Object Recognition and Color Classification for Solving a Rubik's Cube

Brian Quiroz
University of Kansas
Lawrence, KS
r229q057@ku.edu

***Abstract*—This paper discusses approaches for object recognition and color classification in the context of solving a Rubik's cube. Specifically, the aim is to detect a face of a Rubik's cube and recognize and classify the different colors on that face using MATLAB. Once the colors on each face are detected, they are meant to be used by a C++ program to find the specific state of the cube and provide a solution for the cube. The focus of this paper is on the Computer Vision side of the cube: cube detection and color classification. After extensive testing, I determined that the algorithm is not very robust to changes in scale, translation, rotation, background, or lighting. However, if the pictures are taken correctly, the results of color classification give a 97% accuracy.**

I. Introduction

This paper discusses methods used to detect the state of a Rubik's cube given one picture of each face. This process can be divided int two subtasks to be performed for each face: (1) Detect the Rubik's cube and (2) Classify its colors into one of six discrete categories: red, green, blue, orange, yellow, and white. To accomplish this, we utilize a MATLAB script to process the six images and identify each color. The input to this program is 6 images and the output is a text file showing the colors of each sticker on each face, represented by the upper case first letter of each color.

The aim of the approach presented in this paper is simplicity of implementation. Hence, I have chosen not to use any complicated image processing algorithm nor any machine learning techniques. Furthermore, unlike other approaches that focus on efficiency and robustness [1], the focus of this approach is on simplicity and reliability. That is, pictures taken in "standard form", as shown in (1), should yield high accuracy. Future work will involve heavily improving the robustness and efficiency of the algorithm.

II. Related Work

The task of correctly recognizing colors in a Rubik's cube has been attempted several times before. We can divide these approaches into those that do not make use of machine learning techniques and those that do.

A. Non-Machine Learning Approaches

Many approaches detect Rubik's cubes and the colors within each face using only classical image processing techniques. Unfortunately, a lot of these approaches lack academic documentation and are presented only via YouTube videos, blog articles, or GitHub repositories [2]. They typically start the process by converting the images to RGB and blurring them. Then they employ some edge detection technique, such as Canny or Laplacian of Gaussian [1]. Lastly, color detection of each individual sticker can be done in a variety of ways. For examples, one approach uses a histogram of the hues of each sticker to find the most frequent hues [1].

B. Machine Learning Approaches

Lately, convolutional neural networks are used to solve a variety of problems in computer vision, and the problem of classifying colors in a Rubik's cube is no exception. For example, one approach used a CNN to find the approximate location and scale of the cube in the image [2]. They also used K-means clustering to aid in the segmentation of the image and a random forest classifier to determine if superpixels are cube pieces. Lastly, they used a Support Vector Machine for color recognition. I will not be using any of these approaches for the sake of simplicity of implementation.

III. Cube Detection

In the following section, I will discuss each step of my process of cube detection.

A. Photo Acquistion

The first step of the process is to obtain 6 pictures of the different faces of the Rubik's cube. The algorithm is not very robust and is generally not invariant to scale, translation, rotation, background, or lighting. Preferably, pictures should be taken in a light background and under a light yellow or white light (lamp) pointed above the cube.

Additionally, the Rubik's cube should be centered and have its edges parallel to the edges of the image. The picture should be taken at approximately the same distance as in (1). Even though the algorithm is somewhat flexible to variations (see (4)), this standard version of the photographs is strongly suggested for accurate results.

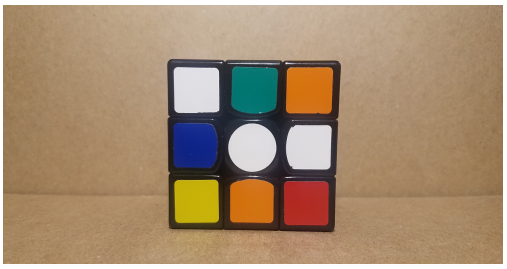

Figure 1

What should be strictly enforced, however, is the specific order of the photographs. Knowing the colors of each face of a Rubik's cube is not enough to reconstruct the state of a cube. Information about the relative orientation of each piece is equally important. To simplify the process of finding the orientation of each piece, the solver program assumes that the pictures of the faces of the Rubik's cube are input in the following order: yellow, orange, blue, red, green, and white. The solver takes information about the colors in that order and reconstructs the state of the cube. Hence, images should be input to the cube detection program in that order.

For testing, I will sometimes run the program with several pictures of the same face or without regard for the order mentioned above. This is because I will typically be testing either the cube detection or the color recognition and the state of the cue itself is not relevant. However, the way to properly run the program in practice is as indicated in the paragraph above.

B. Grayscale Conversion

To simplify the process of detecting a cube, we start by converting the image from RGB to grayscale.

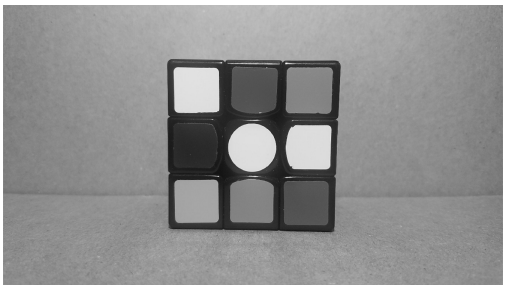

Figure 2

C. Median Filtering

The next step is to perform median filtering. We do this to remove the noise in the image. I have experimentally determined that median filtering and Gaussian filtering give similar results for cube line detection. I chose to use median filtering.

