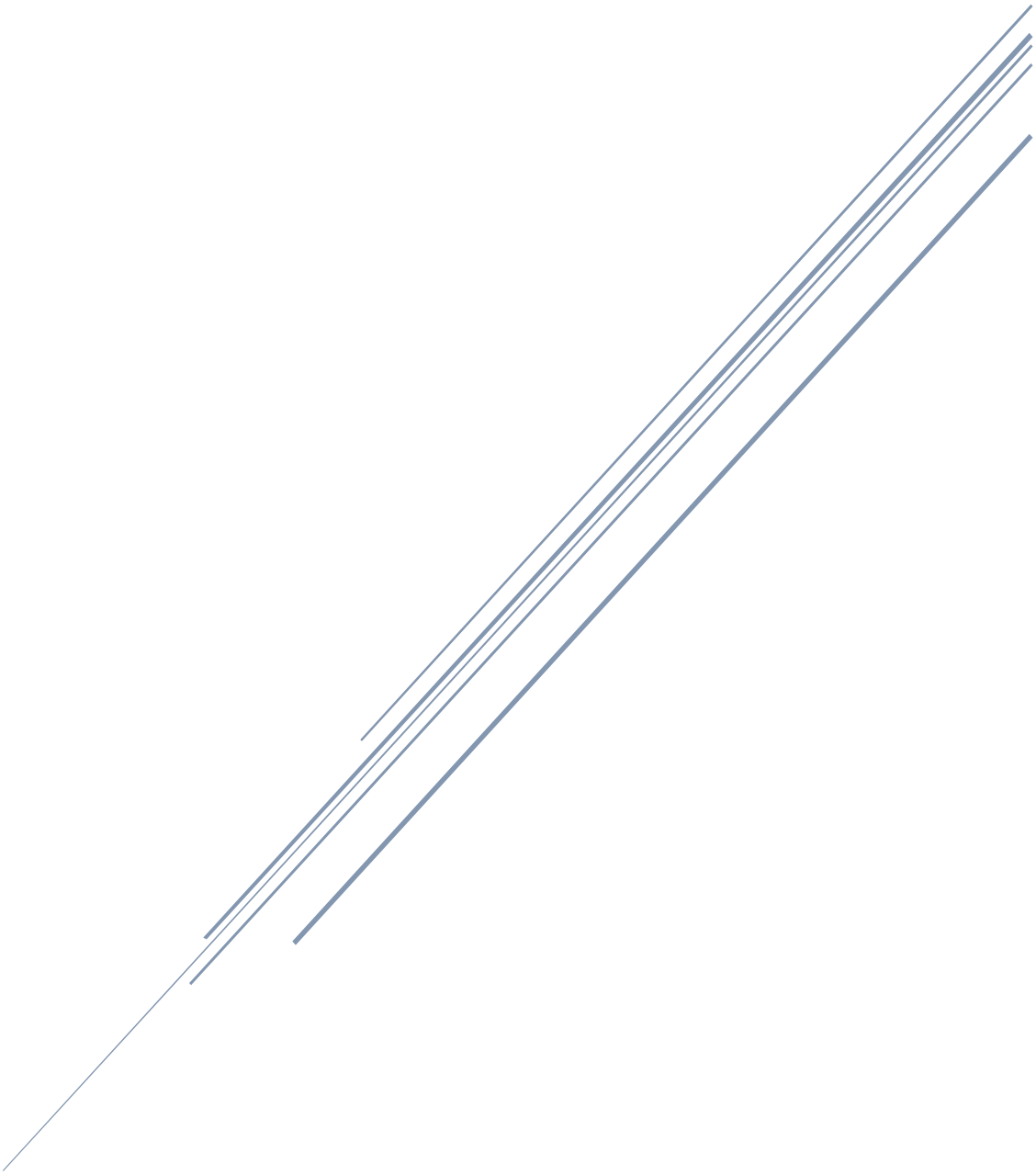
COM672 – COMPUTER VISION

Coursework One



Brógán McShane

Task One – Devising a QR Code Reader

Upon first look, the provided image "QR_1.jpg", wasn't entirely made up of black and white graylevel values, meaning that some form of thresholding needed to be undertook.

The first experiment of applying thresholding to the image was through the use of the matlab function *'graythresh'*. This function will compute a global threshold value by analysing the grayscale image, using this, we can apply the matlab function 'imbinarize' to convert a graylevel image to a binary image in conjunction with the calculated threshold value. However, upon examination of the returned result, it was clear that this method of thresholding wasn't accurate enough on its own.

