WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

RBE 2001 Final Project Report

SUBMITTED BY TEAM 19
Dhionis Zhidro
Conrad Tulig
Garett Ruping

Submission Date: 12 October 2019
Course Professors: Bertozzi and Miller
Section: RBE 2001, A02

Abstract

The objective of this project was to design a mechanically sound, electrically safe, and programmably competent robot to achieve the task of placing and retrieving solar panels of two different materials on multiple different orientations. The robot that was designed implements a four-bar, line tracking, a limit switch, and a servo driven gripper.

During the presentation, our robot was able to fully traverse the course, picking up and placing both versions of the solar panels on both sides.

Table of Contents

Table of Figures

Introduction and Background

Many modern robots operate under the premise of receiving human input at the highest level of abstraction, that are then carried out through lower levels of autonomy to complete the command. Roombas are given a set of parameters of where and when to clean, and even NASA's Mars rovers rely on human control to give instructors for the robot to carry out autonomously. The final project simulates this important concept through the field controller's connection with the robot. Through the high level input of solar panel locations and action approvals, the robot must be able to interpret and carry out these commands by sensing its environment in order to properly execute the commands. In addition, the robot must be capable of physically performing these tasks with a four bar linkage, tuned motor transmissions, and structurally and electrically robust robot.

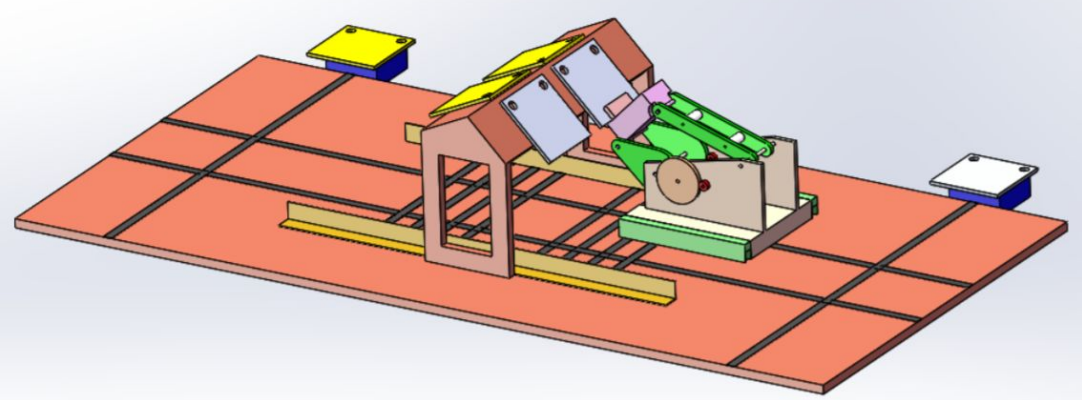


Figure 1: Field Showing all Possible Solar Panel Locations

The basic objective of the challenge is to be able to autonomously be able to pickup and dropoff solar collectors at both roof structure angles, 45 and 25 degrees, and the staging area. After receiving information from the field controller about which side the robot is on and which

solar panel is missing, with multiple possible configurations, as seen in Figure 1, the robot must be able to interpret this data in its execution rather than running a hardcoded solution. It is up to the robot to be able to navigate to the empty collector spot to place the new solar panel, as well as navigate to the old collector spot for removal. Finally, the robot can also drive to the other side and repeat the process.

Our team approached this challenge by beginning with mechanically designing the robot in Solidworks with a calculated four-bar mechanism. The robot was primarily built through 3D printing and laser cutting each of the parts. Then, the program heavily utilized motor encoders, line followers, and a limit switch to analyze its internal and external conditions to consistently complete the challenge.

Design Methodology and Analysis

The programmatic and mechanical challenges of the project were greatly considered in the design of the robot. Some of these aspects included alignment for drop off and pick up, required torque for the heavier payload, and how to best manipulate the panels.

Mechanical design:

We began by establishing a few constraints to give direction to the design process with the first constraint essentially being that we would use what was already available to us in terms of actuators and framework. We established that it would be much more time effective to use the basebot that we had already built as a drive base and to use one of the predesigned manipulators.

At this point, we also established what materials were going to be used and what manufacturing strategies we were planning on, these were considered as we were constrained by overall cost which is primarily derived by material choice and manufacturing. We decided to primarily use plywood as it is cheap and widely available and to form it by laser cutting which is a fast and simple process. For a few select components, we would use 3D printing. We observed many teams opting for a strictly 3D printing approach but we viewed this as too costly, both in terms of capital and overall time. In addition, we feared 3D printing accessibility as demo day approached would become scarce, and a robot designed for 3D printed parts would be too vulnerable to any last minute changes. Later on, we also found that the cost of 3D printing the manipulator was more expensive than the cumulative cost of all of the laser cut wood

components, of which encompassed 80% of the rest of the robot. In addition, any tapped holes were set to be #8-32 as that was the tap we had on hand.

There were two predesigned manipulators, the bottom out and linear carriage design. We ultimately opted with the bottom out design as it was simpler, both in terms of operation and part count. We were also attracted to it as it was designed such that the linkage in the manipulator prevents the manipulator from being back driven when it's in the closed position. One of our design concerns was the weight of the heavier payload overpowering the output torque of the manipulator's servo motor. The fact that the manipulator design prevents this was very attractive to us. Once the decision was made, mounting holes were added to the main body of the manipulator, keeping them as close as possible to minimize deflection.

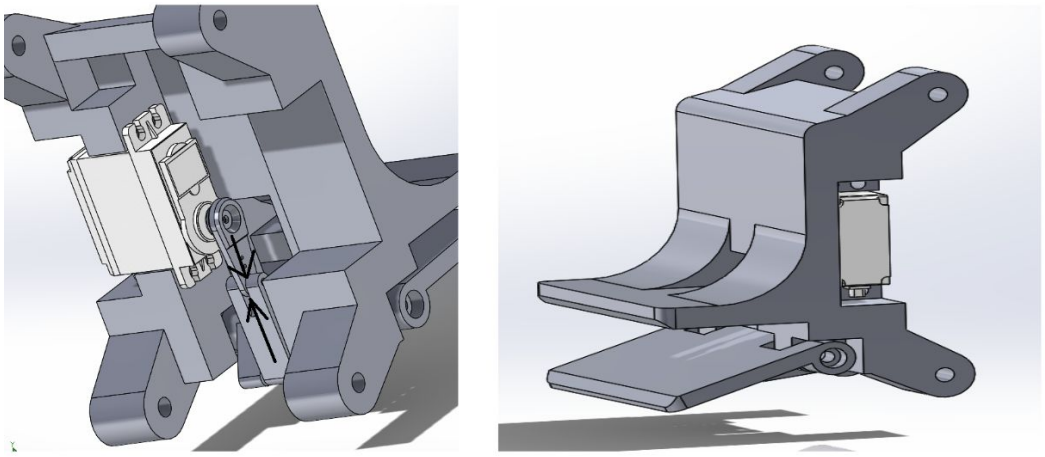


Figure 2: Gripper
Link force vectors in closed configuration preventing back driving (L). Added Geometry (R)

We knew that we would have to conduct a 3 position linkage synthesis to develop a four bar mechanism with the manipulator acting as the coupler. Before we could conduct the synthesis, we first had to establish the 3 positions and orientations and whether we wanted to prioritize picking up or dropping off as we only had 3 positions. We ultimately decided that

dropping off would be a more sensitive procedure and that picking up wouldn't require as much precision. Of course we established that one position would have to be dedicated to picking up from the staging area as we felt it was the most fundamental task to accomplish during the mission.

We manually manipulated the panels on the field to see what were the best possible ways of dropping off the panel on each side of the roof. We would hover the plate above the mounting pins at different distances and angles and drop it and see what combination would yield the most success. We found that on the 25 degree side, orienting the panel parallel to the roof and positioning it slightly above the pins would give us the highest rate of success. Whereas on the other side, positioning the panel slightly above the pins and at a slight angle would give us the best success. The panels in their drop off configuration are in the figure below and made transparent with some added dimensions.

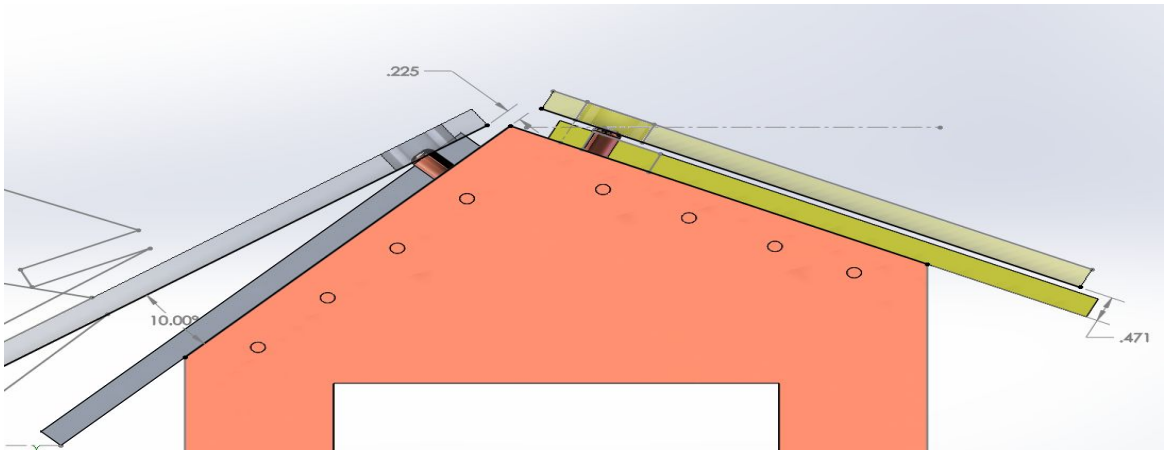


Figure 3:Panels in their drop off configuration, transparent.

From how the drop off configurations were established, one can observe that the panels can be placed by driving right up to the appropriate structure side and simply release the panel to drop it off onto the pins, no additional manipulations necessary.

And now that the drop off configurations were established and all 3 positions for the linkage synthesis were determined, came the actual synthesis process. The first step was generating a simpler coupler geometry for making the synthesis process easier. The top and bottom left hand vertices represent the pivot points whereas the slot represents where the panel sits.

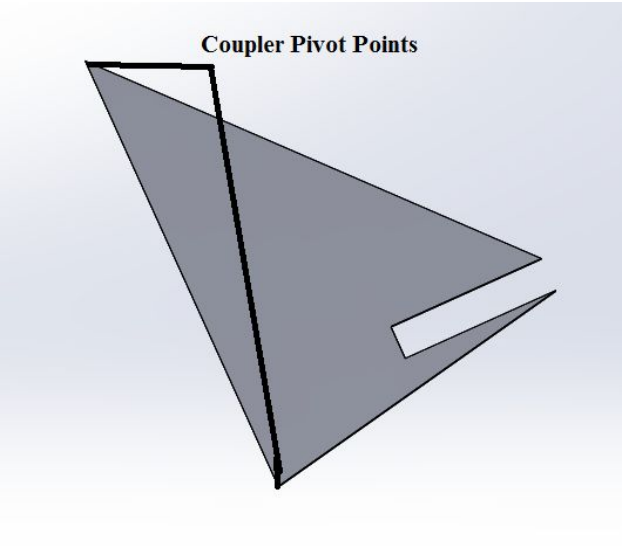


Figure 4:Simplified Coupler geometry.

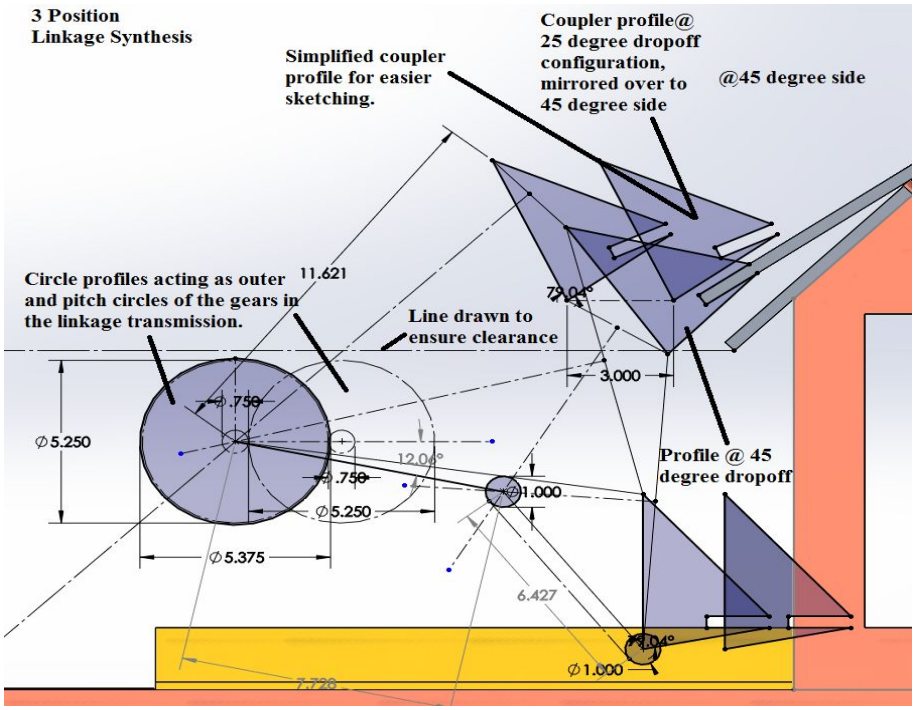
From there we created a sketch in the field assembly file and mated the panels and manipulator profiles in their respective spaces. A few things were kept in mind during the actual synthesis, minimizing link lengths, packaging the transmission to ensure the robot could clear the structure, and ensuring drop off positions corresponded to one of the roof alignment lines on the field. We wanted to be able to drive right up to a roof alignment line to be in drop off configuration.

