

Rubik's Cube Solving Robot

Jacob Swiezy, Nathaniel Knopf

6.111 Fall 2017

Abstract

Since its invention in 1974, the Rubik's Cube has challenged users to solve a colorful puzzle in record time. While humans have managed to solve the puzzle in as little as 4.69 seconds, robots are able to do so in under a second. This seemingly impossible puzzle can be solved amazingly quickly through the use of algorithms - sequences of moves that move specific pieces of the puzzle from one location to another.

We developed a Rubik's Cube solving robot that implements such algorithms on an FPGA. The FPGA interfaces with two RGB color sensors to sequentially observe each sticker of the Rubik's Cube in a scrambled state. We implemented a method for taking this raw data and translating it into a representation of the scrambled state of the puzzle. We then implemented an algorithm that takes as input the scrambled state of the Rubik's Cube and outputs a sequence of moves that solve the Rubik's Cube. This output is translated into a series of instructions for six stepper motors that interact with the Rubik's Cube. Each stepper motor, controlled by stepper motor drivers connected to the FPGA, turns a face of the puzzle. The stepper motors execute the moves produced by the algorithm for solving the Rubik's Cube, fully solving the puzzle.

Contents

- Motivation
- Introduction
- Bill of Materials
- Modules
- Hardware
 - Stepper Motors
 - The Frame
 - RGB sensors
- Challenges and Concluding Thoughts
- Appendix - Block Diagram

Motivation

Since its invention in 1974, the Rubik's Cube has challenged users to solve a colorful puzzle in record time. While humans have managed to solve the puzzle in as little as 4.69 seconds, robots are able to do so in under a second¹. However, many existing solutions are implemented with high level programming languages. Even one example found implemented on an FPGA involved first programming an entire processor on the FPGA, and then writing C code to compile on it.

The goal of this project was to create a robot that could determine the state of a scrambled Rubik's Cube and then solve it, all implemented on an FPGA.

Introduction

The design of the robot was created in house, and consisted of 6 stepper motors and stepper motor drivers, two RGB color sensors, a laser cut acrylic frame, 3D printed parts to allow the stepper motors to actuate the Rubik's Cube, a power supply, and a Nexys 4 FPGA. The design of the robot, including a solved Rubik's Cube, can be seen in Figure 1 below.

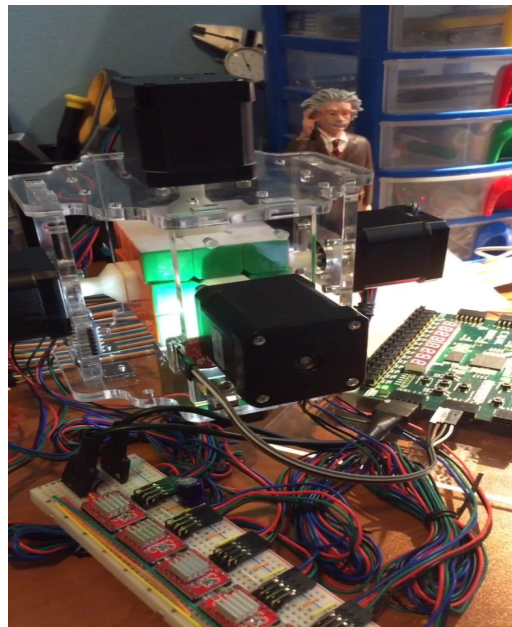


Figure 1 - The fully assembled robot, interfacing with a Rubik's Cube

¹ <https://www.youtube.com/watch?v=ixTddQQ2Hs4>

The initial state of the Rubik's Cube was determined through the use of 2 RGB color sensors. The sensors were oriented on the frame of the robot such that one sensor could observe corner pieces and one could observe edge pieces. By applying sequences of moves to shift each corner and edge piece to the location of the color sensors, and then applying the inverse of these sequences to revert any changes, the entire Rubik's Cube could be observed without changing the starting state of the puzzle. The initial state of the Rubik's Cube was saved in a register called "cubestate". Since there are six possible colors for each sticker of the Rubik's Cube, 3 bits were required for each of 54 stickers, resulting in a total of 162 bits to store the initial state. Each sticker on the Rubik's Cube was referred to with a specific set of 3 bits from within the cubestate register, as detailed in Figure 2 below.

