regions which total pixel count does not exceed a certain threshold value. In this way we retain the regions that are not too big, neither too small to be a bar. Following this, we loop through all the possible regions and use the other properties of the marker to correctly identify it.

Since the marker also has three fixed corners, we find the center of each of the possible regions and also its upper left corner to estimate the size of the pixels; thanks to the fact that the guide bars have a fixed pixel length as well. With the estimate of the pixel length we move in the picture to try to find the fixed corners. We then repeat the procedure to try to find the other fix bar. When the algorithm detects the three fixed corner and the fixed bars, it reads the code data by approximating the size orientation and perspective in the marker given the fixed corners and guide bars. Then each of the found codes goes through a last 'filter' where we compare the retrieved code with a template of the original code having only the fixed elements. This final comparison helps us eliminate all those fake readings. Finally we check that none of the codes have similar coordinates and data, so we do not read the same code twice.

2 Implementation

The first step towards the detection of the markers is to convert the image to a grayscale image, so the Sobel algorithm for edge detection can be applied. This conversion is done by the function $black_white(img)$, which is simply the weighted average of the color components of the image. The weight assigned to each color are the ones described by MatLab: 0.2989 for color red, 0.5870 for color green and 0.1140 for blue. Using the resulting grayscale image the edge detection is done with the MatLab function $edge(imgbw,'sobel')$, which does edge detection using the Sobel matrix.

Secondly we dilate the image containing just the edges using a square matrix of size 3x3 of ones as the window. The dilation was done using the function $imdilate(edge_img, strel('square', 3))$. Afterwards we labeled the connected components of the negative of the dilated image using $bwlabel(\ dilatedimage, 8)$.

These labeled components were then thresholded by their number of pixels they had. Every labeled region had to contain more than 20 pixels and less than 600; these constants were named min_region and max_region respectively. The number of pixels of each region are calculated using the function $regionprops(labeledimage,'Area')$. Furthermore, we only select regions that have a shape close to rectangular, to do this we calculate the ratio between the minor and major axis of the region, using the function $regionsprops(labeledimage,'MajorAxisLength')$ and $regionsprops(labeledimage,'MinorAxisLength')$. This helps us reject unnecessary regions to improve the execution time of our program.

Thirdly we take each of the regions that made it pass the threshold and try to find all the elements of the marker. To do this we look at the length and width of the region using the function $regionprops(importantregions,'BoundingBox')$. Then we take a portion from the original image centered at the centroid of the given region. The centroid is obtained with $regionprops(importantregions,'Centroid')$. The portion of the image taken is of size $(2 \cdot m_size + 1)x(2 \cdot m_size + 1)$ with $m_size = 120$. This portion of the original image is then rotated to have the guide bars aligned with the coordinate axis. To rotate the image we used the function *imrotate*. The angle of rotation is determined by $regionprops(importantregion,'Orientation')$. Since the guide bars form a 90 degrees angle we cannot determine right away which one is which. Therefore when we do the rotation of the image we have to take this into consideration. For this purpose we create additional images by rotating 90 and -90 degrees the portion of the image we oriented. Hence creating 3 images for each possible region.

Fourthly, since we know that the fixed elements of the marker are black. We take only the black pixels in the test image created from a portion of the original after the rotation and region labeling described in the above paragraphs. In order to identify the darkest pixels we have a relative measure to each image. This measure consists of taking the brightest pixel of the the region which gave rise to the image and add to it a given constant.

The next step calls the function $getCorners(labeledimage, long)$, where we specify through the variable long whether we are looking for the vertical or the horizontal bar in the marker. Since the vertical and the horizontal bars have different pixel lengths, therefore assuming one or the other changes the search algorithm slightly. To get the four corners of the marker we approximate the pixel length in a given direction using the region we are looking at. Then we take the center of that region using *regionprops* and move in the given direction (x or y) to find the fixed corners. Surely not every region we look at gets us the right marker, it is possible to take a region that is within the marker, or also take a line that does not have anything to do with the marker. Nonetheless these regions will be eliminated when we try to read the code from them. Additionally, since the markers have a rectangular shape, we eliminate all those fake markers that could be obtained by just taking the ones that have a shape similar to rectangular.

