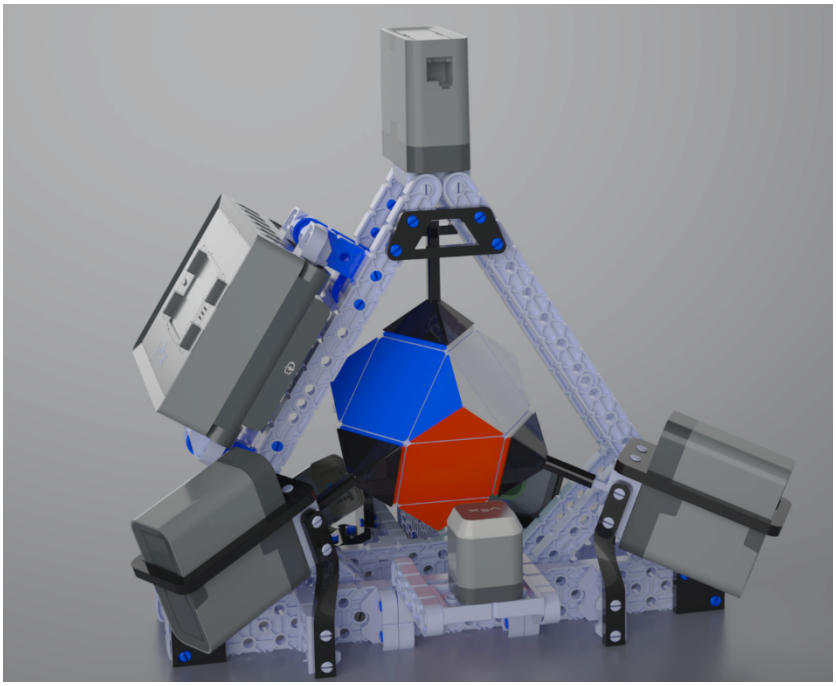
University of Waterloo

MTE100 / MTE121 Final Project

December 2nd, 2025

Skewb Solver



Submitted by group 15:

Adam Turaj

Robert Vigneron

Jason Liang

Andrii Bessarab

Table of Contents

Table of Contents ii
List of Tables iv
List of Figures iv
Summary v
Acknowledgments v
1.0 Introduction 1
2.0 Scope 1
2.1 Project Scope 1
2.2 Interactions with the Environment 2
2.2.1 Sensor Interactions 2
2.2.2 Interactions 2
2.3 Shutdown Procedure 3
2.4 Changes in Scope 3
3.0 Constraints & Criteria 3
3.1 Constraints 3
3.2 Criteria 4
3.2.1 Reliability 4
3.2.2 Solving Speed 6
4.0 Mechanical Design and Implementation 6
4.1 Grippers 6
4.2 Motor Orientation 7
4.3 The Frame 9
4.3.1 High Level Overview 9
4.3.2 Rotational Reinforcement 10
4.3.3 Rear Motor Mounting 10
4.3.4 Additional Reinforcements 11
4.4 Motor Mounts 11
4.4.1 Top Motor Mount 11

4.4.2 Back Motor Mount 11
4.4.3 Front Motor Mounts 12
4.5 Distance & Optical Sensor Mounts 14
4.6 Brain Mount 15
5.0 Software Implementation 16
5.1 High Level Overview 16
5.1.1 Move Notation 17
5.2 Solution Finder 17
5.2.1 God's Number 17
5.2.2 Benefit of Pre-computing 17
5.2.3 Bit-packing 18
5.3 Web Interface 18
5.4 VEX Program 19
5.4.1 Serial Communication 19
5.4.2 State Normalization 19
5.4.3 Matching State to Solution 21
5.4.4 Placement Validation with Sensors 21
5.4.5 Solving & PID Controllers 21
5.5 Demonstration Task Sequence 22
5.6 Functions Breakdown 23
5.7 Testing and Troubleshooting 25
6.0 Verification 26
7.0 Project Plan 26
7.1 Time Deviations and Revisions 26
8.0 Conclusion 27
9.0 Recommendations 28
9.1 Hardware Improvements 28
9.2 Software Improvements 28
10.0 References 28

List of Tables

Table 1: Vex functions breakdown 23

List of Figures

Figure 1: A picture of the Skewb [3] 1
Figure 2: The orientation of grippers 2
Figure 3: Motor mount coming off pegs 5
Figure 4: The Grippers 7
Figure 5: Internal geometry of grippers 7
Figure 6: Gripper Shaft Offset 7
Figure 7: Original motor layout 8
Figure 8: The motor mounting 8
Figure 9: Example of removing motors to simplify installation 9
Figure 10: The triangular structure 9
Figure 11: Frame reinforcements 10
Figure 12: Example of rotating motor 10
Figure 13: Rear motor extension 10
Figure 14: Additional reinforcements 11
Figure 15: Cross section of top motor mount 11
Figure 16: Side view of top motor mount 11
Figure 17: Rear motor mount assembly 12
Figure 18: Motor mount front view 12
Figure 19: Back motor brace 12
Figure 20: Motor mount outer arm 13
Figure 21: Motor mount inner arm 13
Figure 22: Motor brace 14
Figure 23: Lack of space near sensor face 15
Figure 24: 3D printed sensor mount 15
Figure 25: Brain Mounting position 15
Figure 26: High-level flow chart of software architecture 16
Figure 27: Web interface demonstration 18
Figure 28: High-level flow chart of VEX Brain Code 19
Figure 29: State normalization process flow chart 20

Figure 30: Flow chart of solution finding..... 21
Figure 31: Flow chart of executing a move with custom PID 22
Figure 32: Project timeline 27

Summary

This project involved the design and implementation of a semi-autonomous robot capable of solving a Skewb in the fewest number of moves. The mechanical system consisted of four motors paired with custom 3D-printed grippers arranged such that no two grippers were positioned on adjacent corners, enabling full rotational control of the cube. The software design incorporated a PID control loop for accurate and repeatable motor movements, a web-based cube-scanning interface, a Breadth-First Search (BFS) algorithm compiling all solutions into a binary-encoded file, a collection of functions that normalized the scanned state before matching it to the appropriate solution in the database.

The robot successfully solved the puzzle from any scrambled state in under fifteen seconds. Major project constraints including motor accuracy, state normalization, and the limitation of using only four motors were addressed, although full motor speed optimization was not fully achieved. Overall, the project exceeded its initial constraints and demonstrated a reliable and efficient Skewb-solving robot.

Acknowledgments

All 3D printed components used in this project were designed and created entirely by the team. No pre-existing parts or external designs were used during the development of this robot.

All C++, and web code as well as algorithms were designed from scratch by the team. Relevant ideas and concepts that have not been explored in the course are mentioned in comments at the top of respective files of the graded component.

1.0 Introduction

Rubik's Cubes are widely recognized mechanical puzzles first invented in 1974 with the original 3×3 design. Since then, the puzzle family has expanded to include ten major official variants and many more unofficial ones, each with its own active speedcubing community dedicated to solving these puzzles as quickly as possible [1]. Due to the constraint of using only four motors, solving the 3×3 with this limitation would be highly inefficient. Therefore, the project focuses on a different variant that is compatible with this constraint: the Skewb.

The Skewb (Figure 1) resembles a standard cube in appearance, but unlike the 3×3, which rotates around its faces, the Skewb rotates around its corners [2]. The goal of the robot is to solve the Skewb from any scrambled state in as little time as possible, using the starting state provided as a thirty-character string of colours.

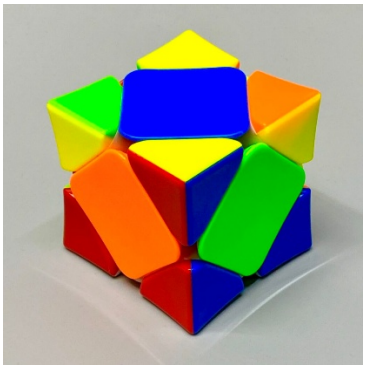


Figure 1: A picture of the Skewb [3]

2.0 Scope

2.1 Project Scope

The main function of this project outlined in Section 1.0. To achieve this, the robot performs four main primary tasks: it communicates with external web interface through serial, normalizes the scanned corner colours into a consistent representation, cross-references this normalized string with a database of valid Skewb solutions, and executes the corresponding move sequence once it verifies that the cube is correctly oriented and properly seated in the grippers.

