**Rubik Cube –**

**Final Project : ICVB**



Submitted by :

Eldor Mouyal 207352451

Omer Dor 207828120

Saja Jbareen 207929761

**INTRODUCTION**

1. **Background**

Rubik’s Cube is a 3-D mechanical puzzle famous worldwide.

In this puzzle, we are presented a cube made of pieces , each consisting of different colors (typically up to six colors: white, red, blue, orange, green, and yellow).

The idea of the puzzle is to arrange all the pieces (with the different colors) of the cube in a way that each of the six faces consists only of pieces that match in color with each other.

Some individuals struggle to begin solving the cube, while others manage to solve part of it but become stuck halfway through.

Moreover, solvers aim to solve the cube or find a solution with minimal effort.

In our project, we provide the steps for solving Rubik’s Cube (using conventional notations, as specified below) with only two images of the cube as input.

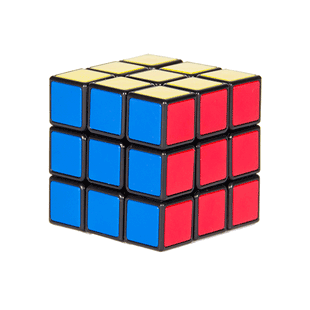


Figure 2 : Solved Rubik's Cube

Figure 1 : Unsolved Rubik's Cube

1. **Conventional notations of solution’s steps**

• Note that the notation containing a single letter refers to a clockwise rotation, while the notation containing a letter followed by a prime symbol (′) refers to a counterclockwise rotation.

The notations are divided into 3 types:

Face Rotations:

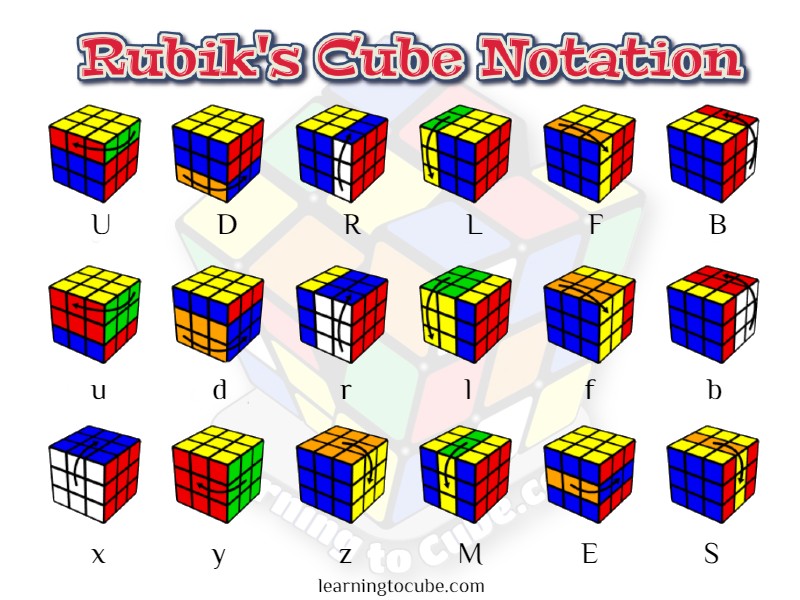
* F (Front) - the side currently facing the solver.
* B (Back) - the opposite side of the Front.
* U (Up) - the side above or on top of the Front side.
* D (Down) - the side opposite of the Top, underneath the Cube.
* L (Left) - the side directly to the left of the Front.
* R (Right) - the side directly to the right of the Front.

Slice Turns:

* M (Middle layer turn) - direction as an L turn, between R and L.
* E (Equatorial layer) - direction as a D turn, between U and D.
* S (Standing layer) - direction as an F turn, between F and B.

Whole Cube Rotations:

* X - direction along the x-axis.
* Y - direction along the y-axis.
* Z - direction along the z-axis.



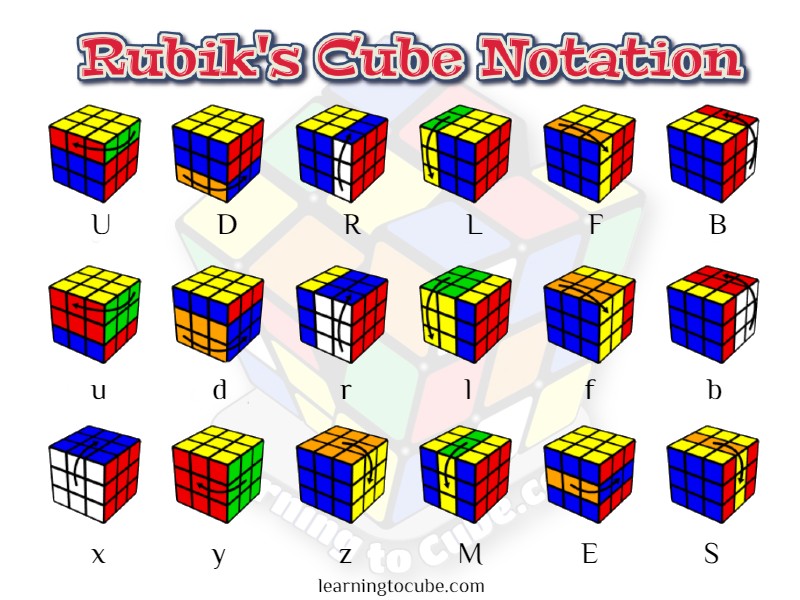


Figure 3: Notations for Rubik’s Cube solution (arrows indicating the direction of the rotations)

