Andrew Fernandez

CAP 4401

Project 4

Project 4 Report

This project was interesting because we got to use the OpenCV C++ library. It was interesting to go through their documentation and really get familiar with their methods and classes they use to process images. With this in mind, it was easier to understand how the library worked because of everything I've learned this semester. This report will cover the parameters file, compilation and execution instructions, and each function implemented in this project, including examples with images and their timings that are printed out every time.

I. Parameters

There is one file that is read by the program. It has very strict options for each line in the file. It includes the information for each ROI, parameters for the various functions for each ROI, and the images the appropriate functions should be run on. There are cycles for the lines. For every image/function that you want to have operated on, the first line will contain the name of the source image, what you wish to name the target image, the function that you want to operate over the image on, and the number of ROIs (n) used for the function. The following n lines will be information given for the parameters for each ROI for each function. Each of these n lines will contain the following information in the following order: x, y, sx, sy. If the function is hist_stretch, then following the ROI parameters will be two values: a and b used for thresholding. If the function is sobel_edge or comb_ops_sobel, then following the ROI parameters would be one value

that represents a threshold value. If the function is canny_edge or comb_ops_canny, then following the ROI parameters would be two values that represent two threshold values. Everything is separated by spaces. The first lines will be in this form: input_image output_image function. Where the function can be hist_stretch (for histogram stretching), hist_eq (for histogram equalization), canny_edge (for Canny edge detection), sobel edge (for Sobel edge detection), comb_ops_sobel (for combining histogram equalization and Sobel edge detection), comb_ops_canny (for combining, histogram equalization and Canny edge detection), and qr_decode (for QR detection and decoding). This qr_decode function is unique in that it is the only function that takes 0 in for the number of ROIs. The output image's name should also be the same as the input image's name as the function does not generate an image, just prints the QR code's message. All these functions are implemented using OpenCV's C++ library. A side-note is that the parameters file should be in the following path: /Project2/project/bin/. The following is an example of a properly constructed parameters file called parameters.txt:

project > bin > parameters.txt

```
floor.pgm floor_hist_stretch.pgm hist_stretch 2
0 0 100 250 100 200
400 0 100 300 50 100
lena.pgm lena_hist_stretch.pgm hist_stretch 2
0 0 200 200 100 200
0 300 200 150 150 200
floor.pgm floor_hist_eq.pgm hist_eq 2
0 0 100 250 100 200
400 0 100 300 50 100
lena.pgm lena_hist_eq.pgm hist_eq 2
0 0 200 200 100 200
0 300 200 150 150 200
tree.pgm tree_canny.pgm canny_edge 1
0 0 500 300 200 300
picasso.pgm picasso_canny.pgm canny_edge 2
0 0 200 200 100 200
600 100 300 200 150 200
tree.pgm tree_sobel.pgm sobel_edge 1
0 0 500 300 15
picasso.pgm picasso_sobel.pgm sobel_edge 2
0 0 200 200 15
600 100 300 100 20
tree.pgm tree_comb_ops_sobel.pgm comb_ops_sobel 1
0 0 500 300 15
picasso.pgm picasso_comb_ops_sobel.pgm comb_ops_sobel 2
0 0 200 200 15
600 100 300 100 20
tree.pgm tree_comb_ops_canny.pgm comb_ops_canny 1
0 0 500 300 200 300
picasso.pgm picasso_comb_ops_canny.pgm comb_ops_canny 2
0 0 200 200 100 200
600 100 300 200 150 200
qr_alphabet.jpg qr_alphabet.jpg qr_decode 0
qr_cap4401.png qr_cap4401.png qr_decode 0
qr_stuff_paper.jpg qr_stuff_paper.jpg qr_decode 0
qr_stuff_monitor.jpg qr_stuff_monitor.jpg qr_decode 0
qr_wikipedia.jpg qr_wikipedia.jpg qr_decode 0
```

II. Compilation and Execution

In order to compile this program, make needs to be run twice. Once inside the Project4 directory, run make inside of /Project4/iptools to compile both image.cpp and utility.cpp. Then travel to /Project4/project and run make to compile iptool.cpp. This will create the executable in /Project4/project/bin of the name iptool. To run the program, travel to /Project4/project/bin/ and execute ./iptool <file_name> where <file_name> is the name of your parameters file located inside /Project4/project/bin/.