The scope of the project changed from earlier stages. Initially, the robot was intended to semi-automatically scan the cube by rotating it and capturing colour data directly using onboard sensors. However, due to the inaccuracy of the colour sensor, the mechanical difficulties associated with rotating the cube reliably, and the challenge of designing a

straightforward scanning workflow, this approach was removed. Instead, the scanning process was relocated to a website, which simplified the mechanical design and significantly increased the reliability of detecting the cube's initial configuration.

2.2 Interactions with the Environment

2.2.1 Sensor Interactions

The robot measures and detects the initial configuration of the Skewb using a custom React-based cube-scanning website. The scanned colours are encoded as a string representing the orientations of all corners and sent to the robot over serial communication. This data is then normalized and matched to an optimal Skewb solution stored in a solution file. Before the robot begins solving, it uses a distance sensor to confirm that the cube is properly installed in the grippers and employs the colour sensor to scan one of the main faces to verify correct orientation. As described in Section 2.1, the colour sensor was repurposed from full-cube scanning to orientation validation during development. Finally, the user presses the TouchLED button to begin the solve. Other than performing the external scan and initiating the solve, no further human input is required.

2.2.2 Interactions

The robot interacts with the physical environment through four motors that manipulate the Skewb. Each motor drives a custom 3D-printed gripper, arranged such that no two held corners are adjacent (Figure 2). This configuration allows any one corner to be rotated independently without disturbing the others, simplifying the mechanical design. Motor control is handled autonomously using eight independent PID controllers (described in Section 5.4.5), which execute the calculated move sequence required to transform the cube from its scrambled state to the solved configuration.

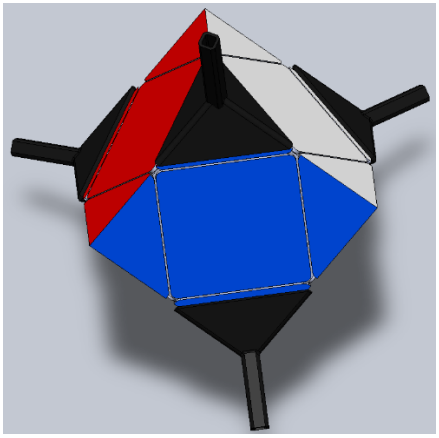


Figure 2: The orientation of grippers

2.3 Shutdown Procedure

The robot recognizes that all tasks are complete once the final move in the solution sequence has been executed. Throughout operation, the controller tracks progress by iterating through the move list and verifying that each commanded rotation has reached its target position. When no further moves remain, the robot concludes that the solve sequence is finished.

The shutdown procedure is straightforward. After the final move, the controller disables all motors, displays the total solve time, and sets the TouchLED to green to indicate completion. After a 20-second delay, the program is terminated using `Brain.programStop()`.

In case of an issue during the solve, the program can be terminated early by pressing right-arrow button on the Brain. It skips the rest of the moves and fast-forwards to the `Brain.programStop()` at the end of the main module.

2.4 Changes in Scope

Throughout the project, the overall scope remained the same: to quickly and reliably solve the Skewb. However, several minor changes occurred during development that changed how this goal was achieved. As described in Sections 3.1 and 5.3, the colour-scanning process was moved from onboard sensors to an external web interface, and the onboard colour sensor was repurposed for orientation validation. Minor adjustments were also made to the mechanical design and motor control strategy to improve reliability and reduce system complexity. These changes did not alter the project's core functionality but simplified design and increased overall performance.

3.0 Constraints & Criteria

3.1 Constraints

When creating the project proposal, possible constraints were a major consideration. It was important to not over define constraints to prevent unnecessarily limiting the design. However, it is also important to understand the limitations to avoid wasting time. The constraints were:

- The VEX IQ Brain has severely limited RAM. This necessitated storing solutions on an SD card and compressing them via bit-packing (Section 5.2.3).
- The VEX Smart Motors produce limited torque and have compliant internal gearing. This required custom PID controllers since built-in PIDs are too gentle to combat Skewb's friction and magnets.
- Due to the deep-cut nature of the puzzle, rotating one corner affects three adjacent faces, so the robot cannot grip adjacent corners. As described in Section 4.2, the grippers therefore hold every second (non-adjacent) corner to avoid mechanical interference.
- If a rotation is not completed within a tight tolerance, the subsequent move will jam the mechanism. This required high precision turns which can be achieved directionally tuned PID control (see Section 5.4.5).
- The robot cannot mechanically inspect the hidden faces of the cube once grasped. Therefore, the full state of the puzzle must be known and validated before the cube is inserted into the solver.

The constraints did not change much over the project, because they were adhered to while designing the robot. However, for reasons explained in Section 2.1, the following constraint was added:

- The standard VEX Optical Sensor proved unreliable for full cube scanning due to sensitivity to lighting. This constrained the design to use an external web interface for state acquisition.

3.2 Criteria

Choosing the criteria carefully was crucial towards influencing design decisions, especially in cases where success in one aspect may sacrifice success in another aspect. Because of this, the criteria measuring the overall success of the robot were simple and did not change from the beginning of the project.

3.2.1 Reliability

The most important evaluation criteria was the robot's overall reliability, defined as successful solves per ten attempts. The target reliability was set at 80%, or eight successful solves out of ten. Throughout development, reliability was measured at the subsystem level, since overall performance could not be evaluated until all components were complete.

To begin, the reliability of sending data from the web interface to the VEX IQ Brain had to be verified. This stage of the solving process was desired to have a 100% success rate

because of its predictable nature. This was confirmed through testing and was achieved before beginning the other aspects of the project.

As explained in the Section 3.1 and Section 5.3, the cube could not be reliably scanned with the VEX colour sensors. Instead, a web interface converts RGB to HSV and classifies sticker colours, improving scanning reliability to approximately 95%.

Finally, the reliability of the robot itself had to be verified. This was measured by the number of successful moves per 100 total moves. This reliability relied on the motor mounts (Section 4.4), Grippers (Section 4.1), and kinematics software (Section 5.4.5).

As detailed in Sections 4.1 and 4.4, both the motor mounts and grippers went through several iterations to eliminate failures such as motors popping out of mounts and gripper shafts snapping. After adding custom braces and redesigning the grippers, these components did not fail during the final round of testing and demos, so their reliability was treated as effectively 100% for the purposes of this project. It should be noted that by the end of the project there was visible wear on the holes in the 3d printed mounts causing them to tilt at slight angles, as pictured in Figure 3**Error! Reference source not found.**. They never failed within the timeline of the robot, but because of this, it is doubtful that the mounts would be 100% reliable should more testing be done.



Figure 3: Motor mount coming off pegs

As described in Section 5.4.5, the built-in motor PID was too gentle to overcome friction and magnets. Replacing it with custom, direction-specific PID controllers increased per-move reliability from roughly 20% to about 95%, which in turn allowed the overall system to reach the 80% solve reliability target.

In all, the overall reliability resulted in 80%. Due to lack of further data and complexity of absolute calculations, all the data above are estimates. It should also be noted that the achieved reliability exactly met the initial goal of 80%, and so rather than allocating more resources to improving reliability, the other criteria was focused on.

3.2.2 Solving Speed

The project aimed to beat the fastest known solve of the Skewb by a robot, claiming the world record. This time is approximately 1:20 [4] , and so the approximate goal for a solve time was deemed to be under 1 minute. However, as the project progressed it became clear that this goal was an underestimation of the robot's capabilities.

To begin, while creating the file of all possible Skewb states to find solves from any state, it was concluded that any state can be solved with a maximum of eleven moves with the average being eight. Because the robot does not require rotation of the cube between moves, the total solve time will be the time it takes to perform one move multiplied by a number less than or equal to eleven. This makes the under-one-minute target easily achievable, prompting a new goal of under 30 seconds.

As mentioned in Section 3.2.1, the reliability of the robot was prioritized before improving solving speed past the 30 second target. Once the reliability goal was achieved, the solving speed were improved to by increasing the maximum rotation speed along with adjusting PID values to maximize solving speed while still achieving the reliability goal.

4.0 Mechanical Design and Implementation

When approaching the mechanical design, the group adopted a philosophy that balanced achieving a product fulfilling the criteria with creating an aesthetically pleasing form. In this context, "aesthetically pleasing" refers to maximizing the cube's visibility from the front and maintaining smooth, simple geometric features throughout the design.

4.1 Grippers

The grippers were the first components designed for the Skewb-solving robot. Their geometry is displayed in Figure 4. Their primary purpose is to securely hold the Skewb's corners and are manufactured through 3D printing. The internal geometry is created by offsetting the shape of a Skewb corner, which accounts for the natural tolerance variation of 3D-printed parts. Additionally, 1 mm was removed from the top surface to prevent the grippers from interfering with the cube's center pieces. Figure 5 illustrates this internal offset.