**Our Method**

The implementation includes 4 main steps :

1. Line Detection: This step involves locating contour lines that separate the squares on each face of the Rubik's Cube. These contour lines serve as the boundaries for individual squares.
2. Color Identification: This goal of this step is to determine the colors of the squares within each face. This involves analyzing the pixel values within each square region to identify the predominant color, which corresponds to the color of the sticker on the cube.
3. Input String Generation: Once the colors of all squares on each face are identified, we will generate a string representation of the Rubik's Cube configuration. This string serves as the input to the cube solver algorithm.
4. Cube Solver Execution: With the generated input string as input , we run the cube solver algorithm to determine the sequence of moves required to solve the Rubik's Cube. Finally, we will present the output, which typically includes the solution moves.

**1.Line Detection**

First, we read 2 images in color and resize them to be on 400X400 pixels .

For finding the edges of every image, we use Canny algorithm for edge detection. This algorithm returns an edge map, considering identification the edges present in an image while minimizing the detection of noise and maximizing the localization of true edges.

After that, we use Hough Line Transform to detect the contour lines of the cube.

* 1. **Performing Canny’s algorithm and Hough Line Transform**

When analyzing an image of the cube, we can categorize the contour lines into three groups based on their orientation relative to the y-axis: lines forming sharp angles (approximately close to 60 degrees), obtuse angles (approximately close to 110 degrees), and vertical angles (approximately close to 0 degrees). Consequently, we employ the Hough Line Transform separately to find each group.

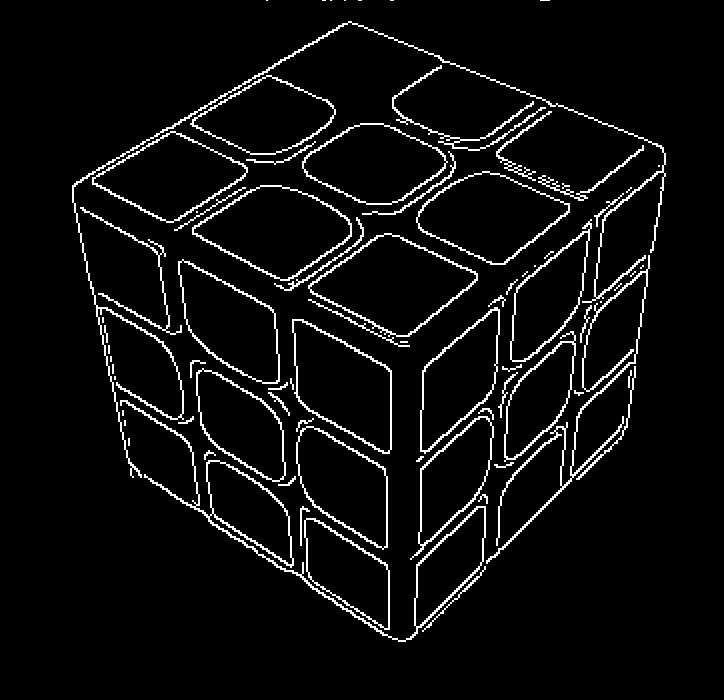


Figure 4.1: Lines with vertical angles (in orange)

Figure 4.3: Lines with obtuse angles (in orange)

Figure 4.2: Lines with sharp angles (in orange)

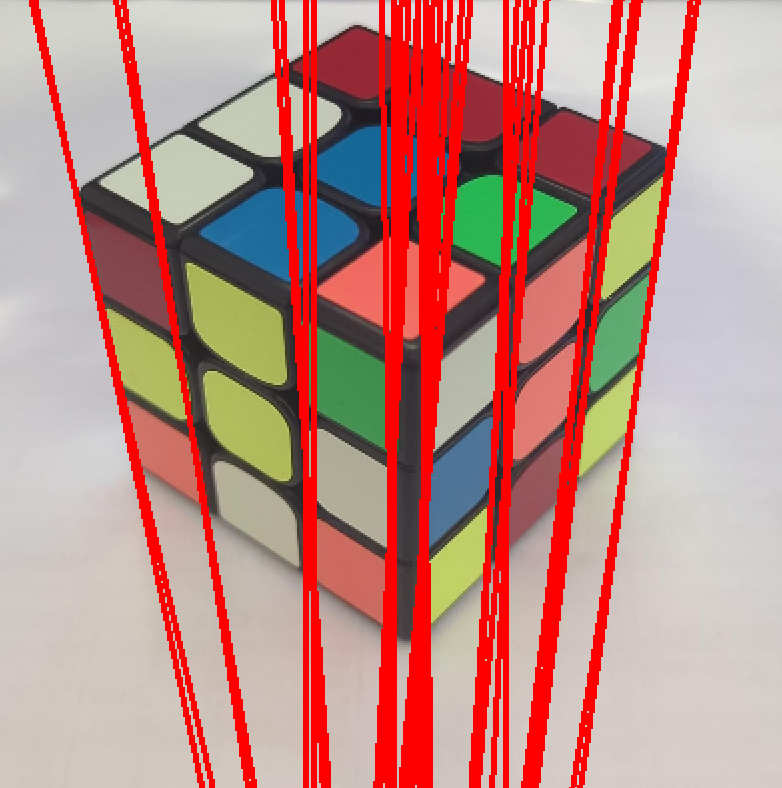
To address the variability among images and cubes, we've devised an algorithm that accepts an image and a classification (based on angle) as input. It then computes the optimal thresholds for the Canny algorithm and Hough Line Transform. This process involves using two thresholds (200 and 50) and calculating a weighted average, with more weight assigned to the lower threshold.