			0	72	9							
			75	144	81							
			3	78	6							
21	90	18	24	105	33	39	108	36	54	129	51	
93	147	87	102	150	96	111	153	117	126	156	120	
12	84	15	27	99	30	42	114	45	57	123	48	
			60	138	69							
			135	159	141							
			63	132	66							

Figure 2 - Mapping of cubestate indices to stickers of the Rubik's Cube

The algorithm used to solve the Rubik's Cube was based off of a modified version of a human method for solving the Rubik's Cube called "The Beginner's Method" or "Green Cross."² In general, each step of the method was handled with a case statement, and a variable for counting which substep of the method was being determined. The algorithm checked for various patterns on the Rubik's Cube, and produced sequences of moves that, when applied, would solve the pieces involved in that pattern. Our method differed from The Beginner's Method in the way it solved the last layer. While The Beginner's Method solves the last layer by orienting edge pieces, permuting edge pieces, permuting corner pieces, and then orienting corner pieces, our

² <https://solvethecube.com/>

method first oriented both edge and corner pieces in two steps, and then permuted them in two steps.

As the algorithm for solving the Rubik's Cube produced sequences of moves, these moves were outputted to two modules. One module, the sequencer module, queued the moves for execution on the physical robot. Another module, called update state, took these moves and applied them to cubestate, the internal representation of the Rubik's Cube's state. This module functioned by swapping values in the cubestate register corresponding to the 12 possible moves that can be executed on the Rubik's Cube - a 90° clockwise and a 90° counterclockwise rotation of each of the six faces. The updated cubestate produced by this module was then fed back into the solving algorithm module to produce further moves for the solution.

After the entire solution to the scrambled cube was found and queued in the sequencer module, the same sequencer module would begin to provide moves to the stepper motor driver modules. Each move would be translated into a series of steps by one of the six stepper motors attached to one face of the rubik's cube.

In the end, we were able to produce code that successfully determined the initial state of the Rubik's Cube, produced a sequence of moves to solve it, and executed the solution by driving stepper motors actuating the faces of the Rubik's Cube. Unfortunately, due to inconsistent lighting conditions and inaccuracies with the RGB sensors, we were not able to reliably determine the initial state of the Rubik's Cube. However, given successful determination of the initial state (or if the initial state was programmed into the FPGA), the robot was able to reliably solve the Rubik's Cube.

Bill of Materials

- 1 12" x 12" x 1/4" sheet of acrylic
- 12 (6 sets of 2) 3D printed parts for interfacing the stepper motors with the Rubik's Cube
- 6 stepper motor drivers
- 1 Nexys 4 FPGA
- 2 ISL29125 RGB color sensors
- 1.2 pounds of coffee beans

Modules

Main

Jacob Swiezy and Nathaniel Knopf

The main.v module was responsible for wiring together all of the different secondary modules involved in our project, as well as the finite state machine responsible for controlling the behavior of the robot in the process of determining the initial state of the Rubik's Cube and solving it.

The finite state machine in main.v began by placing the robot in the determine state mode. In this mode the robot began rotating the faces of the cube to determine the color of each individual square on the cube. After determining the state of the cube using the spin all and determine state modules, the state machine transitioned into the user debug mode where it sent the state representation of the cube over serial, so that a connected computer running a parser python program could output the expected state of the cube. Using this, a user could determine whether the robot had successfully determined the state of the cube. This python program also used the cube state data to quickly determine if the cube state was feasible given the number of believed squares of each color (i.e. if the robot determined the cube had 10 blue squares, the robot could warn the user of an error). After outputting the believed cubestate, the state machine transitioned into solution mode where the robot would use the solving algorithm module and the update state module to determine a solution to the rubik's cube. After finding a solution to the cube, the robot would transition into the physical solve mode where the sequencer module would be responsible for feeding the stepper motor modules one command at a time to solve the cube. Once the state was determined and the solution was found and implemented, the robot would enter into an idle mode where it would wait for a user reset signaling the presence of a new cube to be solved.

Additionally, main.v included verilog code responsible for interfacing with the inputs and outputs of the FPGA (including buttons, switches, and LEDs) for debugging and user interface with the robot; it also included some predetermined cubestate registers for scrambled Rubik's Cubes for which a sequence of moves was known to get the Rubik's Cube to the corresponding state.

Determine State

Nathaniel Knopf and Jacob Swiezy

The initial design of determine state consisted of a finite state machine consisting of five states - setup, prep, idle, observe, and done. In the setup state, all variables are reset. The state then transitions to prep.

In the prep state, `send_setup_moves`, which is responsible for signalling `spin_all.v` to output a new sequence of moves, goes high. The `cubestate` register is bit shifted left by 3 in preparation of observation of a new value. The state is then set to idle.

In the idle state, the `determine_state.v` module waits for the color sensors to produce a stable value. Once this occurs (determined by an input signal called `color_sensor_stable` going high), the finite state machine transitions to the observe state.

In the observe state, the value given by the color sensor is recorded in the three lowest order bits of `cubestate`. The counter (which keeps track of which piece is being observed) is then incremented, and the state returns to prep.

Once the counter reaches 48, signifying that all pieces have been observed, the state is transitioned to done (`spin_all.v` is also told to output one final sequence of moves to return the Rubik's Cube to its original state). The module then signals to `main.v` that the `cubestate` has been determined and that the program can proceed to generate a solution.

