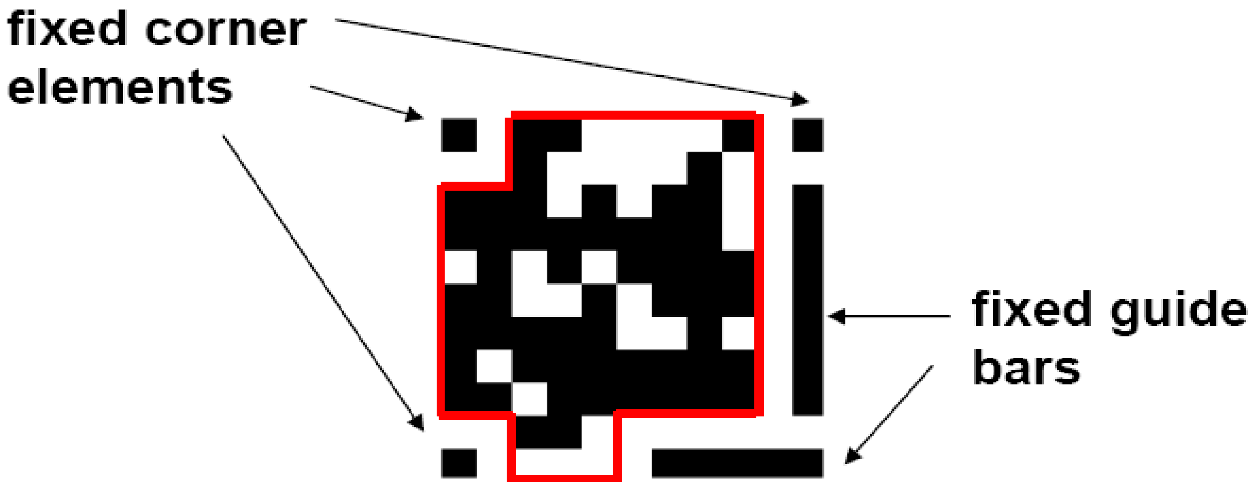
EE 368 - Project

Submitted by Punyanjan Sen

1.0 Introduction

The project requires us to devise an algorithm to detect visual code markers placed on color images, and report the value of all data bits on the markers.

- The visual code marker is a 11x11 array.
- There are three fixed corner elements and two fixed guide bars
- There are 83 data bits.



A Visual Code Marker

- Each image can contain one to three code markers.
- The code markers can be placed at arbitrary orientations on the image
- The camera angle, distance and illumination can all vary.

This report describes an algorithm used to achieve the project's objectives. During the training and optimization phase, some alternative algorithms were attempted. The report discusses these and why they were discarded.

Finally, further possible optimizations to the algorithm are discussed.

2.0 Converting from Color to Binary

Since the code marker is just a set of bits, it made sense to convert the image to a binary image as soon as possible. Processing color components, or even grayscale intensities would be needless and computationally expensive.

However, choosing the correct process for this proved crucial to the success of the algorithm.

Two algorithms were evaluated to convert to black and white.

2.0.1 Gray Scale to Binary

1. Use the ITU standard thresholding algorithm to convert to a gray scale image:

```
I =  0.2126 * red + 0.7152 * green + 0.0722 * blue
```

2. Convert to the resulting image to binary, using the function im2bw.

This algorithm did not work well because of the different levels of luminence in different regions in the image. Code markers at lower luminence levels were not preserved very well.

2.0.2 Using an averaging filter

1. Transform the image to gray intensity levels using the *mat2gray* function.
2. Run an averaging filter over the subsequent image.
3. Subtract the result from the original image
4. Convert the resulting image to binary, using the function im2bw

This approach works much better since the averaging filter adapts to different intensity levels in the image.

2.1 Region labeling and counting

Because the code markers guides are segmented into clear regions in the binary image, it was extremely useful to label the individual regions.

- It's easy to calculate the size of a region. We can use this to evaluate that the vertical guide bar is larger than the horizontal guide bar. Also, the corner pixels should all be roughly the same size.
- Once a region is labelled, its easy to find it's coordinates, even if the coordinate system changes (eg. if the image is cropped). If the region is a corner pixel, its easy to find it's center coordinates by just taking the average of the minimum and maximum values in each axes.
- It's also easy to find the rectangle that bounds a region, through its minimum and maximum coordinates

- If an algorithm finds slightly different coordinates by different means, it's easy to find out if the two coordinates belong to the same region.

2.2 Identifying the vertical and horizontal guide bars

This was the most computationally expensive part of the algorithm. For efficiency purposes, very small (less than 120 pixels) and very large regions (greater than 600 pixels) were removed from the image before identifying the guide bars.

After the guide bars were identified, we resumed processing the original binary image (since the corner pixels were lost when small regions were removed).