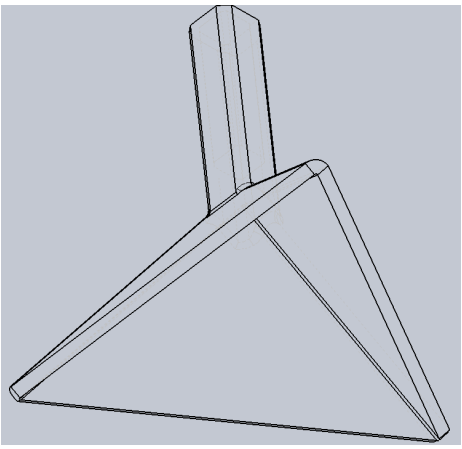


Figure 4: The Grippers

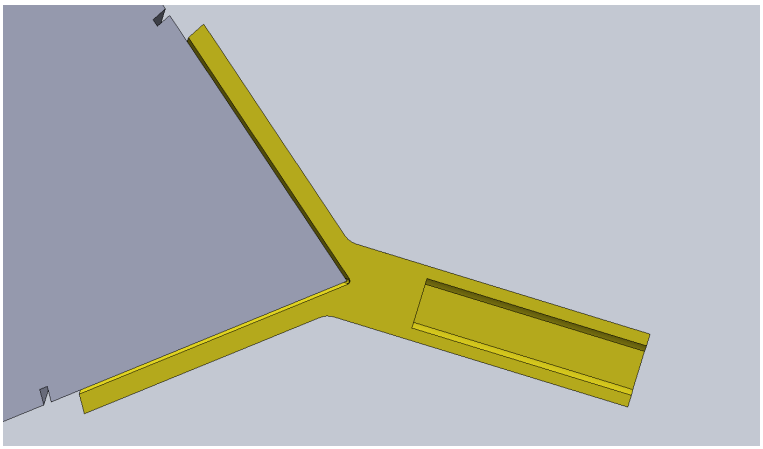


Figure 5: Internal geometry of grippers

Each gripper is attached to the robot through a shaft that mates directly with the motor output. To minimize play and ensure the grippers remain attached during installation, the interface uses a tight 0.1mm clearance fit (Figure 6), and the shaft is extended to increase friction and distribute lateral loads over a larger area. A simple fillet connects the shaft to the gripper body, reducing stress concentrations at the joint. These design choices improve durability and extend the grippers' lifespan, especially during intensive testing where the cube is repeatedly installed and removed.

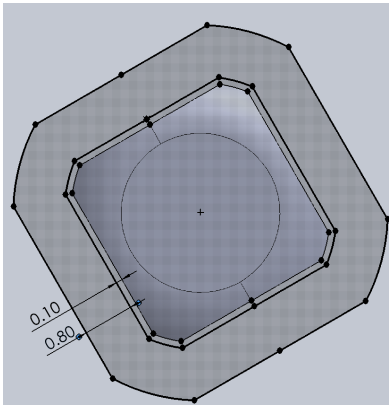


Figure 6: Gripper Shaft Offset

Some minor trade-offs were made in the design of this part. Offsetting the internal geometry introduced a small amount of play between the gripper and the cube's corner, which was necessary for print tolerances but reduced overall precision. Likewise, extending the shaft improved friction and alignment but reduced structural stiffness and ultimately became the primary failure point during testing.

4.2 Motor Orientation

As explained in Section 2.2, the robot requires very specific corners of the Skewb to be gripped to access all the moves. The initial concept placed the four motors in a box-like

arrangement around the cube (Figure 7). This was the most straightforward approach and the most intuitive mechanically.

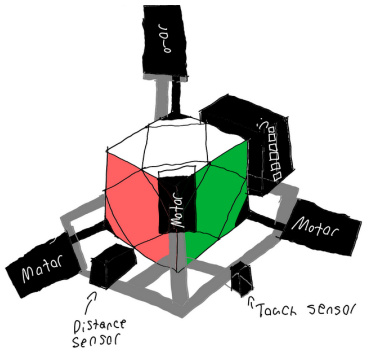


Figure 7: Original motor layout

However, after further brainstorming, the team identified a significantly better configuration: a pyramidal layout as shown in Figure 8. In this design, three motors are aligned on the same vertical plane, with the fourth motor positioned directly above the cube. This arrangement greatly simplifies the mechanical structure by reducing the number of unique mounting angles and placing most of the motors in a shared orientation. It also lowers the overall centre of gravity, improving stability during operation. Additionally, the open-front geometry enhances visibility of the cube from the user's perspective, supporting the project's aesthetic goals.

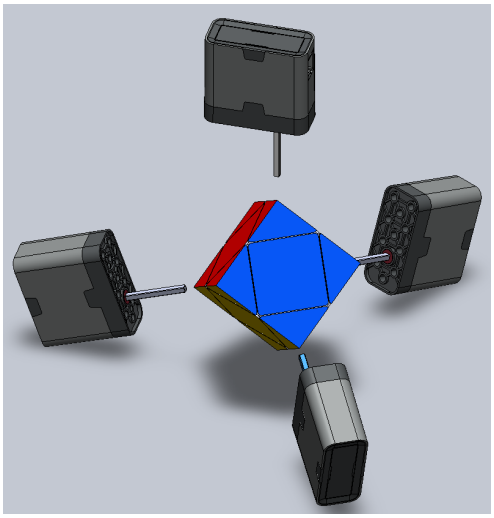


Figure 8: The motor mounting

Some trade-offs accompany this design. The lower motors must operate at angles not supported by Vex IQ, inserting and removing the cube becomes less accessible, and the footprint of the structure increases. These limitations were manageable: custom 3D-printed motor mounts resolved the angular constraints, the front two motors can be

removed from their mounts to simplify cube installation (Figure 9), and the larger footprint was not a concern for this project's scope. Overall, the advantages provided by the pyramidal configuration outweighed its drawbacks and aligned well with the design priorities.

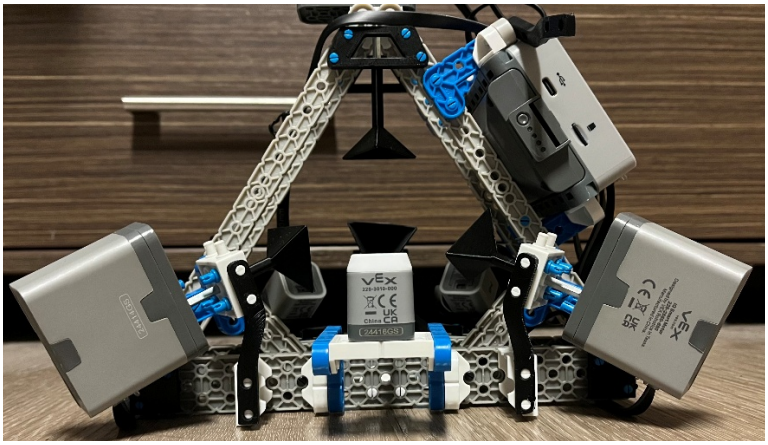


Figure 9: Example of removing motors to simplify installation

4.3 The Frame

4.3.1 High Level Overview

At a high level, the frame is constructed as a triangular prism that fully encompasses the cube. This geometry provides unobstructed access to the puzzle, reduces the overall vertical footprint, and offers convenient mounting locations for three of the four motors. The prism is assembled using four 1x16 beams and two 2x18 beams (Figure 10). These components provide adequate clearance around the cube and give each motor sufficient space to accommodate its shaft.

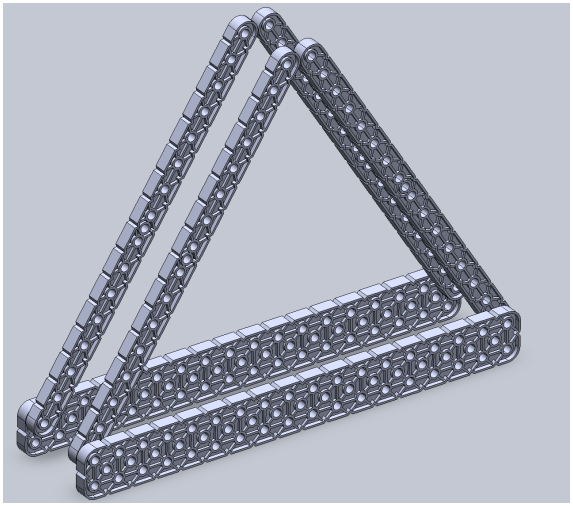


Figure 10: The triangular structure

4.3.2 Rotational Reinforcement

However, this design introduced a few structural issues. Because each 1x16 beam was originally anchored at only one point, the top motor assembly was able to rotate as shown in Figure 12. To address this, two additional reinforcements were added: a custom 3D-printed brace and a set of structural beams. Specifically, Figure 11 shows 1x6 beams were connected to the base at two points, and a top brace with two mounting points was added to further restrict unwanted rotation. While this solution did not eliminate movement entirely, it provided enough stiffness to ensure that the motor remained stable during cube rotations.