However, during testing, inaccuracies in the color sensors required a more intricate implementation of `determine_state.v`. To correct for errors in sensor readings, extra steps were added to the determination of the state of each edge square. While the original determine state module, used the spin all module to obtain exactly one RGB reading from each edge of the cube, the updated module took four measurements of each edge of the cube (i.e. it spun the left side of the cube four times and took a new measurement with each turn). The module then compared the four readings and determined the color of the cube edge to be the most popular reading. This process eliminated some of the error associated with color contamination from adjacent squares in the color reading.

Once the state of each edge was determined, the module transitioned to determining the state of each corner of the cube. Originally, the module took one reading of each corner square and used this reading on its own to determine the state of each corner, however, in the updated module the reading of the adjacent edge, which was already determined, was used in addition to the reading provided by the color sensor in order to more accurately predict the color of each corner piece.

Spin All

Nathaniel Knopf

The spin all module consists of a finite state machine with two states - idle, and send moves. In the idle state, the module waits for single case statement operating on the counter used in `determine_state.v`. Each case returns a sequence of moves that, when executed on the Rubik's Cube, places the sticker corresponding to the given value of counter in front of a color sensor.

Solving Algorithm

Nathaniel Knopf

The solving algorithm module was responsible for producing a sequence of moves that, when executed to the scrambled Rubik's Cube, would solve it. The inputs to this module were the clock, a reset signal from `main.v`, a start signal from `main.v`, a 162 bit register containing the state of the Rubik's Cube (at first its initial scrambled state, and later its state after applying the moves produced so far by `solving_algorithm.v`), and a signal from `update_state.v` signifying that it had produced the cubestate occurring after applying the most recently produced sequence of moves.

The method implemented in this module is a modified form of the human method called "The Beginner's Method," or "Green Cross." In this method, a cross is formed on the bottom layer by solving the four edge pieces on the bottom layer. The bottom layer corners are then solved, followed by the middle layer edges. The last layer of the Rubik's Cube is then solved in four steps: first, the edges are oriented (i.e. the white stickers of the edges are all on the white side). Second, the corners are oriented. Third, the edges are permuted (i.e. the edge pieces are moved into their final positions such that both of their stickers match the color of the adjacent centers. Finally, the corners are permuted, at which point the Rubik's Cube is solved.

A number of variables are used in the process of solving the Rubik's Cube. A variable called "step" tracks which step of the method is currently being executed - either cross, bottom_corners, middle_layer, oll_edges, oll_corners, pll_edges, pll_corners, or solved (signifying completion of the solution). A variable called "piece_counter" is also used to track substeps of each step of the method. For example, in solving the middle layer edges, there are four edge pieces which must be solved. The piece_counter variable keeps track of which edge should be solved next by the solving_algorithm module.

The core finite state machine of this module consisted of four states - move, update_state, wait, and setup. The finite state machine begins in the setup state (and returns there upon receiving a reset signal). In this state, all variables associated with solving the Rubik's Cube are set to their initial values.

Once a start signal has been received by the module, the finite state machine transitions to the move state. In the move state, the next sequence of moves to be applied to the Rubik's Cube was determined. Once a sequence of moves has been produced, the finite state machine transitions into the update_state state.

In the update_state, state, the finite state machine waits for the solving_algorithm module to receive a signal from update_state.v signifying that it has completed updating the cubestate according to the moves produced by solving_algorithm.v. Once this has occurred, the finite state machine transitions to the "wait" state.

The purpose of the wait state is to ensure that the solving_algorithm module doesn't produce new moves before the sequencer module has finished queuing up the moves produced thus far. This state consists of a counter that counts to 60 clock cycles, at which point the finite state machine transitions back to the move state, and the cycle continues until the puzzle has been solved.

The actual part of the move state of the finite state machine that produced sequences of moves to solve the Rubik's Cube consisted of a case statement operating on the step variable. Within each case corresponding to a step of the method with multiple substeps (cross, bottom_corners, and middle_layer), there was a second case statement operating on the piece_counter variable. These ultimate case statements recognized patterns on the Rubik's Cube that were of particular interest to the step (or substep) of the method, and returned moves that would advance the puzzle to the next step.

For example, if the step variable read to bottom_corners and the piece_counter variable read 0, the module would attempt to solve the bottom-front-left corner (i.e. the yellow-green-orange corner). The module identified patterns by checking each of the corner positions on the Rubik's Cube (of which there are 8, each one with three stickers, yielding 3 values that must be checked for each corner). In the case of corners, the corner piece of interest can be in one of 8 locations, with 3 possible orientations per location, yielding 24 different possible patterns. Each of the 24 possible patterns had a corresponding sequence of moves that could be applied to solve the yellow-green-orange corner. Once the sequence of moves had been produced, the piece_counter variable increased, signifying that the second bottom layer corner should be solved the next time the finite state machine returned to the move state. If the module had solved the last of the four corners, piece_counter was set to 0 and the step variable was incremented to the next step of the method (in this case, middle_layer).