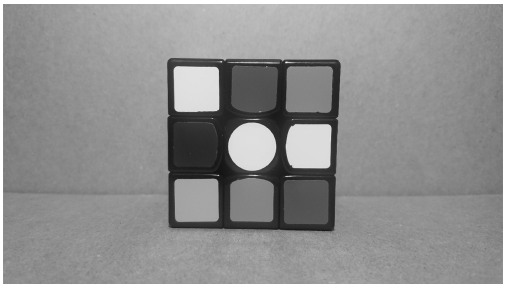

Figure 3

Median filtering is the process whereby, for each pixel, we replace that pixel with the median of an $n \times n$ neighborhood around it [3].

D. Thresholding

Thresholding is a technique that is used to convert images to binary images. There have been many techniques and algorithms used for both analog and digital thresholding. Some of the more modern techniques involve dynamically changing the threshold [4]. However, to find the edges of a Rubik's cube a simple thresholding function is sufficient.

Figure 4

It is important to note that the threshold value of choice depends on the lighting, contrast, and background. In particular, I successfully ran my algorithm with two different sets of images with slightly different backgrounds. Figure (4) shows the different backgrounds for the cube used in the different image sets. The algorithm is able to detect the cube in both image sets, but the threshold must be changed to adjust for the background change.

Figure 5 shows the images in (4) after thresholding with the different threshold values. For the first type of image, a threshold of 0.25 gives the best results. For the second type of image, a threshold of 0.175 works best. The second image requires a lower threshold because we do not want to allow as many gray values because of the darker background. I will be using a threshold of 0.25 as the standard for most of my experiments.

E. Edge Detection

The next step is to detect the edges in the image. The goal in this step is to highlight the edges well enough so that the line detector will find all the lines. First, we rotate the image counterclockwise by 33 degrees. Then, we apply an edge detector.

Figure 5

There has been much discussion about which edge detector is best [5][6]. Gradient-based operators for edge enhancement such as Roberts, Sobel, and Prewitt typically work well for simple applications. In contrast, more advanced techniques such as Laplacian of Gaussian and Canny work well for images with a lot of noise or complex edges [3]. For example, [5] determined that the Canny edge detector worked best for their sample images. They did not mention, however, that the images they used require a more "advance" technique such as the Canny edge detector, but some applications do not require such a technique.

Detecting the edges of a Rubik's cube given images such as the ones in (5), where noise has been greatly removed and the

edges are simple, requires a simple gradient edge detection algorithm. Since the Roberts edge detector uses a 2x2 kernel, it does not carry enough information regarding the direction of an edge [3]. The Sobel edge detector has more noise suppression than the Prewitt edge detector. However, as mentioned earlier, since we have smoothed our image and removed noise, further smoothing is unnecessary. In fact, I have experimentally determined that the Prewitt edge detector works best for this application. Figure 6 shows the Prewitt edge detector applied to the image after thresholding and a 33-degree rotation.

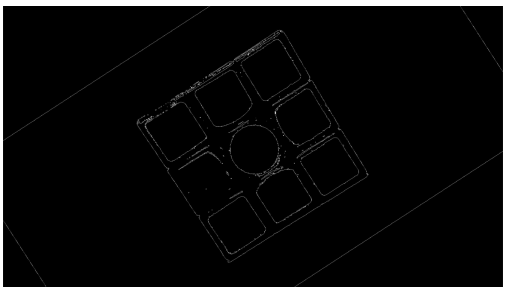

Figure 6

F. Line Detection

My goal of performing line detection is to detect the line's endpoints and then perform corner detection to select which of these endpoints are the four corners of the cube. Given the edge map provided by the Prewitt edge detector, we can find the lines and their endpoints using the Hough transform.

We transform points in cartesian coordinates in the x-y (picture) plane into sinusoidal curves in the θ-ρ (parameter) plane defined by:

$$\rho = x\cos\theta + y\sin\theta \quad [7].$$

We then use the fact that the intersection of curves in the parameter plane indicate that the corresponding curves in the picture plane are colinear. To avoid the computational cost of exhaustively finding all intersection in thee parameter plane, we quantize this plane into a grid of squares [7]. We can think of this grid as a large histogram. Each square in the grid serves as a bin; we count the number of lines passing through it. Essentially this is the same concept as counting the intersections but by using an area of intersection rather than a point, we reduce the computational complexity.

A graph of the intersections is shown in (7). Although the sinusoidal curves are hard to see, we can see seven squares that highlight the intersections in the parameter plane. These points of intersection represent colinear points in the picture plane.

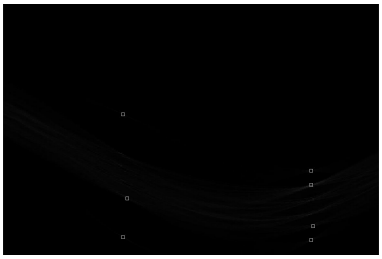

Figure 7

Notice the points can be separated into two groups, each almost forming a straight vertical line. This is because the detected lines approximately have one of two directions or theta values. Recall that the picture is taken in in such a way that the edges of the cube parallel to the edges of the picture. Hence, all lines either have the direction that was originally vertical or the direction that was originally horizontal (before rotation).

Figure 8 shows the lines detected using the Prewitt edge detector and the Hough transform superimposed onto the image after thresholding and rotation. The start points are yellow, the end points are red, and the lines are green. The longest line is identified and showed in cyan.