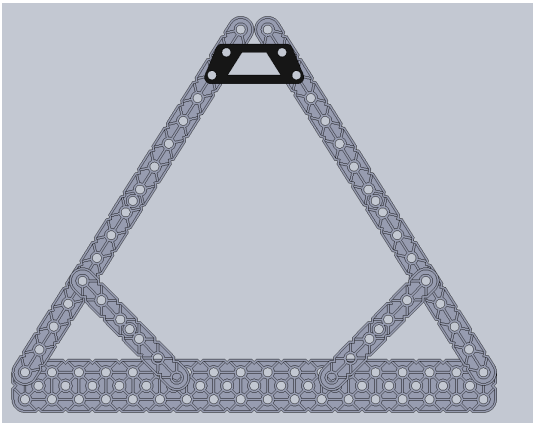


Figure 11: Frame reinforcements



Figure 12: Example of rotating motor

4.3.3 Rear Motor Mounting

The final component of the frame is the rear-motor extension. This subassembly serves two key purposes: it increases the footprint of the base to improve overall stability, and it provides a mounting location for the rear motor. Figure 13 demonstrates this extension is constructed using two 2x7 beams connected to the triangular prism via 90-degree connectors. However, this approach introduced an issue: when the motor experienced higher loads, the extension behaved like a pivot, causing the entire assembly to lift slightly and reducing overall performance. This was temporarily mitigated by adding a small clip that prevented the structure from lifting under stress.

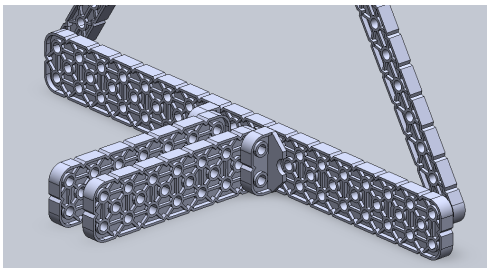


Figure 13: Rear motor extension

4.3.4 Additional Reinforcements

Additional reinforcements were added to prevent the parallel main structure and the rear motor mount structure from buckling toward each other under load. The main structure was stiffened using a central 3x2 connector, while a standoff was installed to support the rear motor mount (Figure 14). Together, these reinforcements eliminated all major buckling.

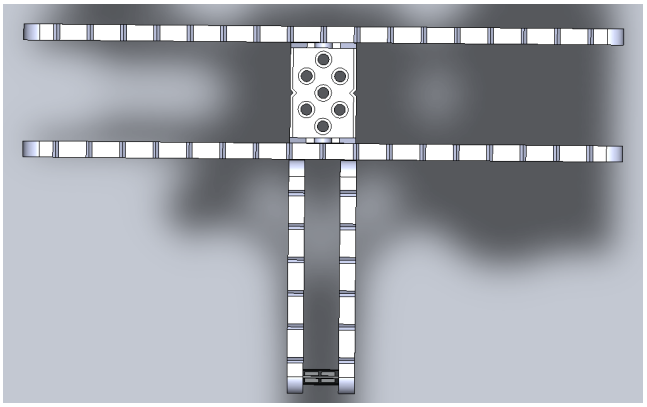


Figure 14: Additional reinforcements

4.4 Motor Mounts

4.4.1 Top Motor Mount

The robot includes three unique motor mounts, the simplest of which is the top motor mount. Figure 15 and Figure 16 show the cross-section and side view of this assembly, respectively. In the final design, four pegs were used instead of the two shown in the figure. This change ensured that the motor remained securely seated and prevented it from dislodging under abnormal or unexpected loads.

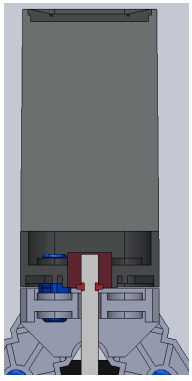


Figure 15: Cross section of top motor mount

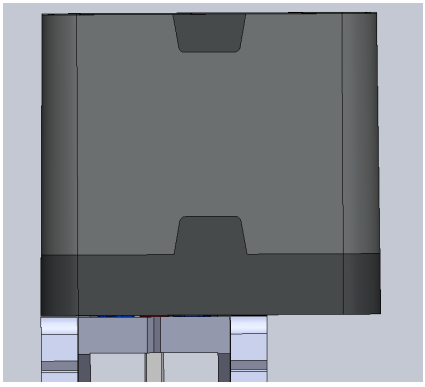


Figure 16: Side view of top motor mount

4.4.2 Back Motor Mount

The back motor is secured using a single 3D-printed mount. This mount attaches to the frame using three pegs on each side, constraining it in all degrees of freedom. Two curved

arms extend from these peg connections to a front face plate (Figure 17). The face plate includes two holes for the mounting pegs and a central opening for the motor shaft. The triangular cutouts on the face plate shown in Figure 18 are purely aesthetic and do not affect structural performance.

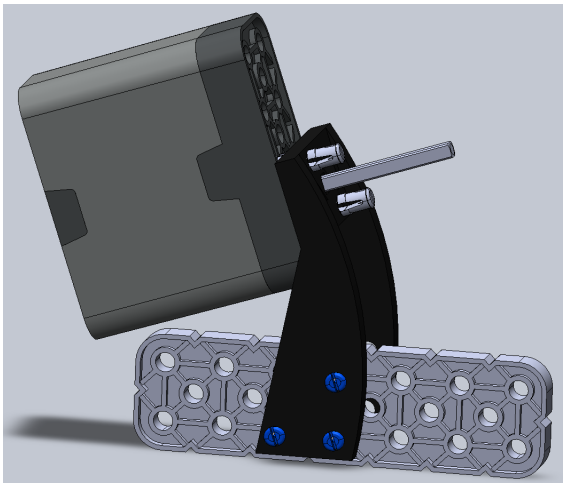


Figure 17: Rear motor mount assembly

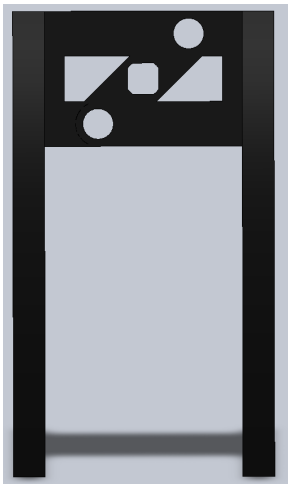


Figure 18: Motor mount front view

Unfortunately, this design was not fully sufficient. Because the Skewb requires a firm grip to reliably flip corners, the motors occasionally popped out of their mounts under load. To address this issue, additional braces were added, as shown in Figure 19. These braces wrap around the motor and provide an external constraint. This addition resolved the issue and ensured that the motor remained securely seated during operation.

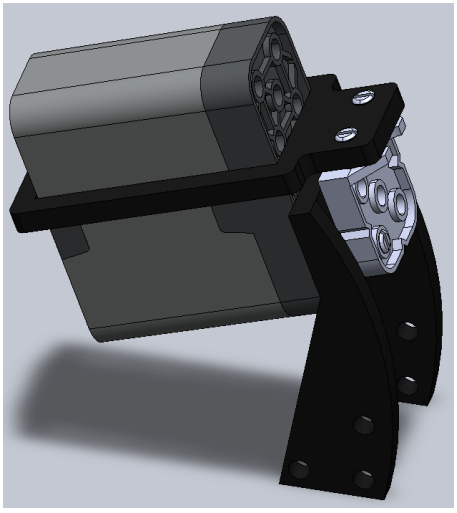


Figure 19: Back motor brace

4.4.3 Front Motor Mounts

The third motor mount supports the two front motors. These mounts are mirrored versions of each other, which simplifies the overall design and avoids unnecessary complexity. Due

to limited space in the front region, each mount was divided into two separate arms. The outer arm is a straightforward lofted component, as shown in Figure 20. This design is strong, visually clean, and mechanically simple, although it must be 3D printed in an orientation which requires supports for adequate strength.