Wherever possible, some redundancies in pattern recognition and move production were eliminated. For example, in solving the bottom layer, a single clockwise rotation of the top face will not disturb any progress done so far in solving the Rubik's Cube (since the top face is on the opposite side of the puzzle from the side being solved). Therefore, the module can check the four bottom layer corner positions and only one of the four corners on the top face (in our implementation, the top-front-left corner), and if it doesn't see the piece of interest, apply a clockwise rotation of the top face until the piece of interest is moved into the top-front-left corner. This has the added bonus of assisting in the process of debugging - if the `solving_algorithm` module is attempting to solve a corner piece, but due to some error in the `solving_algorithm` module or the `update_state` module the `cubestate` register doesn't contain the piece of interest in any location, the `solving_algorithm` module will produce an infinite series of clockwise rotations of the top face. This makes it immediately obvious where an error occurred in the production of the solution.

Once the code for `solving_algorithm.v` had been written, there was very little structurally that had to be changed. The simple nature of the finite state machine meant there was little opportunity for bugs to emerge involving the actual structure of the module. However, due to the complexity of the algorithm (1393 lines of nested case statements that produce ultimately hard coded sequences of moves), debugging was very difficult. The final module spanned 1393 lines, and bugs involving the hard coding moves didn't appear until one of the scrambled cubes happened to be solved in a way that involved the corresponding case statement. However, once bugs had been identified, correcting them was often as easy as correcting a typo in a sequence of moves.

Update State

Nathaniel Knopf

The update state module is responsible for updating the internal representation of the Rubik's Cube (`cubestate`) in response to moves produced by `solving_algorithm.v`. The inputs to this module are the clock, a 200 bit register called `new_moves` consisting of a sequence of up to 50 moves (each move is represented by a 4 bit number), the current `cubestate` before executing the new moves, and a signal `new_moves_ready`, signifying that the moves on the `new_moves` input need to be applied to the `cubestate`. The finite state machine implemented in this module has three states - idle, moving, and done.

In the idle state, the module waits for the `new_moves_ready` signal to go high. When this occurs, the input `cubestate` is copied over to a register internal to the module. After this, the finite state machine transitions to the moving state.

In the moving state, the module progresses through the moves on the 200 bit `new_moves` register, applying them one by one to the internal cubestate. There is a case statement that operates on the next move to execute, and for each of the twelve possible moves (clockwise and counterclockwise rotations of each of the six faces) the values at various indices in the cubestate register are swapped in accordance with the update to a Rubik's Cube after executing the given move. After the transformation corresponding to the move had been applied, the moves inputted to the module were bit shifted by 4, and the next move was selected. Once this process had repeated 50 times, the state transitioned to done.

In the done state, the newly updated cubestate is sent to the `solving_algorithm.v` module, and a signal signifying that the update had completed is set to high. The state of the finite state machine is then sent to idle.

Because the actual portion of this module responsible for swapping values in cubestate consists of 300 lines of hard coded indices, the module was prone to bugs that occurred due to typos. Detection of the bugs was also relatively difficult, because when a move updates cubestate incorrectly, the error often doesn't present itself until further in the solution when an incorrectly updated piece becomes crucial to the production of a solution. At this point, the code in `update_state.v` for each move had to be double checked to ensure that the indices were coded correctly.

Sequencer

Jacob Swiezy

The sequencer module is responsible for creating a queue of moves to execute in the solution determination phase and for sequentially returning one move at a time for the stepper motors in the physical solve phase.

While the robot is determining a solution to the scrambled cube, the sequencer module acts to queue the next moves in the solution. The module maintains 200 4-bit wide registers capable of holding 200 moves to execute in solving the cube. The module takes as input up to 50 of the next moves and then sequentially stores these moves in one of the 4-bit registers. If less than 50 moves are provided to queue, as is often the case, the sequencer module scans the input MSB to LSB, for the first nibble of data that is nonzero. Once found, the module adds the remainder of the data in the 200-bit input to the queue in 4-bit chunks and increments a total move counter. When the sequencer module receives a sequence complete signal, the module transitions from the move queueing phase to the move returning phase.

While the robot is physically solving the rubik's cube, the sequencer module produces the next move to execute as output to the stepper motors. Each time that the sequencer module receives a

signal that the previous move has been completed by the stepper motors, the module loads the next 4-bit move and increments the counter representing the number of completed moves. When the number of completed moves equals the move counter, the final move in the calculated solution has been executed and the sequencer module stops returning new moves, sets a signal signifying that the sequence has been fully executed, and goes into an idle state.