תמונה שמכילה קובייה הונגרית, פזל מכני, פזל, צעצוע

התיאור נוצר באופן אוטומטי

Figure 5: Input image

Figure 6: Edges detection after Canny’s algorithm, in this image we use threshold1=50 and threshold2=150



תמונה שמכילה אומנות, צבעוני, קובייה, עיצוב

התיאור נוצר באופן אוטומטיתמונה שמכילה צבעוני, פזל מכני, קובייה, אדום

התיאור נוצר באופן אוטומטי

Figure 7: Image after Hough Line Transform with vertical angles.

Figure 9: Image after Hough Line Transform with obtuse angles.

Figure 8: Image after Hough Line Transform with sharp angles.

* 1. **Performing Clustering on the lines from the Hough Line Transform**

After detecting a group of lines using the Hough Line Transform, we clustered these lines based on parameters determined later.

From each cluster, we selected a single representative contour line, which ultimately identified one contour line from the face of the cube.

Please note that our representation of a line consists of two parameters: rho (the distance from the origin to the line) and theta (the angle between the line, formed by the distance rho, and the x-axis).

We employed the same classification approach for both Canny’s algorithm and the Hough Line Transform to categorize angles during clustering.

Initially, we filtered the lines based on parameters determined by the classification.

Subsequently, we partitioned all lines generated by the Hough Line Transform into a specified number of bins (adjusted based on the classification).

From each bin, we selected one line according to the filtering rule.

In the group of vertical angles, we divided them into 10 bins and selected the line closest to the average theta from each bin.

For the group of sharp angles, we divided them into 80 bins, and for the obtuse angles, we divided them into 30 bins. In these cases, we chose the line closest to the average rho of each bin.

After applying the initial filtering, we proceed with another two filters, this time operating between the lines within the bins.

For the vertical angle group, we selected the line with the minimum rho from each group of crossing lines.

For the other two groups, we selected the line with the maximum rho from each group of crossing lines.

After applying the second filtering, we apply a third filtering step based on the distance between the lines.

This involves removing lines that are too close to each other and inserting side lines if they were not included in the final set of lines.

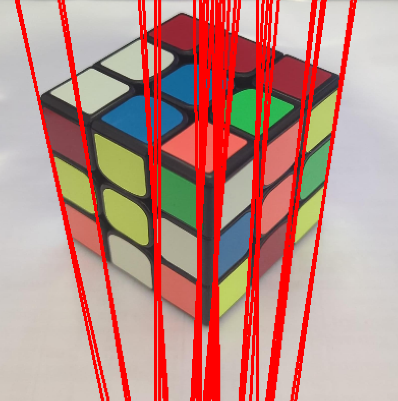
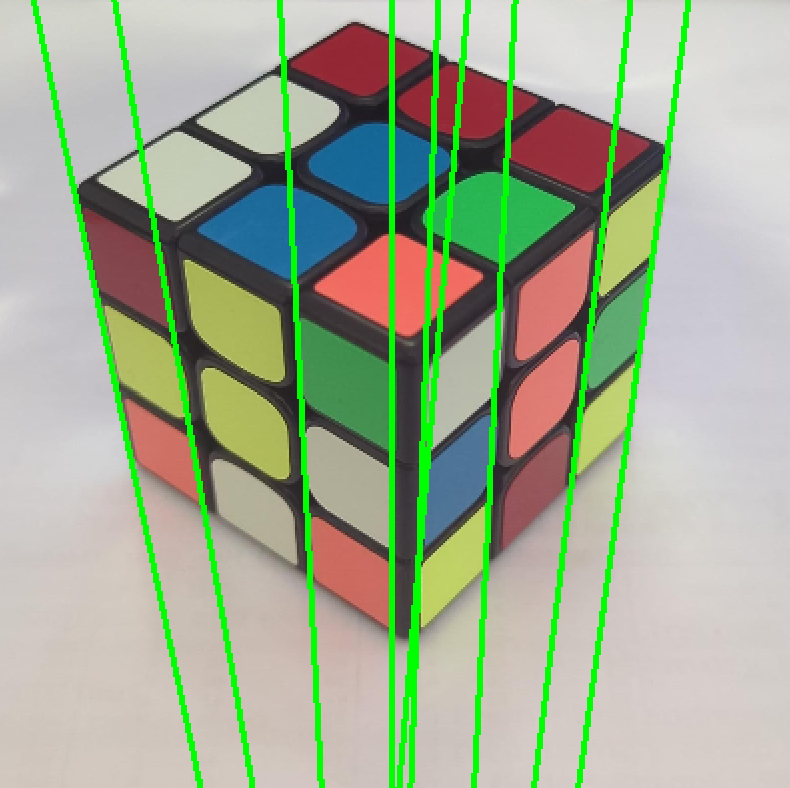


Figure 10.1: Image after 1st filtering for Vertical angles

Figure 7: Image after Hough Line Transform with vertical angles.

