Raider Robotics Engineering Notebook

2020-2021 - VEX Change Up

Andrew Kempen Nathan DuPont Everen Wegner Alex Kempen
Brian Kim Dylan Powers Joseph Weller

Note to the Reader

During the 2020-2021 season, many of our actions were dictated by the COVID-19 epidemic and the restrictions placed on in person meetings by our university. As such, we met virtually up until January, when we were finally able to meet in person.

Regarding formatting of our notebook, the astute reader will notice many of our sections are timestamped 1/25/2021. Before this point, this document was primarily written in Microsoft Word, where timestamps were not recorded. On 1/25/2021, we transitioned the notebook to LaTeX and began tracking timestamps, marking all sections from before the transition with their primary authors and the date 1/25/2021.

It should be noted that this notebook has not been organized sequentially by date, but rather by sections/revisions of mechanisms. This choice was made to improve readability of the document. As such, the timestamps of subsequent pages may skip around as in many cases, mechanisms and revisions of mechanisms were developed and documented in parallel.

Contents

1 Meetings **12**

1.1 4/25/2020 . . . 12

1.2 1/27/2021 . . . 13

1.3 1/28/2021 . . . 14

1.4 1/29/2021 . . . 16

1.5 1/30/2021 . . . 17

1.6 1/31/2021 . . . 19

1.7 2/03/2021 . . . 21

1.8 2/05/2021 . . . 22

1.9 2/06/2021 . . . 24

1.10 2/07/2021 . . . 27

1.11 2/10/2021 . . . 29

1.12 2/11/2021 . . . 31

1.13 2/12/2021 . . . 32

1.14 2/14/2021 . . . 33

1.15 2/17/2021 . . . 34

1.16 3/3/2021 . . . 35

1.17 3/13/2021 . . . 36

1.18 3/18/2021 . . . 37

1.19 3/24/2021 . . . 38

2 Overall Robot Design **39**

2.1 Early Season Brainstorming . . . 39

2.2 V1 Robot Design Cycle . . . 52

2.2.1 Overview . . . 52

2.2.2 Design Review . . . 53

2.3 V2 Robot . . . 56

3 Strategy **58**

4 Drivetrain **59**

4.1 V1 Design Cycle 59

4.2 V2 Design Cycle 69

4.3 V3 Design Cycle 75

4.4 Progression of Swerve Modules 81

4.5 V4 Design Cycle 82

4.5.1 Problem Definition 82

4.5.2 Solution 82

4.6 V4.1 Drivetrain Design Cycle 83

4.6.1 Problem Definition 83

4.6.2 Solution 84

5 V5 Drivetrain Design Cycle **84**

6 V6 Drivetrain Design Cycle **84**

6.0.1 V5 Cons and Pros 84

6.0.2 Prototype V6 85

6.1 V7 Drivetrain Design 89

7 Intake **92**

7.1 V1 Intake Design Cycle 92

7.2 V2 Intake Design Cycle 98

7.3 V3 Intake Design Cycle 100

7.3.1 Goals 100

7.3.2 Intake V3.1 Design Review 100

7.4 Intake V3.1 Assembly 108

7.4.1 Intake V3.2 Design Review 110

7.5 V1 Design Cycle 111
7.6 Potential options analysis 112
7.7 Testing and Iteration 115
7.8 Design Cycle II 116
7.9 V2 Design Cycle 117

8 Hood **118**

8.1 V1 Hood and Pooper Design Cycle 118
8.2 V1 Hood Design Cycle 121
8.3 Manufacture and Iteration 123
8.4 Analysis of Function 125
8.5 V2 Hood Design Cycle 127

9 Programming **133**

9.1 Reflection on the Previous Year 133
9.2 Identifying Challenges and Goals in Software Architecture 133
9.3 Researching Possible Architectures 134
9.4 Researching and Selecting a VEX Development Platform 136
9.5 Designing our Initial Software Architecture 139
9.6 ROS Experimentation 140
9.7 Phase Development Introduction 142
9.7.1 Phase 1 Overview 143
9.7.2 Phase 1 Development 145
9.7.3 Phase 2 Overview 156
9.7.4 Phase 2 Development 156
9.7.5 Phase 3 Overview 160
9.7.6 Phase 3 Development 161
9.8 CI 164

9.9 Code 168
9.9.1 NodeManager 168
9.9.2 MotorNode 171
9.9.3 ConveyorNode 174
9.9.4 HolonomicDriveNode 180
9.9.5 Odometry 185
9.9.6 FollowerOdometry 186
9.9.7 SwerveModule 188
9.9.8 Auton 194
9.9.9 AutonSequence 197
9.9.10 Path 198
9.9.11 PathPoint 200
9.9.12 V5 Serial Driver 202
9.9.13 V5 Smart Port Driver 205

List of Figures

1 Prototype peacock latch . . . 14
2 1x5x1 C-channel Tensioner . . . 17
3 preload hold-able rubber band . . . 18
4 Robot out in the snow to cool down overheating swerve motors . . . 19
5 First time a three wheel differential swerve robot has ever driven in VEX . . . 20
6 Tank drive-train . . . 21
7 Custom RS485 to USB PCB render . . . 22
8 Attached and replaced Tank Drive train . . . 23
9 Ball Blocker V1 . . . 25
10 Ball Blocker V1 preload . . . 25
11 dead zone intake roller . . . 26
12 dead zone intake roller holder . . . 27
13 centering & ball blocker V2 . . . 30
14 V2 design Gantt chart . . . 35
15 Continuous intake design . . . 39
16 Toss up robot . . . 40
17 Separated continuous intake design . . . 40
18 Motorized center goal descoring arm . . . 41
19 Linear extension descoring arm . . . 42
20 FRC #118 2016 gear pickup design . . . 42
21 Linkage arm for descoring center goal . . . 43
22 Dual tray robot design concept . . . 44
23 Motorized scissor lift design . . . 45
24 Pnematic alignment device . . . 45
25 Center goal descoring fins idea . . . 46

26 Ball capture fins from VRC 2587 in 2014 46
27 Rotary hopper idea 47
28 FRC # 33's 2020 robot with rotary hopper 47
29 Ball capture idea for early match 48
30 FRC #1114's 2015 can grabbers 49
31 Deployable wall-bot for center goal 49
32 Wall bot deployment during autonomous 50
33 2W's 2012 robot defending a center goal 51
34 Goal tender for ejecting balls from goals 51
35 side view of deploy, gearing on left and right 56
36 super structure and front view 56
37 Potential skills routine 58
38 Potential skills routine 58
39 A CAD model detailing a X-Drive frame made by Vex team 7405P 60
40 A CAD model created by Daniel for a potential tank drive design 60
41 CAD model of a mecanum drivetrain from the BLRS Wiki 61
42 An example differential swerve created by Trevor Glasheen on ChiefDelphi 62
43 Armabot's commercially available FRC differential swerve 63
44 Compact differential swerve idea 64
45 V1 of the differential swerve 65
46 The inside of the differential swerve gearbox 65
47 The CAD of the second iteration of V1 swerve 66
48 Initial printing of herringbone planet gears on a Prusa Mini 67
49 2nd layer of ball bearings and herringbone planet gears 67
50 The fully assembled V1 differential swerve module 68
51 A demonstration of the tight space requirements for a V1 differential drivetrain . . . 69
52 V1 bevel gear set 70

53 V2 bevel gears with undercutting . . . 71
54 V3 bevel gears with tapering . . . 72
55 V4 bevel gear . . . 72
56 CAD of the initial design of V2 swerve . . . 73
57 Testing V2 of the differential swerve . . . 74
58 Top view of V2 differential swerve inferences with ball . . . 75
59 A variant of JVN's mechanical design calculator adapted for VEX V5 and differential swerve . . . 76
60 V3 differential swerve CAD . . . 77
61 Parts for nearly 2x V3 modules . . . 77
62 V3 differential swerve assembly process . . . 78
63 3D printed wheel with neoprene tread . . . 78
64 V3 differential swerve module fully assembled . . . 79
65 Single V3 swerve module attached to chassis for testing . . . 80
66 CAD of v3 modules integrated into drivetrain . . . 80
67 V3 swerve modules installed on frame . . . 81
68 V1, V2, and V3 differential swerve size comparison . . . 81
69 Concept drawing of 3-layer ball bearing design . . . 82
70 Ball bearings in V4 module CAD . . . 83
71 V6 prototype drawing option 1 . . . 85
72 V6 prototype drawing option 2 . . . 86
73 V6 prototype drawing option 3 . . . 87
74 V6 prototype drawing option 4 . . . 87
75 V6 prototype gear ratio . . . 88
76 V2 Holonomic Drive CAD . . . 89
77 V2 Holonomic Drive CAD Close Up . . . 90
78 V2 Holonomic Drive In Real Life . . . 90

79 Two motor X-drive module 91

80 An example of a horizontal beater bar intake from Brampton Robotics 92

81 An example of a conveyor belt intake on team 99999V's CAD challenge robot . . . 93

82 An example of an omni wheeled intake employed by VEX team 2602H 94

83 A top down sketch with the intake rollers drawn in their in and out positions 95

84 A diagram of the initial intake design 96

85 Completed CAD of the V1 right arm of the intake 96

86 V1 intake testing 97

87 V1 intake testing different intake angles 98

88 V2.1 Intake 99

89 V3.1 Intake 100

90 V3 Intakes retracted linearly to be within frame 108

91 V3 Intakes in their extended position after they are passively deployed via rubber band tension 109

92 The V3 intakes utilize a double-acting pneumatic on each side to spread the front wheels apart, significantly increasing the intake range of the robot. 109

93 V3.2 Intake 110

94 VEXU team SJTU utilizes a deployable hood ejector ramp at the top of their robot to eject balls 111

95 A pooper system utilized by VEX team 27905B 112

96 A diagram of a trapdoor ejection system 112

97 Geometry sketch showing the path of the ball through the rollers with the possibility of going up to the hood mechanism or out the back of the robot 114

98 The finished design of the pooper roller system 114

99 Testing the first prototype of the lower rollers in real life 115

100 The updated CAD of the lower rollers 116

101 Assembled Lower Rollers 117

102 Retracted Peacock Diagram 118

103 Expanded Peacock Diagram . . . 119

104 SJTU example of a two part hood mechanism . . . 120

105 Extended One Part Hood Diagram . . . 120

106 Retracted One Part Hood Diagram . . . 121

107 Peacock Geometry Sketch . . . 122

108 V1 Peacock Assembly Extended . . . 122

109 V1 Peacock Assembly Retracted . . . 123

110 Assembled Version of the V1 Peacock . . . 123

111 Improved V1 Hood with Increased Gear Ratio and less trusses . . . 124

112 Latch added to lock the peacock mechanism in its deployed state after the start of the match and to ensure more consistent compression . . . 125

113 Retracted Peacock . . . 125

114 The 7 ball capacity . . . 126

115 Three Hood Roller Geometry . . . 128

116 Four Hood Roller Geometry . . . 128

117 Final Hood Roller Geometry . . . 129

118 Hood V2 CAD . . . 129

119 The V2 Hood in it's extended position with the roller and backing deployed with rubber band tension . . . 130

120 The V2 Hood in it's collapsed position with the backing folded down and over the deployed top roller holding it down. The hood is released by outtaking the ball from the lowest roller. . . . 131

121 Ramps make a smooth transition from the ring of the goal or ground to the robot that conform to the profile of the ball. The use of two rails keeps the ball centered throughout its travel through the robot until it is scored. The use of drawer liner material prevents the ball from spinning in place, thereby increasing speed of the mechanism . . . 131

122 Two line sensors were added between the rails to detect when a ball is in front of it to automatically store the balls. We found this to be a necessary addition due to the ball's rapid travel through the robot from intake to being launched out of the top of the robot and to be able to retain both balls required in skills runs. . . . 132

123 Initial NodeManager Structure 144
124 Trello Node Assignments 145
125 ROS Node Communications 146
126 Robot Topics Registered with ROS Master Node 151
127 Readouts on the MotorNode Topic 151
128 Initial Swerve Math Discussions 156
129 GitHub Backlog from Phase 3 162
130 WebDashboard Application 162
131 Example YAML File for Python 165

1 Meetings

1.1 4/25/2020

We discussed the new game, possible strategies, and listened in on strategy from other teams and individuals. Being so early in the season, we wanted to focus on gathering as much information as possible and working on producing as many variant designs as possible. This way, we can evaluate many different concepts of playing the game from a diverse group of sources.

1.2 1/27/2021

Goals:

1. Test RS-485 smart port communication as a replacement for the buggy USB connection between the raspberry pi and V5 cortex
2. Finish up the V4 differential swerve design, or at least finalize a few parts so we can get things printing
3. Add more "feathers" to the peacock so we have consistent compression on the balls as they leave the top of the mechanism

Results:

- We got the RS-485 communication working somewhat, still have tons of ROS errors popping up that we do not understand but we are successfully publishing topics
- We attempted to test the full swerve code with the new RS-485 communications, but we ran into some issues and weren't able to get any motors spinning. Will work on next meeting
- Did not get all the feathers on the peacock as some members had to leave early to finish homework. Will work on this next meeting

1.3 1/28/2021

Goals:

1. Figure out the ROS errors being thrown in the RS-485 communication line
2. Apply the RS-485 communications to the previous ROS architecture and test system latency
3. Continue adding "feathers" to the peacock to fix the top end of the mechanism
4. Look into locking the top tower in place after expansion

Results:

- We spent a majority of the time digging into the RS-485 communications, referencing documentation from other teams and the drivers we wrote for serial USB communication. We were able to read the incoming byte stream and get somewhat predictable data, but ROS running on our Raspberry Pi wasn't able to comprehend it. Changing the drivers to read/write through the USB port made everything work, isolating the problem down to just the drivers we wrote for communicating over the smart ports with RS-485
- We identified a potential memory leak in the smart port drivers we wrote, causing sporadic behavior on the brain and crashing it after a few minutes of port IO activity
- We made a simple two-way locking mechanism on the peacock to prevent the tower from moving much when it expands. Right now we need to look at 3D printing the lock to make it more robust

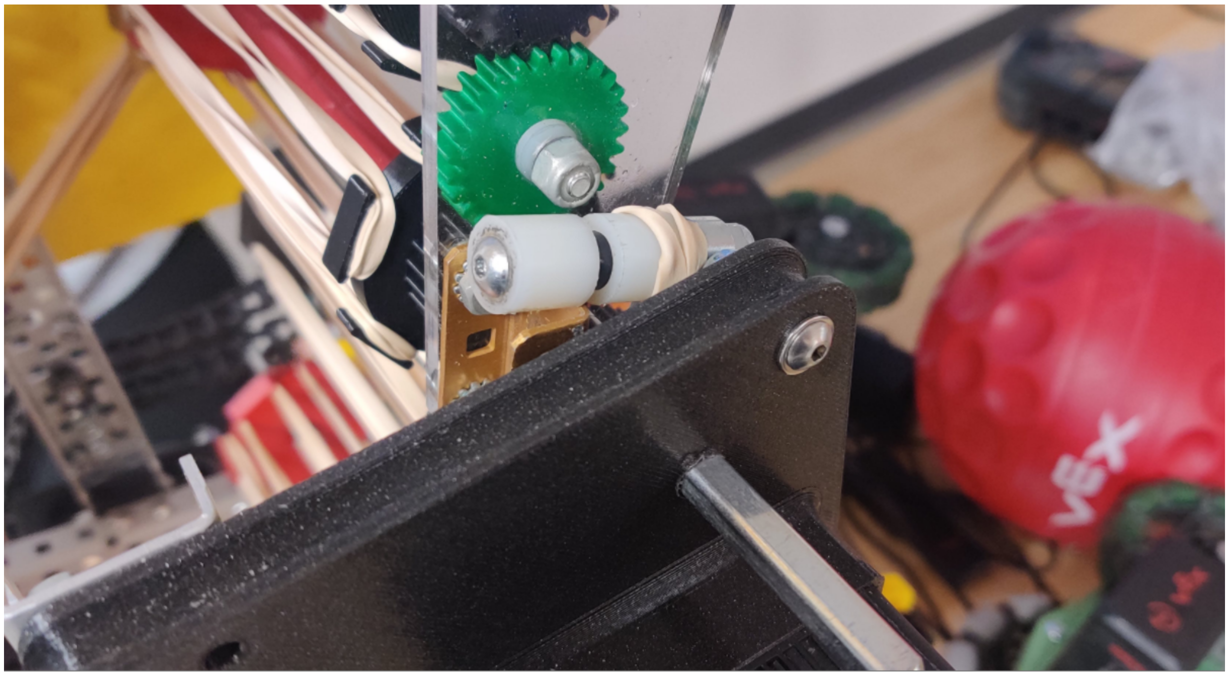


Figure 1: Prototype peacock latch

- We didn’t get to fixing the "feathers" on the peacock, but looked at better ways for adding a backing to the hood for the future
- We started porting our dependencies over to PROS, allowing us to run a minimal setup of our current robot completely off of the V5 brain without depending on a co-processor. We got mostly through the Eigen library transfer, and will look to test it in the near future once the Swerve code is transferred

1.4 1/29/2021

Goals:

1. Test rosserial on an Arduino to determine if the errors we are seeing are V5 cortex specific or limitations with rosserial
2. Continue working on implementing the swerve controller directly on the V5 cortex as a backup plan
3. Start assembly of the new V4 swerve modules
4. Finish up peacock feather modifications
5. Make a 3d printed latch for peacock
6. Work on peacock top rollers

Results:

- We tested the new RS-485 fixes and they seem to work well for publishing data from the V5 cortex at high speeds (50Hz) and are relatively stable over, only encountering errors every few minutes or so
- With the new fixes, the full robot code still does not appear to be working with the RS-485 serial communication cutting out after about 10 seconds. We will need to do more troubleshooting and possibly look into our backup plan
- We got almost all the way through the backup plan implementation of the swerve controller on the cortex. Still some work to be done
- Inner Module of differential assembled, although bottom bearing run needs more support.
- 3D designed new peacock latch has been attached

1.5 1/30/2021

Goals:

1. Finish up implementing the swerve controller directly on the V5 cortex as a backup plan
2. Finish assembly of one of the new V4 swerve modules
3. Mount electronics on the robot and wire them up
4. Tune PID loops for swerve drive
5. Get the robot moving (finally!)
6. Troubleshoot issues with RS-485 seen at end of last meeting

Results:

- Attached pivot materials (1x5x1 C-channel) to help pushing ball up to the peacock roller which

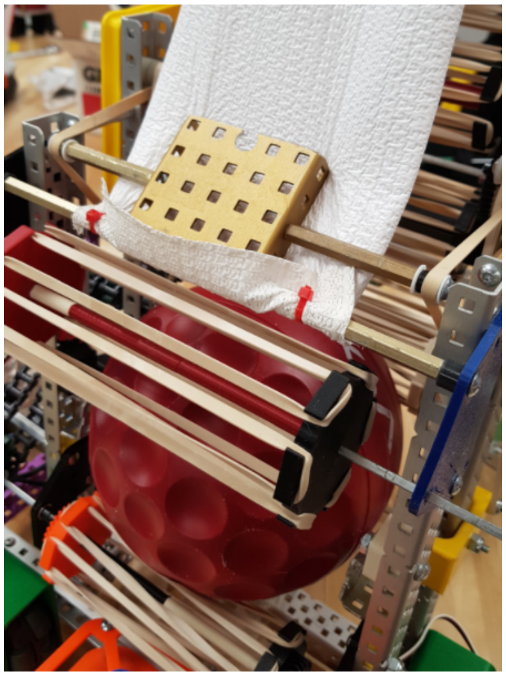


Figure 2: 1x5x1 C-channel Tensioner

- gives enough tension on supporting elevating rubber plate
- Re-adjust the peacock head to reduce the compression between ball to roller

- Assembled preload supportive material for intake, so when intake start to rotate, it possibly deploy
- Attached rubber band on the outside of peacock head which help to preload (fold in) and possibly deploy when back roller move.

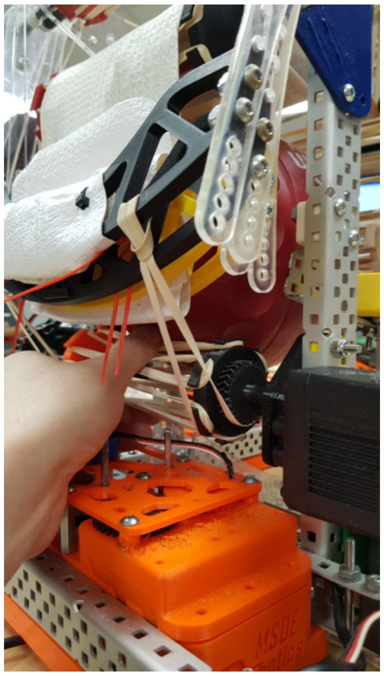


Figure 3: preload hold-able rubber band

- Tested entire intake mechanism and significantly improved
- Tested entire preload setting and fit in 15inches limit

1.6 1/31/2021

Goals:

1. Test all 3 swerve modules on robot
2. Test field oriented drive
3. Make our first official driver skills run
4. Build 3rd V4 differential swerve module
5. Work on troubleshooting rosserial communications over USB

Results:

- First driver skills run was recorded, we scored 56 points!
- Decided to work on improving intake and reduce friction of Drive-train
- Found some major problems with swerve. Had to switch to lower gear ratios before run due to overheating issues. We are looking into switching to a tank drive temporarily while we troubleshoot issues with the modules.
- Discovered putting the robot outside in the snow is a great way to cool it down

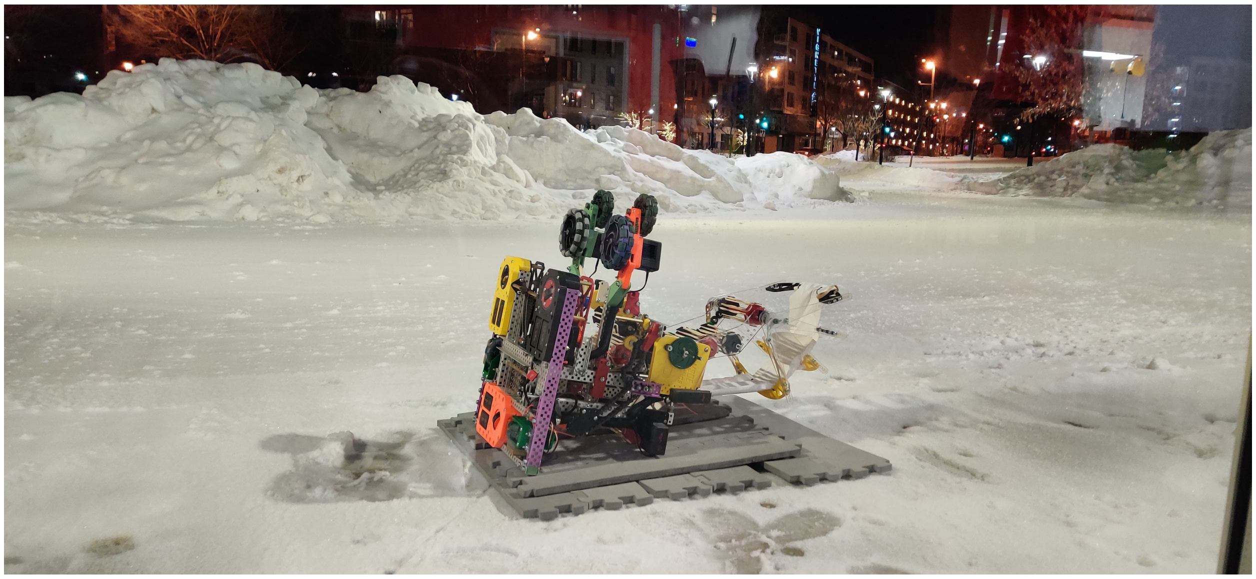


Figure 4: Robot out in the snow to cool down overheating swerve motors

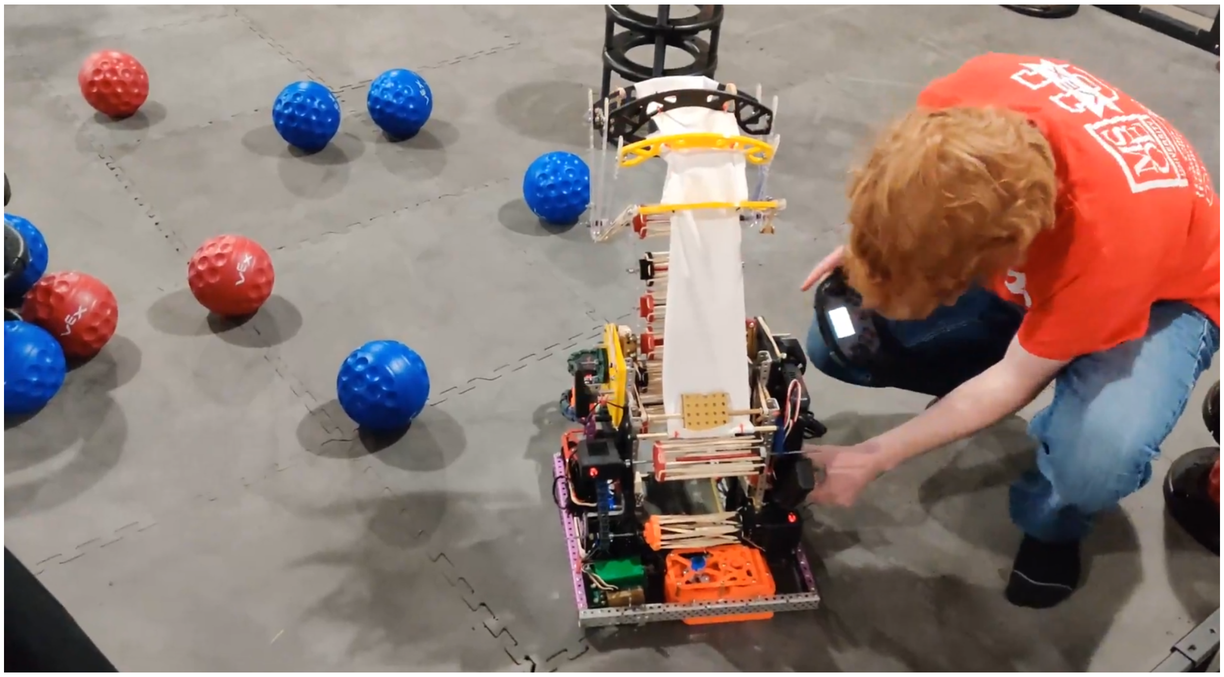


Figure 5: First time a three wheel differential swerve robot has ever driven in VEX

1.7 2/03/2021

Goals:

1. Create another simple tank drive train
2. Assemble a third V4 swerve module

Results:

- Done with assembling tank drive train individually

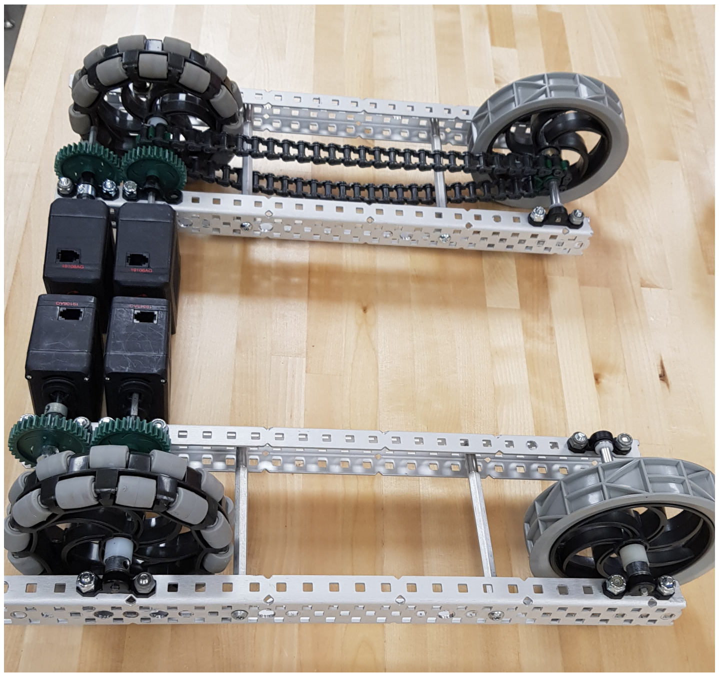


Figure 6: Tank drive-train

1.8 2/05/2021

Goals:

1. Finish converting robot to tank
2. Tune swerve PID's
3. Fixing design of V4 swerve modules
4. Assemble another V4 module
5. Finish custom RS485 PCB
6. Add classes to handle Odometry on the robot
7. Introduce some of the freshman to debugging on the V5 system
8. Fix problems with the CI workflow
9. replace to tank drive train

Results:

- PCB is complete!

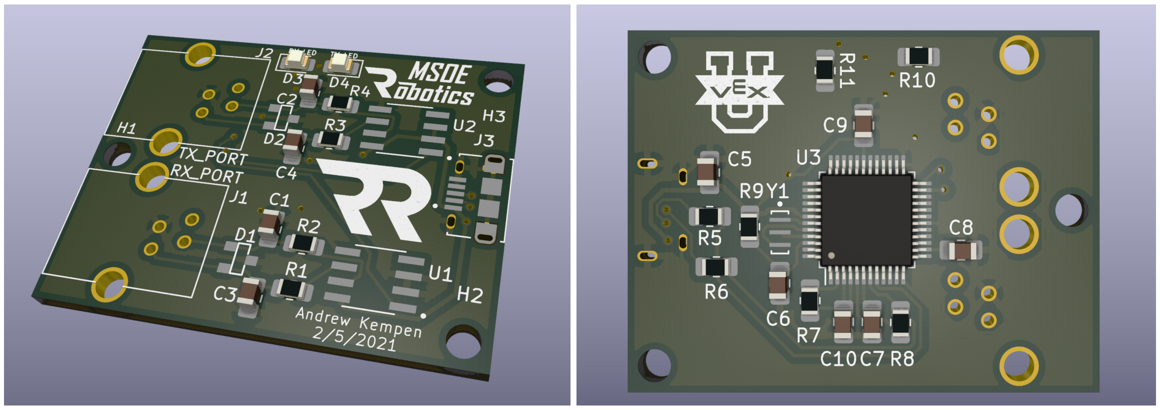


Figure 7: Custom RS485 to USB PCB render

- Completed classes for tank drive odometry (based on Andrew's previous code) and for follower wheel odometry
- Taught Dylan how to work with the V5 system, and set up rqt plot to show PID values
- Fixed a needed installation for Eigen3 in the CI workflow and got it working correctly

- Replaced to tank drive train

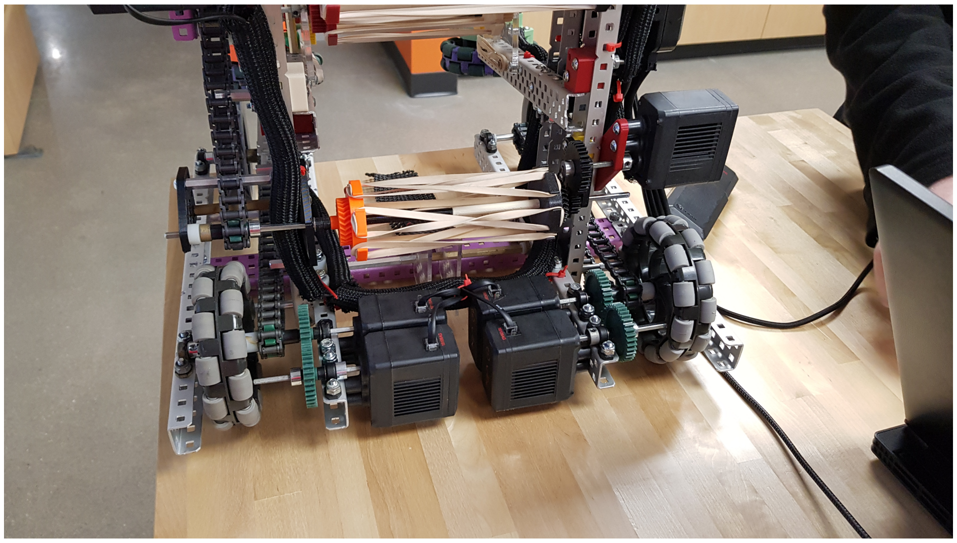


Figure 8: Attached and replaced Tank Drive train

1.9 2/06/2021

Goals:

1. Practice driving new tank drive robot in driver skills
2. Figure out how to load autonomous sequences on to the robot from our JSON files created by the web dashboard
3. Change drivetrain gears out with high strength gears
4. Keep debugging friction on new V4 differential swerves
5. Work on reducing dead zone between front intake wheels and lower top rollers
6. Debug previous problems with serial communications with ROS on the Raspberry Pi
7. Get ROS running automatically on startup for the Raspberry Pi
8. design and attach prototype of ball blocker V1
9. design new dead zone roller for intake
10. change normal rubber band to black rubber band on intake

Results:

- Created a cron job to run the ROS code on startup
- Fixed ROS serial code to read data from the NavX
- Added an indicator to the V5 screen based on the ROS connection status
- Got some practice driving with the new tank drive; from this we found and fixed a few issues (such as an intake dead zone)
- Decided that we would spend time tomorrow working on a simple autonomous mode, not using JSON files and path finding (to get something submitted for programming this week)
- Decided to focus more on perfecting the tank drive over the next two weeks instead of attempting to perfect the V4 swerve module. Since we still have programming and tuning to do for swerve, we thought it would be best to focus on improving our score for our current competition instead of trying to get a run in with the new swerve

- Attached ball blocker V1

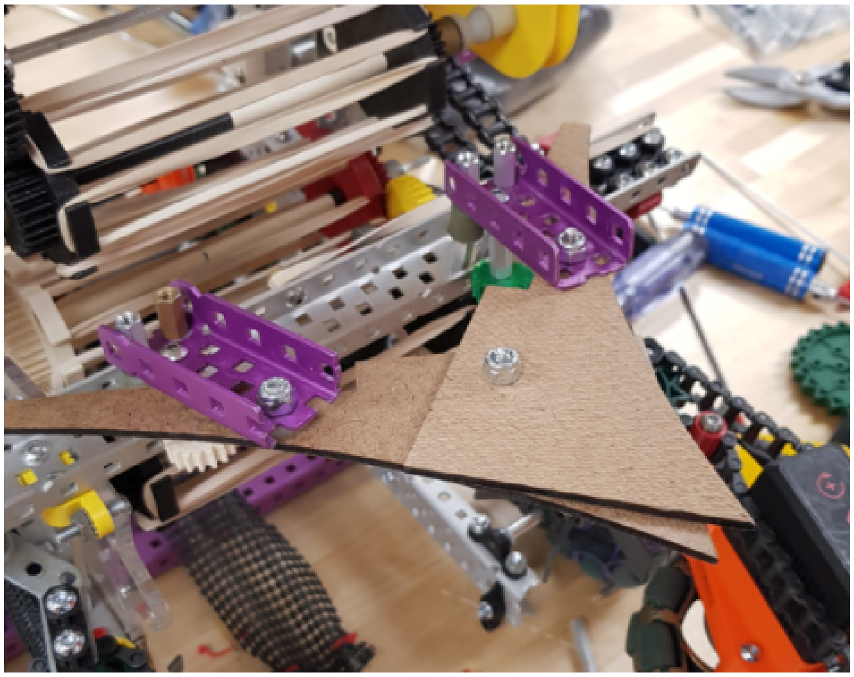


Figure 9: Ball Blocker V1

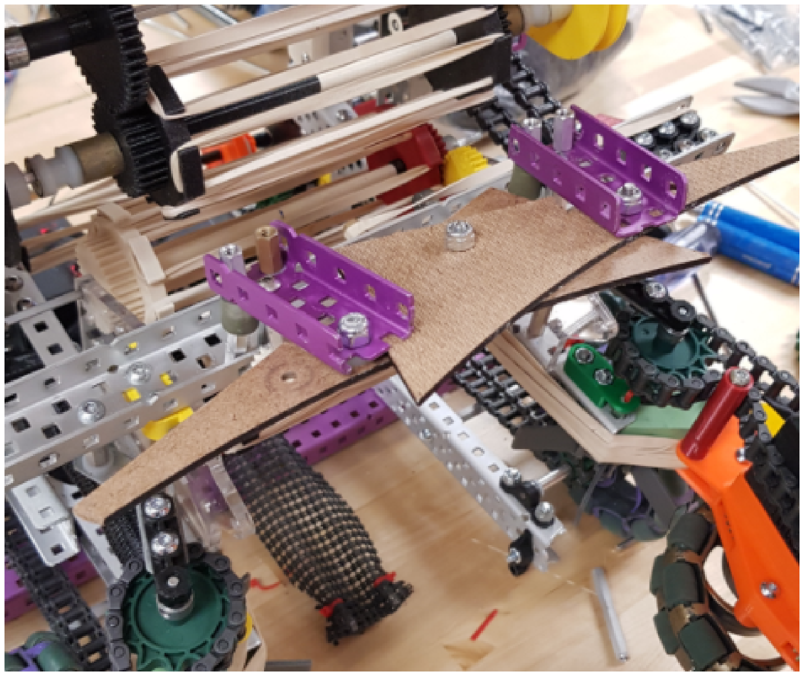


Figure 10: Ball Blocker V1 preload

- Attached new dead zone roller

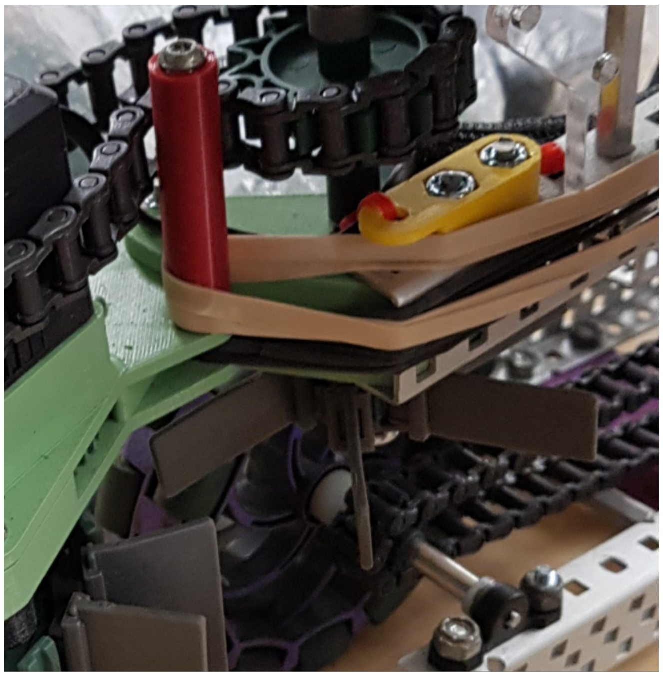


Figure 11: dead zone intake roller

- Developed dead intake roller; chain was on the way of goal shaft so printed new spacers which possibly deploy from preload also give smooth line that gives less friction of roller
- changed to black rubber band on intake

1.10 2/07/2021

Goals:

1. Practice driving for skills so we can submit pre-recorded runs this week
2. Build a simple autonomous using basic driving and peripheral commands
3. Submit runs for W3 of our pre-recorded competition
4. using 3D printer to print and attach drive-train motor shaft holder so gear doesn't slip
5. record video of skill challenge
6. develop the preload motion which fit in 15inches
7. Fix the ball rolling not sucking up in the center of robot
8. attach dead zone roller shaft holder

Results:

- Attached dead zone roller shaft holder
- Attached drive-train motor shaft holder
- Attached dead zone roller shaft holder
- Fixed the ball rolling not sucking up in the center of robot

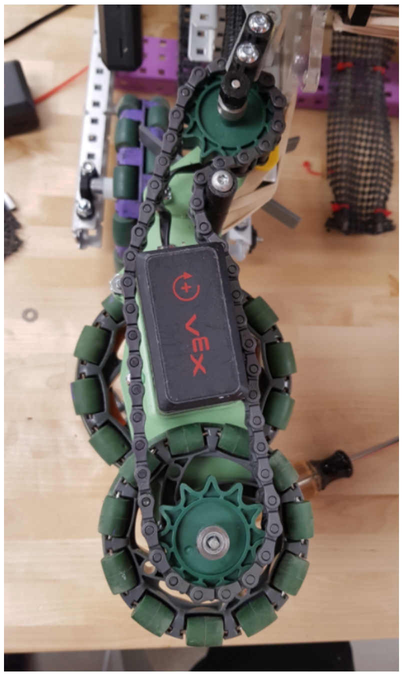


Figure 12: dead zone intake roller holder

- well developed preload motion and recorded skill challenge

1.11 2/10/2021

Goals:

1. Work on getting tank path following code working
2. CAD new lower roller ramp so it is closer to the ground and leaves space for the new follower omni wheel location
3. Attach new follower omni wheels to chassis
4. design prototype of goal centering support with ball blocker V.2

Results:

- Fixed problems in tank path following code in simulation, wasn't able to try it on the robot yet
- Worked on new follower omni-wheel locations
- Worked on updating CAD for previous robot changes, and new ramp design
- Worked on creating actions within the code, for use with action scheduling to create autonomous routines
- Added velocity control in TankDriveNode and MotorNode for use in FollowPathAction
- Broke out node structure to include periodic loops for teleop and autonomous
- finish designing goal centering support

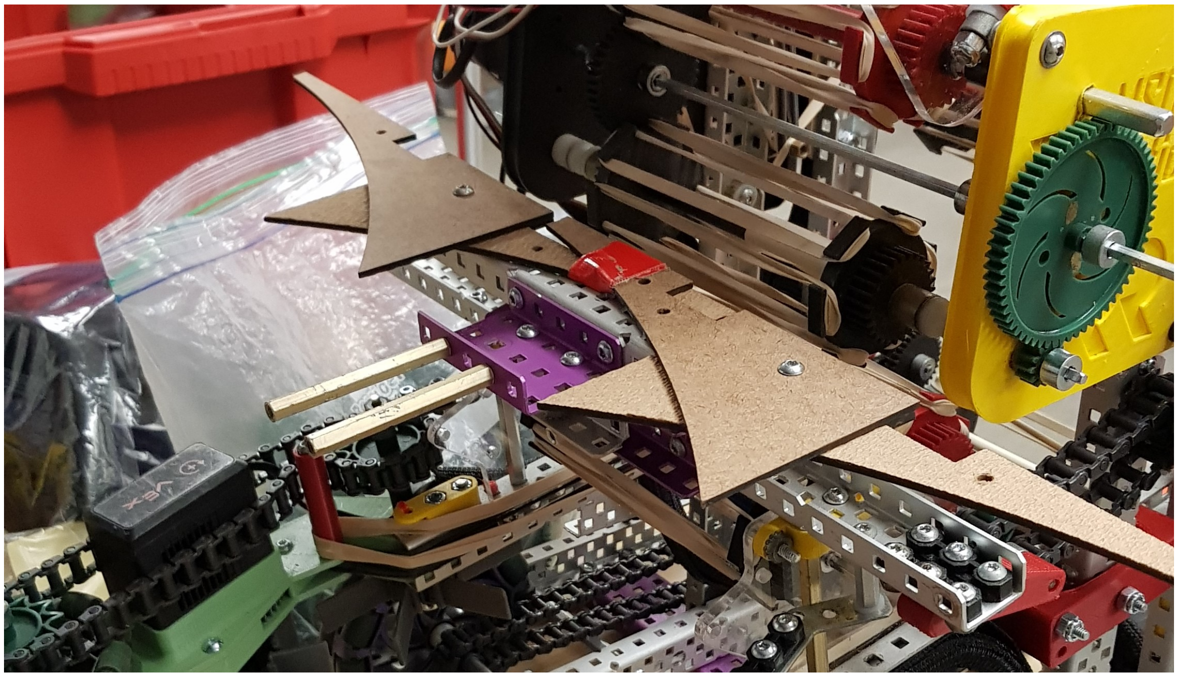


Figure 13: centering & ball blocker V2

1.12 2/11/2021

Goals:

1. Finish adapting FollowPathAction to current classes and methods
2. Add action to turn robot to an angle
3. Add follower omni wheels to the robot for odometry
4. Added autonomous state machine for the upper conveyor
5. Add new lower ramp in robot

Results:

- Finished adapting FollowPathAction
- Added TurnToAngleAction to turn the robot to an angle using a proportional value
- Added follower omni wheels to the robot
- Added new lower ramp to robot

1.13 2/12/2021

Goals:

1. Test path following on the robot
2. Begin updating CAD to match V1 Robot
3. Resolve any remaining misc. mechanical issues on the robot

Results:

- Resolved a few mechanical issues, including friction in the drive train due to an incorrect spacer
- We ran into issues with path follower crashing on the robot, so we spent a lot of time trying to troubleshoot the issue
- Updated some of the UI on path follower to match skills, and began working on creating our desired path
- Updating the code to run path following took longer than expected, so we will continue working on that next meeting

1.14 2/14/2021

Goals:

1. Test path following on the robot
2. Keep updating CAD to match V1 Robot
3. Check robot deploy
4. Write programming skills autonomous
5. Record programming and driver skills

Results:

- We spent all morning trying to fix the pathfinder crashes we saw Saturday. We eventually gave up trying to fix the issue as we needed to get something working. Current theory is that the robot gets too far from the path which leads to the adaptive pursuit controller crashing.
- Instead of the path following based system for autonomous, we reverted to simple rotate and move distance commands for moving the robot around. While not ideal, we at least can get some points up on the board for programming skills.
- While recording our runs, we ran into several issues. First off, the recording stopped halfway through our first try. The second time, the recording got messed up, but we did not discover the issue until after the run had been recorded. Still usable, but video shakes randomly.
- Programming skills almost worked, but our robot battery died halfway through our second goal

1.15 2/17/2021

Goals:

1. Discuss where V1 succeeded and where it can be improved
2. Start brainstorming solutions to problems
3. Start laying out timeline for V2 robot

Results:

- We wrote up a list of all of the problems we had with the V1 robot
- We will be taking a break next week for final exams, so no meeting next week

1.16 3/3/2021

Goals:

1. Review pros/cons of V1 robot
2. Start talking about V2 robot changes, ideas
3. Layout schedule for V2 robot design

Results:

- We came up with lists of changes to make on each of the mechanisms for the V2 robot design
- Developed Gantt chart for V2 robot design

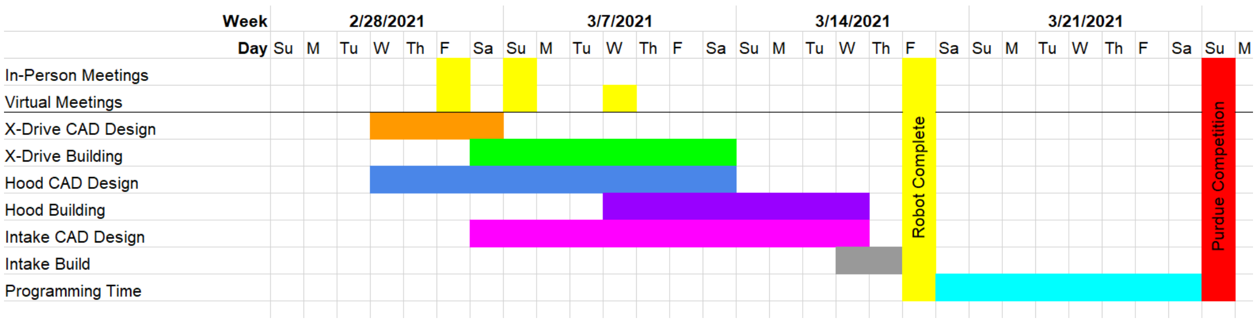


Figure 14: V2 design Gantt chart

1.17 3/13/2021

Goals:

1. Start building new X-drive modules
2. Get everyone on same page with intake
3. Start working on path segment object in program
4. Test custom PCB for RS-485 to USB
5. Feedback on the corner alignment mech

Results:

- We started working on the new X-drive modules, but were not able to complete them
- Intake design review resulted in several changes that will be addressed
- RS-485 to USB PCB is not working, will need further rework and analysis

1.18 3/18/2021

Goals:

1. Finish building new X-drive modules
2. Come up with Gantt chart for software
3. Finalize and start printing new intake parts

Results:

- We were able to finalize the intake and have started printing many of the parts for the intake

1.19 3/24/2021

Goals:

1. Finalize the holonomic drive, intakes, and conveyors
2. Remove inner c-channel from drivetrain
3. Start testing code on all components of the robot

Results:

- The drivetrain worked, but overheats easily. We need to look at adding additional motors to each of the pods
- The basic code for the holonomic drivetrain and kinematics are working well
- Need to revise the conveyor state machine slightly to help it catch the ball earlier

2 Overall Robot Design

2.1 Early Season Brainstorming

The purpose of this section was to record and outline as many of our ideas as possible, not pertaining to any specific design cycles or strategies. At the beginning of the year, we encourage people to come up with as many ideas as possible, pertaining to nothing specific on the robot, so we can create a collection of “high level” ideas that can be referenced in future design cycles. This repository of ideas doesn’t constitute design cycles themselves but can be invaluable when considering ideas for future designs or helping us pick good solutions to current problems that we are facing. The ideas in this section are all from early season and may be referenced in the future.

