ME 101: EV3 2x2 Rubik's Cube Solver

Department of Mechanical and Mechatronics Engineering

A Report Prepared For:
The University of Waterloo

Prepared By:
Evan McIntyre, Tanay Mehta, Edam Jin, Jeong Choi
987 Your Street
150 University Ave W.
Waterloo, Ontario, N2N 2N2

April 5th, 2024

Table of Contents

Table of Figures

Table of Tables

Summary

The “EV3 2x2 Rubik’s Cuber Solver” project aimed to create a robot capable of solving a 2x2 cube using Lego EV3 interfacing. Motivated by the challenge and complexity presented by the Rubik’s cube, the group set out to create a user-friendly solution that fully automated this multi-step process.

Within the need analysis stage, the project objectives were established, focusing on aspects such as time efficiency, automation, and compact design. Engineering specifications were outlined to aid the design process, ensuring the robot’s performance and functionality aligned with the project objectives.

Through the design phase, various design concepts were considered, with the final selection focusing on a decision-making matrix that prioritized speed, accuracy, and complexity. The chosen design featured a gripper claw mechanism, color sensor, and pre-programmed algorithms to enhance the cube manipulation and solving process.

Within the mechanical design, a rotating baseplate and cube grabbing arm, supported by structural elements were utilized. Sensors were also positioned strategically throughout the robot for efficient use of scanning the cube, and detecting weather the cube was present within the robot apparatus. Fabrication processes were also employed to produce necessary components.

In conclusion, the project successfully met it’s objectives to some extent, demonstrating effective collaboration, problem solving, and technical skills. Recommendations for future improvements included altering the coding language used to enhance file I/O, as well as altering mechanical components to improve efficiency, and accuracy.

1.0 Need Analysis

1.1 Background Description

Rubik's cubes are very unique puzzle. With just 6 sides, and a several tiles on each side, there are millions of different combinations that can be created. Something interesting is that any given state of a 2x2 Rubik's cube, it can be solved in under 11 moves. These motivating factors gave our group a great challenge to design a robot capable of solving a 2x2 Rubik's cube using the Lego ev3 interfacing. With the help of online algorithms, this problem posed a great tactical challenge, that we believed was in the scope of the course.

1.2 Problem Objectives

Before stating the construction process, our group brainstormed several objectives we wanted to aim for, to ensure the project had achievable goals that were in the scope of the course. For example, one goal was we wanted the robot to fully solve a 2x2 Rubik's cube under 3 minutes We also wanted to fully automate the process, using as little human intervention as possible. Lastly, we wanted to make compact design, to enhance the transportation of the robot, as well as to provide a building challenge within space limitations.

1.3 Need Statement

Looking within the problem objectives, and the background description, the need statement was developed. The need statement incorporated a primary constraint and a primary function that we thought best fit the problem our Lego EV3 robot would be solving.

Need Statement: There is a need for a user-friendly 2x2 Rubik's cube solving robot, that has a fully automated solving process.

1.4 Engineering Specification

Requirements Specification:

No.	Characteristic	Relation	Value	Units	Verification Method	Comments
1	Mass of Design	<	25	lb	Test	The mass of the robot can be measured via scale
2	Speed to Solve	<	3	Minutes	Test	Do time trials and ensure all cube combinations can be solved under time limit
3	Aesthetics	>	5	1-10 Scale	Test	Rating in terms of aesthetics will be recorded amongst a survey from classmates. 1 being the lowest, most bad looking design, with 10 being the best-looking design.
4	Longevity of Fabricated Components	NA	NA	NA	Examination	Components will be made of sturdy materials to ensure they last over the term, and do not require future maintenance
5	Cost Effectiveness	<=	40	$	Analysis	Total expenditure will be calculated, and budgeted to meet requirements
6	Space Limitation	>=	0.5	m^3	Test	Measure the outer dimensions of the structure with a ruler, or measuring tape
7	User Interface	>	5	1-10 Scale	Test	Rating in terms of aesthetics will be recorded amongst a survey from classmates. 1 being hard to use, 10 being super simple to use
8	Ensure the Cube is Fully Solved	NA	NA	NA	Demonstration	Validation through checking all 6 sides of the cube for completion

Table 1: Engineering Specification of Design

Outlined within *Table 1* are 8 engineering requirement specifications. Some of the most notable ones include: ensuring the cube is fully solved (each side has only 1 color), the robot apparatus fits within 0.5 m^3 space, the robot itself weighs less than 25 lbs, the robot is aesthetically pleasing, and the cube is solved within a 3 minute time frame. Overall, establishing these specifications early within the project allowed for time optimization, as it was clear which constraints needed work. Also, by having the specifications within the table, it allowed everyone within the group to work towards the same goals that we all agreed upon.