The next experiment for applying an effective thresholding technique was through the use of imbinarize's adaptive thresholding option, where the thresholding value would be calculated for smaller regions in the image. However, again, this method did not provide the ideal result, with many pixels having rough edges, and some of the qr code cells not even looking like squares.

Upon reflection, a spatial filtering technique could be used in conjunction with either of these methods, where the average value of a cell would be assigned to the entire value of each pixel making up said cell. However, before applying this, other methods of thresholding would be visited.

Figure 1: Original QR Code (left), Binarized QR Code after applying global thresholding value returned by function graythresh (middle), Binarized QR Code after applying adaptive thresholding (right)

Next, an automatic thresholding algorithm would be written by the author, where the original grayscale QR code would be passed to a function that would compute the automatic thresholding value through analysis of the total number of graylevel values, the cumulative graylevel values and the mean graylevel values. The image would then be passed to another function that would loop through every pixel and determine whether the pixel value should be set to 255 or 0 by comparing the pixel's graylevel value to the automatic thresholding value. The result of this process proved to be effective in thresholding the image.

Following this, the image would be binarized through the use of the *imbinarize* function, and then be resized to 29 by 29 cells through the use of the *imresize* function. Although custom functions could be written to provide this same result, it was deemed unnecessary to undergo this process seeing that the built in matlab functions provide the same result, with equal standards of quality.

Figure 2: Original QR Code (left), QR Code after applying automatic thresholding (middle), Binarized 29 by 29 QR Code after applying automatic thresholding (right)

Task Two – Exploring and Overcoming Salt and Pepper Noise

After numerous levels of salt and pepper noise was added to the "QR_1.jpg" image, it was discovered that the original QR code reader algorithm would be continue to be effective for salt and pepper noise when set to a value in the range of 0 and 0.3.

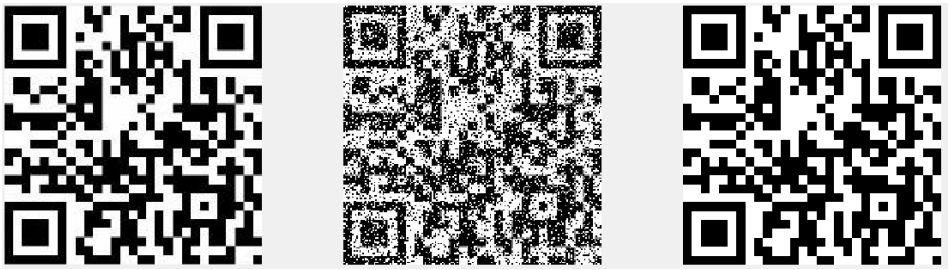


Figure 3: Original QR code (left), QR code with salt and pepper noise of 0.3 (Middle), QR Code returned from original reader function

However, the Original QR code reader would struggle with accurately reading a QR code that had salt and pepper noise above 0.3, meaning it should be improved through the use of spatial filtering.

The first spatial filtering technique utilized was median filtering. An algorithm was written to loop over every qr code cell in the image, take the median filter value of this cell, and assign it to a new image. This QR Code image, now containing median values for each of the noisy cells, would be passed to the QR code reader function to calculate the automatic thresholding value, apply the automatic thresholding value, then binarize and resize the image to 29 by 29 cels. However, the result of this proved the current process to be a weak solution when applying a salt and pepper noise value of over 0.35.

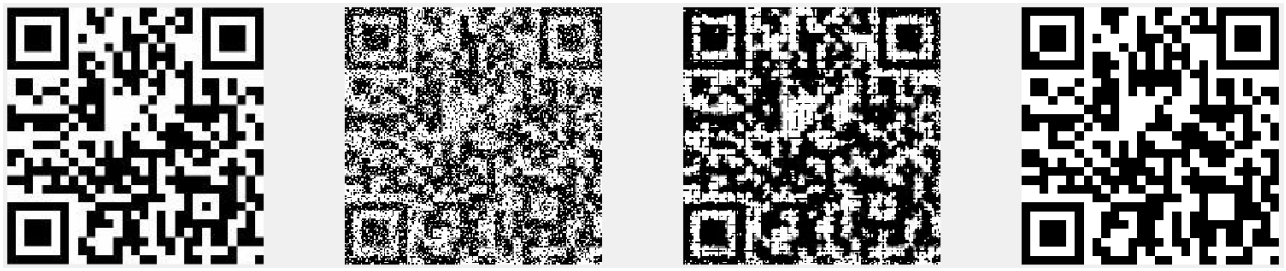


Figure 4 (From left to right): Original QR code, QR code with salt and pepper noise of 0.4, Salt and Pepper QR Code after median spatial filtering, QR Code Reader Result After analysing Median Filter Image