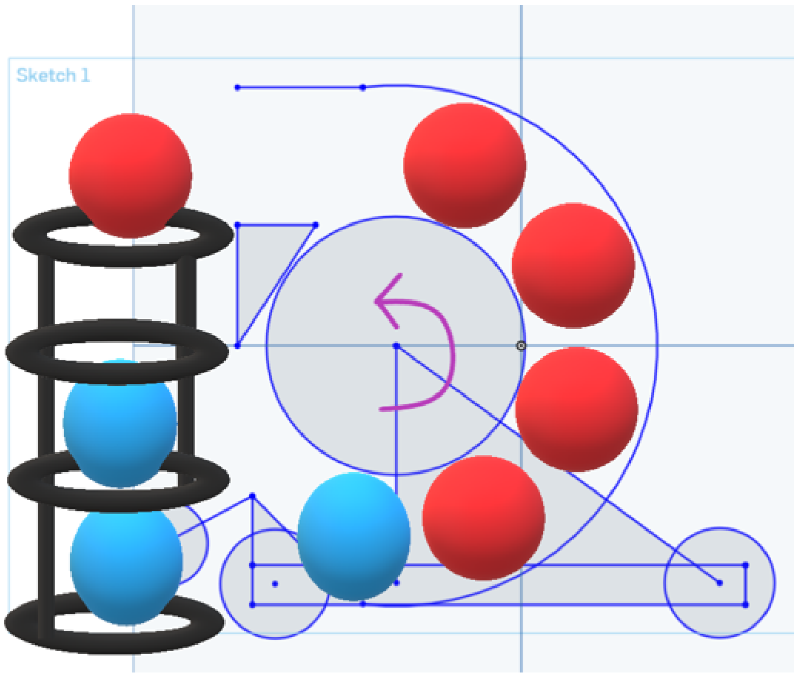


Figure 15: Continuous intake design

Figure 15 was one of the initial designs we thought of when first watching the reveal. The purpose of it is to quickly cycle balls out of the base of the tower with a roller intake while simultaneously adding balls of our alliance’s color. After it’s done cycling the balls at a goal, it can either get rid of the opposing alliances balls in its system onto the field through either the intake or outtake, or it can strategically use them as filler for other goals.

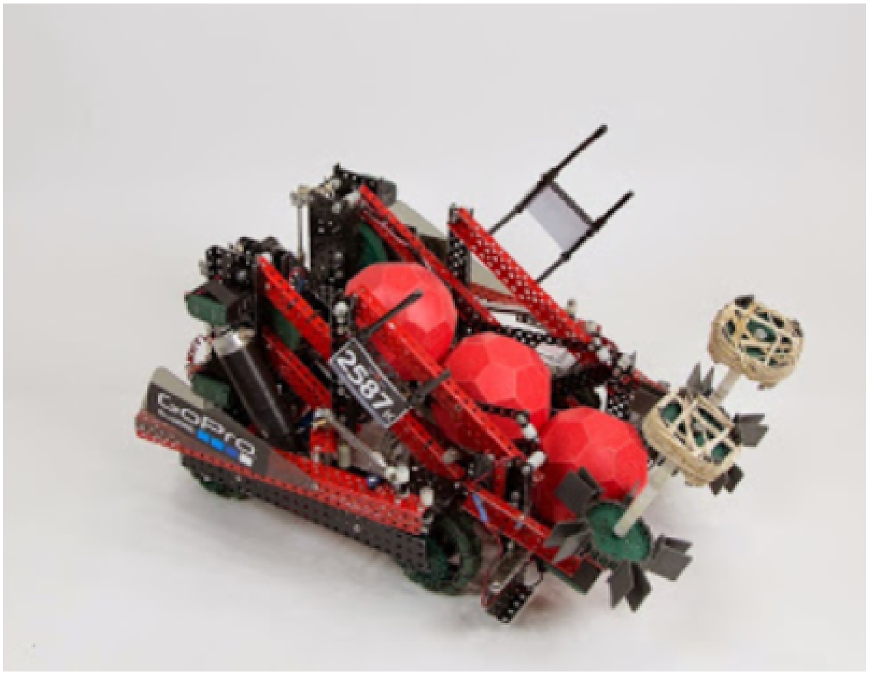


Figure 16: Toss up robot

Figure 16 shows a robot (Team 2587k) from 2014's VRC game, Toss Up. This was another design style we determined might be efficient for playing Change Up when first watching the reveal. However, some modifications would need to be made to it to better fit the playstyle of this year's game, such as complying with larger elements.

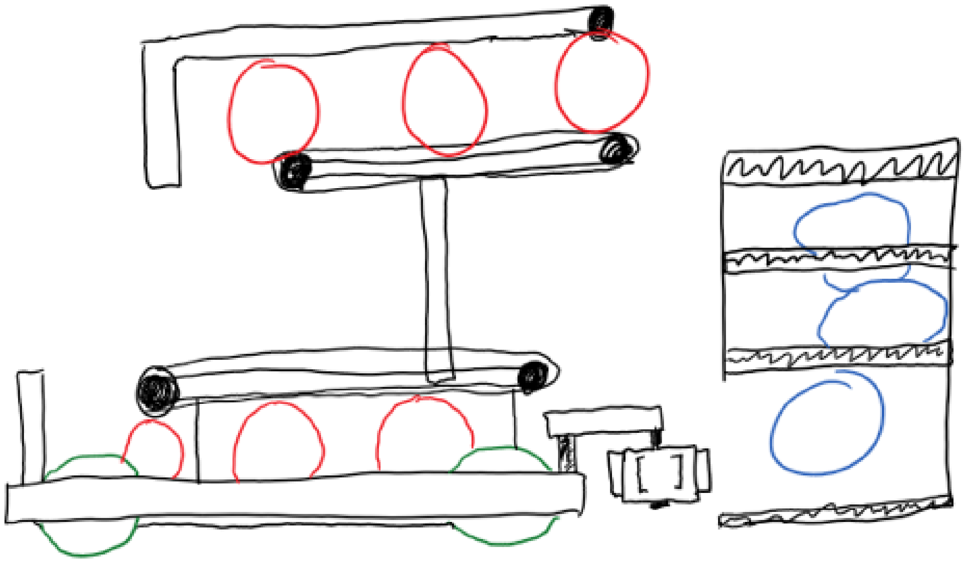


Figure 17: Separated continuous intake design

Figure 17 illustrates a similar design to the continuous intake system in Figure 1.1, but with a slight modification to adapt to the 15-inch robot chassis. Since the smaller 15-inch chassis cannot reach the top of the 18-inch goals, it will require some type of lifting mechanism to get to the top. In this idea, the continuous intake would be split into a top and bottom section, where the top section is able to be raised up above the goal height (using either a chain-bar, four bar, or similar mechanism). This idea allows balls to also be taken into the robot while the lift is in scoring position, allowing us to quickly change the controlling (top) color ball in the goal, even when the goal is full. A potential concern of this design is it's first-in first-out nature, where a ball needs to travel through the entire robot before being in a potential scoring position. This extra time makes scoring cycles longer, which could make the robot less effective in general.

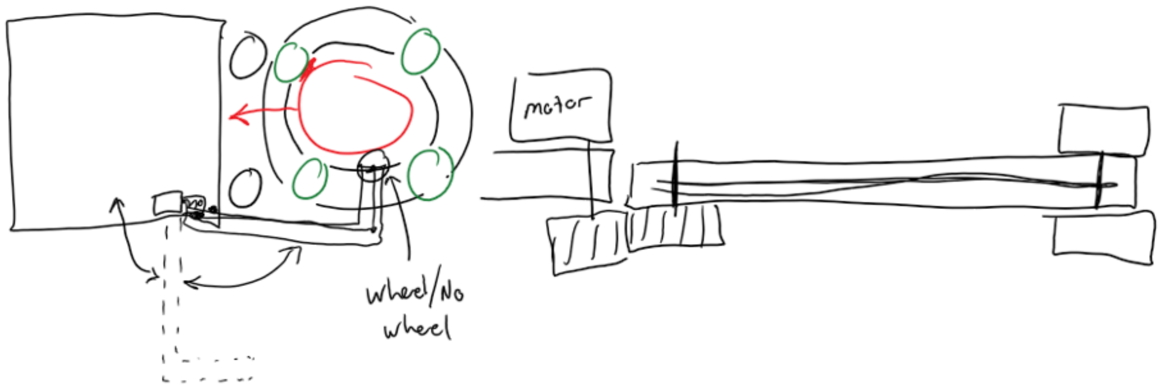


Figure 18: Motorized center goal descoring arm

Figure 18 addresses a problem we discussed with de-scoring goals on the field. While all goals on the outside of the field have two of the vertical posts removed to allow greater accessibility to the balls in the goal, the center goal does not. This means that robots wishing to de-score balls from this goal and obtain possession of the ball when it is removed from the goal (using intakes to remove the ball) need to have a specialized mechanism to do so. Since the clearance between the posts and the ball itself is so low, we wanted to think of a way in which we could pull the ball out of the goal.

This design uses a single motor, which controls the deployment of the arm, as well as rotation of the outer wheel. Allowing the motor to serve as the powering force behind both actuations would be convenient but lower the effectiveness of the component in general. In order to get the rotational motion out of the arm, more friction can be put in the rollers than in the initial gear drive, allowing the arm to pivot first before stopping at a mechanical stop and forcing the wheel to rotate.

We thought this idea could certainly help us get the ball out of the goal but were also concerned about the pivot radius it will have. Being a single arm, it will not be a very robust mechanism, and likely will need to pivot a decent distance from the frame of the robot. This poses a large risk to the arm being damaged, and other teams interfering with us.

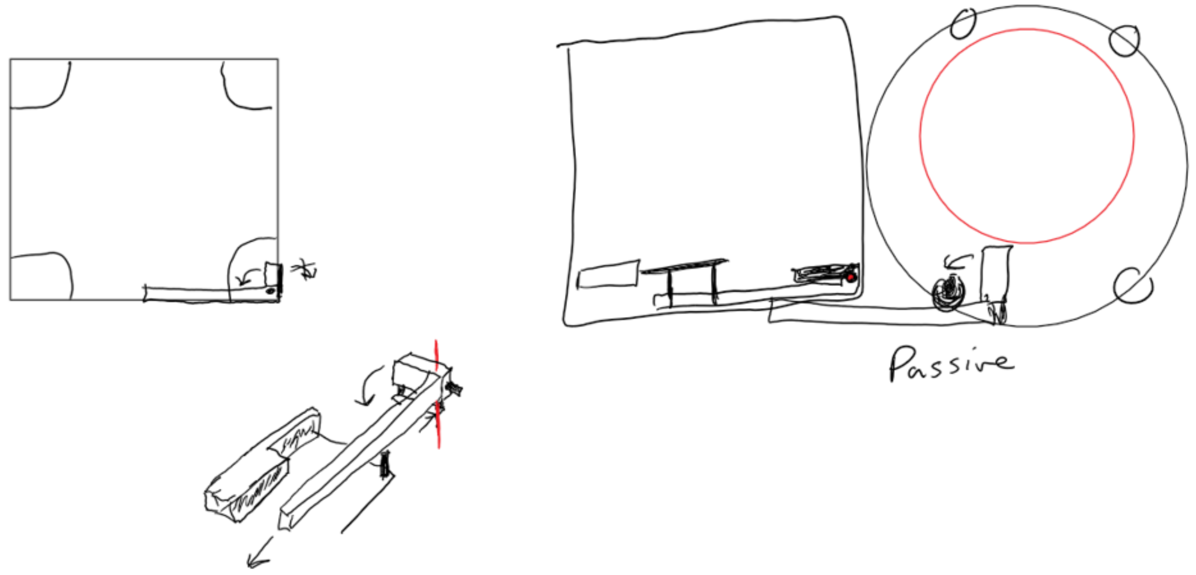


Figure 19: Linear extension descoring arm

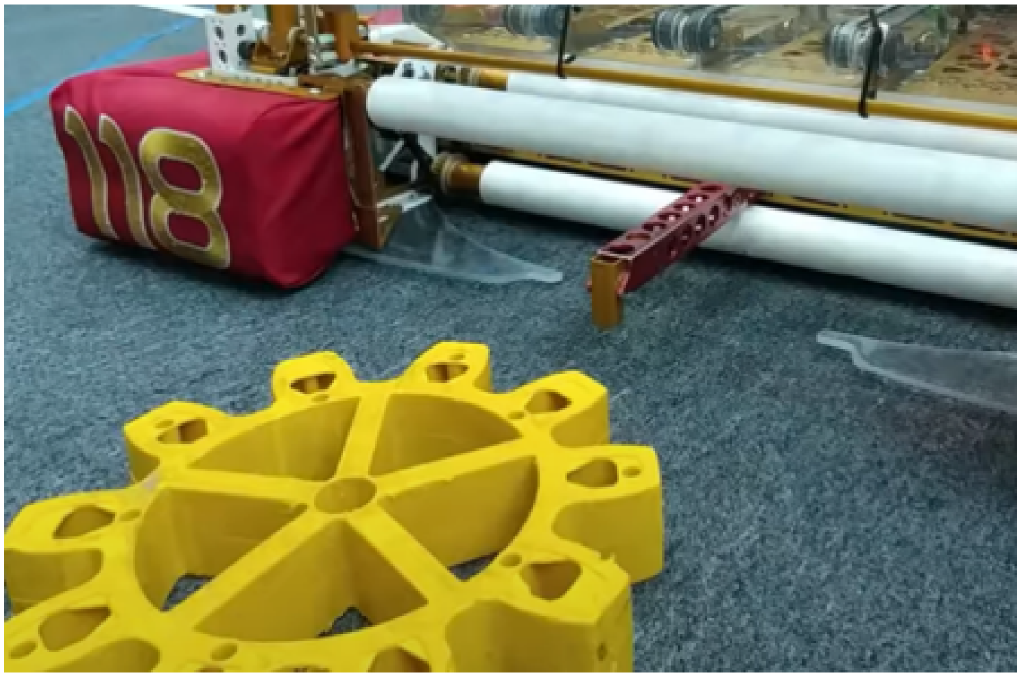


Figure 20: FRC #118 2016 gear pickup design

To mitigate the risks of the pivot radius of the arm, two alternate ideas to this arm were generated. The first of which is using a passive hook to get past the goal. Using a hook on the end of an arm, we can allow the hook to fold inwards (allowing it to slip past the pole in the goal, see Figure 19). This passive hook would extend linearly forward, minimizing the pivot radius of the arm. This does have a downfall of not being able to have an easily motorized end, however.

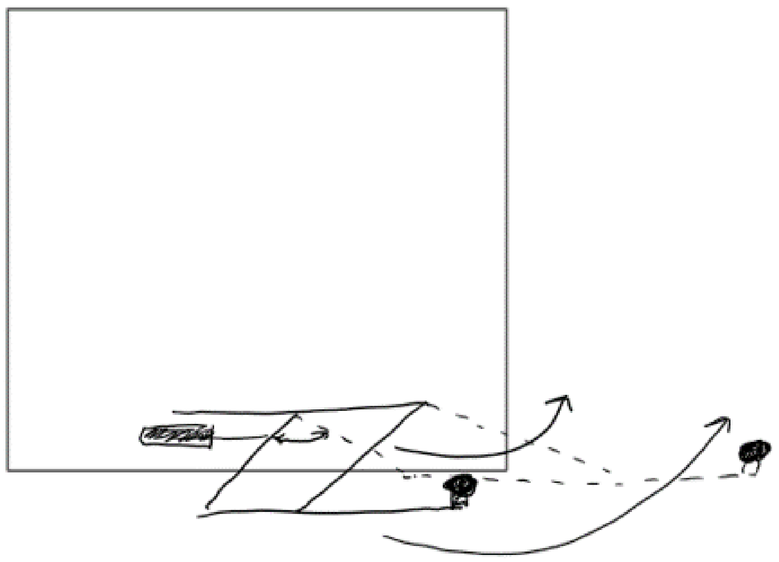


Figure 21: Linkage arm for descoring center goal

To allow for a wheel on the end of the arm, we designed a similar concept as shown in Figure 21 that uses a four-bar mechanism to extend into the goal and pull balls out, using a motorized roller on the end to help drive the balls out. This has the benefit of easily being able to be powered by a pneumatic cylinder (as the four-bar arm provides leverage to extend a farther distance), while allowing for a stable endpoint to be used to mount a wheel.

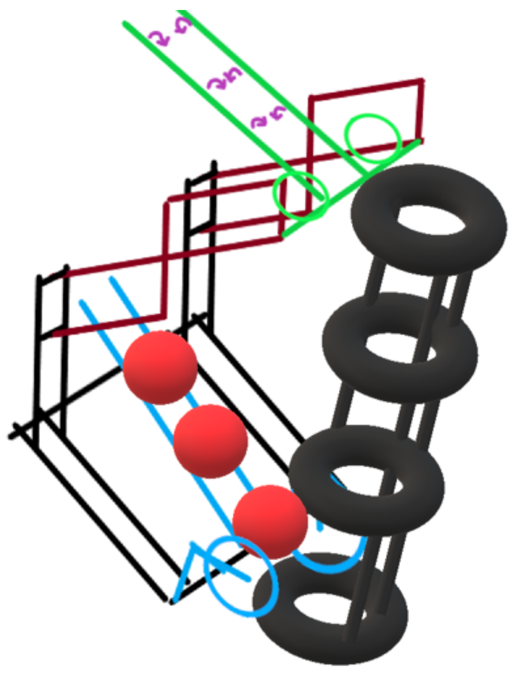


Figure 22: Dual tray robot design concept

As shown in Figure 22, some of our early ideas were very similar to the Toss Up designs from 2014. In an addition to these ideas, we wanted to create a design that allowed for simultaneous de-scoring and scoring. Since balls need to be removed from the goal to often free up room for others, we wanted a design that could do both in a single action. This lead to a dual tray-bot design, where our top tray would raise up on a six bar mechanism to lift balls into the goal, and the bottom tray and rollers would pull the current balls out of the goal. When the top tray lowers down, a passive plate on the bottom will fold out to allow the balls to pass through and into the top tray, allowing it to easily “reload” after scoring. This seemed efficient for cycling, however it encountered problems if the top tray wasn’t completely empty, as the balls in the top tray would then be pushed out of the robot.

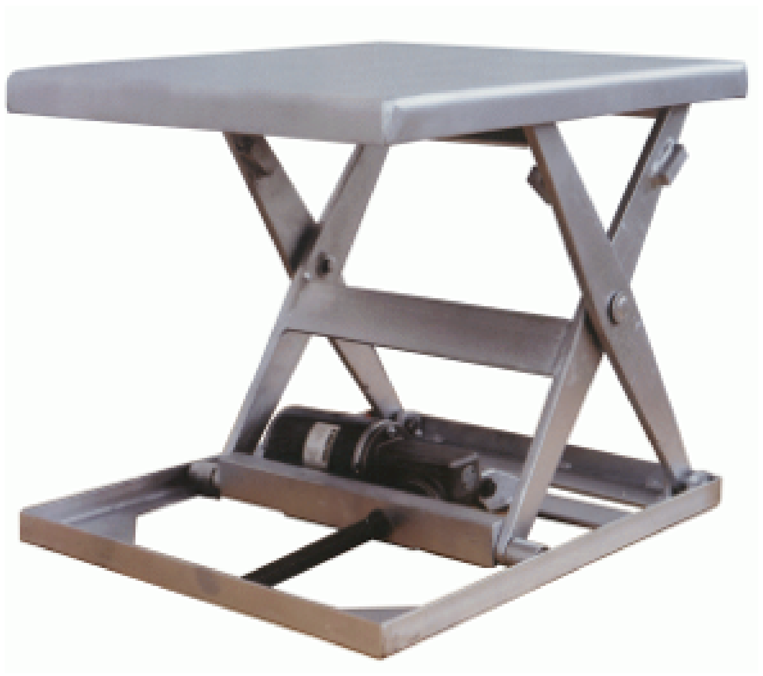


Figure 23: Motorized scissor lift design

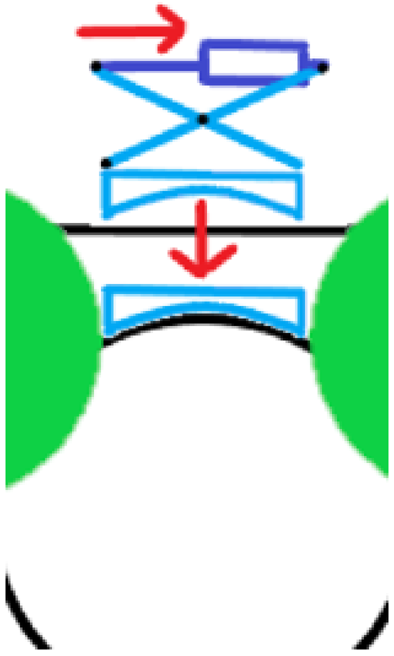


Figure 24: Pnematic alignment device

We also discussed the importance of having "alignment" devices on our robot. Since this game is very dynamic and there are no protected zones, it is very likely we will be approaching goals fast and will have opposing teams pushing us around while trying to score. In order to efficiently keep us close and aligned to the goals, and prevent us from missing shots into the goal, we wanted to think of mechanisms to keep us in line with the goal. One idea we had was to use an actuating scissor lift type of alignment similar to that shown in Figure 24, allowing us to push a platform or other curved surface forward, so we could run into the goal and automatically center on it. We were targeting one of the bottom two rings in the goal for alignment, which the curved surface would align to as we drive into it.

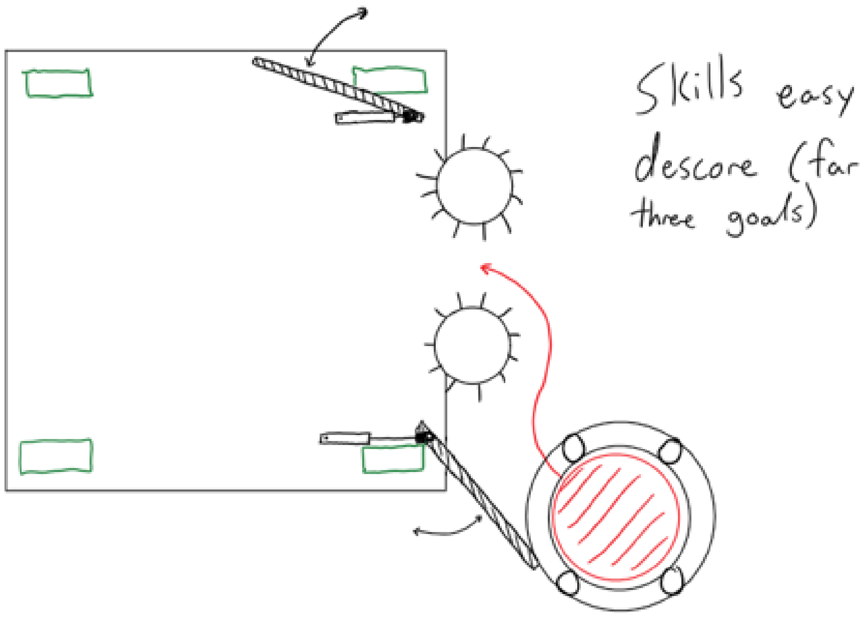


Figure 25: Center goal descoring fins idea

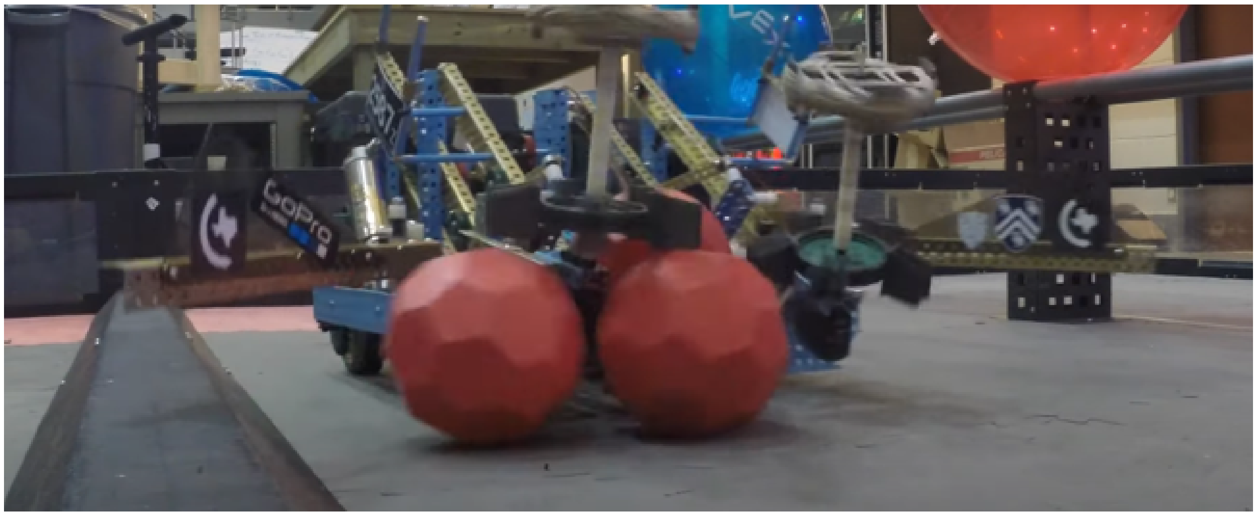


Figure 26: Ball capture fins from VRC 2587 in 2014

Another topic we discussed was the efficiency of skills runs, and how we could make quick movements to de-score the opposing balls. As COVID doesn't seem to be going away soon, we assume that skills will be an important part in competing and qualifying for the World Championships. We thought that deploying "wings" that would allow us to scoop balls from the bottom of goals would be an efficient way of emptying the opposing balls from the goal, with minimal cycle time on the goal. This would also allow for routines where we drive along the wall and empty all three goals, without needing to stop directly at each goal.

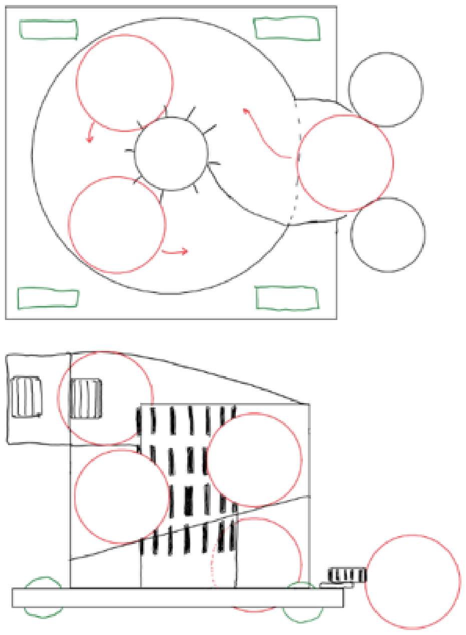


Figure 27: Rotary hopper idea

Figure 28: FRC # 33's 2020 robot with rotary hopper

A concept we talked about was ball storage. We believed that higher ball capacity was important to have, as we will often be crossing the field, and we want to be able to store our balls within the robot at any opportunity we get. For instance. If all of our balls are on our side of the field and we need to go score goals on the other side of the field, it would be much more efficient to pick up everything on our side and score it all on the opposing side, when compared to running multiple cycles back and forth. With this idea in mind, we had an idea of a cyclical container with a center brush to rotate the balls around the robot. This was based off of a design from FRC 33 in their 2020 robot as shown in Figure 28, where they passed balls around a rotary hopper with a set of spinning brushes. This mechanism would only really require one motor, as brushes can be used from bottom to top to spin the balls around the hopper.

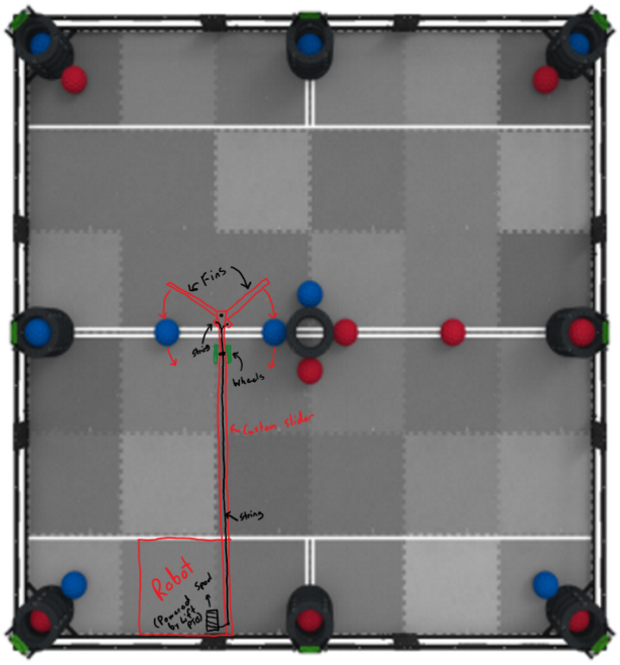


Figure 29: Ball capture idea for early match

We discussed the importance of autonomous in our matches. As we learned last year, autonomous played a significant role in putting teams in an advantage at the start of the match. A team with a good autonomous can easily set up an unbeatable lead, even when the opposing robots are better performing during the remaining match.

With this in mind, we wanted to think of ideas to get control of a majority of the game pieces early into the match, making us much more likely to win the autonomous bonus. For this, we discussed having an expanding arm that would be used to pneumatically pull the balls in center field onto our side of the field.

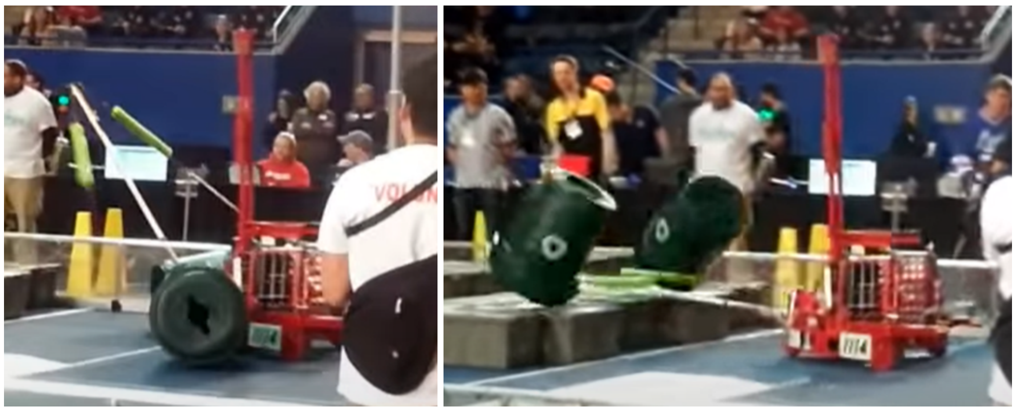
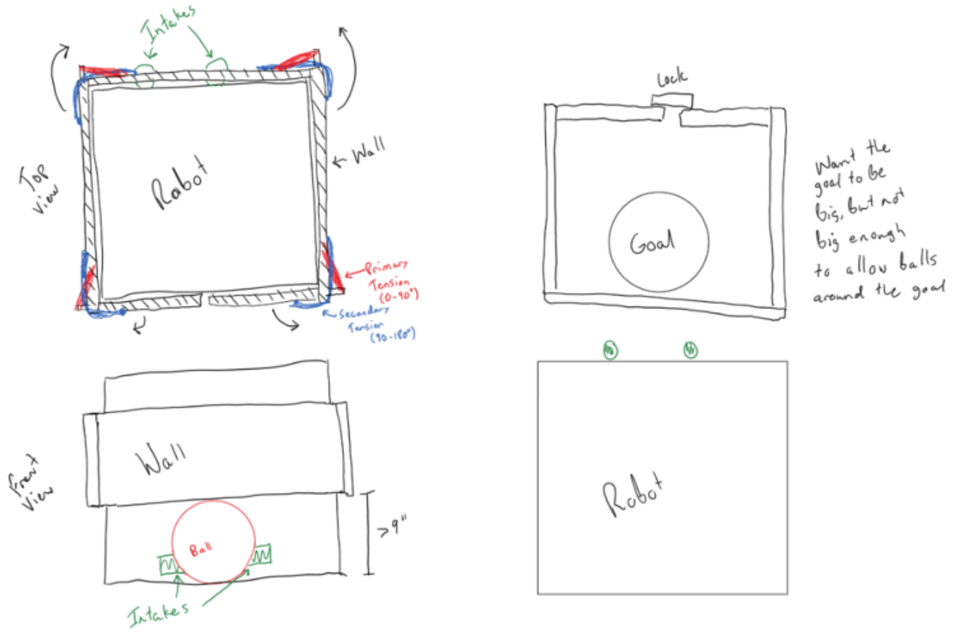


Figure 30: FRC #1114's 2015 can grabbers

This is analogous to the "can races" in FRC during the 2015 game, Recycle Rush. Teams designed mechanisms specifically for pulling cans onto their side as they would provide noticeable contributions to the alliance's score. One design we looked at in particular was from FRC 1114, who used an actuating arm shown in Figure 30, which expanded linearly and used two pivoting arms to pull cans towards them. This could be used to expand to our center line, and push balls over to our side, being collapsed by a winch or other mechanism early in the match.


Intakes
Top View
Robot
Wall
Primary Tension (0-90°)
Secondary Tension (90-180°)
Front View
Wall
>9"
Ball
Intakes
Lock
Goal
Want the goal to be big, but not big enough to allow balls around the goal
Robot
Figure 31: Deployable wall-bot for center goal

Another idea we had was on protecting goals from being de-scored. To prevent from breaking rules on "grabbing" game elements, we thought of creating a big box that would cover the bottom portion of the goal and prevent other teams from removing balls. At the start of the match, this would be folded around the outside of our robot and would flip out and deploy around the goal once we were in position.

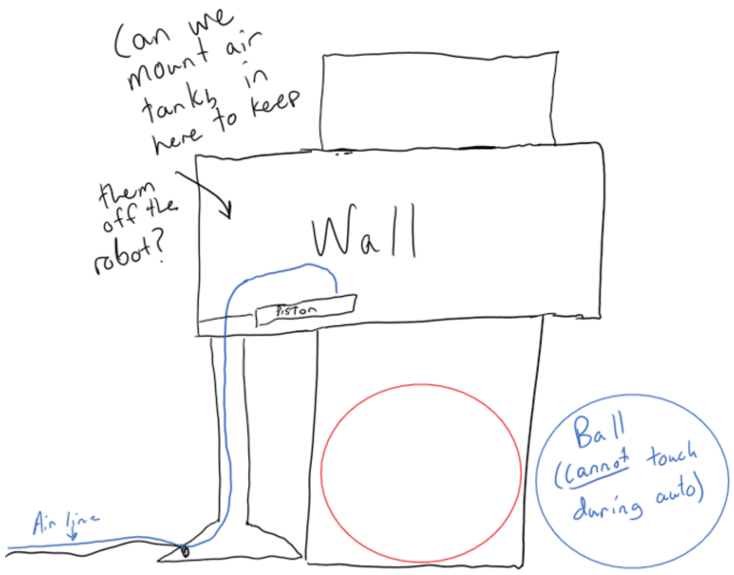

Can we mount air tanks in here to keep them off the robot?
Wall
Piston
Air line
Ball (cannot touch during auto)

Figure 32: Wall bot deployment during autonomous

To allow us to deploy the wall-bot on the center goal during autonomous (without touching the other side of the field, or the game pieces on that side), we looked at using a tower to hold it above the goal until the driver control starts, and then letting it fall to protect the goal.

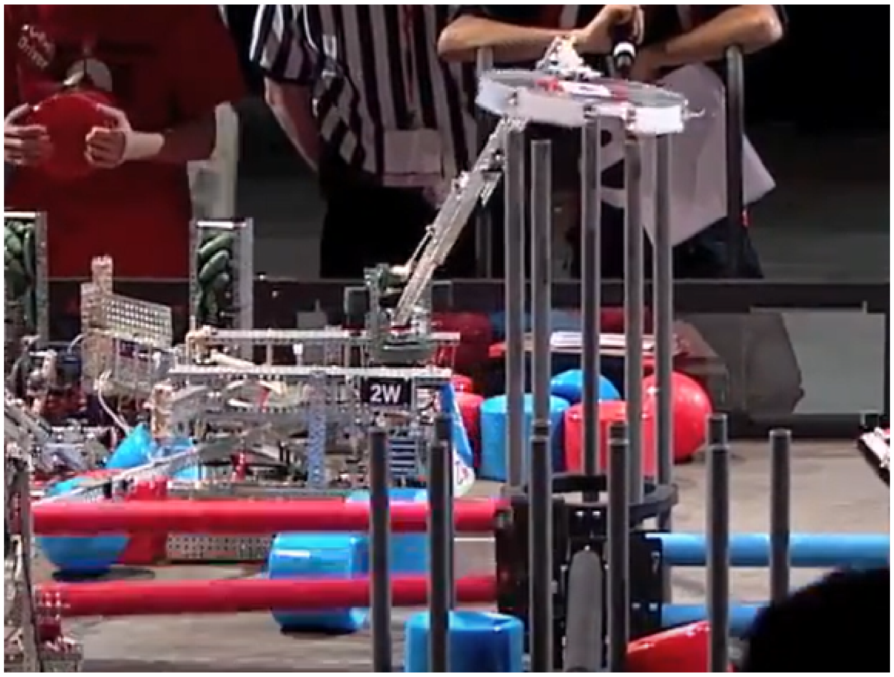


Figure 33: 2W's 2012 robot defending a center goal

Another idea for defense that we looked at was covering the tops of goals, with either an arm, a deployable plate, or our robot itself. This is similar to the idea shown above, as 2W that year was able to successfully prevent opposing teams from scoring in the cylindrical goals with their goal topper, which is similar to what we were thinking about for this year.

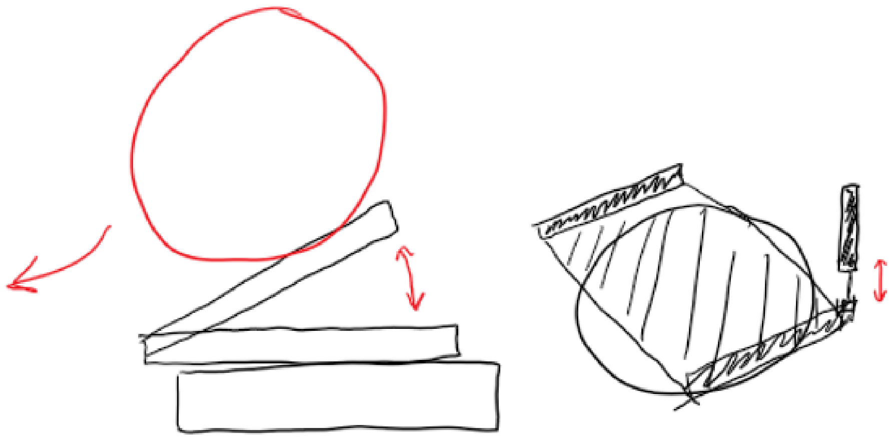


Figure 34: Goal tender for ejecting balls from goals

To remove the need for us to need to manually de-score goals, we looked at deploying small "goal tender" tether bots that would be attached to our robot with cables/string but would be deployed in proximity of a goal. This idea would allow us to de-score certain key goals (such as corner goals) with a tether-bot automatically, which would remove the need for us to go to the goal to remove balls. This would vastly increase cycle times and allow us to control key points on the field that teams would need to get high numbers of row bonuses.

2.2 V1 Robot Design Cycle

Overview

V1 of our robot was built for the Upper Midwest VEXU Pre-Recorded Skills Only League. The final iteration of V1 features the following revisions of mechanisms:

- V4.1 Drivetrain (subsection 4.6)
- V2 Intake (subsection 7.2)
- V1 Hood (subsection 8.1)

V1 features pneumatically spreadable intakes with a wide intake range, an ejector mechanism capable of quickly ejecting opposite colored balls, a tank drive geared for high travel speeds, and a hood mechanism capable of placing balls into goals with and without a backboard.

We started building V1 over winter break (end of December), with the design reaching completion at the conclusion of the Upper Midwest VEXU League (2/14/2021). Due to the situations around COVID-19, we did not get to start building this revision until much later than we would have liked.

During the Upper Midwest VEXU League, V1 scored 116 points in driver skills, and 0 in programming skills.

Design Review

Wins:

- Robot is about as dense as a 15" robot can get, fully utilizing the allotted 15" starting configuration cube
- It worked! Scored pretty decent driver and programming skills
- Actuated intakes were a good idea. Really improved intake range
- Able to hold a ton of balls (6-7)
- Had a good ejector system
- Drivetrain was fast. Good speed
- Pretty much impossible to tip due to intakes acting as wheelie bars
- Hood design takes a lot of space, but compacts really well
- Centering mechanism worked really well
- Follower omnis worked well

Areas for improvement:

- Robot is rainbow colored :(
- Programming skills was rushed at the end, did not have enough time to create full 1 minute run
- Did not utilize ejector system
- Intake actuation was hard from a driver controls standpoint. Possibly add automation?
- Lower center of gravity would help driving
- Better implementation of hood would be nice. Three small gears on top of hood do not mesh super well
- Right and left sides of hood flex a lot
- Drivetrain gears flex apart and don't mesh well when they do
- Entire superstructure needs more rigidity
- Hard to get intakes into starting configuration
- Intake rollers interfere with posts on center goal if not perfectly lined up
- Really really hard to get robot into starting configuration
- Centering could be better for corner goals
- Run through a lot of rubber bands, they seem to break a lot on the intake
- Chains on intake are hard to deploy right (get stuck, break)
- Hood back material was not great, would cause entire hood to flex back and create deadzones
- Hood is under powered, could an extra motor
- Not enough pneumatic tanks (could use another tank)
- **Optimize design for skills.** Robot does not need to be as tall as it is with only skills
- Robot could be even faster (look into lighter robot, faster wheel gear ratios)
- Programming needs a lot of work (path following did not work, turning to angle was not great)
- Need better way to record runs
- Webdashboard needs some work (not stable)

- Outer wheels on intake are exposed, when they touch the walls the intakes stop (big problem with the corner goals)
- Need dedicated space for Raspberry Pi and battery bank
- Need to start recording runs earlier
- Want corner alignment jig for corner goals

2.3 V2 Robot

Check the drivetrain page for a specific drivetrain information.

