Cubr: 3x3x3 Puzzle Solver

JT Aceron, Lily Chen, Sam Fazel-Sarjui

Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*— Cubr is a system capable of solving a 3x3x3 Rubik's Cube. Cubr uses computer vision to scan a real cube and map its configuration into 2D space to create a cube string. This cube string is passed into our 3x3x3 solver, which determines the optimal set of moves to solve the puzzle, known as a solution string. The solution string is then handed over to the physical robot. The robot contains six motors, one for each face of the cube, and will execute the solution string to solve the Rubik's Cube in under 20 seconds.**

***Index Terms*— Arduino, computer vision, cube configuration string, cube notation, cube state detection, motors, motor drivers, Rubik's cube, solution string**

I. Introduction

Cubr, a puzzle cube solving robot, is an approach to introduce technology to a common but sophisticated household puzzle. The motivation of Cubr is to mechanically solve a 3x3x3 puzzle cube. We constituted Cubr's solution as successful in three ways. The first metric was to correctly classify the colors of each cube piece on all six sides of the cube 100% of the time. An error in color classification would result in an invalid cube configuration string and would lead to an inaccurate representation of the real cube. Furthermore, our solving algorithm would not find a solution for an invalid cube string. The second metric was for our solver module to find a solution string of less than 300 rotation moves using an intuitive approach known as the Beginner's Method. We considered a solution string with more than 300 moves to be an inefficient solution. We also classified a solution string that does not fully solve a valid cube configuration as an insufficient solution. This constrained our solver module from being too computationally expensive and naive. With the Two-Phase Algorithm, we were able to obtain a solution string containing a maximum of 20 moves. With this, the motor arm had to turn its designated cube face in less than 1 second and be able to perform the full Two-Phase solution string in less than 20 seconds. Due to its proximity to human intuition, the Beginner's Method implementation in Cubr can be used as a learning tool. The last metric was the execution time for a motor arm to turn a face of the cube. Cubr is important in proving that technology can be introduced to everyday objects that previously have not been integrated with technology.

II. Design Requirements

In mechanically solving the cube, there are three major components that are critical to the success of our project. The first is cube state detection, which is responsible for successfully mapping the cube to 2D space. The second component is the solving software which will take in a cube configuration string and will output a solution string. The third component is responsible for mechanically turning the cube faces to physically solve the cube.

To verify that the first component, cube state detection, has met our specification, we will scan 10 differently scrambled cubes and ensure that all pieces are correctly scanned and mapped. The testing and validation will not be complete until this requirement is met. We will be scanning a standard 3x3x3 cube with the traditional sticker color scheme through an external webcam.

To ensure algorithmic efficiency of the solving software, we are first evaluating whether or not a solution is found. If a solution is not found, then the solving software is not correct, and we fail our success benchmark. We also fail our success benchmark if a solution of less than 300 moves is not found. The Beginner's Solving Method should approximately take 200 moves; therefore, we have a leniency margin of 100 moves. A solution string of greater than 300 moves indicates that our solving software is inefficient and is too naively solving the cube.

The hardware component is the most challenging for our group and requires the most attention. We will be testing the hardware components in steps. First, we want to make sure we can move a single motor with the control we desire. From that, we would like to map each possible move for each face to a keypress. If we can do this successfully, we'll be able to transition the code from reading keypress events to parsing a solution string and ultimately executing the set of moves to solve the puzzle.

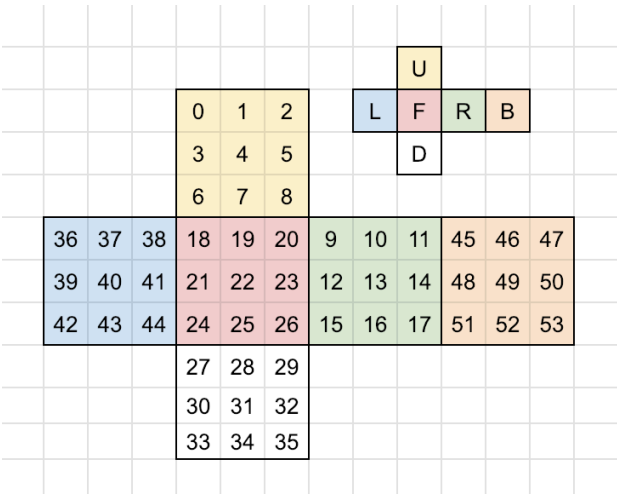

Fig. 1. 2D Mapping of 3x3x3 Cube

III. Architecture and Principle of Operation

Our solution approach consisted of three main modules to create the Cubr system. The first two, cube state detection and the solver, were software-based using OpenCV and Python. The third and final module, physical solving, was hardware-based with an Arduino as the mode of computation and control. Lastly, we needed to ensure smooth integration between all three components. The Cubr system block diagram is depicted in Figure 8.

A. Cube State Detection

The first software module in Cubr's pipeline was the cube state detection. Cube state detection, an integral part of the process, identified the given configuration of the cube and interfaced the mapped cube to our solver module. To scan all sides of the cube, a webcam was used with OpenCV to capture and identify the sides of the cube.

In the cube scanning portion of this module, there was a defined region to place a single face of the cube in front of the webcam, indicated by nine small rectangles for each cube face's cubies in the video frame. In addition, the cube state detection module featured a live color tracker in the top left corner of the window that showed the colors detected by the color classification algorithm for each cube piece of the current face. This acted as a validation step to ensure the right colors were being tracked. To capture and record the single side of the cube, the user had to press the spacebar when the cube face was aligned properly with the rectangles and the live color tracker was showing the correct colors. The software stored the colors of a given face of the cube in a data structure. The mapping of a given face was based on the color of the centerpiece as the centerpieces cannot be moved. This process was repeated for each of the six sides of the cube.

Originally, for the design review, we standardized the scanning process by partially restricting the orientation in which we scanned. To scan each face, the yellow centerpiece had to be on the top and the white centerpiece on the bottom. Then, the user had to turn the cube by 90 degrees in either direction while the white and yellow centerpieces were still oriented correctly. To scan the yellow and white faces, the user had to make another 90-degree turn along a different axis that allowed the camera to scan the top yellow face and the bottom white face.