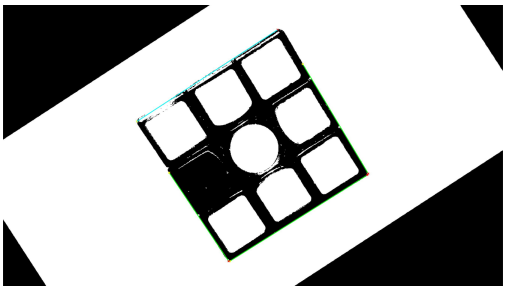

Figure 8

Notice that the lines corresponding to the edges of the pictures are ignored. This was done by ignoring lines that either have

$$\theta = 57° \text{ and } (\rho \leq 580 \text{ or } \rho \geq 2020)$$

or

$$\theta = -33° \text{ and } (\rho \leq -599 \text{ or } \rho \geq 1961).$$

I have experimentally determined that lines with these characteristics are either edges or otherwise irrelevant. Furthermore, to remove outliers produced by noise, I ignored lines whose endpoints in cartesian coordinates have x values such that:

$$x \leq 230$$

or

$$x \geq 2330.$$

These values have also been experimentally determined. They also somewhat constrain what area the cube can be in when taking pictures as will be illustrated by some of the experiments. The values were chosen so that the most outliers are discarded while retaining the greatest possible area for the cube.

G. Corner Detection

The Hough transform will rarely detect all four edge lines of the cube. Nonetheless, as long as the four corners of the cube are among the endpoints of the lines found by the Hough transform, we can determine the four corners of the cube.

To find the corners, we use a heuristic based on an observation. Let us label the corners starting from the upper

leftmost corner and proceeding clockwise as c1, c2, c3, and c4, respectively. Consider again (8). The x-y plane starts at the top left corner, and x grows to the right while y grows downwards. Notice that c1 has the smallest x-value among the endpoints. Similarly, c2 has the smallest y-value among the endpoints, c3 has the largest x-value among the endpoints, and c4 has the largest y-value among the endpoints. Hence, to find c1, c2, c3, and c4 we must find the smallest and largest x- and y-values and label those points as the corner they correspond to.

Figure 9 shows the detected points c1…c4. In the figure, c1 is yellow, c2 is red, c3 is blue, and c4 is green. The points shown in the figure are larger than the actual points, for the purpose of ease of visualization. Since it is still difficult to see the points, please zoom in accordingly to see them. The points were identified on the tilted image, but the displayed image has been rotated back to its original position and the identified points have also been rotated along the center of the image.

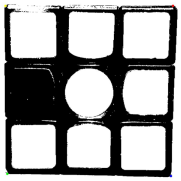

Figure 9

To verify the accuracy of corner detection, we minimize the Mean Squared Error of the square generated by connecting the corners. The length of the longest line in the image is taken to be the "ground truth" for the length of the edge of the cube. Since we have removed the image's edge lines and other outlier lines, we can be confident that the length longest line detected is in fact the length of the edge of the cube.

We form lines by connecting the corner points in the form of a square and measure the mean square error between the length of those lines and the ground truth edge line.

$$MSE_{Square} = \sum_{i=1}^{4} (max\_len - d(c_i, c_{(i+1)\%4}))^2$$

, where $d(c_i, c_{(i+1)\%4})$ is the distance between corner i and corner $i+1$. The modulus 4 part is included to account for $d(c_4, c_{(4+1)\%4}) = d(c_4, c_1)$. These distances represent the hypothetical sides of the quadrilateral formed by the four points.

The idea is that the more different one or more of the hypothetical sides are from the true edge length, the more likely it is that a perfect square cannot be formed by those four points and hence our corner detection is incorrect. I have empirically determined that when

$$MSE_{Square} \geq 10000,$$

then the corner detection was incorrect, and we cannot proceed further with the algorithm. If this happens, an error message is displayed and the program halts.

H. Cube ROI Finding

The Region of Interest (ROI) is the region of the image where we estimate the cube is in, given the corner points we found. Assuming we can form a somewhat perfect square with the four corner points found, we use the starting point

$$p_o = (\min(x), \min(y)),$$

and width and height

$$W = \max(x) - \min(x) \text{ and } H = \max(y) - \min(y).$$

to crop the image and save it to a file. We repeat the entire process of cube detection for each of the six faces and generate six ROI files containing cropped images.

An example ROI is shown in (10). This example was chosen to illustrate that the cube detection algorithm is robust to really small affine transformations in the original image. That is, it tolerates some degree of error in how parallel the camera plane is to the image (cube) plane. Further changes in rotation will be explored in the experiments section.

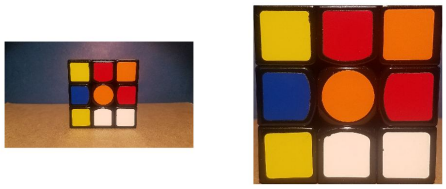

Figure 10

IV. Color Extraction

Once the six ROI's have been identified, the next step is to identify the nine colors on each one corresponding to the nine stickers of that face of the cube. In the following section, I will discuss each step of my process of color extraction.

A. Sticker ROI Finding

To find the locations of the stickers, [2] proposed using K-means and SLIC Superpixel Classification to segment the image. Then they create feature vectors using histograms and run a random forest classifier to classify superpixels as stickers or non-stickers. [1] proposed using dilation and adaptive thresholding on the Laplacian of Gaussian edge map to find the relevant contours of the stickers. [8] uses a calibrated camera to perform a 3D reconstruction of the cube.

Given that my Rubik's cube detector algorithm outputs a ROI, the simplest approach was to use the dimensions of the ROI to find smaller ROI's for the nine stickers rather than using these other approaches.