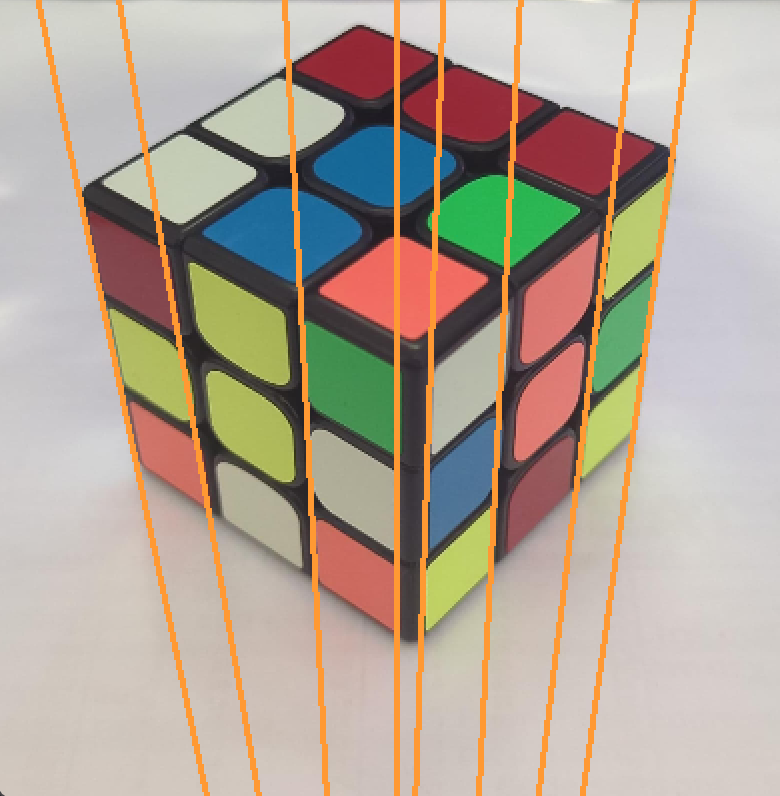
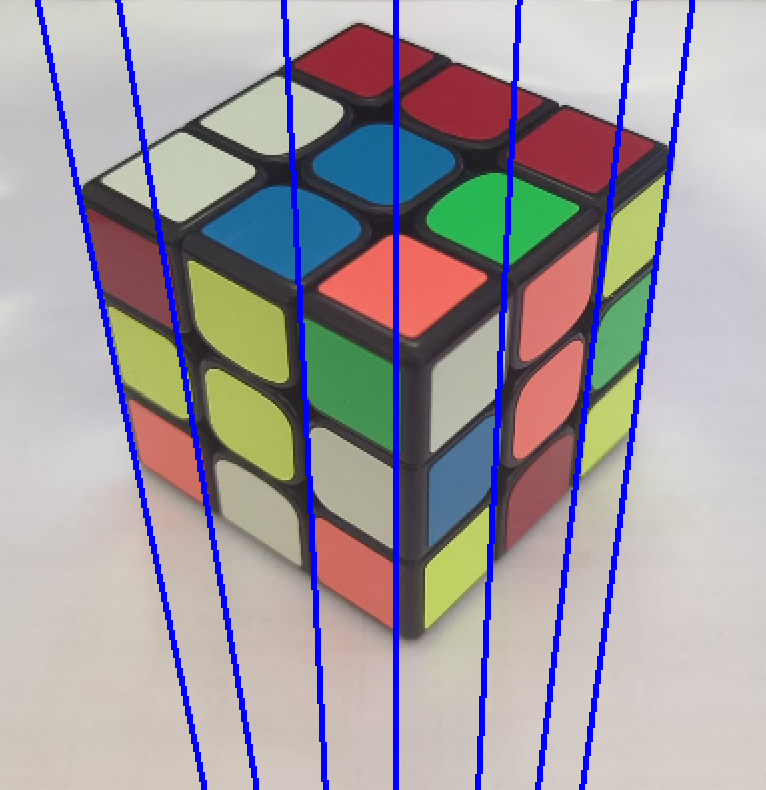
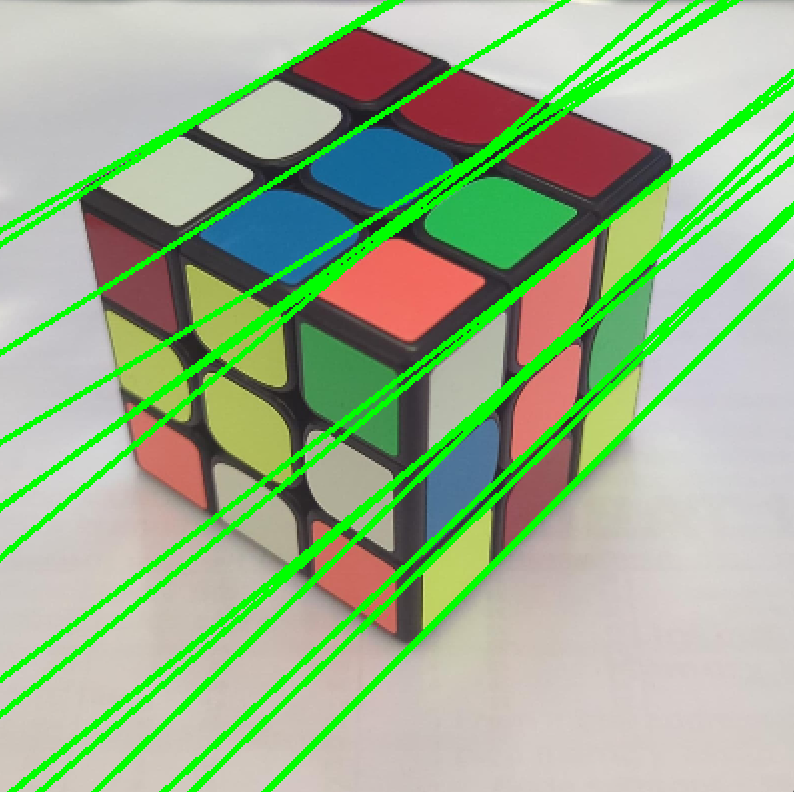