We iterated upon this original design because when we physically place the cube into the robot housing, we have to take the centerpiece caps off the cube so that our coupling arms could attach. This is a tedious process that we would have to repeat every time we scan a cube and place it into the housing, so our solution was to do the scanning portion without the center caps. With the initial iteration of cube state detection, the module knew which face it was mapping by the color it detected from the centerpieces. Without caps, the color was not detected correctly which consequently incorrectly mapped the cube. Thus, we defined a preset way to scan and orient the cube so that we can hard code the centerpiece color. So, with our latest iteration of cube state detection, our order to scan the cube while yellow is oriented top and white oriented bottom is to scan the red face, green face, orange face, and then the blue face. With the red face-oriented front, we then rotate 90 degrees to scan the yellow face, then go the opposite direction to scan the white face. This scanning aligns with the 2D mapping of the cube as shown in Figure 1.

After all sides of the cube were scanned, the cube state detection module was responsible for properly mapping the pieces of the cube and providing a cube string with the correct notation as defined in our software interface between the cube state detection and the solver modules.

B. Solving

The second module is the 3x3x3 puzzle cube solver. From the cube state detection module, the solver module received the cube configuration string for processing. We decided to implement the solver module in Python given its readability, dynamic typing, automatic memory management, and object-oriented programming (OOP) features.

Once the solver module instantiated a Cube object and identified the individual cubie pieces and locations, the module executed either the Beginner's Method of solving, which reflects the way a human would solve a 3x3x3 puzzle cube, or the Two-Phase algorithm, which found a solution of 20 moves maximum for any valid cube state.

The Beginner's Method works by solving each layer of the cube according to its corresponding sublayer algorithm(s) and maintaining the pieces of previous layer that were already oriented in place. Since the design report, because we aimed for a solution consisting of a maximum of 300 moves, we decided to implement a hybrid algorithmic solver derived from the naive Beginner's Method and from the CFOP (Cross – F2L [First Two Layers] – OLL [Orient Last Layer] – PLL [Permute Last Layer]) speedcubing method. With this hybrid of the two solving methods, the top layer can be solved in a fewer moves than the naive Beginner's Method.

For our solver, the first two layers of the cube were solved with the naive Beginner's Method: in the first/bottom layer, the solver oriented and placed the cubies to reach the "White Cross" and then the "White Corners" states; from there, the second layer edge pieces were solved. The CFOP method typically combines the "White Corners" and second layer algorithms into a state called F2L. The third layer is solved via the OLL and PLL steps of the CFOP method.

As the module proceeded to solve the cube state by layers, it concatenated each sublayer solving algorithm to a solution string of moves. Once the cube object reached its fully solved state, the resulting solution string of moves was used to direct the hardware module for physical solving of the cube.

C. *Physical Execution*

The third and final module in our approach was the physical execution of a solution string to solve the puzzle. The cube solving robot took a single solution string as its sole input. With a valid solution string, this module read the set of instructions and performed each move serially via a configuration of bipolar stepper motors and motor drivers. Though the robot could operate on any given string, our pipeline and workflow required that the motors were only on and turning when a correct solution string was found. In any other cases, outside general testing and debugging, Cubr should never execute an invalid solution string or input.

As for the main hardware components of the robot, this module consisted of a microcontroller, six bipolar stepper motors, six stepper motor drivers, and a power source. Secondary components included a clear acrylic housing for the motors, cube coupling arms, a simple breadboard, and wires to make all the necessary connections.

The computation power of the robot came from an Arduino Uno Rev3 microcontroller. This was also how the robot communicated with our solving module to receive a solution string. Furthermore, the Arduino was responsible for controlling each of the six motor drivers in order to execute all 18 possible moves on the 3x3x3 cube. A Rubik's Cube has six faces where each face has only three possible moves. These 18 possible moves are defined in a universal cubing standard called cube notation used in official World Cube Association Speedcubing competitions.

Cubr used NEMA-17 Stepper Motors to attach to the center of each cube face with custom cube coupling arms to turn the puzzle. These motors could turn either direction, clockwise or counterclockwise, at any number of steps between 0 and 200, where 200 steps was a full 180-degree revolution. To drive and move these stepper motors, Cubr utilized A4988 Stepper Motor Driver Carriers. These drivers allowed the Arduino to interface with the stepper motors without the need to write excess and pre-existing code or to implement third-party libraries.

As for the physical infrastructure of the robot, more thought and consideration needed to be placed in actual integration of all the hardware components since the design review. All stepper motors and drivers were connected using a basic breadboard. Extra steps needed to be taken to ensure power and current draw was not too high while supporting the function of all six motors. Being that NEMA 17 stepper motors continuously pull current, even in a dead state, Cubr needed to properly limit and monitor current through each driver. Our breadboard recommended a maximum of 0.5 Amps and had heat limitations. Using the potentiometer on the motor drivers, we limited the maximum current through any motor. In order to avoid melting the breadboard, thicker 18-gauge wire was used to branch the power from a protoboard to the breadboard containing the rest of the components.

Lastly, the housing mechanism was properly measured and cut with respect to the motors, coupling arms, and the cube in order to perfectly hold the cube within turning distance without putting extra load on any of the stepper motors. Since motor arms were attached to the cube on every side, it was important that our design allowed flex so that the cube could be placed and removed easily while still maintaining enough structural pressure to properly turn a cube face.