With the four corners of the marker detected, we proceed to read the code from the possible marker. Taking into consideration that the marker can have perspective or have a certain inclination, or both, the found corners help us determine this and search the correct code values through the image. The function that reads the code values is $readMarker$. The inputs to this function are the position of the corners, the value of the brightest pixel and the rotated portion of image where the marker is. All this will return an 11x11 marker which will later be compared to a template of the marker having $0's$ in the variable code positions. Therefore, since the fixed corners are surrounded by white pixels, and the bars too, there are 38 positions that the possible marker should match. This give us, according to Kolmogorov's Complexity, a probability of about $\approx 2^{-38}$ of having random pixels that would match a marker. Hence we only take the markers that completely match the template.

Finally we make use of the corners found in the rotated image, convert them into coordinates in the original plane and return the detected data from the code and the position of the marker if this position is not within a certain position of another marker that we know is valid and was also detected.

2.1 Example

To illustrate the implementation of the algorithm we are going to show step by step what would be seen when implementing the code on an image. In this case we are making use of image 5 from the training images, this image is shown in figure 2 (a). In the same figure we can see what happens after the edge detection in figure 2 (b) and also after dilatation in figure 2 (c). Moreover we also show the output of the region labeling figure 2 (d) and when we take only the regions we are interested in figure 2 (e). After this we call the function $getCorners$. Then if any of the corners cannot be retrieved because the region we are looking at is not a marker, one or more of the corners will have the value zero. Therefore the algorithm only uses the images that are possible markers, and that is how we get rid of all the vertical lines shown in figure 2 (e). Then we only call $readMarker$ given the portions of the original image where the markers are. This is shown in figures 3, 4, and 5. Finally the algorithm reads the markers and identifies their locations, as it can be seen in figure 6. The locations displayed on figure 5 correspond to x,y coordinates and not row, column coordinates.

3 Results

After testing our algorithm with the 12 training images using the function *evaluate.m* we got the results shown in table 1. As you can see our algorithm performs very well, we got all the bits as they should have been and the execution times are all below 15 seconds except for training image 8 which has a running time of almost 21 seconds. The average execution time was 7 seconds. This results were obtained using with the computers from the SCIEN lab.

4 Limitations

Our algorithm works with two simple assumptions: that the markers have every of its bits in the picture bigger than 2 pixels in length and width, and that the image is not only one big marker. This assumption was taken from looking at the general form of the training images. Moreover for the algorithm to work properly there must be enough illumination in the image as to distinguish correctly between black and white.

5 Conclusions

We can see that the algorithm uses very simple ideas to detect the code markers in the images. It mainly focuses on the fixed corners, fixed guide bars of the marker and the fact that the markers are black and white. The performance of the algorithm matched the constraints of the project, which specify that it most take less than 1 minute to decode each image. Moreover the decoding of the markers was very well executed.

The only error of the algorithm was one bit in training image 3. However this error could have been corrected if we allowed the function *detect_code* to have the variable *brgt_pix* as an input. In this way we could change its value manually to correctly read every image given the illumination conditions.

6 Log

Both of us work together most of the time so that the division of work would be even. Therefore here is the final outcome of our work in terms of how we worked on each function.

- *detect_code*. Francisco Godoy, first revision: Included image edge detection and black and white conversion as well as image rotation, first attempt to locate the markers, and added correlation with original marker. Johan Schonning, second revision, added region labeling and a way to estimate the markers that was the final and also added ratio of axis of the regions to improve the speed of the algorithm.
- *getCorners* Johan Schonning.
- *readMarker* Johan Schonning, first revision, detected long vertical bar. Francisco Godoy, second revision, also included horizontal bar search.
- *get_mask* NOT INCLUDED in running version of the algorithm. Johan Schonning, first revision, initial approach to create a mask to correlate the ones we read from the image. Francisco Godoy, second revision. Upgraded to make a variable mask according to the perspective and the angles of the fixed guide bars on the image.
- *detect_corners* NOT INCLUDED in running version. Francisco Godoy, first revision. This function was created with the purpose of finding all the corners in an image, two versions where made however due to its many returned data points we decided not to use this approach.
- Report. Francisco Godoy, first revision. Johan Schonning additional comments.