We also wanted to achieve something close to a 50:1 gear reduction while minimizing stages of gearing, more gearing results in more backlash which would greatly compromise

sensor values. In the figures on the next page, one can see pitch and addendum circles of the gears we selected. We had decided that one of the gear profiles would be cut into the crank arm itself so that the crank arm functions as a gear. We opted for 2 7:1 reductions which gave us a net reduction of 49:1 with each stage consisting of an 84 tooth and 12 tooth gear meshing. As mentioned before, we would be using laser cut wood wherever possible, this included large gears and while VEX gears come in a diametral pitch of 24, we opted for a diametral pitch of 16. This was precisely because we were using laser cut wood, having a diametral pitch of 16 as opposed to 24 would result in wider gear teeth, reducing the chance of component failure by mode of shearing. This was decided after consulting a former FIRST robotics mentor on the matter. Decided to use laser cut gears of this diametral pitch and reduction did have a negative impact on the design, the gear diameters grew to the point where driving the shorter link was spatially impossible and the linkage would have to be driven from the longer link, leading to less optimal transmission angles. This ultimately would have little real impact on mission performance as we had more than enough mechanical advantage to drive the linkage.

As mentioned earlier, a sketch was placed on the field assembly file and the couplers were mated to their appropriate drop off spots. The sketch was done on the 45 degree side of the field with the coupler geometry on the 25 degree side mirrored over. The gearing was drawn out and positioned such that there was clearance on the top surface of the top plate of the basebot and clearance from lowest hanging panel on the field, which happened to be the 45 degree side panel.

Figure 5: Captioned 3 Position Linkage Synthesis



As most 3 position linkage are done, the top most coupler vertices are connected in series with perpendicular bisectors running through them, likewise for the bottom most vertices.The intersection points of the sets of the perpendicular bisectors serve as the pivot points of the linkage. The bottom coupler and mirrored 25 degree coupler profile were shifted around to better link lengths and pivot point positions. We found that leaving the 25 degree coupler profile in its original mirrored state would result in very inefficient geometry. We did find, however, offsetting the coupler by 3 inches, which is the sum of the widths of a roof alignment line and a gap between the lines, would result in much more efficient packaging and maintaining the linkages optimization towards dropping off at the alignment lines. This is reflected in the figure below. After enough shifting around, the linkage was fully synthesized. It's worth noting that the gearing was later shifted such that the 12 tooth gear driving the 84tooth crank arm would be on

the left side instead of the right side as depicted in the synthesis sketch above. This was done for an obvious reason that driving the right side would require very awkward crank arm geometry.

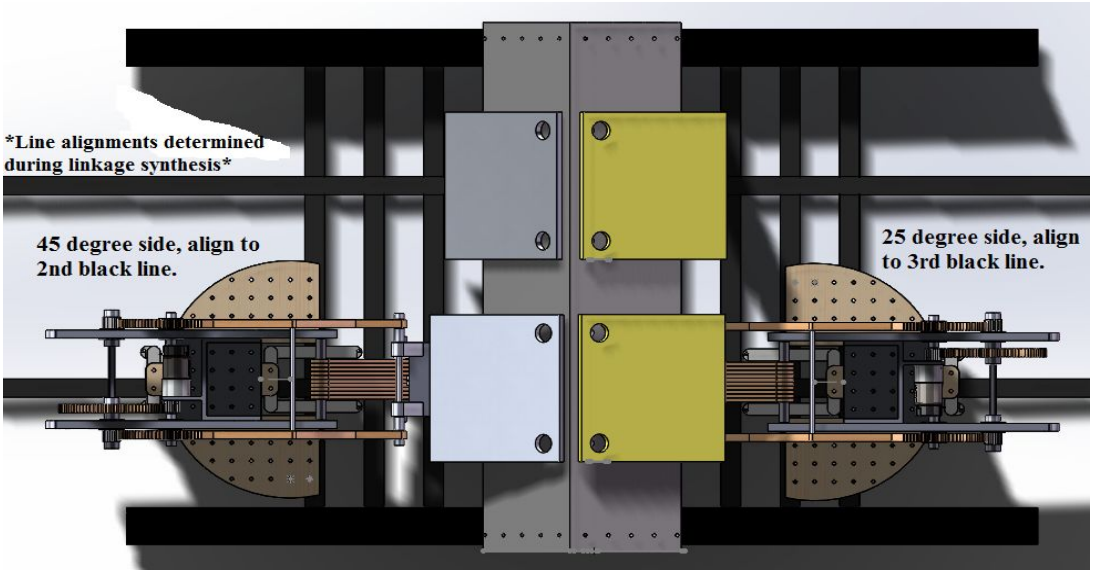


Figure 6: Robot at dropoff locations.

We were confident that the linkage was right for the design, however, we still iterated through the critical crank angles and observed the transmission angle at each one. Typically a transmission angle close to 40 is desired, any lower runs into the realm of highly inefficient required input torque.

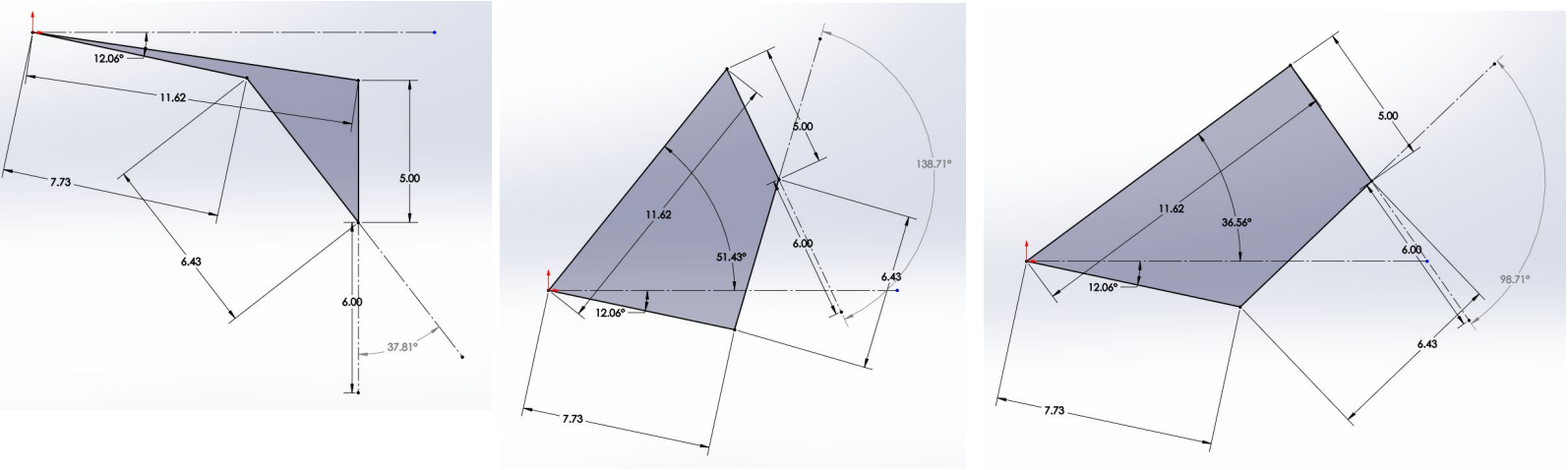


Figure 7: From left to right, linkage at picking up from staging area, 45 degree dropoff, and 25 degree dropoff configurations.

While the mission performance would not be affected by linkage fulfilling the Grashof or not, for the sake of analysis it was evaluated for it. The linkage is non-Grashof but this would not affect mission performance, tabulated below are the link lengths.

Crank link	11.621”
Coupler(Gripper)	5.000”
Rocker	6.4270”
“Ground”	7.7280”

At this point, the synthesis sketch was projected onto a gearbox plate mounted onto the basebot assembly on the field. The rest of the linkage transmission was underway. As mentioned previously, the aim was to maximize use of laser cut wood parts and minimize 3D printed parts. The gearbox plates would be composed of layers of ¼” nominial plywood, .216” in reality according to our calipers, cheap and widely available, one sheet ended up producing most of the robot parts and cost less than what it cost to 3D print the manipulator.

Due to their size and geometry, the links and gears were also manufactured in this fashion. While the 84 tooth gear and the 84 tooth crank arms were ideal for laser cutting, the 12 tooth motor pinion and the 12 tooth gear were not suitable for this. The effective wall thickness of the gears would be too thin for layers of laser cut wood gears, considering the torque on the components there were bound to fail from internal cracking. The pinion and gear would be 3D printed, aside from a few spacers, the manipulator and the main gearbox support plate, those are the only 3D printed parts on the robot.

Now the diametral pitch was altered to reduce the chances of tooth shearing on the wooden gears. To further reduce the chances, the crank arms would be driven on both sides,

supported by ¼” round shaft, cheap and readily available. One can see this in the transmission packaging image below. The 84 tooth gear synchronously drives the two 12 tooth gears through a ¼: square shaft, cheap and available at your local Home Depot. Here, we saw an opportunity to add stiffness to the gearbox by using shaft collars to act as reaction surfaces to prevent the gearbox plates from shifting along the shaft. Of course the shaft collars were also used to capture the gear in place. We were concerned about the shear stress on the inside of the gear by the shaft. To better distribute load, we decided to make a ½” 3D Printed square hub which presses right into the gear and is rigid enough to endure the loads without deformation. A part of this geometry is simple and quick to make and provides great benefit to the design. This is shown in the right image below.

Figure 8: Top down view of transmission packaging.

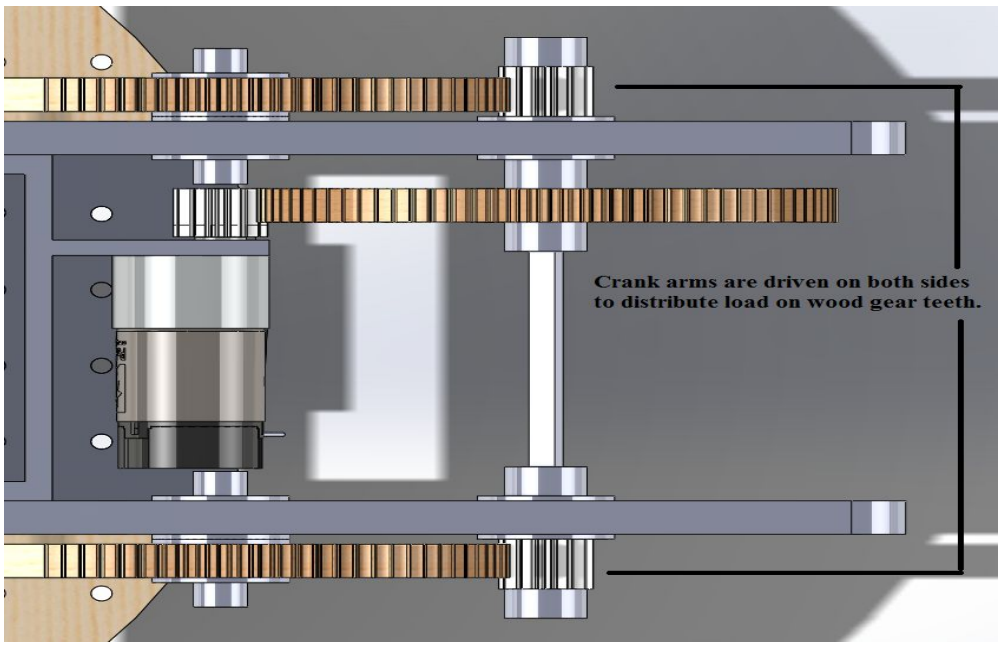
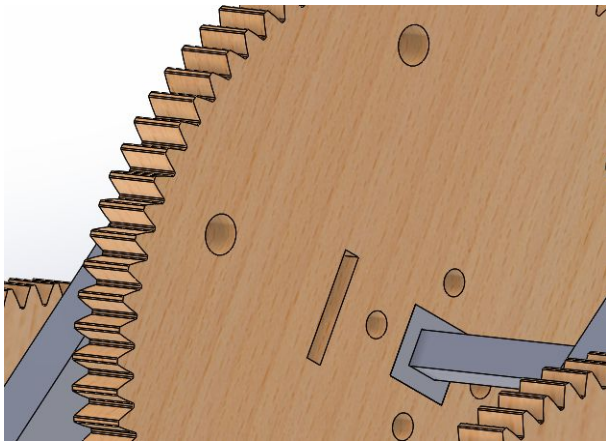


Figure 9:84 Tooth gear hub.