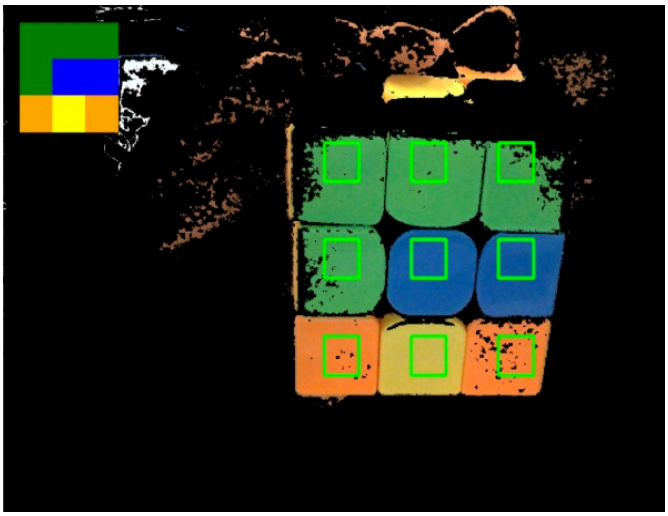

Fig. 2. Cube State Detection Interface

In summary, one driver was used for each stepper motor, and all stepper motors directly communicated with the Arduino to receive direction and step instructions through the Arduino's digital input/output pins. The stepper motor and driver setup were repeated six times in parallel to the power supply to create the core base of the robot. From here, each setup was positioned into a housing that allowed each stepper motor to perpendicularly connect with each of the six faces on the puzzle. To turn the face of the cube, custom 3D-printed coupling arms were created to attach to the stepper motor and center piece of each cube face.

D. *Integration*

The cube itself was first scanned in front of the webcam for our cube state detection module. A key change since the design review is that we now scan the cube without its center pieces in. Because of this we know have to predefine the scanning order. Previously, we allowed the user to scan any side of the cube as long as they adhered to the layout of the cube in a two-dimensional space. This change allows for fast integration when moving the cube into the robot. From cube state detection, a cube configuration string is outputted to STDOUT. From this, the solver module read this cube string as input and created a solution string as its output. The Arduino Uno Rev3 received the solution string and parsed each move in the solution string in order to communicate with the appropriate motor driver in controlling the motor arms to physically solve the cube. After it was scanned, the cube was placed in the housing with the correct orientation. As mentioned in the previous section, we predefined this as the yellow side facing upwards, and the red side facing forwards. Once the cube was firmly attached to the coupling arms, we started the physical solving process on the Arduino.

	Color Classification & Thresholding	Naive Beginner's Method	CFOP Method	Our Beginner's Method	Two-Phase Algorithm	Turn Speed (90 degrees)	Turn Delay
Expected Performance	100% success	150 moves	57 moves	100-200 moves	≤ 20 moves	1 sec	2 sec
Actual (Average) Performance	95% success	120 moves	87 moves	84.4 moves	≤ 20 moves	0.065 sec	0.5 sec

Fig. 3. Summary of Solver and Motor Metrics

IV. Design Trade Studies

A. Cube State Detection

In designing a software solution for cube state detection, our top priorities were to have the highest accuracy in color detection and to design the most lightweight solution so that we could spend the majority of our efforts on the cube solving robot. The first design tradeoff for cube state detection was to use OpenCV for color detection as opposed to using color sensors. We chose this option as there is less physical moving parts needed in OpenCV, a computer vision library. It was also easier to customize our cube scanner to account for different cubes and their color profiles.

For validation and testing, we scanned all sides of 10 differently scrambled cubes and recorded the number of centerpieces, edge pieces, and corner pieces scanned correctly. Since we used a sticker less cube for the entire system, the cube state detection module was able to accurately classify the colors of the center pieces. We noted which colors were incorrectly scanned against which colors were correctly scanned to get a percentage of the colors that were misread.

B. Solving

The main priorities of the solver software were to correctly reach a fully solved state with a solution string of less than 300 moves. Due to this, we decided to implement a hybrid algorithmic solver derived from the naive Beginner's Method and from the CFOP (Cross–F2L–OLL–PLL) speedcubing method. The naive Beginner's Method requires greater number of moves to solve the top layer than typical speedcubing methods such as the CFOP method, since it permutes the top layer with the same set of moves until it eventually reaches the solve state. The hybrid of the two solving methods, naive Beginner's and CFOP, resulted in the top layer being able to be solved in a fewer moves than projected.

Invented by Professor Jessica Fridrich, the Fridrich Method or (commonly known as) the CFOP Method is one of the most popular solving methods for the speedcubing sport. The steps for solving the top layer are the OLL and the PLL. There are 57 different orientation cases for OLL to orient both the edges and the corners of the top layer at once. There is a simpler sub-method known as the “2-Look OLL” of which there are 10 different orientation cases and orients edges separately from the corners; however, like the Beginner's Method, the 2-Look OLL works by permuting the cubies with the same algorithm several times until the desired state is achieved, which would have resulted in a longer solution string. The last step is PLL, which does a single permutation of the pieces with a unique algorithm for each of the 21 orientation cases.

Solving a cube by hand, in 3D space, typically allows for slice turns, or middle layer rotations, two-layer turns, and whole cube rotations. The translation of these types of turns into simple face rotations for our virtual and physical solver proved to be a hefty challenge. The reason we could not utilize these types of moves was due to the structure of our housing: since the motors were mounted in place to a static housing, the motors were unable to execute middle layer, two-layer, and whole cube rotations. Consequently, we were limited to only implementing face rotations. For example, say the algorithm called for a 90-degree clockwise turn of the bottom two layers and then a 90-degree counterclockwise turn on the resulting Right face. This would translate to a 90-degree clockwise turn of the top layer (the Up face) and a 90-degree counterclockwise turn on the Front face. Translations of the special moves extended each into 1-3 face rotation moves, which kept our solution string small. Unfortunately, due to the requisite spatial awareness skills needed for flawless translation and the sheer number of orientation cases for both OLL and PLL, we encountered many difficulties and bugs with those translations. We often had to sacrifice limiting the number of moves for ease of translation of longer but less spatially difficult move sets.

In addition, we implemented a random scramble function so that we could run our solver on randomized cube states and to aid in debugging sublayer algorithms. To gather metrics, we ran the random scrambler and solver approximately 300 times. For the first layer, our implementation averaged around 22.6 moves to solve both the "White Cross" and "White Corners" states. The second layer consisted of an average of 36.4 moves. The third layer, based on CFOP's OLL and PLL, averaged 25.4 moves. Furthermore, we compared the average solution string length from our solver to the number of moves usually taken for fully solve a cube with both the naive Beginner's Method and the 2-Look version of the CFOP method (2-Look OLL and 2-Look PLL, as depicted in Figure 3. As shown in the chart, our implementation resulted in a smaller solution string than that of the naive Beginner's Method and approximately the same of the 2-Look version of CFOP. It is important to note that variations in the number of moves recorded for the naive Beginner's and the 2-Look CFOP methods could be attributed to human

intuition, spatial awareness, and error of sorts. In the end, we successfully reached our goal of maintaining solutions strings of both less than 200 and 300 moves.