Figure 10.3: Image with the representing lines

Figure 10.2: Image after 2nd filtering for Vertical angles

תמונה שמכילה צבעוני, פזל מכני, קובייה, אדום

התיאור נוצר באופן אוטומטי

Figure 8: Image after Hough Line Transform with sharp angles.

Figure 11.1: Image after 1st filtering for Sharp angles

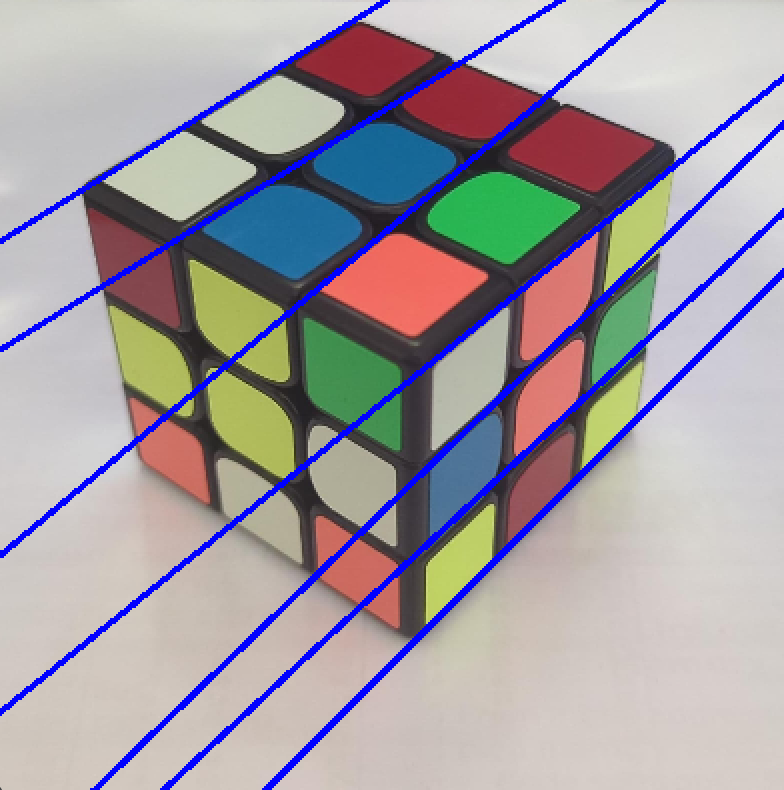
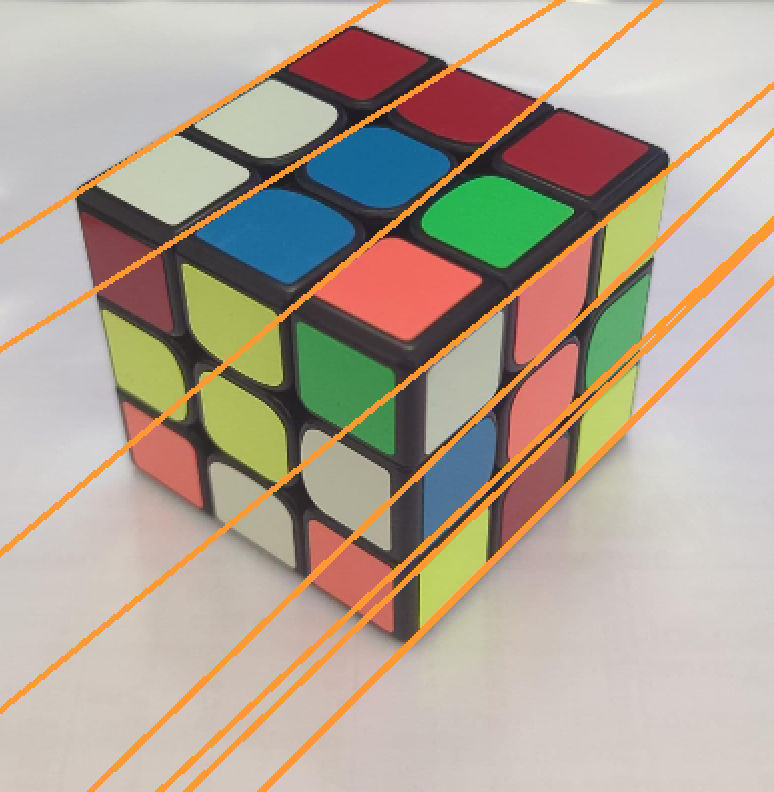
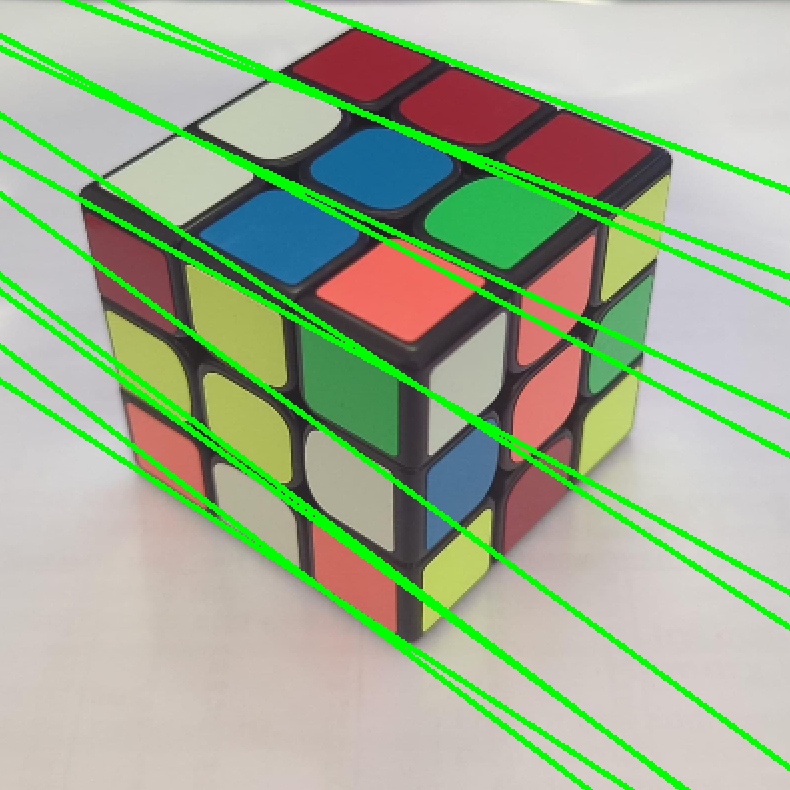