2.0 Conceptual Design

2.1 Conceptual Designs Considered

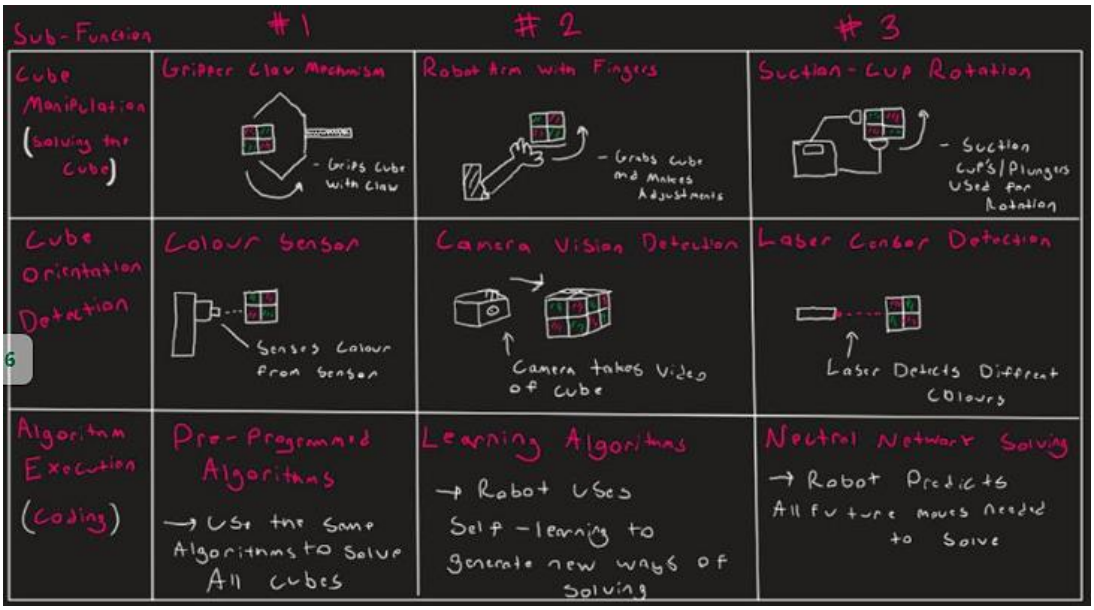

Figure 1: Morphological Chart of Sub-Functions

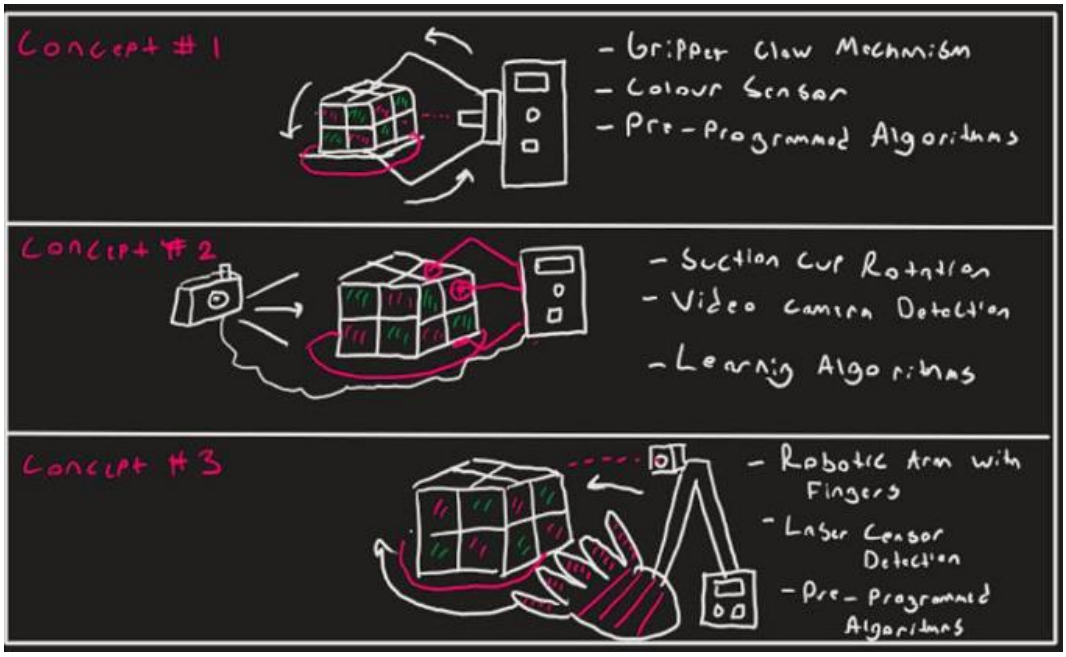

Figure 2: Conceptual Designs Formulated from Morphological Chart

Highlighted within *Figure 1* are different ideas generated based on sub-functions for the robot. For each sub-function, 3 ideas were generated. Within *Figure 2*, these ideas were combined into 3 different concepts. Concept 1 involves the gripper claw mechanism, the colour sensor and pre-programmed algorithms. Within concept 2, there is the implantation of a suction cup rotation device, with video camera detection and learning algorithms. Lastly, within concept 3, there is the implantation of a robotic hand - similar to a human hand, laser sensor detection, and pre-programmed algorithms.

2.2 Decision Making Matrix

Criteria	Concept 1 (Datum)	Concept 2	Concept 3
Speed of Solving	0	0	-1
Accuracy of Solving	0	+1	+1
Complexity of Design	0	-1	-1
Cost of Design	0	-1	-1
Totals:	0	-1	-2

Table 2: Decision Making Matrix

Within *Table 2*, the 3 concepts were compared on their speed of solving the cube, the accuracy of solving the cube, and the cost and complexity of the design. Looking at the totals, the datum used – concept 1, was shown to be the best option, as it had the largest total. In other words, the design of our robot was shifted in the direction of the gripper claw mechanism, color sensor, and the pre-programmed algorithms.

3.0 Mechanical Design

3.1 Layout

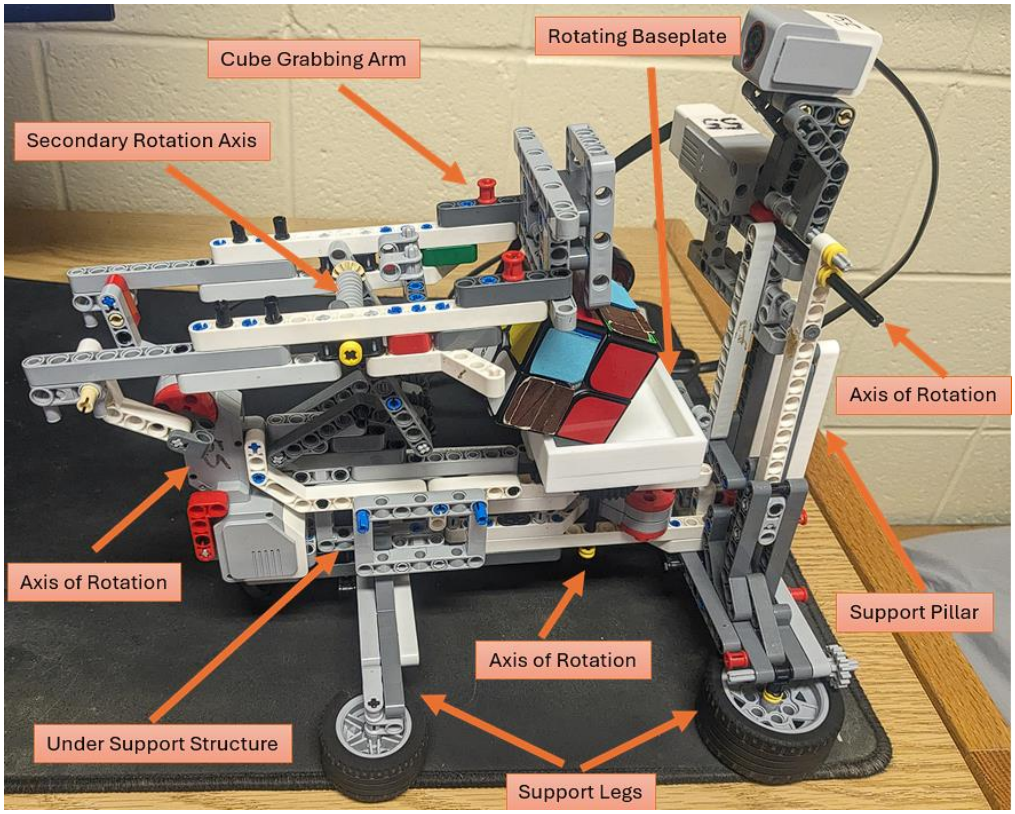

Figure 3: Layout of Mechanical Components

Shown in *figure 3* are the mechanical components used within our design. With 3 primary axes of rotation, all the mechanical motion can be generated to solve the cube. The key feature within the design is the cube grabbing arm. The arm has two primary actions: 1) holding the top of the cube, so the bottom rotating base can rotate the bottom row, 2) rotating the entire cube 90 degrees by pulling it backwards, then flipping the entire cube on it's side. The bottom of the robot structure has a framing system, supported by 4 Lego tires, which stops the apparatus from slipping on the table, and ensures that all the EV3 sensors and motors are properly supported. A support pillar was also utilized to suspend the color sensor above the rotating base, while still ensuring structural integrity.

Looking at the functions of the device, the rotating baseplate in combination with the cube grabbing arm allows for the cube to be rotated 90 degrees, the cube to be flipped on its side 90 degrees, and the bottom layer of the cube to be rotated 90 degrees. The retractable color sensor allows for all 6 sides of the cube to be scanned and implemented into the solving algorithm. Lastly, the ultrasonic distance sensor allows for the code to be executed on command, when the cube is placed within the apparatus.