Driving the transmission is the Pololu 50:1 Gearmotor, of course the driving force behind the linkage needed to be rigidly fixed. We realized the mounting geometry for the Gearmotor

could double as gearbox plate cross-bracing and support. Leading to the development of the part shown on the next page.

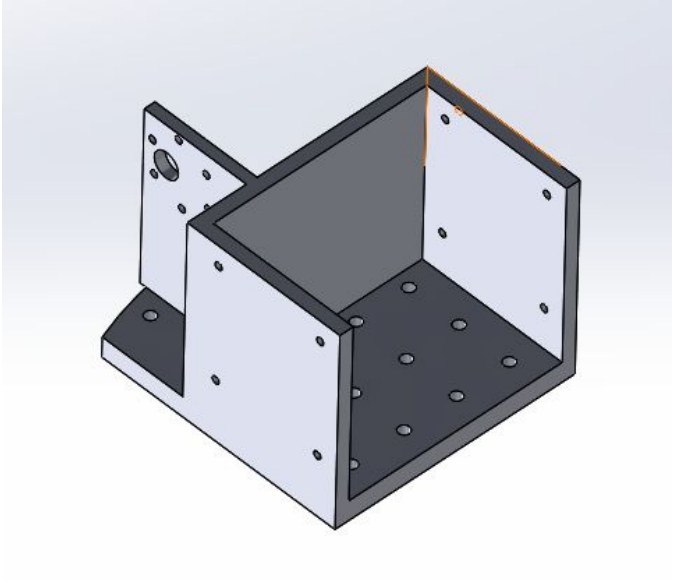


Figure 10: Hybrid Gearmotor mount and gearbox plate cross-brace.

The part makes the entire assembly very sound as it fixes the baseplate and gearbox plates together and keeps the Gearmotor well mounted relative to the rest of the linkage transmission as it is fixed right to the plate through the mount. The Gearbox plates attach to the the side using #8-32 screws which thread right into the mount whereas the baseplate attaches to the bottom side using #10-32 screws and nuts. This part makes the overall packing on the gearbox very clean and at this point we felt that the Gearbox was as most efficiently packaged possible given the constraints we were considering. This part is a pivotal part in the actually assembly process and in fact one of the first steps is fastening to the Gearmotor to the mount and pressing the pinion into place.

Now friction is always a concern when designing transmissions, in figure 8, one can observe washers separating moving surfaces such as between the gears and gearbox plates or the shaft collars and gearbox plates. Note that between the crank arms and the gearbox plates are two washers, the extra washers were added in to prevent the crank link from contacting the gearbox plate in the event that the crank link had any bending to it due to the nature of plywood. supporting the shafts and preventing friction along the shaft surface were nylon spacers pressed into the gearbox plates. ½" OD, ¼" ID nylon bushing are easy to find in ½" lengths at local hardware stores, the spacers in the later manufacturing stages were shaved to proper length using a rotary cutter after pressed into place. The issue is finding a nylon spacer which has and ID of the diagonal length of ¼" square. Unfortunately spacers of that dimensioning aren't readily available, so instead the spacers for that shaft were drilled out using a 23/64" drill bit, of which is not commonly available as the nearest one available was at a Lowes 16 miles from campus. It does however grant an ID of .3593" which is only 6 thousands of an inch greater than the critical shaft dimension of .3536", perfectly acceptable tolerance for a moving assembly of this nature.

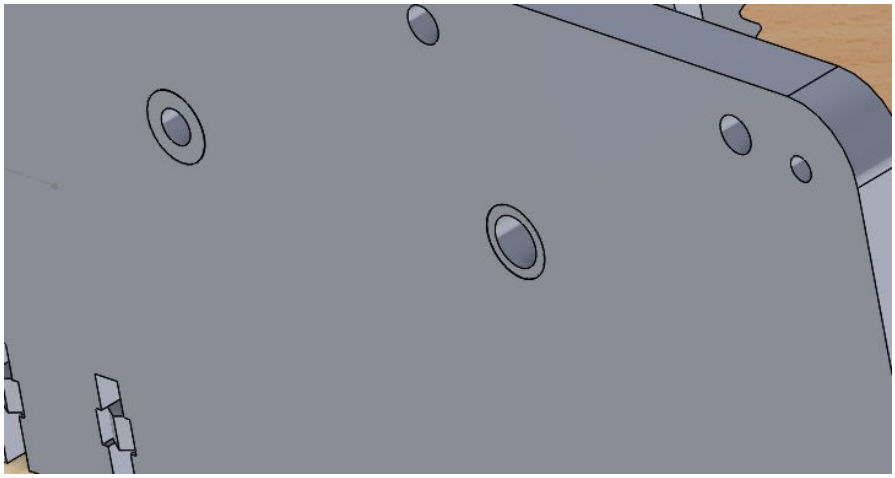


Figure 11: Nylon spacers acting as bushings.

That was not the only instance of nylon spacers being pressed in as bushings. In the figure below, one can observe the construction of the rocker link, many layers of laser cut plywood which ultimately led to a more rigid linkage. The link was just wide enough to fit 1” ¼” ID spacers on each side. Friction on this side of the assembly was further reduced by adding in fiber washers between any moving surface, like the shaft collars. One can see this in the figure in the lower right.

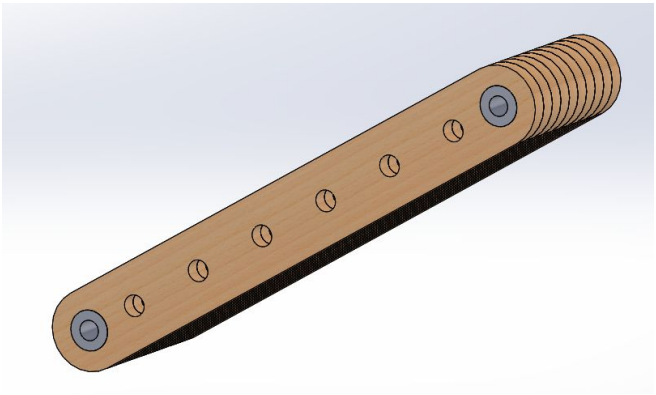


Figure 12: Nylon spacers pressed into the rocker link

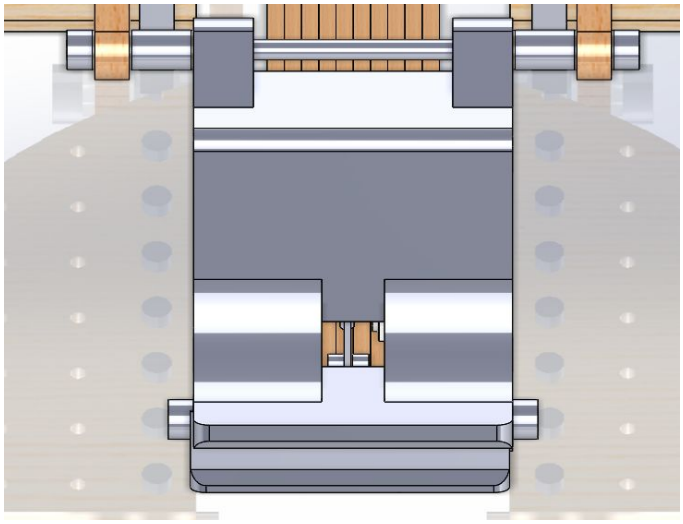


Figure 13: Fiber washers being used on the manipulator side of the assembly.

As for shafting for the rest of the pivot locations, the shaft were dead axle and turned from ¼” round steel rod, cheap and widely available on a Craftsman lathe from 1956. These dead axles dual functioned to act as cross-braces. Another cross brace was added to the crank lengths to maintain rigidness and angular alignment.

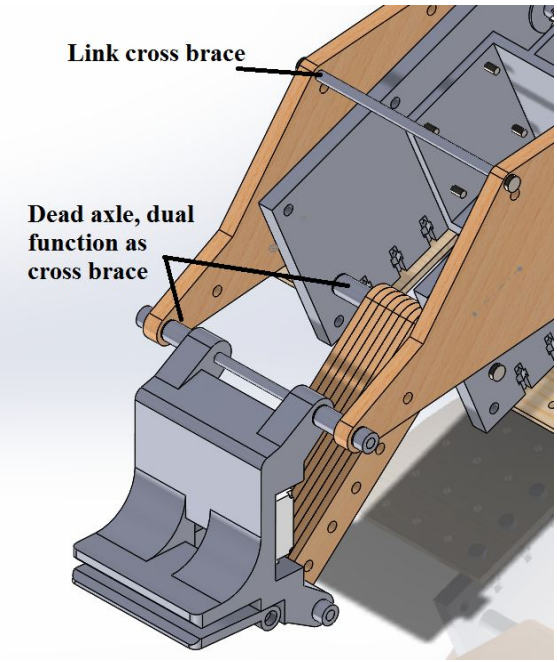


Figure 14: Dead axles and cross braces.

Another design detail worth pointing out, these parts were going to be laser cut from wood and then layered but the actual process of wood gluing layers together has its own challenges of alignment and required clamping force. One can observe in the figure below that the layered parts are peppered with ¼" screw clearance holes, which function for alignment and clamping and is how those parts were properly layered. The t-slots on the plates were only added in as a precaution, ultimately unnecessary.

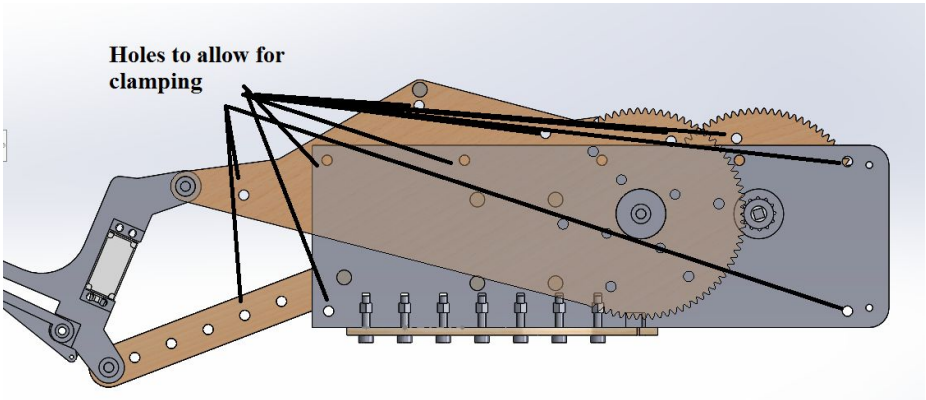


Figure 15: Clamping and alignment solution.

The last real design step was slightly modifying the manipulator, we had chamfered one of the back corners and added in a layer of rubber grip material for better panel manipulation.

Figure 16: Chamfered manipulator

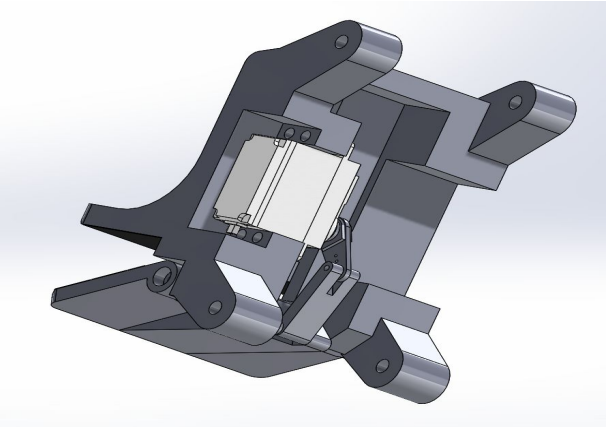
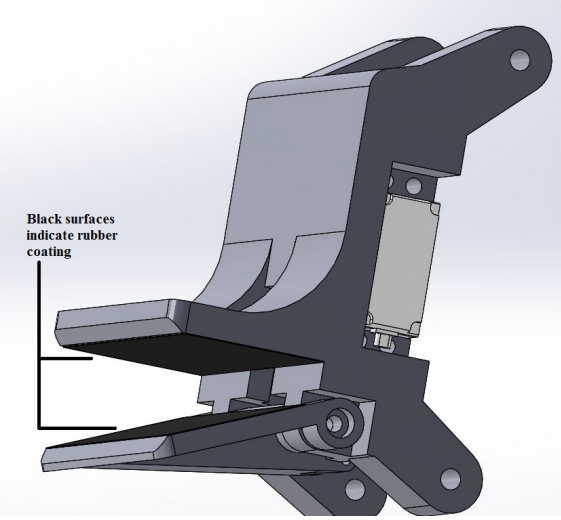


Figure 17: Rubber grip



That just about fully describes the mechanical design and manufacturing strategy of the linkage. Of course after design comes further analysis, free body diagrams for the different linkage components were done and are shown in the figures on the next page.