III. Functions

A. Histogram Stretching

This function is under the utility class and is called cv_hist_stretch. It takes in the src and tgt Mat objects as well as a vector of ROIs for it. It generates the stretched image for the source image. It starts by creating a Mat object to store the src to temporarily store each stage of the image's processing for each ROI. It then loops through the regions and extracts the parameters. Then it loops through the Mat object of the original image and checks if the current pixel is within the ROI. If it is, then it looks at the current pixel and if it is less than the parameter a, it sets the target's pixel at that location to 0. If it is greater than the parameter b, then the target's pixel at that location is set to 255. If it fits neither of these conditions, then a new value is calculated using the same formula that I used from Project 2 and the target's pixel at this location is set to the new value. If it is not in the ROI, then the target's pixel at that location is just set to the original image's pixel at the same location. Here is my function:

```
88  void utility::cv_hist_stretch(Mat &src, Mat &tgt, const vector<roi>& regions) {
89      Mat temp_img;
90      temp_img = src.clone();
91      tgt = temp_img.clone();
92
93      for (int r = 0; r < regions.size(); r++) {
94          int x = regions.at(r).x;
95          int y = regions.at(r).y;
96          int sx = regions.at(r).sx;
97          int sy = regions.at(r).sy;
98          int a = regions.at(r).a;
99          int b = regions.at(r).b;
100
101         for (int i = 0; i < temp_img.rows; i++) {
102             for (int j = 0; j < temp_img.cols; j++) {
103                 if (
104                     i >= y &&
105                     i < (y + sy) &&
106                     j >= x &&
107                     j < (x + sx)
108                 ) {
109                     int curr_pixel = temp_img.at<uchar>(i, j);
110                     if (curr_pixel < a) {
111                         tgt.at<uchar>(i, j) = MINRGB;
112                     }
113                     else if (curr_pixel > b) {
114                         tgt.at<uchar>(i, j) = MAXRGB;
115                     }
116                     else {
117                         int newVal = (curr_pixel - a) * (255 / (b - a));
118                         tgt.at<uchar>(i, j) = checkValue(newVal);
119                     }
120                 }
121                 else {
122                     int newVal = temp_img.at<uchar>(i, j);
123                     tgt.at<uchar>(i, j) = checkValue(newVal);
124                 }
125             }
126         }
127         tgt.copyTo(temp_img);
128     }
129 }
```

Here is floor.pgm input image:

Here is the output floor_hist_stretch.pgm:

Here is the performance of the function on this image:

```
Hist Stretch time for floor.pgm = 12ms
```

Here is another input for lena.pgm:

Here is the output for lena_hist_stretch.pgm:

Here is the performance of this function on this image:

```
Hist Stretch time for lena.pgm = 17ms
```

This was a little slower than the other image. However it is only a few milliseconds so it is nothing significant. I will compare the two histogram modification OpenCV modules at the end of the Histogram Equalization section.

B. Histogram Equalization

This function is under the utility class as cv_hist_eq and it takes in a source and target Mat object as well as a vector of regions. It is the only function other than the QR Detection and Decoding that does not take any parameters. It is a pretty simple function and the OpenCV library handles most of the heavy lifting. I use the OpenCV function equalizeHist to generate a Mat object for the entire image. Then, I loop through the original image and if the pixel is in one of the ROIs, I set