Overall peacock and lift roller:

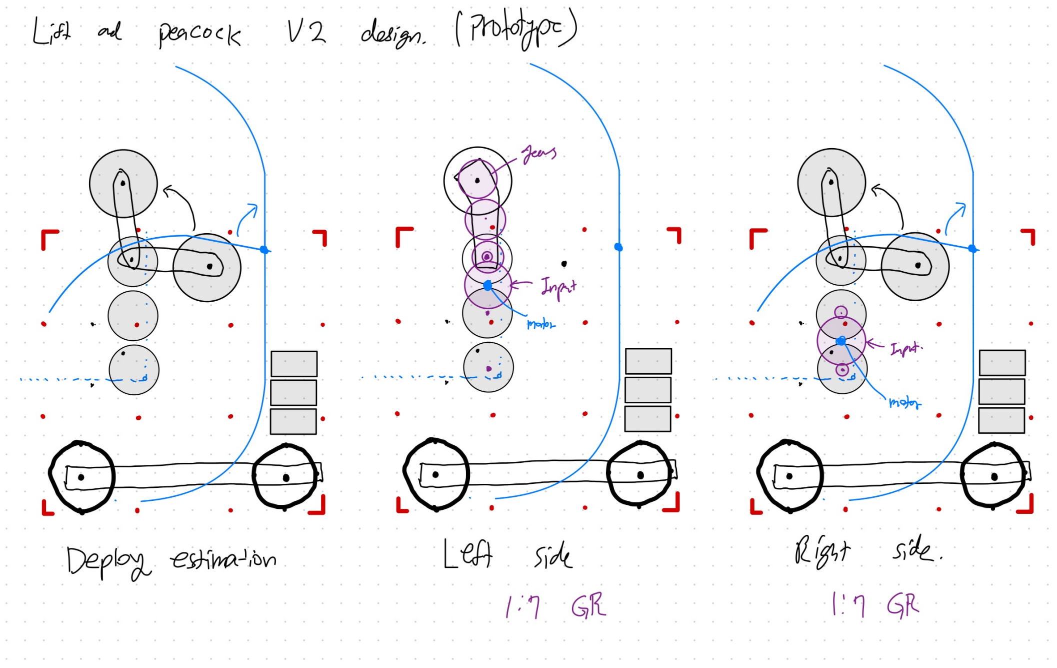


Figure 35: side view of deploy, gearing on left and right

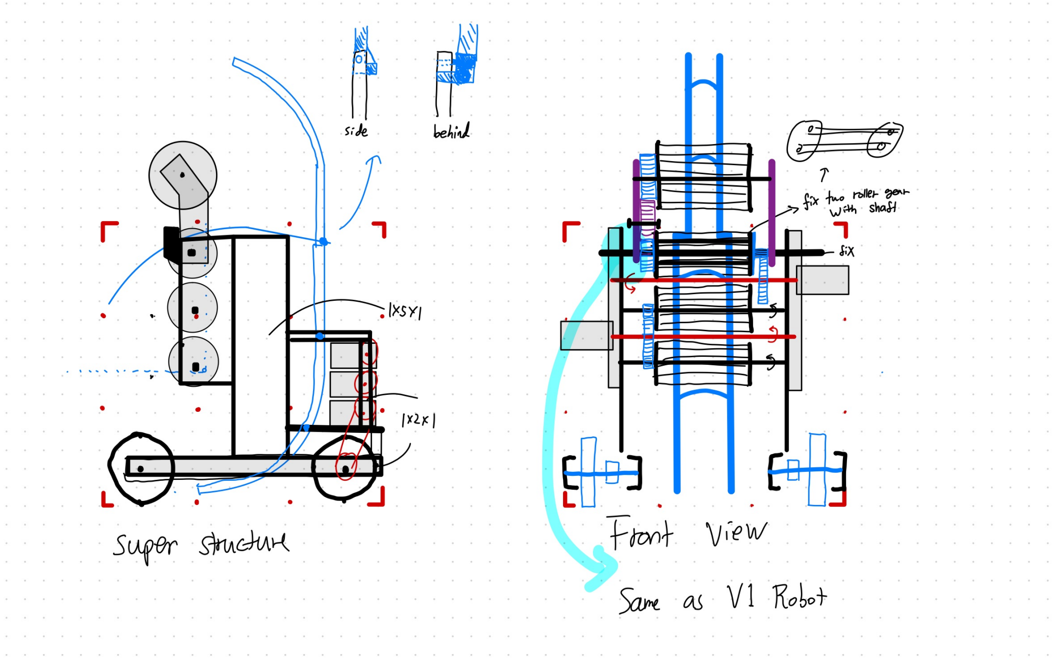


Figure 36: super structure and front view

Goals for V2 robot:

- Better for programming skills
- Light fast robot
- Lower ball capacity

The V2 robot is intended to be a major robot iteration which will feature an all new robot design. This allows us to improve the robot across the board in dramatic ways. In particular, the architecture of V2 is radically different from V1 since V2 is intended to be optimized for programming skills specifically.

3 Strategy

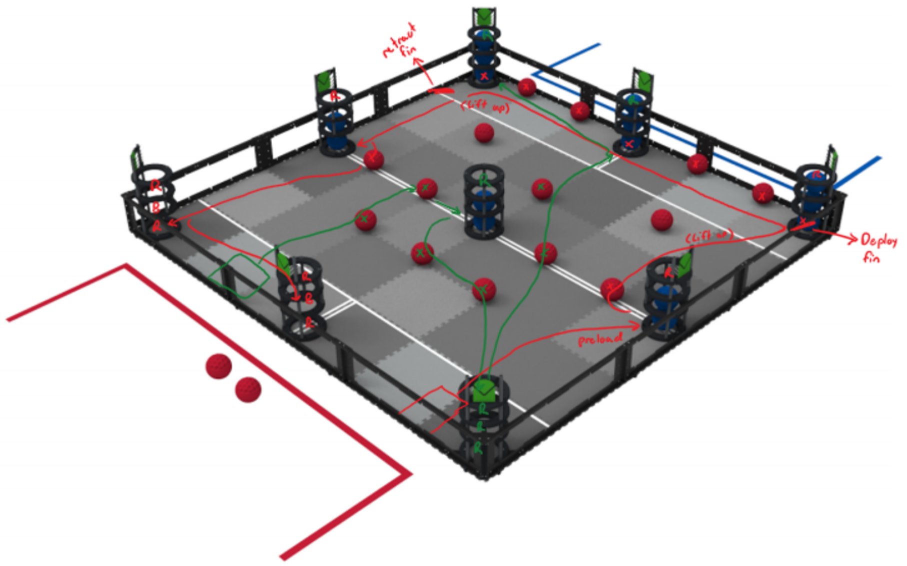


Figure 37: Potential skills routine

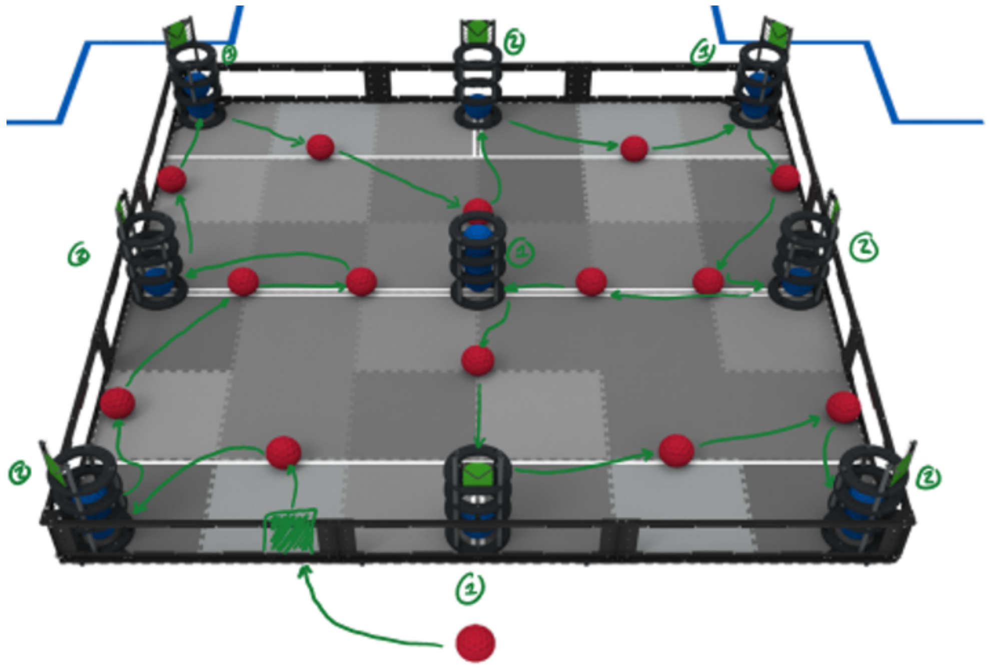


Figure 38: Potential skills routine

4 Drivetrain

4.1 V1 Design Cycle

The goal of this design cycle is to create a drivetrain suitable for our 15” robot. In creating a suitable drivetrain, we have several criteria that should be met:

1. We want a drivetrain that is maneuverable and fast so it can effectively traverse the entire field. This game is very dynamic; we expect to be driving for most of the match, and we will often be crossing the field to score points. With this in mind, our cycle time from scoring a ball on one side of the field to scoring a ball on the opposite side of the field should be minimized to let us score points and address opponent scores as quickly as possible.

2. We want a drivetrain that can be strong in pushing matches. Since we will often need to cross the field, and since the center and outer goals effectively section off the field, we predict that we will encounter pushing matches in normal gameplay. We want a drivetrain that will have lots of power for pushing, while also being able to maneuver around opponents easily

3. We want a drivetrain capable of fine adjustments, or precision. Since the 15” robot is also the robot that we need to use for skills, we want to easily be able to get into a position to pick up a ball on the field, as fast and easily as possible. This also means keeping us in a good orientation for future movements, to make the entire run more efficient.

4. We wanted a drivetrain that had a small footprint within the robot. Since we have limited space within the robot, having the drivetrain take up as little space as possible is ideal. Especially for future design, we need a side to intake balls into the robot, without having them go over the frame

5. We want an innovative design for our drivetrain. While not a primary criteria, we all want to learn new things while competing this year, and create a robot which is unique and interesting. We want to find new ideas to challenges to help grow our robots for the future.

Starting on this problem, we brainstormed several different types of drivetrains that we believed would meet this criteria. Through past experiences, data we found online, and simple testing, we gathered details about each. For some of them, we also took the time to create CAD mockups in order to better understand how a frame might be constructed around them.

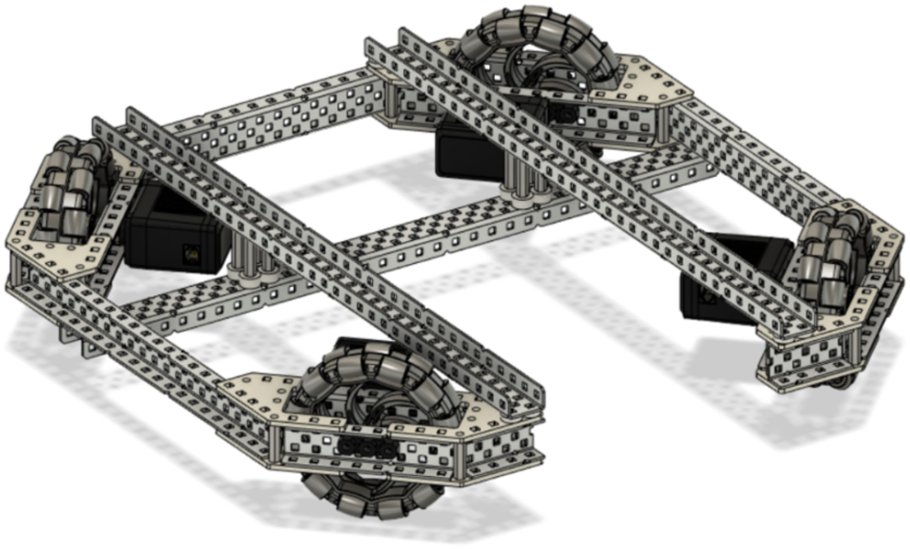


Figure 39: A CAD model detailing a X-Drive frame made by Vex team 7405P

Idea 1: Holonomic “X” drive This design uses four omnis mounted in an x configuration to drive the robot in any direction. Although it has extremely low friction due to the omni wheels, it has limited power since only two motors help drive the robot in the four cardinal directions.

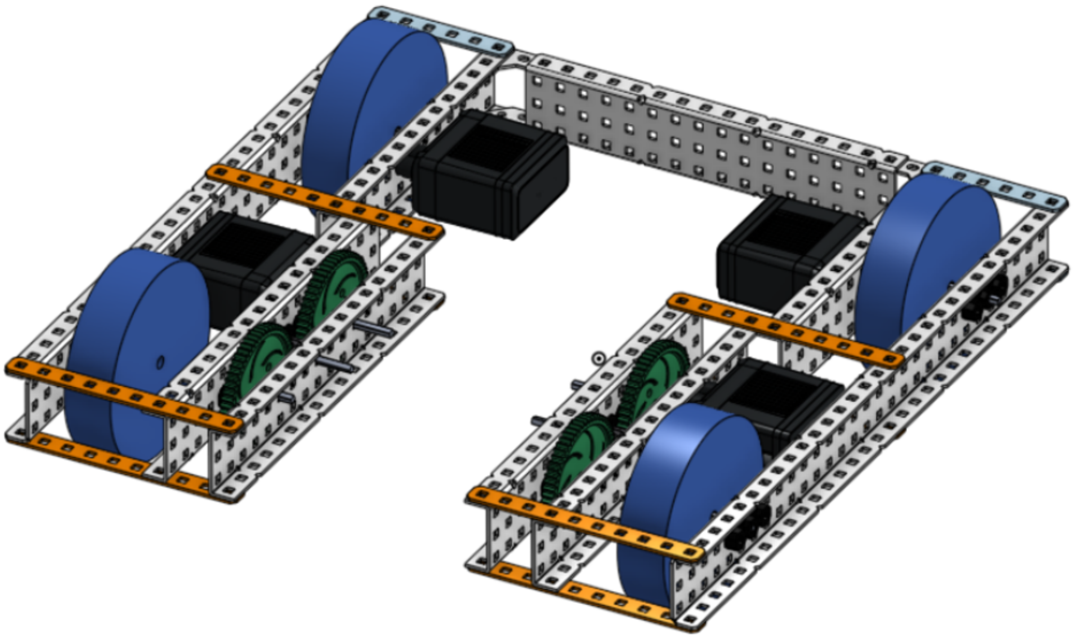


Figure 40: A CAD model created by Daniel for a potential tank drive design

Idea 2: Tank drive A tank drive is a simple and robust drivetrain that is reliably used in VEX and VEXU by a large amount of teams every year. However, it is limited in its functionality and versatility options available to the driver

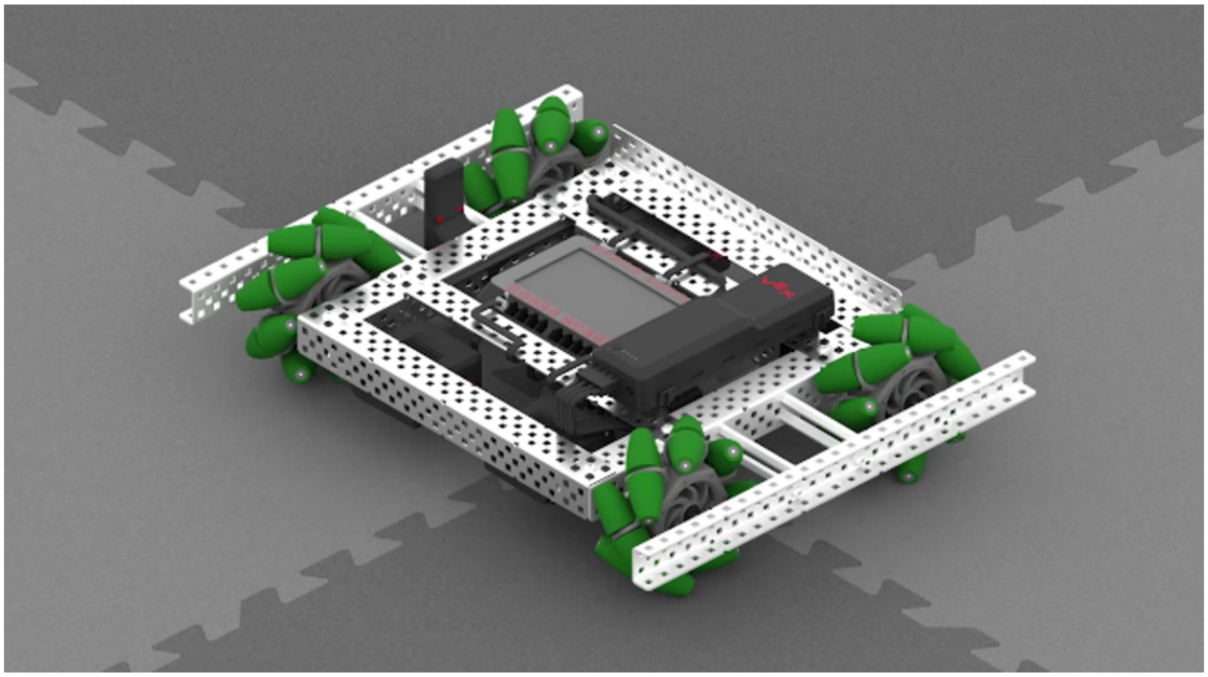


Figure 41: CAD model of a mecanum drivetrain from the BLRS Wiki

Idea 3: Mecanum drive A mecanum drivetrain is very similar to a tank drive, but uses special mecanum wheels (vectored omni wheels at a 45 degree angle) to enable omnidirectional movement. The downside is that the angles of the wheels create lots of additional friction, which can slow the robot down and making pushing much more difficult.

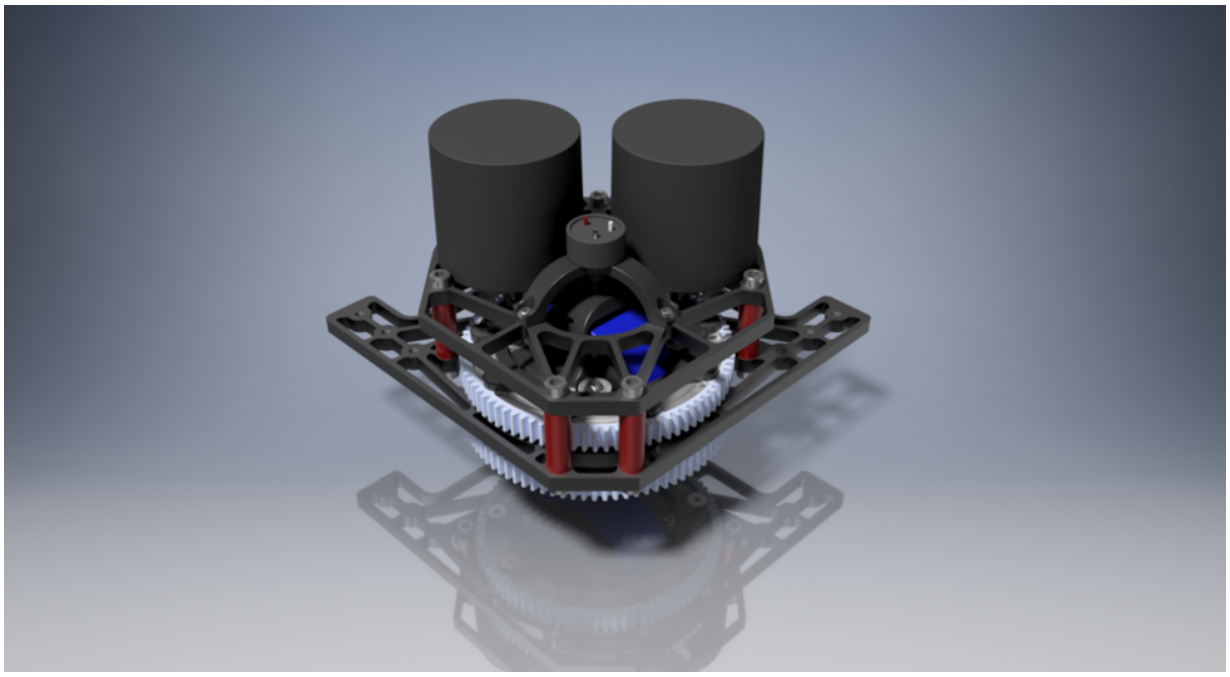


Figure 42: An example differential swerve created by Trevor Glasheen on ChiefDelphi

Idea 4: Differential swerve drive We noted that a differential swerve typically uses two rings independently driven in opposite directions to create motion. When the rings spin perfectly in opposition, the wheel spins forwards or backwards. If a difference in speed exists, then the entire wheel rotates, allowing the wheel to face in any direction.

We then created a decision matrix comparing the different designs. Notably, we also decided to add a somewhat subjective "coolness" factor to the decision matrix, since we feel that as a VEXU team, it is important for us to attempt to create a robot that is not only effective, but also interesting and unique.

	Maneuverability / Speed	Pushing Strength	Precision	Size	Innovation	Total
X-Drive	4	3	4	2	2	15
Tank	3	5	2	4	1	15
Mecanum	3	3	3	3	2	14
Differential Swerve	5	5	5	1	5	21

Table 1: Decision matrix - drivetrain

Ultimately, the differential swerve beat out the other designs. While more complex and larger than the other designs, individual movements in each wheel allow for much more precision in movements around the field. Differential swerve also had the most available pushing power out of all of our ideas. And although differential swerve has significant downsides in both size and complexity, we felt that we would ultimately be able to overcome these disadvantages thanks to our extremely capable team of designers.

Once we settled on building a differential swerve, we created a plan for actually designing and building the swerve. Because of the extreme complexity revolving around swerve, the plan for developing differential is much more involved and complex than the design plans for other mechanisms on the robot. That is also why we're choosing to start on differential first, before beginning to focus on the other aspects of the robot.

We plan to start by creating a V1 design of differential swerve which will serve as a proof of concept. For this initial version, we plan on pursuing an architecture similar to Armabot's commercially available differential swerve drive for the FIRST Robotics Competition (FRC), scaled down for our much smaller VEX robots. Since this version is mainly a proof of concept before we move on to the next design, we don't plan on adding chassis mounting, and we don't expect to make more than a single module for testing. V1's utility will also likely be constrained since it will be a very large, but the primary goal is to create a proof of concept to determine if differential swerve is feasible in VEX before focusing on heavily miniaturing the module.

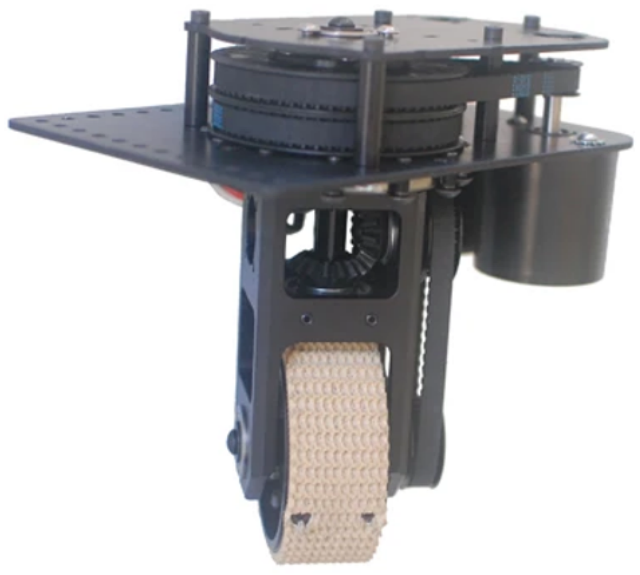


Figure 43: Armabot's commercially available FRC differential swerve

After completing and testing the design of V1 differential, we plan to then move on to V2, which will most likely be based on a different architecture than V1. In particular, we hope to make V2 a much more compact design which is more suitable for VEX. We believe that the main way we will accomplish this is by shifting to a design similar to the architecture we discovered while first researching differential, where the wheel is sunk between the differential gears in order to achieve a lower profile. Similar to V1, we don't plan to worry about mounting to the chassis or producing a full set of modules for this design since the design will still likely be flexible at this stage of the process, especially given the shift in architecture between V1 and V2.

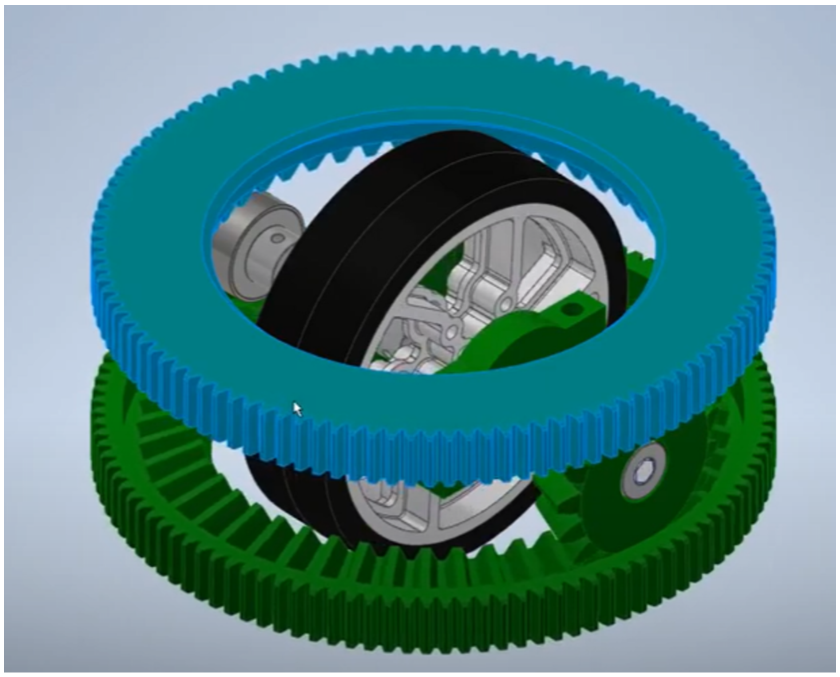


Figure 44: Compact differential swerve idea

Lastly, we plan on creating the final V3 version of differential swerve, which will be the version we plan on ultimately using on our competition robots. Thus, this will be the version of differential which will feature mounting to the chassis, which will most likely be developed concurrently with this version. Depending on how the design progresses, we may also choose to design and/or create additional versions of each planned major version of differential swerve depending on whether we feel additional changes and design work is necessary. Also, after completing the V1 version of differential swerve, we plan to also use our experience from V1 to reevaluate our plan and decide if we need to change our development plan in some way or switch to a different type of drivetrain altogether.

After deciding on this plan, we started working on designing the first V1 differential swerve in CAD. Notably, we decided to use small plastic ball bearings to enable the differential gears to spin more easily. We also added an encoder to the top of the module to track the orientation of the wheel, and used a vex bevel gear to transfer motion down to the wheel.

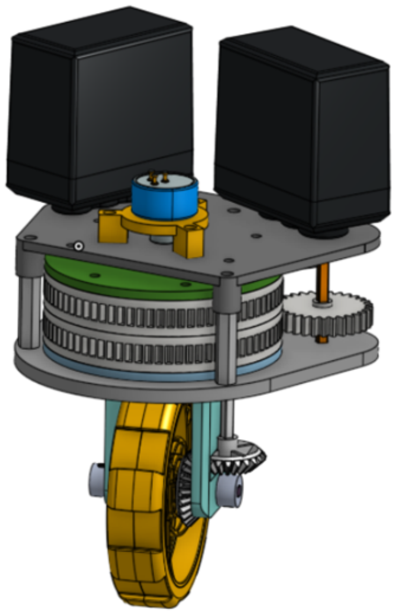


Figure 45: V1 of the differential swerve

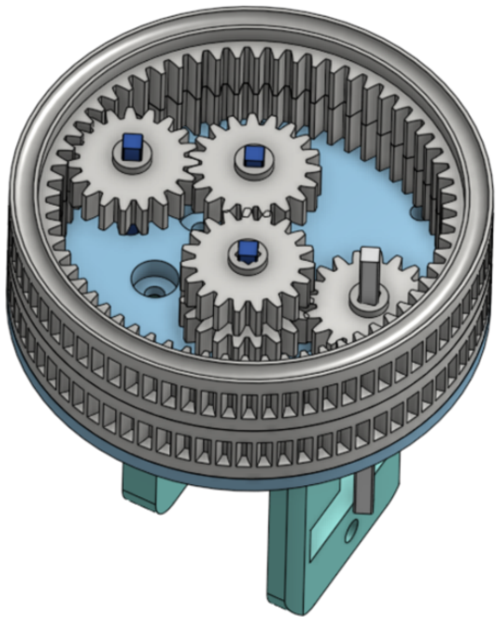


Figure 46: The inside of the differential swerve gearbox

After completing the initial design of V1 in CAD, we performed a design review, looking at how we felt about the design so far. One idea that was put forward during the design review was that we could switch the gears in the swerve to use a herringbone pattern, which would give the gears additional strength and help keep the gears in line with each other. We really liked that suggestion, so we then decided to update the design of V1 to use a herringbone pattern. We also instituted a few other minor changes to the design, like adjusting the standoffs to be slightly longer and reinforcing the forks before production.

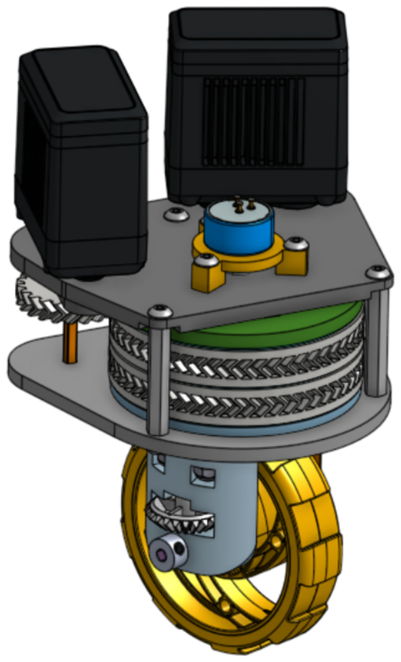


Figure 47: The CAD of the second iteration of V1 swerve

After performing another design review of the revised design, we felt good enough about it to start getting it printed off and assembled.

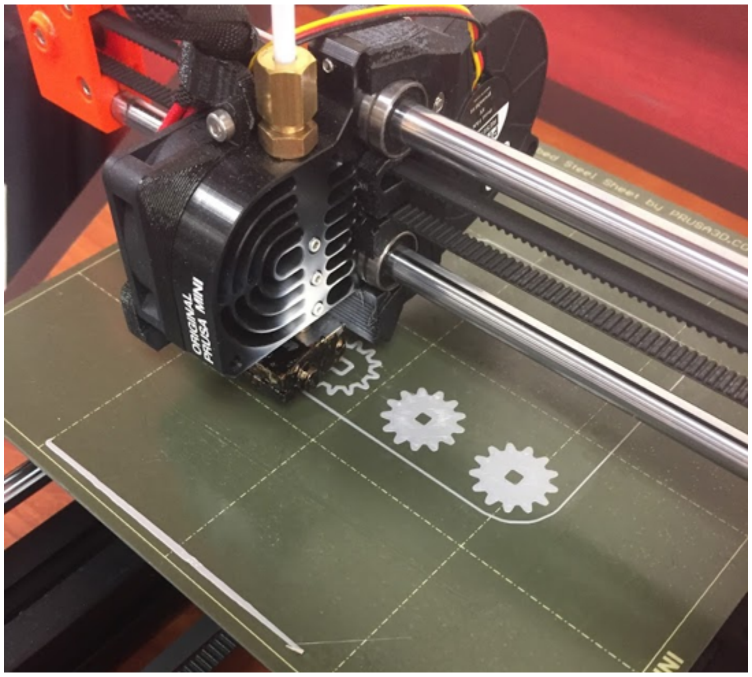


Figure 48: Initial printing of herringbone planet gears on a Prusa Mini

While assembling V1, we quickly realized that the small plastic ball bearings are extremely annoying to use, as they require tweezers to pick up and are easily dropped or misplaced. Furthermore, this design required many gear stages internally that were challenging to accurately assemble in the proper configuration.

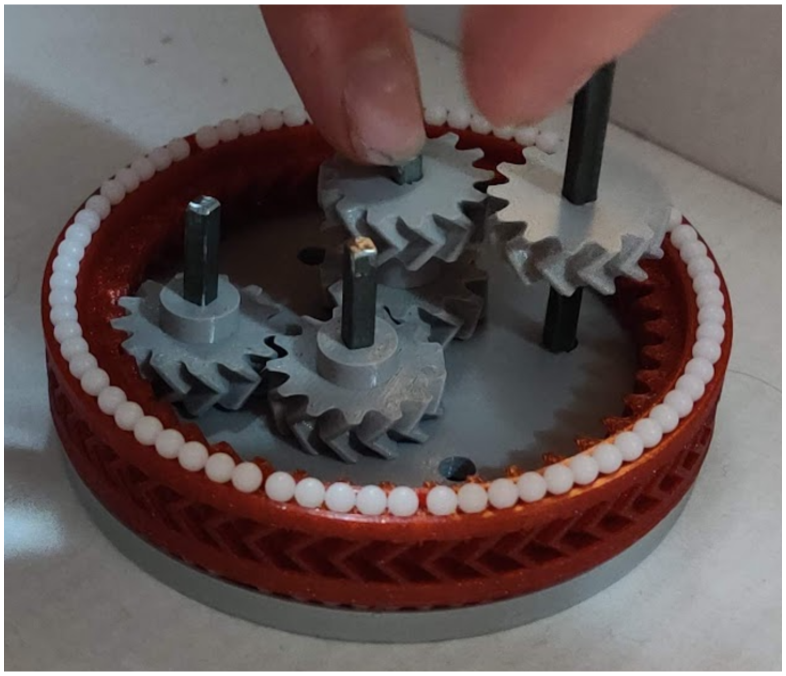


Figure 49: 2nd layer of ball bearings and herringbone planet gears

We then proceeded to test V1. The main focus of our testing was on the strength of the module, especially the many gears inside it. In order to do so, the gears were sufficiently strong to transmit the force required, and we were unable to break the gears or other internal parts during testing. This made us confident that a 3D printed differential swerve would probably work on the robot, assuming we could get it small enough to fit within frame. We also concluded that the ball bearings were probably unnecessary since 3D printed plastic was able to perform as low friction bearing surface, especially when lubricated with silicone in low RPM applications such as VEX.

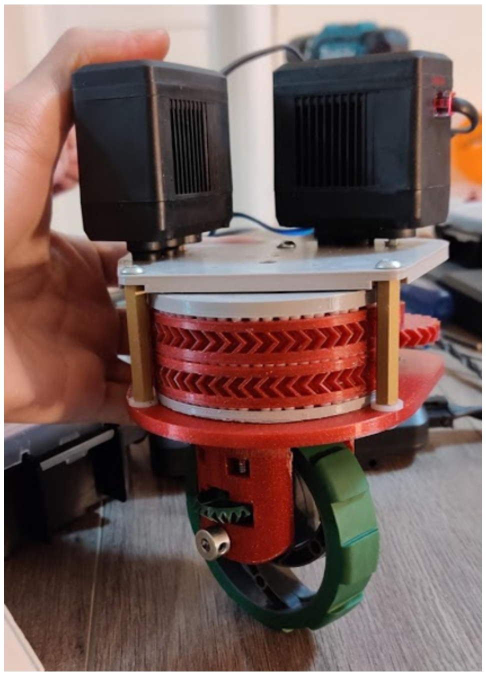


Figure 50: The fully assembled V1 differential swerve module

After finishing V1, we revisited our original decision matrix to decide whether we needed to make any changes to our plan. However, because of the success of the initial version, we decided that no changes were necessary, and that we could continue on with our original plan, working to next miniaturize and simplify the design of the first module.

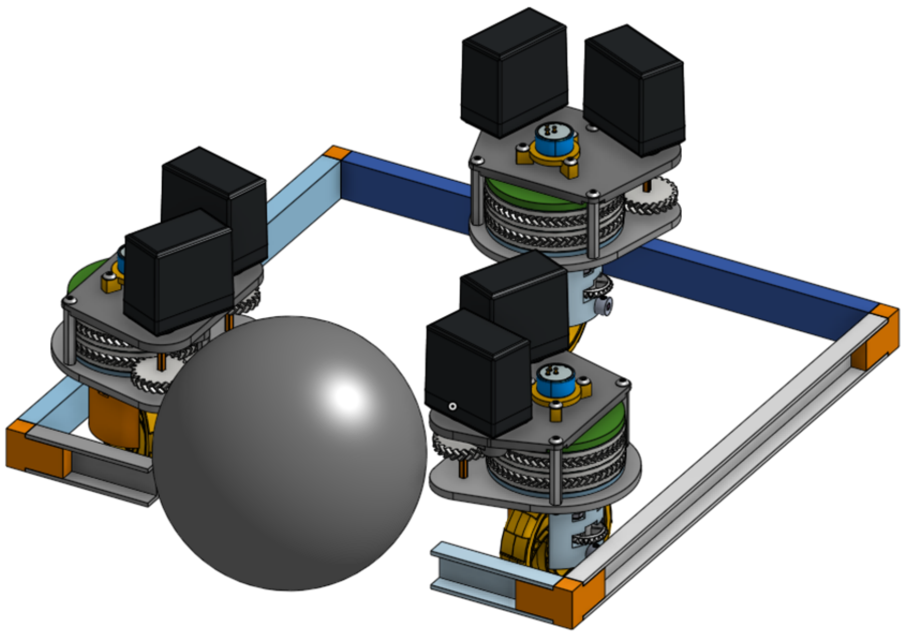


Figure 51: A demonstration of the tight space requirements for a V1 differential drivetrain

4.2 V2 Design Cycle

The goal of this design cycle is to redesign the V1 differential swerve created during the first drivetrain design cycle to be more compact and manufacturable by changing the architecture of the module to have the differential gears above and below the wheel rather than entirely above it. The design should also use a smaller custom wheel and avoid using plastic ball bearings, as they were hard to assemble with marginal benefit.

Because we already have a relatively good idea of what we wanted this version of the module to look like, we decided we would once again use our standard CAD-prototype design workflow described in [appendix something] to create this version of the module. With this in mind, we can now begin designing the V2 module. The first part requiring the development of the bevel gears.

As a single custom printed bevel gear pinon would be transmitting all the power to the wheel, we had concerns about the durability of such a design. As such we printed, analyzed and revised several sets of bevel gear sets before determining the best ones for the application and confirming that they were strong enough after being exposed to shock loads. The initial version used Autodesk Inventors' Bevel Gear Generator, but after printing it was discovered that the generator only created approximations of the bevel teeth and due to the small size of the bevel gear, this approximation was significantly different than the involute profile that was required.

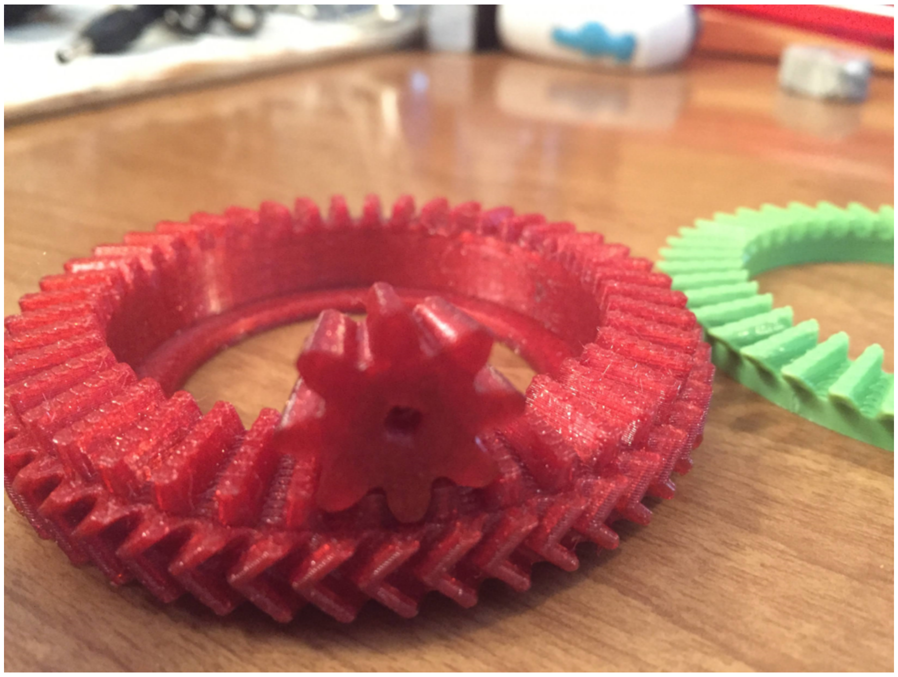


Figure 52: V1 bevel gear set

Due to this, we designed our own custom bevel gears following the process outlined by Brian Zweerink to convert the involute spur gears in Inventor to bevel gears at a desired angle. This next version had a correct profile, but it also had significant undercutting, which compromised the integrity of the gears. Although we were unable to shear the gears in our basic testing, it still was not worth running the risk of them shearing off over time. As such another iteration was needed.

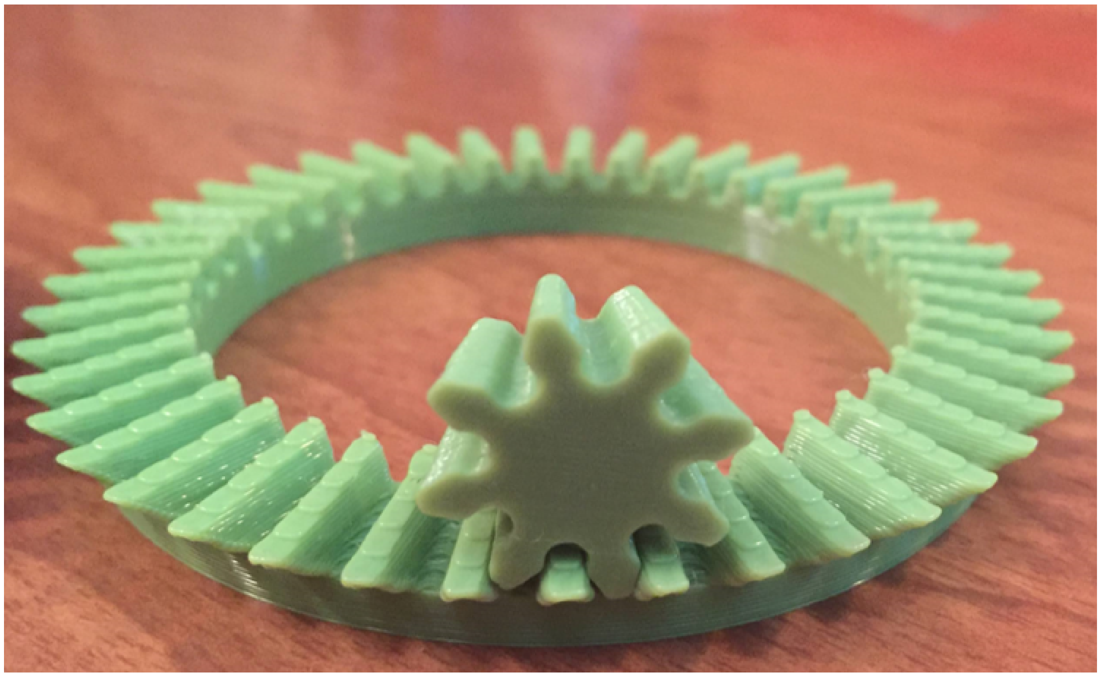


Figure 53: V2 bevel gears with undercutting

The next version specifically focused on preventing undercutting by changing the pitch adjustment and was successful in removing the undercutting from the teeth, however it did so at the cost of tapering, which also reduces tooth strength. For the next revision we would attempt to change the number of teeth among other settings to try to get a better output while still producing the size of gear required to still have the wheel protrude from the bottom of the module as necessary to drive.

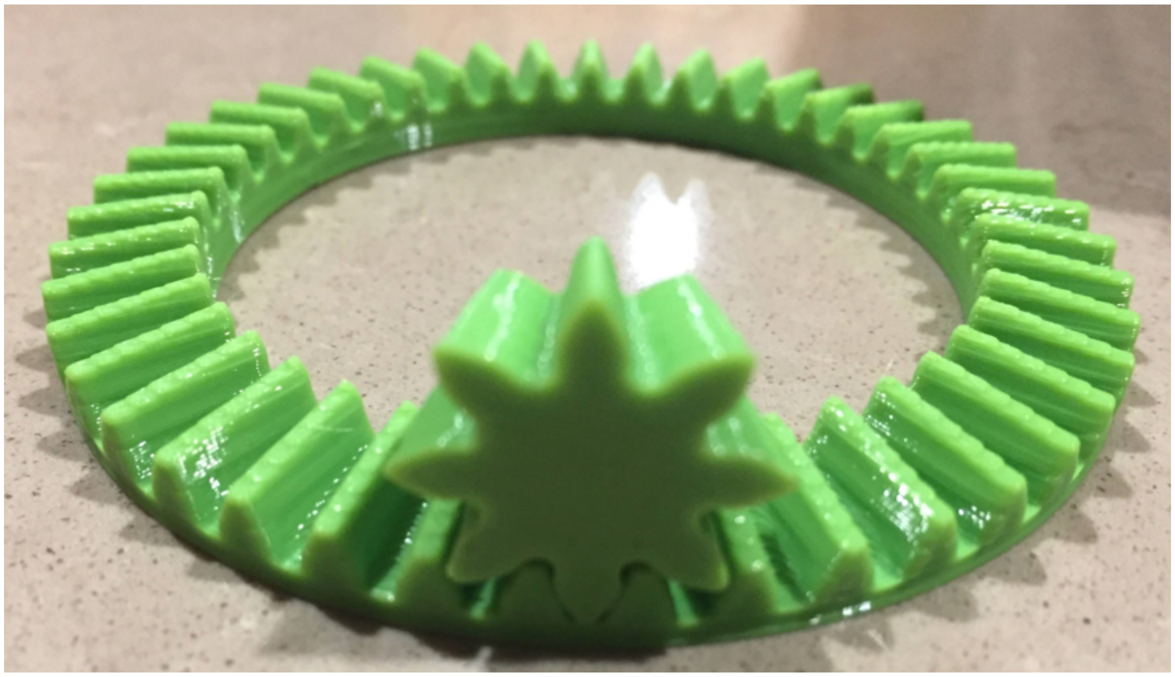


Figure 54: V3 bevel gears with tapering

The fourth iteration of the bevel gears was much more successful and with an additional tooth, we were able to avoid the pitfalls of tapering and undercutting of the teeth, greatly increasing the strength and durability of the gear and is the one that we selected to move forward with for V2 of the differential swerve.