C. Physical Execution

As shown in the design requirements, we needed a fast and reliable method of physically solving a 3x3x3 Rubik's Cube. We chose to configure the robot with six bipolar stepper motors, one for each face. Having a stepper motor for each side of the cube allowed for the fastest solve time as no moves were wasted to reposition the cube. Other methods such as the claw and gripper method for physically solving the robot required less motors but necessitated more fine-tuning and precision of the gripping movements. On top of this. the solve time would have been significantly slower due to no direct connection with each cube face. However, our method required the most moving parts and ultimately had more power requirements. To address these new constraints, a 24V 5A Power Supply was used to power all stepper motor and driver setups in parallel. This power supply was specifically chosen to operate at the higher voltage range for each of the six A4988 drivers while still providing enough current for each of the motors. Since each of the stepper motors operate at 350 mA, the power supply needed a total of at least 2.1 A to supply adequate current to all 6 motors simultaneously since the motors pulled current while stationary. A significantly larger power supply on a breadboard required extra steps to ensure safety of operation. The breadboard in use for Cubr is a basic one and recommends a current limit of no more than 0.5 A at any given time. The A4988 drivers came with a built-in potentiometer that allowed us to control the amount of current going through each of the motors. We used the following equation to calculate the current through each of the stepper motors:

$$I = Vref * 2 * 0.7 \quad (1)$$

In our robot, we set the Voltage Reference value on the A4988s to be 0.5 V. Theoretically, current through each stepper motor should be 0.7 A. However due to different manufacturing standards, we found the actual current value to be 0.5 A as desired.

The NEMA-17 Stepper Motors are rated for 12 V and operate with a maximum speed of 600 rpm. To operate at its higher end speeds, we introduced the use of A4988 drivers. These drivers allowed us to push each stepper motor's voltage rating to achieve higher step rates through adjustable current control. These drivers also aided in directly communicating to the motors through the use of two digital input/output pins from the Arduino for each driver-motor configuration. With this setup during the design review, we were able to achieve an operating speed of 1 cube turn per second at the very minimum. We tested our robot's turn speed with a baseline solution string of 20 moves. Since the Two-Phase Algorithm could output solutions around 20 moves or less, we wanted to see how fast we could execute 20 sequential moves. Since the design review, significant progress had been made in regard to the speed of the robot. Stepper motors operate in steps, and the speed of these steps was dictated by how fast or slow the delay was between each step. The delay was so small the delayMicroseconds() function was used instead of the traditional delay function. Our final project was able to achieve a 90-degree turn speed of 0.065 seconds and 180-degree turn speed of 0.13 seconds, as shown in Figure 3. Slower delays, and thus faster turn speeds, were only possible with larger power supplies. An important factor of properly turning the cube fast was that not all speeds were compatible with all cubes. All cube brands have different tensions and rigidities. Turning too fast can result in overshooting and ultimately disrupting the flow of the robot. The speed of a standard 90-degree turn was calculated as:

$$microsecond\ delay * 100 = 90\ degree\ turn\ speed \quad (2)$$

D. Physical Infrastructure

We choose to use a basic breadboard to combine all of our physical components for multiple reasons. While soldering to a protoboard would have been more beneficial for the consistency and longevity of connections, our team lacked the time and experience to execute this. Protoboards also have a higher heat capacity and permits much larger currents. While this would have been beneficial for the robot, these features were neither absolutely necessary nor worth the time tradeoff. With many other moving parts and components, the last thing we wanted to worry about was ensuring proper soldering peaks and the correct amount of lead. A breadboard was best for quick connections and consistent experimentation. All three team members are software-focused and thus needed the buffer and affordability that a basic breadboard provided.

Multiple housing options existed to contain the cube, but our first choice was to laser cut clear acrylic. Both ⅛ and ¼ inch acrylic were rigid enough to hold each of the six motors and the cube. 3D printing an entire housing system required substantial 3D modeling and time to actually print the parts. As an alternative, laser cutting allowed the team to work efficiently and quickly while also be able to work with completely transparent materials, something 3D printing cannot do. It was an option to create a basic housing structure with wood. Wood framework would have created a studier and more robust house, but lack of tools and precision would have created an unpolished final product. Laser cutting was easy to design for and was cost effective.

V. System Description

We developed Cubr for Unix-based systems. However, the execution of our software is platform agnostic. This allows for future work in migrating our software to different operating systems or for others to download our repository and create or modify the robot for their personal uses.

A. Cube State Detection

For the cube state detection subsystem, the modules required were OpenCV 4.0.0 and Python 3.6+. The cube state detection software read each frame in the live video stream from a Logitech C270 webcam using OpenCV. Since OpenCV's default color space is BGR (blue, green, red), we converted each frame into the HSV (Hue, Saturation, Value) color space due to its robust lighting invariance for color detection.

Sampling regions were indicated in the web frame with small green rectangles. Within these regions, the HSV values were averaged frame by frame for the live color tracker, which

showed what colors were detected for each cube piece. The spacebar key event captured a live video frame and mapped the side of the cube in an internal data structure. In the color detection pipeline, after the HSV values were averaged, the L2-Norm distance function was employed to find and classify the closest color detected in those regions. We calculated the HSV value of a color by taking the midpoint of the non-overlapping HSV lower and upper bounds set for each color.

With cube state detection, we found it difficult to classify each color correctly with any type of variance in lighting, which included glare from the cube itself. We remedied this issue by executing the cube detection in a lighting soft box to standardize the lighting for scanning the cube.

As another precautionary measure, the green rectangles in the web frame, as shown in Figure 2, were adjusted to be smaller. This adjustment made it easier for the user to align the cube in front of the webcam. The smaller the sampling area, the less variance of color, which improved the accuracy of the cube state detection. Showing a live color tracker also shifted the error towards the user. This helped assure high accuracy readings.