3.2 Sensors

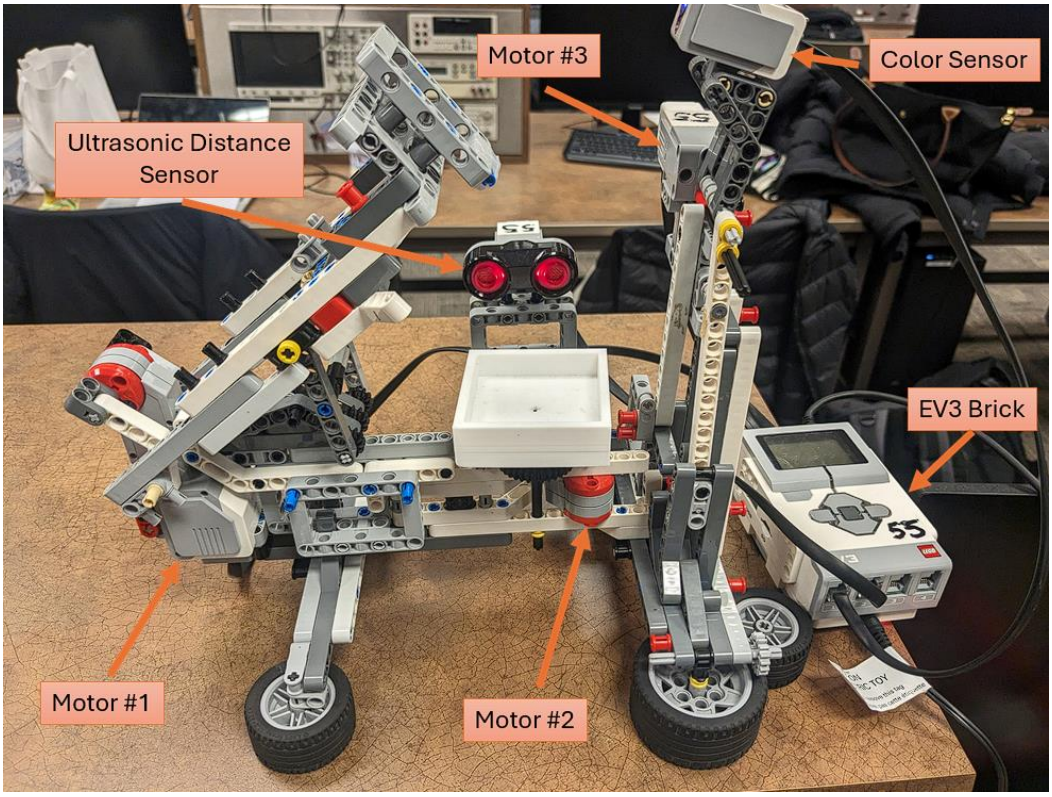

Figure 4: Sensor Configuration on Robot

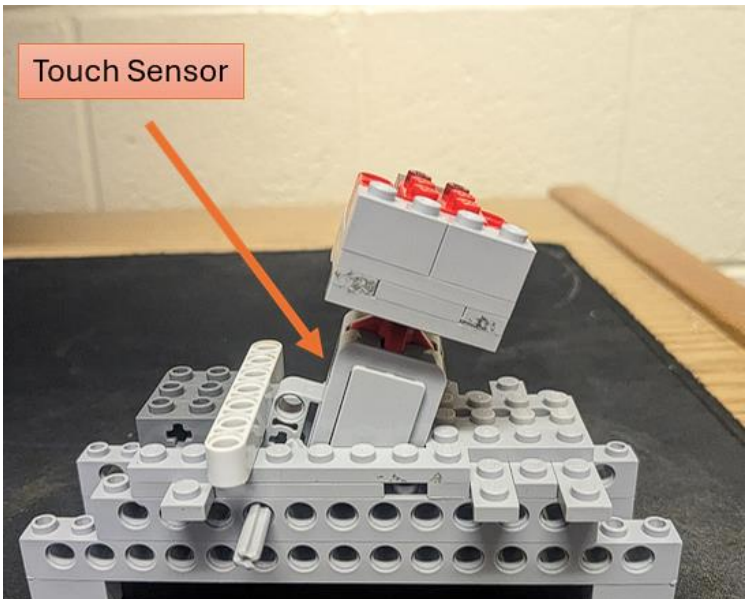

Figure 5: Touch Sensor Mechanism

Shown within *Figure 4* and *Figure 5* is the placement of all the sensors utilized within the robot. The robot makes use of 3 different motors - generating mechanical motion for the grabbing arm, rotation of the baseplate, and rotation of the color sensor to scan all sides of the cube. An ultrasonic distance sensor is also used to detect when the cube is placed within the apparatus. In other words, the robot won't start the solving process until a cube is detected. The color is present to scan all 6 sides of the cube. Being on a rotational axis, it allows the sensor to scan the cube, then retract back to it's upward state so the grabber arm can operate without interference. A touch sensor emergency stop button was also created to stop the solving process upon activation. The button consists of the touch sensor on an angle, within a Lego base structure.

3.3 Fabrication

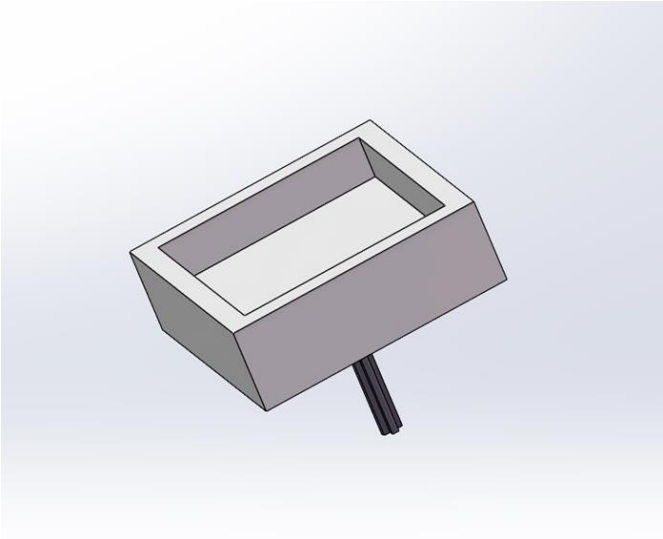

Figure 6: Rotating Base Plate

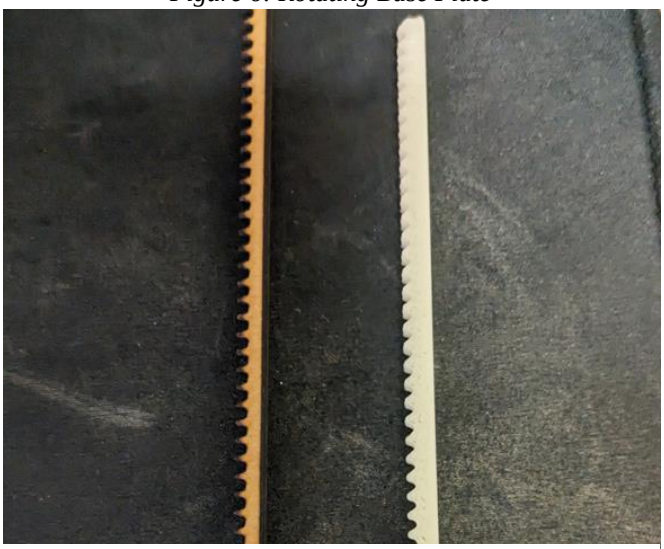

Figure 7: Gear Track for Moving Cube

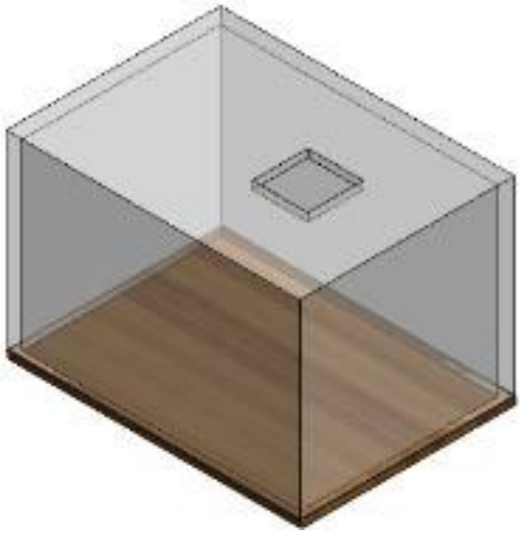

Figure 8: Enclosing Box and Baseplate

Figure 6 outlines the cad model used for 3d-printing the baseplate. The bottom of the piece utilizes a Lego axel, so the part can easily be rotated by the EV3 Motors. Within *Figure 7* are 2 prototype parts created for making a system that lowers the cube into the robot apparatus. Due to time constraints, these fabrication parts were not utilized within the final state of the robot, as the focus was more shifted towards function, rather than aesthetics. *Figure 8* shows the SOLIDWORKS file of the enclosing box made from acrylic, and the wooden baseplate. The box was designed so that the cube can be placed into the square on top and return in it's fully solved state. Again, due to time constraints, these parts were not utilized in the final design, as the cube elevator posed too many additive challenges, for just aesthetics, and not benefiting the function of the robot (solve the Rubik's cube).

4.0 Verification of Design

Requirements Specification:

No.	Characteristic	Relation	Value	Units	Verification Method	Comments
1	Mass of Design	<	25	lb	Test	The mass of the robot can be measured via scale
2	Speed to Solve	<	3	Minutes	Test	Do time trials and ensure all cube combinations can be solved under time limit
3	Aesthetics	>	5	1-10 Scale	Test	Rating in terms of aesthetics will be recorded amongst a survey from classmates. 1 being the lowest, most bad looking design, with 10 being the best-looking design.
4	Longevity of Fabricated Components	NA	NA	NA	Examination	Components will be made of sturdy materials to ensure they last over the term, and do not require future maintenance
5	Cost Effectiveness	<=	40	$	Analysis	Total expenditure will be calculated, and budgeted to meet requirements
6	Space Limitation	>=	0.5	m^3	Test	Measure the outer dimensions of the structure with a ruler, or measuring tape
7	User Interface	>	5	1-10 Scale	Test	Rating in terms of aesthetics will be recorded amongst a survey from classmates. 1 being hard to use, 10 being super simple to use
8	Ensure the Cube is Fully Solved	NA	NA	NA	Demonstration	Validation through checking all 6 sides of the cube for completion

Figure 9: Final Verification of Design

4.1 Assessments

Figure 9 highlights the final version of the verification of design chart. Looking at the mass of design, a scale was used to identify the mass of the robot with the EV3 brick to be 22.1 lbs. In terms of speed to solve, 15 time trials were completed, with the average speed being 2:30 seconds for each solve attempt. It's important to note that all attempts that were considered a failure (cube was not fully solved at the end of algorithm) were not recorded. Also, none of the trials exceeded 3 minutes. Looking at aesthetics of the robot and the user interface, surveys were recorded on demo day from 5 classmates, the average consensus rated the aesthetics a 8/10, and the user interface a 7/10. In terms of longevity of the fabricated components, all components manufactured over the project held up to all requirements, not requiring any further maintenance. With cost-effectiveness, the money spent including the fabrication of parts, and the cost of the Rubik's cube was $28 CAD. Using a meter stick, the robot apparatus, including the EV3 brick was able to fit within a 0.5 m^3 box. Looking at if the cube was fully solved, assuming the correct values were recorded by the colour sensor, the algorithm was successful in generating a solution, and physically solving the cube by demonstration.