it equal to the pixel at the corresponding location from the image that was generated from equalizeHist(). If it is not in a region, then the pixel is just set to the same pixel from the corresponding location in the source's image. At the end I use OpenCV's copyTo method of the Mat class to generate the image into the target image. Here is my function:

```
132  void utility::cv_hist_eq(cv::Mat &src, cv::Mat &tgt, const vector<roi>& regions) {
133      Mat temp_img;
134      temp_img = src.clone();
135      tgt = temp_img.clone();
136
137      Mat eq_tgt;
138      eq_tgt = tgt.clone();
139      equalizeHist(temp_img, eq_tgt);
140
141      for (int r = 0; r < regions.size(); r++) {
142          int x = regions.at(r).x;
143          int y = regions.at(r).y;
144          int sx = regions.at(r).sx;
145          int sy = regions.at(r).sy;
146
147          for (int i = 0; i < temp_img.rows; i++) {
148              for (int j = 0; j < temp_img.cols; j++) {
149                  if (
150                      i >= y &&
151                      i < (y + sy) &&
152                      j >= x &&
153                      j < (x + sx)
154                  ) {
155                      tgt.at<uchar>(i, j) = checkValue(eq_tgt.at<uchar>(i, j));
156                  }
157                  else {
158                      tgt.at<uchar>(i, j) = checkValue(temp_img.at<uchar>(i, j));
159                  }
160              }
161          }
162          tgt.copyTo(temp_img);
163      }
164  }
```

Here is floor.pgm as input:

Here is floor_hist_eq.pgm as output:

Here is the performance of this function for this image:

```
Hist Eq time for floor.pgm = 14ms
```

Here is another input, lena.pgm:

Here is the output, lena_hist_eq.pgm:

Here is the performance for this function on this image:

```
Hist Eq time for lena.pgm = 20ms
```

Comparing the two Histogram Modification functions, it seems like OpenCV does some extra operations when Equalizing than while Stretching because both the images were found to be generated just a little slower than my stretching

functions.This makes sense to me because I think the equalized images are much clearer and easier to see after being operated on. So the extra functionality it has under the hood slightly slows the function down but creates a clearer picture.

C. Canny Edge Detection

This function is under the utility class as the name cv_canny_edge. It takes a Mat object source and target as well as a vector of ROIs. It is similar to the histogram equalization function in that it lets the OpenCV function do most of the heavy lifting. The function takes two thresholds as arguments. They are T1 and T2 and they are taken from my parameters file and they are stored with each ROI. If the pixel being looked at is inside the region, then it sets it to the pixel of the image generated by OpenCV's Canny() function. Otherwise it is just set to the source's pixel at the location. Here is my function:

```cpp
void utility::cv_canny_edge(cv::Mat &src, cv::Mat &tgt, const vector<roi>& regions) {
    Mat temp_img;
    temp_img = src.clone();
    tgt = temp_img.clone();

    for (int r = 0; r < regions.size(); r++) {
        int x = regions.at(r).x;
        int y = regions.at(r).y;
        int sx = regions.at(r).sx;
        int sy = regions.at(r).sy;
        int T1 = regions.at(r).canny_T1;
        int T2 = regions.at(r).canny_T2;

        // creating canny edge detected image using parameters for ROI
        Mat canny_tgt;
        canny_tgt = tgt.clone();
        Canny(temp_img, canny_tgt, T1, T2);

        for (int i = 0; i < temp_img.rows; i++) {
            for (int j = 0; j < temp_img.cols; j++) {
                if (
                    i >= y &&
                    i < (y + sy) &&
                    j >= x &&
                    j < (x + sx)
                ) {
                    tgt.at<uchar>(i, j) = checkValue(canny_tgt.at<uchar>(i, j));
                }
                else {
                    tgt.at<uchar>(i, j) = checkValue(temp_img.at<uchar>(i, j));
                }
            }
        }
        tgt.copyTo(temp_img);
    }
}
```

Here is tree.pgm input image:

Here is tree_canny.pgm output image:

Here is the performance of this function for this image:

```
Canny Edge time for tree.pgm = 14ms
```

Here is picasso.pgm input image:

Here is picasso_canny.pgm output image:

Here is the performance of this function for this image:

```
Canny Edge time for picasso.pgm = 32ms
```

As you can see, the time for picasso.pgm is about double the time of the tree.pgm. This makes sense because the area of the image is larger than the tree's. I will be comparing this function with OpenCV's Sobel function at the end of the Sobel Edge Detection section.