The following approach was tried first:

2.2.1 Using the Hough Transform

1. Find all edges by applying an edge detection algorithm
2. Run the Hough transform to find locations and orientations of all edges
3. Find pairs of orthogonal edges (or roughly orthogonal for code markers in a different plane) that intersect at a point
4. Ensure the vertical guide bar is larger than the horizontal
5. Find the corresponding fixed corner elements

This algorithm was discarded because

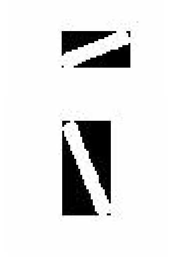
- there were hundreds of lines detected on some images.
- also most lines occurred on similar orientations. This made it difficult to look for peaks in the Hough transform.
- to cater for the images on different projection planes, several alternative angles had to be evaluated

The following algorithm was tried next.

2.2.2 Using Morphological Operations on individual regions

1. Loop through all regions in the image
2. Create a small image for each region, by cropping the original image. The cropping coordinates are found by the rectangle that has diagonals at the minimum and maximum coordinates of the region. Processing the cropped region was more efficient that operating on the entire image.

Some examples of cropped regions are shown below:



3. Remove any pixels from neighbouring regions in the cropped image. Because the image is labelled its easy to find pixels from other regions.
4. Run a “thin” morphological operation on the region, using the *bwmorph* function with the “Inf” option. This reduces the region to a minimally connected line (if there is one). Additionally this line will lie in the middle of the region (rather than its edges). This makes it easier to find regions that intersect that line.

The images below show the cropped regions after the thinning operation

5. If a minimally connected line is found, calculate its gradient and equation.
6. Assume that this line is the vertical guide bar. Traverse the line in either direction, looking for neighbouring regions. We know that the vertical guide bar intercepts the horizontal guide bar after a short distance (approximately 1/7 the length of the vertical guide bar). Since there shouldn’t be any intervening regions, some flexibility was allowed in the distance.
7. If a neighbouring region is found, ensure that

- it’s size is smaller
- it’s gradient is different (by at least 0.4).

This algorithm proved to be more efficient and reliable than the previous one.

The corresponding regions in the image are shown below:

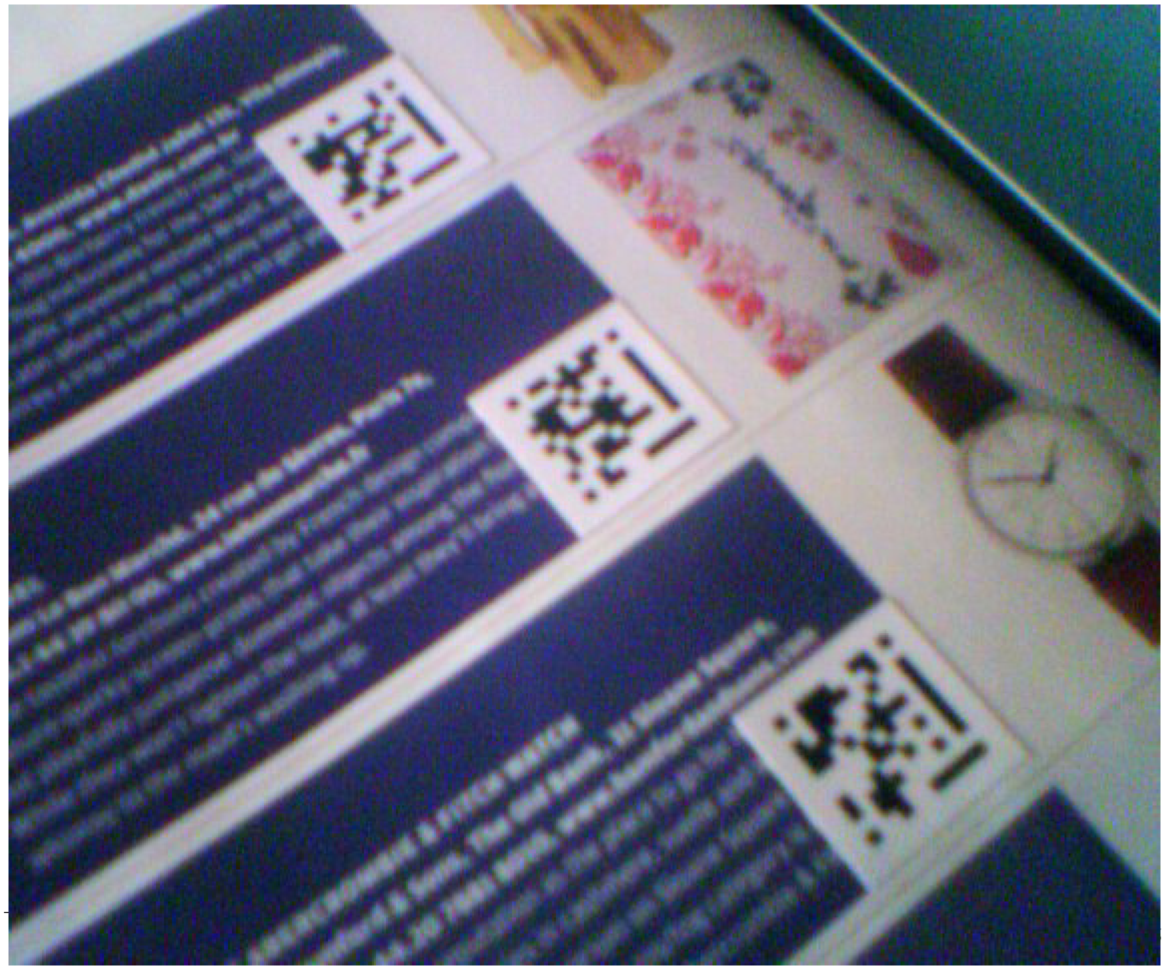
2.3 Identifying the fixed corner elements