Figure 11.3: Image with the representing lines

Figure 11.2: Image after 2nd filtering for Sharp angles

תמונה שמכילה אומנות, צבעוני, קובייה, עיצוב

התיאור נוצר באופן אוטומטי

Figure 9: Image after Hough Line Transform with obtuse angles.

Figure 12.1: Image after 1st filtering for Obtuse angles

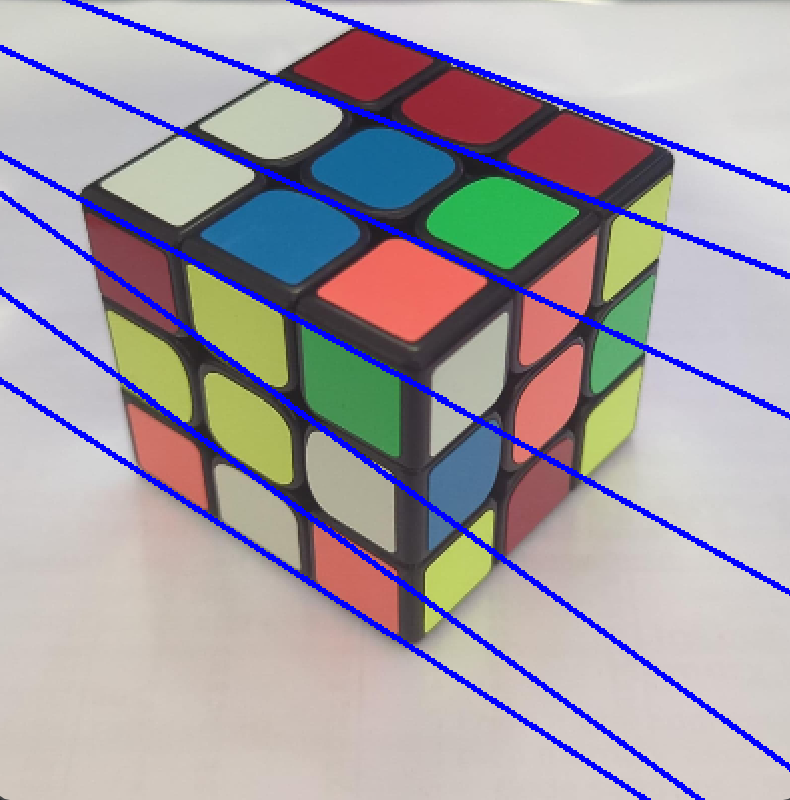
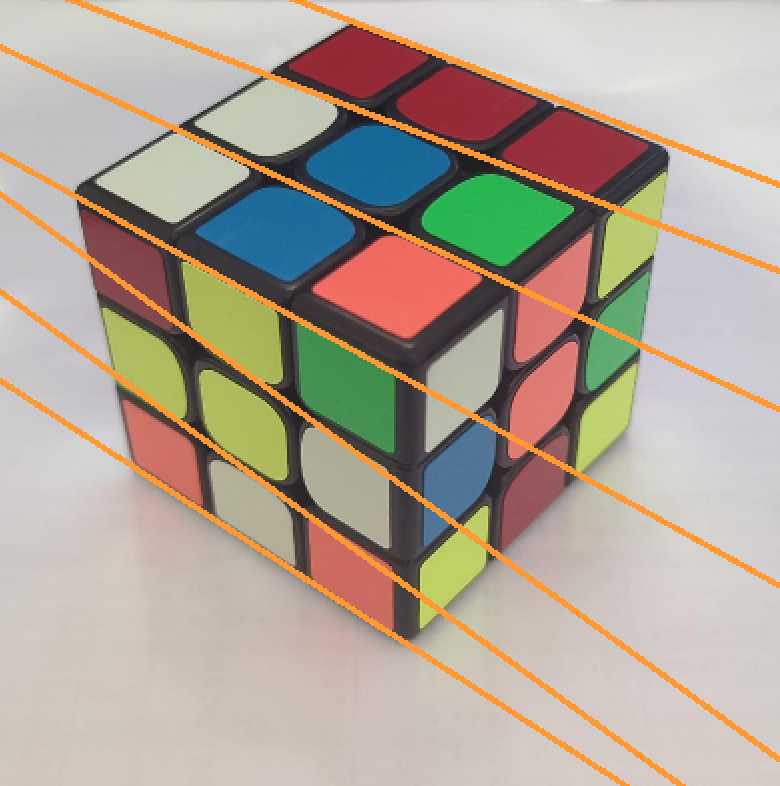