D. Sobel Edge Detection

This function is under the utility class as cv_sobel_edge. It takes in the source and target images as Mat objects and a vector of ROIs to be operated on. I begin this function by creating variables for ddepth and ksize for OpenCV's Sobel function that I will be using later on. I then loop through my regions, extract appropriate parameters, and then create two Mat objects for the dx and dy. Then, I use these images for my target image when I call the Sobel function on the source for each corresponding image (dx or dy). Then, I loop through the image and calculate the gradient of the x and y directions respectively. Then, I find the magnitude by using the formula. Finally, I threshold the magnitude by using the T read from the parameters file to binarize the image. Here is the main body of my function:

```cpp
void utility::cv_sobel_edge(cv::Mat &src, cv::Mat &tgt, const vector<roi>& regions) {
    Mat temp_img;
    temp_img = src.clone();
    tgt = temp_img.clone();

    // params for Sobel OpenCV Function
    int ddepth = -1;
    int ksize = 3;

    for (int r = 0; r < regions.size(); r++) {
        int x = regions.at(r).x;
        int y = regions.at(r).y;
        int sx = regions.at(r).sx;
        int sy = regions.at(r).sy;
        int T = regions.at(r).sobel_T;

        Mat dx_sobel, dy_sobel;
        dx_sobel = temp_img(Rect(x, y, sx, sy));
        dy_sobel = temp_img(Rect(x, y, sx, sy));

        Sobel(temp_img, dx_sobel, ddepth, 1, 0, ksize);
        Sobel(temp_img, dy_sobel, ddepth, 0, 1, ksize);

        for (int i = 0; i < temp_img.rows; i++) {
            for (int j = 0; j < temp_img.cols; j++) {
                if (
                    i >= y &&
                    i < (y + sy) &&
                    j >= x &&
                    j < (x + sx)
                ) {
                    double gx = dx_sobel.at<uchar>(i - y, j - x) / 8;
                    double gy = dy_sobel.at<uchar>(i - y, j - x) / 8;

                    double magnitude = sqrt(pow(gx, 2) + (pow(gy, 2)));

                    if (magnitude < T) {
                        tgt.at<uchar>(i, j) = MINRGB;
                    }
                    else {
                        tgt.at<uchar>(i, j) = MAXRGB;
                    }
                }
                else {
                    tgt.at<uchar>(i, j) = checkValue(temp_img.at<uchar>(i, j));
                }
            }
        }
```

Here is tree.pgm input image:

Here is tree_sobel.pgm output image:

Here is the performance of the function for this image:

```
Sobel Edge time for tree.pgm = 64ms
```

Here is the picasso.pgm input:

Here is picasso_sobel.pgm output image:

Here is the performance of the function on this image:

```
Sobel Edge time for picasso.pgm = 134ms
```

This Sobel function takes a significant amount longer than the Canny function. This may be because it is called twice for the x and y direction. But it is necessary in order to get readable results. In terms of output images, I feel like Canny did a better overall job at edge detection. I think the branches on tree.pgm are much clearer than Sobel's output image. Also, Canny shines during operating picasso.pgm because it is really clear to see all the edges of it. Sobel does not show all of the edges. OpenCV's Sobel function seems to be just a little bit faster performance wise compared to my Sobel function from Project 3. However

OpenCV's Canny function is much faster than both the OpenCV Sobel function and my Sobel function from Project 3.