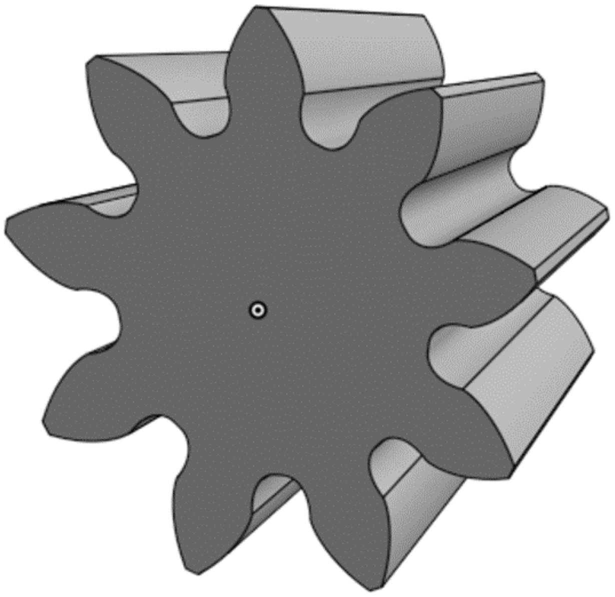


Figure 55: V4 bevel gear

With these bevel gears, we were able to complete the rest of the design in Onshape within the next few days for 3D printing.

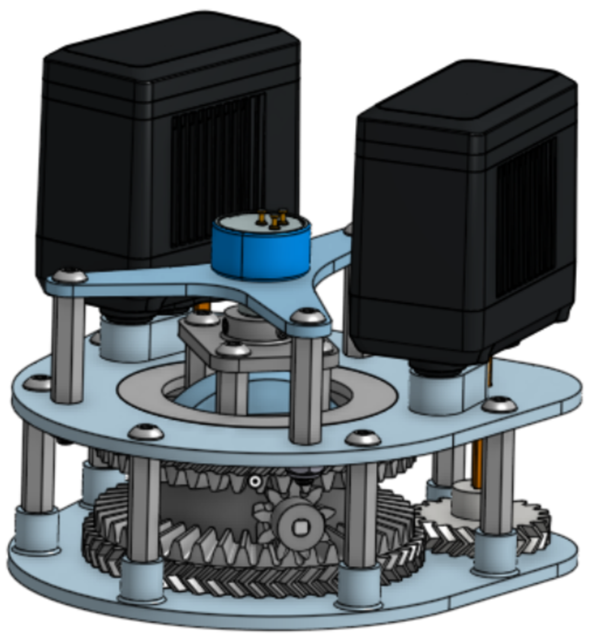


Figure 56: CAD of the initial design of V2 swerve

We found that the assembly of V2 was much simpler than in V1 and was easier to maintain due to its lesser part count and increased accessibility. The design also had less friction between the motor and the output wheel due to the fewer gear count despite not utilizing ball bearings.

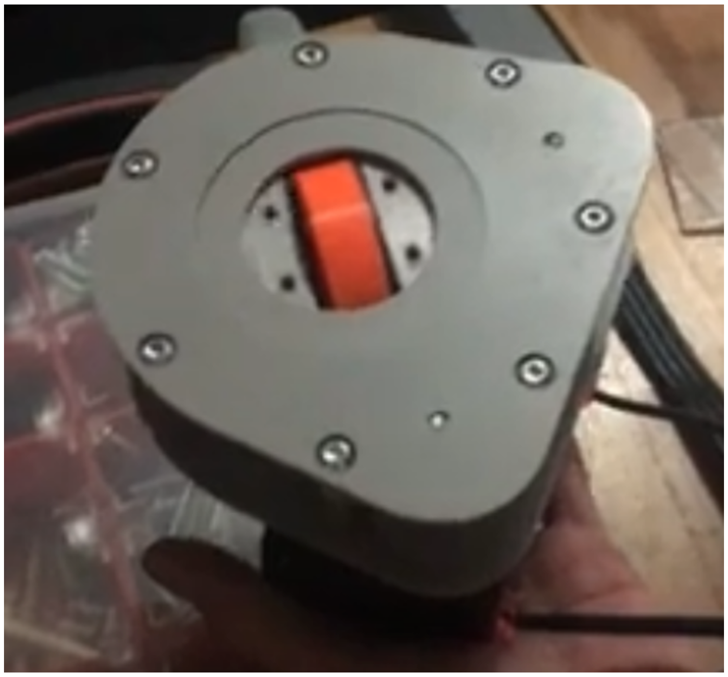


Figure 57: Testing V2 of the differential swerve

With this version, we were largely meeting the goals we had laid out in the initial planning of the design, but there still were some improvements to be made. First off. We had used a fully 3D printed wheel test the geometry and function of the module, but with testing it was evident that the coefficient of friction between the PETG wheel and the foam tiles would result in losing the effectiveness of a six motor drivetrain due to slippage. As a result, we would need to make the custom wheel be printed out of a more grippy filament or attach an external tread to the module. In addition, with the current profile of the module although more compact than in V1, it still was too restrictive on the 15" robot to add an intake and a ball could not pass between the two modules on the front to corners. Now that V2 had proved the concept was feasible for VEX, we now worked to package this in an even slimmer profile that would work with the rest of our robot's architecture.

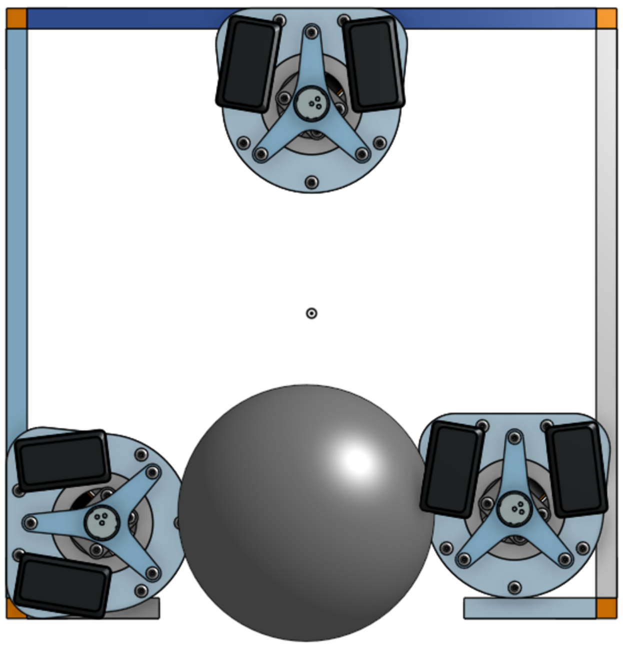


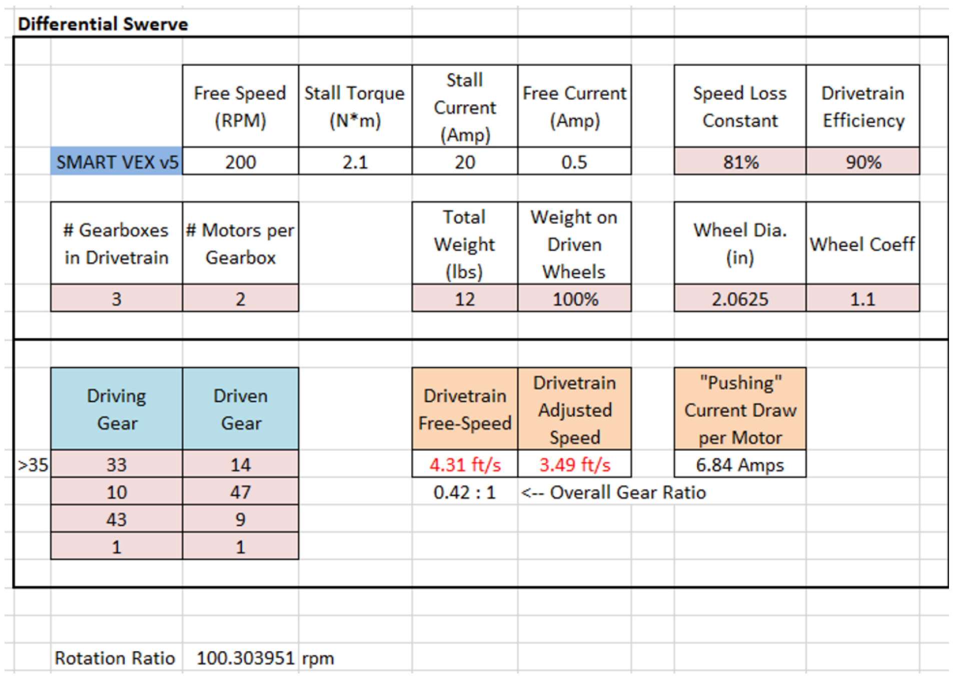
Figure 58: Top view of V2 differential swerve inferences with ball

4.3 V3 Design Cycle

The goal of V3 of differential is to create a set of three modules that can be integrated onto our final 15" robot. With this goal in mind, many specific considerations must be taken into mind that were of lesser importance for the earlier versions of the modules:

1. Width: In order to fit a ball between the two modules, each module must now be less than 3" wide.
2. Height: In order to allow the intakes to fold back into frame, the module must be less than 2.75" tall.
3. Debris protection: With exposed gears, this module needs to be more enclosed to prevent the accumulation of debris inside of the gears.
4. Maintenance: In case something goes wrong with a module, the module must be able to be removed from the frame to make repairs and/or replacement faster and easier.
5. Gear ratio: The speed of the differential should be faster than that of a standard VEX robot, in order to allow it to advantage of the additional motor power and maneuverability that the differential design provides.

The design process began with determining the speeds that were controllable, yet maneuverable. Based on the reference points of the prior two prototypes, we selected a module speed which was around 25% faster than a standard 200 rpm 4" wheel tank drive.



Differential Swerve

	Free Speed (RPM)	Stall Torque (N*m)	Stall Current (Amp)	Free Current (Amp)	Speed Loss Constant	Drivetrain Efficiency
SMART VEX v5	200	2.1	20	0.5	81%	90%

# Gearboxes in Drivetrain	# Motors per Gearbox	Total Weight (lbs)	Weight on Driven Wheels	Wheel Dia. (in)	Wheel Coeff
3	2	12	100%	2.0625	1.1

	Driving Gear	Driven Gear	Drivetrain Free-Speed	Drivetrain Adjusted Speed	"Pushing" Current Draw per Motor
>35	33	14	4.31 ft/s	3.49 ft/s	6.84 Amps
	10	47	0.42 : 1	<-- Overall Gear Ratio	
	43	9			
	1	1			

Rotation Ratio	100.303951	rpm

Figure 59: A variant of JVN's mechanical design calculator adapted for VEX V5 and differential swerve

The remaining constraints drove the remainder of the design and CAD process [see appendix]. One other noteworthy element of the design was the switch to fully integrated motor mounts; these mounts leveraged the advantages of 3D printing and proved critical to fitting everything within the tight space requirements given. We also decided to switch to a much more enclosed design to help keep debris from entering the module. The module also attached in a way that gives it the ability to drop out of the frame if needed to allow for quick replacements or to work on the module independently of the robot.

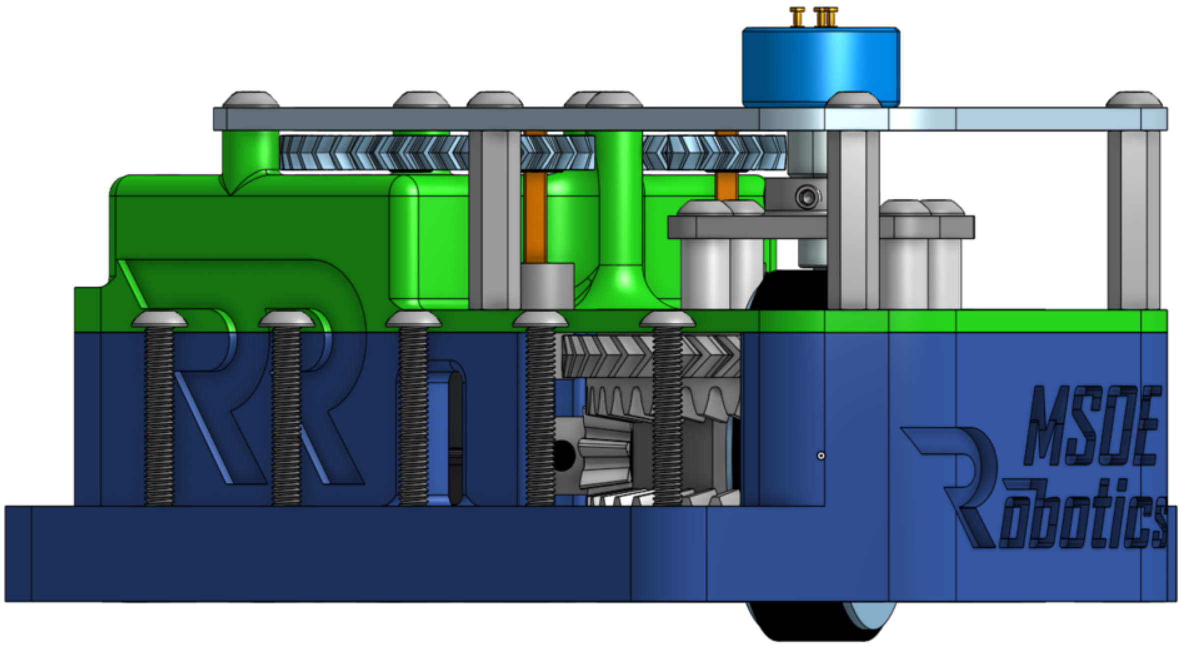


Figure 60: V3 differential swerve CAD

As a result of the changes implemented in this version, these versions of the modules took much longer to print than the other versions but were also much more refined.

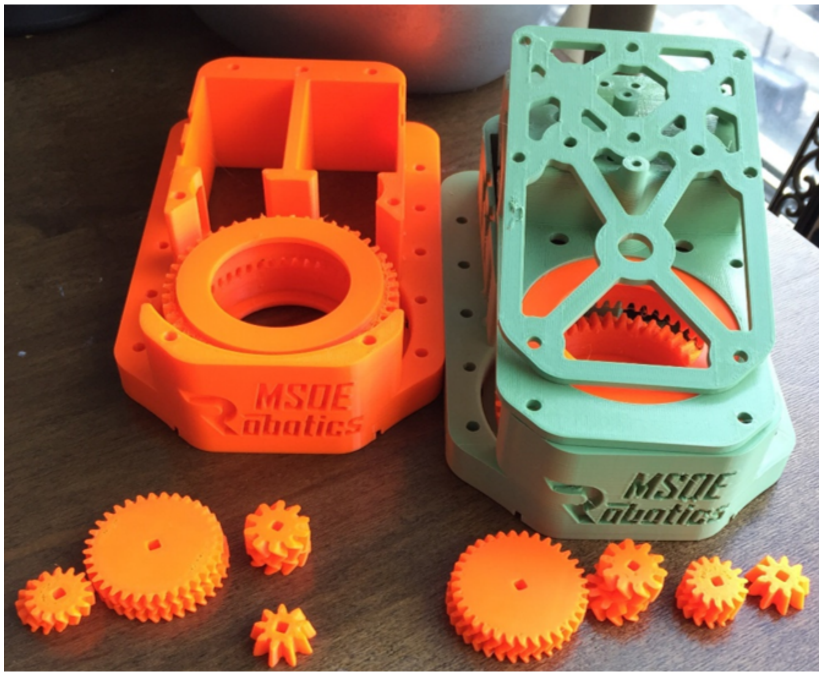


Figure 61: Parts for nearly 2x V3 modules

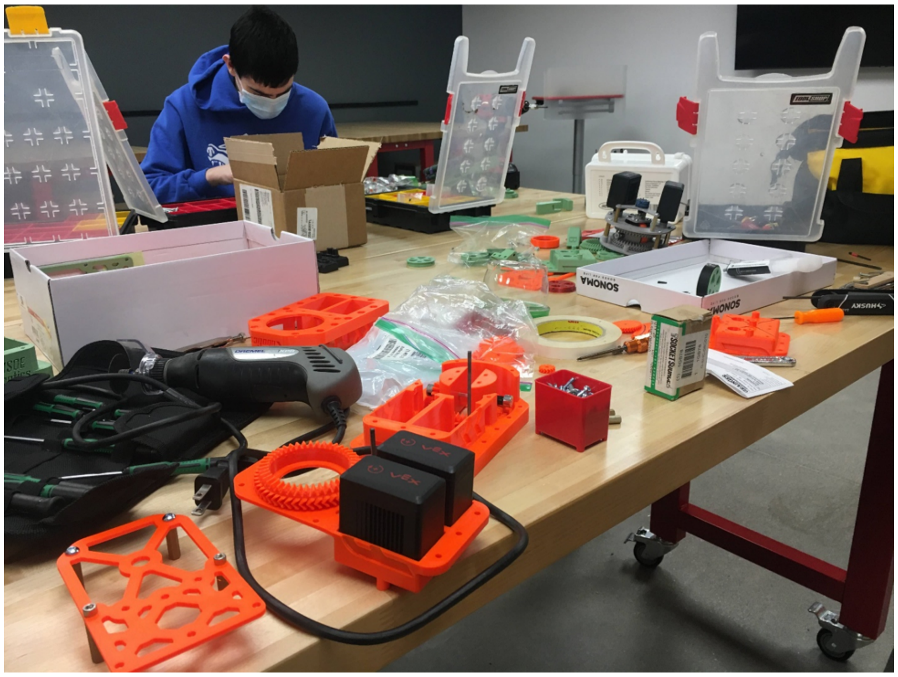


Figure 62: V3 differential swerve assembly process

For this version, we also decided to use 50A neoprene rubber as a tread, JB welded to the 3D printed hub to resolve the issue of the coefficient of friction between the wheel and the foam tiles not being adequate. We found that the 50A neoprene was more grippy than traditional VEX wheels and allowed us to design to the exact size needed.

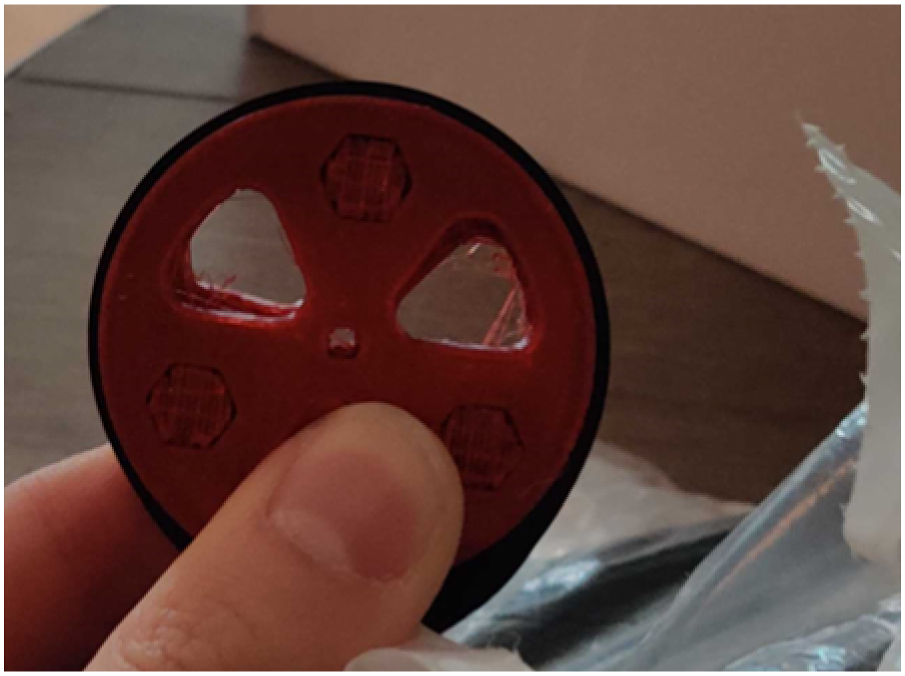


Figure 63: 3D printed wheel with neoprene tread

We also decided to utilize the modules cases to interface with the lowest ring on the tower structures, which would enable the robot to center itself on the goal without requiring intervention.

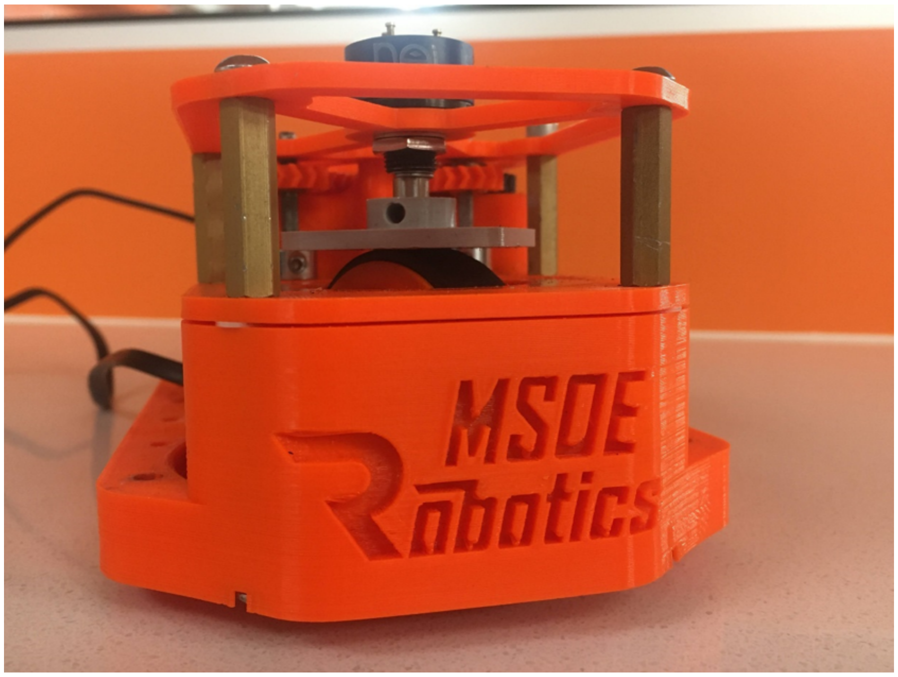


Figure 64: V3 differential swerve module fully assembled

After fully assembling the first module, we tested it in real life by attaching the module to the drivetrain and driving it manually around the field. Because we had yet to assemble the other modules yet, this test setup featured only a single module, and omni wheels were used to support the rest of the frame.

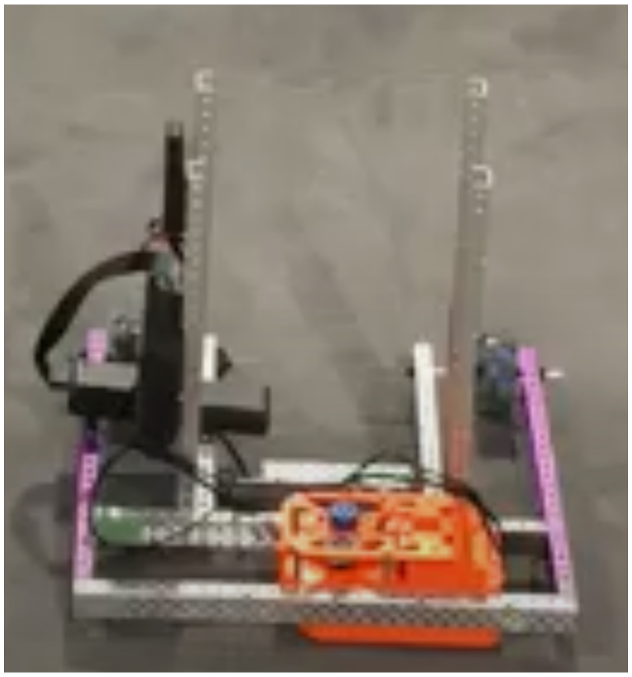


Figure 65: Single V3 swerve module attached to chassis for testing

Although it was difficult to maneuver this version due to the lack of programming integration, it was extremely exciting because it proved that differential swerve was capable of handling the forces outputted by the weight of the chassis and the power of the motors.

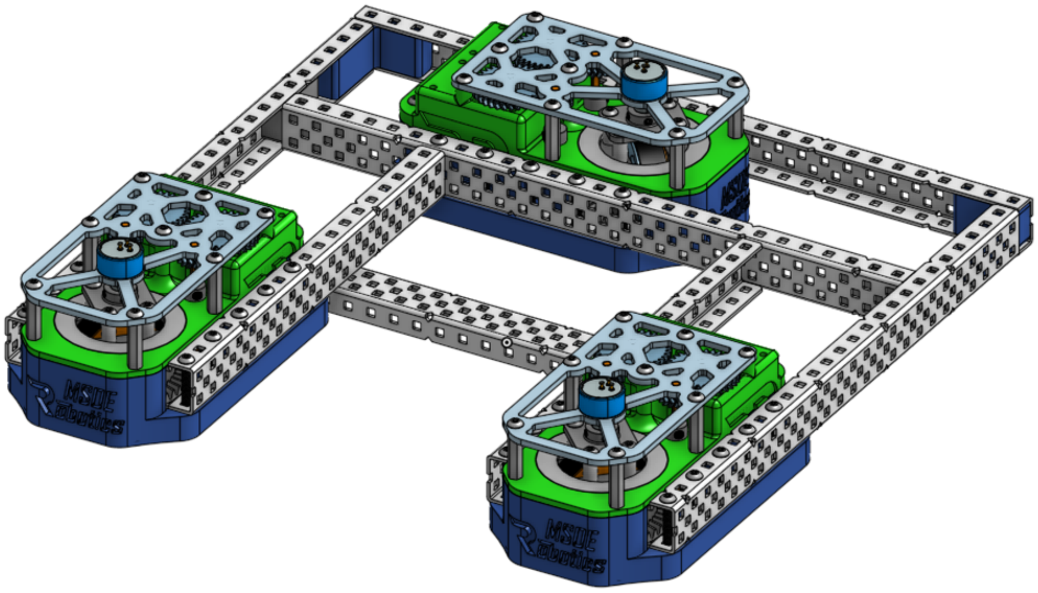


Figure 66: CAD of v3 modules integrated into drivetrain

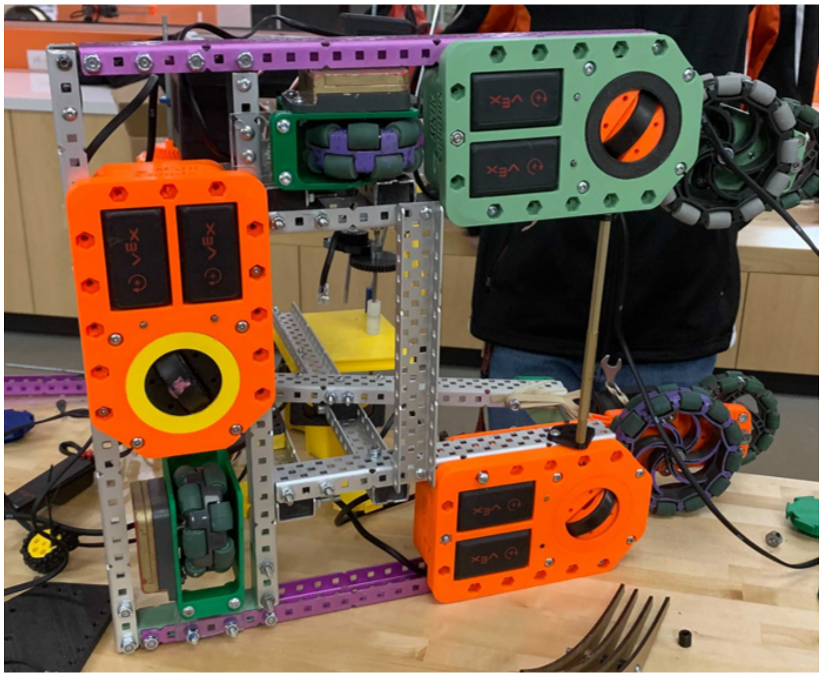


Figure 67: V3 swerve modules installed on frame

4.4 Progression of Swerve Modules

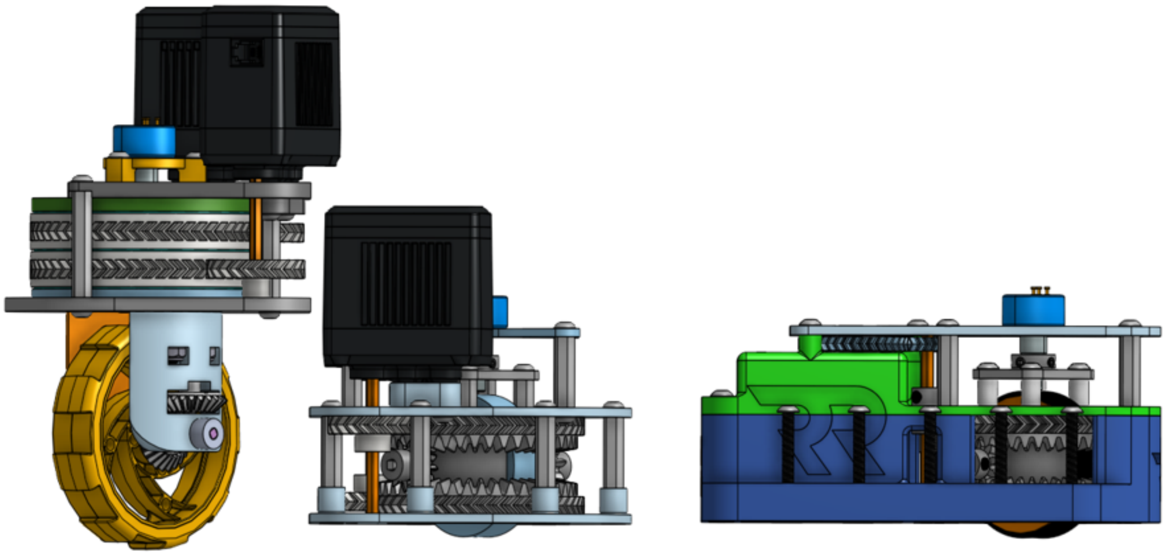


Figure 68: V1, V2, and V3 differential swerve size comparison

4.5 V4 Design Cycle

Problem Definition

While the V3 modules represented a major improvement over the V2 modules, there are still a lot of friction problems in the system. To reduce the friction in the system, we have been filing and sanding down each of the modules. While this works for one module, with three modules this is becoming a bit of a pain and it is clear that something will need to be done to improve the friction.

Solution

To do this, we went back to the drawing board on the modules. Playing around with the V3 modules, we found that the modules would be fine with no load, but as soon as they were put on the robot and the full weight of the robot was applied to the modules, the friction would increase drastically. Comparing all of the modules we have assembled up to this point, the most friction-less by far has been the V1 modules with ball bearings. With this in mind, we came up with a rough concept for a new version of swerve based on ball bearings.

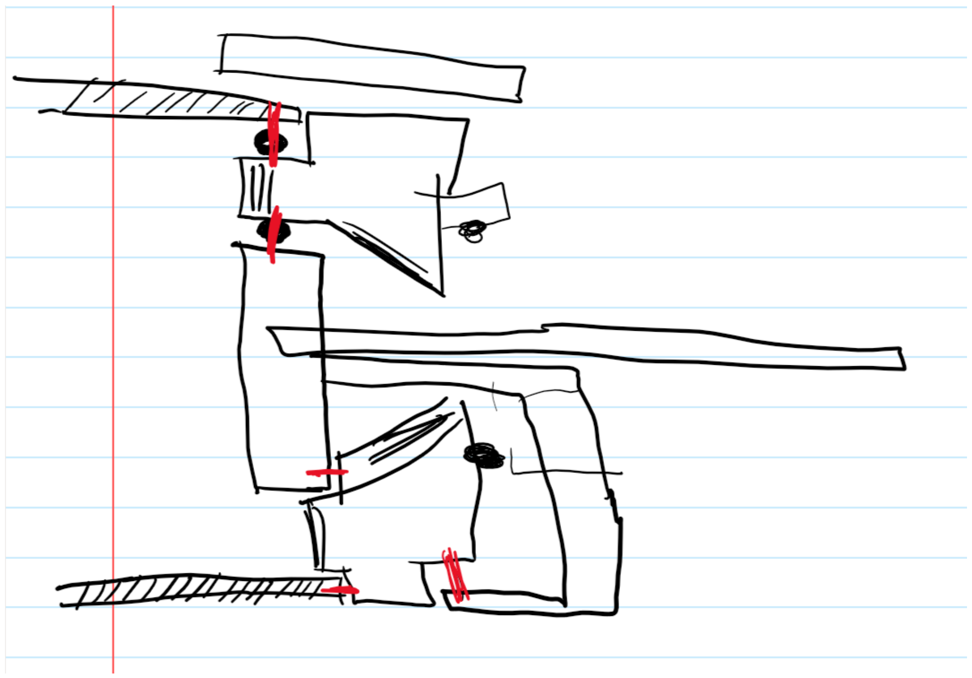


Figure 69: Concept drawing of 3-layer ball bearing design

In Figure 69, a basic drawing of a cross section of the 3 layer ball bearing design can be seen. The red represents the areas where contact would be made between bearing surfaces, with the black circles representing ball bearings. The top and bottom hatched rectangles are the top and bottom plates of the module. Below that is the bevel gear, then the new wheel carrier, and then the bottom bevel gear.

A disadvantage of this approach is that we will have to go back to the ball bearings, which we went away from in the first place due to how painful it was to put them in the ball bearing grooves, literally requiring tweezers to move the ball bearings. We will try to use bigger ball bearings this time around, although this may not be possible due to size constraints.

From there, we went on and modified the V3 CAD to incorporate ball bearings.

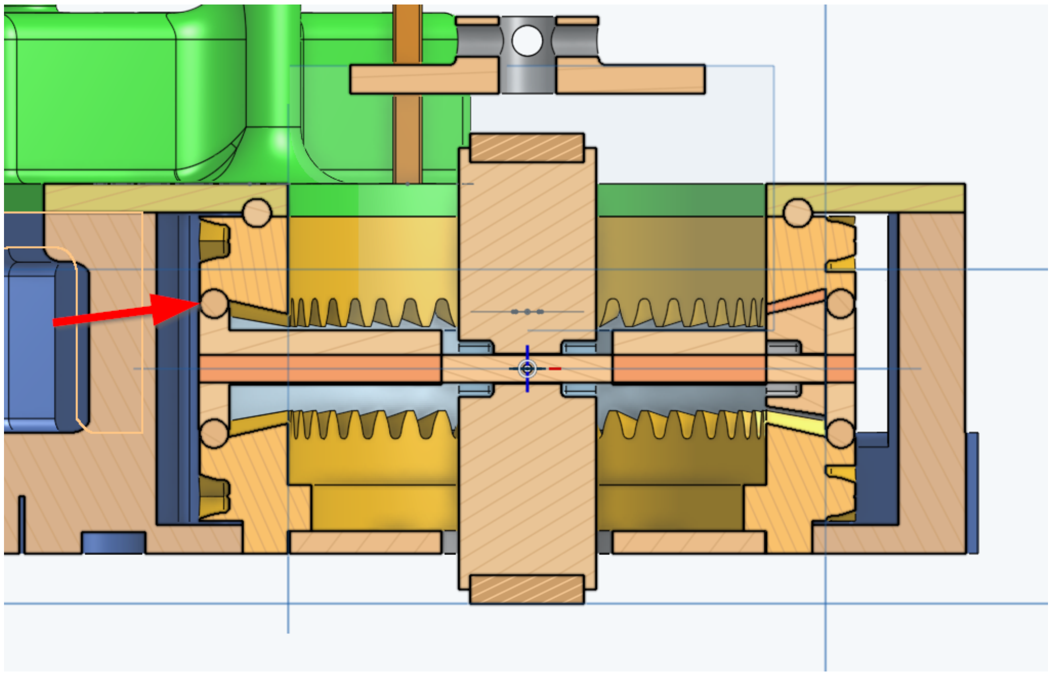


Figure 70: Ball bearings in V4 module CAD

As can be seen in Figure 70, it is extremely tight getting the ball bearings in the CAD. Unfortunately, we had to go with the 1/8" ball bearings as before, but we have a few ideas on how the assembly could be improved with a funnel for the bearings.

4.6 V4.1 Drivetrain Design Cycle

Problem Definition

On Sunday, we preformed our first official driver skills run, scoring 56 points. The big finding from Sunday is that the modules do not have enough power to move the robot for more than 30 or so seconds before the motors overheat.

Things we noticed:

1. The programming is not perfect, and the PID's are not quite getting all the modules in the same direction, leading to the modules fighting each other.

2. The gear ratios might be too high. We aimed to be really fast, but maybe we went a bit too far. We might want to look into a speed lower than our initial speed, but higher than our second speed we switched to at our 1/31/2021 meeting as a temporary solution.

3. There might be too much friction in the system. We think there is a lot of friction where the wheel shaft goes through the 3D printed part.

Solution

Looking at all the places where the problem might be coming from and how poorly the current design preformed in our run, we decided that we should switch to a tank drive temporarily. We still plan on working on the problems listed above, but at the current moment, with only two weeks left in the league, we really want to be able to put up respectable scores and prove out the rest of the robot.

The good news is this should be an easy change to make as the entire super structure of the robot is only mounted to the chassis by 8 bolts.

5 V5 Drivetrain Design Cycle

The goal of this design cycle was to quickly develop a tank drivetrain which would fit onto the current robot architecture and serve as a replacement for the differential swerves on the current robot.

It's important to note that we decided to forgo the traditional process of CADding a design before building it onto the robot,

6 V6 Drivetrain Design Cycle

V5 tank drivetrain worked well on current robot but we want faster speed for V2 robot.

V5 Cons and Pros

Cons:

1. gear was slipping even we use 3d printed axle holder
2. super structure was used only 1*2*1 C-Channel so unstable (bending with mass)
3. 1:1 gear ratio with 4 motor(slow)
4. 2 out of 4 omni wheel axle holder was plastic (delay of driving and programming)

Pros:

1. Simple to build (less structures assembled; light enough)
2. motor and wheels are directly connected (high energy transfer efficiency)
3. 1:1 gear ratio with 4 motor(simpler than V4)
4. chained front and back wheel (low friction than gearing)

Prototype V6

For V6 Drivetrain, we want to keep advantages and remove the disadvantages plus faster speed.

1. 3.75inch omni wheel 6 motor drivetrain with gear ratio (3:4 or 2:3 ideally)
2. use only sprocket or use gear if axle is fully fixed to the position
3. keep 1*2*1 C-Channel for keeping the weight lighter
4. try to not place the motor to the front and middle part of robot for intake
5. 3d printed customize gear input sprocket

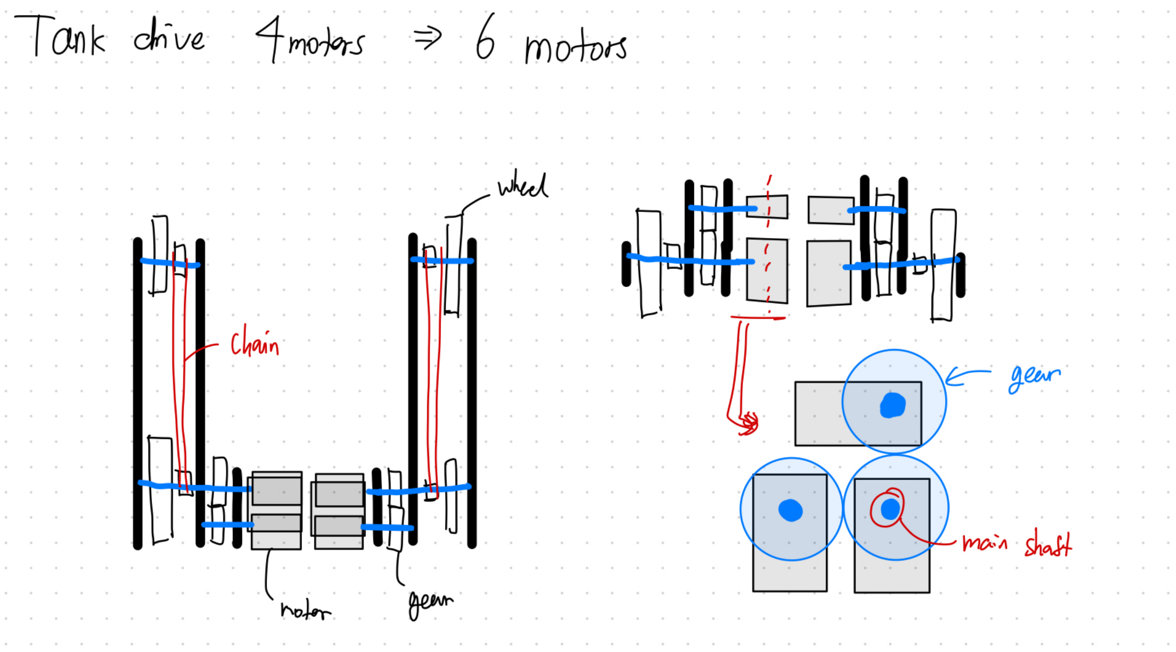


Figure 71: V6 prototype drawing option 1

71 -> 1 to 1 gear ration with directly connected and easily assemble, but speed is not faster. (easy to build and no change of drivetrain width)

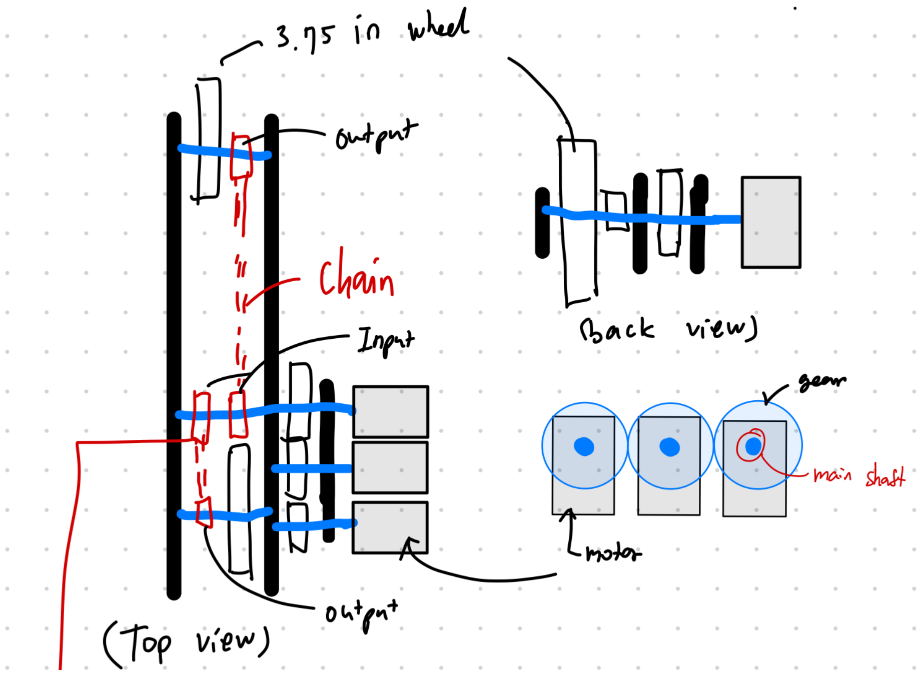


Figure 72: V6 prototype drawing option 2

72 -> There are no enough space to put quad encoder so location of motor should be placed behind. Also there are a possibility that the motor locate where ball going through(block the way). Drivetrain width will be narrower(more space in the middle)

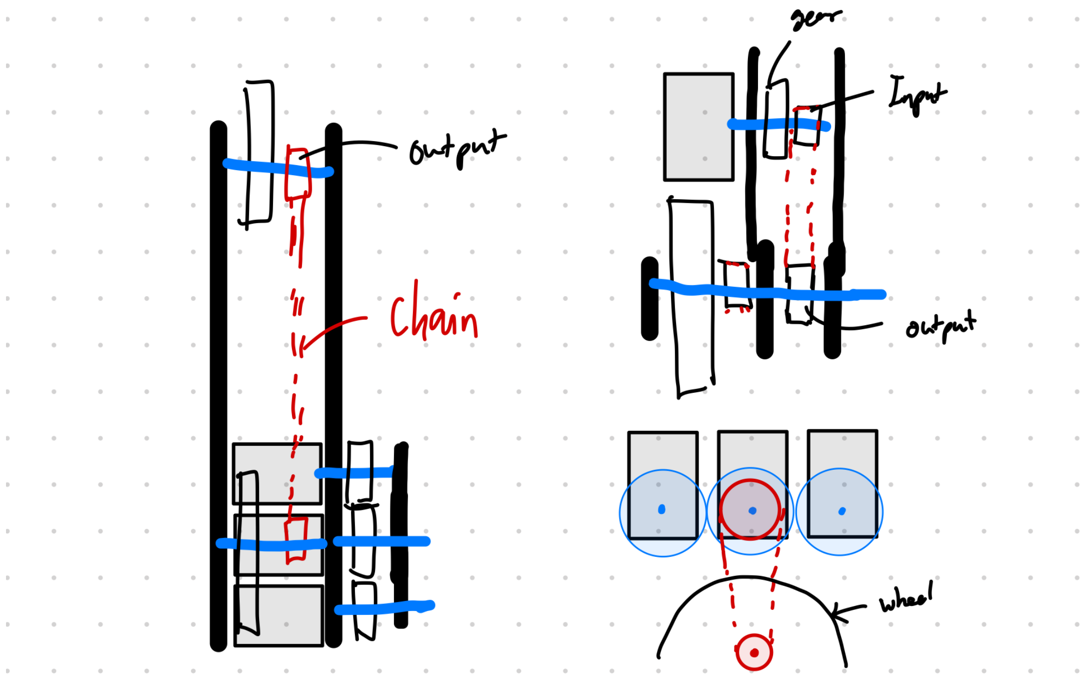


Figure 73: V6 prototype drawing option 3

73 -> There are enough space to put quad encoder so location of motor. complicate building but able to locate quad encoder and do not interfere the balls and faster speed. still first front motor takes a lot of space in the middle so it should be improved more. Width can be changed but optional depend on spacing gears

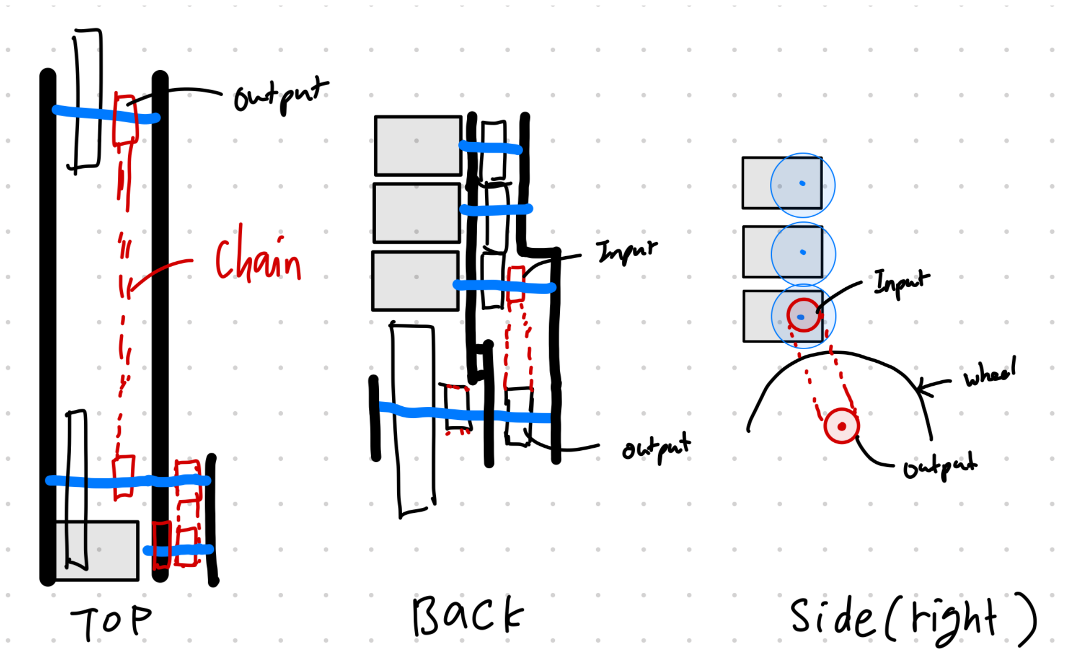


Figure 74: V6 prototype drawing option 4

74 -> Upgraded version of V6 option 3 (relocate motor to the back). Width can be changed but optional depend on spacing gears

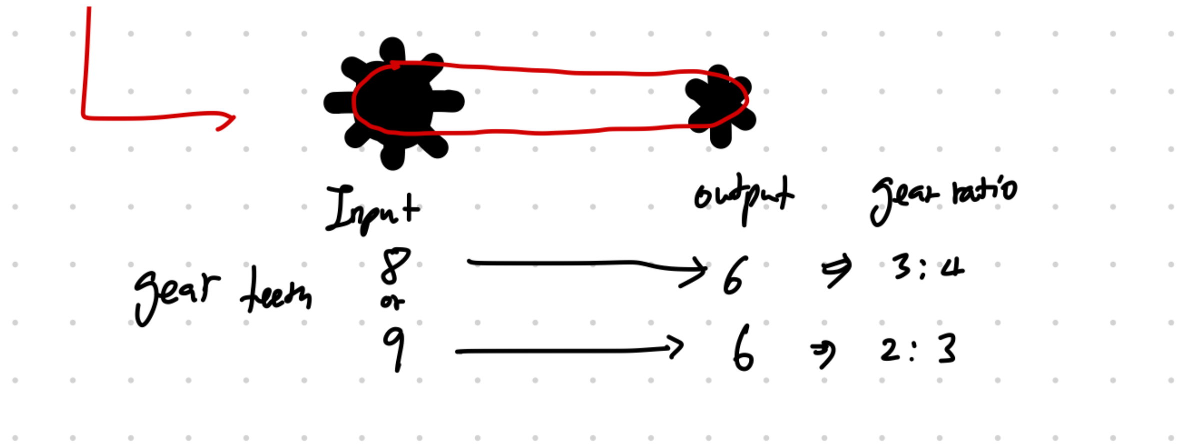


Figure 75: V6 prototype gear ratio

gear ratio can be changed by vary the input sprocket teeth depend how fast and how much power we want.