Move to Step

Jacob Swiezy

The move to step module is responsible for translating a 4-bit move into the appropriate sequence of physical steps on a stepper motor. Because of our clever assignment of moves to numbers, the top 3 bits of each inputted move corresponds to the stepper required to rotate and the LSB bit corresponds to the direction of rotation required.

We determined early on for our robot that it would only be feasible to execute one move at a time and thus only one stepper motor would ever be moving at a given time. Because of this, we determined that it would be allowable to tie all of the step inputs together to one step clock and to tie all of the direction inputs together for the physical stepper driver chips. Therefore, for each move, all six of the stepper driver chips are commanded to rotate in the same direction for the same number of steps (50 steps for a quarter turn), but only one of the stepper driver chips is enabled. In other words, for every move all of the stepper driver chips receive the exact same set of commands, but only one of the stepper driver chips has their output FETs enabled to physically drive its corresponding stepper motor. The enabled stepper driver chip thus rotates its stepper motor while the remaining five stepper driver chips are in a disabled state which allows the corresponding stepper motors to rotate freely.

Using the enable inputs as a means to control which stepper motor moved was an important design choice. A stepper motor that is not being commanded to move, but is still enabled will attempt to hold its current position while a disabled stepper motor will rotate freely. Our robot depends on this free motion in order to correct for small amounts of under or over rotation in previous moves.

Additionally the move to step module implements a step clock responsible for the frequency at which the stepper motors take steps while operating. This step clock is always enabled, but is reset with the start of each new move. As explained above, this signal can be always pulsing because it is the enable signals that are responsible for determining which stepper motor is actually moving at a given instance.

Finally, this module implements a move finished signal. This signal is a simple logical AND of all of the done signals for each stepper driver. When a move is in progress by any of the stepper motors, this goes low, and it returns high at the completion of the move.

Stepper Driver

Jacob Swiezy

This module is implemented once for each stepper motor in the move to step module. Each implementation consists of a simple counter. When the corresponding stepper motor is commanded to move, this module enables the stepper motor and then increments a counter on each low-to-high transition of the step clock implemented in the move to step module. When the counter reaches the number of commanded steps, the module then disables the stepper motor. After disabling the stepper motor, the module continues to count until an extra number of step cycles has elapsed. The module then transitions the done signal from low (move in progress) to high (move complete). This delay between the completion of the step motion and the assertion of the done signal allows the steppers to settle before a new move is loaded and commanded.

Color Sensor

Jacob Swiezy

The color sensor module contains the necessary setup values for the RGB color sensors and acts as a finite state machine for each sensor. The module implements two I2C modules for communication with the color sensors. One module, the setup module, is responsible for writing the necessary configuration registers of the color sensor. This must be done once at the beginning of operation. The second module, the poll module, is responsible for polling the value of the red, green, and blue light sensors at a set frequency.

The module begins in a setup state where it places the poll module in a reset state while enabling the setup module. Once the setup module signals completion, the color sensor module transitions into a polling state where it enables the polling module.

I2C Setup

Jacob Swiezy

The I2C setup module is responsible for writing the appropriate values to the RGB color sensor configuration registers. For proper operation, two of the configuration registers on the ISL29125, need to be modified from their power on configuration, CONFIG-1 and CONFIG-2. We modify the CONFIG-1 register to enable the three light sensors and to increase the allowable light sensing range for bright environments. In addition we modify CONFIG-2 to increase the IR blocking filter to be at a maximum.

This module implements the burst write operation of the I2C protocol allowing a master device to specify the slave device address and a starting register address followed by an unrestricted number of bytes for consecutive register writes. The entire operation is implemented as a finite state machine clocked at roughly 200kHz by an auxiliary clock module. In all 21 different states are required to properly modify the necessary configuration registers. At a high level the state machine functions as below:

1. Send start signal
2. Send device address
3. Wait for acknowledgement
4. Send address of first configuration register
5. Wait for acknowledgement
6. Send data byte
7. Wait for acknowledgement (if more data bytes available go to 5 otherwise proceed to 7)
8. Send stop signal

I2C Poll

Jacob Swiezy

The I2C poll module is responsible for reading the values of the Red, Green, and Blue data registers at a set frequency. The ISL29125 has 3x 16-bit registers each containing the most recent sensor values for their respective color sensor. The values are updated by the sensor at a rate of roughly 10Hz, but since we don't care if we receive repeat values we will continually read the sensor data registers to ensure we have the most up-to-date readings.