E. Combining Histogram Equalization and Sobel Edge Detection

This function can be found under the utilities class with the name cv_comb_ops_sobel. It takes in the source and target images as Mat objects, the vector of ROIs, and the outfile name. It generates 2 images. One of the edges detected after being equalized and another of the subtraction of this image from the equalized image, or just the histogram equalized image. This function is fairly simple. It just takes advantage of the functions that I have already written in cv_hist_eq and cv_sobel_edge. It finds the difference between the images generated by these functions and outputs them respectively. Here is my function:

```
258  void utility::cv_comb_ops_sobel(cv::Mat &src, cv::Mat &tgt, const vector<roi>& regions, char* outfile) {
259      Mat hist_eq_img, sobel_eqd_img;
260
261      cv_hist_eq(src, hist_eq_img, regions);
262      cv_sobel_edge(hist_eq_img, sobel_eqd_img, regions);
263
264      Mat diff_img = hist_eq_img - sobel_eqd_img;
265
266      tgt = sobel_eqd_img.clone();
267
268      char diff_img_name[100] = "diff_img_";
269      imwrite(strcat(diff_img_name, outfile), diff_img);
270  }
```

I used the same input images for tree.pgm and picasso.pgm as above.

Here is the output image tree_comb_ops_sobel.pgm:

Here is the output of the difference:

It is just the equalized version of the image. The black part is there because it got subtracted from not being changed.

Here is the performance of this function for this image:

```
Combine Ops for HE and Sobel ED time for tree.pgm = 76ms
```

Histogram Equalization seemed to improve the quality of the edge detection, but not by much. The images looked similar, but it was not a major difference.

Here is the output picasso_comb_ops_sobel.pgm:

Here is the output of the difference image:

It is just the equalized image.

Here is the performance of the function for this image:

```
Combine Ops for HE and Sobel ED time for picasso.pgm = 158ms
```

Again, I think equalizing before edge detection helped out the output image. More edges showed up when equalizing prior to calling Sobel in the output file.

F. Combining Histogram Equalization and Canny Edge Detection

This function can be found under the utilities class with the name cv_comb_ops_canny. It takes in the source and target images as Mat objects, the vector of ROIs, and the outfile name. It generates 2 images. One of the edges detected after being equalized and another of the subtraction of this image from the equalized image, or just the histogram equalized image. This function is fairly simple. It just takes advantage of the functions that I have already written in cv_hist_eq and cv_canny_edge. It finds the difference between the images generated by these functions and outputs them respectively. Here is my function:

```
273  void utility::cv_comb_ops_canny(cv::Mat &src, cv::Mat &tgt, const vector<roi>& regions, char* outfile) {
274      Mat hist_eq_img, canny_eqd_img;
275
276      cv_hist_eq(src, hist_eq_img, regions);
277      cv_canny_edge(hist_eq_img, canny_eqd_img, regions);
278
279      Mat diff_img = hist_eq_img - canny_eqd_img;
280
281      tgt = canny_eqd_img.clone();
282
283      char diff_img_name[100] = "diff_img_";
284      imwrite(strcat(diff_img_name, outfile), diff_img);
285  }
```

I used the same input images for tree.pgm and picasso.pgm as above.

Here is the output for tree_comb_ops_canny.pgm:

Here is the difference image:

Again, it is just the equalized image.

Here is the performance for this function on this image:

```
Combine Ops for HE and Canny ED time for tree.pgm = 25ms
```

This function was significantly slower than the combined operations that used the Sobel function. I feel like the edges it found were more bolded, but it found less edges than the Sobel operator.

Here is the output for picasso_comb_ops_canny.pgm:

Here is the difference image:

Again, it is just the equalized image.

Here is the performance of this function for this image:

```
Combine Ops for HE and Canny ED time for picasso.pgm = 53ms
```

I think in the picasso.pgm input image, you can really see that the Canny function did a really good job of detecting edges after being equalized. It definitely did a better job of detecting images than when it was not equalized prior to being

operated on. In conclusion, applying histogram equalization improves edge detection when using either Sobel or Canny methods.