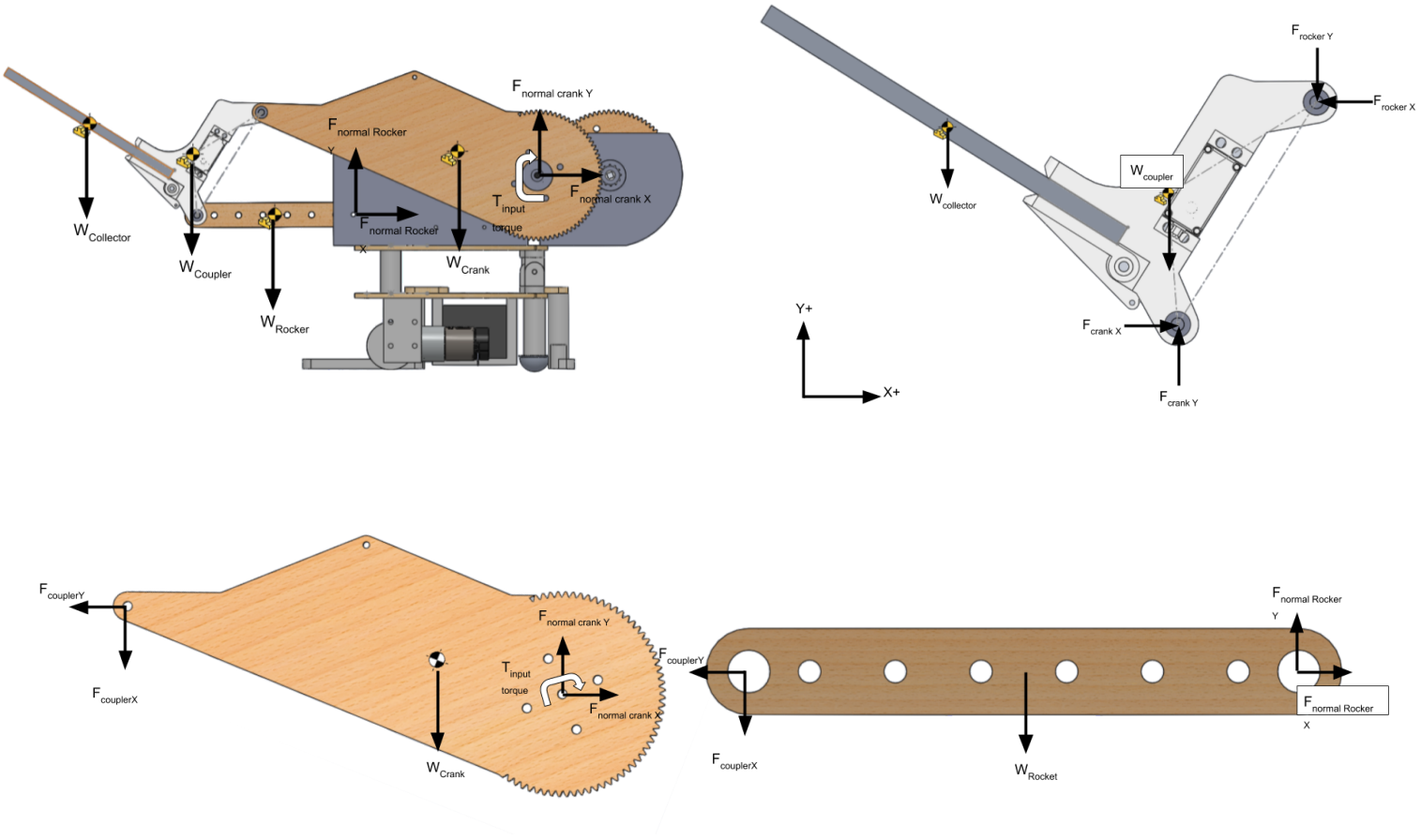


Figure 18: Clockwise from the top left, System FBD, Coupler FBD, Rocker FBD, Crank FBD

Sensors:

Our team utilized three main types of sensors, encoders, IR line trackers, and a limit switch, to accomplish the functionalities of the robot. Each of the two drive motors and the arm motor had a built in encoder that allowed us to accurately and control them with a PID. Threw our experimentation, we found that a Kp of .025, Ki of .02, and a Kd of -.6 gave us low overshoot and high accuracy for all three motors. We started off by using the Ziegler-Nichols Method to find our PID (Figure 19) but found that that was not giving us accurate results. Threw changing numbers high to low, positive to negative, we found that having a high negative Kd value with lower positive Kp and Ki values gave us high accuracy and low overshoot. Also, while at first we were confused as to why a negative Kd value worked over a positive one, we determined this was due to the way we coded the PID calculations as seen in Figure 20.

TABLE 1 Effects of independent P, I, and D tuning on closed-loop response.
For example, while K_I and K_D are fixed, increasing K_P alone can decrease rise time, increase overshoot, slightly increase settling time, decrease the steady-state error, and decrease stability margins.

	Rise Time	Overshoot	Settling Time	Steady-State Error	Stability
Increasing K_P	Decrease	Increase	Small Increase	Decrease	Degrade
Increasing K_I	Small Decrease	Increase	Increase	Large Decrease	Degrade
Increasing K_D	Small Decrease	Decrease	Decrease	Minor Change	Improve

Figure 19: Effects of increasing Kp, Ki, Kd on Rise Time, Overshoot, Settling Time, Steady-State Error, and Stability[1]

Ziegler–Nichols method

Control Type	K_p	K_i	K_d
P	$0.50K_u$	—	—
PI	$0.45K_u$	$0.54K_u/T_u$	—
PID	$0.60K_u$	$1.2K_u/T_u$	$3K_uT_u/40$

Figure 20: Ziegler-Nichols method for finding PID values [2]

In order to implement the PID control in the Calc() method of RBEPID.cpp, the proportional error term, derivative error term, and the integral error term was calculated as seen in Figure 21. The proportional error term was just the difference between the set-point motor position and the current motor position, which was already done. We then made the derivative error term the difference of the last error current error. Finally, the integral term is a bit more complicated and involved keeping a running average of the errors. For every new error added to the sum of errors, the oldest error was subtracted from the sum, and then the current average was taken for the integral term.

```
if((curPosition - setPoint < 0 && err_positive) || (curPosition - setPoint > 0 && !err_positive)) {
        clearIntegralBuffer();
        err_positive = !err_positive;
}
// calculate error
float err = setPoint - curPosition;
// calculate derivative of error
float err_d = last_error - err;
// calculate integral error. Keeping a running average of 16 values: Add the new value and subtract the oldest value
sum_error = sum_error + err - errors[err_index];
float err_i = sum_error / err_size;

// sum up the error value to send to the motor based off gain values.
float out = err * kp + err_d * kd + err_i * ki; // simple P controller
```

Figure 21: PID Control for the robot

We found that it was important to reset the sum of the integral errors however when the error switches sign. We did this by keeping track of the sign of the error, and when the current sign of the error did not match, the clearIntegralBuffer() method was called. All this method had to do was reset the running sum of the errors to 0. The array holding all the previous errors did

not need to be reset however because the items in this array are never added when calculating the integral term, it is only used for removing the oldest element.

The next critical sensor we utilized was the infrared line tracker. As seen in Figure 22, we used 4 total sensors from the array available. The middle two line sensors were used for the actual line following code and turning. By default the robot would go straight if they were both on the line, and if either sensor happens to go off the line, then the robot would gradually turn back on to the line. In addition, whenever our robot performed a turn, we designed it so that the robot would make most of the turn through interpolation, then for the last few degrees the robot would slow down and keep turning until these middle two sensors detected the line.

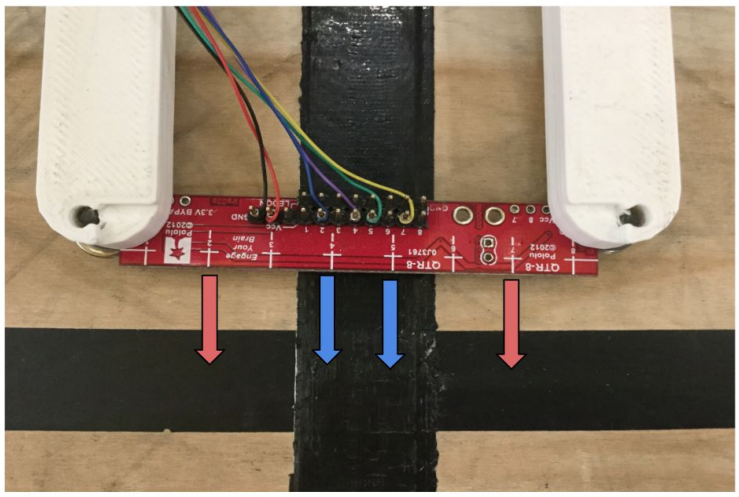


Figure 22: Line Follow Sensors

Inner Line Trackers(blue), outer line trackers(red)

The outer two line trackers were used for detecting intersections along the game board. These were a critical factor in determining how to get to either the near or far roof side and for reaching a target roof alignment line. We were able to count these lines by line following until the outer line trackers detected an intersection, incrementing the line counter. From here, we had

to use a seperate line follow for distance method which made use of both the middle line trackers and the encoders to track the distance traveled. This drove past the current line to prevent multiple intersection readings from the same line so that we could go back to accurately looking for the next line.

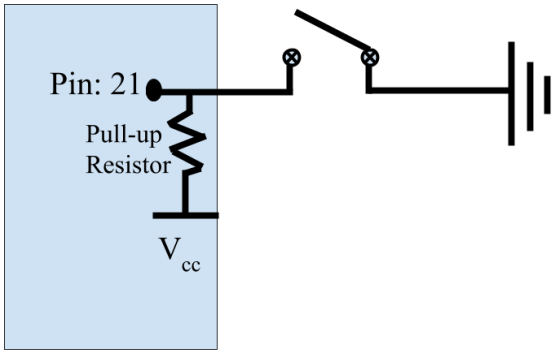


Figure 23: Limit Switch Circuit

The last of the sensors we used was the limit switch for zeroing our arm motor at the beginning of every program startup. As seen in Figure 23, we configured the limit switch to be connected to ground and the input pin with a pull-up resistor to V_{cc} which helped reduce noise along the wire. Since the switch was setup to be active low, we looked for a digital change of 1 to 0 as we gradually lower the arm until it connects with the limit switch. Once the motor arm is zerod, we were able to record consistent motor degrees for all pickup and dropoff angles in respect to this position.

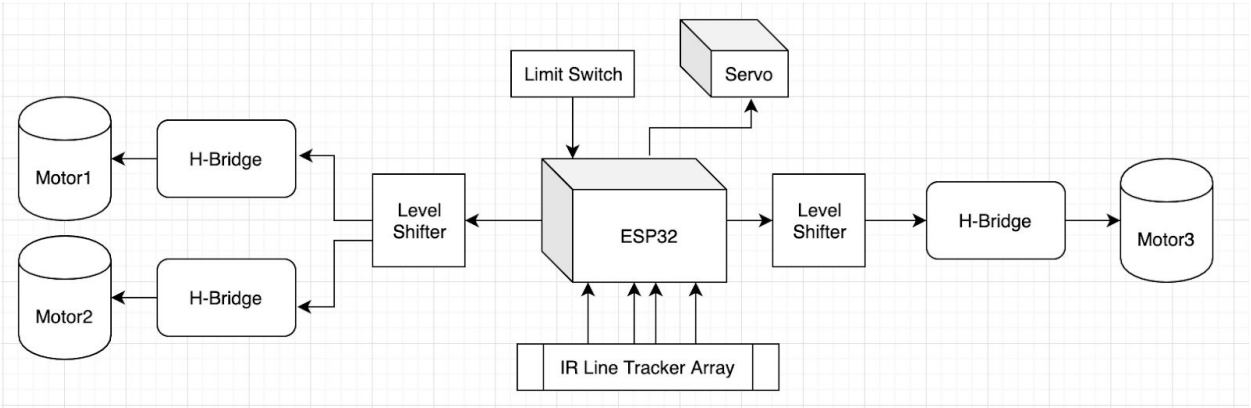


Figure 24: Circuit Diagram

We then wired each of these sensors along with the motors and servo following the logic in Figure 24. The two motors on the left correspond with the two drive motors while the motor on the right is the arm motor. Each of the motors are connected to an H-Bridge which made sure the motors turned the right direction and speed as well as interpreted the motor encoders. These H-Bridges are then connected to level shifters which shift the voltage from 5 V to 3.3 V. The 4 line tracker array sensors were analog inputs into the ESP32. The limit switch as described earlier connects to the ESP as a digital input. Finally the servo motor controlling the gripper is directly controlled by the ESP as an output.