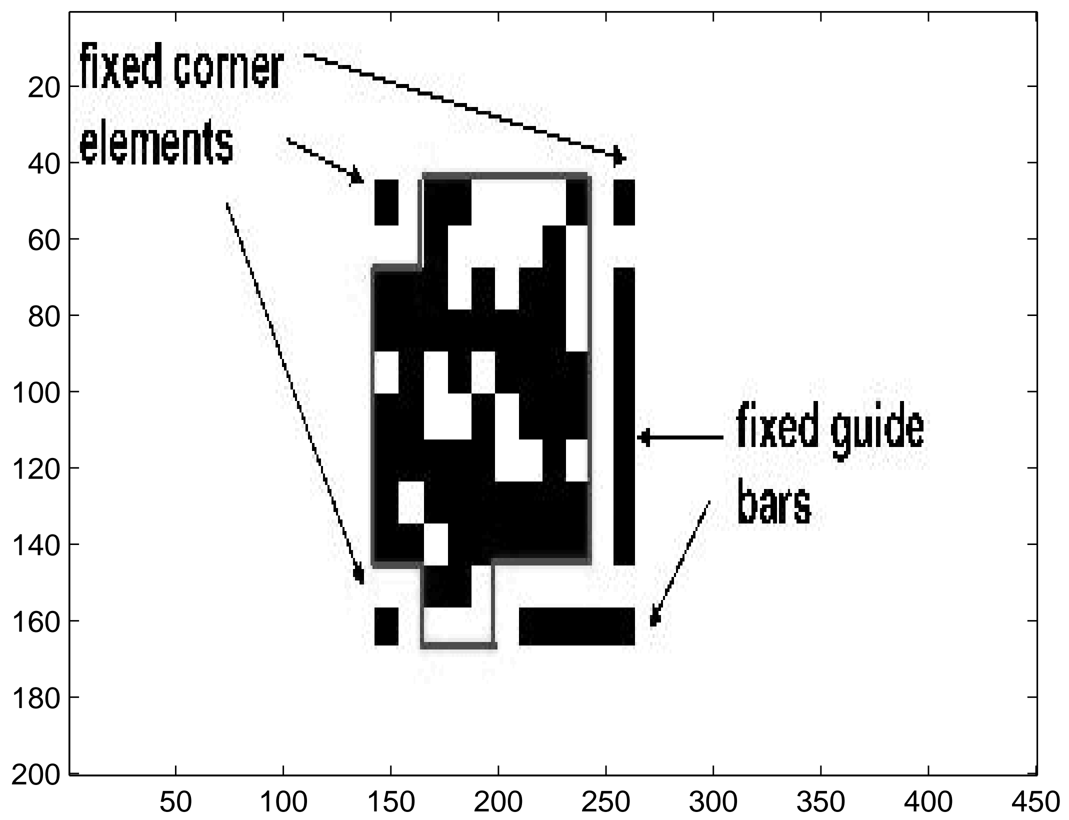

Figure 1: Visual Code Marker Details

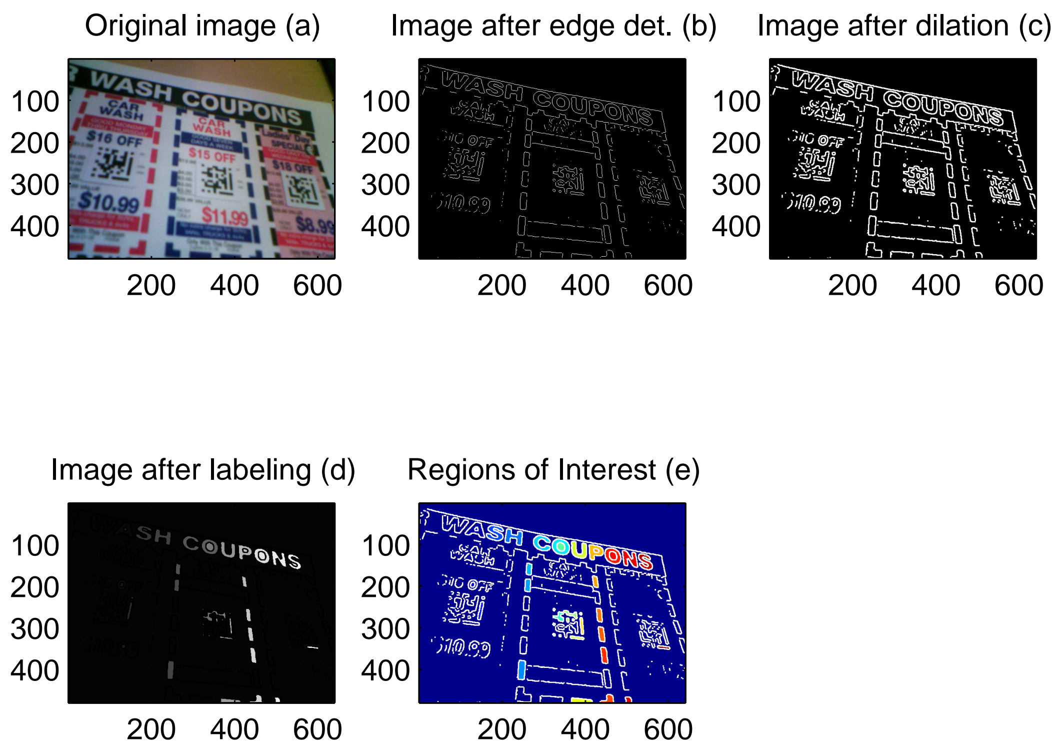

Figure 2: Training image 5

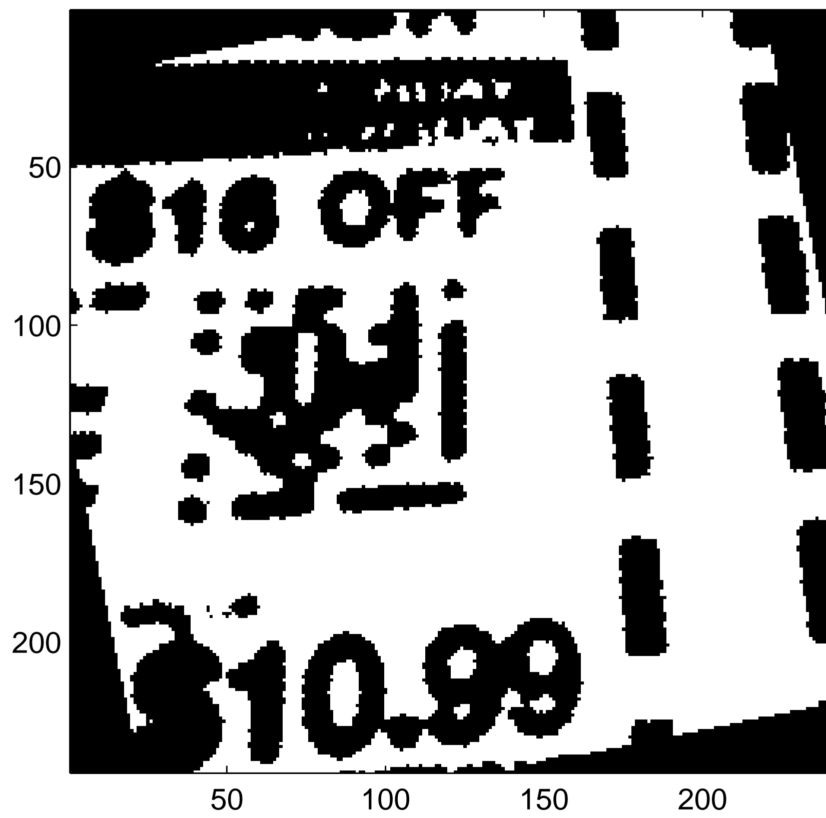