G. QR Detection and Decoding

This function is found under the utilities class as cv_qr_decode and it takes in the source and target images as Mat objects and the name of the outfile as arguments. All it does is use OpenCV's QRCodeDetector object on the image input before and after being histogram equalized. It outputs the encoded message to the console, or it tells you that it failed to detect/decode the QR image. Here is my function:

```
288  void utility::cv_qr_decode(cv::Mat &src, cv::Mat &tgt, char* outfile) {
289      cout << endl;
290      Mat src_hist_img;
291      equalizeHist(src, src_hist_img);
292
293      QRCodeDetector qrd = QRCodeDetector();
294      std::string msg = qrd.detectAndDecode(src);
295      cout << "Original QR Message Decoded: " << msg << endl;
296
297      msg = qrd.detectAndDecode(src_hist_img);
298      if (msg.length() > 0)
299          cout << "Histogram Equalized QR Message Decoded: " << msg << endl;
300      else
301          cout << "Histogram Equalized QR Message Failed to be Decoded" << endl;
302
303      char hist_qr_name[100] = "hist_";
304      imwrite(strcat(hist_qr_name, outfile), src_hist_img);
305      cout << endl;
306  }
```

I did not have access to a printer at home, so I could not use additional input images other than one that I found on Google. The rest were taken as the sample images used on the TA's site. Here are all my input images, their histogram equalized versions, and the text output for each of them.

qr_alphabet.jpg

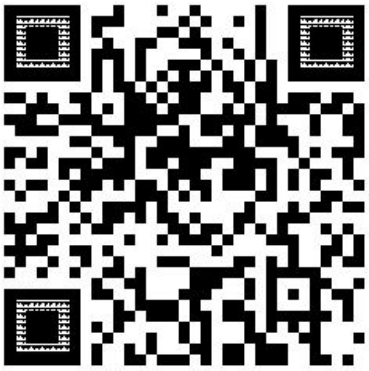
histogram equalized

Here is the text output to the console:

```
Original QR Message Decoded: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy
z
Histogram Equalized QR Message Failed to be Decoded

QR detection and decoding time for qr_alphabet.jpg = 906ms
```

qr_cap4401.jpg



histogram equalized

Here is the text output to the console:

```
Original QR Message Decoded: http://marathon.csee.usf.edu/~chihyun/index_CAP4401
.html
Histogram Equalized QR Message Failed to be Decoded

QR detection and decoding time for qr_cap4401.png = 715ms
```

qr_stuff_monitor.jpg

histogram equalized

Here is the text output to the console:

```
Original QR Message Decoded: http://www.qrstuff.com
Histogram Equalized QR Message Failed to be Decoded

QR detection and decoding time for qr_stuff_monitor.jpg = 1429ms
```