This module implements the burst read operation of the I2C protocol allowing a master device to specify the slave device address and a starting register address followed by an unrestricted number of bytes for consecutive register reads. The entire operation is implemented as a finite state machine clocked at roughly 200kHz by an auxiliary clock module. In all 32 different states are required to properly read the appropriate data registers. At a high level the state machine functions as below:

1. Send start signal
2. Send device address
3. Wait for acknowledgement
4. Send address of first data register
5. Wait for acknowledgement
6. Send restart signal
7. Read data byte
8. Send acknowledgement (if more data bytes to read go to 7 otherwise proceed to 9)
9. Send stop signal

Serial Module

Jacob Swiezy

A serial module was implemented to allow for debugging information to be transmitted from the FPGA to a computer for user review. The communication operated with a standard 9600 baud clock and with the default serial setting of 8 bits per transmission, no parity bit, and one stop bit. We will not go into depth on this module for the reader's sake because it was not critical to robot functionality.

Supporting Modules

In addition to the modules described above, the following (relatively simple) supporting modules were used:

- A 200 kHz clock signal was generated using a simple counter for the I2C clock.
- A 200 Hz clock signal was generated using a simple counter for the step clock.
- A delay module was used to produce a delay for the color sensors to stabilize.
- A baud rate generator (i.e. 9600 Hz clock) was generated for use by the serial module.

Hardware

Stepper Motors

Jacob Swiezy

To physically rotate and solve the cube we used six NEMA 17 bipolar stepper motors. Stepper motors were chosen over other types of motors because they are continuous (i.e. they can rotate in one direction infinitely), they have a relatively high torque, and they are relatively precise without additional feedback. The stepper motor operates using two coils and a permanent magnet rotor. When the coils are energized in a particular sequence, the stepper rotates by precisely 1.8 degrees (i.e. it takes 200 steps for the shaft of the motor to make a complete rotation).

Unfortunately, rotating stepper motors is a nontrivial task and often requires a specialized driver. In order to reduce the control complexity and the computational load on the FPGA, we opted to offload the control of the six stepper motors to six specialized stepper motor drivers. Each A4988 stepper motor driver requires 3 control signals (step, direction, enable).

The step signal is responsible for controlling when the stepper motor takes a step (i.e. rotates 1.8 degrees). A low-to-high transition at this input causes the stepper to take a step in the direction specified by the direction input.

The direction signal is responsible for specifying which way the stepper will take a step on the next low-to-high transition of the step input.

The enable signal directly controls the FETs responsible for energizing the stepper motor coils. A high input on this input disables all output FETs while a low on this input allows the output FETs to operate in accordance with the step and direction inputs. Importantly, if this input is left enabled (i.e. at a logic low level), the stepper motor will not rotate freely even when it is not being directly controlled by the step and direction inputs.

Interfacing to the Cube (Stepper Shafts)

Jacob Swiezy, Nathaniel Knopf

In order to interface between the stepper motors and the rubik's cube, we designed custom 2-piece hubs to turn each face of the cube. One piece of the hub fit snugly on the stepper motor shaft while the other piece of the hub pressed snugly into the center square of each face of the rubik's cube. These hubs were designed as 2 pieces for easy removal of the cube from the robot.

Both pieces were designed in Solidworks and then manufactured using a 3D printer. The parts were designed to interfere as little as possible with the movement of the cube and for easy assembly (i.e. both pieces were press fit).

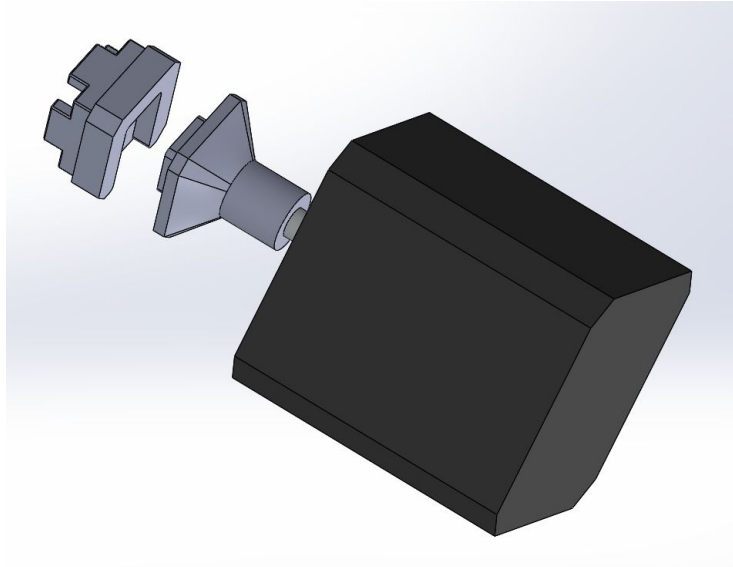


Figure 3 - 2 part stepper shafts

Robot Frame

Jacob Swiezy

The frame of the robot was carefully designed in Solidworks as a 3D CAD model before being manufactured out of ¼” acrylic. The CAD was developed around the size of the cube, mounting pattern of the color sensors, and mounting pattern of the NEMA 17 stepper motors. The design was meant to be easy to manufacture, simple to construct and repair, and most importantly provide a relatively unobstructed view of the entire cube during the entire solving process. The design also needed to be rigid so that the movements of the stepper motors translated into precise rotations of the rubik’s cube faces.