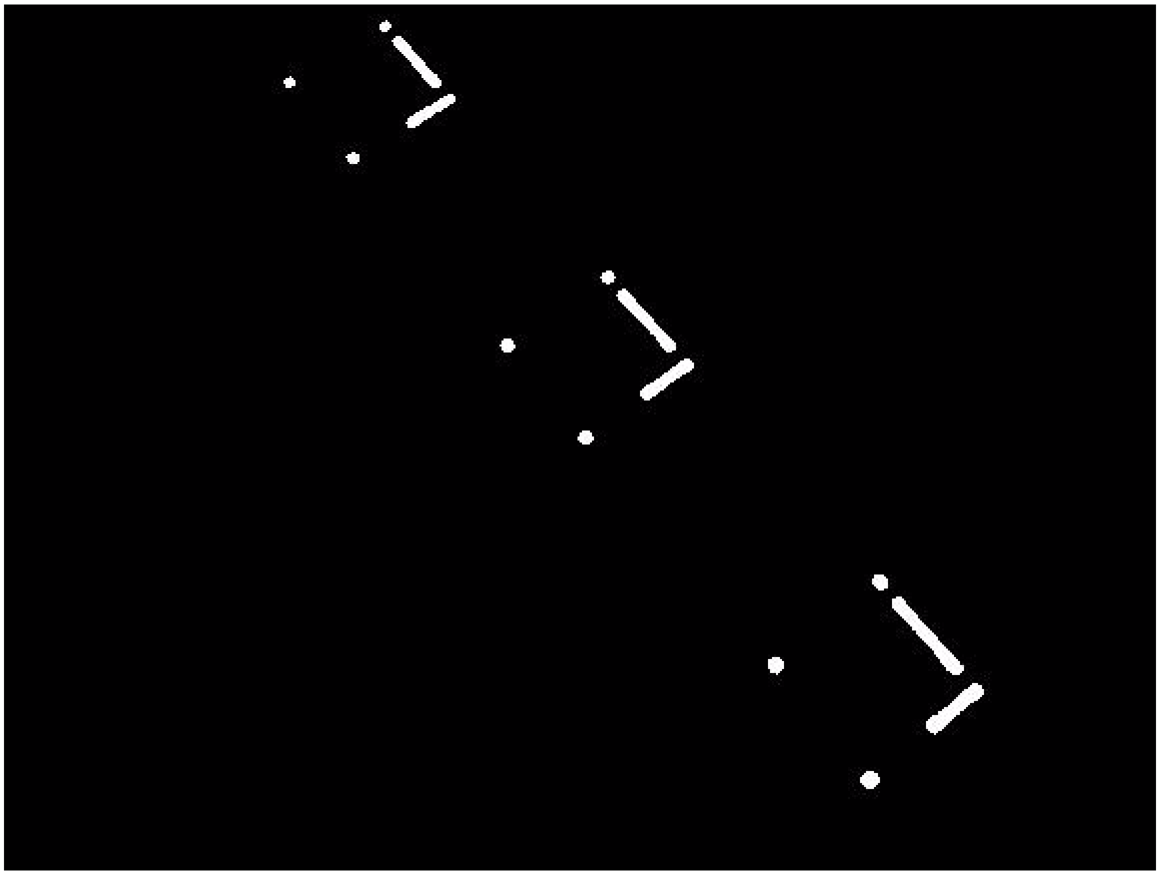
The original binary image (without small and large regions removed) was now processed once more.