Code:

Culminating the mechanical design and sensor integration, we tied everything together with a robust state machine. This not only simplified the code into easy to follow discrete steps, it allowed us to handle multiple data input and outputs at the same time. This way we were to be driving the robot forward and setting the arm to a given angle while also polling the inner line

checkers to stay on the line and polling the outer line checkers for intersections. We also made sure to utilize reusable states and methods while inside other states. For example, we had two reusable states for either line following until an intersection or line following for a given distance, that then each made use of our reusable line follow method.

This kind of abstraction from line following is also something our group focused heavily on in trying to simplify the hierarchy of the code from the highest level down. With a multitude of possible configurations the field can be in, we tried breaking down the process of navigating the field into simpler and simpler methods. Using the staging area as our home base, we designed our highest level method to essentially drive the robot along an L shape (Figure 25) that was our foundational method of going to and from the staging area and roof as well as crossing to the other side of the field. From here, we further abstracted this method into two turning methods and two driving to target line methods, whose breakdown you can see in Figure 26.

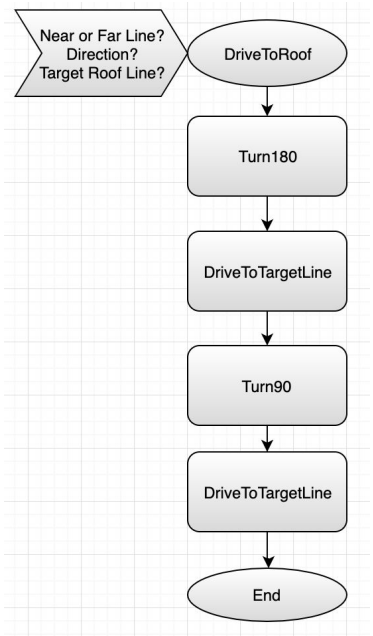


Figure 25: DriveToRoof method
Even though we called it DriveToRoof,
It could still be used for driving back.

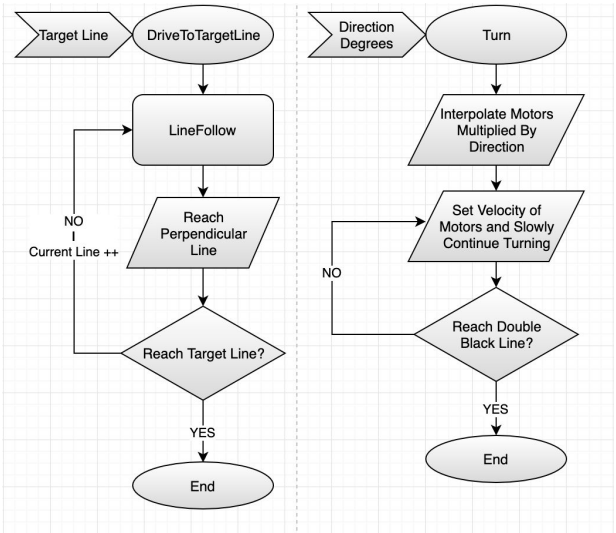


Figure 26: Sub-methods

Power and Current:

Deciding to drive our robot at 4 inches a second, we found through the spreadsheet, Figure 27, that the current for each drive motor would be 1.76 Amps. For the arm, we did a solidworks torque motion study, Figure 28, to find the torque needed to lift the arm at peak conditions. Using that we approximated that the peak current needed to do so was 3 amps. This adds up to 6.52, which is below the 7.5 amp fuse.

50:1 Metal Gearmotor 37Dx70L mm with 64 CPR Encoder (Spur Pinion) Data

Tstall (in-lbf)	18.125
wnoload (RPM)	200
Inoload (A)	0.15
Istall (A)	5.5
Ref Voltage	12
R_A	2.181818

$V_T = R_A I_A + E_A$

Speed (RPM)	Torque (N-m)	Torque (in-lbf)	Current (A)	Pout (W)	Efficiency	Pin (W)	Heat (W)	back-EMF (V)
0	2.05	18.13	5.50	0.00	0.00	66.00	66.00	0.00
10	1.95	17.22	5.23	2.04	3.24	62.79	60.75	0.58
20	1.84	16.31	4.97	3.86	6.48	59.58	55.72	1.17
30	1.74	15.41	4.70	5.47	9.70	56.37	50.90	1.75
40	1.64	14.50	4.43	6.86	12.91	53.16	46.30	2.33
50	1.54	13.59	4.16	8.04	16.10	49.95	41.91	2.92
60	1.43	12.69	3.90	9.01	19.27	46.74	37.73	3.50
70	1.33	11.78	3.63	9.76	22.41	43.53	33.77	4.09
80	1.23	10.88	3.36	10.29	25.53	40.32	30.03	4.67
90	1.13	9.97	3.09	10.61	28.60	37.11	26.50	5.25
100	1.02	9.06	2.83	10.72	31.63	33.90	23.18	5.84
110	0.92	8.16	2.56	10.61	34.59	30.69	20.08	6.42
120	0.82	7.25	2.29	10.29	37.45	27.48	17.19	7.00
130	0.72	6.34	2.02	9.76	40.20	24.27	14.51	7.59
140	0.61	5.44	1.76	9.01	42.76	21.06	12.05	8.17
150	0.51	4.53	1.49	8.04	45.05	17.85	9.81	8.75
160	0.41	3.63	1.22	6.86	46.87	14.64	7.78	9.34
170	0.31	2.72	0.95	5.47	47.84	11.43	5.96	9.92
180	0.20	1.81	0.69	3.86	46.96	8.22	4.36	10.51
190	0.10	0.91	0.42	2.04	40.66	5.01	2.97	11.09
200	0.00	0.00	0.15	0.00	0.00	1.80	1.80	11.67

Figure 27: Excel Spreadsheet to calculate motor currents

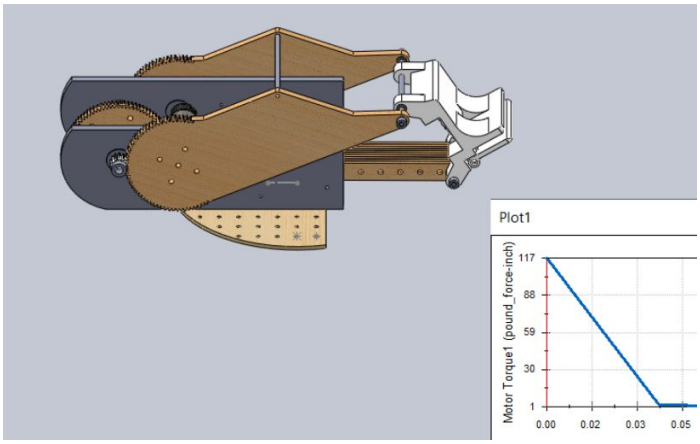


Figure 28: Solidworks Torque Motion Study

Results

Once assembled, our robot was able to fully accomplish the tasks at hand. We could remove and place solar panels ad both positions and materials consistently as well as line follow and go under the course to the other side when needed.

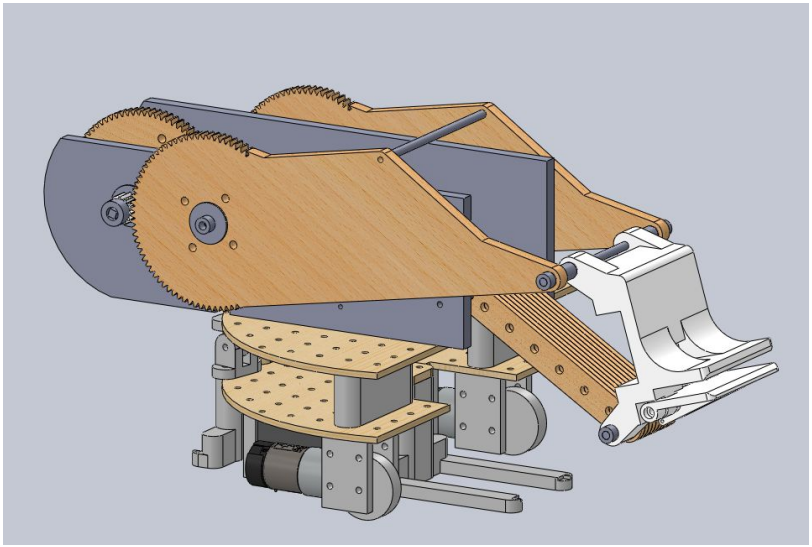


Figure 29: Solidworks Model of the Robot.

Discussion

Through all our work discussing strategy on where our robot should be when placing the solar panels and picking it up, as well as specific angle and positions for the gripper we were able to come up with a 4-bar that was able to do all that was required in a very systematic and calculated way so when it was time to code the robot to manipulate the solar panels, it was just a matter of finding the correct motor positions to get the desired positions which we tested earlier in the term. Furthermore, with deciding to design our robot early into the term, we were able to go through many iterations of improving the design until we decided on the one we used for the challenge. Many different aspects of the design were taken into account, mainly the materials and how we were going to drive the 4-bar.

During testing, we were able to traverse the full course, and lift both types of plates, but eventually because of fatigue, the two wood arms stripped on the gears. Noticing that this could be a problem, we decided before this happened to cut acrylic versions of the arms and big gears. As we were prepared for this occurrence, it only took us half an hour to replace the arms with the much stronger acrylic counterparts which have yet to fail. Another occurrence of problems we found was that the wheels would rub up against the plastic on their holder. Causing the wheels to stop turning, causing issues with line following and interpolation. To fix this problem, we used a dremel with a milling bolt to dig out the plastic around the wheels to stop the collision. This

along with driving the robot straight forward and loosening/tightening the wheels so that it would go straight greatly increased our ability to traverse the course.

For most of the competition we opted to use line following instead of PID and interpolation, except for on the lifting mechanism. Our line follower was capable of taking in the amount of black line it could go over before stopping, allowing us to go to specific lines easily with one method which we first developed in an early lab. For the lifting arm, simple interpolation with accurate values found through testing for each position seemed to work best, as well as very accurate PID values found in the prior lab. We would always zero the arm at the same position using a limit switch which was able to give us consistent positions for interpolation and placing the solar collectors. One of our biggest coding issues was turning. At first we used the interpolation commands that we made in earlier labs to turn the needed degrees, but found that it could be inaccurate. Because of this we decided to turn in a combination of interpolation and line following. For this method we would turn around 20 degrees less than what we would want through interpolation, such as turning 70 degrees when we want to go 90, than we would turn the rest with a constant velocity until both middle line followers hit the black line, where we could continue to drive to where we need to go.

In the end the programming and mechanical design gave us little trouble. Using Solidworks allowed us to easily prototype our robot as well as see what possible issues may arise before even touching the wood to make it. Using the state machine made it very easy to see what parts of the code had issues to easily troubleshoot issues that may come up in testing.

Conclusion

Overall, our robot met all expectations we set for it. In the end, our robot was able to perform every task needed on the day of the presentation. Through proper communication and delegation of tasks with strict deadlines, our team was able to have a fully functional design and code that moved the robot to the correct positions for either side the weekend before the presentation. This gave us plenty of time to focus on the lifting mechanism code, testing the best positions and distances to consistently place the solar panels, as well as fine tune the movement code to almost never fail in movement.

An area of possible issues that we had identified early on was that our wiring of the robot could have been much better. In the end, we had no issues as we made sure each component was hooked up correctly, but if an issue were to come up, it would be hard to diagnose the issue and come up with a solution.

Comments

Garett:

Im very happy with the way the project turned out. I feel like everyone was able to learn about the different mechanisms coming together in the robot as well as see how proper design can help make a project perform smoothly.

Conrad:

This final project has been the most fun and successful experience I've had with robotics to this date. Every member put in a significant amount of effort and time into the project and as a

result we were actually able to successfully finish our robot in a relatively timely and healthy manner.

Dhionis:

This is one of my favorite projects at WPI to date. I was lucky enough to work with very cooperative people and very happy we were able to pull together make the robot happen.

Works Cited

[1] "PID Control System Analysis and Design," EEE Control Systems Magazine, 2006. [Online]. Available: https://www.researchgate.net/publication/3207704_PID_control_system_analysis_and_design. [Accessed 23 September 2019].

[2] "PID Controller," Wikipedia: The Free Encyclopedia, [Online]. Available: https://en.wikipedia.org/wiki/PID_controller. [Accessed 25 September 2019].