V5 gear ratio => 200Rpm 4in Diameter = 200X4Xpi = 800pi (2513.3)in/m = 2.38 mi/h

V6 gear 4:3 gear ratio => 200Rpm 3.75in Diameter 3:4 = 200X3.75XpiX4/3 = 1000pi (3141.6) in/m = 2.975 mi/h (25 percent faster than V5)

V6 gear 3:2 gear ratio => 200Rpm 3.75in Diameter 2:3 = 200X3.75XpiX3/2 = 1125pi (3534.3) in/m = 3.347 mi/h (40 percent faster than V5)

6.1 V7 Drivetrain Design

Goals:

- Holonomic
- Faster than previous drivetrain
- Better place for follower odometry wheels

This version of the drive train is intended for the V2 version of the robot, which will essentially be a full redesign of the robot. Thus, we have a a lot of flexibility to design what we want. After reflecting on our V1 robot, we decided that we wanted to proceed with a holonomic drivetrain for V2. A holonomic drivetrain offers several benefits; it's a lightweight, omnidirectonal drivetrain, which is especially key for our programming team.

After setting on our goals, we began designing the drivetrain. Because of the compactness of the frame, we were forced to fit the motor into a 3 wide c channel in order to make everything package nicely. After some tinkering around in CAD, we finished a version we were happy with.

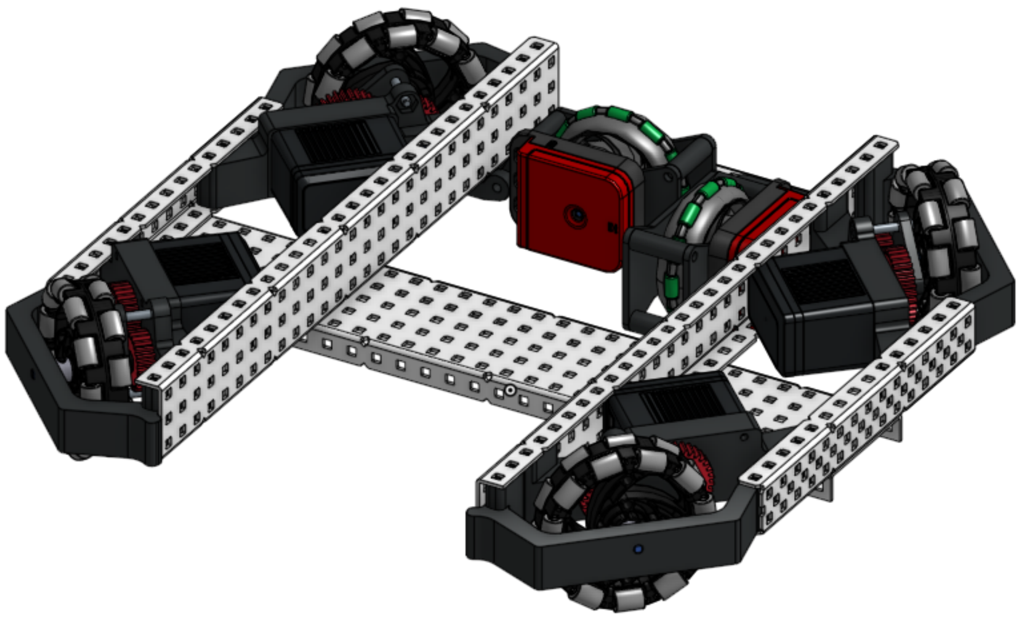


Figure 76: V2 Holonomic Drive CAD

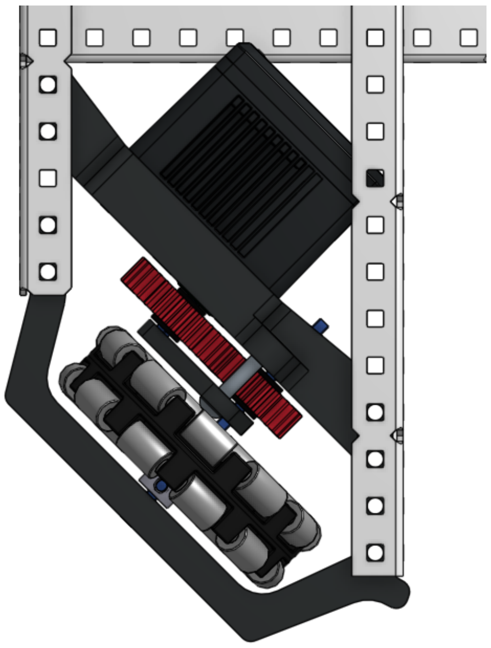


Figure 77: V2 Holonomic Drive CAD Close Up

This version focused heavily on ease of assembly and mechanical robustness, and leverages 3D printed parts to fit everything within the tight space requirements. Notably, this design uses all COTs shafts, and also features an updated version of the odometery wheels, which are smaller and lighter than the previous ones used on V1.

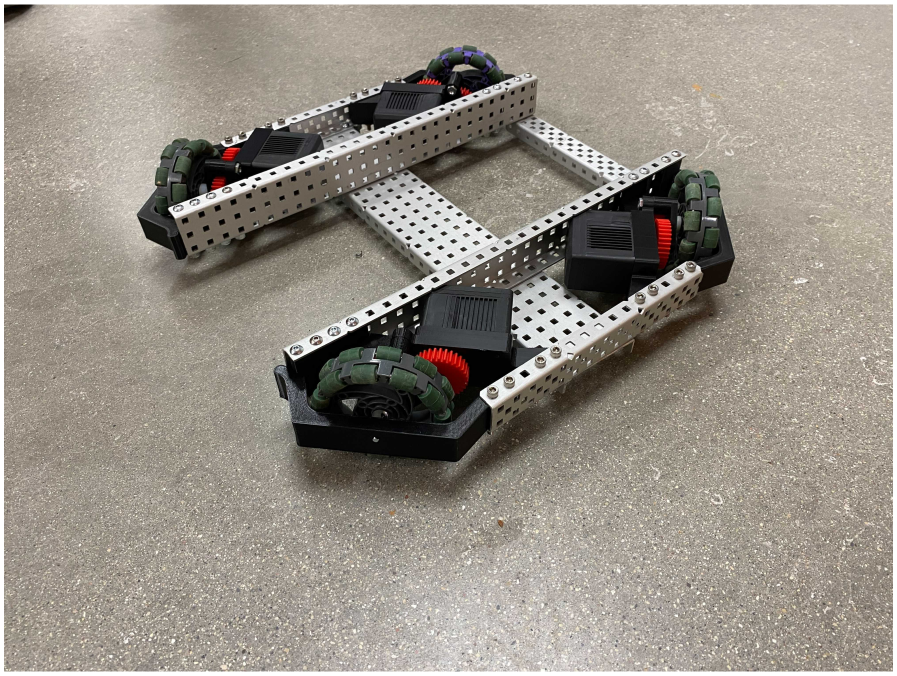


Figure 78: V2 Holonomic Drive In Real Life

In testing of the holonomic drivetrain, we found that the motors quickly overheated within the span of just over a match. With the event we are in being scheduled with 6 matches within a 20 minute period, this would be unsustainable. Furthermore, the robot also was not making it to it's top speed even cooled down and on a full charge. As a result of this and not being able to easily significantly reduce robot weight, the design was modified to include another motor for each of the four modules to spread out the load and reduce the thermal throttling.

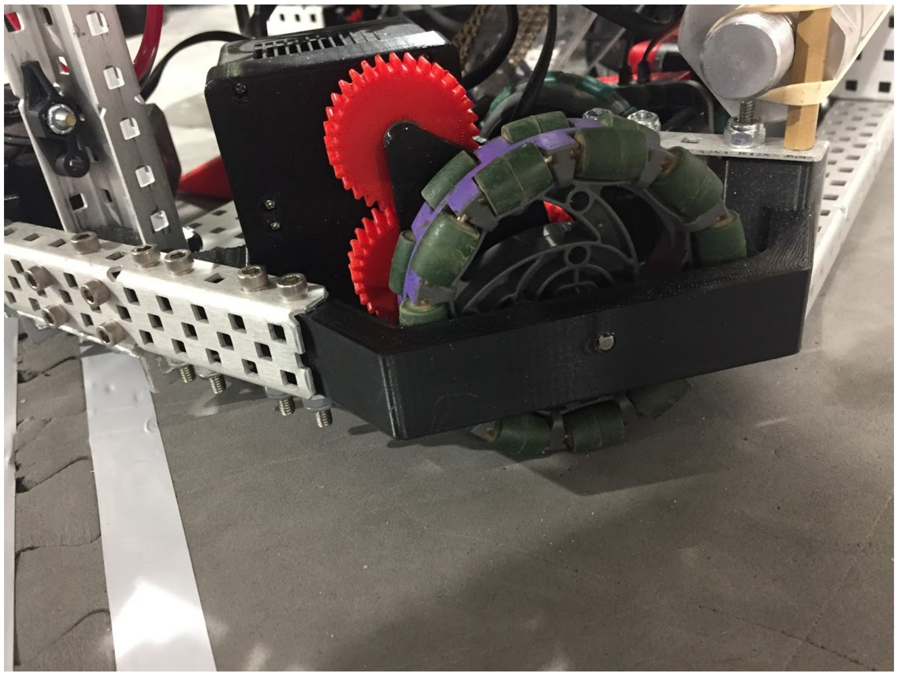


Figure 79: Two motor X-drive module

7 Intake

7.1 V1 Intake Design Cycle

The goal of this design cycle is to create an intake suitable for our 15” robot that will be able to intake balls into the robot. The intake should be able to effectively pull balls from the floor, center, and corner goals, and it should have enough power to force balls out of goals, even when the goal is too small and/or the goal is full of other balls. Furthermore, the intake should be able to cleanly fold into the robot at the start of every match without taking too much room away from other mechanisms.

We started by brainstorming several different types of possible intake designs.

1. A horizontal beater bar intake

   This intake design uses a horizontal bar, as opposed to vertical rollers, to intake the ball. This has the advantage of always pulling balls into the robot, and never pushing them away, making intaking simpler.

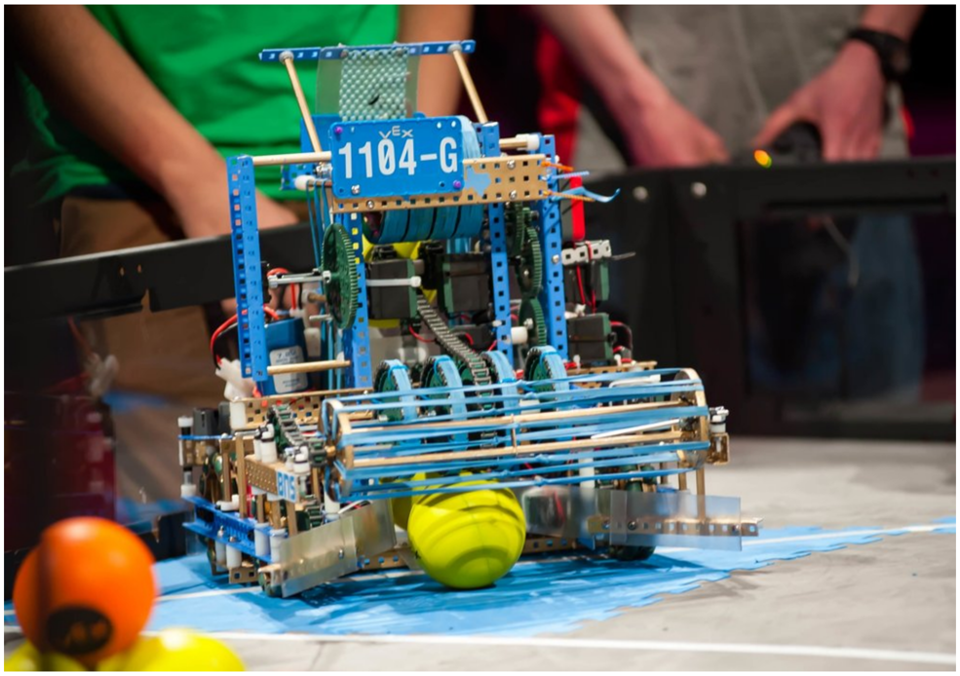


Figure 80: An example of a horizontal beater bar intake from Brampton Robotics

2. A conveyor belt chain intake

   This intake design uses vex conveyor belt chain to pull balls into the robot. The design is relatively straightforward, but we do have concerns about how strong the flaps are.

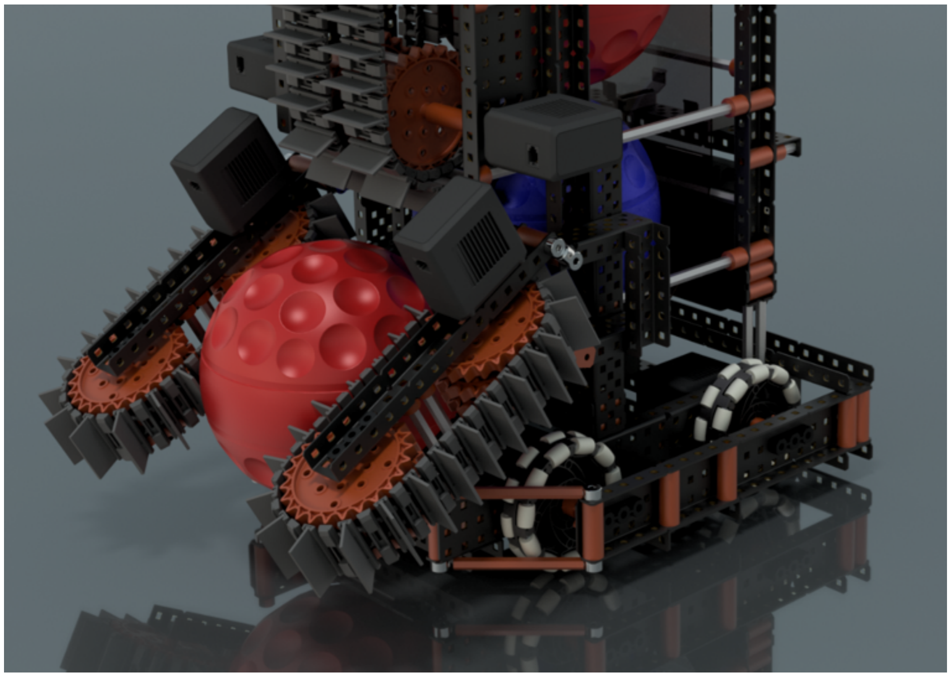


Figure 81: An example of a conveyor belt intake on team 99999V's CAD challenge robot

3. An omni wheeled intake

   This design uses omni wheels to pull balls into the robot. This design seems to be really effective, but the omni wheels are pretty big, and smaller ones are both expensive and less effective. The advantage of using omni wheels is that they allow balls to drop down more seamlessly between the intakes when cycling multiple balls from a goal.

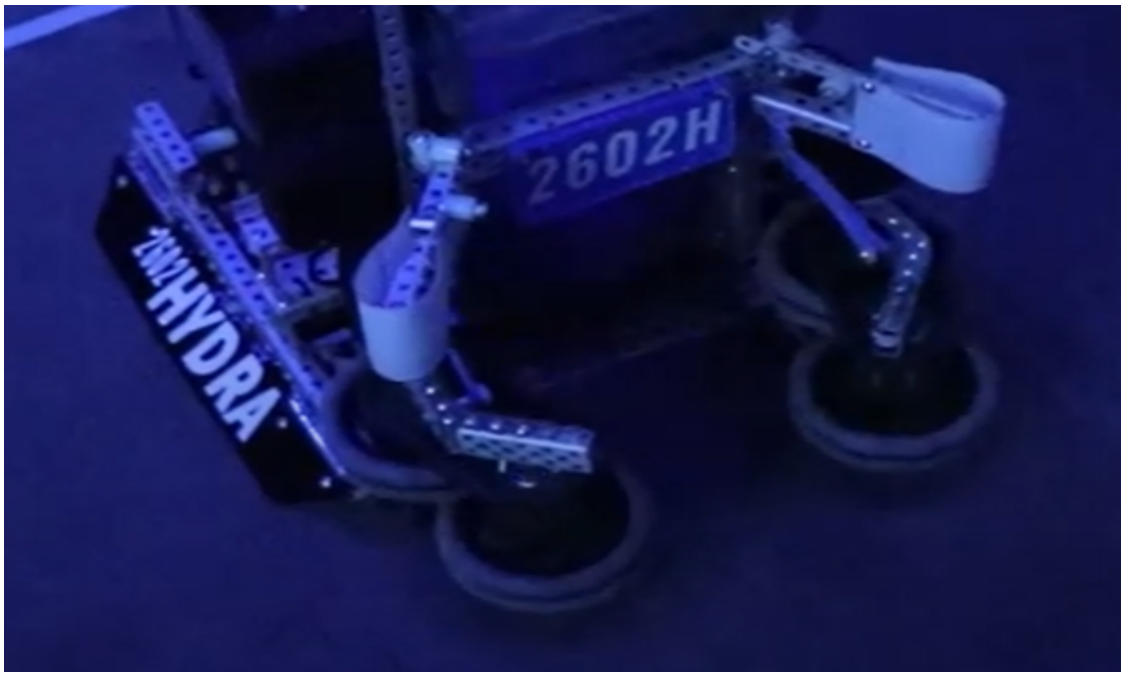


Figure 82: An example of an omni wheeled intake employed by VEX team 2602H

After brainstorming different types of intakes, we created a decision matrix comparing our ideas.

	Floor Effectiveness	Goal Effectiveness	Size	Complexity	Total
Beater Bar	5	1	2	1	8
Conveyor Belt Chain	3	4	5	4	16
Omni Wheels	4	5	1	4	14

A beater bar intake was deemed impractical largely because it was so difficult to adapt to intake from the goal. An omni wheeled intake seemed like the most effective type of intake, but was extremely difficult to package into the robot due to the large amount of space the wheels would require while the robot was in starting configuration. Thus, despite the lesser effectiveness of the conveyor belt chain intake, the simplicity and ease of packaging made it the most practical design.

After settling on a conveyor belt intake, we began working on a 3D printed design for it in CAD. In order to create this version of the intakes, we plan on following our CAD-prototype design cycle, described in section [add section] of the notebook, to design, build, and improve the mechanism. Because the rest of the robot is still in development, it's important to make the intakes as compact as possible. We started our design by making initial sketches examining possible positions of chain intake rollers.

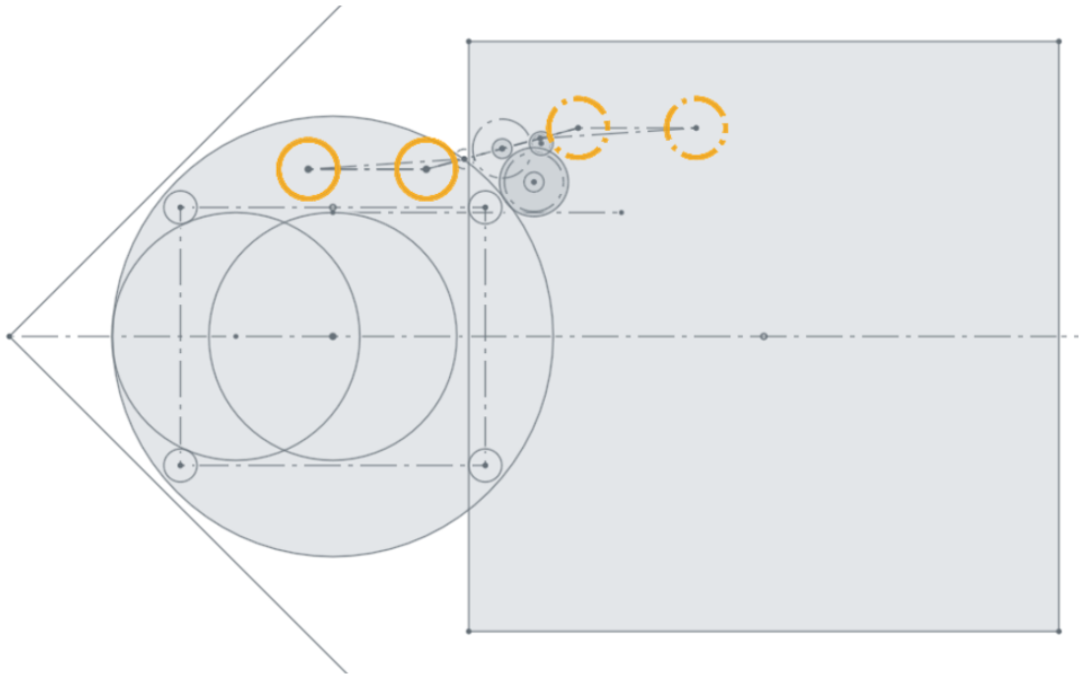


Figure 83: A top down sketch with the intake rollers drawn in their in and out positions

While designing the V1 intake, we also took several other design considerations into mind.

1. Ideally, the intakes needed to be able to retract at least partially during the match, in order to make lining up to intake balls from the floor and from the center goal easier
2. Angling the intakes downward would make pulling balls over the base of each goal easier, especially when the goal was full of balls
3. Making it so we could swap the gear ratios out would be advantageous for testing, as we were uncertain at what speed we wanted the intakes to run

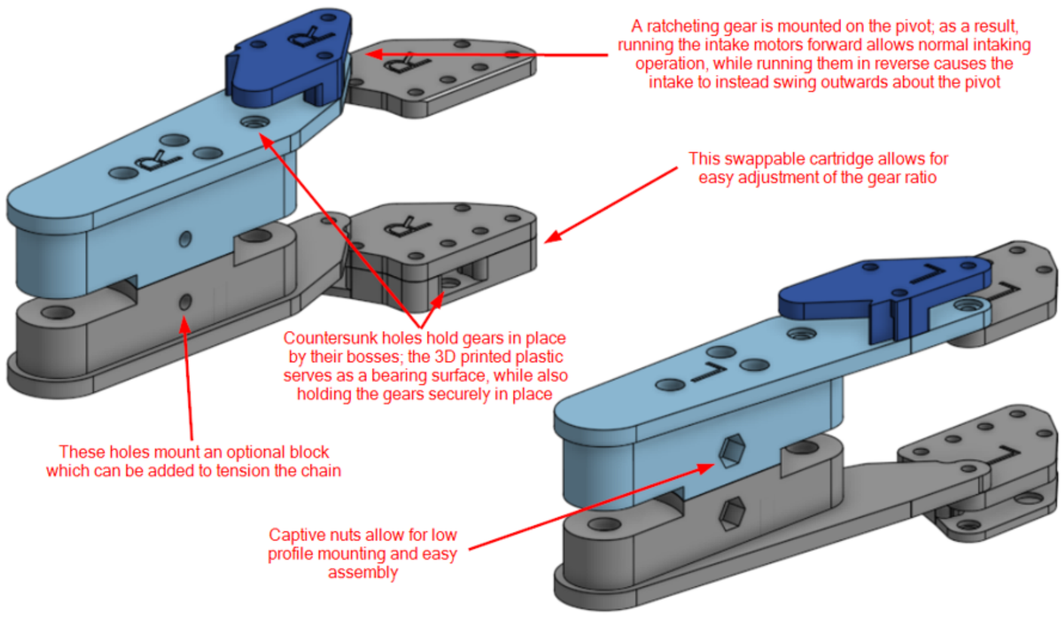

A ratcheting gear is mounted on the pivot; as a result, running the intake motors forward allows normal intaking operation, while running them in reverse causes the intake to instead swing outwards about the pivot
This swappable cartridge allows for easy adjustment of the gear ratio
Countersunk holes hold gears in place by their bosses; the 3D printed plastic serves as a bearing surface, while also holding the gears securely in place
These holes mount an optional block which can be added to tension the chain
Captive nuts allow for low profile mounting and easy assembly

Figure 84: A diagram of the initial intake design

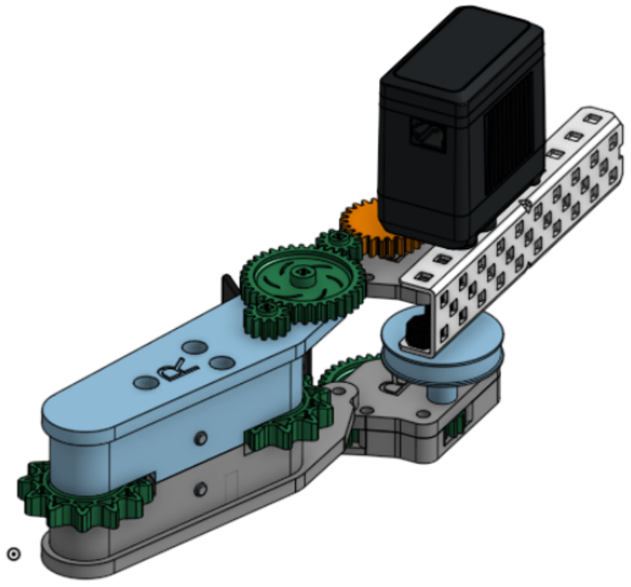


Figure 85: Completed CAD of the V1 right arm of the intake

After designing the intake, we printed the intake and assembled it in real life. We also created a test frame which allowed us to quickly change the angle at which the intake was mounted relative to the ground. We then proceeded to perform a variety of tests evaluating the effectiveness of the intake by running it and attempting to intake balls. Positions tested are as follows:

- A ball resting on the floor
- Intaking balls from a stack of one, two, and three balls in the center goal
- Intaking balls from a stack of one, two, and three balls in a corner goal

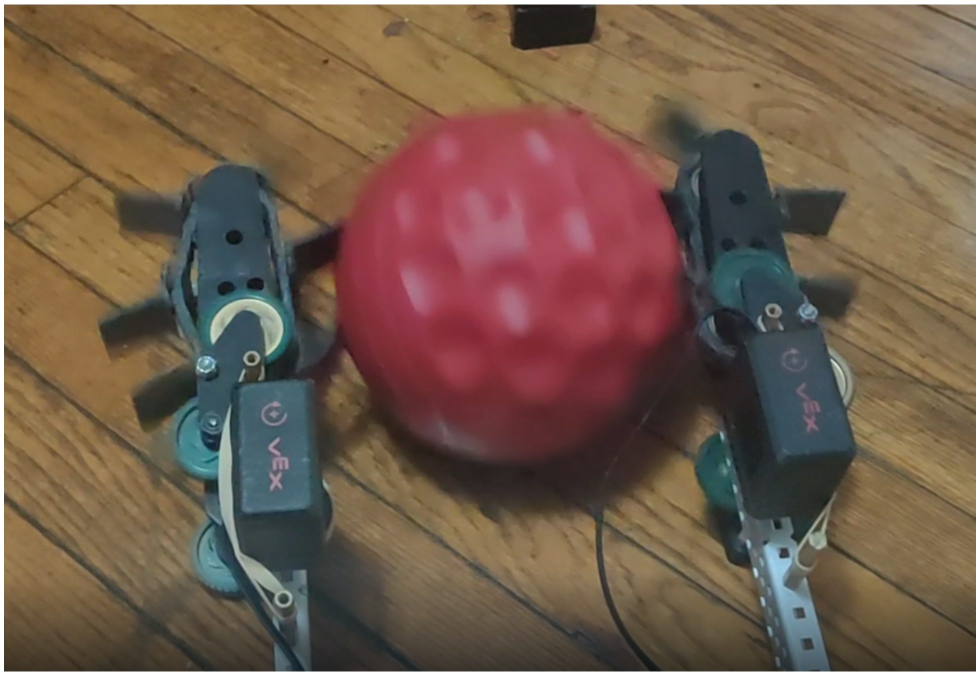


Figure 86: V1 intake testing

While performing these tests, we focused on making qualitative observations regarding the efficacy of the intake. This was done because the quality of an intake is largely subjective, based on the standards of the team that designs it. Furthermore, collecting accurate quantitative data regarding the performance of the intake is extremely difficult since the margins between intakes are so often razor thin.

We did, however, take an additional opportunity to test various angles of the intake, in order to get an idea of how angling the intake would impact performance, and to gain an idea of what an optimal intake might look like. In order to do so, we tested the intake by placing it at various angles relative to the ground and then intaking a ball off of the ground and from a corner goal. We tested angles ranging from parallel to the ground (0 degrees) to 25 degrees relative to ground in 5 degree increments.

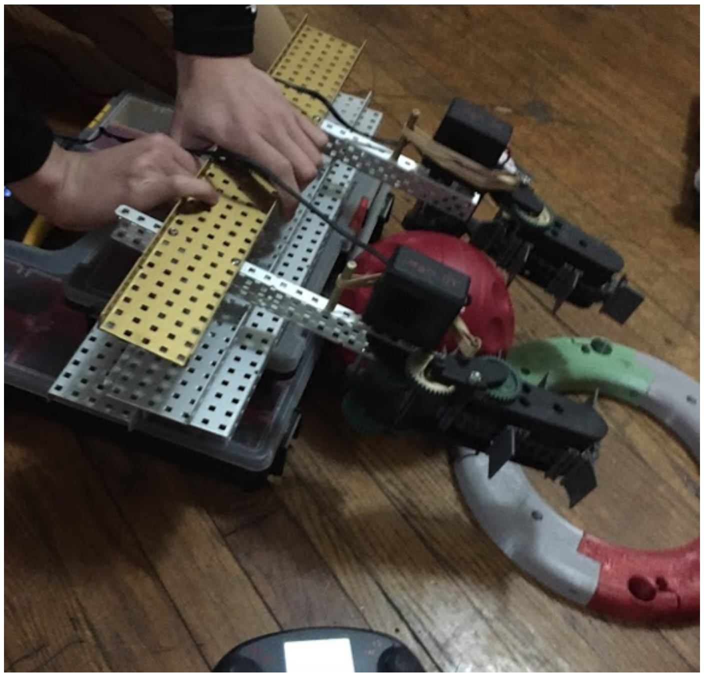


Figure 87: V1 intake testing different intake angles

After performing these tests, we found that the intake performed best when mounted at an angle of about 10 to 15 degrees relative to the ground. The performance of the intake when placed at these angles was essentially identical, but tended to drop off when not at these angles.

Based on our tests, we ultimately decided that we were unhappy with the performance of our design. The fundamental issue with the design was that it would often struggle to impart sufficient force on balls in order to remove them from goals, particularly when the goal had two or three balls in it; the conveyor belt flaps were simply unable to impart sufficient force on the ball to actually drive them up and out of the goal. As such, this design's ability to pull balls from the bottom of goals was significantly diminished, and it often struggled or failed entirely to pull remove balls.

7.2 V2 Intake Design Cycle

The goal of this design cycle is to change the design of the intake to a design which will be much more effective at intaking balls from the center and corner goals, especially goals which are full with three balls.

After designing, building, and testing our first version of the intake, we found that we had overestimated the effectiveness of the conveyor belt chain intake when intaking from a goal. As a result, we updated our original decision matrix to reflect this discovery.

	Floor Effectiveness	Goal Effectiveness	Size	Complexity	Total
Beater Bar	5	1	2	1	8
Conveyor Belt Chain	3	2	5	4	14
Omni Wheels	4	5	1	4	14

Since effectiveness is now a much greater priority for us, we found that the conveyor belt chain intake was unable to attain the results we desired, and that the additional effectiveness of the omni wheeled intake made it worth the additional size and complexity costs. As a result, we decided to design and build an omni wheeled intake, which will be more effective at intaking balls from the goal due to the larger and harder exterior rollers, and which will therefore allow the intake to impart much more force on the balls.

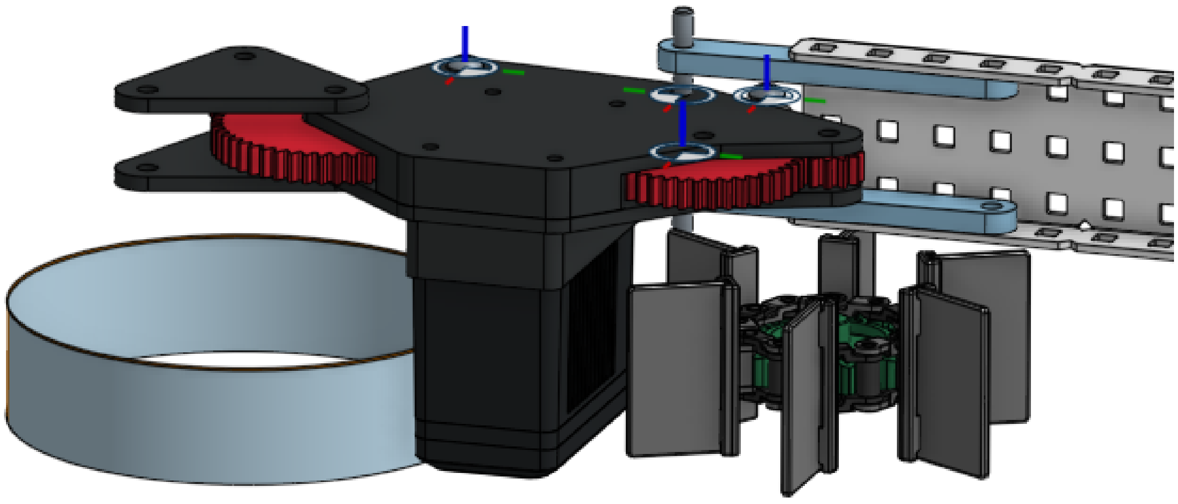


Figure 88: V2.1 Intake

After completing the initial version of the CAD, however, we realized that packaging this design so that it could both fit in frame and deploy to the desired position using a four bar linkage was going to be extremely difficult and require a lot of additional complexity. As a result, we decided to look for a new, simpler approach.

7.3 V3 Intake Design Cycle

Goals

- Get rid of dead zones
- Reduce mass
- Make it so it does not stall when hung up on center goal pipes or corner goal wall
- Simplify design
- Make it easier to deploy and stow
- Use less rubber bands

After setting our goals, we began working on updating the CAD of the V2 intake to reflect the desired changes.