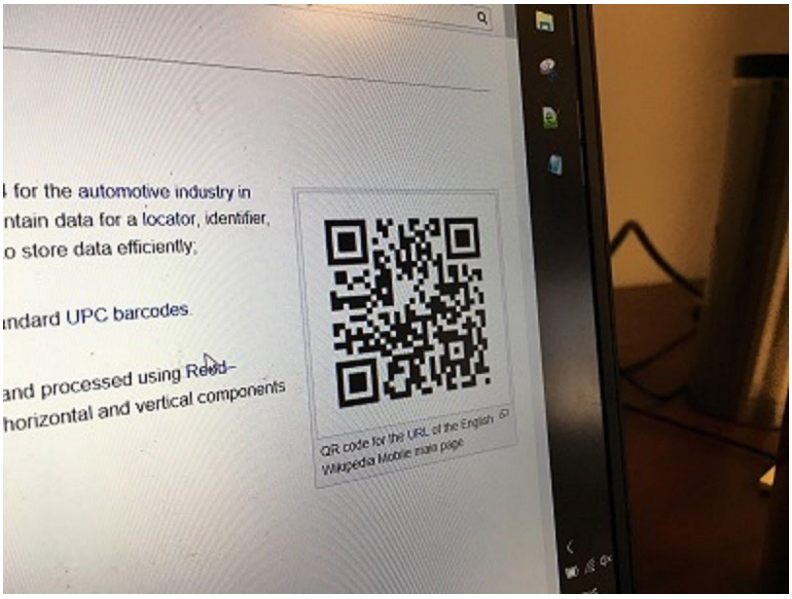

qr_stuff_paper.jpg


histogram equalize

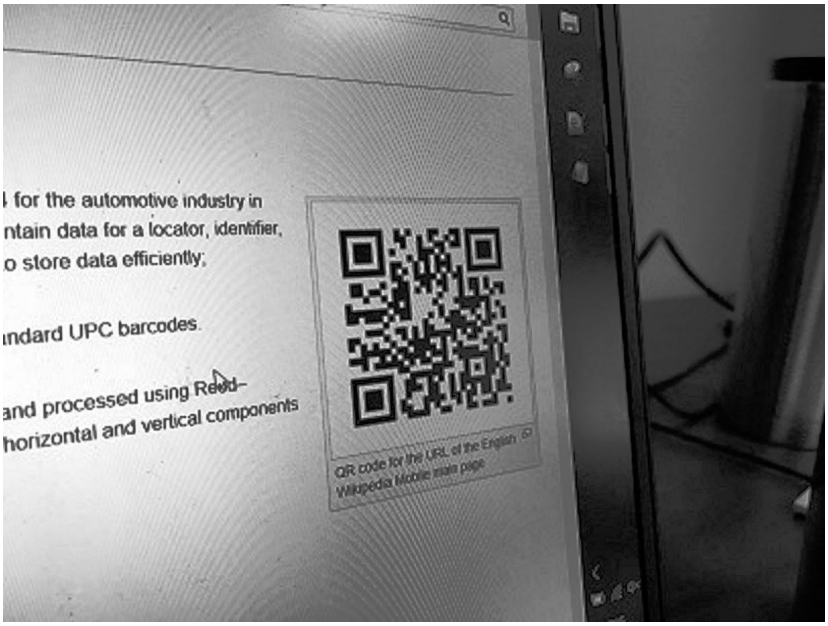
Here is the text output to the console:

```
Original QR Message Decoded: http://www.qrstuff.com
Histogram Equalized QR Message Failed to be Decoded

QR detection and decoding time for qr_stuff_paper.jpg = 951ms
```



qr_wikipedia.jpg



histogram equalized

Here is the text output to the console:

```
Original QR Message Decoded: http://en.m.wikipedia.org
Histogram Equalized QR Message Decoded: http://en.m.wikipedia.org

QR detection and decoding time for qr_wikipedia.jpg = 2530ms
```

This image was unique because it was the only one that was still able to be detected after being histogram equalized. My hypothesis for this is the angle that it was taken in was unique enough that the equalizeHist function from OpenCV did not distort the actual QR code like it did the other images. The QRCodeDetector object was still capable of detecting and decoding the message of the QR code in this image. However, overall, I would say that histogram equalization definitely does not help detecting and decoding QR codes as can be seen from my results. It distorts the code in such a way that the QR code loses meaning and OpenCV cannot read it. Performance wise, they are slower than the other functions in this report. This makes sense because the function must detect

it first and then decode it. The largest QR image took ~2 seconds to completely find.

IV. Conclusion

The performance of all the functions were analyzed in their corresponding sections above. This was a very beneficial project. I learned a lot more about image processing by using the OpenCV library. Having to read the documentation really helped me realize why some things work the way they do and that is good information to know I believe. Some troubles that I had while working with this library was not knowing what to use for the at method of the Mat class. At first I was using <int> when I should have been using <uchar>. This makes sense now because pixel intensities are only 0-255 and are not negative values. It was definitely a great experience to read through the documentation to figure out how to implement these functions that I had done in the past. It was definitely much easier to implement some functions, like the Sobel operator, when OpenCV does the heavy lifting and abstracts the main parts. Overall, it was a great and fun project that definitely took some time to complete and had its own challenges.