Looking at the overall performance of the robot, all 8 specifications were met to some degree – in most cases, fully being met. The robot was successful in solving the 2x2 cube within the 3-minute time frame, assuming the correct colour values were read by the colour sensor. By establishing the verification of design chart early in the project, we were able to ensure that the majority of the specifications were met to the fullest extent possible.

5.0 Software Design and Implementation

5.1 Software Explanation

Figure 10 outlines the general overview of the code. Upon starting, the user is required to press the Enter Button to begin the series of operations. An ultrasonic sensor detects if the cube has been placed, a while loop with a condition distance >10cm. Once the cube has been placed in the tray the function inspectCube is carried out. This function has been mentioned in greater detail in

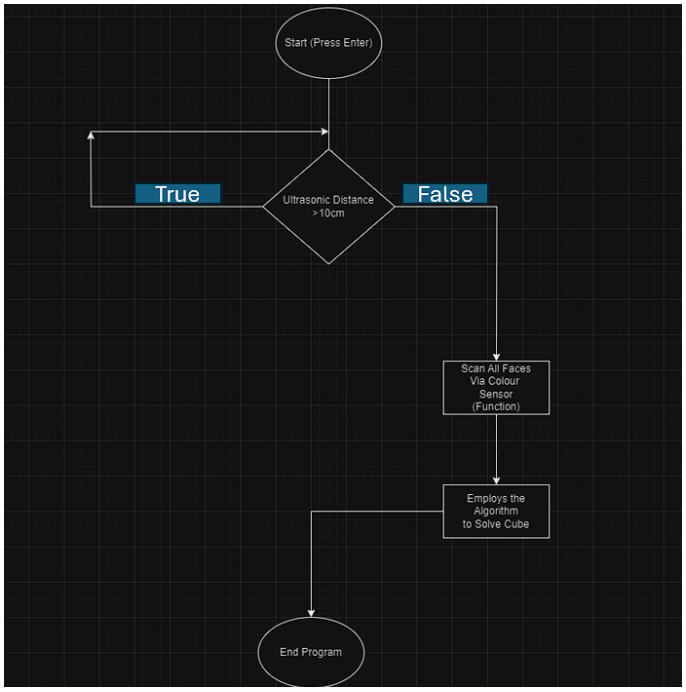

Figure 10: Overview of Code Flow Chart

Figure 11. Once the entire cube has been scanned, a text file containing numbers which depict the colour of tile, in a specific order, is sent to the computer. On the computer an advanced python algorithm is run [1]. A text file containing a certain set of moves to perform is inputted to the EV3 Brick. A solving function then reads the text file, performing a corresponding set of moves in the order in which they appear. This function has been elaborated on in *Figure 12*. A celebratory spin is performed once the cube has been solved, and the user can then press any button to end the program.

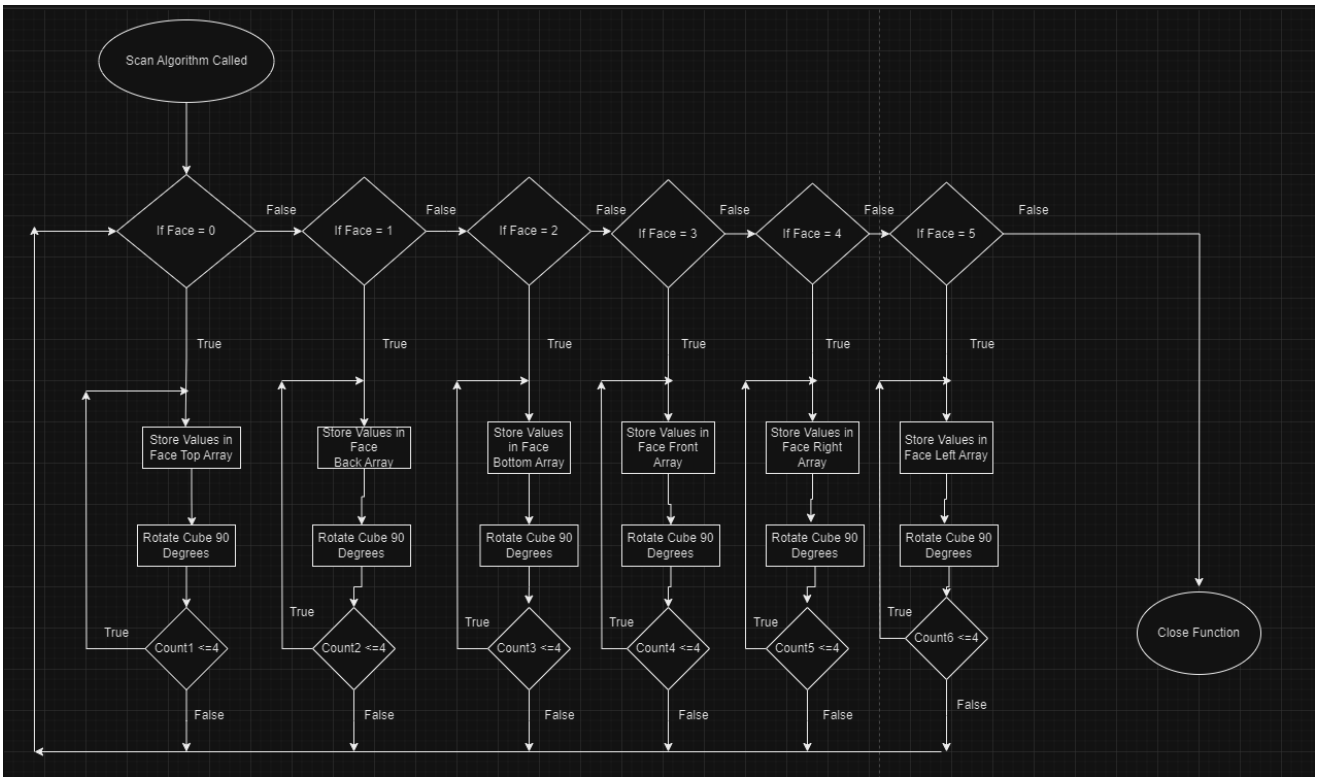

Figure 11: Solving Algorithm Flow Chart

Figure 10 is an overview of the inspectCube function. This function has been reference from a prior GitHub project, and modified for our use[2]. Once the function has been called, a for loop that runs 6 times is activated. This for loop controls each face of the cube, with each increment representing a different face. In this for loop is a nested for loop, controlling the rows and columns, with row = 0; row < 2 and col = 0; col < 2. Depending on which face the loop is currently on, colours are stored in a corresponding integer array at an index of row and col. A mix of sub-functions within each loop such as flipCube, turnCW, and turnCCW ensure each tile of the cube has been accurately read and a new face is being scanned each time. Below is an example of an array that would store the colours.

$$\text{(integer) faceTop} = \begin{bmatrix} 1 & 2 \\ 5 & 4 \end{bmatrix}, \text{ signifying } \begin{matrix} black & blue \\ red & yellow \end{matrix}$$

Here faceTop symbolises the top face of the cube relative to when the cube was first place in the tray. This would mean that the first face to be scanned would be the top face and consequently faceTop would be the first array to be filled.

Once the primary for loop has run six times, the Rubik's cube is returned to its original configuration and the function ends.

