Code Marker Detector
EE 368 Digital Signature Processing

David Pond

Stanford University, Morrison, CO

***Abstract* — This report describes a Matlab program that detects two-dimensional binary code markers embedded inside images.**

***Index Terms* — Code markers, threshold, binary morphology, feature extraction, connected components.**

I. Introduction

The goal of this project[1] was to develop code in Matlab that could extract code markers embedded in images. The code markers of interest are 11 x 11 two-dimensional arrays of binary elements. The code markers contain two guide bars, one horizontal and one vertical, 83 elements that represent data. The code markers can be embedded in images of varying complexity and quality. Additionally, the code markers may be rotated or in perspective. The program needs to extract valid code markers and reject data that is similar to the code markers. This report will describe my method to implement code marker detection and data extraction in Matlab.

I just finished getting my program working about an hour before the deadline, so this report is going to be a tad less detailed than I would like.

II. Algorithm

My basic tact was to divide the problem up into stages. The first stage involves trying to fine "candidate guide bars." By this, I mean looking for lines that might be good candidates to be examined to see if they are guide bars.

The first problem I had to solve was how to detect line segments in the image. I chose to implement a connected component algorithm, but this first required converting images to binary. Attaining a suitable thresholding value across images with varied lighting and contrast proved difficult. So I chose to implement a block-wise threshold algorithm that operates by walking over the image in small blocks. For each block the algorithm computes a local threshold to be applied. If there is insufficient variation in intensity within a given block, then if the average intensity of the block is close to black or white, the entire block is either marked to black or to white. With this algorithm, I was able to generate suitably thresholded images.

Next I needed to isolate the "line segments." To do thisi, I applied a connected components algorithm to break the image up into pieces. The algorithm then goes through each connected-component region and disposes of those regions that are not "line-like." The algorithm looks at several criteria such as the eccentricity of the region, whether the region has any holes, and how solid the region is. After removing the non-linelike regions, I apply a skeletonization and de-spurring operation. These are applied in sequence and multiple times, respectively. When done, I have for most image nice thin lines that align with the guide bars. In most of the training images (about 10 of them), I get lines that just match the guide bars. In a few additional images, there are a couple of extraneous lines. The adaptive thresholding, connected componnet analysis, and skeletanization/despurring are very sufficient to get an idea where there guide bars are in the image.

The problem with the above algorithm is that the resulting lines then to wiggle and aren't as long as the underlying guide bars. To proceed, I analyze the line segments looking primarily at each line segment region's length, centroid and orientation. With this information, I am able to a rough intersection between imaginary lines that run through the centroids at the given orientations. The resulting intersection corresponds to the "lower right-hand" corner of a code marker.

At this point, I have a set of lower-right coordinates that might correspond to code markers, and I have an idea of the rough orientation of the horizontal and vertical guide bars, this this data corresponds to a marker.

At this point, the process is to essentially start over, but this time using the information from above as "hints." The code walks over each "candidate" guide bar and uses the lengths and orientations of the line segments to figure out an approximate region for the marker. A new threshold is computed and applied to that specific region. This thresholding, because it is primarily applied directly to a code marker region, yields very good results. At this point, use the line segment centroids to perform another connected components to what should be the guide bars.

This time, because of the thresholding that was well suited to the marker data, we get very good data on the guide bars. This includes accurate lengths, width, and orientation. Using the knowledge of the guide bar extents, we use this to compute a rough element size. It is considered rough because the image might be in perspective in which case the elements actually may change in size across the marker.

At this point, the algorithm has done some work to reject line segments that did not yield guide bars underneath. This happens during the connected components analysis if the new regions detected do not have guide-bar like properties (e.g., eccentricity, solidity, etc.).

The next step is to locate the corner elements. Using the better guide bar measurements, the code first computes the intersection of a line running through both the vertical and horizontal guide bars. This is the second time to compute this intersection, but this time the intersection value calculated is more accurate because we have well-defined guide bars.

The code then uses the element sizes (computed individually in the horizontal and vertical directions to account for perspective) to make a rough guess at where the other three corners are located. The algorithm then extracts a region surrounding each corner in turn. Another connected-components algorithm is performed in each corner's approximate vicinity. If a component that is element-like is detected, and if it is sufficiently close to the anticipated corner location, it is deemed to be a corner element.

At this point, the algorithm has computed with a high degree of accuracy the guide bars and corner elements for all markers. It is actually quite good at this point.

But this is where things start to break down a bit. From this point, the algorithm attempts to extract the data elements. The problem I ran into was how to account for perspective and curvature that distorts the marker non-uniformly. I came up with an algorithm that takes the markers that we've found with good accuracy and it attempts to build a grid that is distorted to match any distortion in the perimeter of the code marker. Unfortunately, while the grids I generate line up well in excess of 60-75% of the elements, I have been unable to tune the algorithm such that all element locations are accurately determined. However, I do not believe this is because of a flaw in my strategy, so much as it is a flaw in my grid-generating algorithm. For instance, it makes a rather poor assumption that only shearing is occurring, as I did not have time to implement proper perspective warping on the grid elements.

Lastly, one the grid of where the elements are expected to be has been generated, each element region in the image is probed to detect its value. Again, I ran out of time here to do a respectable implementation. Currently, the algorithm simply examines a small element a few pixels wide at the center of each element region. If the average of the pixels is greater than .5 (this is in the thresholded binary image), the element is counted as a "1," otherwise it is treated as a "0." There are lots of flaws with this strategy, the biggest of which is that it requires that the grid alignment be almost perfect, which mine is not.

III. Performance and Results of Execution

The performance of my algorithm seems to be much under a minute. I've developed it on a very old 500 MHz PowerBook and the slowest file takes about 36 seconds to process. If given more time, I believe the algorithms could be made to be much faster.

As far as running the evaluate.m program, well, that was a bit disappointing. My code is clearly detecting most markers in the training set, but is getting almost no data 100% accurate. Again, I don't believe this is a flaw in strategy so much as lack of time to complete the implementation.

Running the code in a mode that displays overlays on the marker elements, corners, guide bars, etc. illustrates that the program is very close to being able to accurate read the data values. In fact, the guide bar and corner detection has proven to be 100% accurate on the test images. Data extraction is where the algorithm breaks down.

VII. Conclusion

A program has been developed to extract code markers from image data. The algorithm that is implemented by the program was described. Unfortunately, due to time constraints, the program does not extract data very well despite doing a good job of locating the code markers.

Acknowledgement

Myself for the incredibly long hours I had to put in working on this project by myself. Oh, and the TAs and Instructor for a running a very interesting and informative course.

References

[1] http://www.stanford.edu/class/ee368/project.html