In order to make the design easy to manufacture, we designed the robot to consist of only 2 unique parts. The robot consisted of only two identical top plates and four identical side plates. This not only made manufacturing easy, but also allowed us to create spare parts without wasting a bunch of extra material.

In order to make the design easy to assemble, we designed the robot using a simple interlocking structure such that the side pieces of the robot snugly fit into the top and bottom pieces of the robot. In addition to the interlocking structure, metal 90 degree angle brackets secured the side plates to the bottom plates. The top plate was left secured only by the interlocking structure of

the robot, so that the rubik's cube could be easily removed and replaced from the top of the robot.

Finally, we designed the frame of the robot to be minimalistic, so that the user could view the entirety of the cube throughout the solving process. In addition, we chose to use transparent acrylic for construction of the robot, to provide an even less obstructed view of the cube throughout.

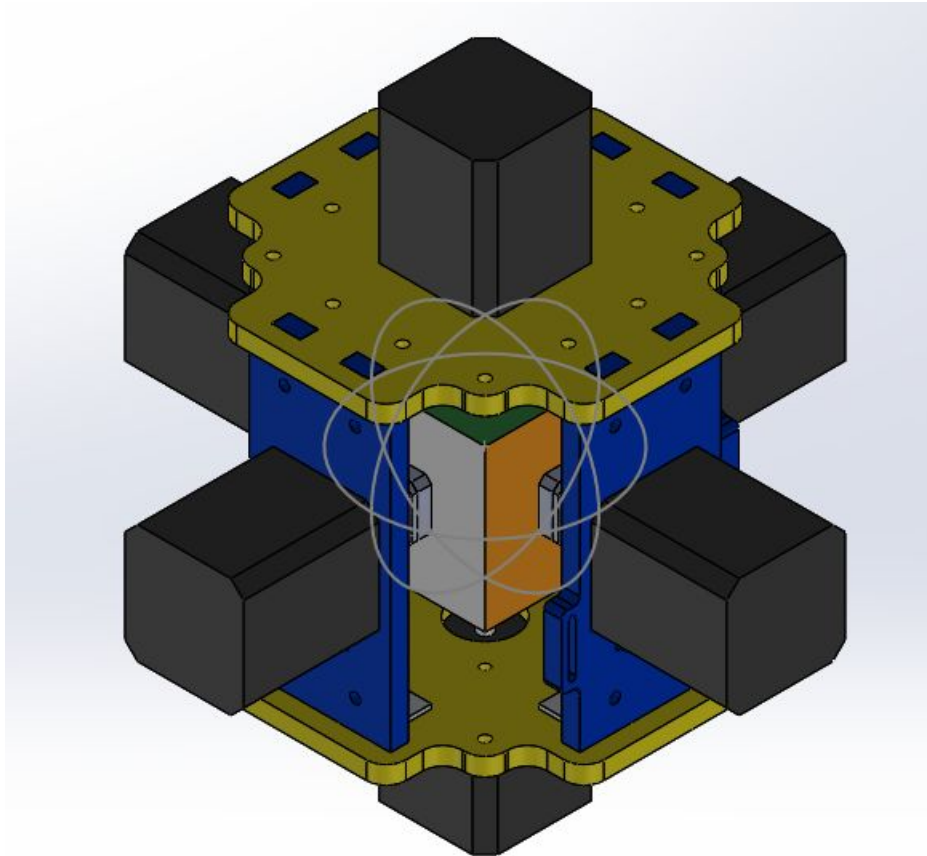


Figure 4 - Robot frame

RGB Color Sensors

Jacob Swiezy

An RGB color sensor is a sensor composed of 3 photodiodes each masked with a different filter. Each filter allows one of red (~600nm wavelength), green (~560nm wavelength), or blue (~440nm wavelength) light to pass through. The sensor integrates the amount of light detected over a specified period of time and returns the results as 3 16-bit readings. Using I2C, the user can read this data to determine the RGB light values present at the face of the sensor. Because

the readings depend entirely on the amplitude of each wavelength of light present at the sensor, the results returned by the sensor will depend heavily on the lighting conditions of the sensor's environment (i.e. a sensor placed in a dark environment will return much smaller results than a sensor directed at an identically colored object in a bright environment).

For our project, we tested two different RGB color sensors for determining the initial state of the rubik's cube. Below we will detail, the pros and cons of each sensor and what affected our final decision.