B. Solvers

For the sake of continuity, the solver module was built on Python 3.6. All Python libraries used in the solver are part of the Python Standard Library and thus require no additional installation. The general structure of the code followed the object-oriented programming paradigm. After the solver module received the cube configuration string from the cube state detection module, the solver module instantiated a Cube object for the given cube state and identified the individual cubie pieces and locations prior to executing the solving algorithms.

With an object-oriented structure, we were able to create class instantiations for each input configuration string. Taking an OOP approach allowed us to view and identify the cube based on the individual cubies, or pieces, instead of by colors; doing so made it much easier to find cubie locations during each sublayer algorithm for orientation and placement. Furthermore, OOP made it easier to maintain and modify our existing code such that we could run both our hybrid implementation of the Beginner's Method and the pre-existing Two-Phase Algorithm on the same cube states. These different method calls worked with the basic but concrete structure, types, and properties for our modular OO classes.

The hybrid Beginner's Method (White Cross, White Corners, Second Layer, OLL, and PLL) solved each layer of the cube by applying a given algorithm to each sublayer and maintaining the pieces already in place. Each sublayer-solving algorithm consisted of a multiset of the 18 types of face rotation moves: for each face, there were turns of 90 degrees clockwise, 90 degrees counterclockwise, and 180 degrees. Since our hardware could not execute middle layer, two-layer, and whole cube rotations, these types of moves were translated to face rotation moves. As the module solved the cube state by each (sub)layer, it concatenated each sublayer-solving algorithm's list of moves into what we referred to as the solution string.

The Two-Phase Algorithm outputted a solution string of a maximum of 20 moves. In its initial run, the algorithm had to produce tables for memoization that amounted to

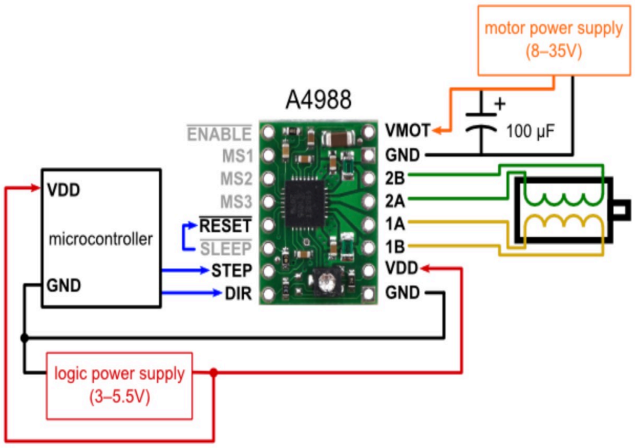

Fig. 4. Minimal Wiring Diagram for the Driver-Motor Setup

approximately 140 million lookup references at 13 different depths. In addition, during subsequent runs of the Two-Phase Algorithm, the algorithm did return at the first solution found; instead, Two-Phase would continue its search for shorter solutions from suboptimal parts of the prior solution. Moreover, the Two-Phase Algorithm used mathematical properties of the 3x3x3 to find and perform symmetry reductions.

Once the Cube object reached its fully solved state, the resulting solution string was used to direct the hardware module for physical solving of the cube.

C. Physical Execution

The NEMA-17 Stepper Motor is a four-wire bipolar stepper that rotates 1.8 degrees per step. The motor operates at a maximum current of 350 mA and operates with a power rating ranging of 8-35 V. 200 steps are required to make one full rotation. Notable step inputs include 100 steps and 50 steps to execute 180-degree and 90-degree turns respectively. These motors provided ample torque to smoothly turn a single cube face.

The A4988 Stepper Motor Driver Carrier is the driver of choice to interface between the NEMA-17 motors and the Arduino. The driver allows for a simple step and direction control interface while having five different step resolutions, though the Cubr robot will operate at the full-step resolution. These drivers also have adjustable current control, which allows for the use of higher voltages above the NEMA-17's rated voltage to achieve higher step rates. The A4988 has two pins called "STEP" and "DIR": these were the pins that were each directly connected with one of the 14 digital input/output pins on the Arduino. Because there were six setups, Cubr used 12 Arduino digital I/O pins.

These parts in conjugation as depicted in Figure 4 allowed for control over a single NEMA-17 stepper motor. This setup was repeated six times to create the core of the cube solving robot.

Wrapping up this project, we found that all physical components worked very well achieved significant performance increases since the design review. However, there were a few unforeseen and important problems that needed to be addressed. The NEMA-17 stepper motors worked as expected in conjunction with the A4988 stepper motor drivers. Full step resolution always turned the motors and cube with