To split a cube ROI into 9 sticker ROI's, I cropped the image using the starting point

$$p_o = (0 + m \times w + off, 0 + l \times h + off),$$

where m and l are integers that vary from 0 to 2, hence generating $3 \times 3 = 9$ ROI's. I will explain off briefly. Variables w and h are a third of the width, W, and height, H, respectively, of the original ROI. That is,

$$w = ceiling\left(\frac{W}{3}\right) \text{ and } h = ceiling\left(\frac{H}{3}\right).$$

The actual width and height used for the cropping is:

$$w' = w - 2 \times \text{off} \text{ and } h' = h - 2 \times \text{off}.$$

The offset off was added to the starting coordinates and subtracted from the width and height in order to narrow in into the sticker and avoid the black edges on Rubik's cubes or any other background produced by cubes that were not parallel to the camera plane.

Consider the sticker in the bottom leftmost corner in (10). Figure 11 depicts the sticker ROI obtained without using an offset (left) and the sticker ROI obtained using an offset of 60 (right). Clearly there is a significant reduction of non-relevant colors in the ROI using an offset. An offset of 60 is used in my program. This number was determined experimentally.

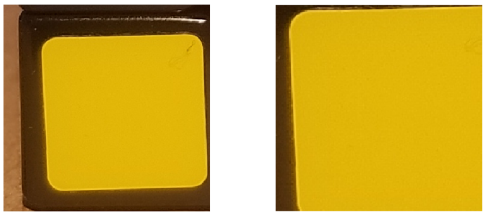

Figure 11

B. Random Sampling

Many methods use alternative color spaces rather than RGB. For instance [1] and [2] use the HSV color space and [8] uses the YUV color space. Although [1] found that using HSV is more robust than using RGB, for the sake of simplicity and because my algorithm cube detection algorithm is already non-robust, I chose to use RGB.

Furthermore, [1] and [8] use histograms to classify the colors whereas [2] uses a Support Machine Vector. My approach is to take the median of the RGB values of 50 random pixels in the sticker ROI and find the MSE between that median value and each of 6 "ground truth" RGB values for each of the six colors. To get a prediction, we simply have to minimize the MSE. That is, determine which color gave the smallest MSE for that median of RGB values.

The first step is to take 50 random points, obtain the RGB values, and store them into three separate arrays representing each of the three channels. I removed the values of the dark edges of the Rubik's cube by removing pixels whose RGB value's sum was less than 80. This value was found experimentally.

C. MSE Minimization

Next, I found the median of each array and found the mean squared error between the median and each of the six colors of interest. The MSE is given by:

$$MSE_{c,s} = (R_c - R_s)^2 + (G_c - G_s)^2 + (B_c - B_s)^2$$

where (R_c, G_c, B_c) is the RGB value of ground truth color c, and (R_s, G_s, B_s) is the RGB value composed of the median R, G, and B values of the 50 random points sampled from sticker s.

The last step is to generate the predicted color c for sticker s by finding the smallest $MSE_{c,s}$. That is,

$$c_{pred_s} = \{c_i \mid \min(MSE_{c,s}) = MSE_{c_i,s}\}$$

The ground truth RGB values for each color were calibrated manually. I use RGB values that seemed distinguishable enough for each color. Needless to say, this is very dependent on what Rubik's cube is used as well as the illumination of the image. Hence, this method of finding colors is not very robust. That being said, it is simple to implement and a good starting point for perhaps a more robust method.

While doing the calibration, I had issues with the program confusing yellow with orange and blue with green. Those issues have been fixed for the most part, although it is quite possible that under different illumination, different errors will show up. In fact, some of these errors did still show up, as discussed in the experiment section. On a more positive note, the algorithm is able to identify colors correctly in both scenarios presented in (4) by using a single set of ground truth color values. That is, it can identify colors under slightly different scale and illumination.

The output of the program is a text file containing a 2D array with the information about each face's colors. Figure 12 shows an example.

```
B B G R Y R G Y G
W Y R G O O G O B
W G W Y B Y R O Y
O W Y B R G B B B
R R O W G R Y W O
W G O B W W Y O R
```

Figure 12

EXPERIMENTS

All experiment figures can be found in Appendix A. The following sections explore a qualitative analysis of invariance of different parameters regarding cube recognition. Next, I perform a quantitative analysis of color classification. I conclude this section with a brief remark regarding the runtime of the algorithm.

A. Testing Scale Invariance

The first experiment I performed was to test for scale invariance. I ran the algorithm up to line detection (Hough transform) on the same cube configuration under 6 different scales. The results are shown in (13).

As expected, the algorithm is robust to very little variation in scale. Only in image 2 and image 3 were the four corners found. As shown in (14), if we increase the threshold to 0.3, we are able to detect all the corners in images 2 through 4. However, increasing the threshold further does not help. Image

1 always fails because the cube is so close to the camera that when rotating the image, two of the corners are out of sight.

B. Testing Translation Invariance

I tested for translation invariance by taking 6 pictures of the cube in different positions with respect to the camera. As shown in (15), the corners were found only in images 3 and 4. The reason for this is that images 1, 2, 5, and 6 all had at least one corner "hidden" by the rotation of the image in preprocessing.

A possible solution is to make the rotation "loose" rather than "crop". That is, we could make the rotated images contain the entire image rather than get cropped to fit the size of the original image. Unfortunately, since the code is fit to work on $2560x1640$ images, the result is that no lines are found (see (16)).

C. Testing Rotation Invariance

When checking for rotation invariance, I took one image in standard orientation (cube edges parallel to picture edges). Then I took four images in different orientations, by rotating the cube counterclockwise by increasingly larger amounts. The sixth picture is a picture from a higher angle, looking downwards at the cube. The results are shown in (17).