Following this, the Wiener spatial filter was used to increase the capability of the QR code reader to accurately analyse the QR code when the salt and pepper noise was set to above 0.35. Again, an algorithm would loop over every QR code cell in the image, however, instead of taking the median value of the cell under analysis, the Wiener filter would apply a linear averaging mask to the neighbourhood of pixel values that make up the current cell, the result of this filter would be set to the corresponding range of pixels in a matrix the same size of the original salt and pepper QR code. This process would continue until the algorithm had finished looping over the image. Afterwards, the image would be sharpened using the imsharpen method. Unlike the median filtering method, this technique proved to increase the accuracy of the QR code reader when an image with a salt and pepper noise in the range of 0.35 - ~ 0.4 was to be analysed.

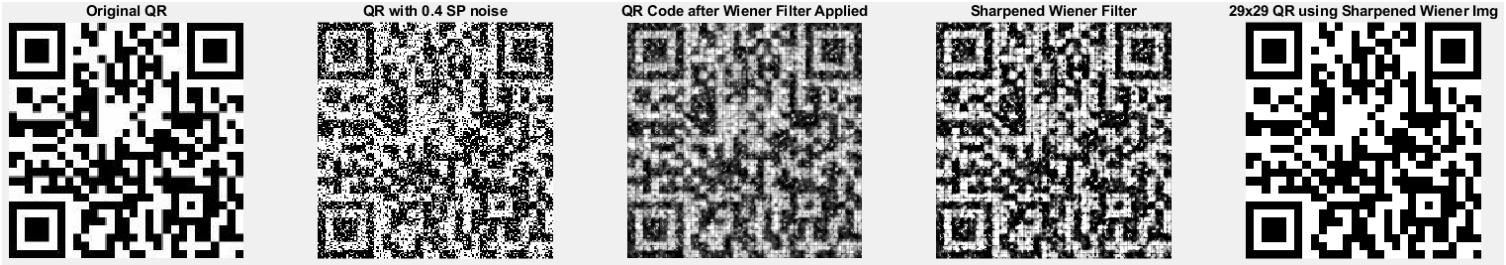


Figure 5 (From left to right): Original QR code, QR code with salt and pepper noise of 0.4, Salt and Pepper QR Code after wiener spatial filtering, Wiener filter Image after being sharpened, QR Code Reader result after analysing Wiener Filter Image

Task Three – Exploring and Overcoming Blurring

Similar to how the initial QR Code reader algorithm was still able to accurately read the image with a small amount of Salt and Pepper Noise, we see similar results when applying little blurring to the image. The QR code reader is effective in reading a blurry image in the range of 0 for the Gaussian Lowpass Filter and 0 for the Standard Deviation, up to and including 6 for the Gaussian Lowpass Filter and 6 for the Standard Deviation. However, the QR code reader will need to be improved upon in order to account for images which feature heavier blurring.

The initial effort to increase the robustness of the QR code reader included the development of a blind deconvolution algorithm, known for its effectiveness when no information about the blurring is known. However, the attempt of implementing this algorithm proved challenging. After the first set of results to calculate the Undersized PSF (point-spread function), the Oversized PSF and the true PSF provided insufficient results in deblurring the image, an attempt to improve the blind deconvolution algorithm was undertook. This attempt aimed to reduce the blurring effect by specifying a weighting function, through the analysis of finding sharp pixels using the edge function. Unfortunately, this further attempt didn't provide the expected result, and a different approach needed to be evaluated to deblur the image effectively.

Figure 7 (From left to right): Original QR code; QR code with a gaussian lowpass filter of size 10 and standard deviation of 10; QR code with an undersize array applied as the initial guess of the PSF; QR code with an oversized array applied as the second guess of the PSF; QR code with an true sized array applied as the final guess of the PSF; The weighted array generated using the true sized PSF; and the poorly "restored" image using the weighted array.