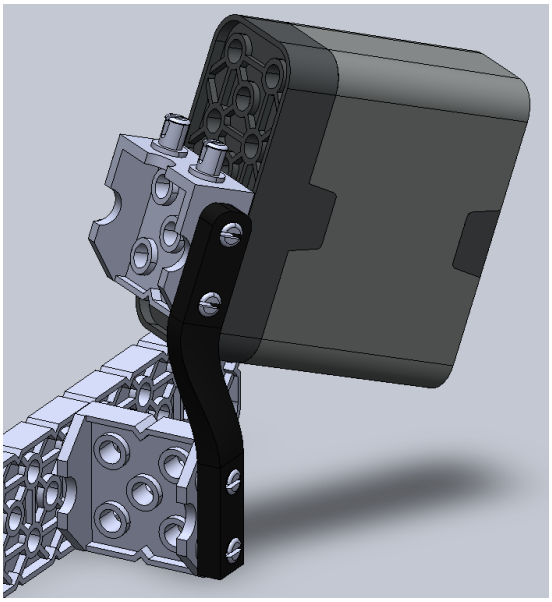


Figure 20: Motor mount outer arm

The inner arm required additional routing to achieve the correct positioning. It was shifted slightly toward the rear of the motor using a small L-shaped connector, which helped avoid extreme loft angles that would otherwise be difficult to print reliably. Similar to the outer arm, the inner arm is lofted toward the 2×18 base and is secured using two connecting pegs on each side to prevent unwanted rotation, as illustrated in Figure 21.

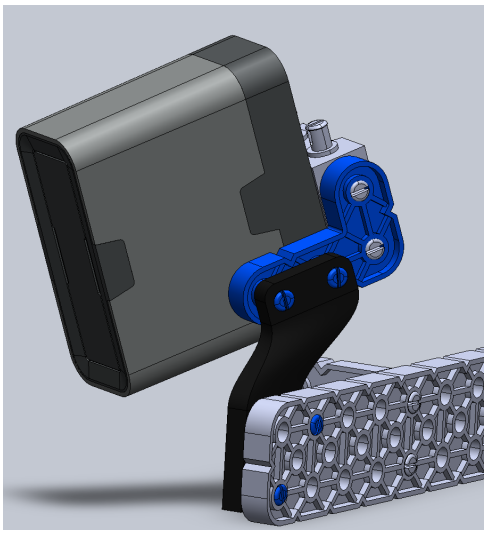


Figure 21: Motor mount inner arm

Both front motors experienced the same issue as the rear motor, where they could occasionally pop out under load. To resolve this, the same brace design used for the rear motor was incorporated. This brace, shown in Figure 22, effectively holds the motor in place and prevents any unintended movement during operation.

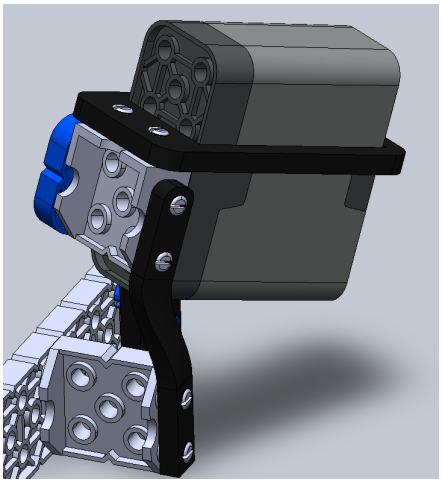


Figure 22: Motor brace

4.5 Distance & Optical Sensor Mounts

The sensor mounts share the same design but are mirrored. From the outset, it became evident that the sensors required stable mounting. Although a loose, adjustable configuration was initially considered to simplify design and manufacturing, it was clear that a fixed orientation was necessary. The distance sensor must remain in a consistent position and direction to avoid recalibrating the custom distance thresholds each time it shifts. Similarly, any offset of the colour sensor could cause it to read the cube's cracks, miss the cube entirely, or sample an area with inconsistent colouring. These considerations led to the decision to use rigid, fixed mounts that do not require adjustment.

It was originally intended that the sensors be mounted using only VEX parts. However, as shown in Figure 23, there was not enough available space to mount them without interference. Because each sensor needed to point directly at the cube face, the optimal placement was determined to be near the center of the large square region of its respective face, maximizing tolerance for cube misalignment or slight sensor offset. No VEX mounting location allowed this orientation without obstructing cube rotation or complicating cube installation. As a result, a 3D-printed mount was chosen to simplify the design. As shown in Figure 24, the printed mount provides the precise angle needed to keep each sensor's viewing axis perpendicular to the cube face while also aiming toward the central region, as required.

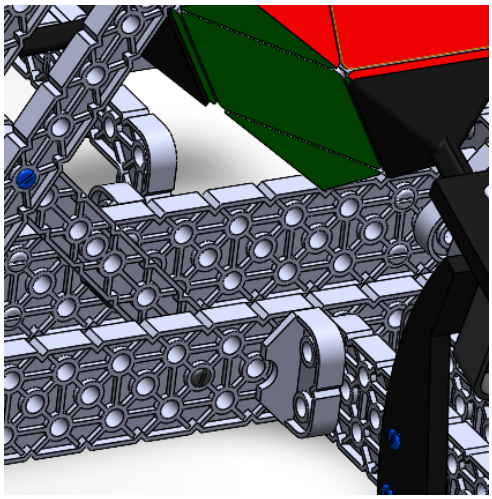


Figure 23: Lack of space near sensor face

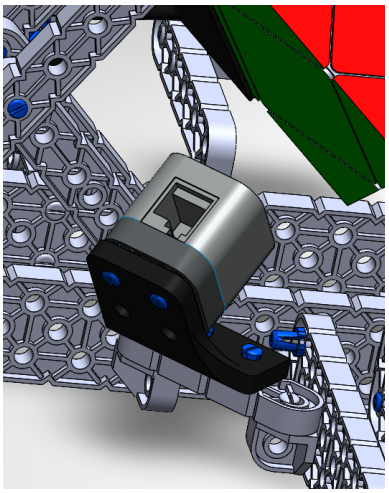


Figure 24: 3D printed sensor mount

4.6 Brain Mount

The VEX IQ Brain had to be mounted so that the screen remained clearly visible, all buttons and ports were accessible, and the battery could be easily removed. The screen must be viewable for reading error codes, the buttons must be reachable for starting the program, the USB-C port must be exposed for uploading code, and the battery must be removable for efficient testing. These requirements led to mounting the Brain on the main upright structure, as shown in Figure 25.

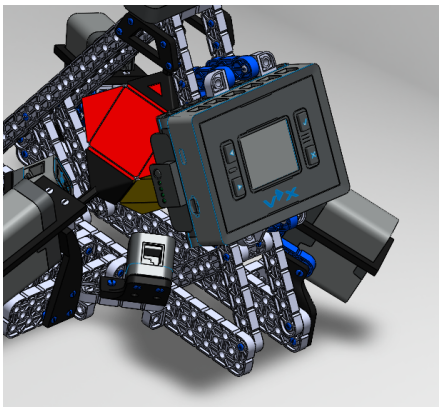


Figure 25: Brain Mounting position

It should also be noted that the Brain could be mounted using only VEX parts, as its orientation is not constrained, only its visibility. Through testing, the four mounting points to the frame were found to be more than sufficient to support the Brain's weight, even though it is positioned approximately 2 cm away from the frame to allow for easy battery removal. The visibility of the Brain was essential for functionality and was therefore prioritized over the aesthetic preference of maintaining full visibility of the cube from the side on which the Brain is mounted.

5.0 Software Implementation

The software for the Skewb-solver can be separated into three main parts:

1. **Solution Finder:** C++ Breadth-First Search (BFS) algorithm that finds all the optimal skewb solutions and stores them in a binary-encoded file.
2. **Scanning App:** web interface that scans the skewb and transmits the data over serial to the VEX brain.
3. **VEX Code:** C++ VEX program that executes a list of tasks on the brain and controls the robot.

This section will briefly discuss Solution Finder (Section 5.2) and Scanning App (Section 5.3). Primary focus will be put on VEX code part of the software (Section 5.4), and will reference others as needed.

5.1 High Level Overview

The chosen software design relies on pre-computing. Finding the solution in the real time is very computationally expensive. Instead, the software relies on database of pre-computed optimal solutions.