Interestingly, int the cube in standard form (first cube) only three corners were found. As expected, detection in the four counterclockwise rotations failed. This is because we did not account for rotation in our algorithm. Surprisingly, in the last image taken from a higher angle, all corners were detected correctly.

At first thought, this might indicate some robustness to variance in vertical rotation. However, this result might be better explained by the fact that a change in position of the camera caused a change in lighting. This better lighting caused the corners that were not found in the previous camera position to be found.

D. Testing Background Invariance

To experiment with background invariance, I again took 6 pictures. Image 1 had a light, uniform background. On images 2 and 3 I removed different halves of the uniform background. On image 4 the background was simply my desk and stuff on the back. I added more noise to the background in images 5 and 6.

The results are shown in (18). Since my program accounts for noise via thresholding, all four corners were found on the first four images. The program was able to detect most corners even in the images with most noise. When I adjusted the threshold to 0.18, better results were obtained (see (19)). Specifically, the noise and outliers were reduced, and more corners were found.

E. Testing Lighting Invariance

The last qualitative check was the check for lighting invariance. Image 1 had the standard white lighting I've been using for all other experimental tests. Images 2 and 3 have yellow lighting and have more shadows. Images 4-6 were taken in a dark room with only lighting from a tablet. Figure 20 shows the results.

Clearly, this algorithm is not robust to variation in lighting. The results get progressively worse the darker picture is. Good lighting is required for this algorithm to detect the Rubik's cube correctly.

F. Testing Color Classification

Inspired by the testing method employed by [1], I decided to perform a quantitative test of my algorithm for color classification. To do this, I took 18 images from previous tests that I knew passed the edge detection successfully with a threshold of 0.25. I randomly split them into 3 sets of 6 images, since the program expects to process 6 images at a time (although the order of the images is not important for the purpose of testing color detection). Next, I verified the results manually by comparing the actual colors with the predicted colors for each of the 18 faces show in the 18 pictures.

Then, using an adaptation of the metric employed by [1], I classified every prediction as either:

- Pass: correctly recognized color
- Failure: wrongly recognized color.

I am only interested in quantifying the number of correct/incorrect colors regardless of changes in scale, translation, rotation, background, or lighting. Hence, unlike [1], I only made one plot to measure the overall accuracy. Figure 21 shows the results. correctly.

As shown in the prediction, 157 of the predictions were correct while only 5 of them were incorrect. That is, the accuracy of the color prediction on correctly recognized Rubik's cube faces is 97%. A lot of this high percentage is due to the fact that many types of pictures are "filtered out" before we even get to the color detection algorithm, because we are unable to find a ROI to fit the cube face in those bad pictures. That being said, the accuracy found is still a good indication of the reliability of the color detection algorithm for the "standard" type of picture used for this program.

Furthermore, 4 of the incorrect predictions confused orange for red and the last one confused yellow for orange. It might just require calibration of the red, orange, and perhaps yellow "ground truth" RGB values to obtain 100% accuracy on these 18 images and perhaps an overall higher accuracy if further experiments with different images are ran.

G. Runtime

The algorithm's runtime is about 25 seconds. The first 20 seconds are spent finding the 6 cube faces and the last 5 seconds are used to classifying the 9 stickers of each face (54 stickers).

CONCLUSION

This Rubik's cube algorithm for cube recognition and color classification has a high accuracy for very specific types of pictures of Rubik's cubes. It is highly reliable when appropriate images are input. In addition, it is simple to implement and uses basic image processing techniques.

A. Future Work Improving Cube Detection

First and foremost, the cube detection can be optimized for robustness. Allowing a wider variety of images would make the

program more user-friendly and would speed up the photo acquisition process.

Second, the pre-processing done to the image before edge detection could be further improved. The algorithms and values used for blurring and thresholding the image were chosen experimentally and are fine-tuned for a small set of images with good lighting. Perhaps different values would help create a more robust solution.

Third, different image acquisition and pre-processing techniques might yield a different choice of edge detector. Perhaps the Canny or the Sobel edge detector are better for more general images. Further experimentation might yield better results.

Fourth, it is likely that using alternative approaches rather than the Hough transform combined with my own corner detection heuristic might improve not only the robustness but also the accuracy of the cube detection. For instance, [1] provided a straightforward way to find sticker regions. In addition, [2] suggested using machine learning for several steps of the cube and sticker detection process.

Furthermore, a big shortcoming of my custom algorithm is that a lot of parameters are "hard-coded" and hence will only work for images of certain dimensions (2560×1440). This was clear to me during the testing part of my experiment when I realized that changing the dimensions of the image, as in (16), results in no lines being found. This is definitely a problem to be solved in the future.

Perhaps more algorithmic approaches like RANSAC could be used to remove outliers rather than relying on a heuristic and specific values. Another option for improvement would be to use Harris corners for edge and corner detection.

B. *Future Work Improving Color Recognition*

Although color recognition yielded an accuracy of 97%, more can be done to improve this part of the algorithm. If we take into account the fact that improving the robustness of cube detection will very likely lower the accuracy for color recognition, it will be vital to improve color recognition after improving cube detection.

Instead of entirely replacing my approach by that used by either [1] or [2], I could apply some of the concepts explained by [1]. For instance, [1] used the gray world assumption as a color balancing technique to apply before color recognition. In essence, this technique involves finding a coefficient of adjustment for each channel. Another important insight to gain from [1] is the use of HSV rather than RGB.

Any combination of these techniques could help improve the accuracy my color recognition algorithm for a more varied input set.

C. *Future Work Reducing Runtime*

Reducing the runtime of the algorithm is important for real-time applications such as robots. In addition, if this program is to be used by more people, it is reasonable that they would not be satisfied by waiting 25 seconds just for the program to detect the colors of each face of the Rubik's cube.