Figure 12.3: Image with the representing lines

Figure 12.2: Image after 2nd filtering for Obtuse angles

**2.Color Identification**

After obtaining the contour lines of the cube, the next step is to determine the color of each square on the cube's faces, which is essential for generating the input string for the cube solver.

To do this, we identify the crossing points of the contour lines, as they will indicate the squares on the cube's faces.

For every group of 4 points defining a square on the face (or a rhombus in the image), we determine the most common color among those 4 points.

Initially, we represent every possible color on the cube (yellow, red, green, blue, white, and orange) in the HSV color space (Hue, Saturation, and Value).

Following that, we obtain the Region of Interest (ROI) by performing a bitwise AND operation between a white polygon and our image, constrained within the boundaries defined by the 4 points.

We convert the Region of Interest (ROI) to the HSV color space and iterate over every possible color to find the color with the most pixels in the ROI.

Within the loop, we count the pixels in the ROI that fall within the range of each color.

The color with the highest pixel count is selected as the color of the square.

This process occurs for all the squares on the faces of the cubes.

Afterward, we compile a list containing all the colors present on the faces of the cube, following the order in which the faces were scanned.

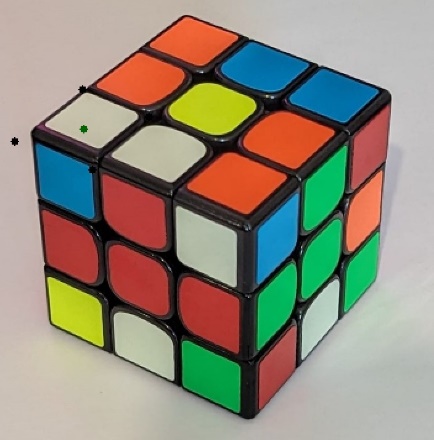
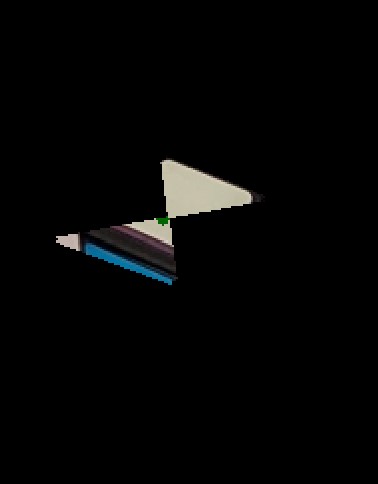


Figure 13: Image of the cube and the points defining the square of the face (in black)

Figure 14: Image of the ROI (Region of Interest) of a square from Figure 13

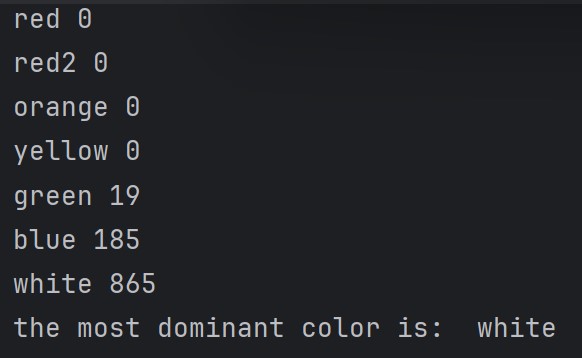


Figure 15: The pixel counts of the ROI from Figure 14 tabulated according to colors.

**3.Input String Generation**

When obtaining the colors of each square on the cube's faces, we construct a string that serves as input for the cube solver.

Each letter in the string corresponds to a direction for manipulating the cube: UP (U), DOWN (D), LEFT (L), RIGHT (R), FRONT (F), and BACK (B).

To establish the mapping, we refer to two images:

In the first image:

* The face oriented upwards becomes our UP face.
* The face oriented to the left becomes our RIGHT face.
* The face oriented to the right becomes our FRONT face.