Product	Where to buy	Quantity	Price per unit	Shipping fee	Total Price	Purchase Date
NEMA 17 Stepper Motors	https://www.adafruit.com/product/324	12	$ 12.60	$ 7.81	$ 159.01	Feb 11
A4988 Stepper Motor Driver	https://solarbotics.com/product/51090/	12	$ 6.49	$ 19.14	$ 97.02	Feb 11
12V 2A Power Supply Adaptor	https://www.amazon.com/TMEZON-Power-Adapter-Supply-2-1mm/dp/B00Q2E5IXW	1	$ 7.99	$ -	$ 7.99	Feb 11
Arduino Uno Rev3	https://store.arduino.cc/usa/arduino-uno-rev3	1	$ 22.00	$ 4.37	$ 26.37	Feb 11
24V 5A Power Supply Adaptor (alternative)	https://www.amazon.com/ALITOVE-100-240V-Adapter-Converter-5-5x2-1mm/dp/B01GC6VS8I/ref=sr_1_17?keywords	1	$ 19.99	$ -	$ 19.99	Feb 20
Logitech - C270 (Cheap webcam alternative)	https://www.amazon.com/Logitech-Widescreen-designed-Calling-Recording/dp/B004FHO5Y6/ref=pd_lpo_vtph_147_b	0	$ 19.86	$ -	$ -	Feb 19
Lighting Box Tent Kit	https://www.amazon.com/BrightBox-Portable-Photo-Studio-Light/dp/B01N75CIVP	0	$ 21.99	$ -	$ -	Feb 19
Valk 3	https://www.thecubicle.com/collections/3x3/products/valk-3	2	$ 19.99	$ 3.22	$ 43.20	Feb 11
Mofang JiaoShi MF3RS2	https://www.thecubicle.com/collections/3x3/products/mofang-jiaoshi-mf3rs2	2	$ 7.99	$ 3.22	$ 19.20	Feb 11
MoYu WeiLong GTS2 M	https://www.thecubicle.com/collections/3x3/products/moyu-weilong-gts2-m	1	$ 25.99	$ 3.22	$ 29.21	Feb 11
Breadboard-friendly 2.1mm DC barrel jack	https://www.adafruit.com/product/373	3	$ 0.95	$ 10.13	$ 12.98	Feb 11
3D Print Coupling Arms (prelim)		4	$ 0.90	$ -	$ 3.60	March 27
3D Print Coupling Arms (final)		5	$ 0.90	$ -	$ 4.50	April 8
3D Print Coupling Arms (final)		4	$ 0.90	$ -	$ 3.60	April 10
VACASSO Acrylic Paint Set (12 Vivid Colors)	https://www.amazon.com/dp/B07FVVG29K/ref=cm_sw_em_r_mt_dp_U_80jTCbRFX3G3N	1	$ 8.99	$ -	$ 8.99	April 15
12x24x1/8" Acrylic Sheets	from Makerspace	2	$ 8.00	$ -	$ 16.00	April 20 & 24
18 Gauge Wire	https://www.amazon.com/gp/product/B01LH1FQJ0/ref=ox_s	1	$ 16.99	$ -	$ 16.99	April 24
Jumper wire kit	https://www.amazon.com/gp/product/B06XRV92ZB/ref=ox_sc_act_title_5?smid=A2QLFR4HNCTTLU&psc=1	1	$ 7.49	$ -	$ 7.49	April 24
Cable sleeves	https://www.amazon.com/gp/product/B07JJCLJ1Q/ref=ox_sc_act_title_4?smid=A2GQEFKYWQUJT0&psc=1	1	$ 6.95	$ -	$ 6.95	April 24
Wire pins	https://www.amazon.com/gp/product/B0774NMT1S/ref=ox_sc_act_title_3?smid=A1SFG00GMDVKI5&psc=1	1	$ 9.99	$ -	$ 9.99	April 24
Protoboards	https://www.amazon.com/gp/product/B00SK8QR8S/ref=ox_sc_act_title_2?smid=AM0JQO74J587C&psc=1	1	$ 13.52	$ -	$ 13.52	April 24
Screw terminal	https://www.amazon.com/gp/product/B071DN7GQ5/ref=ox_sc_act_title_1?smid=AB9370916I9PH&psc=1	1	$ 8.99	$ -	$ 8.99	April 24

Fig. 5. Cubr Bill of Materials

proper torque, assuming the power supply was large enough. This is where the bulk of our last hurdles were. Powering six motors at once on a basic breadboard with a 24V 5A power supply can lead to significant heat problems. Though the circuit was properly connected and matched our schematics, the power draw was too hot for the plastic on the breadboard. As a result, the power supply would generate enough heat to melt the thin 22-gauge wires and eventually parts of the breadboard it was connected to.

With our impending deadline, our short-term solution was to separate the power supply from the breadboard entirely. We purchased much thicker 18-gauge wires and connected them to a breakout protoboard that had the sole purpose of containing the heat from the power supply. By soldering screw terminals to a breakout board, we could use thicker to also help mitigate the melting. 18-gauge wire was used to connect both the power supply to the protoboard and protoboard to the breadboard that contained the rest of the circuit. This configuration handled our heat problems and allowed the robot to continue to operate at higher voltages and faster speeds.

VI. Project Management

A. Schedule

Cubr has three major parts that are modular. The modularity permitted the team to work simultaneously without blocking one another in the production pipeline. Each of the three major components had a well-defined input and output. Working within these constraints, the team was able to make progress on their respective tasks without waiting on a task outside of their module to be completed. However, modularity was lost once all three tasks are completed, at least on a rudimentary level. As the team converged, the bottlenecks rose from assembly-related tasks such as construction and laser cut time. The team schedule is located on page 10, Figure 8.

B. Team Member Responsibilities

As depicted in Figure 8, JT was in charge of items in red, Sam had the tasks in blue, Lily performed jobs in green, and multiple team members collaborated on the yellow assignments. While we were all in charge of specific components, all three members had a responsibility to understand every aspect of the project.

JT was primarily focused on everything related to firmware programming and hardware design for the physical robot. At the same time, he worked on housing assembly and hardware integration and testing. In conjunction, JT assisted Sam with refining the cube state detection. JT also provided insight on speedcubing specifics and tricks for the solver module.

Lily was focused on writing and debugging the Beginner's Method solver. She led the design of the 3D-printed coupling arms and the laser-cut housing models. After designing the housing, she also assisted with housing assembly and hardware integration and testing. In addition, Lily was in charge of the soldering and 3D-printing for this project.

Sam's primary role was the oversight and development of the cube state detection and software integration between the modules. Upon its completion, Sam joined the rest of the team in addressing problems in the hardware integration phase and was in charge of the laser cutting. In addition, Sam helped with improving and organizing the overall software design.

C. *Budget*

From Figure 5, blue purchases were considered to be priority and essential purchases; orange were alternative purchases that were made to better fit the project as its needs were further realized. Light blue items were secondary purchases that were not essential in the ongoing production of the project but had purpose in our final product. For instance, we upgraded to a new power supply that had a higher voltage and current rating to better accommodate the power requirements of the parallel six stepper driver-motor setups. In the latter half of the project, we found we were not touching our budget as much since we already had all the necessary core components. The majority of purchases made in the second half were quality of life changes in order to address our heating issues and cable management.

D. *Risk Management*

To manage risk as much as possible, the team front loaded most of the design and solution approach for a better part of the first three weeks. Having a team comprised of software-focused engineers, we knew we had to plan out as much as possible in order to address our weakness in hardware design and execution. Everyone on the team was more than capable of accomplishing the software goals with high fidelity. However, none of us worked extensively with robotics or component-level hardware beyond the required courses for our undergraduate studies. JT was in charge of research and designing as much as he could before the team started to assemble physical pieces. We also defined clear modules that we assigned with well-defined inputs and outputs. This permitted us to ensure smooth integration between all three modules and adherence to the rules we defined at the beginning of the semester. These clear definitions also allowed other members to lend their efforts towards the robot or other lacking modules after certain software milestones were completed.