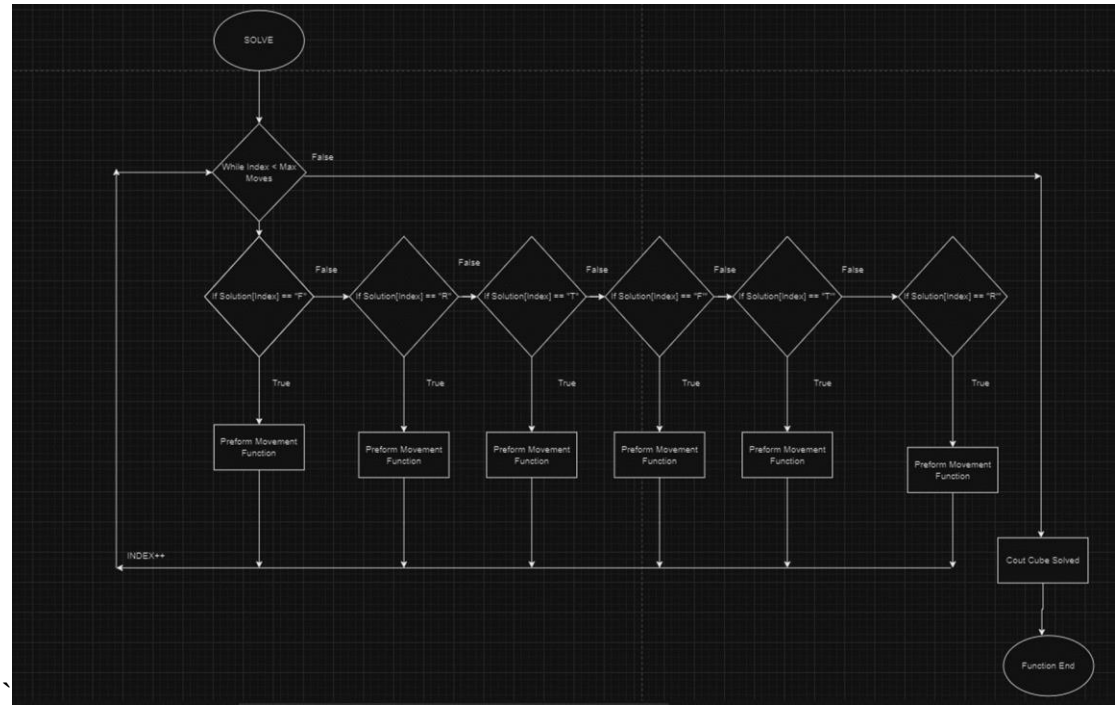

Figure 12: Solving Algorithm Flow Chart

Figure 12 depicts the solving algorithm implemented. Once a text file has been inputted into the EV3 brick, contents of the file are then stored in a string array. The solving function calls this string array. This function compromises of multiple if-else statements, with each representing a different move. A 2x2x2 Rubik's Cube can be optimally solved in 11 moves or less, the moves being {F, F', F2, R, R', R2, U, U', U2}. The 11 moves is stores as a constant max_moves. Based on the move present in the solution array, a combination of rotations and flips are performed. The entirety of if-else statements are encompassed in a for loop which runs a maximum of 11 times. A counter is set up to count and then display the number of moves it took to solve the cube, which is then returned and displayed on the screen through the main function.

Updated Task List

- Turn on program, scan all colours of all 6 faces on cube with colour sensor
- Utilize ultrasonic distance sensor to detect when cube is placed
- Flip cube 90 degrees
- Rotate bottom of cube 90 degrees
- Rotates entire cube 90 degrees
- Utilize solving algorithms to generate solving solution

Figure 13: Updated Task List

Function Name	Function Type	Parameters	Description	Written By
PIDController	Void	Int target Int motor	A proportional integral derivative system to ensure near perfect rotations of the bracket and cube.	Jeong Choi
flipCube	Void	Int flips	Flips the cube around z-axis, with number of flips indicated by ‘flips’	Tanay Mehta
turnCubeCW turnCubeCCW	Void	Int rotations	Rotates the cube clockwise and counterclockwise respectively, with number of rotations indicated by ‘rotations’	Evan Mcintrye
turnCW turnCCW	Void	Int rotations	Rotates only the bottom half of the cube clockwise and counterclockwise respectively, with number of rotations indicated by ‘rotations’	Edam Jin
moveColourSensor	Void	Bool isMoveForward	Moves the colour sensor up or down to scan the colour, with up indicated by false and down true.	Tanay Mehta
Solve	Int	NA	Performs the aforementioned moves which have been acquired from the solution array.	Jeong Choi

Table 3: Non-Trivial Functions Explanation

Data Storage:

The bulk of the data comprises of the colour values of each tile of the Rubik’s Cube. Since a 2x2x2 Rubik’s Cube has 4 colours on each face and 6 distinct colours, this gives a total of 24 elements to store. These values are stored as integers in six different 2x2 arrays, each one corresponding to a certain side of the cube.

A 1D string array of size 11(max moves) stores the solution sent from the computer, containing the moves to be performed.

Software Decisions:

Since RobotC does not directly support file I/O, a separate code had to be used to facilitate the transfer of the scrambled file to and the solution file from the computer[3]. While this did make the overall process quite lengthy as files had to be sent, saved and redownloaded it was the only way to move forward.

A significant trade-off that was mutually agreed upon was trading speed for accuracy. To ensure near 100% accuracy for flipping and turning the cube, motor speeds were kept very low. For the rotation of the bracket, a combination of gears was used to ensure more accurate turning.

A code for a PID was also implemented, used to control motors accurately. It sets up constants like MAX_TIME of 0 and MAX_POWER 100 to limit control time and motor power. The controller's coefficients (kpCoeff, kiCoeff, kdCoeff) determine how much each part (proportional, integral, derivative) affects the motor's behaviour. The kp, ki and kd values were adjusted to keep the time below the constraint of 3 minutes in total, as well as managing to align the cube perfectly for the next movements.

Initially, only Proportional and Integral was used, however during the cube rotation, the overshooting became a major issue. The base motor movement was also quite imperfect, having intervals of pausing motion.

Using the PID formula, it calculates the proportional, integral, and derivative components of the control output. It then combines these to get the total motor power needed to reduce the error. If this total power exceeds the maximum allowed, it adjusts it to stay within limits.

Finally, it sets the motor's power accordingly and introduces a delay before the next iteration. This loop keeps adjusting the motor's output based on the error, helping the system reach and maintain the desired position accurately.

The creation of 6 distinct 2x2 arrays meant easier storing and visualising of the code but resulted in writing individual if-else statements for multiple sections of the code.

Testing and Problems Encountered:

Each function was tested individually in the main program, starting with the sub-functions such as flipping and turning the cube. To debug, the speed of motor and degree of rotation was changed to ensure 100% accuracy.

The EV3 colour sensor is extremely inaccurate. To find which colours were the hardest to detect, each colour was printed on the screen the minute it was detected. This helped in figuring out that green and blue were the most irregular. To fix this issue green was replaced with brown paper on the Rubik’s Cube. This did not help much as brown was detected as black. Since this was a sensor inaccuracy and not a code error, the cube was placed into the robot with one face pre-solved and the incorrect values were manually changed in the text file. This allowed for debugging the rest of the code.

Since a group member had prior experience with professional Rubik's Cube solving, he was able to visualize each move and write the corresponding code for it, thus creating the solve function.