Intake V3.1 Design Review

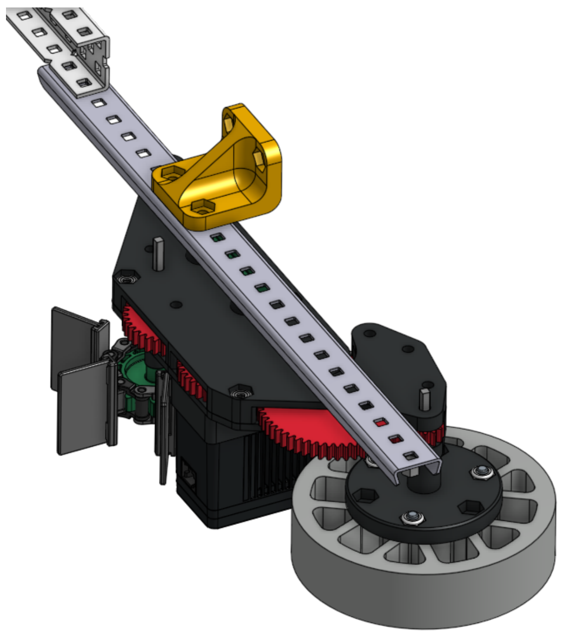
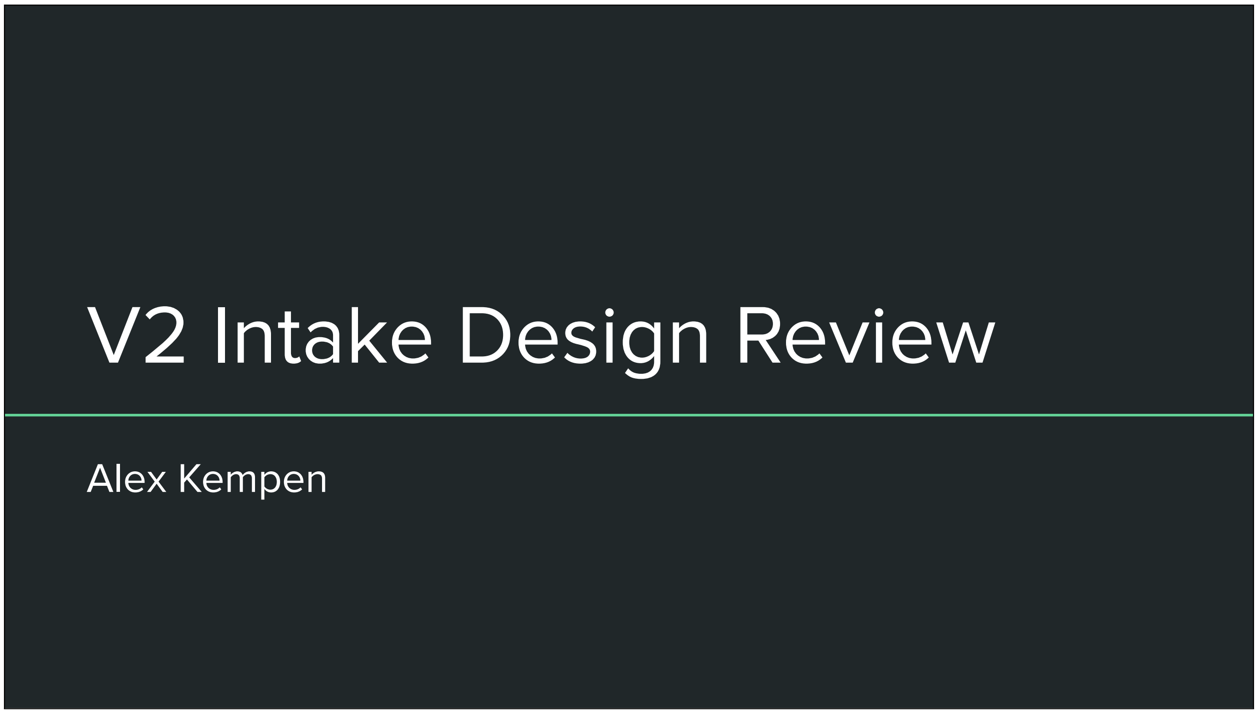
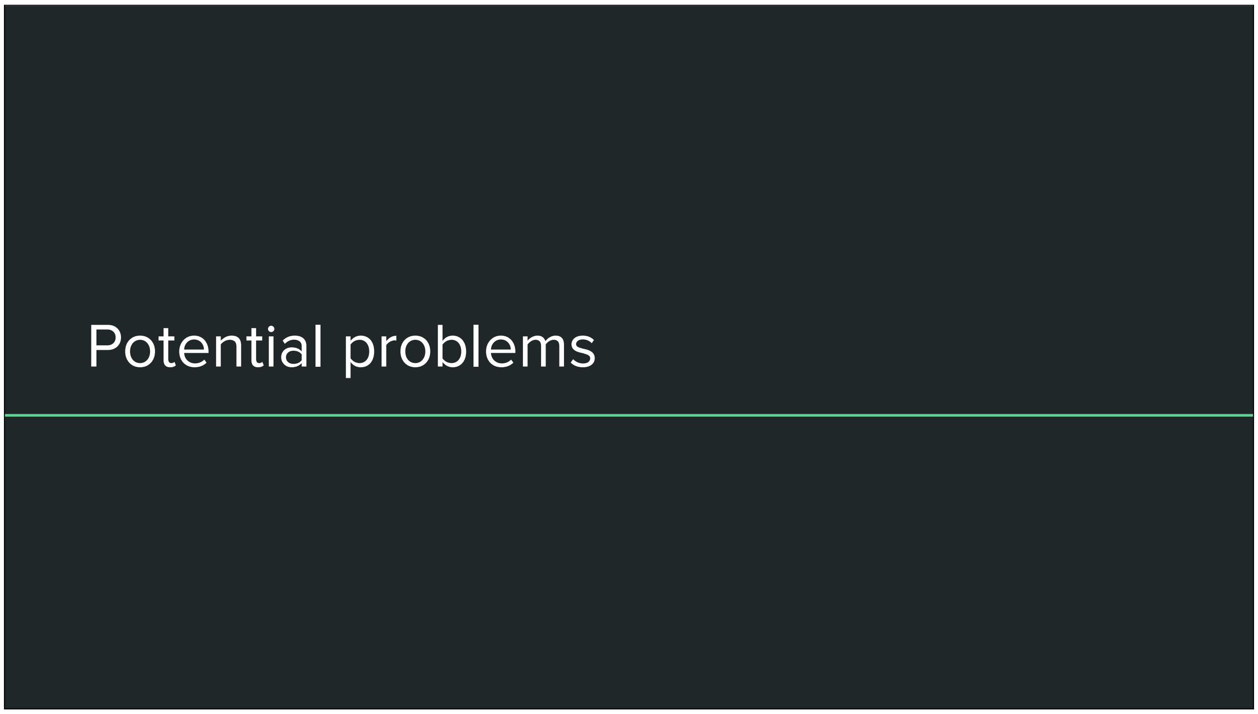


Figure 89: V3.1 Intake

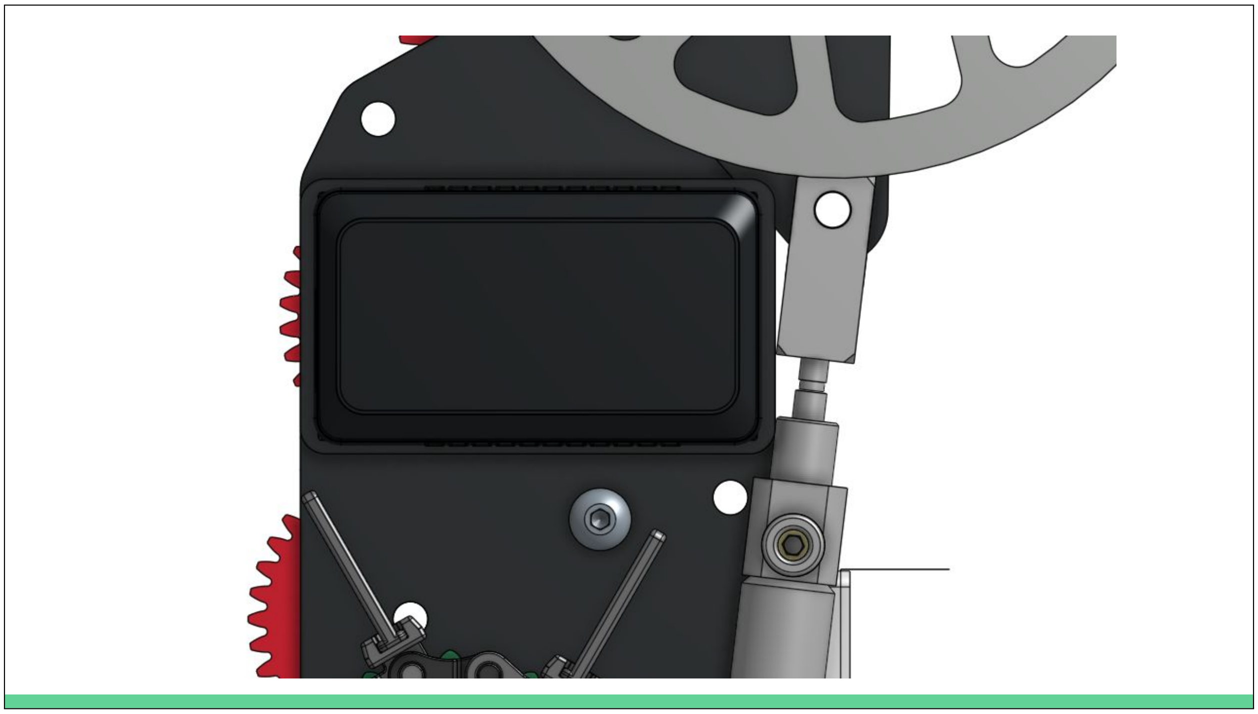


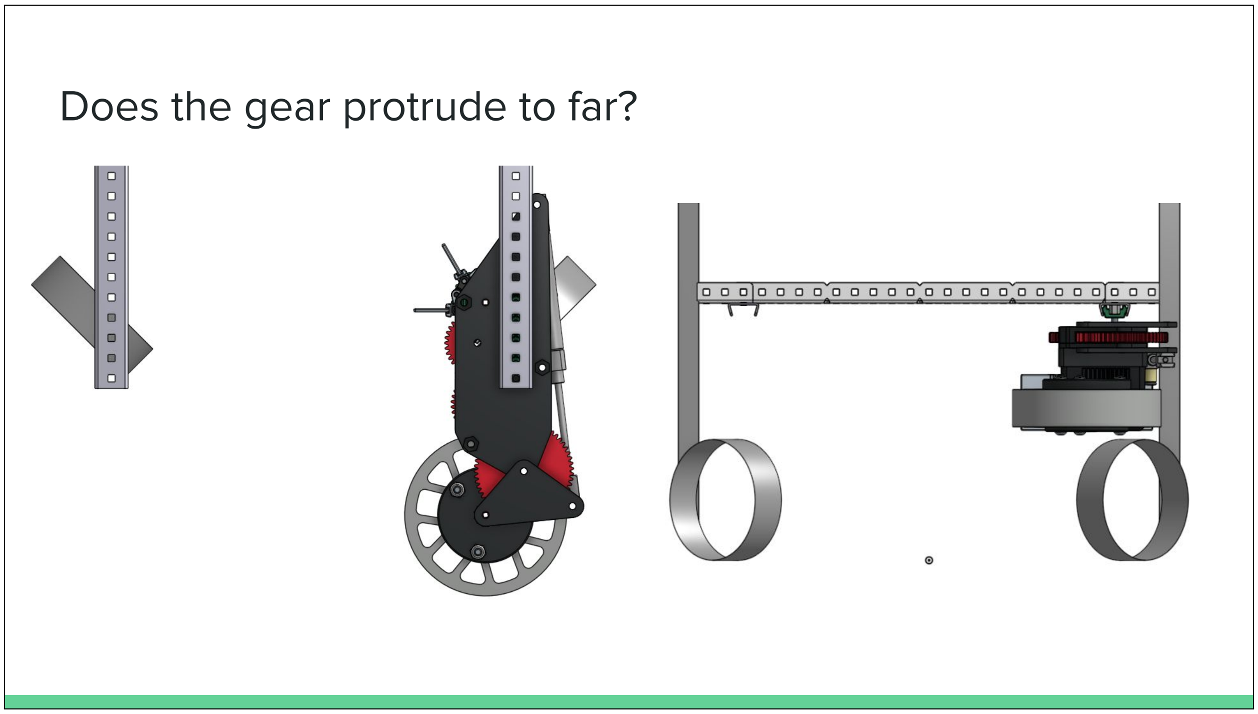
V2 Intake Design Review

Alex Kempen

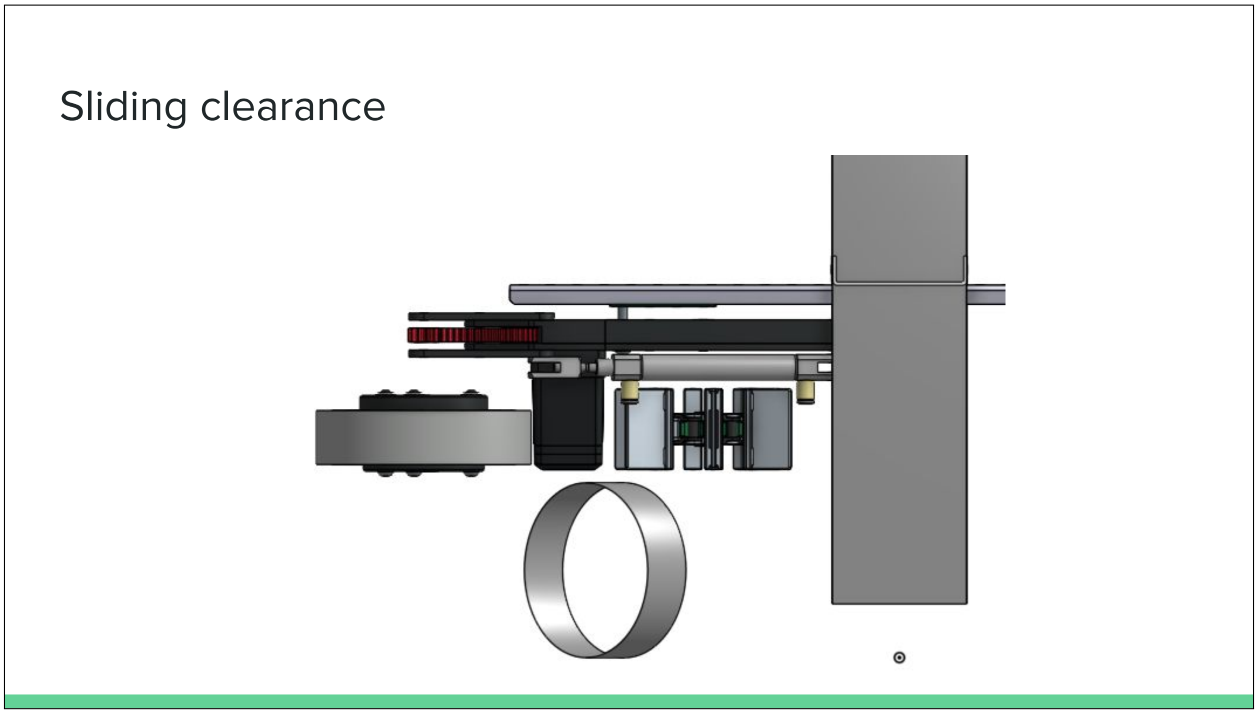


Potential problems

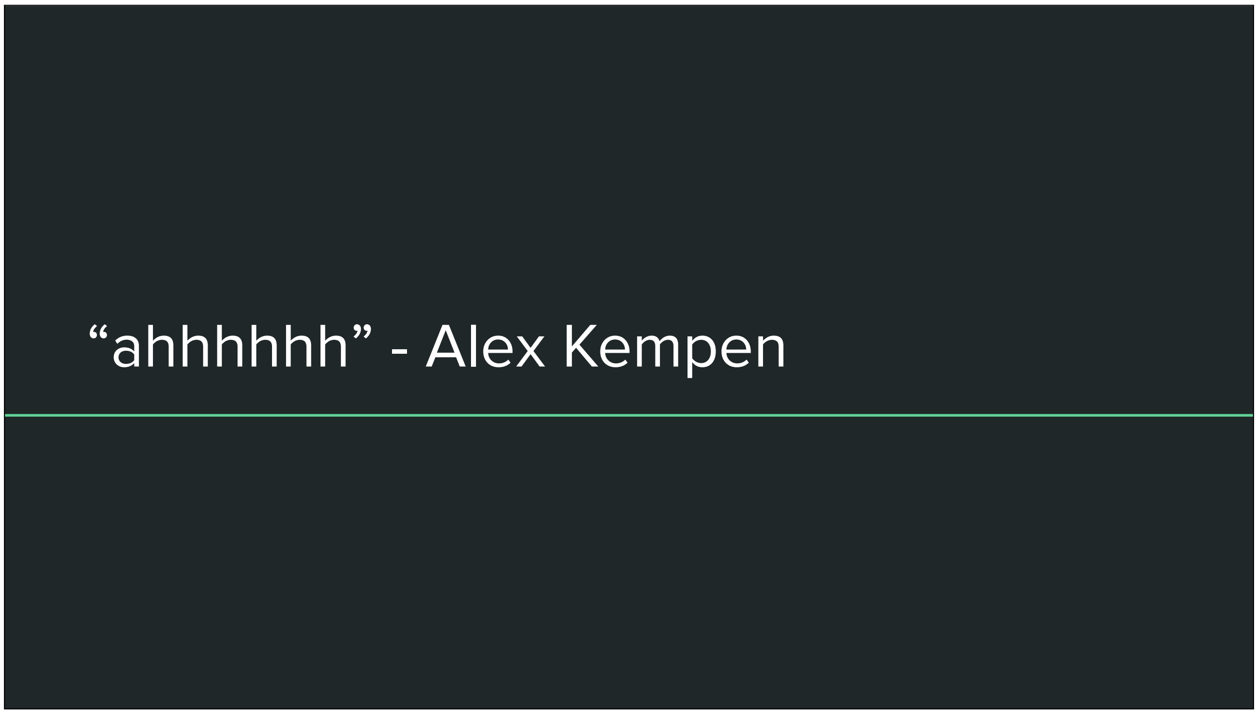




Does the gear protrude to far?



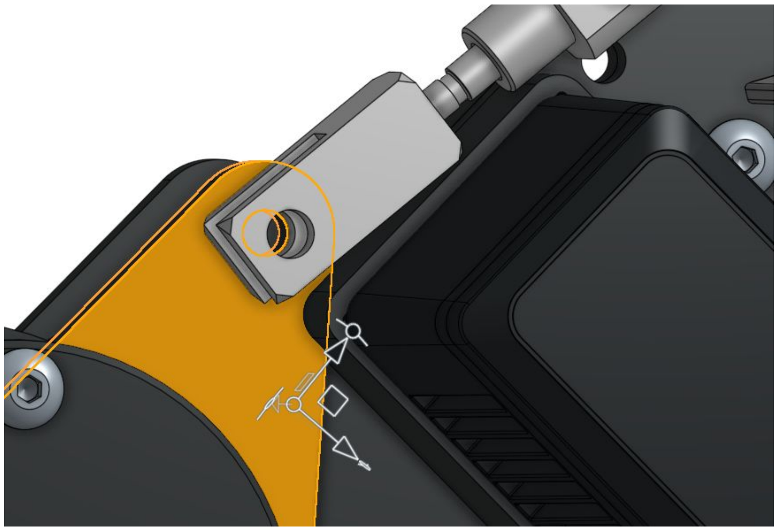
Sliding clearance



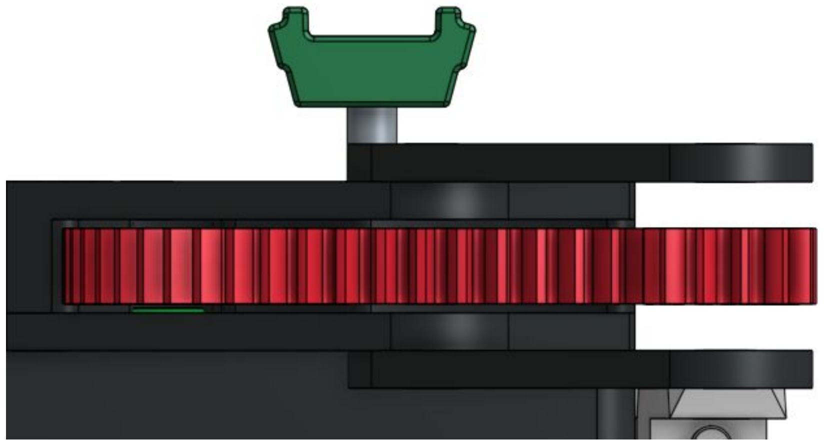
“ahhhhhh” - Alex Kempen

Things to do

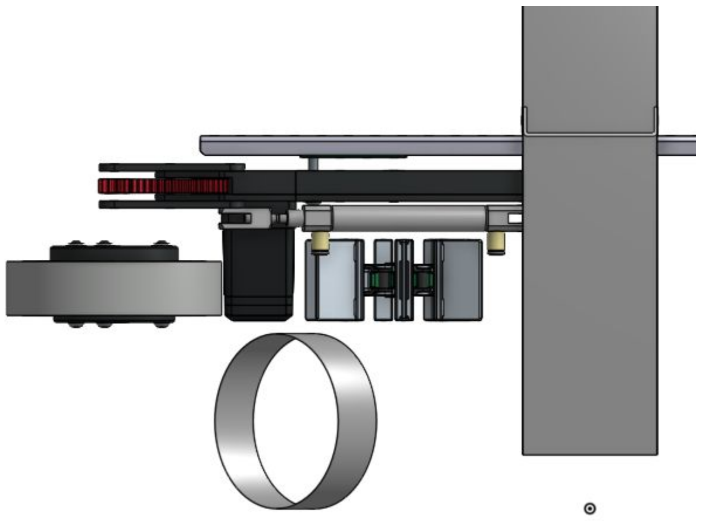
Add cutout to pivot plate

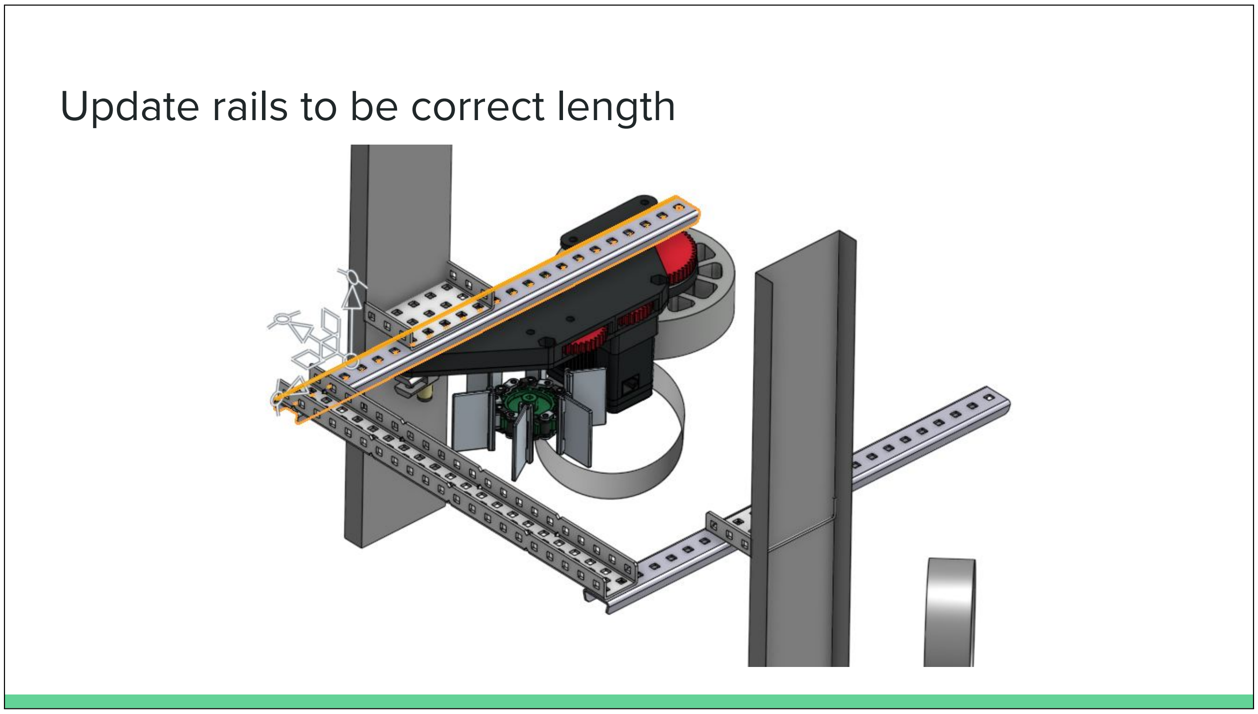


Add filler to pivot plate + mounting spacers

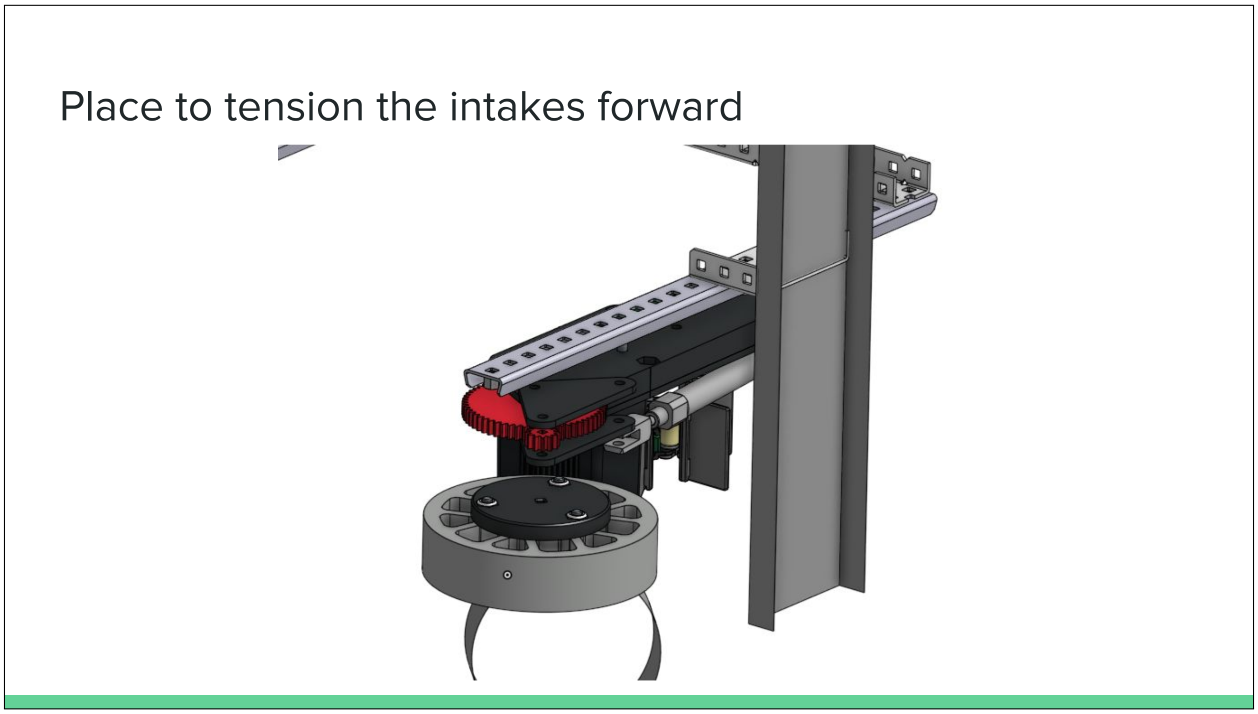


More spacers + add shafts

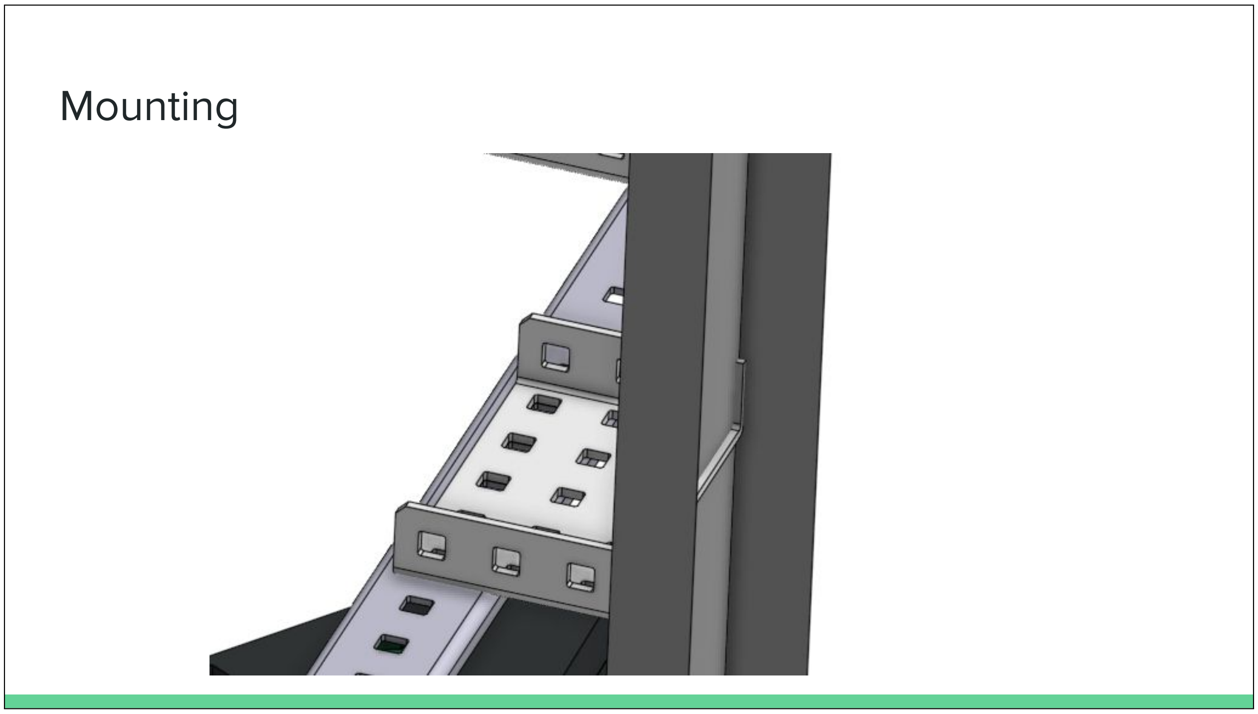




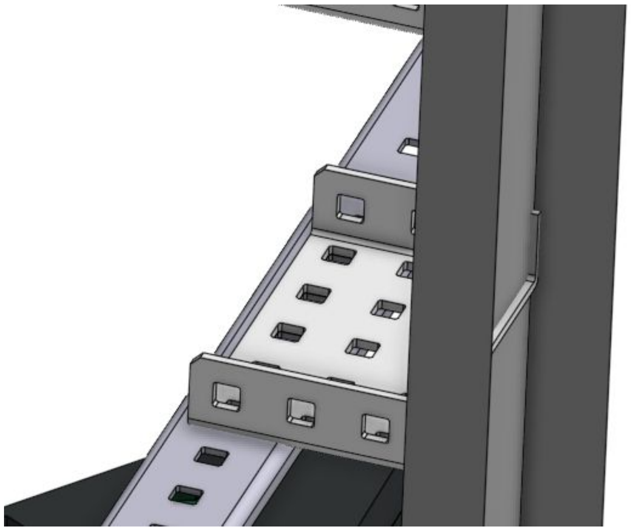
Update rails to be correct length

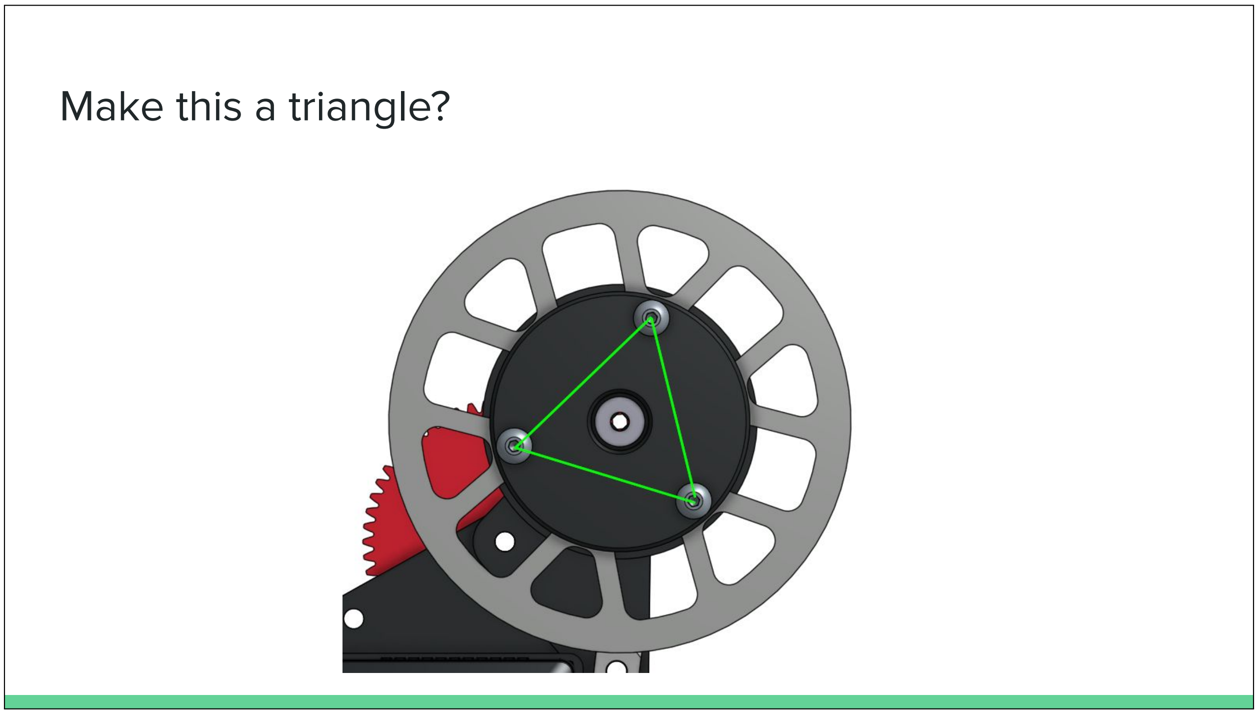


Place to tension the intakes forward

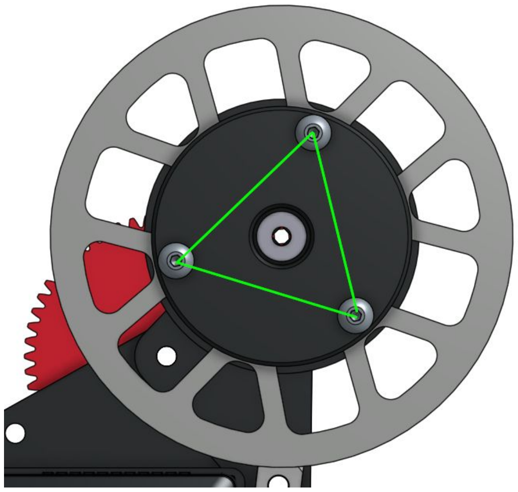


Mounting





Make this a triangle?



Notes from review: We are concerned that the intake will not be able to get inside frame. In the current state of the design, the intakes are not in the right position when deployed, although we think this is an issue with the constraints or parts and not an issue with the overall design.

To be absolutely sure that the intakes would get into frame, we considered using a pneumatic latch to hold the intake inside frame. After some discussion, we decided that this will be the best way to guarantee that we get inside frame, so for now, we will plan on setting aside at least one pneumatic cylinder for this.

7.4 Intake V3.1 Assembly

After completing the design review, we updated the intake CAD and assembled the intakes in real life.

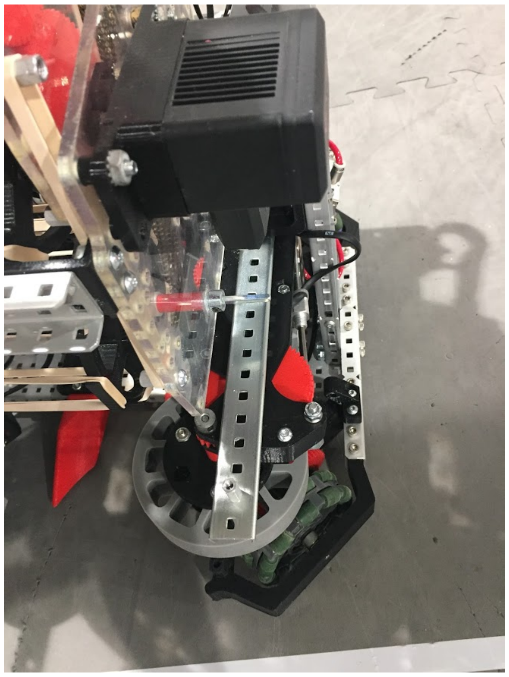


Figure 90: V3 Intakes retracted linearly to be within frame

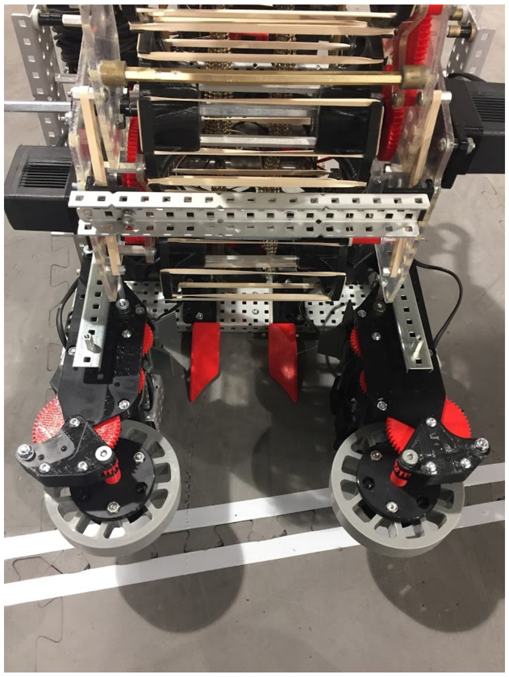


Figure 91: V3 Intakes in their extended position after they are passively deployed via rubber band tension

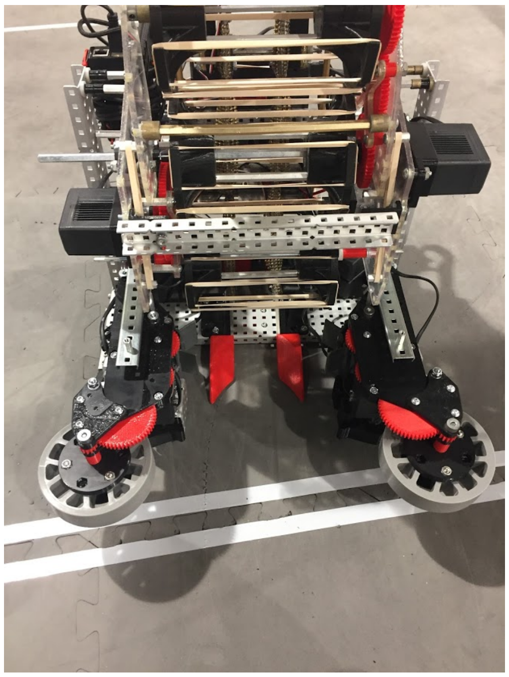


Figure 92: The V3 intakes utilize a double-acting pneumatic on each side to spread the front wheels apart, significantly increasing the intake range of the robot.

We then proceeded to test the intakes by driving around the field and in taking several balls. However, we quickly found several flaws with the intakes. In particular, we rounded out the 3D printed hubs on several of the gears; furthermore, we found that the intakes hit on the center goal, preventing us from pulling balls from it. Accordingly, we decided to begin working on another version of the intakes.

Intake V3.2 Design Review

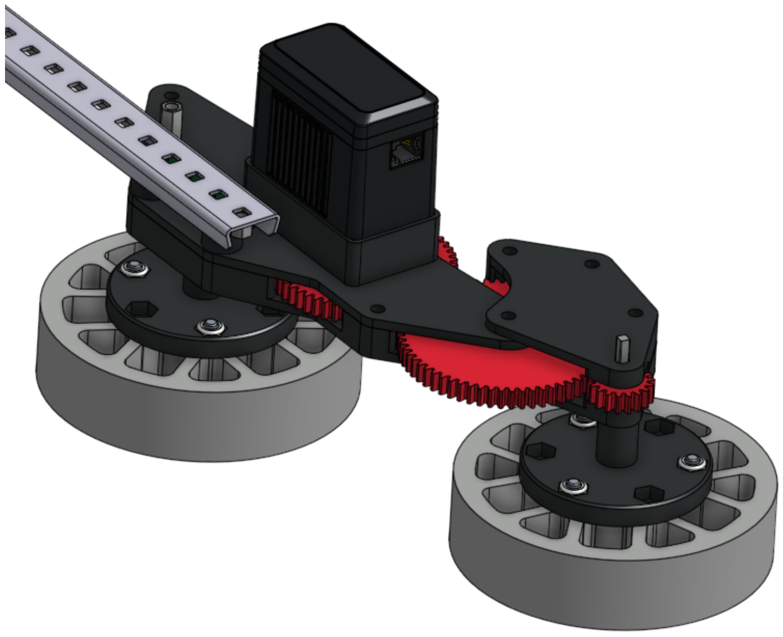


Figure 93: V3.2 Intake

7.5 V1 Design Cycle

The goal of this design cycle is to develop a system which will bring balls from the intakes at the front of the robot to the hood system at the top. Furthermore, this system should be able to move balls quickly and be able to eject balls which are not of our alliance's color in order to facilitate the rapid "cycling" of a goal, where balls of our color are inserted into a goal while simultaneously removing our opponents' balls.

We started by examining a few different possibilities for ways we could accomplish this. After a bit of brainstorming, we came up with three possible solutions, with varying levels of efficacy. 1. Hood ejector This system utilizes a top mounted deployable hook/ramp which can be actuated to redirect balls from the goal to the left or right of the robot.

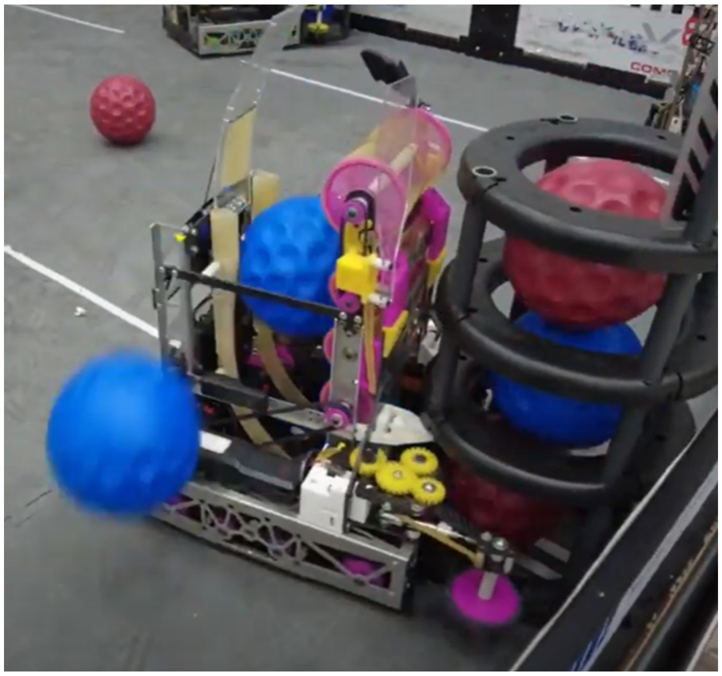


Figure 94: VEXU team SJTU utilizes a deployable hood ejector ramp at the top of their robot to eject balls

2. "Pooper" system This system utilizes a series of triangular rollers to allow the option for ejecting balls out of the rear of the robot. This is done by reversing the direction of the upper rightmost motor, which causes the balls to be redirected backwards out of the robot.

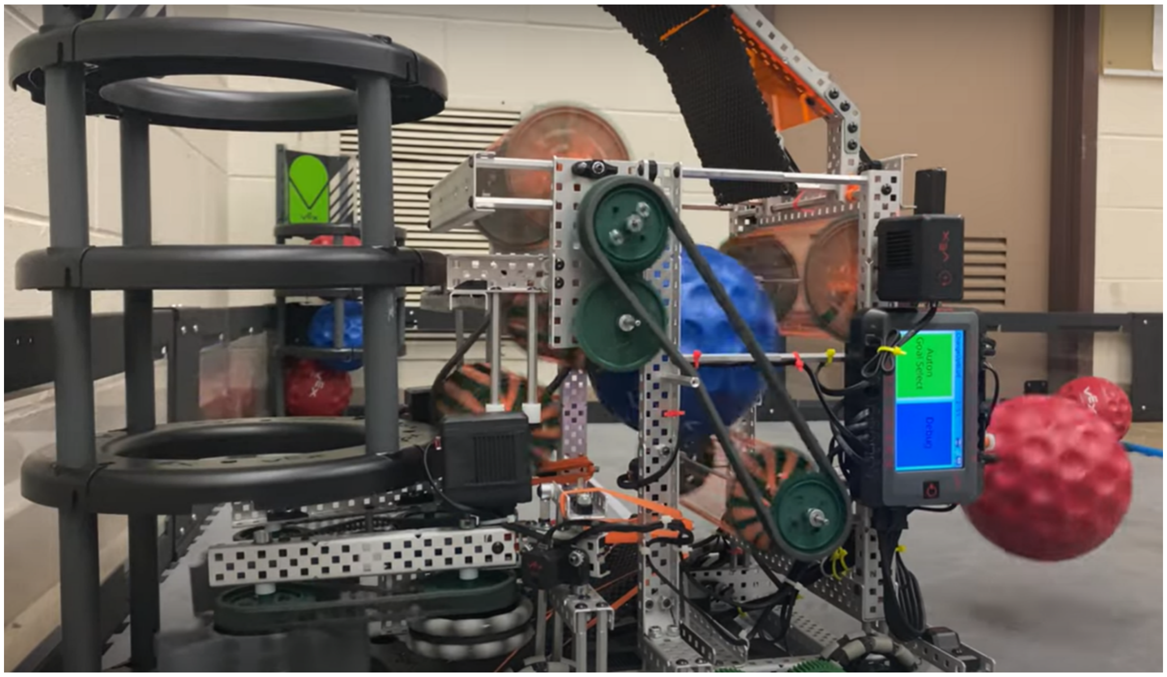


Figure 95: A pooper system utilized by VEX team 27905B

3. Trapdoor ejection system This system is a variant of the pooper system, which utilizes an openable section of the rear of the robot to allow the ball to escape.

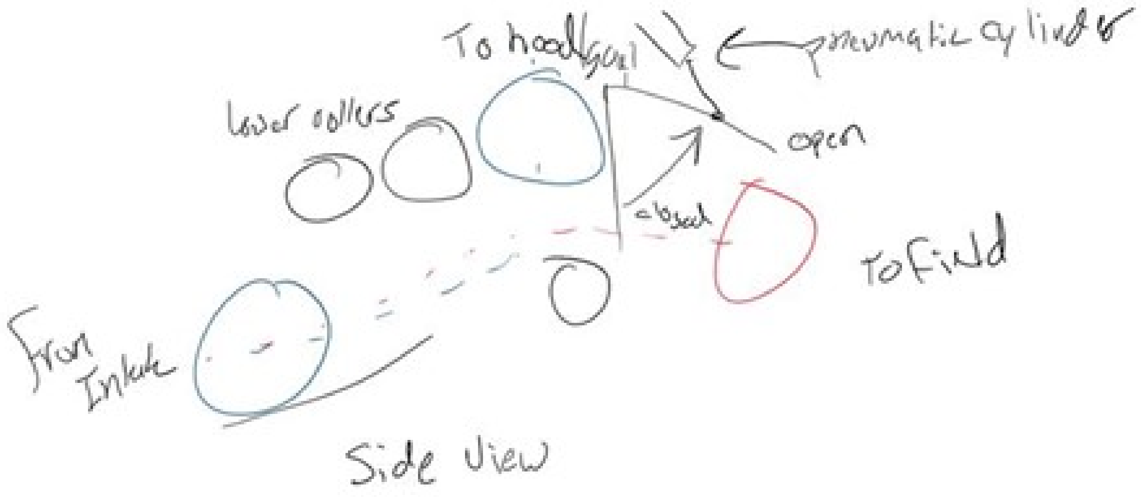


Figure 96: A diagram of a trapdoor ejection system

7.6 Potential options analysis

Notably, each of these systems has distinct advantages and tradeoffs which makes comparing them together in a decision matrix challenging. This is mainly because each design has various unique advantages and disadvantages which are difficult to compare using a simple grading scale as the primary consideration is the space available for such a design, which is reliant on the other architectural decisions made for the rest of the robot. Furthermore, we feel that each system has about the same level of design complexity, and each is relatively feasible for us to implement on our robot, so using these elements as criteria in a decision matrix doesn't do much to highlight the crucial differences between each design. As a result, we choose to instead discuss each design individually and then select the best candidate based on our discussion.

1. **Hood ejector evaluation** - The major benefit of the hood ejector system is that it is relatively simple and straightforward and allows for balls to be ejected in a robust manner. The downside of the system is that it can only eject the top most ball in the robot, since otherwise the ball will be immediately scored, which could result in an opponent's ball getting effectively "stuck" in the robot until the next goal is cycled. Furthermore, this system can only really eject balls to the side of the robot, which could be problematic in some situations, such as when cycling a corner goal, since deliberately ejecting balls from the field is illegal.

2. **Pooper system evaluation** - The major benefit of the pooper system is that it is well proven by other VEX teams, and thus leaves us with little question about its efficacy. Furthermore, it is quite straightforward to design since it only requires the appropriate placement of the pooper rollers, something which can be experimentally determined and/or figured out in CAD relatively quickly.

3. **Trapdoor ejection system evaluation** - The trapdoor ejection system , we quickly arrived at the conclusion that this system would work best as a part of a rear rail system along the back of the robot. However, it appears that the implementation of this system is made challenging because a large amount of room is necessary to allow the door sufficient room to swing out of the way of the ball, and because balls moving through rapidly could cause the system to jam or get stuck.

After discussing each system individually, we decided to move forward with the pooper system design. This is mainly because it is the simplest solution for us to implement and because it is a well proven design.

We next created a plan for actually implementing the design. In order to create the lower roller system, we decided that our standard CAD-prototype design cycle, described in section [add section] of the notebook, would likely serve us well in creating this system. With that plan in mind, we began designing an initial version of the rollers in CAD.

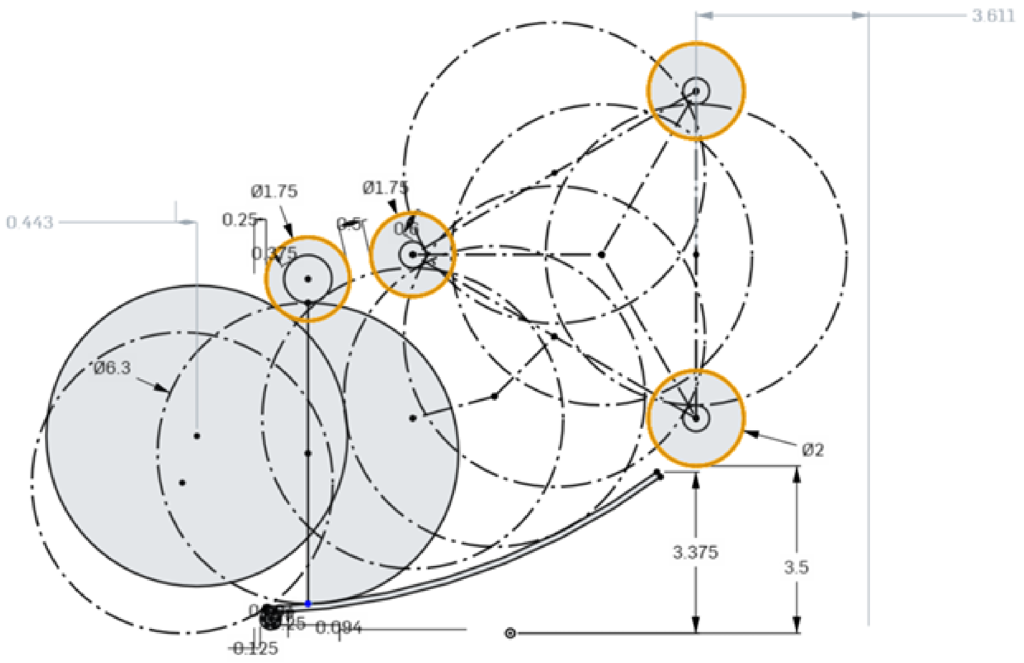


Figure 97: Geometry sketch showing the path of the ball through the rollers with the possibility of going up to the hood mechanism or out the back of the robot

After determining the positioning of all of the rollers, we were able to gear the rollers together to be powered off of the same motor (with the exception of the top-most rollers the upper of which will be powered independently to enable the ability to sort balls.