A first approach would be to not plot intermediate results nor output the cropped images as files. A main reason I did this was my lack of experience with MATLAB. I did not know how to do what I was trying to do without plotting intermediate results or saving a few intermediate results to files.

Another way to reduce the runtime would be to improve the processing before edge detection. This would remove the need to remove many outliers and ease the process of finding ROI's.

D. *Closing Remarks*

The results of the experiment were favorable. I achieved an algorithm that is both simple to implement and predictably accurate. There is much room for improvement in terms of robustness and runtime. That being said, I am fairly satisfied with my results.

REFERENCES

[1] Le Thanh Hoang. Optimally Solving a Rubik's Cube Using Vision and Robotics. Imperial College London – Department of Computing, 2015.

[2] Jay Hack and Kevin Shutzberg. Rubiks Cube Localization, Face Detection, and Interactive Solving. Stanford University, 2015.

[3] Rafael C. Gonzalez and Richard E. Woods. Digital Image Processing. Pearson, Fourth Edition, 2018.

[4] J. M. White and G. D. Rohrer. Image Thresholding for Optical Character Recognition and Other Applications Requiring Character Extraction. J. Res. Develop., vol. 27, no. 4, July 1983.

[5] Pinaki Pratim Acharjya, Ritaban Das, and Dibyendu Ghoshal. Study and Comparison of Different Edge Detectors for Image Segmentation. Global Journal of Computer Science and Technology – Graphics & Vision, vol. 12, no. 13, 2012.

[6] Deepika Adlakha, Devender Adlakha, Rohit Tanwar. Analytical Comparison between Sobel and Prewitt Edge Detection Techniques. International Journal of Scientific & Engineering Research, vol. 7, iss. 1, January 2016.

[7] Richard O. Duda andPeter E. Hart. Use of the Hough Transform to Detect Lines and Curves in Pictures. Communications of the ACM, vol. 15, no. 1, January 1972.

[8] Wlodzimierz Kasprzak, Wojciech Szynkiewicz, and Lukasz Czajka. Rubik's Cube Reconstruction From Single View for Service Robots. Warsaw University of Technology – Institute of Control and Computational Eng., 2019.

APPENDIX A

Note: the t = x in the images refers to the threshold value.