6.0 Project Management

6.1 Work Break Down

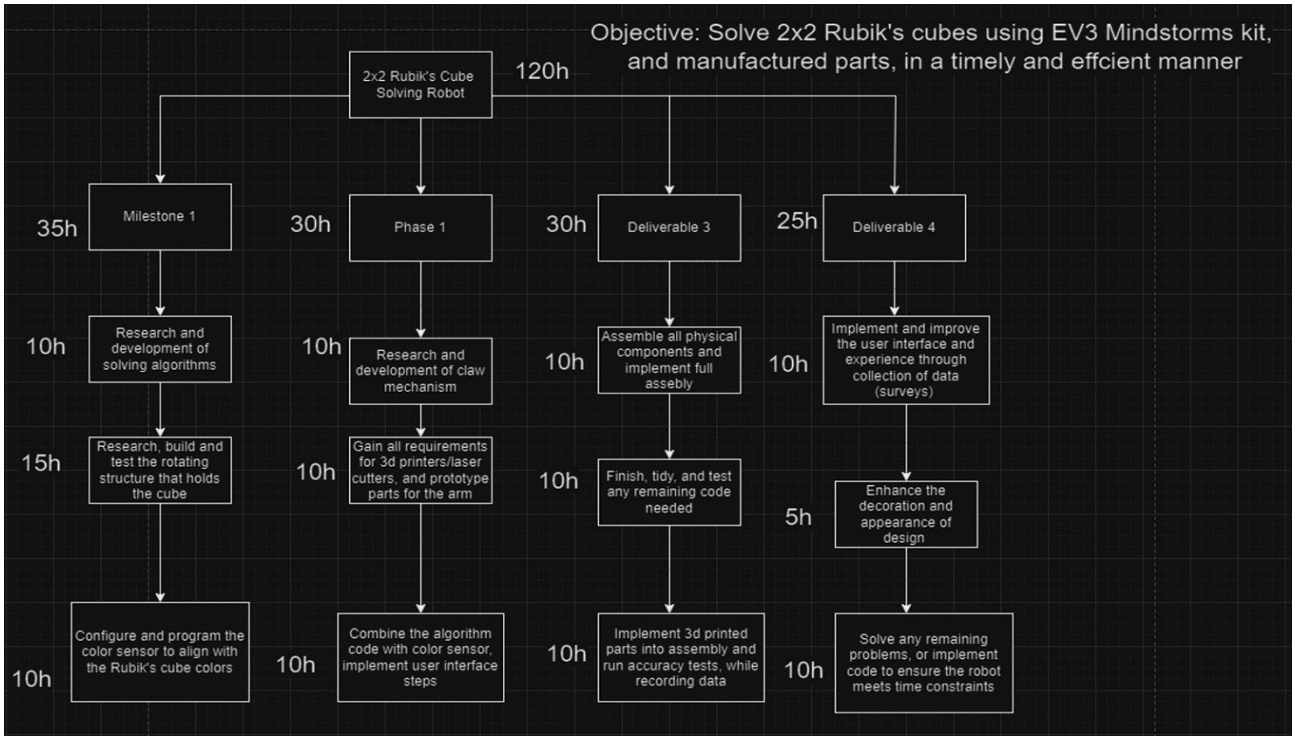

Figure 14: Work Break-Down Chart

Shown within *Figure 14* is the process used to distribute work over the course project. For each milestone, there are several tasks with hour requirements assigned accordingly. Having several tasks for each milestone, allowed for work to be split among all 4 group members, to ensure efficiency working towards deadlines. Some tasks are longer than others, so when group members finished their assigned task, they would help others with their outstanding task. It is important to note that some of the hour amounts shown in *Figure 13* were inaccurate estimates, as the code's complexity proved to be a problem, requiring more time.

6.2 Project Schedule

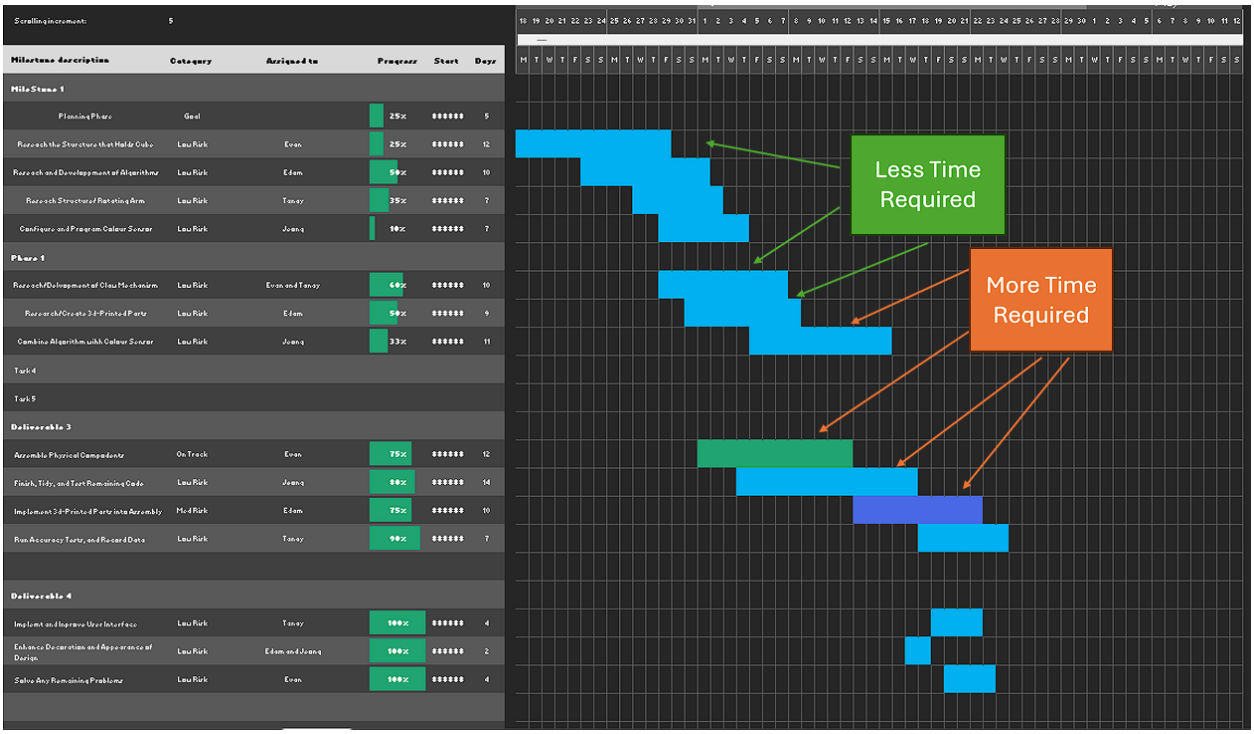

Figure 15: Project Schedule Gantt Chart

Figure 15 outlines some of the changes made to the project schedule. Indicated by the green arrows are changes where less time was required than estimated. In general, these changes occurred during the beginning of the project, as challenges were discovered along the way, causing delays, not directly at the start of the project. The orange arrows indicate areas were more time was required. These areas occurred during the middle of the project to near the end. The majority of these changes were due to the code. Having to utilize a pre-existing algorithm to work directly with the robot formation proved a variety of mechanical and coding challenges. Hence, causing more time to be spent in these areas, and in some cases, taking time from other areas, such as aesthetics. In other words, more time was spent towards function, than any other area of the project during the final stretch of the project (two weeks before demo day).

Overall, the project schedule played a significant role in keeping the group on task. Assigning a variety of tasks using the gannt chart, with individual deadlines, ensured that everyone was participating on an equal basis. In terms of deviation from the schedule, as problems arose over the project, focusing mainly on the coding end, more time was spent in desired areas. As a result, we ended up spending more time than suggested by the schedule.

Looking at the project plan and actual schedule, many differences became apparent. For example, assuming that the entire code can be created, tested, and run perfectly in a selected amount of time is very inaccurate. Also, the schedule did not incorporate feedback utilized from design meetings 1 and 2. Incorporating some of the feedback suggested by the TA, required extra time, hence causing a deviation from the schedule, and altering the plan.

7.0 Closing Ideas

7.1 Recommendations

Looking at the design requirements of the project, there are several areas in which changes could be made to enhance the mechanical and software features of the project. Looking at the mechanical end, the cube grabbing arm could use alternation in the design to improve the accuracy. By ensuring the surface which touches the cube has the perfect amount of friction and angle orientation, it would tremendously reduce the number of times the robot fails to flip the cube. Another alteration to the mechanism within the robot is to use longer axis of rotation for the base platform. By increasing the length of the distance between the motor and the rotating surface, this would increase the torque, putting less stress on the motor, improving the accuracy of the motor under low power implementations. Focusing on the coding end of the project, the primary improvement would be to use an alternative coding language apart from RobotC. Since RobotC does not directly support file I/O, a separate code had to be used to facilitate the transfer of the scrambled file to and the solution file from the computer, adding extra manual steps, taking away from the automated solution. Also, by simplifying the code within another language, coding the emergency stop button would be simplified, reducing the number of times the code needs to check if the touch sensor has been pushed.

7.2 Conclusions

In conclusion, the “EV3 2x2 Rubik’s Cube Solver” project successfully integrated mechanical and software engineering principles to develop a functional robot capable of autonomously solving the 2x2 Rubik’s Cube. Through meticulous planning, innovative design, and effective project management, the group navigated challenges generating solutions that focused on meeting the project objectives. While there are opportunities of further opportunities, this project outlines the groups collaboration, problem-solving and teamwork abilities, building a strong foundation for future endeavors in mechanical engineering practices.

References

[1] MeepMoop and py222, “py222/solver.py at master · MeepMoop/py222,” *GitHub*, Apr. 08, 2019. https://github.com/MeepMoop/py222/blob/master/solver.py (accessed Apr. 08, 2024).

[2] Z. Joffe, M. Robertson, and S. Stephenson, “two-by-two-solver/solver/cube_solver_v3.c at master · ZacJoffe/two-by-two-solver,” *GitHub*, 2017. https://github.com/ZacJoffe/two-by-two-solver/blob/master/solver/cube_solver_v3.c (accessed Apr. 08, 2024).

[3]C. C. W. Hulls, “two-by-two-solver/solver/PC_FileIO.c at master · ZacJoffe/two-by-two-solver,” *GitHub*, 2016. https://github.com/ZacJoffe/two-by-two-solver/blob/master/solver/PC_FileIO.c (accessed Apr. 08, 2024).