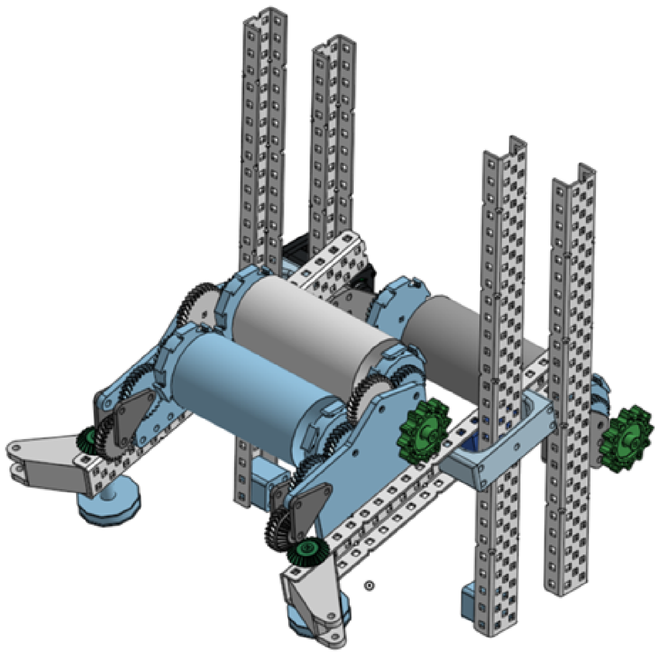


Figure 98: The finished design of the pooper roller system

7.7 Testing and Iteration

After completing the CAD, we decided to move forward with printing and testing the first set of two rollers. We also decided that the rear pooper roller and the upper pooper roller (not pictured above) should be left till later since the upper pooper was reliant on the “peacock” hood geometry, which was still in development. We then proceeded to assemble and test the lower rollers assembly in real life, attempting to intake balls from the various types of goals and cycle balls quickly through the robot.

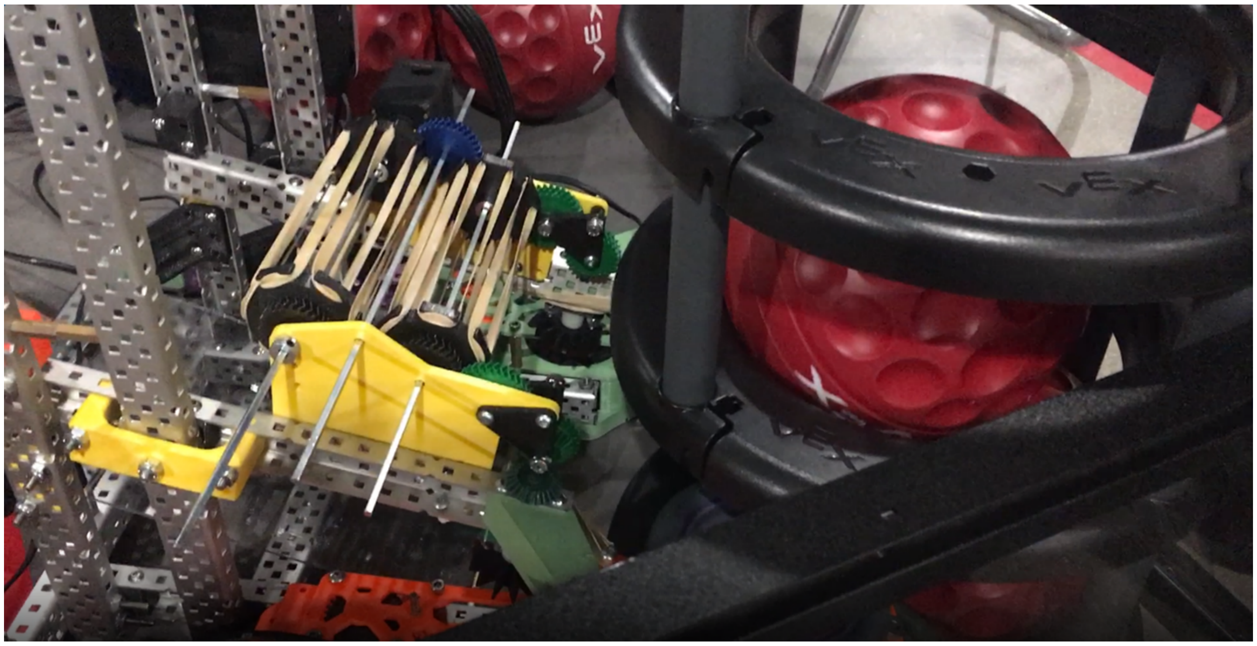


Figure 99: Testing the first prototype of the lower rollers in real life

Unfortunately, our testing quickly revealed that the performance of the lower rollers was substandard. According to the V5 motor cortex, the motor driving the rollers was unable to reach maximum speed, topping out at a measly 50% speed, and this fell further as the ball was passed through the system. After examining the lower rollers a bit more, we ultimately came up with a list of issues and prospective fixes we planned on making for the next version.

1. There’s a dead zone between the intakes and the lower rollers. Although we designed in side rollers to mitigate these issues, these rollers were a bit too far back, causing balls to get stuck. In order to fix this issue, we decided to push the side rollers forward a hole and to change their design to a custom 3D printed roller.
2. Herringbone gears are problematic. We had issues with the gears becoming misaligned, which caused a lot of friction among the lower rollers, and the herringbone pattern made assembly difficult since the entire gear train had to go on or off at the same time. To mitigate this issue in the future, we decided to switch to regular gears, and to switch to using screws or plates to hold them in place.

3. Getting the distance between rollers is important. Since the rubber bands naturally want to pull the rollers together, and aligning the rollers and plates side to side with shaft collars proved difficult, we decided to switch to using spacers between the rollers to ensure that the distance was constant.

4. We needed more rigidity across the arms. We had a lot of issues with the plates being misaligned, so we decided to add a metal crossmember running across the intake arms to add additional rigidity.

7.8 Design Cycle II

With these changes in mind, we quickly moved on to the next version of the lower rollers, implementing the various changes described above into the CAD.

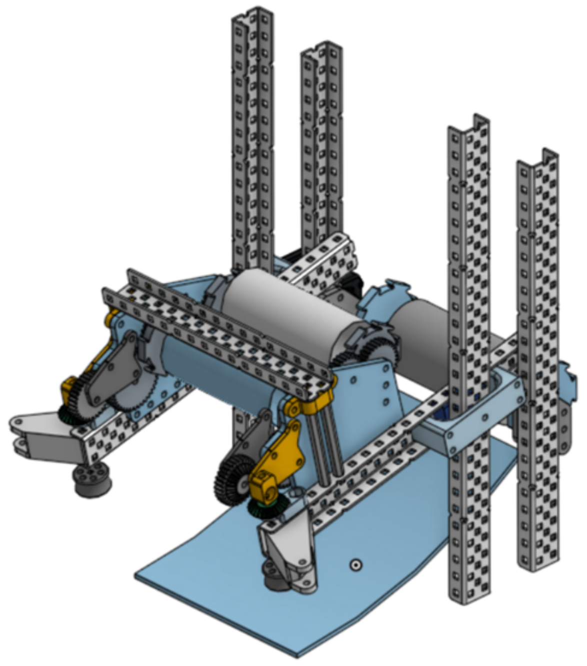


Figure 100: The updated CAD of the lower rollers

After we finished assembling the rollers, we once again tested them, seeing how they performed as balls were passed through. Because of the changes in gears and better alignment of the plates, we found that the motor was able to hit its top speed of 600 rpm (according to the V5 cortex motor info display), and that the speed fell much less while balls were being passed through the rollers. This represented a substantiative improvement over the first version of the rollers. Through the course of our testing, we also found that our revised plan for the small vertical rollers was subpar. This was mainly because they were to large; the ball struggled to get past them, and it tended to force the shafts outward, creating a large amount of friction. Furthermore, the smaller diameter of the rollers caused them to spin more slowly than the rest of the rollers and the intake, resulting in them slowing the ball down as it passed through. Thus, we realized that we needed to move rollers outwards and change their design in order to make them more effective.

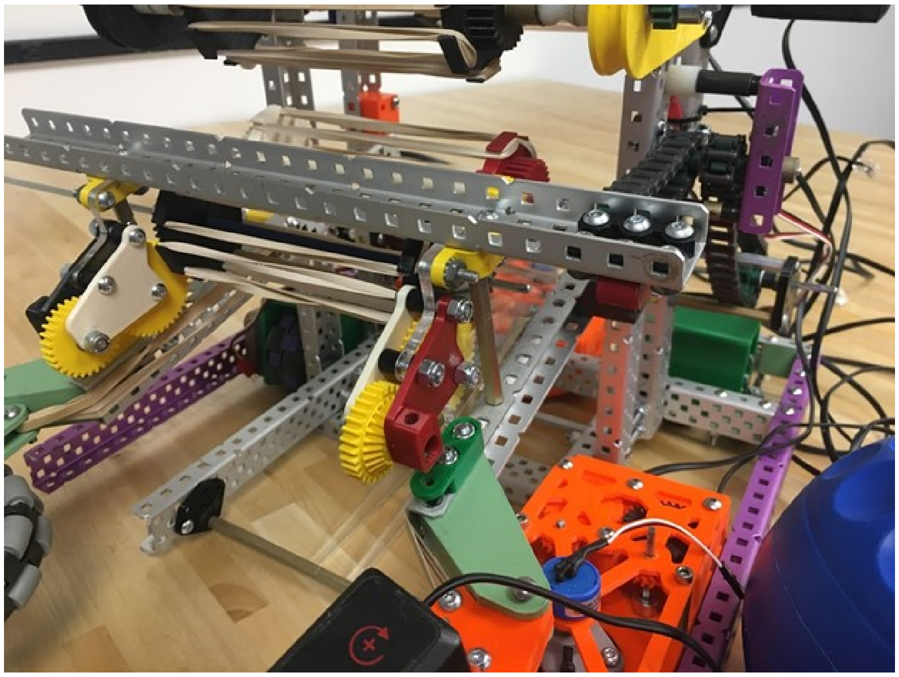


Figure 101: Assembled Lower Rollers

7.9 V2 Design Cycle

With the goals that we determined for the V2 robot, we determined that the necessity of the constraints created by having a mechanism that ejected and automatically sorted opponent balls was not necessary for skills only events where the position of balls on the field is known and does not change from match to match. As such with the removal of this capability, further discussion on the transportation of balls from the intake to the goal is discussed in the Hood section.

8 Hood

8.1 V1 Hood and Pooper Design Cycle

The goal of this design cycle is to develop a hood system for the V1 robot which can bring balls from the pooper system up and into the goal. Ideally, the hood system will also allow us to store a large amount of balls while still being able to cycle balls quickly and effectively. The system must also be able to fit into frame. Furthermore, the goal of this design is to accurately and precisely control the balls throughout its entire range of motion and direct them into the goals so that there is as little chance of the ball missing the goal as possible.

We started by brainstorming several different possible hood solutions.

1. "Peacock" hood

   This hood design, named for the hood holders which fold up and expand similar to a peacock's feathers, features a large hood roller system that swings up and over the goal. The benefit of this system is that it holds many balls and gives greater control over the balls for longer while taking up an extremely small profile while in frame.

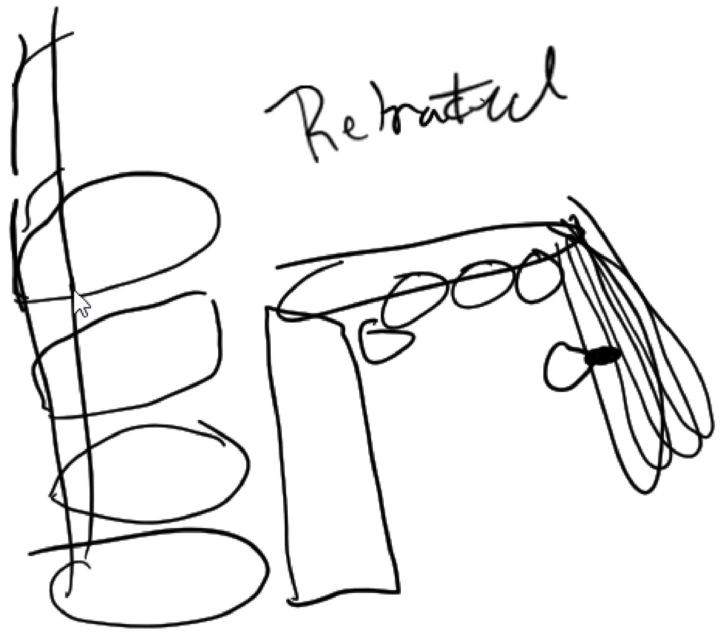


Figure 102: Retracted Peacock Diagram

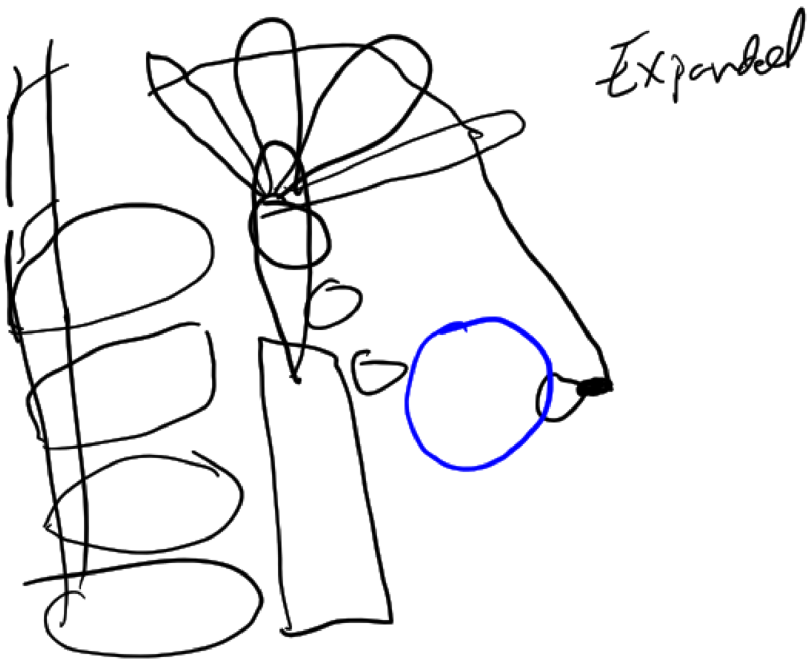


Figure 103: Expanded Peacock Diagram

2. Two part hood

   This hood design is commonly used by advanced high school teams and features two smaller sections of hood to guide the path of the ball. Generally, a roller and hood element deploys to guide balls upwards, while a smaller hood redirector unit goes forward and helps shove balls down into the goal.

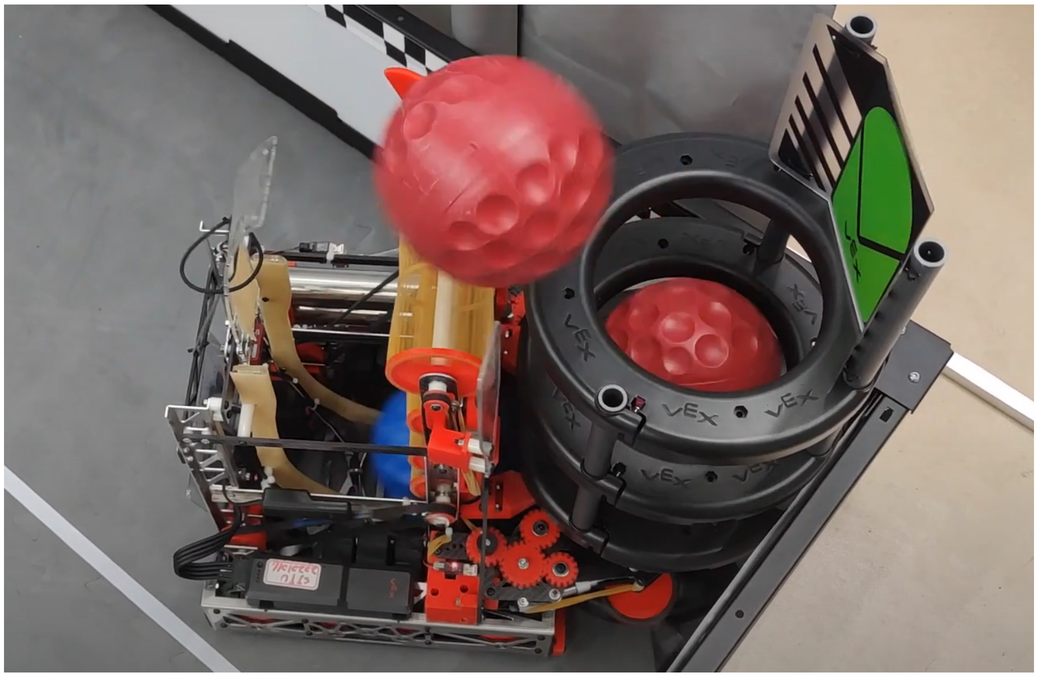


Figure 104: SJTU example of a two part hood mechanism

3. One part hood

   This design would use a folding hood system to deploy into place from only the front of the robot. Each section of hood would be attached to the previous, and would expand up and out at the start of the match.

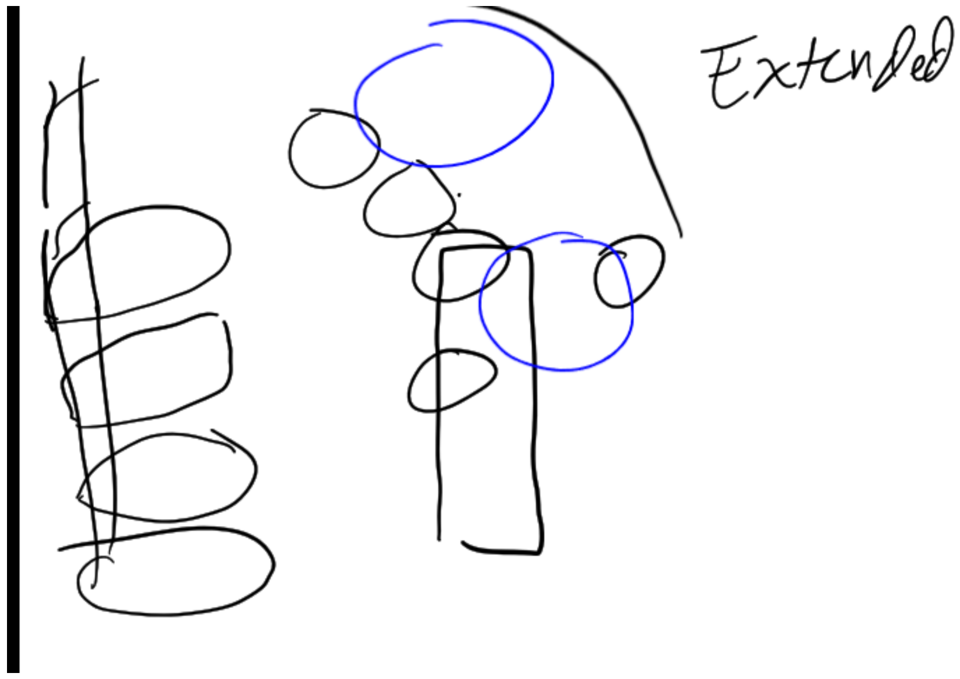


Figure 105: Extended One Part Hood Diagram

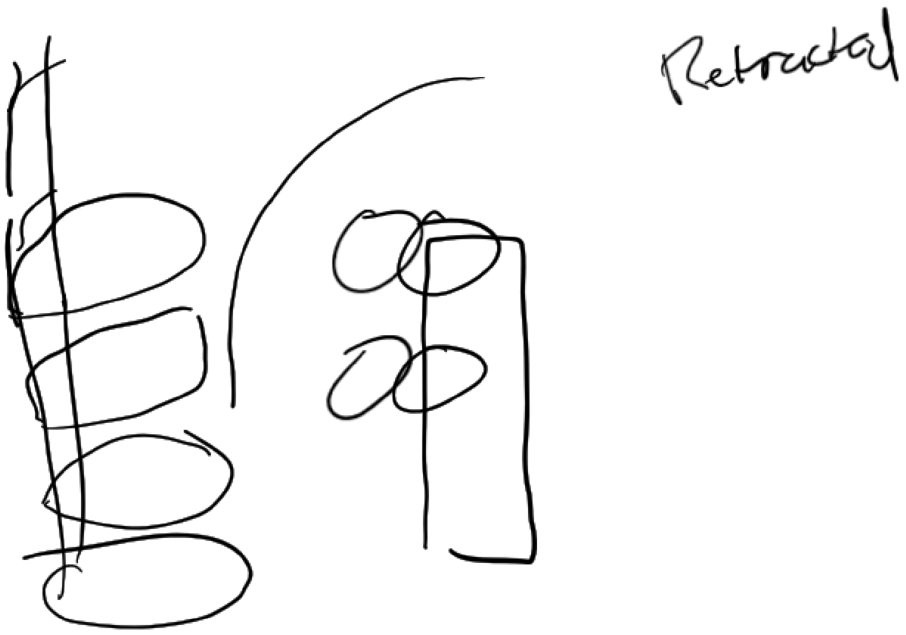


Figure 106: Retracted One Part Hood Diagram

After brainstorming different types of hoods, we created a decision matrix to compare and evaluate possible hood designs.

	Size	Scoring Effectiveness	Speed	Storage	Total
Peacock hood	4	5	3	5	17
Two part hood	4	4	5	2	15
One part hood	2	4	5	3	14

From this analysis we determined that the best potential option to attain our goals for this robot would be the peacock unfolding style of hood. This conclusion was primarily the result of the high value we placed on ball capacity and allowed more space on the robot for the development of the intake and funneling mechanisms on the front part of the robot and was able to transport the balls to a higher location on the robot to enable a more controlled and relatively fast scoring sequence.

8.2 V1 Hood Design Cycle

From these parameters and selected robot architecture, we began developing the geometry for the V1 Hood in Onshape, where we mapped out the path of the ball to determine where rollers needed to be placed and at which spacing to output the balls at our desired height and distance.

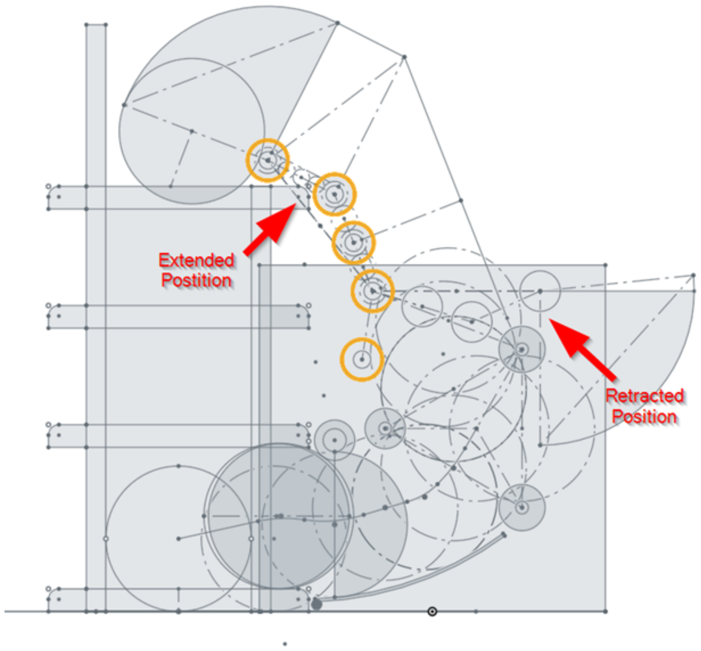


Figure 107: Peacock Geometry Sketch

From this geometry, the peacock was developed into a design that would fit within frame and enable the robot to have a capacity of 7 balls.

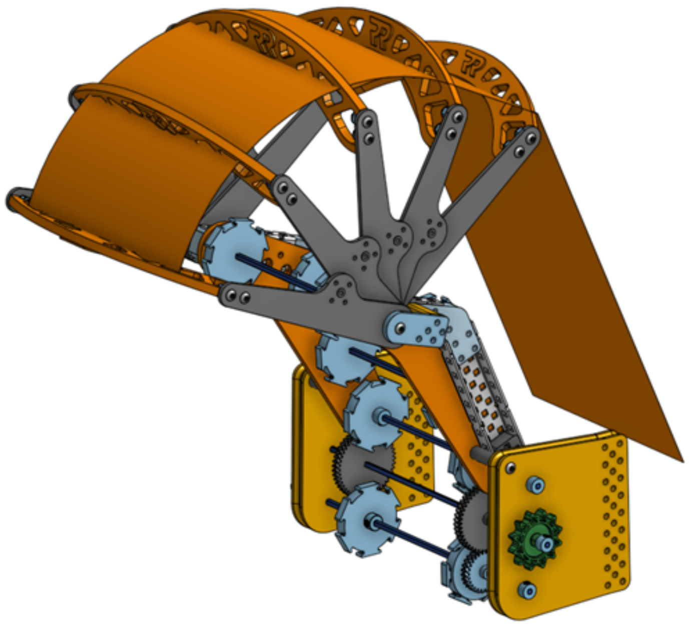


Figure 108: V1 Peacock Assembly Extended

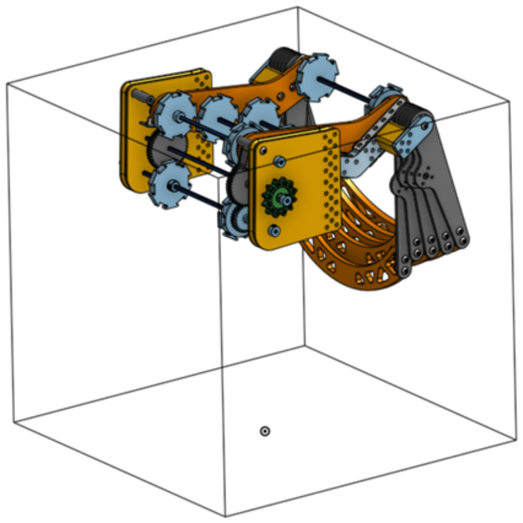


Figure 109: V1 Peacock Assembly Retracted

This was then manufactured largely with laser and 3D printing processes and tested.

8.3 Manufacture and Iteration

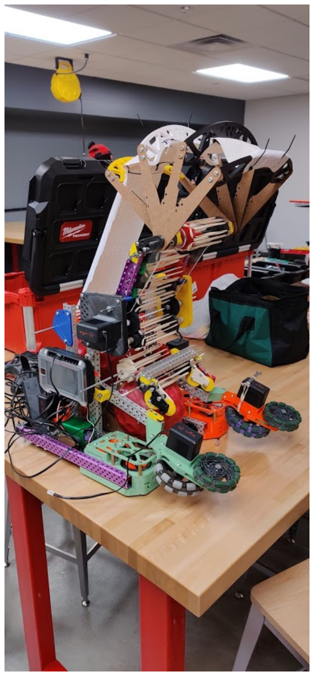


Figure 110: Assembled Version of the V1 Peacock

From this design, a few things became more clear and were addressed.

1. The speed of the rollers was overall slower than we had anticipated causing a single cycle of the balls to take around 3 seconds. **Solution: The motor's speed was increased using gears to have a 5:1 ratio, with a theoretical speed of 3000rpm**

2. The rollers diameter was small, thereby making it necessary for very stiff rubber bands and a lot of compression to maintain force on the ball, but resulted in a lot of energy **The hood was able to be temporarily adjusted to maintain a more constant compression via the flexible hood, but this is something that still needs to be improved in future iterations** .

3. The motor power was largely insufficient with all of the frictional losses of powering 5 rollers off of the single motor **Another issue that was able to be reduced by removing unnecessary gears to power both sides of the rollers, as the tension of the rubber bands was enough to tie them together, but something to improve within the next version**.

4. The back hood drawer liner material was overly flexible along the long straight run, and increasing tension on this would result in pulling the hood back farther creating a dead-zone as the ball loses contact with the rollers **Solution: See latch below**.

5. The trusses that made up the "feathers" of the peacock were more numerous than required to define the shape of the hood.**Solution: Removed unnecessary feathers**

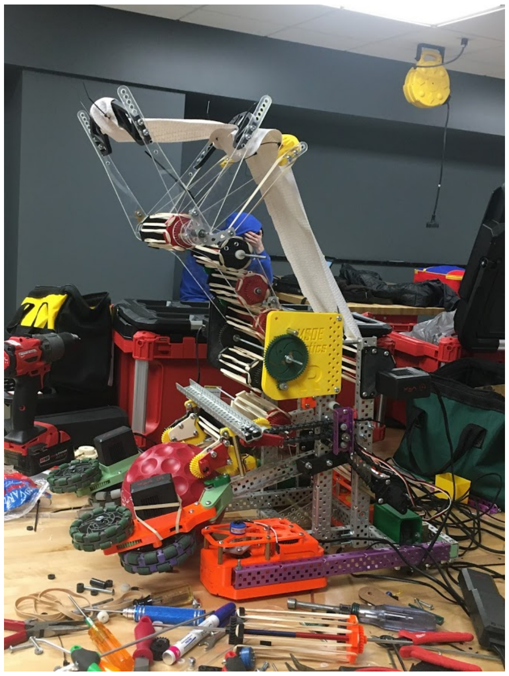


Figure 111: Improved V1 Hood with Increased Gear Ratio and less trusses

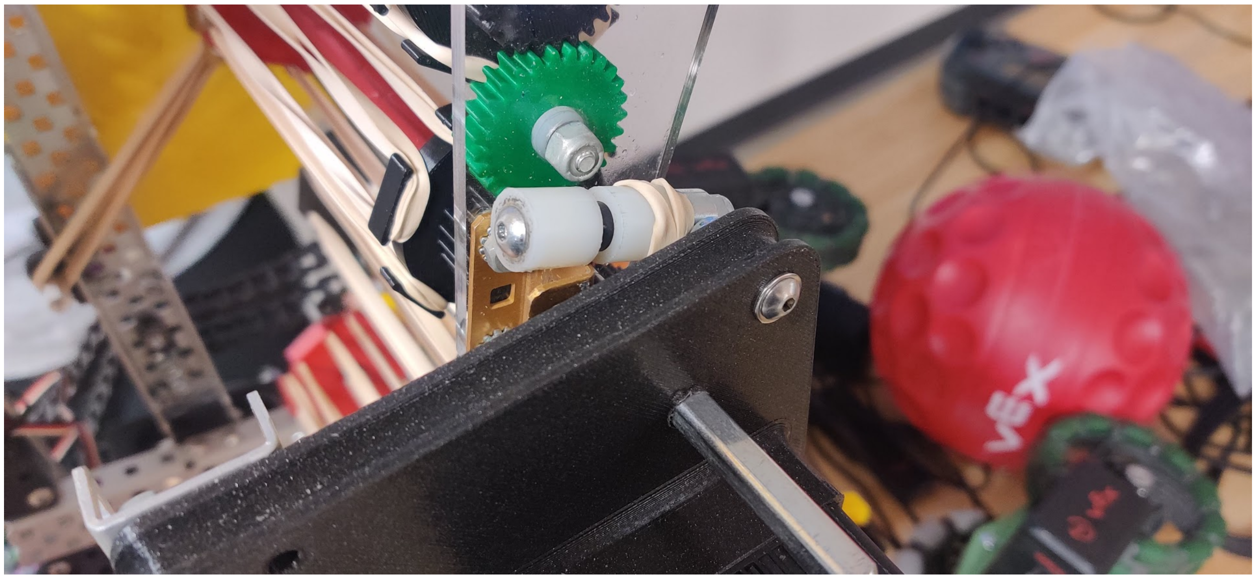


Figure 112: Latch added to lock the peacock mechanism in its deployed state after the start of the match and to ensure more consistent compression

8.4 Analysis of Function

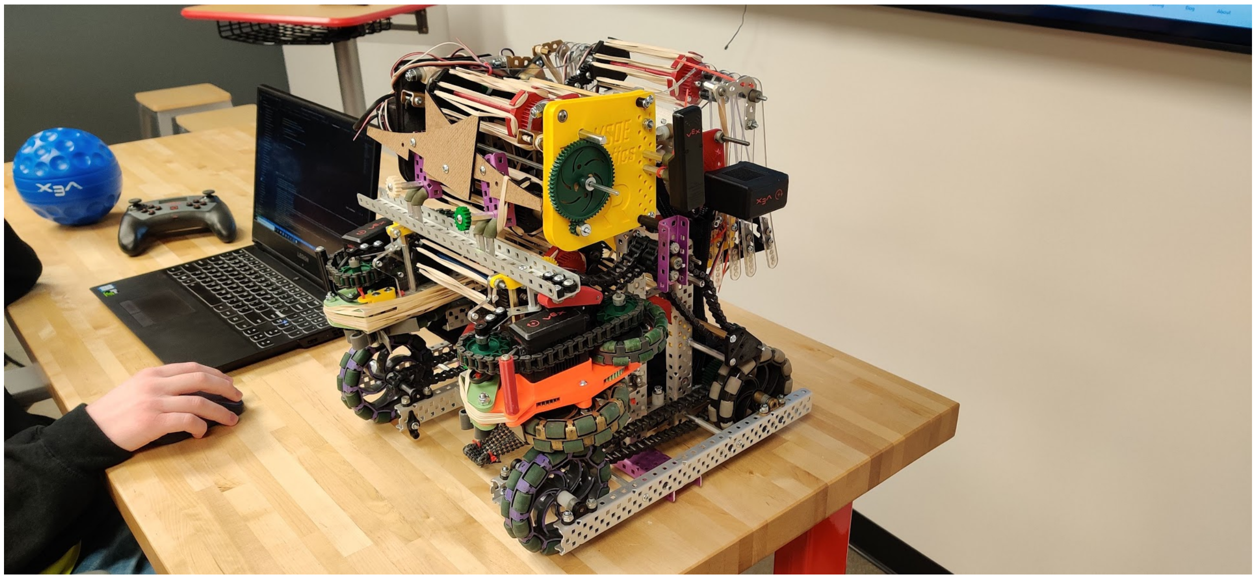


Figure 113: Retracted Peacock

With the completion of our first event, it is clear that the peacock mechanism did accomplish some of the goals we set for it better than others. Some of the highlights of this design are the incredible capacity and compactness of the mechanism considering it nearly doubles the robot's height after deploy. This height and capacity was advantageous as it made scoring in the goal very consistent, as the balls were nearly entirely directed downward. Moreover, the ball capacity of this robot if anything was excessive making it easier to hold more balls within the robot.

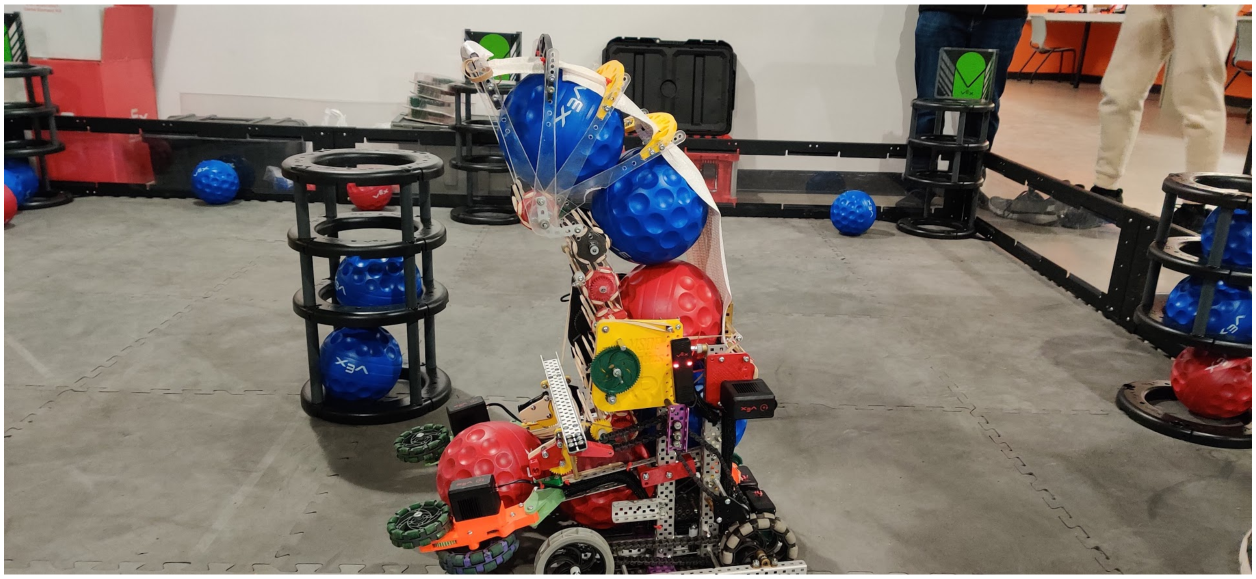


Figure 114: The 7 ball capacity

However, this design also came with its own drawbacks. With such a long path for the balls to follow, the time to cycle a ball was slow. Furthermore, the compliance of the mechanism that made it possible also introduced many changing variables and encouraged friction and dead zones to rise to the fore.

8.5 V2 Hood Design Cycle

With these concepts learned from the V1 version of the hood, we came up with a new set of goals to determine what the mechanism would look like in our V2 robot.

Goals:

- Less compression on rollers
- More reliable starting configuration
- Better up hardstop
- Less "dunk" on balls going into goal
- Better centering for balls in track
- More motor power on rollers
- No deadzones
- Less gears
- Larger rollers if possible?
- More tension on rubber bands without double wrap
- Reliable deploy
- Real mounts for sensors
- Hard backing for hood
- Simple and reliable scoring

Like everything else for the V2 robot, this version of the hood is once again being designed from the ground up. We started by creating a few different sketches of possible roller and hood configurations, then began creating the actual design in CAD. Notably, we decided to omit the pooper mechanism from this version of the hood, and to try to use less rollers overall. Accordingly, the lower roller subsystem is no longer separate from the hood, as it is easier to design and build them together in tandem.

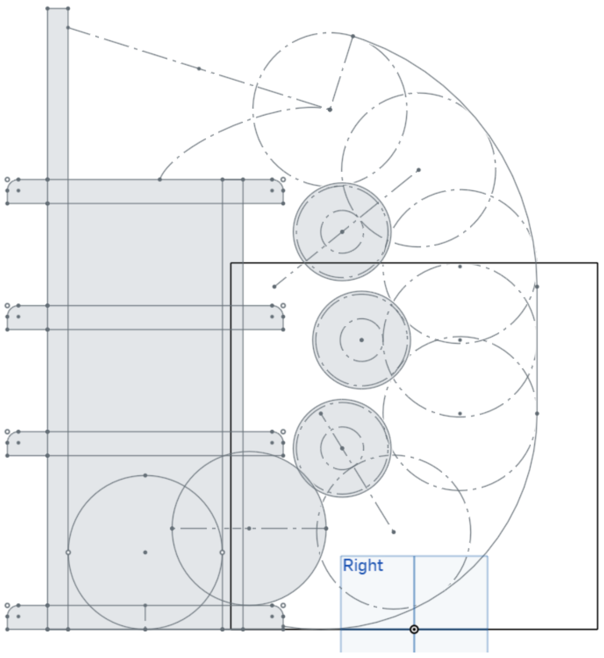


Figure 115: Three Hood Roller Geometry

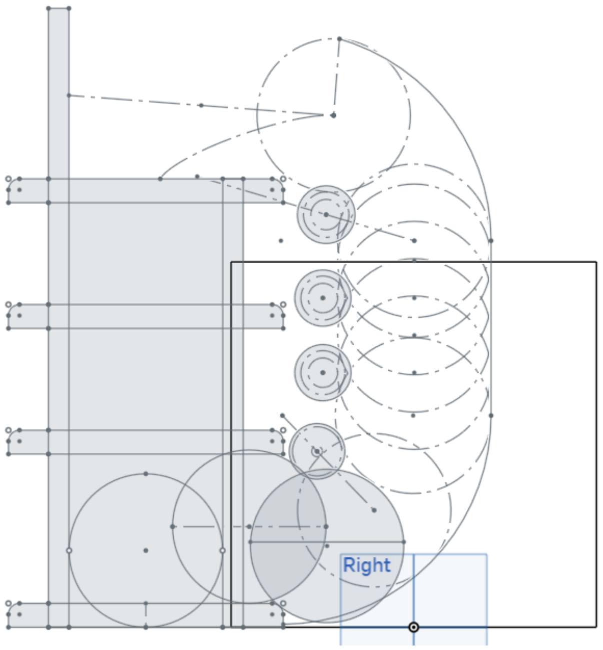


Figure 116: Four Hood Roller Geometry

After some playing around, we finally settled on a version with three large, vertically in-line rollers.

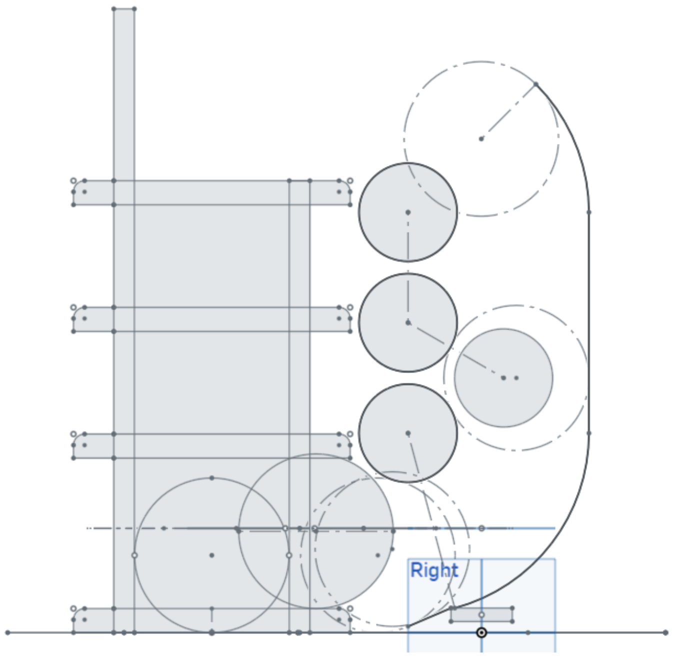


Figure 117: Final Hood Roller Geometry

From there, we used the geometry as a basis for the rest of the CAD.

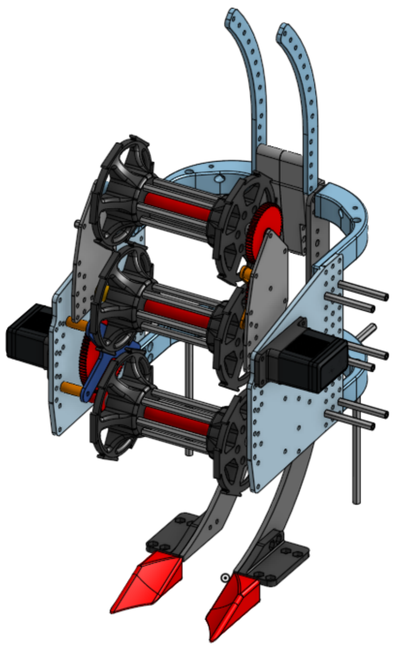


Figure 118: Hood V2 CAD

After performing a design review of the hood, we assembled it in real life and put it on the robot. We then tested it by running balls through it and deploying it from starting configuration.

Based on our tests, we decided to add some drawer liner material to strategic locations on the back rails of the hood in order to help the balls move smoothly along the rails.

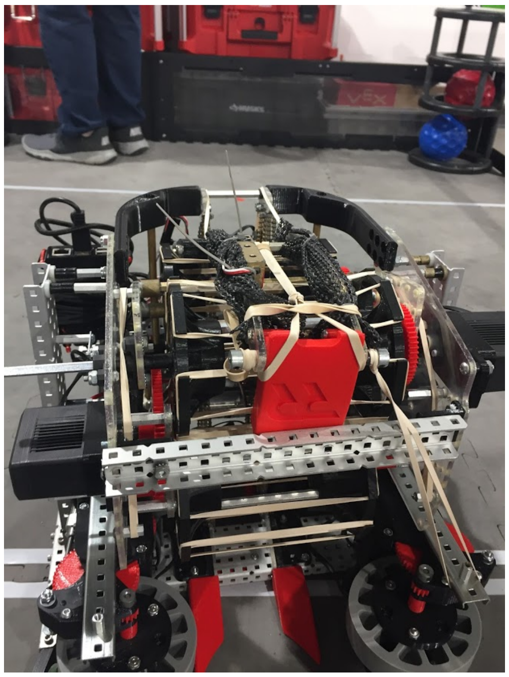


Figure 119: The V2 Hood in it's extended position with the roller and backing deployed with rubber band tension

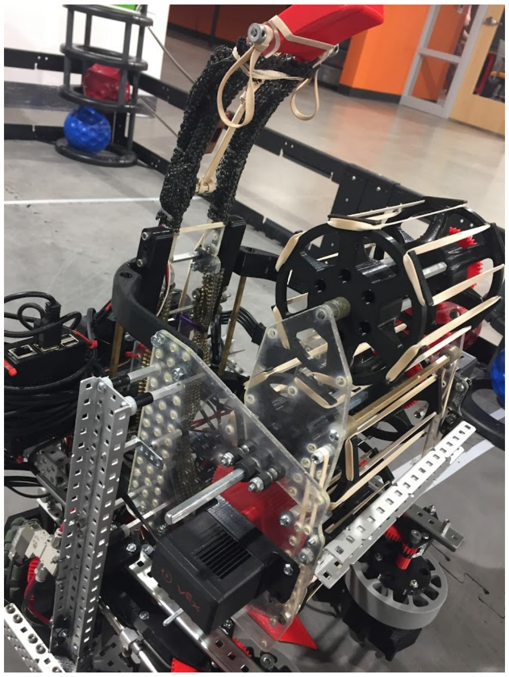


Figure 120: The V2 Hood in it's collapsed position with the backing folded down and over the deployed top roller holding it down. The hood is released by outtaking the ball from the lowest roller.