The overall workflow, shown in Figure 26, has four main parts:

1. **Pre-computation:** C++ Breadth-First Search (BFS) finds optimal solutions to all possible states and stores them in encoded bin file.
2. **Storage:** The file with the solutions is stored on SD Card installed onto the VEX Brain.
3. **Acquisition**: A web interface is used to scan the cube and acquire the state.
4. **Execution**: The VEX Brain receives the state, processes it, finds solution on SD Card, validated placement with sensors, and executes the moves using custom PID controllers.

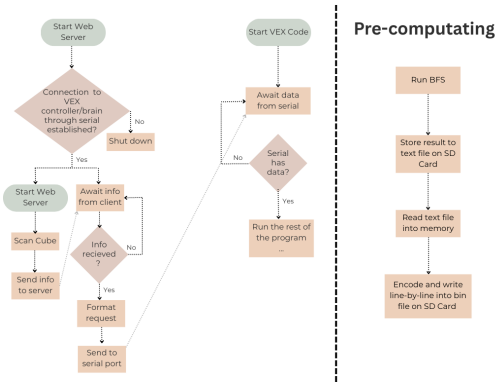


Figure 26: High-level flow chart of software architecture

5.1.1 Move Notation

In order, to standardize interactions between the robot and the solution finder, it was necessary to come up with convention to represent the moves. Unfortunately, standard conventions (like those used in official competitions) would not suffice due to which axis can be rotated (as mentioned in Section 3.1) with current design. Thus, for this project, clockwise rotations along the four fixed axis are denoted as {`R, U, L, F`}. Counterclockwise rotations are denoted as {`r, u, l, f`}, respectively.

5.2 Solution Finder

The Solution Finder is responsible for generating the database of all possible states.

Starting from solved state of a skewb, the algorithm iteratively expands the search frontier by applying all eight possible moves eleven times (see Section 5.2.1 for explanation). After every move, each state is checked against a hash map containing all already computed states and their solutions to identify any duplicates. This assures only the most optimal solutions (the ones with least moves) are recorded. After eleven moves, the hashmap with all states and respective solutions is transferred into a text file. This text file is then encoded into a binary file (see Section 5.2.3).

5.2.1 God's Number

God's number is defined as the maximum number of moves necessary to solve a puzzle. A "move" refers to any single rotation defined in Section 5.1.1. There exist mathematic proofs placing this number at eleven for Skewb.

However, reliance on external source is unnecessary as this software provides an empirical proof. If the depth of BFS algorithm is increased beyond eleven, it was observed that no additional unique states were added to hash map, thus it is safe to say that eleven is God's number for Skewb. Hence, the maximum search depth was capped at eleven.

5.2.2 Benefit of Pre-computing

Following the conclusion of Section 5.2.1, BFS algorithm determined that there are only 3,149,280 unique states (including solved state) of Skewb. Since this number is relatively low, it is feasible to store every outcome in storage.

Therefore, to overcome RAM constraint on the VEX Brain (as mentioned in Section 3.1), the solutions are placed in storage and loaded in bits into RAM (see Section 5.4.3).

5.2.3 Bit-packing

The size of uncompressed text file was around 250 MB. This would take over 5 minutes for VEX Brain to parse. Therefore, both the states and the solutions were stored as two unsigned 64-bit integers using bit packing.

There exist six center colors, eight corner elements, and three possible orientations of each corner. The center colors can be encoded as numbers ranging from zero to five, corners as numbers from zero to seven and orientation as numbers from zero to two. This can all fit in a 64-bit integer ($18 + 24 + 12 = 54 < 64$) using bitwise operations to get the desired bits into each place. Similar procedure is performed to store solution.

This resulted in a file size of 40 MB which speeds up the searching process in Section 5.4.3 by up to 6.25 times.

5.3 Web Interface

The web interface (Figure 27) captures a live video stream of the Skewb. The software samples RGB values from specific regions of interest corresponding to the sticker locations. To improve consistency over variable lighting conditions, these RGB values are converted to the HSV (hue, saturation, value) color space. All colours are then classified as either white, blue, red, green, yellow, or orange.

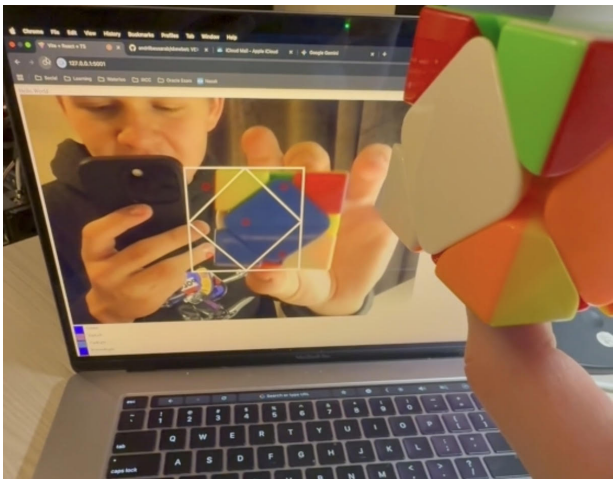


Figure 27: Web interface demonstration

When “Send Data” button is pressed, the information gets sent to a RESTful API endpoint designed in Python with Flask. Endpoint validates and sends the colour string to the Brain through serial port. The connection to serial port is established before the server starts.

5.4 VEX Program

Vex Brain acts as the main controller. It connects Solution Finder and Web Interface to the hardware. The entry point (`main.cpp`) operates as a "sequential state machine." It calls functions from various files and controls the overall flow.

Figure 28 illustrates the high-level execution flow of the system. To maintain clarity and readability, this diagram abstracts low-level command sequences, focusing instead on the logic transitions between major system states.

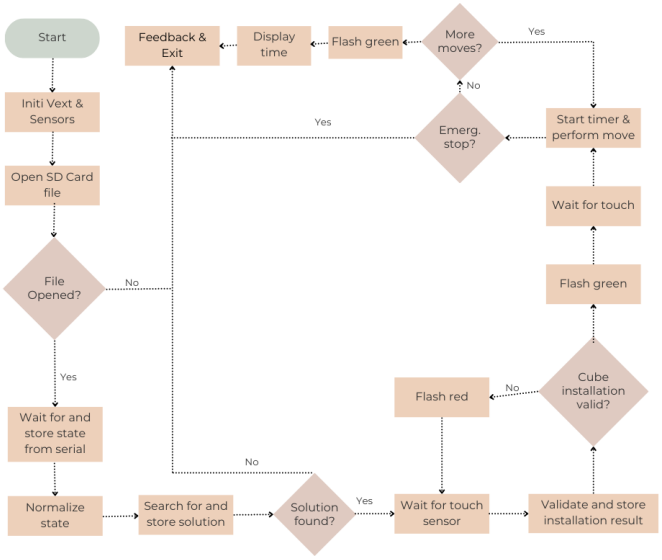


Figure 28: High-level flow chart of VEX Brain Code

5.4.1 Serial Communication

On startup, the serial port is initialized and enters a listening state, awaiting a handshake from the web interface. Due to how VEX Brain works, computer can be connected to either VEX Controller or VEX Brain directly.

5.4.2 State Normalization

This part of the program first orients colour string in a specific way and then converts it into the following C++ structure:

```
struct skewb_state
{
  std::vector<CENTER> center_colors;
  std::vector<CORNER> corner_permutations;
  std::unordered_map<CORNER, int> corner_orientations;
};
```

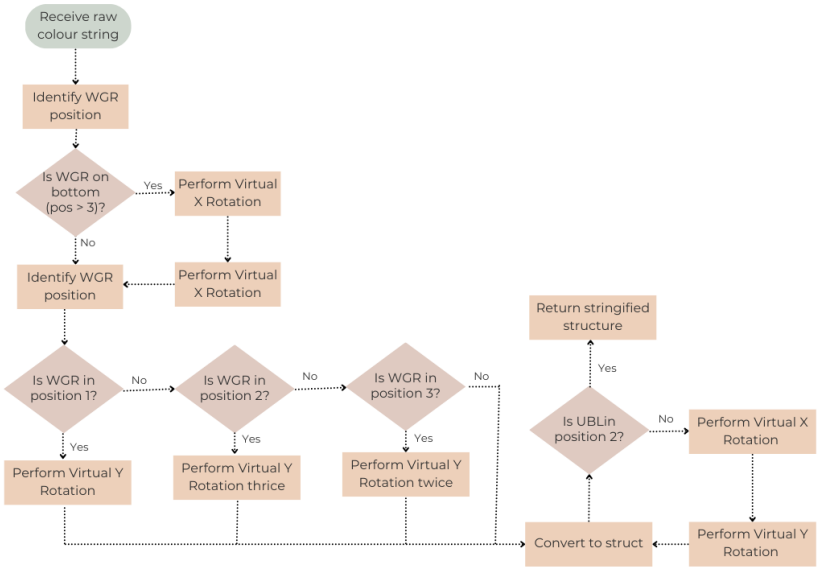


Figure 29: State normalization process flow chart

A big challenge is that the user may scan the cube in 24 different orientations (8 corners * 3 orientations per corner) relative to the camera. To be able to find the solution in the database, this state needs to be converted to the canonical (standard) state.