From a schedule and communication standpoint, it was imperative that we were upfront and communicative. Slack and other team collaboration tools were constantly in use to provide timely status updates and to serve as a place for questions and clarifications about aspects of each module between the team members. As shown in our schedule, Figure 7, we broke down tasks into their simplest forms possible. This let us too quickly see what was to be done and what needed work each week. Thanks to our solid planning, we were able to purchase most of the necessary tools and equipment in the first two weeks of the project. At the time of the design review, we spent $415 out of the budgeted $600. Since then, our total amount spent came out to $515.59.

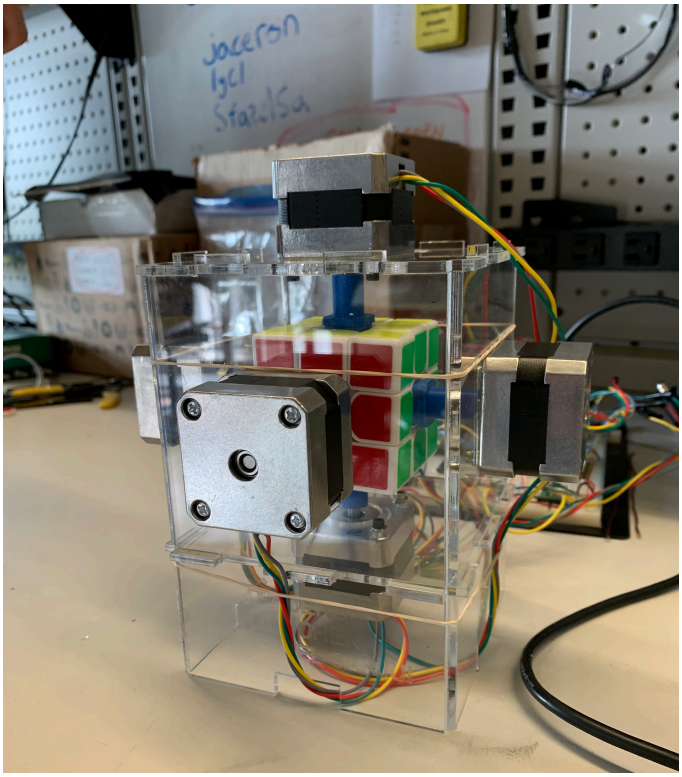

Fig. 6. Finished Cubr Robot

The primary risk of the project was the physical construction of the robot module. With no real experience on the team, there were a lot of obstacles and problems that required attention from all three team members. We opted for a six bipolar stepper motor configuration because we felt it best suited the requirements of the capstone while being within the reach of a group of hardware novices. The use of off-the-shelf parts, such as the motors and drivers, offloaded a lot of the tuning and precision of robotics onto the hardware. The main unknown secondary risk came with the construction and assembly of hardware as mentioned in section four, subsection C in regard to heat and power management.

VII. Related Work

A research team at Massachusetts Institute of Technology developed a similar robot to our project. Using more advanced motors, they broke the world record 0.637 seconds set in 2016 with an incredible solve time of 0.38 seconds[1]. Our fastest solve time was approximately 5 seconds, but this was due of the physical limitations of our motors and housing structure.

VIII. Summary and Future Work

In short, our system was able to reach and exceed our design specifications in both software and hardware modules. The future of Cubr is not set in stone, while we do not plan on working on extra features in the near future, it is definitely

something we wish to consider in our free time. JT and Sam will be taking the physical robot home to reassemble and Lily plans to improve her solver. However, if we had more time, we would certainly like to make some major changes. First, the largest bottleneck in the entire Cubr pipeline comes from the cube state detection. It simply takes too long to scan all the sides then carefully place the cube in the housing. We would like to configure cameras or color sensors around the actual robot so that it can be scanned inside the housing all in one go. Cubr's user interface and experience is also severely lacking. Had we more time, we would have liked to implement a more interactive and informative user interface while also improving the educational components of the project. Lastly, we had lack to transfer all of the physical components to a protoboard for permanent connections and greater heat management.

References

[1] MIT Office, "Featured video: Solving a Rubik's Cube in record time", *MIT News*, 2019. [Online]. Available: http://news.mit.edu/2018/featured-video-solving-rubiks-cube-record-time-0316.