Adafruit TCS34725

Jacob Swiezy

The first sensor we tested was a breakout board for the TCS34725 RGB color sensor supplied by Adafruit. The sensor communicated over I2C, had an IR blocking filter, and had the capability of returning 4 16-bit values (Red, Green, Blue, and White). The breakout for the sensor included a super bright white LED in close proximity to the sensor. This LED ensured more consistent results across different environments the sensor might be placed in. The breakout board also implemented level shifting transistors which allowed the board to be easily interfaced with a 5V system despite the sensors 3.3V operating voltage.

We began testing this sensor with a Teensy 3.6. The sensor required very few setup instructions and had a relatively high refresh frequency. The results of the tests showed that the sensor could clearly differentiate between cube sticker colors in a variety of lighting environments given that the sensor was in close proximity to the cube. Because of our success with this sensor, we chose to move forward with it in our final design.

In testing we discovered that this module was not able to interface with the Nexys4DDR FPGA we had chosen to control our project. Initially, we determined that the incompatibility of this sensor with the FPGA was due to either the onboard level shifting transistors or the weak pullup resistors provided by adafruit. We were concerned that either the weak resistors were not creating a strong enough high level voltage on the sensor/FPGA interface or that the level shifting transistors were not allowing one of the boards to completely pull down the signal line during communication. After extensive testing with an oscilloscope, we determined that the issue was with the sensor which was not acknowledging the FPGA after successful transmission of the device's address. We attempted to solve this problem in two ways. We began by adding stronger pullup resistors to the interface, but problems persisted. Next we bypassed the onboard level shifting transistors. This also produced no successful results. After more testing with an oscilloscope, we determined that there were additional underlying issues with the interface between these boards. Realizing that it could take many hours of debugging to find these problems, we decided to continue with a different sensor.

Sparkfun ISL29125

Jacob Swiezy

The next sensor we tested was a breakout for the ISL29125 RGB color sensor supplied by Sparkfun electronics. The sensor communicated over I2C, had an IR blocking filter, and had the capability of returning 3 16-bit values (Red, Green, and Blue). The breakout for the sensor did not include any additional hardware.

We initially began testing this sensor with a Teensy 3.6 as well. We determined that without an LED on the board, determining the color of a specific square on the rubik's cube would be more difficult and unreliable, but we determined that we could rig up an external LED in close proximity to the board on the robot in order to achieve similar levels of success that we'd had with the Adafruit sensor.

Unlike the Adafruit sensor, this sensor interfaced flawlessly with the Nexys4DDR FPGA. This sensor also had minimal required setup and a high refresh frequency. Unfortunately, even with the addition of an external high brightness LED to our robot, we were unable to obtain the same reliable results with this sensor that we achieved with the Adafruit sensor. Light contamination from adjacent stickers and inconsistent lighting environments both contributed to the limited success we were able to achieve with this sensor.

Challenges and Concluding Thoughts

While the Verilog written was all fully functional after debugging concluded, slight physical problems, such as varying lighting conditions and misalignments of the Rubik's Cube, made achieving our overall goal of producing a robot that determined the scrambled state of a Rubik's Cube and solved it difficult.

The initial set of RGB color sensors used experienced issues when I2C communications were attempted. The replacement sensors ultimately used in this project produced many issues that inhibited the robot's ability to determine the starting cubestate. In a controlled environment with invariable lighting conditions, the robot would be able to reliably determine the color of each square of the cube. Unfortunately, when attempting to determine the color of a given piece, shadows, light contamination from adjacent pieces, and even smudges on the face of the Rubik's Cube caused the color sensors to be unreliable in determining the color of each sticker consistently. This, of course, led to unreliable determination of the initial cubestate, and therefore incorrect solutions.

The robot also experienced issues in the rotation of the cube. The stepper motors and stepper motor drivers used provided no feedback to the FPGA to affirm that a perfect 90° turn had been executed. As a result, slight errors accumulated in the alignment of the Rubik's Cube after turns had been completed. Luckily, we were able to correct for some of this by disabling stepper motors that were not in use, and relying on the ability of the Rubik's Cube to turn while not perfectly aligned. This ability is known as "corner cutting" in the Rubik's Cube solving world, and had the lucky benefit for us of re-aligning misaligned faces on the puzzle.

Unfortunately, sometimes repeated sequences of the same move resulted in such an error in alignment that the robot would attempt to turn an adjacent face and jam up. The robot, not having any feedback from the stepper motors, would not be able to determine that a jam had occurred, and would continue trying to execute moves, usually incorrectly due to the skipped move.

Were we to continue work on this project, the following attributes would be changed

- Feedback would be added to the stepper motors through the use of shaft encoders.
- HSV instead of RGB would be used to determine the color of the stickers. This encoding allows for more tolerance of varying lighting conditions, and could eliminate many of the issues that made determining cubestate difficult.

- The algorithm implemented in solve_algorithm.v would be based on a graph search or maximizing some heuristic, rather than a collection of patterns to be recognized. This would allow for easier debugging of the module itself.

Appendix - Block Diagram