The Skewb has a certain mechanical invariance that is leveraged. Due to internal mechanism (rotation around four axis), any four non-adjacent corners always remain in constant special arrangement relative to each other. This is because Skewb consists of two separate tetrahedrons: one made up of four fixed corners, and one made up of four floating corners.

Due to the choice of moves in Section 5.1.1, a convenient fixed reference frame is the one that assumes:

1. Red-Green-White (RGW) corner in Up-Front-Right (UFR) position.
2. Orange-Blue-Yellow (OBY) corner in Up-Back-Left (UBL) position.

Since the spatial relation between RGW and OBY never changes, by group theory, they are mechanically invariant. Therefore, a canonical orientation can be defined by placing the pieces in specific positions (i.e., UFR and UBL).

After that the colour string is converted into a C++ structure that facilitates bit-packing process as described in Section 5.2.3. In essence, RGW and OBY are converted into UBL and UFR.

5.4.3 Matching State to Solution

Module `solution_finder.cpp` contains lookup logic. It utilizes buffer read approach to scan the 40 MB binary file on SD card without having to load the whole thing into RAM. It is read in chunks (multiples of `state_solution` structure). By utilizing bitwise operators, it unpacks data and compares it to the normalized state derived in Section 5.4.2. Once the matching state is found, it decodes the solution and returns it as a string of moves for execution.

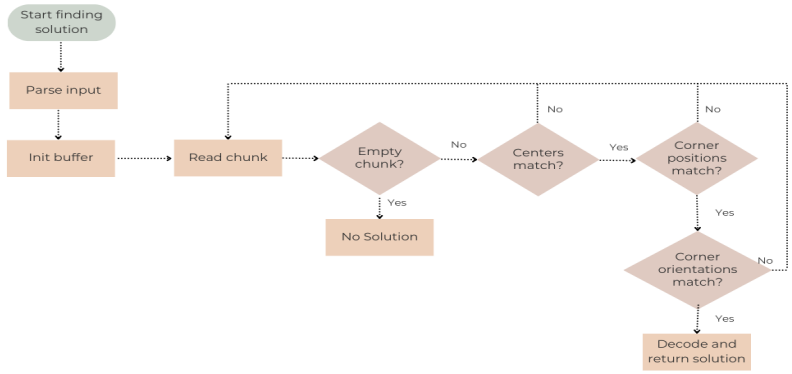


Figure 30: Flow chart of solution finding

5.4.4 Placement Validation with Sensors

Before any of the motors move, the robot performs safety checks via `sensor_validation.cpp`. This makes sure the cube is present and inserted the correct way. It must meet two conditions:

1. **Cube is present:** distance sensor checks that there is an object within 30-40mm.
2. **Cube is oriented correctly:** For reasons related to canonical state (see Section 5.4.2), optical sensor checks that hue and brightness of the center facing the sensor matches the expected center colour derived from state string.

Touch LED visually indicated whether validation passed or failed (by green and red colours respectively). If validation failed, Touch LED eventually turns white and awaits another try.

5.4.5 Solving & PID Controllers

Following validation in Section 5.4.4, pressing Touch LED begins the solve. The built in timer is used to time the solve. Pressing Touch LED during the solve, pauses motors (not the

timer however). Pressing “right arrow” button on the VEX Brain Screen terminates the solve early in case of emergency.

Unfortunately, the PID-controller built into VEX Smart Motors is too gentle to accurately solve a puzzle with friction and magnets. This required creating custom PID controllers. Inside, `skewb_motor.h, lies` a definition of a custom motor class with a VEX motor object and seven floating-point variables that hold values of PID constants and ideal position.

To combat inconsistent performance, PIDs are directionally tuned. Each motor has variables of both clockwise and counterclockwise P, I and D constants. Depending on what move is to be performed, the active P, I, and D constants switch. This allows for accurate and much more reliable performance.

The integral term is capped to avoid overflow; however, it should be noted that in the final implementation, no I controllers were used due to a lack of need.

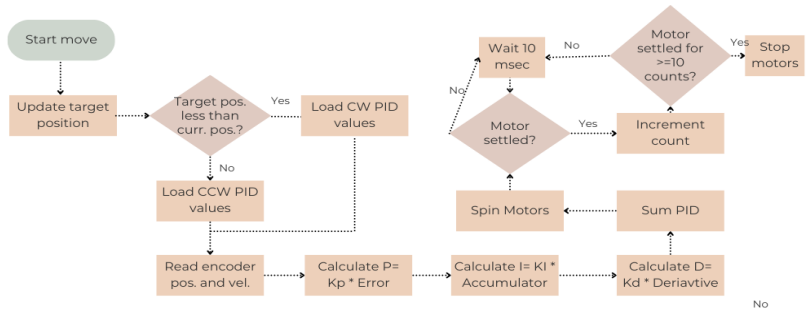


Figure 31: Flow chart of executing a move with custom PID

5.5 Demonstration Task Sequence

Below are the task lists of what was expected during demo as well as what could have gone wrong. This list has remained the same since initial design, other than the task related to web interface.

Expected task list:

- ☐ On startup, robot successfully opens SD card file
- ☐ Robot receives colour string from web interface
- ☐ Robot can convert colour string into needed format
- ☐ Robot successfully finds the solution on the SD CARD

☐ After cube installed, Robot successfully validates cube placement

☐ Robot solves cube in around 6-20 seconds

☐ Robot shows solve time on the screen

☐ The program stops 20s after solving.

Unexpected:

☐ If cannot open file or cannot find solution, Robot aborts automatically

☐ If cube isn't installed successfully, Touch LED goes red for few seconds and screen prompts again

☐ Solve can be paused/resumed by pressing on Touch LED

☐ Solve can be aborted by pressing Left Button on Brain.

5.6 Functions Breakdown

The codebase was developed in a modular fashion, with team members assigned to specific functional blocks. Table 1 details the key functions implemented, their signatures, and the primary author responsible for their logic.

Table 1: Vex functions breakdown

Function Name	Signature	Description	Author
Main / System			
main	void -> int	Entry point; implements the sequential state machine logic.	Andrii Bessarab
vexcode_init	void -> void	Initializes default VEX stuff, sets sensor light power, and configures screen font.	Andrii Bessarab
print_status	std::string -> void	Clears the brain screen and prints a status message (e.g., "Scanning...").	Jason Liang
Data & Solution			
get_state_from_serial	void -> std::string	Reads stdin for incoming color string, removes whitespace, and waits until valid data arrives.	Jason Liang
find_solution	(string, FILE*) -> string	Buffers the binary file from SD card to find the matching state and returns the solution string.	Jason Liang
Normalization			

find_normalized_stringified_struct	string -> string	Wrapper that normalizes the input string and converts it to the canonical orientation (RGW @ UFR).	Andrii Bessarab
convert_state_to_struct	const string& -> skewb_state	Parses normalized string into C++ vectors for centers and corners.	Andrii Bessarab
normalize_position	string -> string	Iteratively rotates the state string until the inertial corner is in the UFR position.	Andrii Bessarab
flip_orientation	string -> string	Flips the orientation of the state (Virtual X + Y rotation).	Andrii Bessarab
state_rotate_x	string -> string	Permutes the string indices to simulate an X-axis rotation of the puzzle.	Andrii Bessarab
state_rotate_y	string -> string	Permutes the string indices to simulate a Y-axis rotation of the puzzle.	Andrii Bessarab
get_position_of_inertial_corner	const string& -> int	Scans the 8 corners to find the index of the Red-Green-White (RGW) corner.	Andrii Bessarab
get_orientation_of_inertial_corner	(const string&, int) -> int	Determines the twist (0, 1, or 2) of the RGW corner.	Andrii Bessarab
skewb_state_to_string	(const skewb_state&, bool) -> string	Helper that serializes the struct back into a string for debugging or lookup.	Andrii Bessarab
center_to_string	CENTER -> string	Helper to convert enum to char (e.g., W -> "W").	Andrii Bessarab
corner_to_string	CORNER -> string	Helper to convert enum to string (e.g., UFR -> "UFR").	Andrii Bessarab
Sensors			
validate_cube_placement	(char, distance&, optical&, touchled&) -> bool	Loops until distance is 30-40mm and optical sensor sees the expected center color.	Robert Vigneron
Motor & PID			
calculate_pid_output	double -> double	Calculates motor velocity percentage based on error, integral, and derivative terms.	Adam Turaj
execute_move	void -> void	Manages the control loop, checking for settling conditions and timeouts.	Robert Vigneron
setPID	(bool, double, double, double) -> void	Updates the PID constants for a specific direction (Clockwise or Counter-Clockwise).	Adam Turaj
move_relative	double -> void	Updates target position relative to current and calls execute_move.	Adam Turaj
move_absolute	double -> void	Sets absolute target position and calls execute_move.	Adam Turaj

hard_reset_zero	void -> void	Resets the internal motor encoders and PID target variables to zero.	Adam Turaj