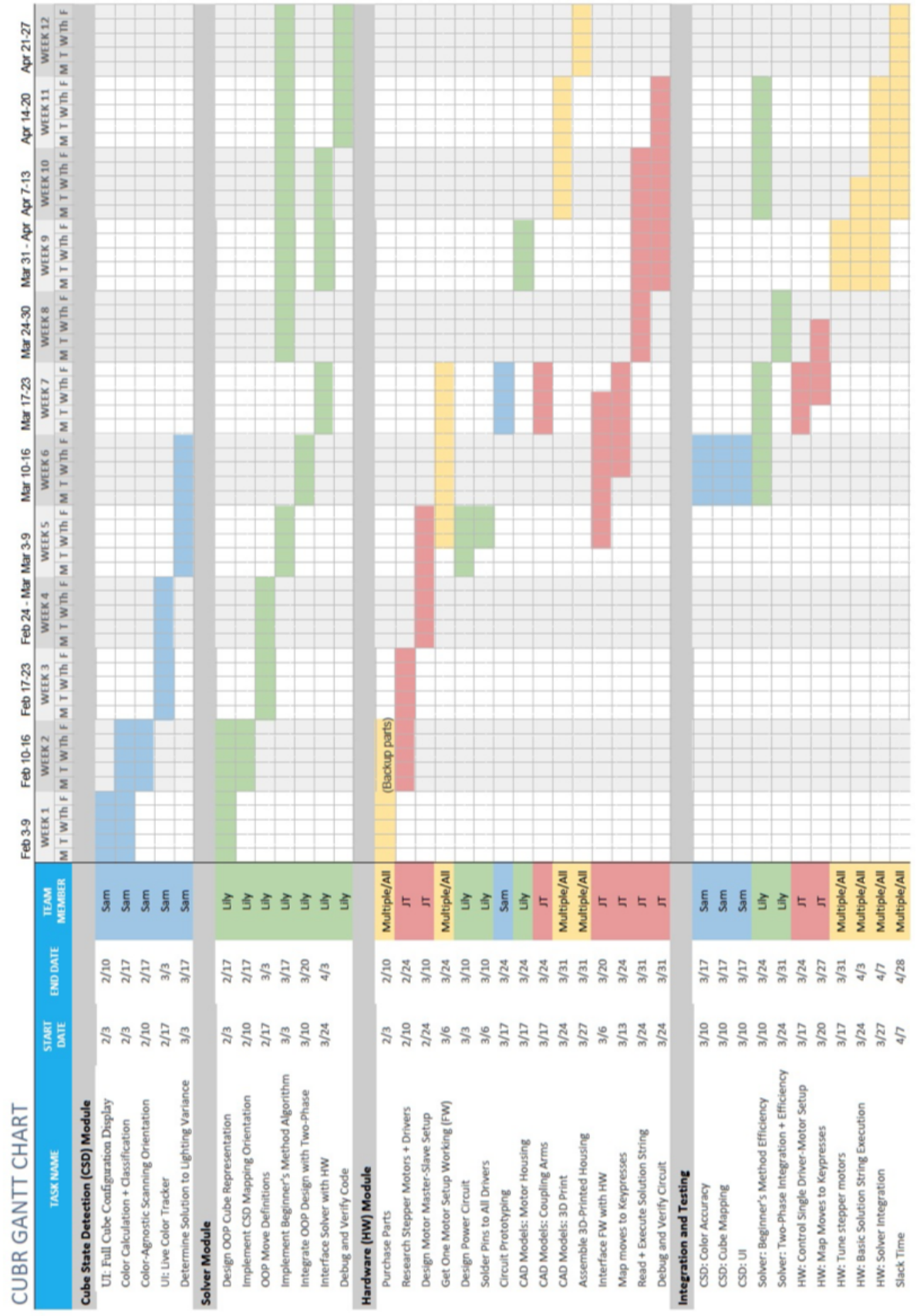

Fig. 7. Cubr Detaled Gantt Chart

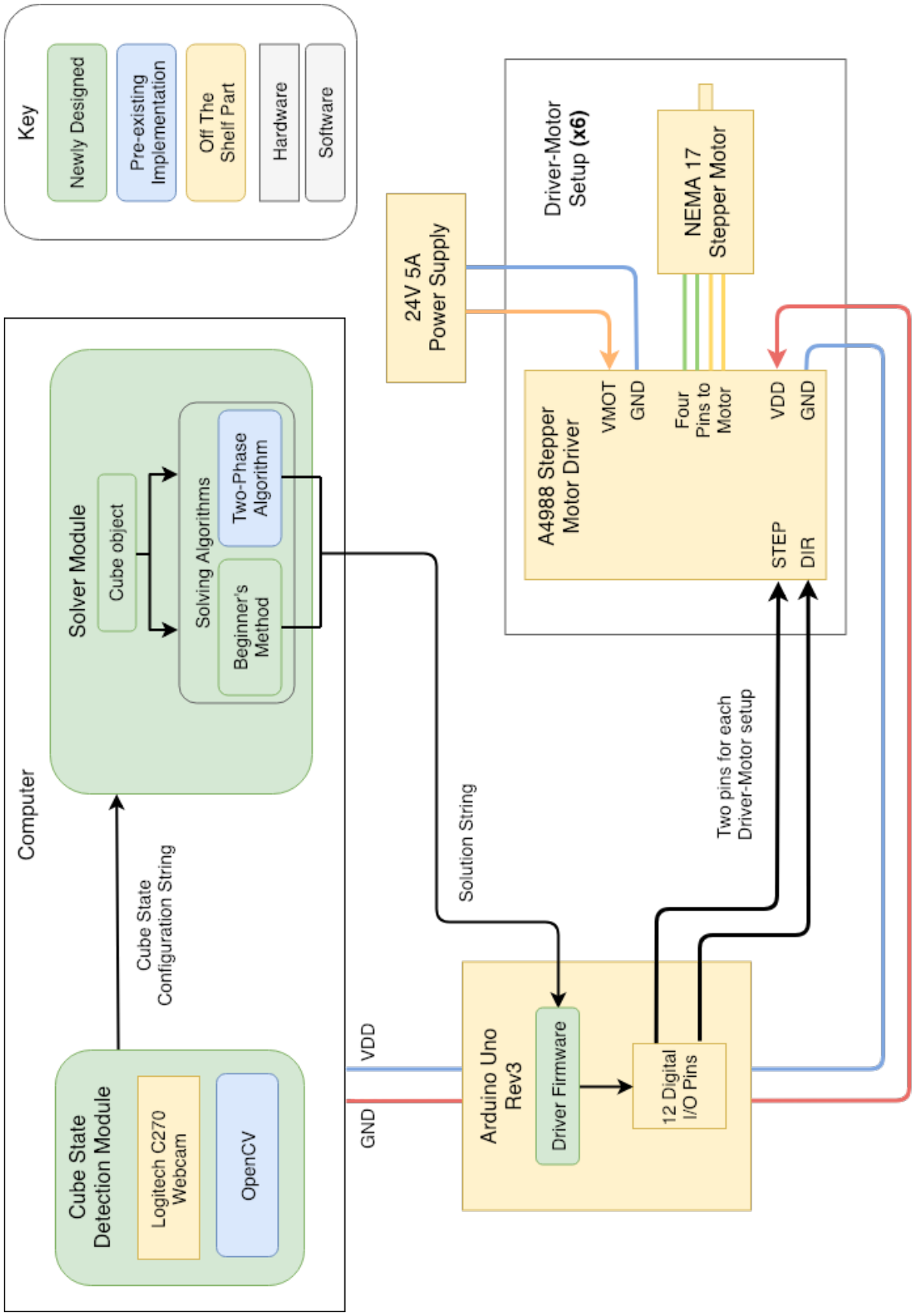

Fig. 8. Cubr Solution Approach Block Diagram Overview