Appendix A

```
1.  // ME101 TERM PROJECT
2.  // DONE BY TANAY MEHTA, EVAN MCINTYRE, JEONG CHOI, EDAM JIN
3.
4.  #include "EV3_FileIO.c"
5.
6.  const int sides = 6
7.  const int rows = 2;
8.  const int cols = 2;
9.  const int max_moves = 11; // max possible amount of moves it could take to solve a 2x2
10.
11. // used for the sensors, made global to use in functions as arrays cannot
12. // be passed as parameters in RobotC
13. int faceTop[rows][cols];
14. int faceBack[rows][cols];
15. int faceBottom[rows][cols];
16. int faceFront[rows][cols];
17. int faceRight[rows][cols];
18. int faceLeft[rows][cols];
19.
20. string sol[max_moves];
21.
22. void configureAllSensors()
23. {
24.     SensorType[S3] = sensorEV3_Touch;
25.     SensorType[S2] = sensorEV3_Ultrasonic;
26.     SensorType[S1] = sensorEV3_Color;
27.     wait1Msec(50);
28.     SensorMode[S1] = modeEV3Color_Color;
29.     wait1Msec(50);
30. }
31.
32. void waitForPress(BUTTONTYPE button)
33. {
34.     while (!getButtonPress(button))
35.     {}
36.
37.     while (getButtonPress(button))
38.     {}
39. }
40.
41. void moveColourSensor(bool isMovedForward)
42. {
43.     nMotorEncoder[motorB] = 0;
44.
45.     if (isMovedForward)
46.     {
47.         motor[motorB] = -7;
48.         while (nMotorEncoder[motorB] > -98) // colour sensor down
49.         {}
50.     }
51.     else
52.     {
53.         motor[motorB] = 7;
54.         while (nMotorEncoder[motorB] < 100) // colour sensor up
55.         {}
56.     }
57.     nMotorEncoder[motorB] = 0;
```

```
    motor[motorB] = 0;
}

void PIDController(int target, int motor)
{

    const int MAX_TIME = 0;
    const int MAX_POWER = 100;
    float kpCoeff = 0.82;
    float kiCoeff = 0.8;
    float kdCoeff = 0.71;

    float kpValue = 0, kiValue = 0, kdValue = 0, prevError = 0, errorDiff = 0,
errorSum = 0;

    clearTimer(T1);
    nMotorEncoder[motor] = 0;

    int motorPower = 0;

    while (time1[T1] < MAX_TIME)
    {
        int currentEncoder;

        if (target > 0)
        {
            currentEncoder = abs(nMotorEncoder[motor]);
        }
        else
        {
            currentEncoder = nMotorEncoder[motor];
        }

        int errorValue = target - currentEncoder;

        errorDiff = errorValue - prevError;
        prevError = errorValue;

        kpValue = kpCoeff * errorValue;
        kiValue = kiCoeff * errorSum;
        kdValue = kdCoeff * errorDiff;


                float totalValue = kpValue + kiValue + kdValue;
                if (totalValue > MAX_POWER)
                {
                    motorPower = MAX_POWER;
                }
                else if (totalValue < -(MAX_POWER))
                {
                    motorPower = -(MAX_POWER);
                }
                else
                {
                    motorPower = (int)totalValue;
                }

                motor[motor] = motorPower;
          }
       }

```

```
119.        void flipCube(int flips)
120.        {
121.            nMotorEncoder[motorA] = 0;
122.
123.            for (int runs = 1; runs <= flips; runs++)
124.            {
125.                nMotorEncoder[motorA] = 0;
126.
127.                motor[motorA] = 25; // send the arm backwards
128.
129.                while (nMotorEncoder[motorA] <= 205)
130.                {}
131.
132.                motor[motorA] = 0;
133.                wait1Msec(200);
134.                motor[motorA] = -10;
135.
136.                while (nMotorEncoder[motorA] > 0) // push the cube forward
137.                {}
138.
139.                motor[motorA] = 0;
140.                wait1Msec(1250);
141.            }
142.
143.            motor[motorA] = -25;
144.
145.            while (nMotorEncoder[motorA] > 0)
146.            {}
147.
148.            motor[motorA] = 0;
149.        }
150.
151.        void turnCubeCW(int rotations)
152.        {
153.            PIDController(rotations * 3 * 90, motorC);
154.            motor[motorC] = 0;
155.        }
156.
157.        void turnCubeCCW(int rotations)
158.        {
159.            PIDController(-rotations * 3 * 90, motorC);
160.            motor[motorC] = 0;
161.        }
162.
163.        void turnCW(int rotations)
164.        {
165.            nMotorEncoder[motorA] = 0;
166.
167.            motor[motorA] = 20;
168.
169.            while (nMotorEncoder[motorA] <= 95)
170.            {}
171.
172.            motor[motorA] = 0;
173.            wait1Msec(200);
174.
175.            PIDController((rotations * 3 * 90) + 65, motorC); // does one turn
176.            PIDController(-65, motorC);
177.            motor[motorC] = 0;
178.
179.            nMotorEncoder[motorA] = 0;
```

```
180.
181.            motor[motorA] = -20;
182.
183.            while (nMotorEncoder[motorA] >= -95)
184.            {}
185.
186.            motor[motorA] = 0;
187.        }
188.
189.        void turnCCW(int rotations)
190.        {
191.            nMotorEncoder[motorA] = 0;
192.
193.            motor[motorA] = 20;
194.
195.            while (nMotorEncoder[motorA] < 85)
196.            {}
197.
198.            motor[motorA] = 0;
199.            wait1Msec(200);
200.
201.            PIDController((-rotations * 3 * 90) + 65, motorC); // does one turn
202.            PIDController(65, motorC);
203.            motor[motorC] = 0;
204.
205.            nMotorEncoder[motorA] = 0;
206.
207.            motor[motorA] = -20;
208.
209.            while (nMotorEncoder[motorA] > -85)
210.            {}
211.
212.            motor[motorA] = 0;
213.        }
214.
215.        void inspectCube()
216.        {
217.            int flips = 1, numTurns = 1;
218.
219.            for (int face = 0; face < sides; face++)
220.            {
221.                moveColourSensor(true);
222.                turnCubeCCW(numTurns * 2); // original position
223.                numTurns = 1;
224.
225.                for (int row = 0; row < rows; row++)
226.                {
227.                    for (int col = 0; col < cols; col++)
228.                    {
229.                        if (face == 0) // top
230.                        {
231.                            if (row == 0)
232.                            {
233.                                wait1Msec(100);
234.                                faceTop[row][col] = SensorValue[S1];
235.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
236.                                turnCubeCCW(numTurns);
237.                            }
238.
239.
240.
```

```
241.                            else
242.                            {
243.                                turnCubeCCW(numTurns);
244.                                wait1Msec(100);
245.                                faceTop[row][col] = SensorValue[S1];
246.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
247.                                numTurns *= -1;
248.                            }
249.                        }
250.                        else if (face == 1) // back
251.                        {
252.                            if (row == 0)
253.                            {
254.                                wait1Msec(100);
255.                                faceBack[row][col] = SensorValue[S1];
256.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
257.                                turnCubeCCW(numTurns);
258.                            }
259.                            else
260.                            {
261.                                turnCubeCCW(numTurns);
262.                                wait1Msec(100);
263.                                faceBack[row][col] = SensorValue[S1];
264.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
265.                                numTurns *= -1;
266.                            }
267.                        }
268.                        else if (face == 2) // bottom
269.                        {
270.                            if (row == 0)
271.                            {
272.                                wait1Msec(100);
273.                                faceBottom[row][col] = SensorValue[S1];
274.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
275.                                turnCubeCCW(numTurns);
276.                            }
277.                            else
278.                            {
279.                                turnCubeCCW(numTurns);
280.                                wait1Msec(100);
281.                                faceBottom[row][col] = SensorValue[S1];
282.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
283.                                numTurns *= -1;
284.                            }
285.                        }
286.                        else if (face == 3) // front
287.                        {
288.                            if (row == 0)
289.                            {
290.                                faceFront[row][col] = SensorValue[S1];
291.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
292.                                turnCubeCCW(numTurns);
293.                            }
294.                            else
295.                            {
296.                                turnCubeCCW(numTurns);
297.                                wait1Msec(100);
298.                                faceFront[row][col] = SensorValue[S1];
299.                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
300.                                numTurns *= -1;
301.                            }
```

```
                        }
                        else if (face == 4) // right
                        {
                            if (row == 0)
                            {
                                wait1Msec(100);
                                faceRight[row][col] = SensorValue[S1];
                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
                                turnCubeCCW(numTurns);
                            }
                            else
                            {
                                turnCubeCCW(numTurns);
                                wait1Msec(100);
                                faceRight[row][col] = SensorValue[S1];
                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
                                numTurns *= -1;
                            }
                        }
                        else if (face == 5) // left
                        {
                            if (row == 0)
                            {
                                wait1Msec(100);
                                faceLeft[row][col] = SensorValue[S1];
                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
                                turnCubeCCW(numTurns);
                            }
                            else
                            {
                                turnCubeCCW(numTurns);
                                wait1Msec(100);
                                faceLeft[row][col] = SensorValue[S1];
                                displayString(5, "Colour sensed is %i",SensorValue[S1]);
                                numTurns *= -1;
                            }
                        }
                    }
                }

                if (face <= 2)
                {
                    moveColourSensor(false);
                    flipCube(flips);
                }
                else if (face == 3)
                {
                    moveColourSensor(false);
                    flipCube(flips * 2);
                    turnCubeCCW(numTurns);
                    flipCube(flips);
                    turnCubeCW(numTurns);
                }
                else if (face == 4)
                {
                    moveColourSensor(false);
                    flipCube(flips * 2);
                    turnCubeCCW(numTurns * 2);
                }