Figure 3: Code Marker Identified

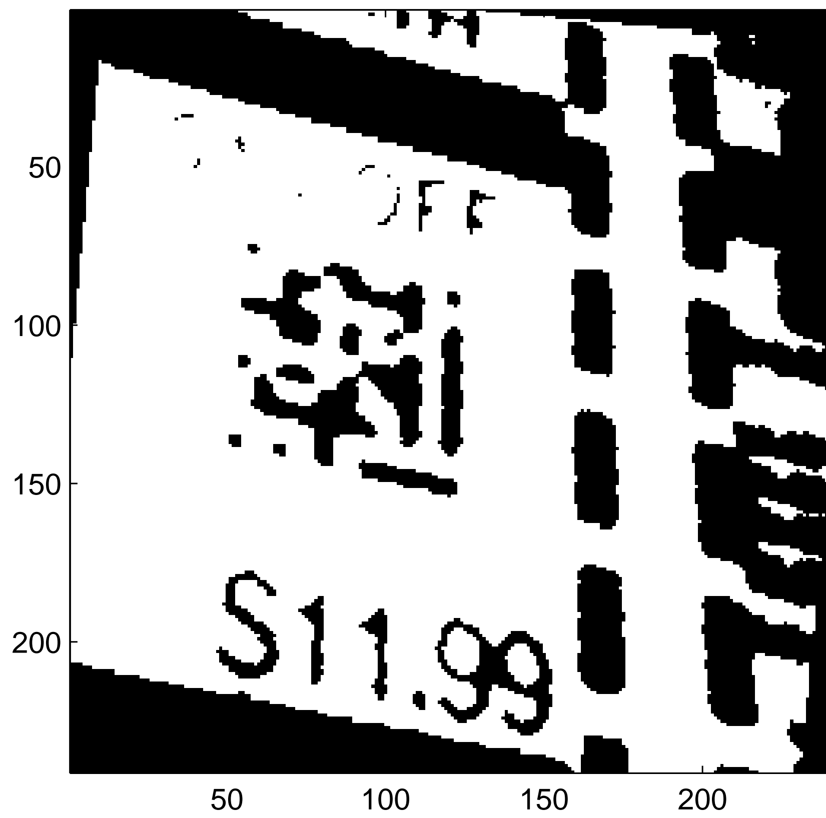

Figure 4: Code Marker Identified

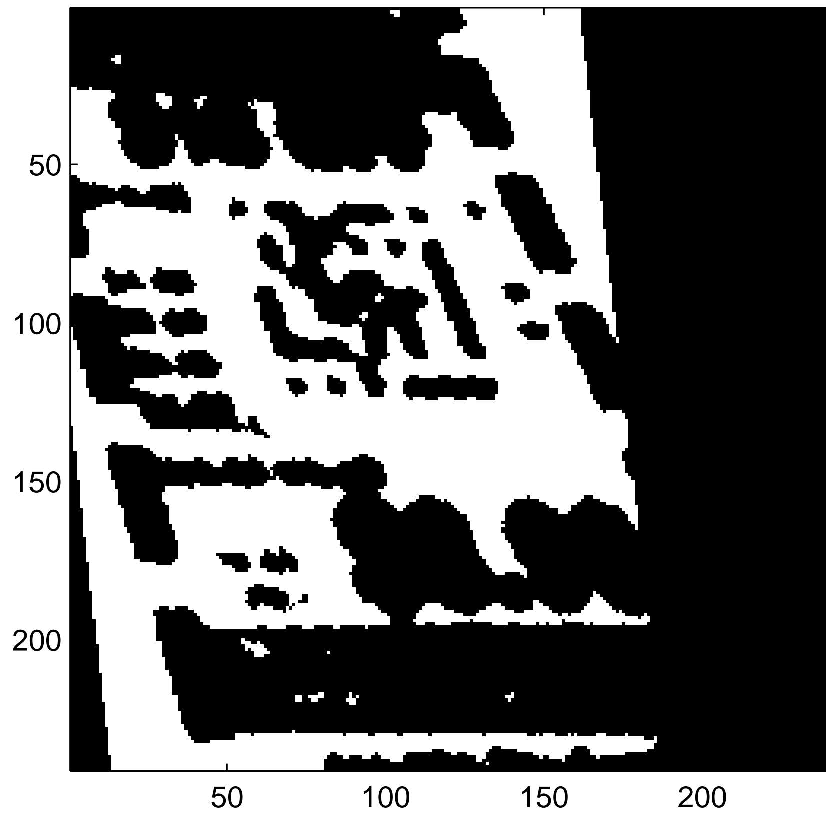

Figure 5: Code Marker Identified

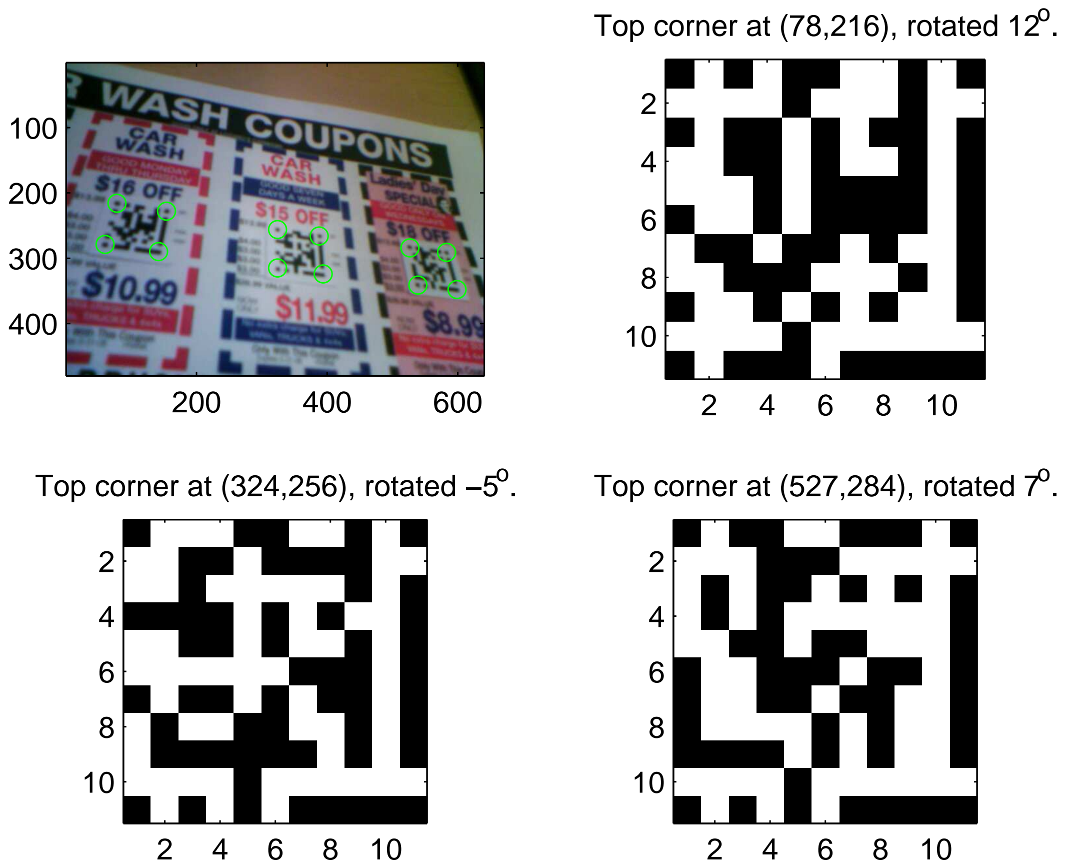

Figure 6: Final Output

Image	# correct bits	Execution Time	# false alarms	# repeats
Training 1	83	5.28	0	0
Training 2	166	3.73	0	0
Training 3	248	4.65	0	0
Training 4	83	16.64	0	0
Training 5	249	10.63	0	0
Training 6	83	2.29	0	0
Training 7	166	3.25	0	0
Training 8	83	20.2	0	0
Training 9	249	7.15	0	0
Training 10	249	10.7	0	0
Training 11	83	3.81	0	0
Training 12	166	3.13	0	0

Table 1: Training Images Results