1. Traverse the vertical guide bar in the opposite direction, looking for the top right corner element. Once again, we allowed for some flexibility in the distance - since there should be no intervening regions.
2. If the top right coordinate is found, traverse the horizontal guide bar, looking for the bottom right corner region. This was the most critical part of the detection process. Because there could be other regions in the middle (actual data bits), we needed to be strict on the distance calculation. The bottom left coordinate should be located at a distance of 1/5 * 11 * (the length of the horizontal guide bar), from the bottom right coordinate.
3. If the bottom right coordinate is found, recalculate the gradient of the line calculating the bottom left corner region and the bottom right corner region (to get a more accurate gradient). Using this new gradient and the location of the top right corner region, find the equation of the line connecting the top right region to the origin region. Trace this line to find the origin region. The distance should be exactly the same as the line connecting the bottom left and right corner regions.

If all four coordinates correspond to valid regions in the image, ensure the sizes of the fixed corner regions do not exceed 150 pixels, and are roughly the same.

At this point the code marker has been identified. The image below is one the training images. The image on the next page shows the identified fixed regions in the image.





2.4 Projecting the code markers into the correct plane

Many of the images were taken at an angle, resulting in code markers in different planes.

In order to accurately read the positions of the bits on the image, it was necessary to do a projective tranform from the code marker plane into the image plane. This transformation was done on each code marker separately.

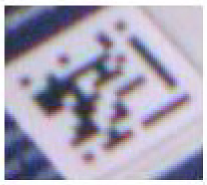
Such a transform is possible, if four coordinates in the reference image are known. We chose the four corner coordinates, since we had already calculated their coordinates.

The following base point coordinates were used:

```
    basepoints = [1 1; 121 1; 1 121; 121 121];
```

Most base points that specify a quare region proved to be sufficient.

The `cp2tform` function was used to perform the transformation into the image plane. The transformation of the image into the image plane and rotation to the correct orientation worked very well. It also worked well for code markers that were in the same image plane but at different orientations. These were just rotated into the correct orientation. It was, of course very important that the four coordinates matched as accurately as possible to the four corners of the code marker.



Original Region in a different orientation and plane

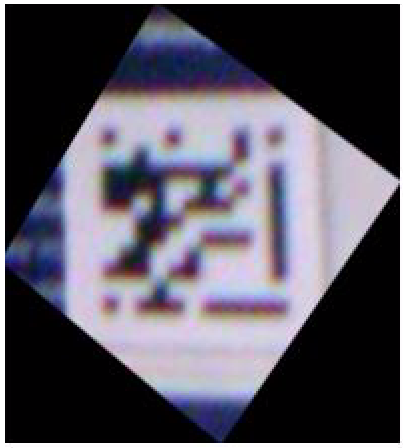


Image transformed to correct plane

2.5 Counting the number of bits in the code markers

To calculate the number of bits in the code markers, the data bits needed to be as accurate as possible. To ensure this, the original image was re-cropped to get just the code marker. Then it was transformed to binary, using the cropped image's gray threshold value - so that intensity levels at other regions of the image did not affect the thresholding value. This ensured that minimal data corruption occured.

The code marker was then split into 121 equally spaced regions, and the algorithm looped from top to bottom, left to right, to find the data bits. The bits that corresponded to the fixed elements were ignored.

It was very important that the image was in the exact plane and orientation. Otherwise, there would be errors in reading the bits. It was easy to calculate the origin, top right and botoom left coordinates, because they were in their own regions, and it was easy to find the center point.

However, I believe there was some error in finding the exact coordinate of the bottom right region, because it was part of a horizontal bar at an arbitrary orientation. Some data errors may be because of a few pixels difference in calculating the exact coordinate of this region.

3.0 Conclusion

The final algorithm proved to be fairly accurate on the training images. The following thresholds may need to change for images with code markers that are much larger or smaller than the training images.

- large and small region removal (for computational efficiency to find horizontal and vertical guide bars)
- determining the gradient between the vertical and horizontal guide bars should be greater than 0.2.
- determining that the size of fixed corner elements should not be greater than 150 pixels

The algorithm does not cater for code markers rotated at greater than plus or minus 90 degrees. It is assumed that the code markers will not be "upside down" in the image. Given some more time, the algorithm can be adapted to do this.

Otherwise, the algorithm should be quite adaptive to images taken at varying intensities, varying distance from the camera and at different camera angles.

4.0 References

1. EE368 lecture notes
2. "Digital Image Processing using Matlab", Gonzales,Woods and Eddins, c2004
3. "Digital Image Processing", Second Edition, Gonzales and Woods, c2001
4. "Real-World Interaction with Camera-Phones", Rohs, Michael, 2004 Swiss Federal Institute of Technology (ETH) Zurich, Switzerland. Online at http://www.vs.inf.ethz.ch/publ/papers/rohs2004-visualcodes.pdf