In the second image:

* The face oriented upwards becomes our DOWN face.
* The face oriented to the left becomes our LEFT face.
* The face oriented to the right becomes our BACK face.

The string is constructed as follows: for each face, we identify the color of the middle square.

Then, by iterating through all squares on all faces and following the defined mappings, we determine the letters in the input string.

For instance, if the middle square of the UP face is red, whenever we encounter a red square during iteration, we append the letter U (for UP) to the string.

At the conclusion of this process, we will have generated a string that serves as input for the subsequent step: solving the cube.

UP face : will be BLUE.



RIGHT face : will be YELLOW.

FRONT face : will be PINK.

Figure ##: Image of the cube with explanation of the mapping defined above.

Images of the program in this phase : **image** of the cube divided of all the blue lines, **image** with the colors written on the screen of the input image above and **image** of the string created by the colors.

**4.Cube Solver Execution :**

We will utilize a cube solver that employs Kociemba’s Algorithm, a highly optimized method for solving Rubik’s Cube developed by Herbert Kociemba.

This algorithm can efficiently find solutions for most cube configurations within a reasonable number of moves.

Due to its efficiency and effectiveness, Kociemba’s Algorithm is widely adopted in Rubik's Cube-solving software and competitions.

This algorithm is built upon two phases:

1. This phase focuses on solving the cube's corners and edges independently.

It utilizes various techniques, including subgroup analysis and pattern databases, to efficiently find a solution within a large search space.

1. In this phase, the algorithm further refines the solution obtained in Phase 1 to solve the cube completely.

It focuses on manipulating the cube's remaining pieces to achieve the final solved state.

In our project, we utilize the Python package for this algorithm, known as 'kociemba'.

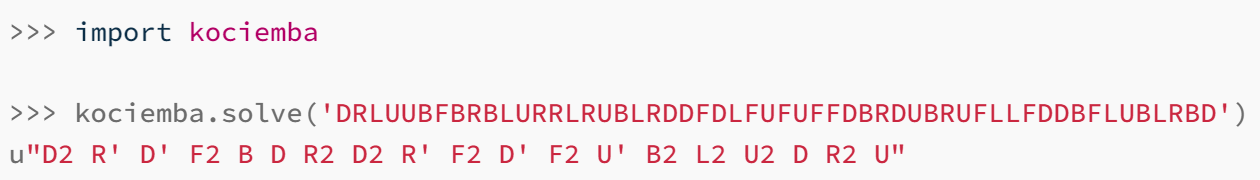


Figure ##: The Python code utilizes the 'kociemba' package, which involves importing the package and obtaining the steps to solve the cube based on an input string representing the cube.

**Results**

Our project can provide steps for solving a Rubik’s Cube using only 2 images that represent the cube.

These images can capture the cube in various configurations.

Here is an example of a cube represented by two appropriate images, along with the steps for solving the cube.

Additionally, images of the cube after executing each step are provided, illustrating the progression until the cube is solved.

Images of the program in this phase : **image** of the cube, **image** of the steps of the solution and **images** of cube after making every step.

**Conclusions**

During our project development, we observed that if the preprocessing steps are not executed properly, we encounter difficulties in detecting the contour lines of the cube and identifying the color of each square's middle on the cube's faces.

Without this crucial information, we are unable to generate an input string for the cube solver, which in turn affects our ability to solve the cube accurately.

We also observed that detection works best when applied to grayscale images.

This is because it proved challenging for the Canny Algorithm and the Hough Line Transform to find a consistent threshold that effectively detects contour lines in colored images.

An essential assumption we made for the project to function effectively is that the images must be provided in a square format.

We found that using non-square images made it more challenging for the Canny Algorithm to detect suitable edges and subsequently create lines using the Hough Line Transform.

Another critical assumption is that the images should have minimal shadows, as these can interfere with the Canny Algorithm's ability to detect accurate edges.

Furthermore, it's important to ensure that the images adhere to the following rule: if the contour lines of the cube are white, then the cube’s background cannot be white, and the same applies to all colors.

This is necessary because it can be challenging for the Hough Line Transform to detect lines when the contour color matches the background color.

To enhance our project, we plan to accommodate images in rectangular formats and address issues related to shadowing.

Furthermore, we aim to develop a method for processing colorful images, although grayscale mode is widely available in modern cameras.

Additionally, we intend to improve the project's visualization to enhance user-friendliness.

We can enhance the accuracy of the lines representing the faces of the cube to better approximate the actual lines of the cube's faces.

**Instructions for running the project**

1. It is necessary to install those Python packages :

* cv2.
* matplotlib.pyplot.
* numpy.
* sklearn.cluster.
* kociemba.

1. complete!

**References**

1. <https://en.wikipedia.org/wiki/Rubik%27s_Cube>
2. <https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/>
3. <https://learningtocube.com/rubiks-cube-notation-what-does-it-all-mean/>
4. <https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html>
5. <https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga46b4e588934f6c8dfd509cc6e0e4545a>
6. <https://kociemba.org/index.html>
7. https://en.wikipedia.org/wiki/K-means\_clustering
8. Presentations of the course Introduction to Computational and Biological Vision by Prof. Ohad Ben-Shahar : <https://moodle.bgu.ac.il/moodle/course/view.php?id=50161>