Appendices

Appendix A: Contributions

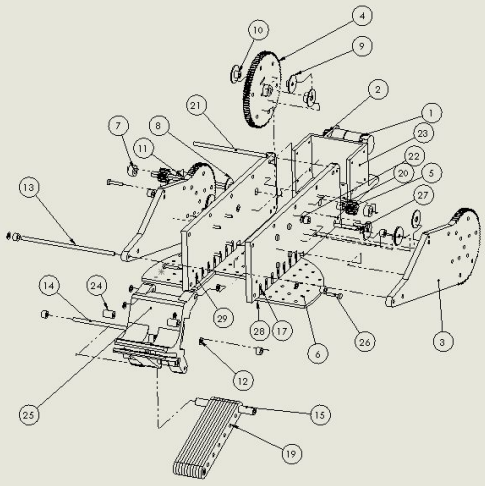
Name	Percent Contribution
Garett	33.3
Conrad	33.3
Dhionis	33.3

Appendix B: Github Link

https://github.com/RBE200x-lab/RBE2001Code19

Appendix C: Exploded View and Bill of Materials

ITEM NO.	PART NUMBER	QTY.	MATERIAL	UNIT WEIGHT LBS	EXTENDED WEIGHT LBS	COST $	EXTENDED COST$
1	37d-gearmotor-50-70-encoder	1	STEEL	2.17	2.17	00.00	00.00
2	12T Motor Pinion	1	PLA	.112	.112	1.10	1.10
3	84T Crank Arm	2	WOOD	.265	5.30	2.05	4.10
4	84T Gear	1	WOOD	.2	.2	2.00	2.00
5	12T Gear	2	PLA	.108	.216	1.05	1.10
6	Baseplate	1	WOOD	1.23	1.23	0.00	0.00
7	.25 Square Shaft Collar	5	STEEL	.11	.55	1.15	5.75
8	.25 Square Nylon Washer	4	NYLON	.05	.2	0.35	1.40
9	.25 Round Nylon Washer	8	NYLON	.06	.48	0.35	2.80
10	.25 Round Shaft Collar	9	STEEL	.08	.72	0.55	4.95
11	Cross Brace	1	PLA	.2	.2	4.65	4.65
12	Fiber Washer	10	FIBER	.01	.1	.15	1.50
13	Top Jaw Shaft	1	STEEL	.2	.2	2.30	2.30
14	Lower Jaw Shaft	1	STEEL	.24	.24	2.10	2.10
15	Rocker Shaft Spacers	2	PLA	.11	.22	0.12	0.24
17	Gearbox Plate	2	WOOD	.28	.56	5.65	11.30
18	84T Crank Arm Hub	1	PLA	.05	.05	0.20	0.20
19	Rocker Assembly	1	WOOD	1.26	1.26	3.10	3.10
20	.25 Square Bushing	2	NYLON	.06	.12	0.35	0.70
21	Crank Crossbrace	1	STEEL	1.26	1.26	2.25	2.25
22	Short Nylon Bushing	2	NYLON	.15	.30	0.30	0.60
23	Steel Shaft	1	STEEL	.87	.87	1.28	1.28
24	Top Jaw Shaft Spacer	2	PLA	.15	.30	0.20	0.40
25	Gripper Assembly	1	PLA	1.25	1.25	9.54	9.54
26	8-32 Screw	12	STEEL	.05	.60	0.22	2.64
27	Crank Link Shaft	2	STEEL	.75	1.50	2.10	2.10
28	10-32 Bolt	14	STEEL	.07	2.80	0.25	3.50
29	10-32 Nut	14	STEEL	.05	0.70	0.12	1.68



TEAM 19	LINKAGE GEARBOX ASSEMBLY	SCALE: 1:1	10/10/2019

Appendix D: Code

StudentsRobot.h

```
/*
 * StudentsRobot.h
 *
 *  Created on: Dec 28, 2018
 *      Author: hephaestus
 */

#ifndef STUDENTSROBOT_H_
#define STUDENTSROBOT_H_
#include "config.h"
#include <Arduino.h>
#include "src/pid/ServoEncoderPIDMotor.h"
#include "src/pid/HBridgeEncoderPIDMotor.h"
#include "src/pid/ServoAnalogPIDMotor.h"
#include <ESP32Servo.h>

#include "DriveChassis.h"

/**
 * @enum RobotStateMachine
 * These are sample values for a sample state machine.
 * Feel free to add ot remove values from here
 */
enum RobotStateMachine {
        StartupRobot = 0,
        StartRunning = 1,
        Running = 2,
        GoToFirstPickup = 100,
        Pickup_Plate = 3,
        DriveToRoofTest = 4,
        WaitForDropoff = 5,
        DropoffPlate = 6,
        GoToSafe = 7,
        DriveToPickupPlatePt1 = 8,
        DriveToPickupPlatePt2 = 9,
        LineUpToPlate = 200,
        WaitForPickup = 10,
```

```
        GoToSafe2 = 11,
        DriveToPlateDropoff1 = 12,
        DriveToPlateDropoff2 = 13,
        LowerPlate = 15,
        SwitchSidesPt1 = 16,
        SwitchSidesPt2 = 17,
        Halting = 20,
        Halt = 21,
        WAIT_FOR_MOTORS_TO_FINNISH = 22,
        WAIT_FOR_TIME = 23,
        LINE_FOLLOW_UNTIL_BLACK_LINE = 24,
        LINE_FOLLOW_FOR_DISTANCE = 25,
        ALIGN_ON_LINE = 26,
        LINE_FOLLOW_FOR_DISTANCE_BACKWARDS = 27,
        WAIT_FOR_ARM_TO_ZERO = 28,
        LineUpToPlate2 = 29


};
/*
 * @enum OffCourseSide
 * Records the side that the robot went off of last
 */
enum OffCourseSide {
        Center = 0, Left = 1, Right = 2

};
/**
 * @enum ComStackStatusState
 * These are values for the communications stack
 * Don't add any more or change these. This is how you tell the GUI
 * what state your robot is in.
 */
enum ComStackStatusState {
        Ready_for_new_task = 0,
        Heading_to_pickup = 1,
        Waiting_for_approval_to_pickup = 2,
        Picking_up = 3,
        Heading_to_Dropoff = 4,
        Waiting_for_approval_to_dropoff = 5,
        Dropping_off = 6,
        Heading_to_safe_zone = 7,
        Fault_failed_pickup = 8,
        Fault_failed_dropoff = 9,
```

```
        Fault_excessive_load = 10,
        Fault_obstructed_path = 11,
        Fault_E_Stop_pressed = 12
};
/**
 * @class StudentsRobot
 */
class StudentsRobot {
private:
        PIDMotor * motor1;
        PIDMotor * motor2;
        PIDMotor * motor3;
        Servo * servo;
        float lsensorVal = 0;
        float rsensorVal = 0;
        long nextTime = 0;
        long startTime = 0;
        RobotStateMachine nextStatus = StartupRobot;
        OffCourseSide offSide = Center;
        DrivingChassis * chassis;
        int lineThreashold = 2000;
        float startPos = 0;
        float distanceToTravel = 0;
        int direction = 1;
        int lineNum = 0;
        int lineState = 0;
        int roofState = 0;
        float pickupMaterial;
        float dropoffAngle;
        float dropoffPosition;
        int turnDirection = 1;
        int turn180Dir = 1;
        int turnAdjustment =0;
        float speedAdjustment = 1;
        int roofLines = 0;
        int liftarmdeg= 100;
        int totalTrips=0;
        int turnState = 0;

public:
        /**
         * Constructor for StudentsRobot
         *
         * attach the 4 actuators
```

```
         *
         * these are the 4 actuators you need to use for this lab
         * all 4 must be attached at this time
         * DO NOT reuse pins or fail to attach any of the objects
         *
         */
        StudentsRobot(PIDMotor * motor1, PIDMotor * motor2, PIDMotor * motor3,
                      Servo * servo);
        /**
         * Command status
         *
         * this is sent upstream to the Java GUI to notify it of current state
         */
        ComStackStatusState myCommandsStatus = Ready_for_new_task;
        /**
         * This is internal data representing the runtime status of the robot for use in its state
machine
         */
        RobotStateMachine status = StartupRobot;
        /**
         * Approve
         *
         * @param buffer A buffer of floats containing nothing
         *
         * the is the event of the Approve button pressed in the GUI
         *
         * This function is called via coms.server() in:
         * @see RobotControlCenter::fastLoop
         */
        void Approve(float * buffer);
        /**
         * ClearFaults
         *
         * @param buffer A buffer of floats containing nothing
         *
         * this represents the event of the clear faults button press in the gui
         *
         * This function is called via coms.server() in:
         * @see RobotControlCenter::fastLoop
         */
        void ClearFaults(float * buffer);
        /**
         * EStop
         *
```

```
         * @param buffer A buffer of floats containing nothing
         *
         * this represents the event of the EStop button press in the gui
         *
         * This is called whrn the estop in the GUI is pressed
         * All motors shuld hault and lock in position
         * Motors should not go idle and drop the plate
         *
         * This function is called via coms.server() in:
         * @see RobotControlCenter::fastLoop
         */
        void EStop(float * buffer);
        /**
         * PickOrder
         *
         * @param buffer A buffer of floats containing the pick order data
         *
         * buffer[0]  is the material, aluminum or plastic.
         *
         * buffer[1]  is the drop off angle 25 or 45 degrees
         *
         * buffer[2]  is the drop off position 1, or 2
         *
         * This function is called via coms.server() in:
         * @see RobotControlCenter::fastLoop
         */
        void PickOrder(float * buffer);

        /**
         * pidLoop This functoion is called to let the StudentsRobot controll the running of the
PID loop functions
         *
         * The loop function on all motors needs to be run when this function is called and
return fast
         */
        void pidLoop();
        /**
         * updateStateMachine use the stub state machine as a starting point.
         *
         * the students state machine can be updated with this function
         */
        void updateStateMachine();

        void FollowLine(int dir);
```

```
        void FollowLineBackwards();

        void DriveToTargetLine(RobotStateMachine currentState, int endState);

        void DriveToTargetLineAndTurn90(RobotStateMachine currentState,
                        RobotStateMachine endState);

        bool DriveToRoof(int roofSide, int roofLine, RobotStateMachine currentState,
                        RobotStateMachine endState);
        void AlignOnLine(RobotStateMachine endState);

        int checkBoardSide(int angle);

        void updateRoofSideDir(int angle, int side);

        int getOppositeRoofSide(int side);

        bool TurnTargetDeg(int smallTurn, int fullTurn, int turnDir,int time,
RobotStateMachine currentState);
};

#endif /* STUDENTSROBOT_H_ */
```

