QR codes were designed to assist in the manufacturing plants of the automotive industry. These easy to scan codes allowed for a rapid way to identify parts and made the entire process more efficient. The concept, a tweak of the traditional bar code, has started to pop up in many places outside of manufacturing. They’re a common part of advertising campaigns, allowing potential customers to scan a code and be directed to a website for a promotion or advertisement of some sort. Because of the explosion in popularity of the QR code, we decided to explore this space and build a QR code reader from the ground up.

One of our first decisions made in this process was, “How are we going to create this?” It was a decision between our familiarity with Java, and the robust image and math tools available in MATLAB. In the end, we decided on MATLAB to take advantage of all the built in features for image processing. Additionally, we made the important decision to pare down the scope of the QR code reader – instead of designing it to read QR codes found in the wild, of the standard design, we built a custom “language” that our reader would recognize.

With those two decisions made, we made a general plan on how to design the program. The idea behind the QR code is simple – recognize the distinct corner patterns, use those as reference points, and then algorithmically “read” the black and white dots and have a parser to have them be something recognizable that can be converted into normal, readable ASCII text.

Unfortunately, we did not approach this problem from the correct angle. Our first step was to create images on the computer and attempt to directly read them. The issues of noise, warping, lighting, we treated them as things to help “boost the accuracy” of our project, and not as essential preprocessing elements that were critical to allowing our program to work at all. With that perspective in mind, we decided to tackle the first roadblock – warping the image to a perfect square.

Taking a picture nowadays is most commonly done with a camera. It’s rare that a person can take a picture with the QR code directly perpendicular to the plane of the camera, so it’s necessary to take that skewed image and change it to one that our reader can make use of. To do this, we used the idea of homography, or projective transformation, to take an image and morph it into a square.
The essential idea behind the projective transformation is taken by comparing points on two images through a theoretical “lens.” In order to do this, we needed data points corresponding to the corners of the QR code to transform onto a theoretical 200 x 200 blank image. The original plan was to use SIFT to find the corners of the original QR code. This was mostly successful, but ultimately proved useless, as we were unsuccessful in finding a way to take the feature points from the SIFT and turn those into usable corner points. Our solution to this issue was less than elegant – the user manually inputs the corner points. While this is not ideal, it does not seem like a completely unreasonable way to handle this process with a program in its early iterations.

After the corner points are found, we use helpful commands in MATLAB to build a transform and then apply it to the original image, which, when applied to figure 1, results in the image seen in figure 2.
The image is then cropped by manually selecting the upper left corner. This results in the QR code being a 200 x 200 image that is then passed over to a function that helps make it more readable.

The function in our reader, `toBlackWhite`, takes an image and produces a black and white version. A number of tools were used to do this. First, the RGB image is simply converted to the grayscale version. In grayscale, every pixel is defined by its luminosity, a number from 0 to 1. Closer to 0 is darker, and closer to 1 is lighter. The next thing that is done is an adjustment for contrast via the `imadjust` function, which makes the image higher contrast (raising the whites, and lowering the blacks). Then, the `weiner2` function is used, which helps to remove noise from the image. A filter is applied, the “unsharp” filter, that actually sharpens the image, to improve the border between black and white. Finally, the function runs through the remaining pixel and raises it to 1 (white) if it is above .65 and 0 (black) if it is below. The threshold was obtained
through trial and error. The results are fairly good, as you can see from the transition from figure 3 to figure 4.

Figure 3: The unprocessed QR code from figure 1

Figure 4: The QR code from figure 3, after being run through the toBlackWhite function
The results are nice, and as we found out later in the project, this processing was essential to turning getting accurate results during the decoding. The process is not perfect, as there is some fuzzing in certain areas, and it does not handle the part of the code that was brightened by lights (which could be commonplace with flash on phones).

From that point, the code was ready to be analyzed. In our QR code, we eliminated code between the corner markers, leaving a 6 x 6 of characters available to be worked with, as we decided that the 200 x 200 would be cut 8 x 8, and each corner marker took up 1 of those grid spots. The algorithm is fairly simple – it would go into each of the 6 x 6 squares, and break that square up into a 5 x 5 square. It would average each one of the squares in the inner grid, and put a 1 into a matrix if the value was above a threshold, or 0 if the value was below. That matrix would then be compared to predetermined matrices to determine the value of the character, and that character would be placed in a matrix to be deciphered later.

The encoding scheme was fairly simple. Each character was a 5x5 matrix, so we essentially used binary representations of the characters. A = 1, B = 2… Z = 26. The binary was on the top row, with white = 0 and black = 1. On the third row, we put numerals, 1 – 9, 0, also using binary (but with 0 encoded as 10). Blank spaces are treated as such.

The overall results from this project were a mixed bag. With the image in Figure 5, we got quality results about 50% of the time, with the errors usually being garbled text. In other cases, with darker, more blurry images, the reader consistently would fail to pick up any of the text. One of the biggest boosts to accuracy is picking the corner points well – cutting off too much often ended in a failed run. Our encoding proved successful, but it might be better to shrink the 5x5 matrix to something smaller, because it would allow for more characters to be stored, and the 5x5 matrix we use is overkill for the 36 alphanumeric characters we are using.
There are a number of ways to improve the quality of the reader and to take the project to the next logical step. Some of the obvious ways are to automatically get the corner points using SIFT. This eliminates user error in corner selection, which proved to be one of the main factors in the success of the program. Another couple of ways to improve accuracy is to make the toBlackWhite function better, as well as creating something of a predictive algorithm to fill in any holes. Another logical step is to make the algorithm more robust with placement, as well as using the standard QR code encoding, to read standard QR codes instead of just the custom ones that we employed. With this as a starting point, a full-featured accurate QR code reader could be created.