Following this, another attempt was made to implement a solution to deblur an image included the deconvolution of an image using the Wiener filter algorithm. This implementation seen success in deblurring the image where the lowpass filter and standard deviation values are both set to the same values used for the blurring of the image. This solution, however, is weak, as the PSF value used to blur the image must be known and then used in order to deblur the image – meaning this technique would not be effective in a real-world scenario.

Figure 8 (From left to right): Original QR code; QR code with a gaussian filter of 12 and standard deviation of 12; QR code after the Deconvolution Wiener Filter has been applied where the PSF values match the blurring PSF values; The 29 by 29 QR code after analysing the deblurred image

The next attempt of overcoming the blurring of an Image included separate experimentation of Laplacian Filtering and High Boost Filtering. Initially, a sharpening function was implemented using a Laplacian Filter that proved to not be useful when using a kernel of [-1 -1 -1; -1 8 -1; -1 -1 -1]. Following this, the High Boost Filter function was implemented, and seen success when the QR code had blurring effects in the range of 7 – 9 for the Gaussian Low Pass filter, and 7-10 for the Standard Deviation – improving the robustness of the QR code reader. One of the main components for the High Boost Filter function's success included using MatLab's *imsharpen* function once the filter had been applied – where estimates were set for the radius (set to 4) and amount (set to 1) parameters.

Figure 9: Unblurring using High Boost Filter (Zoom in on image for more details on each image)

Task Four – Generating the Binary Array When Size of QR Code is Unknown

To determine the size of a single cell, the author visually analysed constant features across both the first and second QR code images. The most constant features were the upper left, bottom left and upper right boxes – formally referred to as the "finder pattern" that allows a QR code scanner to determine the orientation of a QR code.

Using this information, three hypothesised methods could be used to find the size of a single cell of a QR code, and were to be evaluated after being implemented to determine which is most accurate: **method one** which, after applying automatic thresholding, would find the row and column value of the first white cell that features inside one of the Finder Pattern boxes, and then by subtracting the row and column value each by one you will find the rows and columns that make up the upper leftmost black pixel, which shares the same size of all QR code cells inside the image; **method two** being finding the inner solid black box that features inside one of the mentioned corners, and dividing it by 4 to find the size of a single cell; and **method three**, which finds the first white pixel's column location on the first row, then finding the first black pixel's column value after the first white pixel is found, then by subtracting the first black pixel's column value by the first white pixel's column value, and further subtracting one from the result, you could possibly find the length (and therefore the height) of a cell in the image .

Before any of the methods mentioned could be implemented, some image pre-processing had to take place to ensure that only the pixels that make up the QR code would be analysed. For this reason, a "cropping" function was deployed, as upon investigation, the second QR code seemed to have white image padding that was useless to the QR code reader.

The initial cropping function was designed to analyse all edges of the image after automatic thresholding had been deployed (edges include the top rows, bottom rows, leftmost column, and rightmost column), the image would be cropped accordingly until the only pixels that featured were those of the QR code – no matter the number of white padding layers on any edge.

Figure 10 – Analysis of the Cropping Function (From left to right): QR Code 1 before cropping (178 rows by 178 cols); QR Code 1 after cropping (178 rows by 178 cols – There was no white padding); QR Code 2 before cropping (667 rows by 667 cols); QR Code 2 after cropping (665 rows by 665 cols – white padding across all edges of image).

Next, the author decided to implement methods one and three previously discussed in order to find the size of a cell in the image.

Initially, **Method Three** was implemented as a function and seen successful analysis on the second QR code, however, after the function was passed the first QR code and resulted in a 16 by 16 reduced QR code instead of the known 29 by 29 dimensions, it was determined that this process would not be as versatile as it was previously thought, as upon second investigation, it was discovered that many white QR code cells can exist before the first black cell is reached.

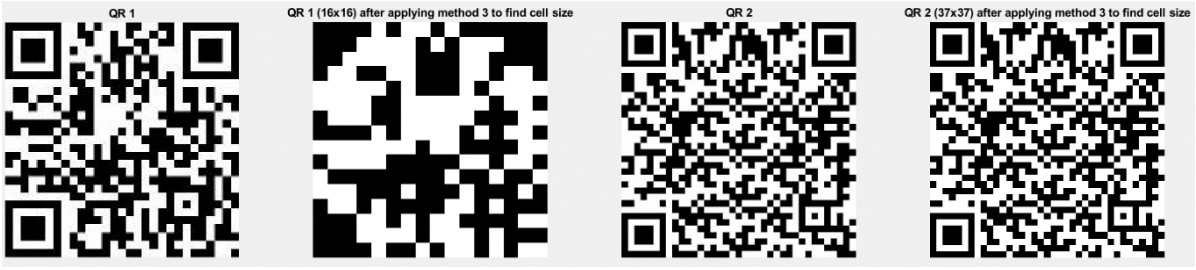


Figure 11: Analysis of Implementing Method Three to find the size of QR code Cells - Zoom in on image for more details