StudentsRobot.cpp

```
/*
 * StudentsRobot.cpp
 *
 *  Created on: Dec 28, 20
 *      Author: hephaestus
 *      Author: GarettR
 *                  Author: Conrad Tulig
 *                  Author: Dhionis Zhidro
 *
 */
```

```
#include "StudentsRobot.h"

StudentsRobot::StudentsRobot(PIDMotor * motor1, PIDMotor * motor2,
                PIDMotor * motor3, Servo * servo) {
        Serial.println("StudentsRobot::StudentsRobot constructor called here ");
        this->servo = servo;
        this->motor1 = motor1;
        this->motor2 = motor2;
        this->motor3 = motor3;

        // Set the PID Clock gating rate. The PID must be 10 times slower than the motors
update rate
        motor1->myPID.sampleRateMs = 1; //
        motor2->myPID.sampleRateMs = 1; //
        motor3->myPID.sampleRateMs = 1;  // 10khz H-Bridge, 0.1ms update, 1 ms PID

        // Set default P.I.D gains
        motor1->myPID.setpid(0.025, 0.02, -0.6);
        motor2->myPID.setpid(0.025, 0.02, -0.6);
        motor3->myPID.setpid(0.025, 0.02, -0.6);

        motor1->velocityPID.setpid(0.1, 0, 0);
        motor2->velocityPID.setpid(0.1, 0, 0);
        motor3->velocityPID.setpid(0.1, 0, 0);
        // compute ratios and bounding
        double motorToWheel = 3;
        motor1->setOutputBoundingValues(-255, //the minimum value that the output takes
(Full reverse)
                        255, //the maximum value the output takes (Full forward)
                        0, //the value of the output to stop moving
                        125, //a positive value subtracted from stop value to creep backward
                        125, //a positive value added to the stop value to creep forwards
                        16.0 * // Encoder CPR
                                        50.0 * // Motor Gear box ratio
                                        motorToWheel * // motor to wheel stage ratio
                                        (1.0 / 360.0) * // degrees per revolution
                                        2, // Number of edges that are used to increment the
value
                        480, // measured max degrees per second
                        150 // the speed in degrees per second that the motor spins when the
hardware output is at creep forwards
                        );
        motor2->setOutputBoundingValues(-255, //the minimum value that the output takes
(Full reverse)
```

```
                        255, //the maximum value the output takes (Full forward)
                        0, //the value of the output to stop moving
                        125, //a positive value subtracted from stop value to creep backward
                        125, //a positive value added to the stop value to creep forwards
                        16.0 * // Encoder CPR
                                        50.0 * // Motor Gear box ratio
                                        motorToWheel * // motor to wheel stage ratio
                                        (1.0 / 360.0) * // degrees per revolution
                                        2, // Number of edges that are used to increment the
value
                        480, // measured max degrees per second
                        150     // the speed in degrees per second that the motor spins when the
hardware output is at creep forwards
                        );
        motor3->setOutputBoundingValues(-255, //the minimum value that the output takes
(Full reverse)
                        255, //the maximum value the output takes (Full forward)
                        0, //the value of the output to stop moving
                        125, //a positive value subtracted from stop value to creep backward
                        125, //a positive value added to the stop value to creep forwards
                        16.0 * // Encoder CPR
                                        50.0 * // Motor Gear box ratio
                                        1.0 * // motor to arm stage ratio
                                        (1.0 / 360.0) * // degrees per revolution
                                        2, // Number of edges that are used to increment the
value
                        1400, // measured max degrees per second
                        50 // the speed in degrees per second that the motor spins when the
hardware output is at creep forwards
                        );
        // Set up the line tracker
        pinMode(MIDDLE_LEFT, ANALOG);
        pinMode(MIDDLE_RIGHT, ANALOG);
        pinMode(FAR_LEFT, ANALOG);
        pinMode(FAR_RIGHT, ANALOG);

        pinMode(LIMIT_SWITCH, INPUT_PULLUP);

        pinMode(H_BRIDGE_ENABLE, OUTPUT);
        chassis = new DrivingChassis(motor1, motor2, 225, 25);
}
/**
 * Seperate from running the motor control,
 * update the state machine for running the final project code here
```

```
 */
void StudentsRobot::updateStateMachine() {
	long now = millis();
	switch (status) {
	case StartupRobot:
		//Do this once at startup
		status = StartRunning;
		Serial.println("StudentsRobot::updateStateMachine StartupRobot here ");
		break;
	case StartRunning:
		Serial.println("Start Running");

		digitalWrite(H_BRIDGE_ENABLE, 1);
		// Start an interpolation of the motors
		//motor1->startInterpolationDegrees(720, 1000, SIN);
		//motor2->startInterpolationDegrees(-720, 1000, SIN);
		//motor3->startInterpolationDegrees(motor3->getAngleDegrees(), 1000, SIN);

		//chassis->turnDegrees(360, 5000);
		Serial.println("We made it here");

		motor3->setVelocityDegreesPerSecond(100);
		status = WAIT_FOR_ARM_TO_ZERO; // set the state machine to wait for the
motors to finish
		nextStatus = Running; // the next status to move to when the motors finish

		startTime = now + 1000; // the motors should be done in 1000 ms
		nextTime = startTime + 1000; // the next timer loop should be 1000ms after the
motors stop
		break;
	case Running:
		// Set up a non-blocking 1000 ms delay
		status = WAIT_FOR_TIME;
		nextTime = nextTime + 1000; // ensure no timer drift by incremeting the target
		// After 1000 ms, come back to this state
		lineState = 0;
		nextStatus = Running;

		servo->write(0);
		//motor1->setVelocityDegreesPerSecond(-260);
		//motor2->setVelocityDegreesPerSecond(260);

		if (myCommandsStatus == Heading_to_pickup) {
			status = GoToFirstPickup;
```

```
                        //nextStatus = GoToFirstPickup;
                        roofLines = checkBoardSide(dropoffAngle);
                        updateRoofSideDir(dropoffAngle, dropoffPosition);
                }

                //Motor Angle Prints
                //Serial.println("motor 1 " + String(motor1->getAngleDegrees()));
                //Serial.println("motor 2 " + String(motor2->getAngleDegrees()));

                //Line Follow Prints
                /*Serial.print("Far left: ");
                 Serial.print(analogRead(FAR_LEFT));
                 Serial.print(",   Middle left: ");
                 Serial.print(analogRead(MIDDLE_LEFT));
                 Serial.print(",   Middle right: ");
                 Serial.print(analogRead(MIDDLE_RIGHT));
                 Serial.print(",   Far right: ");
                 Serial.println(analogRead(FAR_RIGHT));
                 */
                // Do something
                if (!digitalRead(0))
                        Serial.println(
                                        " Running State Machine " + String((now - startTime)));
                break;
        case GoToFirstPickup:
                chassis->resetMotorPos();
                startPos = (motor1->getAngleDegrees() + motor2->getAngleDegrees()) / 2;
                distanceToTravel = 320;
                direction = 1;
                status = LINE_FOLLOW_FOR_DISTANCE;
                nextStatus = Pickup_Plate;
                myCommandsStatus = Waiting_for_approval_to_pickup;
                if (dropoffAngle == 45) {
                        liftarmdeg = -2350;
                } else if (dropoffAngle == 25) {
                        liftarmdeg = -2500;
                }

                break;
        case Pickup_Plate:
                if (myCommandsStatus == Picking_up) {
                        servo->write(105);
                        motor3->startInterpolationDegrees(liftarmdeg, 3000, SIN);
delay
```

```
                nextTime = millis() + 3000;
                status = WAIT_FOR_TIME;
                nextStatus = DriveToRoofTest;
                myCommandsStatus = Heading_to_Dropoff;
        }

        break;
    case DriveToRoofTest:
        Serial.println("Driving to roof test");
        if (dropoffAngle == 25 && dropoffPosition == 2)
                turnAdjustment = -1;
        if (DriveToRoof(dropoffPosition, roofLines, DriveToRoofTest,
                        LineUpToPlate2)) {

        }
        break;
    case LineUpToPlate2:
        chassis->resetMotorPos();
        startPos = (motor1->getAngleDegrees() + motor2->getAngleDegrees()) / 2;
        if (dropoffAngle == 45) {
                distanceToTravel = 0;
        } else if (dropoffAngle == 25) {
                distanceToTravel = 10;
        }
        status = LINE_FOLLOW_FOR_DISTANCE;
        nextStatus = WaitForDropoff;
        myCommandsStatus = Waiting_for_approval_to_dropoff;
        break;
    case WaitForDropoff:
        if (myCommandsStatus == Dropping_off) {
                status = DropoffPlate;
                nextStatus = GoToSafe;
                myCommandsStatus = Heading_to_safe_zone;
        }
        break;
    case DropoffPlate:
        servo->write(0);
        status = WAIT_FOR_TIME;
        nextTime = millis() + 2000;
        myCommandsStatus = Heading_to_safe_zone;
        break;
    case GoToSafe:
        chassis->resetMotorPos();
        chassis->driveForward(500, 1500);
```

```
                status = WAIT_FOR_MOTORS_TO_FINNISH;
                nextStatus = DriveToPickupPlatePt1;

                break;
        case DriveToPickupPlatePt1:
                if (dropoffAngle == 45 && dropoffPosition == 1)
                        turnAdjustment = -2;
                if (dropoffAngle == 45 && dropoffPosition == 2)
        turnAdjustment = 1;
//              if(dropoffAngle == 45)
//                      turnAdjustment = 1;

                if (DriveToRoof(1, dropoffPosition, DriveToPickupPlatePt1,
                                DriveToPickupPlatePt2)) {
                        updateRoofSideDir(dropoffAngle,
                                        getOppositeRoofSide(dropoffPosition));
                        myCommandsStatus = Heading_to_pickup;
                        turnAdjustment = 0;
                }
                break;
        case DriveToPickupPlatePt2:
                if (dropoffAngle == 45)
                        motor3->startInterpolationDegrees(-1700, 1000, SIN);
                if (dropoffAngle == 25)
                        motor3->startInterpolationDegrees(-2500, 1000, SIN);
                DriveToRoof(getOppositeRoofSide(dropoffPosition), roofLines - 1,
                                DriveToPickupPlatePt2, LineUpToPlate);

                break;
        case LineUpToPlate:
                chassis->resetMotorPos();
                startPos = (motor1->getAngleDegrees() + motor2->getAngleDegrees()) / 2;
                if (dropoffAngle == 45) {
                        distanceToTravel = 60;
                } else if (dropoffAngle == 25) {
                        distanceToTravel = 90;
                }
                status = LINE_FOLLOW_FOR_DISTANCE;
                nextStatus = WaitForPickup;
                myCommandsStatus = Waiting_for_approval_to_pickup;
                break;
        case WaitForPickup:
                if (myCommandsStatus == Picking_up) {
                        servo->write(105);
```

```
                        status = WAIT_FOR_TIME;
                        nextTime = millis() + 2000;
                        nextStatus = GoToSafe2;
                        if (dropoffAngle == 45)
                                motor3->startInterpolationDegrees(-2400, 1000, SIN);
                        if (dropoffAngle == 25)
                                motor3->startInterpolationDegrees(-2200, 1000, SIN);
                }
                break;
        case GoToSafe2:

                chassis->resetMotorPos();
                chassis->driveForward(500, 1500);
                status = WAIT_FOR_MOTORS_TO_FINNISH;
                nextStatus = DriveToPlateDropoff1;
                dropoffPosition = getOppositeRoofSide(dropoffPosition);
                myCommandsStatus = Heading_to_Dropoff;

                break;
        case DriveToPlateDropoff1:
                if (dropoffAngle == 25 && dropoffPosition == 1)
                        turnAdjustment = -1;
                if (dropoffAngle == 25 && dropoffPosition == 2)
                        turnAdjustment = -1;
                if (dropoffAngle == 45 && dropoffPosition == 2)
                        turnAdjustment = -1;

                if (DriveToRoof(1, dropoffPosition, DriveToPlateDropoff1,
                                DriveToPlateDropoff2)) {

                        turnAdjustment = 0;
                }
                break;
        case DriveToPlateDropoff2:
                motor3->startInterpolation(0, 3000, SIN);
                //              if (DriveToRoof(getOppositeRoofSide(dropoffPosition),
roofLines-1,
                //                              DriveToPlateDropoff2, LowerPlate)) {
                //                      Serial.println("Why are we still here?");
                //                      myCommandsStatus = Ready_for_new_task;
                //              }
                chassis->resetMotorPos();
                chassis->driveForward(50, 1500);
                status = LowerPlate;
```

```
            myCommandsStatus = Waiting_for_approval_to_dropoff;

            break;
    case LowerPlate:
            if (myCommandsStatus == Dropping_off) {
                    servo->write(0);
                    //u9motor3->startInterpolationDegrees(-100, 500, SIN);
                    chassis->resetMotorPos();
                    chassis->driveForward(50, 1500);

                    status = WAIT_FOR_MOTORS_TO_FINNISH;
                    if (totalTrips == 0)
                            nextStatus = SwitchSidesPt1;
                    else {
                            myCommandsStatus = Ready_for_new_task;
                            status = Halting;
                    }

            }

            break;
    case SwitchSidesPt1:
            if (DriveToRoof(2, 4, SwitchSidesPt1, SwitchSidesPt2)) {
                    turn180Dir = 0;
                    turnDirection = -turnDirection;
            }
            break;
    case SwitchSidesPt2:
            if (DriveToRoof(4, 1, SwitchSidesPt2, Halting)) {
                    turn180Dir = 1;
                    myCommandsStatus = Ready_for_new_task;
                    totalTrips++;

            }
            break;
    case LINE_FOLLOW_UNTIL_BLACK_LINE:
            if (analogRead(MIDDLE_LEFT) >= lineThreashold
                            && analogRead(MIDDLE_RIGHT) >= lineThreashold
                            && analogRead(FAR_LEFT) >= lineThreashold
                            && analogRead(FAR_RIGHT) >= lineThreashold) {
                    Serial.println("Stop");
                    motor1->setVelocityDegreesPerSecond(0);
                    motor2->setVelocityDegreesPerSecond(0);
```

```
                status = nextStatus;
        } else {