```

```
363.                else // original pos
364.                {
365.                    moveColourSensor(false);
366.                    turnCubeCW(numTurns);
367.                    flipCube(flips);
368.                    turnCubeCW(numTurns);
369.                    flipCube(flips);
370.                    wait1Msec(500);
371.                    motor[motorA] = motor[motorB] = motor[motorC] = 0;
372.                }
373.            }
374.        }
375.
376.        void output(TFileHandle fout) // outputs scrambled state of the cube
377.        // to a file to be read in by the python program
378.        // implements PC_FileIO.c
379.        {
380.            for (int face = 0; face < sides; face++)
381.            {
382.                for (int row = 0; row < rows; row++)
383.                {
384.                    for (int col = 0; col < cols; col++)
385.                    {
386.                        if (face == 0)
387.                        {
388.                            writeLongPC(fout, faceTop[row][col]);
389.                            writeCharPC(fout, ' ');
390.                        }
391.                        else if (face == 1)
392.                        {
393.                            writeLongPC(fout, faceRight[row][col]);
394.                            writeCharPC(fout, ' ');
395.                        }
396.                        else if (face == 2)
397.                        {
398.                            writeLongPC(fout, faceFront[row][col]);
399.                            writeCharPC(fout, ' ');
400.                        }
401.                        else if (face == 3)
402.                        {
403.                            writeLongPC(fout, faceBottom[row][col]);
404.                            writeCharPC(fout, ' ');
405.                        }
406.                        else if (face == 4)
407.                        {
408.                            writeLongPC(fout, faceLeft[row][col]);
409.                            writeCharPC(fout, ' ');
410.                        }
411.                        else if (face == 5)
412.                        {
413.                            writeLongPC(fout, faceBack[row][col]);
414.                            writeCharPC(fout, ' ');
415.                        }
416.                    }
417.                }
418.                writeCharPC(fout, ' ');
419.            }
420.        }
421.
422.
423.
```

```
424.        int solve()
425.        {
426.            bool val = true;
427.            int count = 0;
428.
429.            for (int index = 0; index < max_moves && val; index++)
430.            {
431.                if (sol[index] == "F") // clockwise turn front side
432.                {
433.                    flipCube(1);
434.                    turnCCW(1);
435.                    turnCubeCCW(2);
436.                    flipCube(1);
437.                    turnCubeCCW(2);
438.                    wait1Msec(1500);
439.                    count++;
440.                }
441.                else if (sol[index] == "L") //left side down
442.                {
443.                    turnCubeCCW(1);
444.                    flipCube(1);
445.                    turnCCW(1);
446.                    turnCubeCCW(2);
447.                    flipCube(1);
448.                    turnCubeCCW(1);
449.                    wait1Msec(1500);
450.                    count++;
451.                }
452.                else if (sol[index] == "R") // right side up
453.                {
454.                    turnCubeCW(1);
455.                    flipCube(1);
456.                    turnCCW(1);
457.                    turnCubeCCW(2);
458.                    flipCube(1);
459.                    turnCubeCW(1);
460.                    wait1Msec(1500);
461.                    count++;
462.                }
463.                else if (sol[index] == "U") // 2nd layer left
464.                {
465.                    flipCube(2);
466.                    turnCCW(1);
467.                    flipCube(2);
468.                    wait1Msec(1500);
469.                    count++;
470.                }
471.                else if (sol[index] == "B'") // back side clockwise
472.                {
473.                    turnCubeCCW(2);
474.                    flipCube(1);
475.                    turnCW(1);
476.                    turnCubeCW(2);
477.                    flipCube(1);
478.                    wait1Msec(1500);
479.                    count++;
480.                }
481.
482.
483.
484.
```

```
485.                else if (sol[index] == "F'") // front side counterclockwise
486.                {
487.                    flipCube(1);
488.                    turnCW(1);
489.                    turnCubeCW(2);
490.                    flipCube(1);
491.                    turnCubeCCW(2);
492.                    wait1Msec(1500);
493.                    count++;
494.                }
495.                else if (sol[index] == "R'") //right side down
496.                {
497.                    turnCubeCCW(1);
498.                    flipCube(1);
499.                    turnCW(1);
500.                    turnCubeCCW(2);
501.                    flipCube(1);
502.                    turnCubeCCW(1);
503.                    flipCube(1);
504.                    wait1Msec(1500);
505.                    count++;
506.                }
507.                else if (sol[index] == "U'") // 2nd layer to the right
508.                {
509.                    flipCube(2);
510.                    turnCW(1);
511.                    flipCube(2);
512.                    wait1Msec(1500);
513.                    count++;
514.                }
515.                else if (sol[index] == "F2") // front side twice
516.                {
517.                    flipCube(1);
518.                    turnCW(2);
519.                    turnCubeCW(2);
520.                    flipCube(1);
521.                    turnCubeCW(2);
522.                    wait1Msec(1500);
523.                    count++;
524.                }
525.                else if (sol[index] == "R2") // right twice
526.                {
527.                    turnCubeCCW(1);
528.                    flipCube(1);
529.                    turnCW(2);
530.                    flipCube(1);
531.                    turnCubeCCW(1);
532.                    wait1Msec(1500);
533.                    count++;
534.                }
535.                else if (sol[index] == "U2") // 2nd layer twice
536.                {
537.                    flipCube(2);
538.                    turnCW(2);
539.                    flipCube(2);
540.                    wait1Msec(1500);
541.                    count++;
542.                }
543.
544.
545.
```

```
546.                else
547.                    val = false;
548.            }
549.            return count;
550.        }
551.
552.        task main()
553.        {
554.            configureAllSensors();
555.
556.            waitForPress(ENTER_BUTTON);
557.
558.            while (SensorValue[S2] > 10)
559.            {}
560.
561.            displayString(6, "Starting...");
562.            wait1Msec(5000);
563.
564.            eraseDisplay();
565.            displayString(6, "Inspecting...");
566.
567.            inspectCube();
568.
569.            TFileHandle fout; // setup file to be outputed
570.            bool fileOutputOkay = openWritePC(fout, "scramble.txt");
571.
572.            output(fout);
573.
574.            displayString(6, "Waiting for sol to be uploaded to robot...");
575.            displayString(7, "Press enter button once uploaded");
576.            waitForPress(ENTER_BUTTON);
577.
578.            TFileHandle fin; // setup file to be inputed
579.            bool fileInputOkay2 = openReadPC(fin, "solution.txt");
580.
581.            for (int index = 0; index < max_moves; index++)
582.            {
583.                readTextPC(fin, sol[index]);
584.            }
585.
586.            time1[T1] = 0;
587.            clearTimer(T1);
588.
589.            int numMoves = solve();
590.
591.            float time = (time1[T1]/1000.0); // time taken to solve cube in seconds
592.
593.            turnCubeCCW(5); // in celebration!
594.
595.            displayString(4, "It took %d moves to solve!", numMoves);
596.            displayString(5, "It took %f seconds to solve", time);
597.            displayString(6, "Press any button to exit");
598.
599.            waitForPress(ANY_BUTTON);
600.
601.            closeFilePC(fout);
602.            closeFilePC(fin);
603.        }
```

Figure 16: RobotC Primary Code