Next, **Method One** was implemented to find the size of a cell and the total number of cells in an image. This implemented function seen accurate results for both the first QR Code and the second QR code. Through this method's implementation, it was discovered that cells in QR1 had a width and height of 6 pixels, and cells in QR2 had a width and height of 18 pixels. Using this determined width and length, by dividing the rows and columns (both converted to doubles) by the width and height (also converted to doubles), and taking the floor value as the result, it was discovered that QR1 had a cell specification of 29 by 29 (as expected) and QR2 had a cell specification of 37 by 37. This method would be implemented into the QR code reader function, meaning that there is no need for the user to specify the number of cells in an image.

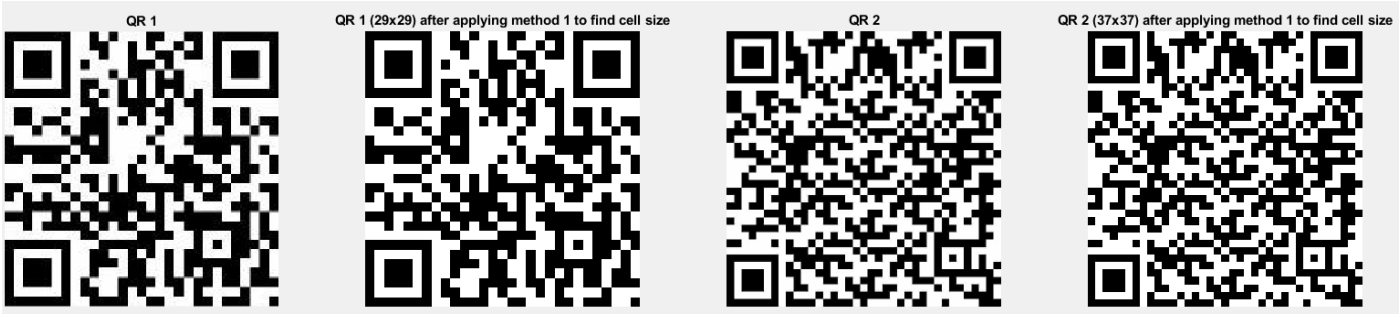


Figure 12 : Analysis of Implementing Method One to find the size of QR code Cells - Zoom in on image for more details

Task Five – Exploring and Overcoming Resized, Smaller Images

Although the QR Code Reader (now implemented to discover the number of cells in the image) performed well when analysing images that were resized to half of the original image's size, it was discovered that the reader would struggle with images that were resized (specifically when using decimal parameters with imresize function) to a quarter, a third and three quarters of the original image and would result in incorrect cell sizes.

Figure 13 : Analysis of the QR Reader on Resized, Smaller Images - Zoom in on image for more details

However, it was discovered that when specific pixels were used to resize the image (i.e. imresize(<original_img>, [222, NaN]) for an image 1/3 the size of the original), the original QR code reader seen accuracy for images ¼ of the original size, ½ the original size, 1/3 of the original size, and 2/3 of the original size.