                FollowLine(direction);
        }

        //status = WAIT_FOR_MOTORS_TO_FINNISH;
        //nextStatus = LineFollow;
        break;
        /*
         * startPos= (motor1->getAngleDegrees() + motor2->getAngleDegrees())/2;
         *      distanceToTravel= 500;
         *      direction = -1;
         */
case LINE_FOLLOW_FOR_DISTANCE:
        Serial.println("Starting distance linefollow");
        Serial.print("Distance traveled: ");
        Serial.print(chassis->distanceTraveled(startPos));
        Serial.print(" Distance Wanted: ");
        Serial.println(distanceToTravel);
        if (chassis->distanceTraveled(startPos) * direction
                        < distanceToTravel) {

                Serial.print("Distance traveled: ");
                Serial.print(chassis->distanceTraveled(startPos));
                Serial.print(" Distance Wanted: ");
                Serial.println(distanceToTravel);
                FollowLine(direction);
        } else {
                Serial.println("Stop");
                motor1->setVelocityDegreesPerSecond(0);
                motor2->setVelocityDegreesPerSecond(0);

                status = nextStatus;
        }

        break;
case LINE_FOLLOW_FOR_DISTANCE_BACKWARDS:
        if (-(chassis->distanceTraveled(startPos)) < distanceToTravel) {

                //Serial.print("Distance traveled: ");
                //Serial.print(chass
        } else {
                Serial.println("Stop");
                motor1->setVelocityDegreesPerSecond(0);
```

```
                motor2->setVelocityDegreesPerSecond(0);

                status = nextStatus;
        }
        break;
        //      case ZERO_ARM:

        //      break;
case WAIT_FOR_TIME:
        // Check to see if enough time has elapsed
        if (nextTime <= millis()) {
                // if the time is up, move on to the next state
                status = nextStatus;
        }
        break;
case WAIT_FOR_ARM_TO_ZERO:
        Serial.println(digitalRead(LIMIT_SWITCH));
        if (digitalRead(LIMIT_SWITCH) == 0) {
                motor3->setVelocityDegreesPerSecond(0);
                motor3->overrideCurrentPosition(0);

                status = nextStatus;
        }
        break;
case WAIT_FOR_MOTORS_TO_FINNISH:
        if (motor1->isInterpolationDone() && motor2->isInterpolationDone()
                        && motor3->isInterpolationDone()) {
                status = nextStatus;
        }
        break;
case ALIGN_ON_LINE:
        //Serial.println("Aligning");
        AlignOnLine(Halting);
        break;

case Halting:
        // save state and enter safe mode
        Serial.println("Halting State machine");
        //digitalWrite(H_BRIDGE_ENABLE, 0); // Disable and idle motors
        motor3->stop();
        motor2->stop();
        motor1->stop();

        status = Halt;
```

```
                break;
        case Halt:
                // in safe mode
                break;

        }
}

/**
 * This is run fast and should return fast
 *
 * You call the PIDMotor's loop function. This will update the whole motor control system
 * This will read from the concoder and write to the motors and handle the hardware interface.
 * Instead of allowing this to be called by the controller you may call these from a timer
interrupt.
 */
void StudentsRobot::pidLoop() {
        motor1->loop();
        motor2->loop();
        motor3->loop();
}
/**
 * Approve
 *
 * @param buffer A buffer of floats containing nothing
 *
 * the is the event of the Approve button pressed in the GUI
 *
 * This function is called via coms.server() in:
 * @see RobotControlCenter::fastLoop
 */
void StudentsRobot::Approve(float * buffer) {
        // approve the procession to new state
        Serial.println("StudentsRobot::Approve");

        if (myCommandsStatus == Waiting_for_approval_to_pickup) {
                myCommandsStatus = Picking_up;
        } else if (myCommandsStatus == Waiting_for_approval_to_dropoff) {
                myCommandsStatus = Dropping_off;
        } else {
                myCommandsStatus = Ready_for_new_task;
        }
}
/**
```

```
 * ClearFaults
 *
 * @param buffer A buffer of floats containing nothing
 *
 * this represents the event of the clear faults button press in the gui
 *
 * This function is called via coms.server() in:
 * @see RobotControlCenter::fastLoop
 */
void StudentsRobot::ClearFaults(float * buffer) {
        // clear the faults somehow
        Serial.println("StudentsRobot::ClearFaults");
        myCommandsStatus = Ready_for_new_task;
        status = Running;
}

/**
 * EStop
 *
 * @param buffer A buffer of floats containing nothing
 *
 * this represents the event of the EStop button press in the gui
 *
 * This is called whrn the estop in the GUI is pressed
 * All motors shuld hault and lock in position
 * Motors should not go idle and drop the plate
 *
 * This function is called via coms.server() in:
 * @see RobotControlCenter::fastLoop
 */
void StudentsRobot::EStop(float * buffer) {
        // Stop the robot immediatly
        Serial.println("StudentsRobot::EStop");
        myCommandsStatus = Fault_E_Stop_pressed;
        status = Halting;

}
/**
 * PickOrder
 *
 * @param buffer A buffer of floats containing the pick order data
 *
 * buffer[0]  is the material, aluminum or plastic.
 * buffer[1]  is the drop off angle 25 or 45 degrees
```

```
 * buffer[2]  is the drop off position 1, or 2
 *
 * This function is called via coms.server() in:
 * @see RobotControlCenter::fastLoop
 */
void StudentsRobot::PickOrder(float * buffer) {
	pickupMaterial = buffer[0];
	dropoffAngle = buffer[1];
	dropoffPosition = buffer[2];
	Serial.println(
			"StudentsRobot::PickOrder Recived from : " + String(pickupMaterial)
					+ " " + String(dropoffAngle) + " "
					+ String(dropoffPosition));
	myCommandsStatus = Heading_to_pickup;
}

void StudentsRobot::FollowLine(int dir) { //1 forward, -1 backwards
	if (analogRead(MIDDLE_LEFT) >= lineThreashold
			&& analogRead(MIDDLE_RIGHT) >= lineThreashold) {
		motor1->setVelocityDegreesPerSecond(-260 * speedAdjustment * dir);
		motor2->setVelocityDegreesPerSecond(260 * speedAdjustment * dir);
		//Serial.println("Driving Forward");
		offSide = Center;
	} else if ((analogRead(MIDDLE_LEFT) < lineThreashold || offSide == Left)
			&& offSide != Right) {
		//Serial.println("Turning right");
		motor1->setVelocityDegreesPerSecond(-200 * speedAdjustment * dir);
		motor2->setVelocityDegreesPerSecond(260 * speedAdjustment * dir);
		offSide = Left;

	} else if (analogRead(MIDDLE_RIGHT) < lineThreashold || offSide == Right) {
		//Serial.println("Turning left");
		motor1->setVelocityDegreesPerSecond(-260 * speedAdjustment * dir);
		motor2->setVelocityDegreesPerSecond(200 * speedAdjustment * dir);
		offSide = Right;
	}
}
void StudentsRobot::FollowLineBackwards() { //1 forward, -1 backwards
	if (analogRead(MIDDLE_LEFT) >= lineThreashold
			&& analogRead(MIDDLE_RIGHT) >= lineThreashold) {
		motor1->setVelocityDegreesPerSecond(220);
		motor2->setVelocityDegreesPerSecond(-220);
		//Serial.println("Driving Forward");
		offSide = Center;
```

```
        } else if ((analogRead(MIDDLE_LEFT) < lineThreashold || offSide == Left)
                        && offSide != Right) {
                //Serial.println("Turning right");
                motor1->setVelocityDegreesPerSecond(-160);
                motor2->setVelocityDegreesPerSecond(220);
                offSide = Left;

        } else if (analogRead(MIDDLE_RIGHT) < lineThreashold || offSide == Right) {
                //Serial.println("Turning left");
                motor1->setVelocityDegreesPerSecond(-220);
                motor2->setVelocityDegreesPerSecond(160);
                offSide = Right;
        }
}
bool StudentsRobot::TurnTargetDeg(int smallTurn, int fullTurn, int turnDir,
                int time, RobotStateMachine currentState) {
        switch (turnState) {
        case 0:
                chassis->resetMotorPos();
                chassis->turnDegrees(smallTurn * turnDir, time);
                status = WAIT_FOR_MOTORS_TO_FINNISH;
                nextStatus = currentState;
                turnState = 1;
                break;
        case 1:
                if (analogRead(MIDDLE_LEFT) > lineThreashold
                                && analogRead(MIDDLE_RIGHT) > lineThreashold) {
                        motor1->setVelocityDegreesPerSecond(0);
                        motor2->setVelocityDegreesPerSecond(0);
                        turnState = 0;
                        return true;
                } else {
                        motor1->setVelocityDegreesPerSecond(250 * turnDir);
                        motor2->setVelocityDegreesPerSecond(250 * turnDir);
                }
                break;
        }
        return false;
}
void StudentsRobot::DriveToTargetLine(RobotStateMachine currentState,
                int endState) {
        if (lineNum > 0) {
                Serial.print("lineNum: ");
                Serial.println(lineNum);
```

```
                switch (lineState) {
                case 0:
                        Serial.println("lineState 0");

                        chassis->resetMotorPos();
                        startPos = (motor1->getAngleDegrees() + motor2->getAngleDegrees())
                                        / 2;
                        distanceToTravel = 25;
                        direction = 1;
                        status = LINE_FOLLOW_FOR_DISTANCE;
                        nextStatus = currentState;
                        lineState = 1;
                        break;

                case 1:
                        Serial.println("lineState 1");

                        lineNum--;
                        status = LINE_FOLLOW_UNTIL_BLACK_LINE;
                        nextStatus = currentState;
                        lineState = 0;
                        break;

                }
        } else {
                lineState = 0;
                roofState = endState;
        }
}

void StudentsRobot::DriveToTargetLineAndTurn90(RobotStateMachine currentState,
                RobotStateMachine endState) {
        switch (lineState) {
        case 0:
                Serial.println("lineState 0");

                status = LINE_FOLLOW_UNTIL_BLACK_LINE;
                nextStatus = currentState;
                lineState = 1;
                break;
        case 1:
                Serial.println("lineState 1");
```

```
                chassis->resetMotorPos();
                startPos = (motor1->getAngleDegrees() + motor2->getAngleDegrees()) / 2;
                distanceToTravel = 100;
                direction = 1;
                status = LINE_FOLLOW_FOR_DISTANCE;
                nextStatus = currentState;
                lineState = 2;
                break;
        case 2:
                Serial.println("lineState 3");

                chassis->resetMotorPos();
                chassis->turnDegrees(90, 3000);
                status = WAIT_FOR_MOTORS_TO_FINNISH;
                status = endState;
                lineState = 0;
                break;
        }

}
void StudentsRobot::AlignOnLine(RobotStateMachine endState) {
        //Serial.println("linestate"+String(lineState));
        switch (lineState) {
        case 0:
                Serial.println("Forward 100");
                chassis->resetMotorPos();
                startPos = 0;
                distanceToTravel = 40;
                direction = 1;
                status = LINE_FOLLOW_FOR_DISTANCE;
                nextStatus = ALIGN_ON_LINE;
                lineState = 1;
                Serial.println(status);
                break;
        case 1:

                if (analogRead(MIDDLE_LEFT) >= lineThreashold
                                && analogRead(MIDDLE_RIGHT) >= lineThreashold
                                && analogRead(FAR_LEFT) >= lineThreashold
                                && analogRead(FAR_RIGHT) >= lineThreashold) {
                        Serial.println("Stop");
                        motor1->setVelocityDegreesPerSecond(0);
                        motor2->setVelocityDegreesPerSecond(0);
```

```
                        status = endState;
                        break;
                } else if (analogRead(FAR_LEFT) >= lineThreashold) {
                        Serial.println("Left hit");
                        motor2->setVelocityDegreesPerSecond(0);
                        motor1->setVelocityDegreesPerSecond(180);
                        break;
                } else if (analogRead(FAR_RIGHT) >= lineThreashold) {
                        Serial.println("Right Hit");
                        motor1->setVelocityDegreesPerSecond(0);
                        motor2->setVelocityDegreesPerSecond(-180);
                        break;
                }

                else {
                        //Serial.println("LineFollowBackwards");
                        FollowLineBackwards();
                        break;
                }

                break;
        }

}

bool StudentsRobot::DriveToRoof(int roofSide, int roofLine,
                RobotStateMachine currentState, RobotStateMachine endState) {
        switch (roofState) {
        case 0:
                /*chassis->resetMotorPos();
                 chassis->turnDegrees((180 + turnAdjustment * 10) * turn180Dir, 6000);
                 status = WAIT_FOR_MOTORS_TO_FINNISH; */

                if (TurnTargetDeg(120, 180, turn180Dir, 2500, currentState)) {
                        nextStatus = currentState;
                        roofState = 1;
                        lineNum = roofSide;
                }
                break;
        case 1:
                DriveToTargetLine(currentState, 2);
                break;
        case 2:
                Serial.println("lineState 1");
```

```
                chassis->resetMotorPos();
                startPos = (motor1->getAngleDegrees() + motor2->getAngleDegrees()) / 2;
                distanceToTravel = 100;
                direction = 1;
                status = LINE_FOLLOW_FOR_DISTANCE;
                nextStatus = currentState;
                roofState = 3;
                turnDirection = -turnDirection;
                break;
        case 3:

                //chassis->resetMotorPos();
                //chassis->turnDegrees(90 * turnDirection, 4000);
                //status = WAIT_FOR_MOTORS_TO_FINNISH;

                if (TurnTargetDeg(70, 90, turnDirection, 1000, currentState)) {
                        nextStatus = currentState;
                        roofState = 4;
                        lineNum = roofLine;
                }

                break;
        case 4:
                DriveToTargetLine(currentState, 5);
                break;
        case 5:
                roofState = 0;
                status = endState;
                return true;
        }
        return false;
}

int StudentsRobot::checkBoardSide(int angle) {
        if (angle == 45) {
                turnDirection = -1;
                return 2;
        } else if (angle == 25) {
                turnDirection = 1;
                return 2;
        }
        return -1;
}
```

```
void StudentsRobot::updateRoofSideDir(int angle, int side) {
	if ((angle == 25 && side == 1) || (angle == 45 && side == 2))
		turn180Dir = -1;
	else
		turn180Dir = 1;
}

int StudentsRobot::getOppositeRoofSide(int side) {
	if (side == 1) {
		return 2;
	} else if (side == 2) {
		return 1;
	}
	return -1;
}
```

Appendix E:Pictures of Robot