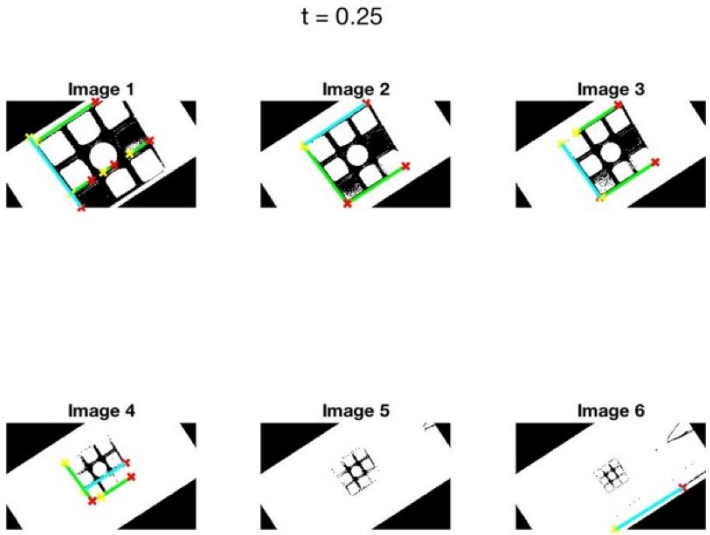

Figure 13 – Testing Scale Invariance

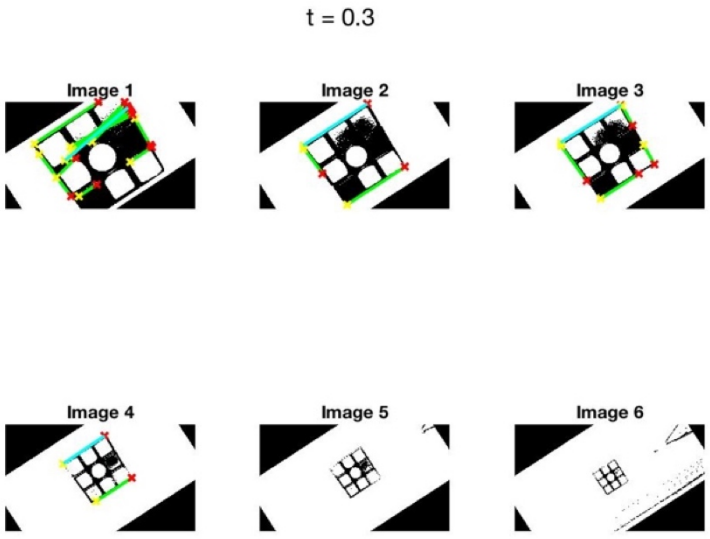

Figure 14 – Testing Scale Invariance

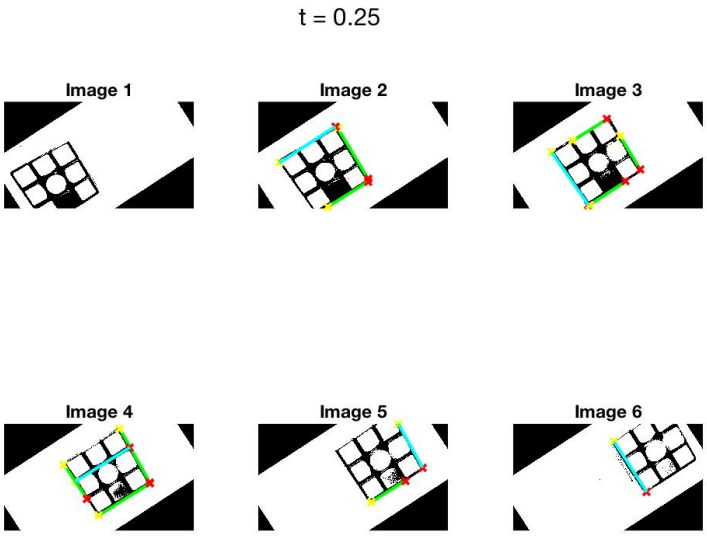

Figure 15 – Testing Translation Invariance

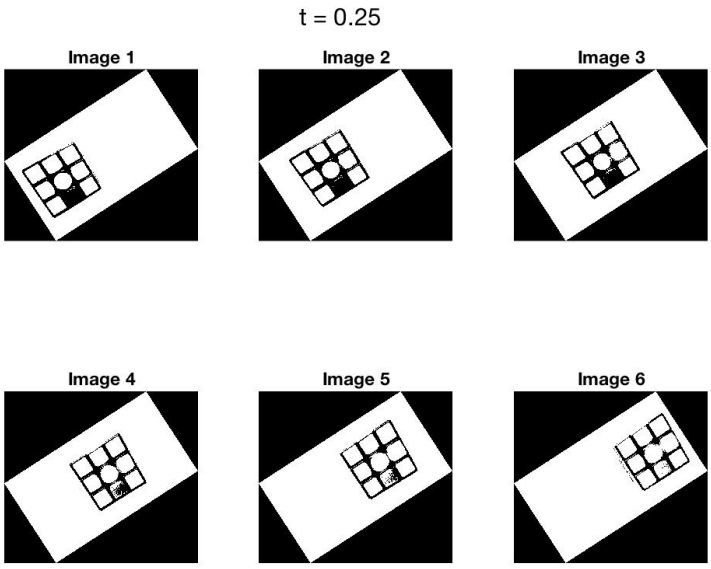

Figure 16 – Testing Translation Invariance

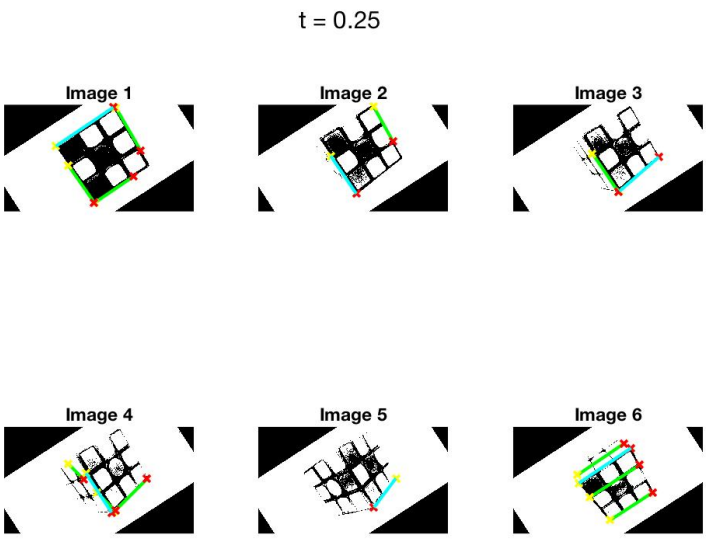

Figure 17 – Testing Rotation Invariance

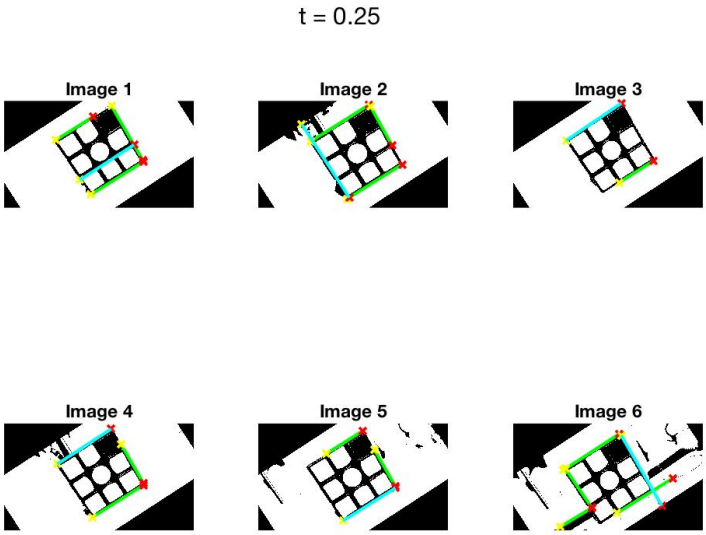

Figure 18 – Testing Background Invariance

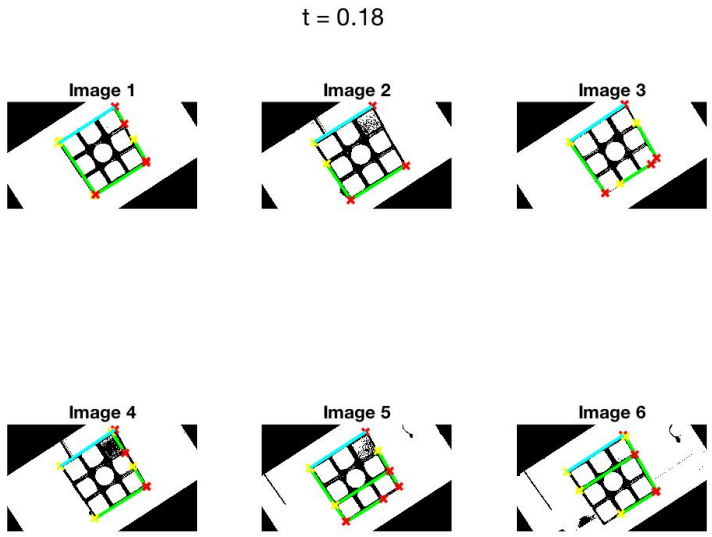

Figure 19 – Testing Background Invariance

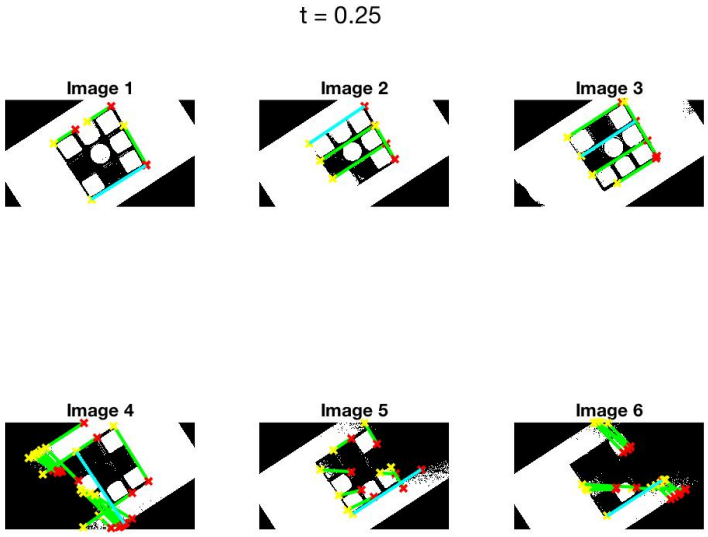

Figure 20 – Testing Lighting Invariance

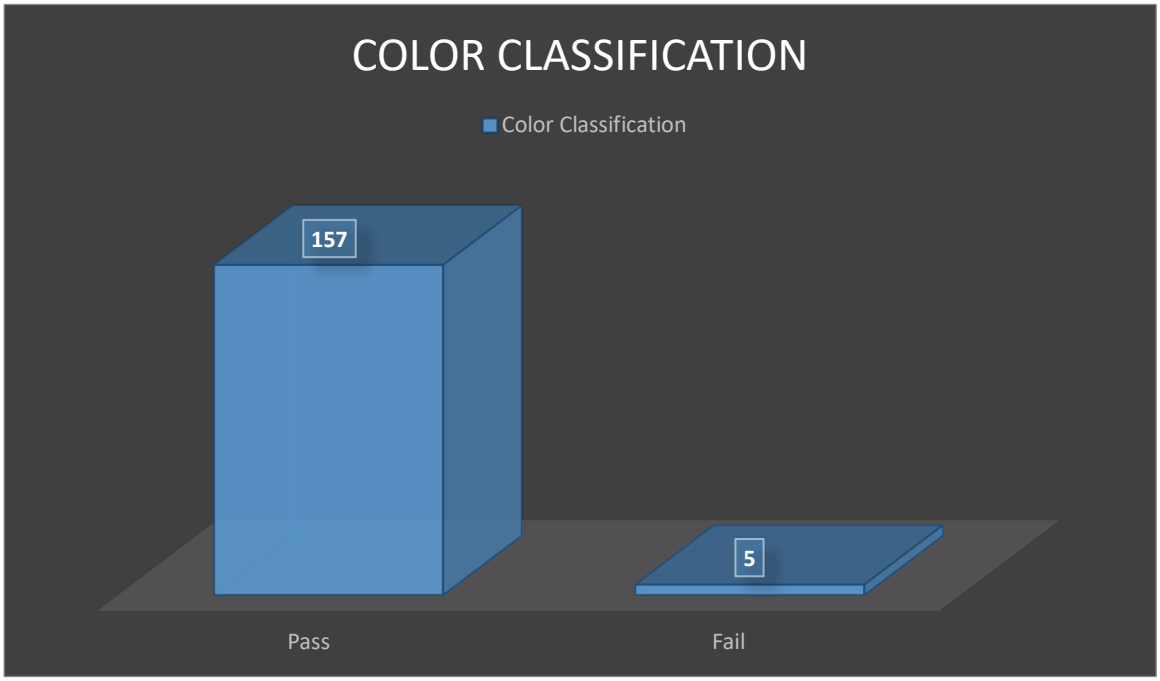

Figure 21 – Testing Color Classification

Appendix B

All of the necessary files, as well as expected results (for reference), and a detailed explanation of each file can be found at: https://github.com/brianquiroz216/Computer-Vision-Rubiks-Cube .

Instructions:

- To run this program, input images are necessary as well as a “CroppedImage” folder for intermediate outputs and an “Experiments” folder for outputting the results of the tests. All of these things are provided in the GitHub repository.
- This program does not work well when ran from the command line. Please run on the MATLAB application.
- Running the code “as is” will run all tests and a sample “proper run”. The outputs of the tests will show up under the “Experiments” folder and the output of the sample run will show up on the current directory as “stickers-set1.txt”.
- The results can then be compared to the expected results under “ExperimentsExpected” and “stickers-set1Expected.txt”, respectively.
- The code can be easily edited to simply run one test, a subset of tests, or no tests at all. This can all be done from the main “CVRubiksCube” function in “CVRubiksCube.m”.