5.7 Testing and Troubleshooting

Due to the complexity of the program, software followed microservice-like architecture, allowing individual modules to be verified before full integration. Here is how each part was tested:

- Scanning (Section 5.3), normalization (Section 5.4.2), and bit-packing (Section 5.2.3) are independent of the VEX Brain, so they were tested individually on a computer first to ensure logic correctness.
- Serial communication (Section 5.4.1) was tested independently from the rest of the program and the website's frontend. POSTMAN was used to send direct HTTP requests to the Flask backend to verify data transmission.
- Solution finding (Section 5.4.3) was tested independently of the web interface. State strings were hardcoded on the VEX Brain to verify that the robot could correctly read from the SD card.
- Solving mechanics (Section 5.4.5) were tested by hardcoding a specific string of moves to verify motor performance without relying on the sensors or database.

There were numerous setbacks and issues discovered along the way. Some have been mentioned previously, such as unreliable sensors which necessitated the Web Interface (Section 5.3), insufficient storage which led to Bit-packing (Section 5.2.3), weak built-in PIDs which required custom controllers (Section 5.4.5), and inconsistencies in orientation during scanning which required the application of concepts from group theory (Section 5.4.2).

Other significant challenges involved:

- **Stream Resource Contention:** While the Serial Communication and Solution Finder modules functioned correctly in isolation, integrating them caused a massive performance degradation. The speed of reading the binary database from the SD card dropped from processing three million of entries in two minutes to only a few hundred. This was caused by resource contention, as both the Serial input (standard input) and SD card file input utilize blocking stream operations, which overwhelmed the VEX Brain's I/O scheduler. Thus, before searching SD Card, "buffer array" is defined manually and set forcefully.
- **Debug Buffer Overflow:** During the integration of the State Normalization logic, the program experienced persistent freezing at a specific iteration within a for loop in

normalization logic. After approximately 12 hours of debugging, the root cause was identified as excessive calls to `printf`. The high frequency of output requests inside the tight processing loop overflowed the output stream buffer, causing the blocking print function to freeze the thread.

- **Derivation of Mechanical Theory:** Due to the lack of technical documentation regarding Skewb mechanics, the team was required to derive many fundamental ideas from scratch. Concepts such as axis dependence, need for canonical state, and mechanical invariance were established through the team's own theoretical analysis and brainstorming rather than existing literature.

6.0 Verification

This section verifies that all constraints and criteria were met successfully on presentation day:

- **RAM Usage Under Limit:** Bit-packing (Section 5.2.3) reduced database size and buffered reading (Section 5.4.3) prevented RAM overflow. Thus, the constraint is met.
- **Mechanical Interference:** The pyramidal layout and non-adjacent gripping pattern described in section 4.2 ensured that all axes could be rotated without collisions. Thus, the constraint is met.
- **Sensor Accuracy:** Web interface was developed that eliminated the reliance on sensor accuracy (in scanning stage). Thus, the constraint is met.
- **Solve Time Under 30s:** Solve time peaks at under 20 seconds. Thus, the criterion is well-beyond met.
- **Reliability Over 80%:** Around 8 demos were shown to TAs and peers on demo day. There was one unsuccessful solve during MTE121 presentation. Thus, reliability criterion is met.

7.0 Project Plan

7.1 Time Deviations and Revisions

Figure 32 shows the project plan as of November 7th. Generally, the project adhered to the established deadlines. However, a major software revision occurred during the week of November 3rd. It was discovered that the moves, as they were defined at the time, could not be performed with the current motor placements. Therefore, the BFS algorithm had to be modified and re-run with a new set of moves.

Replacing the B (Back) move with the F (Front) move (see Section 5.1.1) presented a challenge in normalization. Previously, the UFR corner would always have the same orientation in the solution file, so the reference frame could be determined solely from the position and orientation of that single corner. The change required the reference frame to be determined from the positions of two corners (as detailed in the "Canonical State" logic in Section 5.4.2), as opposed to just one. This led to a complete rewriting of the normalization logic.

There was also a deviation from the schedule, as the Brain code was not finished by November 14th due to issues with the built-in PIDs. Custom PID controllers were implemented on November 19th and tuned on November 20th. Therefore, the troubleshooting buffer week was fully utilized.

Oct 6	**Finish Solution Generator**	• **BFS to find solutions to all states** • **Encode solutions into bin file** • **Store on SD card**
Oct 31	Finish CAD & Prototype Claw	• Finish a rough cad of the frame • First prototype of the corner gripper • Proof of concept that corners can be spun
Nov 4	Finish Scanning Web Interface	• Scan Skewb through web cam • Send scanned state to brain through controller
Nov 7	Formal Presentation	• Frame Complete • Motor mounts complete • Corner grippers complete
Nov 10	Finish Hardware	• Mount sensors • Mount brain • Add spring to top gripper
Nov 14	Finish Brain Code & Mounting Sensors	• Receive state from site • Decode solution from SD card file • Operate robot to solve
Nov 21	Demo Day	• 1 week troubleshooting period
Nov 28	Return parts	• Reassemble into default configuration

Figure 32: Project timeline

8.0 Conclusion

The team successfully designed and constructed an autonomous Skewb-solving robot. All constraints were successfully met. Despite speed constraints, the solve time exceeded expectations by always being under 20 seconds.

The innovative pyramidal layout and custom braces allowed motors to produce enough useful torque to move the puzzle. The “God’s Number” BFS and group theory-based normalization allowed to solve from any state without real-time searching.

While there were concerns related to long-term reliability of plastic parts, this prototype demonstrates that it is possible to solve complex puzzles efficiently even on limited-resource hardware through highly optimized software architecture.

9.0 Recommendations

9.1 Hardware Improvements

Majority of the issues in hardware lie in dependence on VEX parts.

- Replacing colour sensor with computer vision would remove the need for web interface and would make the robot fully standalone.
- Using higher quality DC motors with less backlash would significantly improve consistency. Additionally, ability to edit values on built-in PID would have a great effect as that would allow the controller to be updated at thousands of Hertz per second as opposed to few dozens of times.
- Instead of PLA parts, machined components should be used. Those will assure long-term reliability.
- All motors should have compressible springs to adjust to the varying sizes of Skewbs.

9.2 Software Improvements

- Instead of storing all solutions in a bin file, it would be more efficient to store it in a dedicated data structure such as hash table to increase look up speed.
- If more RAM is available, using IDA* or two-way BFS would eliminate the need for SD Card altogether.
- It would have been beneficial to design a common library since multiple data structures and methods are shared between Solution Finder and VEX Brain State Normalization Code. This would remove need to update it in two places.
- It would be beneficial to design a software calibration routine that tunes PIDs on the go and stores those values in another file as opposed to having them hard coded.
- If this was to be sold to other “cubers,” there would need to be a way to pair bots with the website and a way to communicate through Wi-Fi or Bluetooth since serial only works if server runs locally.

10.0 References

[1] "Rubik's Cube," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Rubik%27s_Cube. [Accessed 25 November 2025].

[2] "Skewb," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Skewb. [Accessed 25 November 2025].

[3] K. Gittemeier, "Skewb image," Kevin Gittemeier, April 2023. [Online]. Available: https://www.kevingittemeier.com/wp-content/uploads/2023/04/Skewb-Cam-Bro-scaled-e1681763814843-1024x1024.jpeg. [Accessed 25 November 2025].

[4] M. Hill, "Skewb Solving Robot," 2 April 2019. [Online]. Available: https://www.youtube.com/watch?v=KnrbZJdmEWg. [Accessed 28 November 2025].

[5] Ruwix , "Skewb - Beginner's Solution Guide," [Online]. Available: https://ruwix.com/twisty-puzzles/skewb-pyraminx-mod-skewb-diamond/.