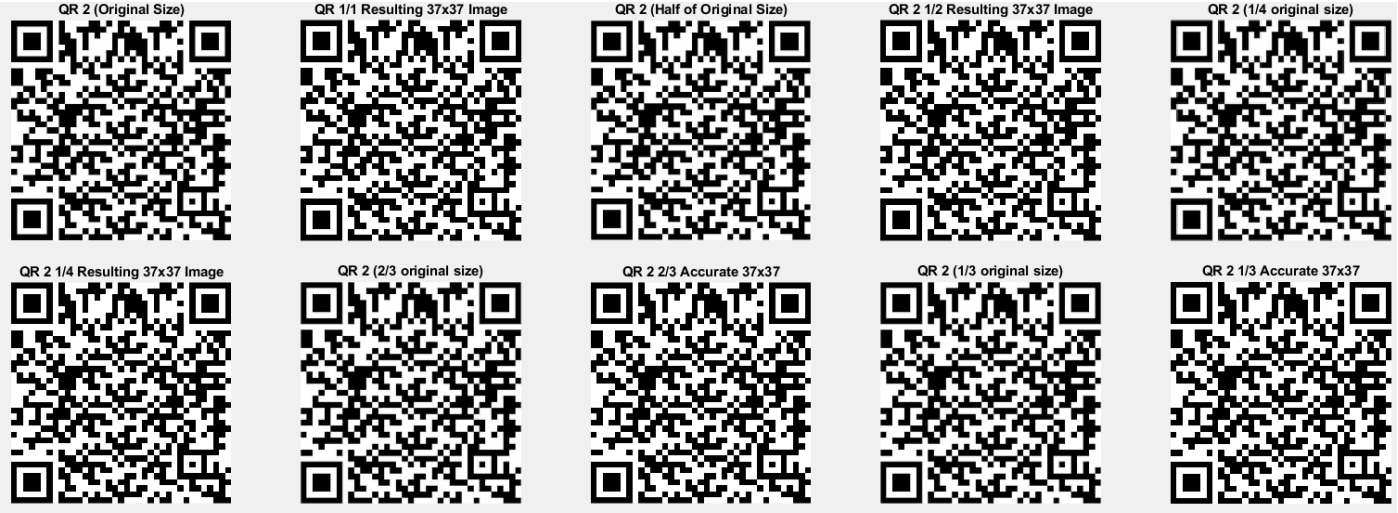


Figure 14: Analysis of the QR Reader on Resized, Smaller Images - Zoom in on image for more details

To experiment with images that were resized using decimal numbers, the methods to discover the size of a single cell and the total number of cells in an image would need to be revisited. An alternative method would be investigated, that again analyses the white cells closest to the Finder Pattern boxes in the QR code.

An alternative algorithm was written, where the size of the first white box along the first row would be calculated, and the size of the first white box along the first column would also be calculated. Using this information, the size (both height and width) of a single cell could be measured by dividing the width of the found cell by the width of the entire image as well as dividing the height of the found cell by the width of the entire image. Unfortunately, this algorithm struggled when trying to compute the size of a QR code that was ¼ the size of its original, for this reason, the original QR code reader will serve to fulfil this purpose of accurately reading QR codes when resized to smaller dimensions. However, this algorithm still exists within the codebase if needed to be demonstrated.

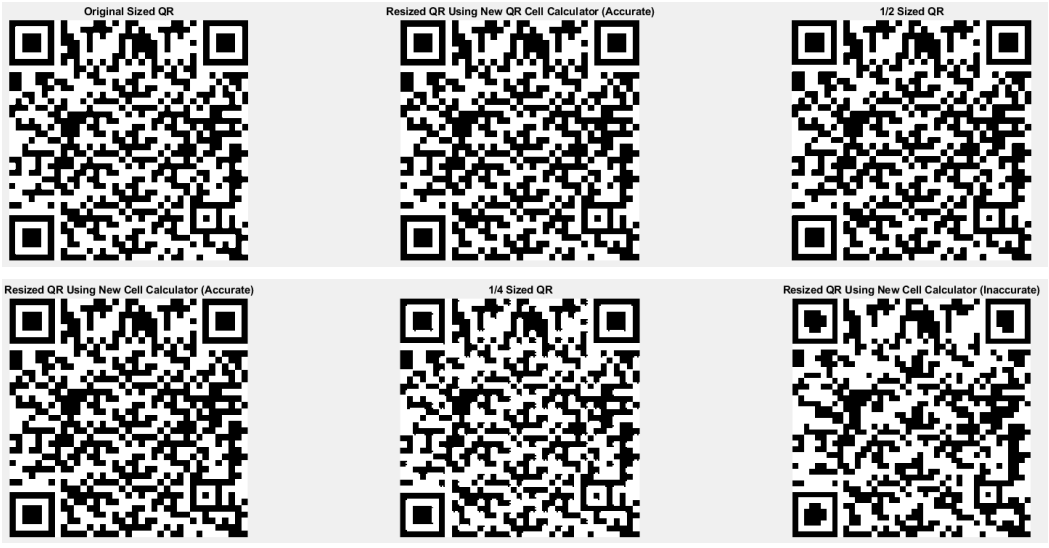


Figure 15: Analysis of the New QR Reader on Resized, Smaller Images - Zoom in on image for more details