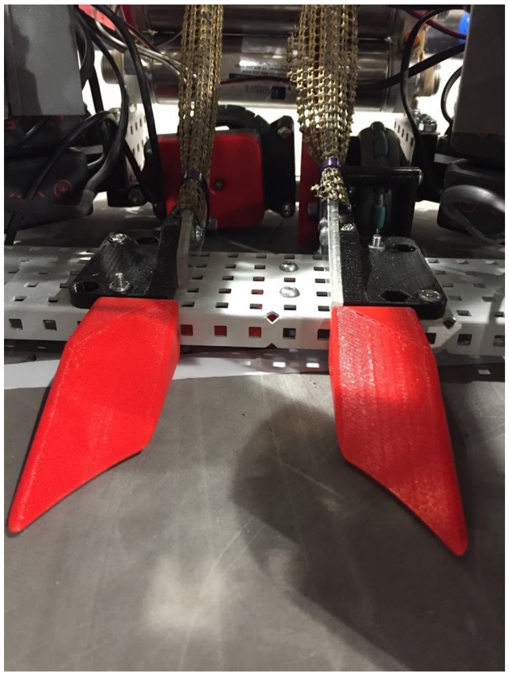


Figure 121: Ramps make a smooth transition from the ring of the goal or ground to the robot that conform to the profile of the ball. The use of two rails keeps the ball centered throughout its travel through the robot until it is scored. The use of drawer liner material prevents the ball from spinning in place, thereby increasing speed of the mechanism

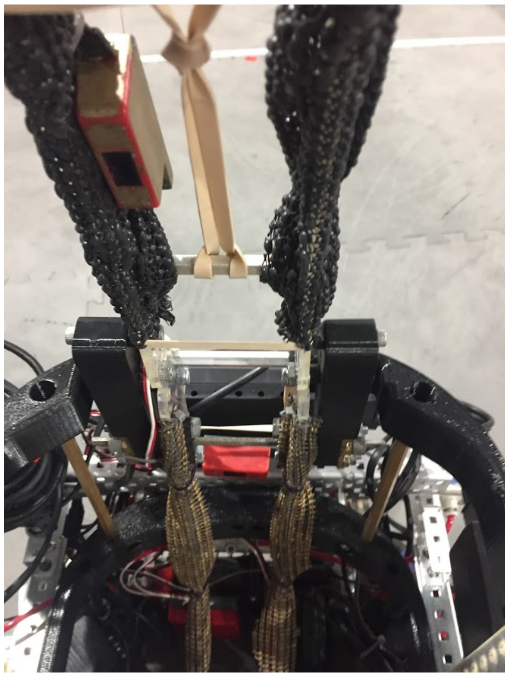


Figure 122: Two line sensors were added between the rails to detect when a ball is in front of it to automatically store the balls. We found this to be a necessary addition due to the ball's rapid travel through the robot from intake to being launched out of the top of the robot and to be able to retain both balls required in skills runs.

9 Programming

As an important side note, all of our code is open source. If you want to reference our current repository, please visit our main repository on GitHub:

https://github.com/msoe-vex/ChangeUp

Additionally, our other non-robot projects are also hosted on our GitHub, such as our Path-Planning Web Dashboard and our React based team website are all hosted publicly.

9.1 Reflection on the Previous Year

During our 2019-2020 season, software became lower priority for our robots and we barely had autonomous modes for our robots. As we learned quickly at our competition at Purdue, the extended autonomous mode of matches allows for teams to easily get an edge in competition and will often create a lead that the opposing team can't beat. This made it more important than having the best performing robot mechanically, as often times a consistent and reliable autonomous will get you additional points (in the form of the autonomous bonus) that can't be changed during game play, and will start you up at an advantage when driver control begins.

Another key detail in programming that we noticed at our event was consistency. Even without an autonomous mode in some of our matches, we were able to tie autonomous with teams who weren't able to run their programs consistently. While this was good for us, it suddenly made the match much closer score-wise and gave the opposing team (us in this case) a much better chance to win. Because of this, we wanted to focus on getting consistency more than points for our programming, to make sure we are able to average out more points instead of maxing out.

9.2 Identifying Challenges and Goals in Software Architecture

Working with our team, we also made some goals for our next few seasons. These goals were more from a leadership perspective of the team but are important in guiding how we develop our software in coming years.

This year, we want to be able to incorporate camera input to control our robots, using vision processing systems to determine the state of goals automatically, give the robot heading and position data on nearby game elements, and allow us to be much more proficient in autonomous. Since we weren't able to complete strongly in autonomous last year, having a consistent autonomous is also our primary priority this year.

Over the next three years, we want to build out a platform for both AI models to run on our VEX U robot (through an external processor such as a NVIDIA Jetson or a Raspberry Pi), along with build up the infrastructure to compete in the VEX AI Challenge (VAIC). VAIC is new this year, and with our school putting heavy emphasis in developing our CS curriculum with the new CS major offering, we wanted to utilize this program to help introduce students into

AI for robotics. We believe it will be too difficult to implement this year alone but think we can start working on the infrastructure to get our team ready for a future VAIC endeavor. Over the next five years, we want to maintain flexible and allow us to continue developing into new technologies, sensors, and algorithms, while not being bogged down by legacy code. A number of us have experiences at companies that produce software (Rockwell, Direct Supply, SpaceX, among others), and all of the companies have had some components of legacy code within them. We were able to compare our experiences between companies, as well as the applications for our respective code bases, to work on designing an infrastructure that will allow us to be expansionary in the future, and not get stuck with lots of technical debt as we try to develop new technology and add new members to the team.

Looking at our software infrastructure, we wanted to outline the needs of our system for this year, as well as the future. These priorities are outlined below:

1. We want a system that will allow us to easily expand to more advanced, and higher quantities of sensors in the future. Sensors are crucial to programming, and as we continue as a program we want to be able to try new sensors out, both to enhance our programming and to give students experiences with these new sensors
2. We want a system that can handle increased amounts of throughput. As we add more sensors, we need to be able to support these electrically, as well as programmatically. This means that we need to have enough I/O interfaces, as well as being able to pass data into/between one or more devices
3. We want a system that is easily testable. Ideally, we want to be able to run testing and build scripts in CI/CD through either Github or Gitlab, allowing us to maintain high code quality as we bring in new members to the team.
4. We want a system that is easy to bring new students into. One of the core missions of our team is to introduce students to new software concepts, regardless of if they're a software engineering major, or a mechanical engineering major. We want to have modularized systems that allow people of all ranges of experience to jump in where they are comfortable

9.3 Researching Possible Architectures

With our criteria specified, we wanted to identify current solutions to robotics architectures both inside and outside of VEX. Through this, we are going to look at systems used across VEX, FRC, and industry with the hopes of finding an ideal system, but also looking for architecture components we want to add.

We don't plan to find an "all in one" solution to what we want. Incorporating testing, CI/CD, future expansion ability, and matching the above criteria will require some sort of custom architecture. We are going to list out any components of software systems that we find below that we want to add to our current system.

- Source control is critical for team development projects. Similar to last year, we plan on using GitHub (while looking at GitLab due to our school having licenses). Our GitHub repository will make it easy for multiple members to contribute to the code, while also enforcing code quality through merge requests. This year, we want to make merge requests standard, allowing students to perform code reviews on incoming code to make sure it works and follows our style/naming conventions

- As mentioned above, most industry software companies use some sort of CI/CD to allow for increased code quality. This is commonly done utilizing Docker to spin up containerized instances of code, where we can test to make sure the code compiles correctly, make sure it passes all tests, and make sure it follows different defined styling conventions before allowing people to push to source control

- Better testing platforms will be important as we expand our code base. Support for testing frameworks such as Microsoft Unit Testing or Google Test will be important as our system expands and becomes more complicated, allowing us to move towards test driven design mentalities, and allowing us to design software that works without needing to test on the robot for each iteration

- We've found some software platforms that use topics for creating software components. Similar to industrial robots, our robots will often have separate mechanisms that are isolated from each other (such as the intake and tray tilter from last year). Some software platforms publish topics for each of these subsystems (commonly on HTTP ports or over data connections such as serial or UART), allowing distributed robotics systems to send and receive data over multiple actuators, sensors, and/or robots

- Keeping our code modular will be essential in keeping high code quality, good testability, as well as allowing more members to contribute. If a code base becomes a monolith, it makes it much harder to manage more engineers on the platform, as each individual needs to be aware of what everyone else is doing and how it affects them. With modular platforms, we can easily distribute the work across multiple sub-systems, allowing people to work with significant less interference and interruptions

- Documentation is essential in making it easy for new members to join onto the team. We want to find a platform or infrastructure to allow new members (with little to no experience) catch up on what we're doing for programming our robots, find areas they're interested in, and learn more about what we're doing. Joining a new team can be easily overwhelming, especially in College. We want our platform to support this transition as much as possible, making it easy for new members to join.

- Allowing for platform independence in our code will be critical for growing our organization within our university. In an ideal platform, we want to create libraries that can be used across multiple robotics organizations, such as our VEX U team, our NASA robotics mining competition team, and any other research and/or competitive robotics programs at our university. This will allow for a larger development effort to be put towards the library, and make it possible for multiple programming teams to contribute to the same code base.

With these points defined, we started conducting research into the different platforms we could pursue in developing our robot code.

9.4 Researching and Selecting a VEX Development Platform

One of the first systems we looked into was VEXCode. VEXCode is available through VEX, and is one of the proprietary systems developed specifically for V5. Through some of our initial testing, we developed some pros and cons of the platform.

Pros:

- Integrates firmware and configuration tools for the V5 system into the IDE
- Supports C++ development
- Clean IDE with configuration settings

Cons:

- Proprietary platform - does not support larger scale products, and doesn't seem to have good support for files outside of C++
- No expandability from within the IDE
- No integrated source control tools
- Little to no integrations or modifications allowed with system files
- No access to the internal build system

In continuing our research, we also looked into the Purdue Robotics Operating System, or PROS. We had used PROS to some depth during the previous year, and were more familiar with it. Through additional exploration, we added more pros and cons about the platform.

Pros:

- Built to be mostly open source - very easy to work with the source code
- Built around C++, and easy to expand and modify the code base
- Primarily uses Atom, an open-source IDE for development
- Higher level of expandability of the platform - easy to include external header files
- Robust documentation for the PROS API
- High levels of access to internal and system files

Cons:

- Atom can be hard to work with, and doesn't provide much customizability in itself
- The PROS build system is accessible, but hard to modify
- PROS has some issues with specific sensors and helper functions, as well as a lack of documentation for some lower level functions

With PROS and VEXCode being the two main build systems available to VEX teams, we looked into both and decided on going with PROS. Through discussion of the points we made above, we found that VEXCode did not support most of the ideal factors we wanted in our software system when compared to PROS, and ruled it out due to being insufficient for our needs. This, however, did not make PROS a perfect system either.

One of our largest problems with the PROS system is it's usage of Atom. The Atom IDE caused us lots of problems last year, and isn't very usable due to a poor interface design (in our opinion), as well as issues with crashing and working with our file systems. In order to make it easier for us to develop code, we want to experiment with porting the build system to VSCode, as VSCode is an extremely lightweight application that we can run on most platforms, while supporting numerous extensions that give it the capabilities of many higher end IDEs.

To start the process of porting over the Atom IDE into VSCode, we wanted to list out the features contained in the PROS Editor that we would lose if we edited the files raw in VSCode. Then, through various extensions and commands in VSCode, we can work to re-add these features back into the program.

- PROS Editor allows for building code directly through the IDE, making it easy to check for problems in the code
- PROS Editor also allows for easy uploading of code through the IDE
- The PROS Editor also has a feature to open up the terminal with a V5 cortex, for debugging over serial
- The PROS Editor also provides strong error highlighting with the PROS APIs, along with providing intellisense functions to help with auto-completion during typing

We tried to find other features through Atom, but weren't able to find anything notable that the IDE provided. We were positively surprised to see that these were the only features that needed to be ported over, as most of these behaviors are encompassed within the PROS command line, using the following commands:

> *prosv5 build*: This command builds the present working directory, assuming it is in a PROS project

> *prosv5 upload*: This command uploads the project in the present working directory to a V5 cortex (or controller for remote uploads) connected through serial

> *prosv5 terminal*: This command starts a terminal over a serial line to a connected V5 cortex (doesn't work remotely through the controller)

We were able to replicate these features through VSCode tasks within a few hours of testing, allowing us to nearly completely transfer over the functionality of Atom into our VSCode environment, allowing us to stack additional features on top of the current build system.

One of the initial problems we ran into was a lack of intellisense with the PROS library, as well as error highlighting in the code. Similarly to the other tasks, we should be able to implement parsers to detect errors from compilation messages, and show them in the current IDE.

Through some testing, Andrew was able to develop a task to perform basic highlighting of errors with Regex, looking for build warnings and errors, and displaying them both in the error output list and within the code segment as well. We had some issues with errors properly showing up in some files, and some errors not showing correctly in the error window. However, we also found that the build errors were descriptive enough to allow us to easily find them within the code. We also found that the error messages themselves included the locations of the files in them, as the compiler would spit out the location of the problem, both within the file, and with it's respective line and column numbers. This made it really easy to track down errors, and with a little bit of practice programming on the platform, we deemed a lack of visual error representation to be a non-issue due to the other ways we could debug the code.

With our new IDE functioning correctly, we moved onto looking for extensions to help our development. Normally extensions can/are be selected by developers based on preference, but since we would likely be working with students who are new to programming in some levels, we wanted to find a recommended grouping of extensions to make programming easier. We tried out random extensions from the VSCode Extension Marketplace, and here are our recommendations:

- C/C++ Plugin (ms-vscode.cpptools): This extension provides many features to improve the intellisense and development experience with C++.
- Visual Studio IntelliCode (visualstudioexptteam.vscodeintellicode): This extension expands the capabilities of intellisense within the code, and makes development easier

Our list was fairly short, but with these two extensions we were able to make our experience using VSCode much better.

With the limits of our platform expanded, and with reasonable knowledge of what our current development system could support, we start talking about how we wanted to structure our code base for this year.

9.5 Designing our Initial Software Architecture

Designing a software architecture is a very challenging task, as there are many components that go into the design. In one of the previous sections, we outlined many of the desired features we wanted to see in our platform. However, it's hard to gauge the success of many of the features without first beginning some rough implementation. Nonetheless, to get started somewhere, we started looking into how other robotics teams developed code for their robots through research on GitHub, VEX forums, and on Chief Delphi.

We started by looking at what some more advanced teams were doing in the FIRST Robotics Competition (FRC), as these teams often have larger team sizes, with more industry guidance from mentors. Many teams in FRC utilize wpilib, which is a build system designed by WPI for FRC teams. This build system is analogous to PROS for us, and we focused our search on unique architectures made to stack on top of this build system. Our philosophy was to find external build systems not integrated directly into wpilib that teams were having success with, in hopes that we could either find or make an implementation for the V5 system.

From our various experiences in FRC, our search for advanced software architectures led us to FRC Team 900. In high school, some of our members followed their various publications from their Zebracorn Labs, where they detail some of the advanced vision, ML/AI, and architecture work. They also have many code releases, and examples that are easy to work off of. We took particular interest in their series of papers focused on implementing ROS on their current architecture. ROS, or the Robot Operating System, is a robotics architecture built for industrial and research-grade robotics. Through our initial research, we were able to find numerous packages on path planning, localization and odometry, sensor fusion and signal processing, and robot control packages.

One of the most interesting features of the ROS architecture is it's modularity. We were extremely impressed at the capabilities of the platform, and how easy it was to develop a large system with multiple modules. The core concept of ROS is based around the Publisher/Subscriber software pattern. Different components of a robot are broken up into different "nodes" which can either publish data to a network, and/or listen for data on a network. This structure allows for development of code with low coupling, producing subsystems that don't have unnecessary reliance on other systems to function. Especially with larger software teams, this is essential for fast development, as it allows for multiple people to work on the same code base at one time, without coupling dependencies between subsystems slowing developers down (such as developer A waiting on developer B to finish a section of code that is a dependency to developer A's work).

We found this platform extremely promising, but wanted to do some additional exploration into code implementations in VEX U to see what other teams were doing for their robots. We started our search across several locations, mainly scraping through GitHub repositories of teams we've competed with in the past, looking for old code samples to preview different architectures used by different teams.

We weren't able to find source code for many VEX U teams online. The Purdue Team (purduesigbots) doesn't seem to post their robot code publicly, which didn't help us much. Upon further searching, we managed to find the repository for the UTAH VEX U team (UTAH-VEXU-Robotics), who also seem to be starting on the path of implementing ROS for some components of their robots. While many of the repositories seem slightly outdated and not fully developed or functional, they had groundwork that showed even more potential towards a ROS-based architecture on the V5 platform.

One of the big problems with utilizing ROS was our lack of networking ports from the V5 system. Typically, ROS is used to communicate over HTTP/HTTPS ports to access data from multiple servers. However, the V5 cortex only has serial peripherals, which added extra complexity from the Zebracorn's ROS implementation. We dug into the Utah VEX U code and found examples from a rosserial package, which included drivers for communicating to and from the V5 cortex through the USB serial port. Searching for more information about the rosserial package, we came across a post on the VEX forums about a user introducing support of the previous VEX Cortex system, and basic support for the new V5 system into ROS through drivers in the rosserial package. In the post (see: Use ROS to Talk to the Cortex or V5), user CannonT posts links to the rosserial libraries, which include needed build scripts to include the necessary ROS files and build system into our PROS pipeline, while also including example code for communicating with the V5 system.

We really like the features that ROS adds to our architecture. With easily usable packages in the ROS environment, we can expand our robots to support many features not possible directly within PROS, such as adding support for custom cameras, robot visualization in an external environment, and external co-processors (such as a Raspberry Pi or Jetson Nano) to do processing at a level not possible on the V5 cortex. With the prospects of this system, especially in potentially running an VAIC team in the near future, we plan on dedicating several weeks towards research and testing of this platform to determine several factors.

9.6 ROS Experimentation

ROS has the potential to fulfill many of the technical needs posed at the beginning of our season for our software architecture, and to determine the viability of using ROS in our software stack, we wanted to set up a multi-week experiment to answer several of our large questions about the platform. As mentioned briefly above, we were only able to find one other example of this being attempted in the VEX platform, and the code in the examples was mostly in a Work in Progress state. We decided that we would run some experiments with implementing this software platform on top of our PROS build system, utilizing the provided rosserial package for communication with the VEX V5 cortex. We plan for this set of experiments to run for several weeks, and during this time we seek to answer the following questions:

1. What hardware changes will be needed to support this architecture? Where are the VEX legal limits on what we can and cannot add?
2. What does a ROS build system look like? Specifically, where can ROS integrate with PROS, and what is the extent of the potential integration?

3. What does the development time on the ROS build system look like, when compared to developing robot code on an entirely PROS build system?

4. What overhead will be introduced into our programming cycles for producing code for a new robot? Is there way to reduce this overhead, and at what cost?

5. ROS primarily communicates over serial and/or HTTP/HTTPS. What level of abstraction should we operate our robots on? What code could theoretically be ported to an external processor, and what code is best to keep onboard the V5 controller?

6. What latency is added into the system with ROS? Will we still be able to perform near real-time operations? (Specifically, will we be able to offload feedback loops onto a co-processor, or will these not be as responsive as needed?)

7. What bandwidth can we achieve between our V5 cortex and a co-processor?

To be clear, latency is the time between an input being received by the robot, and the output being performed by the robot. High latency is extremely problematic, as this will delay actions being performed by the robot (such as increasing the time between the driver pressing forward on the joysticks, and the robot moving forward). Additionally, bandwidth is the total amount of communications (in bytes) that are able to be sent between the V5 cortex and the co-processor. This is important, as this will be the limit on the amount of communications between the co-processor and V5 cortex.

Our initial tests used a laptop as a ROS master node, running through WSL 1. This allowed us to start a ROS environment on the laptop, and wait for incoming connections via serial from an external device, such as the robot. This in the future would need to be isolated on the robot, most likely through a Raspberry Pi or NVIDIA Jetson, or other devices similar. This would also require a battery bank able to supply the required power for either device, which gets more challenging in the case of the Jetson due to it's higher power draw.

Through working with the packages, we became more familiar with catkin. Catkin is the integrated package manager and build system for ROS, which also utilizes cmake for the actual C++ builds. ROS has native support for both C++ and Python, but for now we plan on conducting most of our implementation in C++ in order to stay parallel to any future robot development.

A problem this may pose comes in if we try to add custom packages. Currently, ROS requires catkin to properly compile down external packages into the source files needed to run executable code. Alternatively, PROS also needs to compile code down for the ARM processor in the V5 hardware, which isn't handled in the ROS build system. We believe that it would be best to have a two-stage compilation process, where we first compile all the ROS dependencies, copy them into the PROS project, and compile the PROS project for the robot. While somewhat clunky, the sample code is entirely using these pre-ROS-compiled files anyways, so this workflow would just add features onto the current system we would be using. This would allow us to link the rosserial repository directly into ours, by giving us a way to implement any changes in that repository into our robot code easily.

The ideal setup to us would be to run most code on the co-processor, while limiting the V5 to handling inputs and outputs from other ROS nodes. This would delegate our V5 hardware more so as a hardware wrapper, allowing us to run most of our code on a more accessible system where we have direct access to the OS. Through initial single motor testing, we experienced little to no latency in the communications, which seems promising. We will have to test this with more motors, as well as adding artificial bandwidth usage to strain the system. We weren't able to get any system throttling to occur with a single motor publisher, even when sizing up the publish rates and message sizes.

We started our implementation based off the template software provided to us. This included a simple function for publishing data to ROS from our robot. While this doesn't directly support the PROS structure, we made some small changes to publish basic motor data within the teleop loop present in the robot. We were able to get basic control of the motors pretty easily, except for the implementation wasn't scalable. The testing code was a mess, and didn't support larger robot configurations well. We will need to spend some more time refactoring this code on a larger scale, which will most likely take several weeks of designing and developing correctly.

We realized early on that our entire architecture would need to be built off of Nodes. Through abstracting different components of our robot into "nodes", it should allow for a high degree of modularity to allow most of our code to remain re-usable from year-to-year. Additionally, another benefit we identified in the node structure was it's inherent wrapping of hardware libraries, allowing for more robust testing to be done. By creating an abstraction of hardware function (such as motor and sensor commands), we can use these objects for testing as well (through dependency injection and special configurations of nodes). This can become very important as we start creating additional layers on top of these nodes, such as for autonomous, where we want to run tests to simulate our programming sequences. Being able to configure all of our base nodes to provide testing output instead of motor outputs, we can simulate autonomous routines without needing a robot.

9.7 Phase Development Introduction

The largest problem we've identified with having a wide-scale ROS roll out is the massive overhead of what we're trying to accomplish. In order to get our platform to entirely run on ROS, we need to essentially write wrapper classes for nearly all of the existing robot API functions provided to us by PROS. In pursuing this project, we would be introducing a massive amount of overhead as we are now responsible for writing the hardware abstraction layer functions as well, which need to be completed, tested, and debugged before even moving onto developing robot specific code. However, with several months before our first tournament, we felt that this was a risk we were willing to take in hopes of gaining the potential expandability in the future.

After some thought surrounding the project we are about to enter, we've decided to pursue this implementation in phases. We chose this instead of scrum-style sprints to allow for the team to be flexible with coursework and classes, instead focusing on larger phases that dictate some of the higher level deliverables that need to be accomplished. This will allow us to pursue iterative development over time, while allowing our students to focus on school when they need to as well. Phases will be defined incrementally, with each phase being defined before start. Since we don't know where we directly want to take this platform, we will save planning future phases for a time period closer to their respective start dates.

Phase 1 Overview

Within this first phase, we need to identify a basic roll-out plan for our nodes. To get our software to a point that will allow us to test on a more complex robot, we need to first create the wrapper classes for our basic actuators and sensors. Our primary goals in this development phase is to create the infrastructure and wrappers required to inject ROS functionality into a PROS build system. In a sense, all we need to do is re-write the hardware layer features provided by PROS within ROS-compatible wrappers. This phase in specific will allow us to test our implementation, and develop the building blocks of our framework as well.

Since ROS is built for publishing and subscribing to topics, we want to make all of our nodes have support for these features. Through looking through the ROS documentation and the previous ROSSerial examples for VEX, we've identified a basic workflow that our nodes need to follow:

- A node needs to create a node handle in order to interact with the ROS master
- A node can create 0 or more topics to publish to through the node handle
- A node can create 0 or more subscriptions to topics through the node handle
- Subscriptions are linked with a callback function, which will execute asynchronously when the ROS update is invoked
- Updating all ROS connections can be done one of two ways. ros::spin() provides a blocking and looping update mechanism, while ros::spinOnce() will be non-blocking, but will only provide a single update
- A node can interact with the robot or other systems outside of the ROS architecture

Through some design discussions between Nathan and Andrew, we've decided that the only feasible way to make our nodes interact with the system is through using the spinOnce() function to update all required callbacks and publishers. In our discussions, we also realized that nodes may exist on different tiers of importance, which is something we would like to support. For instance, real time nodes, such as our motor or sensor nodes, will need to be highly responsive on our system in order for us to perform with minimal latency and high accuracy. In contrast, different system nodes (like those to potentially read out system battery voltages) aren't as critical, and don't need to occur nearly as often. This discussion also opened up talks on how we want to structure our classes, and where inheritance should be present. We identified that we need some type of structure to make adding new nodes to the system easy, and that some level of inheritance should be used to simplify any of the shared ROS logic between nodes.

Our initial thoughts focus around a central Node class to handle most of the ROS logic. While not much will be abstracted away from the independent node implementations, inheriting this Node class will allow specific node implementations to get access to different management functions to handle node execution and invocation at specified intervals. This is where our NodeManager class would come in.

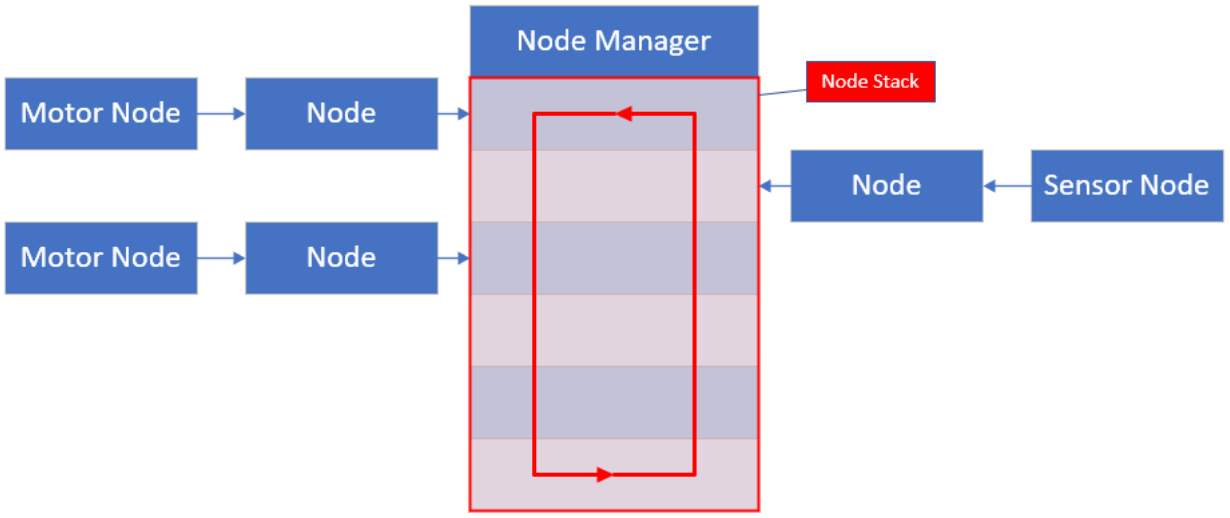


Figure 123: Initial NodeManager Structure

The NodeManager would hold references to all nodes currently on the system, as well as the execution interval for each. This loop would run non-stop on the robot, and nodes would only be activated on their specified interval, based on the difference between the last time they were executed, and the current system time. That way, nodes can be automatically queued and refreshed at whatever interval they need, and any node can be added to the centralized structure through the Node class. As it stands, Node will most likely serve as an interface unless we see a reason to expand it to abstract implementations away from certain nodes. For now, we thought it would be best to not focus on what to abstract before we even start implementing each of the constituent nodes.

For the implementation of the NodeManager, we think that each node reference on the system can be wrapped in a struct with an execution interval, and the previous execution time. Through storing a list of these structs, we can find nodes that are "due for execution" based on the difference between their respective last execution time, and the current system time. If a node is due for execution, we will have to invoke a method on the node class to run any ROS commands, and anything else that is time sensitive.

This doesn't quite solve the issue of getting each node ready for execution. We were stuck on whether an independent initialization method was needed or not, and decided that we can include it but not force implementation of it. However, all nodes will need to utilize the execution loop, so we will be making this pure virtual to enforce nodes creating loops that sync up with the NodeManager execution.

With the plan for NodeManager, we started talking about how to develop nodes for our different components, being our sensors and motors. We thought it would be best to implement nodes for all the currently available sensors and motors currently available in the PROS V5 API, to cover any future implementations we may need on the robot. One of the benefits to the nodes is that most of their functionality is for passing through values to specific hardware classes, making most of the behaviors emulate that of the PROS V5 functions.

Phase 1 Development

We have several newer members on the team this year, and these nodes provide a good combination of introducing the PROS system to them, while also giving them development time on our system code. In order to guide our development, we decided to start utilizing Trello, for Kanban-style development. We are hoping to use this to help give the necessary resources to our members (linking appropriate documentation and steps on each card) while also keeping track of what's still open to be worked on. To start out, we created tasks for each of the basic PROS V5 classes we may need and had our members self-assign onto tasks they were interested in working on.

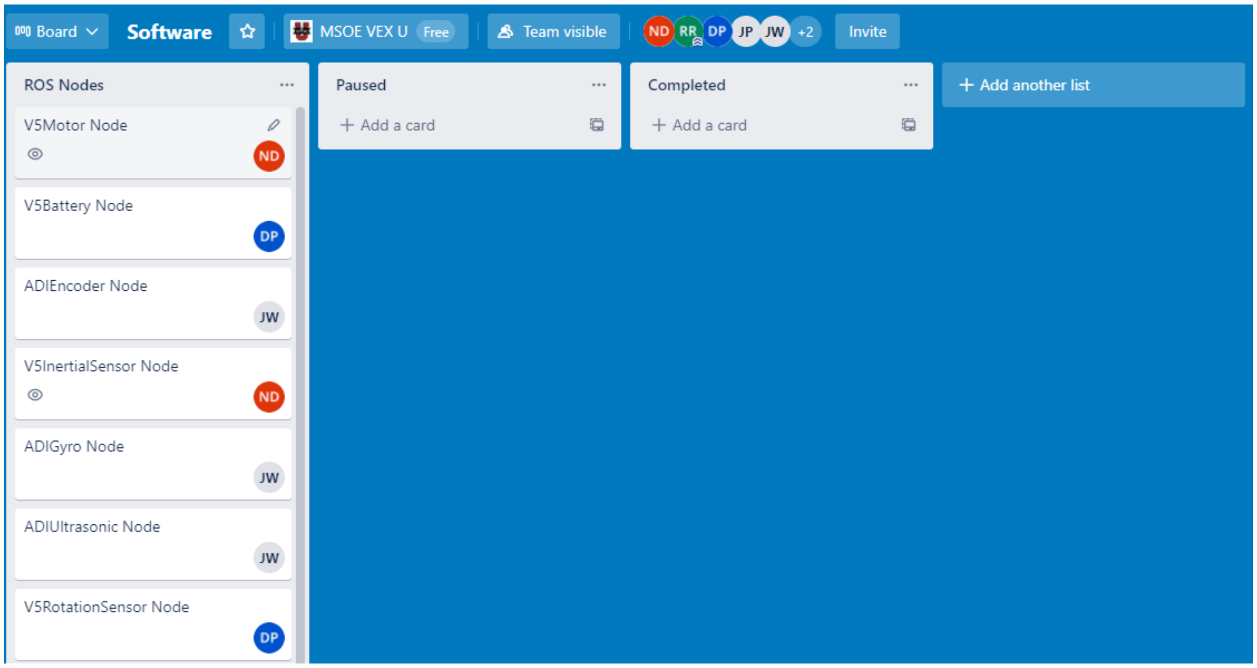


Figure 124: Trello Node Assignments

We were able to create basic implementations of each nodes without many issues. Many of the ADI (three-wire legacy) sensors were very similar, cutting down the development time for each. Once the nodes were close to completion, we started design conversations focused on how to make the ROS system fully function.

The one aspect that we haven't discussed much at this point is the implementation of the ROS Master Node, which handles the primary communications between nodes. We are currently finishing up implementing many of the hardware layer nodes that will be running on the robot, but in order for them to communicate properly, a device on the network needs to facilitate the sending and receiving of messages. This is illustrated with the diagram below:

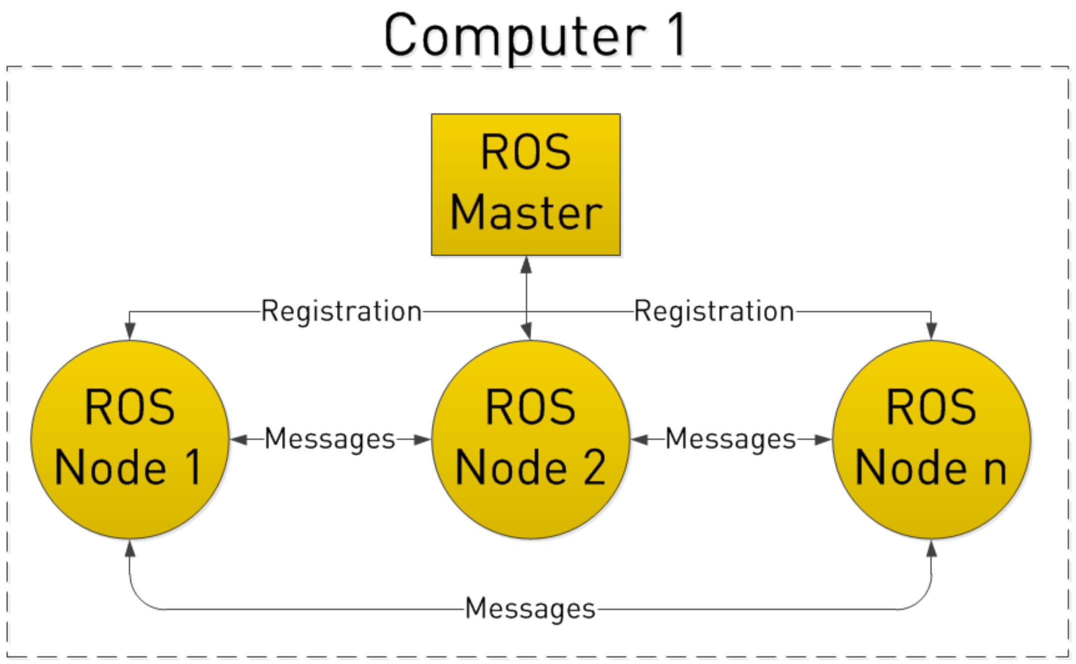


Figure 125: ROS Node Communications

As shown above, nodes need to perform their registration with the master node on the network in order to be properly initialized, and once initialized, can publish and receive messages on the network. Our current implementation only handles various nodes for the robot, and due to specific limitations with the VEX V5 hardware, our ROS master will need to be located on a co-processor.

As mentioned in our initial research section, we found some basic template code for nodes when first looking into the viability of ROS, which became the core structure of the nodes we're working on. This sample code is shown below:

```
/*
 * rosserial Publisher Example for VEX Cortex
 * Prints "hello world!"
 */

#include "ros_lib/ros.h"
#include "ros_lib/std_msgs/String.h"

// this loop is run in setup function, which publishes  at 50hz.
```

```
void loop(ros::NodeHandle & nh, ros::Publisher & p, std_msgs::String &
 ↪  str_msg, char* msgdata)
{
  str_msg.data = msgdata;
  p.publish( &str_msg );
  nh.spinOnce();
  pros::c::delay(20);
}

// The setup function will start a publisher on the topic "chatter" and
 ↪  begin publishing there.
void setup()
{
  // debug logging
  // make a node handle object, string message, and publisher for that
   ↪  message.
  ros::NodeHandle  nh;
  std_msgs::String str_msg;
  ros::Publisher chatter("chatter\0", &str_msg);

  // set up rosserial, and prepare to publish the chatter message
  nh.initNode();
  nh.advertise(chatter);

  // message data variable.
  char* msg = (char*) malloc(20 * sizeof(char));
  while (1) {

    // send a message about the time!
    sprintf(msg, "[%d] Hello there!!", (int) pros::c::millis());
    loop(nh, chatter, str_msg, msg);
  }
}
```

The most important parts here are the nh.initNode() function, as well as the loop() function. The initNode() function performs the initial registration with the ROS master, and the loop function is where the publishing/subscribing occurs. This sample code had the ROS library pre-compiled for the system, which gave us access to the NodeHandle functions needed to register our nodes.

This specific implementation used the serial out available on the V5 brain, as it was built as an adaptation of the public ROSSerial Library, available on GitHub. With this functionality built for us, we should be able to spin up a basic node on a Raspberry Pi (also using the default launch files provided in the samples) to allow the nodes to connect. The serial implementation uses an intermediate buffer to pull bytes off the serial lines, and transcribe them into ROS messages to be handled by the system. This driver also supports writing out messages, causing the messages to be written out when the spin() or spinOnce() commands are invoked on the NodeHandle object.

The NodeHandle object is templated with the specific driver implementation, allowing it to access the serial buffer through the V5Hardware class, as shown from the header file below:

```
#ifndef _ROSSERIAL_VEX_CORTEX_ROS_H_
#define _ROSSERIAL_VEX_CORTEX_ROS_H_

#include "ros_lib/ros/node_handle.h"
#include "ros_lib/V5Hardware.h"

namespace ros {
  typedef NodeHandle_<V5Hardware> NodeHandle;
}

#endif
```

We have completed the first of our nodes; namely our MotorNode has been implemented now. With this implemented, we can start working towards setting up the ROS connection. Deviating from the sample code, we decided to take an object-oriented approach to our nodes, to allow easy expandability for other systems of the robot, and to allow us to treat the ROS nodes similarly to how the PROS API treats hardware components (such as sensors and motors). We ideally want these to be stand-in wrappers for the PROS library objects, and making these object oriented allows for us to retain all of the previous functionality, while adding ROS support as well. Here is the basic code for each of the nodes:

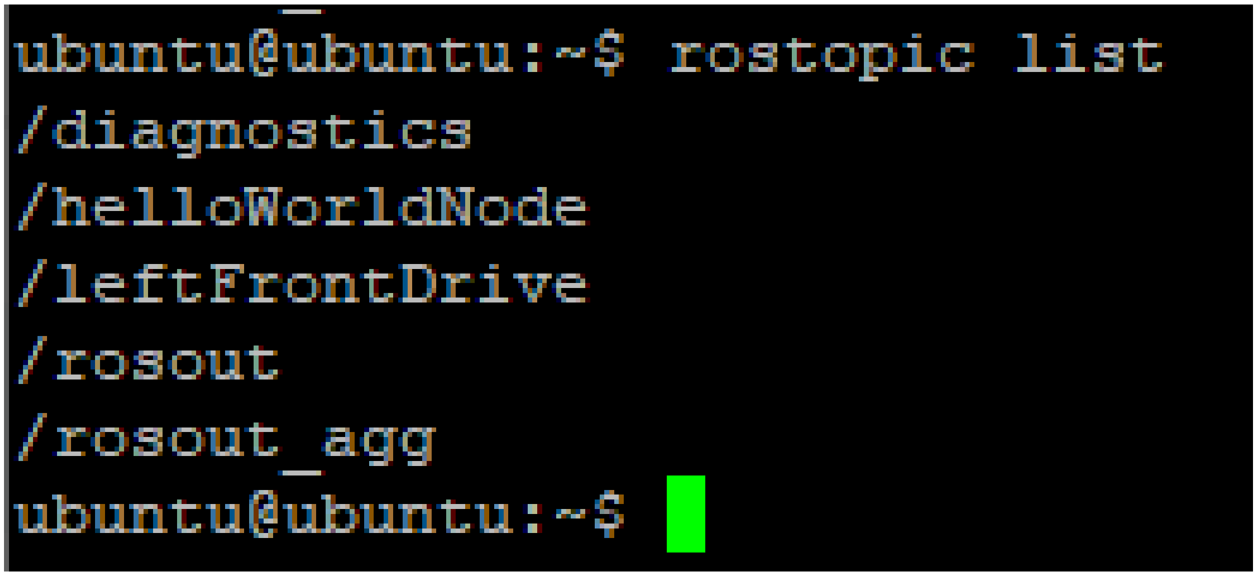
```
#include "MotorNode.h"

#include "DataNodes/MotorNode.h"

// By default, this constructor calls the constructor for the Node object in
 ↪ NodeManager.h
MotorNode::MotorNode(NodeManager* nodeManager, int portNumber, std::string
 ↪ handleName,
    bool reverse, pros::motor_gearset_e_t gearset):Node(nodeManager, 200) {
    m_motor = new pros::Motor(portNumber, gearset, reverse);
    m_motor_msg = new v5_hal::V5Motor();
    m_handle = new ros::NodeHandle();
    m_handle_name = handleName;
}

void MotorNode::initialize() {
    // Define a string handle for the motor
    std::string motor_handle = "Motor_" + m_handle_name;

    // Create a publisher with the custom title, and message location
    m_publisher = new ros::Publisher(motor_handle.c_str(), m_motor_msg);

    // Initialize the handler, and set up data relating to what this node
     ↪ publishes
    m_handle->initNode();
    m_handle->advertise(*m_publisher);
}

void MotorNode::periodic() {
    populateMotorMsg();
    m_publisher->publish(m_motor_msg);
}

void MotorNode::populateMotorMsg() {
    m_motor_msg->current_draw = m_motor->get_current_draw();
    m_motor_msg->direction = m_motor->get_direction();
    m_motor_msg->efficiency = m_motor->get_efficiency();
    m_motor_msg->is_stopped = m_motor->is_stopped();
    m_motor_msg->position = m_motor->get_position();
    m_motor_msg->port = m_motor->get_port();
    m_motor_msg->power_draw = m_motor->get_power();
    m_motor_msg->temperature = m_motor->get_temperature();
    m_motor_msg->torque = m_motor->get_torque();
    m_motor_msg->velocity = m_motor->get_actual_velocity();
    m_motor_msg->voltage = m_motor->get_voltage();
    m_motor_msg->is_over_current = m_motor->is_over_current();
    m_motor_msg->is_over_temp = m_motor->is_over_temp();
}

MotorNode::~MotorNode() {
    delete m_motor;
    delete m_motor_msg;
    delete m_handle;
}
```

After getting our MotorNode done, we have been having lots of issues getting the ROS system to communicate. For some reason, when running the default ROS project on our Raspberry Pi, we cannot get it to pick up the nodes on the robot. Instead of initializing the connection, it continuously times out from not receiving any data after a timeout period has elapsed. This is quite strange, as running just the sample code on the VEX V5 hardware works for us, leading us to believe it has something to do with our implementation. At this point, however, we are unable to get any of our nodes to communicate. We are concerned that we may also run into future issues with more than one node running on the system. We are trying to run multiple publishers and subscribers off of the same system, which should be able to be done on ROSSerial (as there are a max number of publishers and subscribers concurrent in the connection specified in V5Hardware.h), but have only seen implementations of hardware controllers that exclusively publish or exclusively subscribe to topics. Since our implementation is fairly unique as well in the VEX space, we have yet to find any other sample code with working ROS implementations. We will continue to debug the problem to find where our code deviates from the sample code. Our hope is to implement just the sample code with multiple publishers and subscribers to validate or invalidate our other concern, and if that works, walk the implementation towards what we currently have with MotorNode to see if/when it eventually breaks.

After even more testing, we were able to find one large bug that was contributing to the problem. Our original implementation of the NodeManager dispatch system was configured to run at the refresh rate of the system. Likewise, we didn't add a delay as we assumed the loop would be non-blocking since nodes only run when their elapsed trigger time is reached. This assumption turned out to be false, and was causing the system to be starved by NodeManager. We ended up finding this bug when trying to print out test statements inside of V5Hardware (to test which messages were being sent out). Even though the program was running, we were able to see that the scheduler was prioritizing NodeManager before all other tasks, and starving out other operations (including system IO), which was preventing data from being written to the serial buffer, and causing nothing to be sent to the ROS master. We were able to add a small wait onto the loop to prevent starvation and give the system time to schedule other tasks, while not impacting the execution time of any nodes.

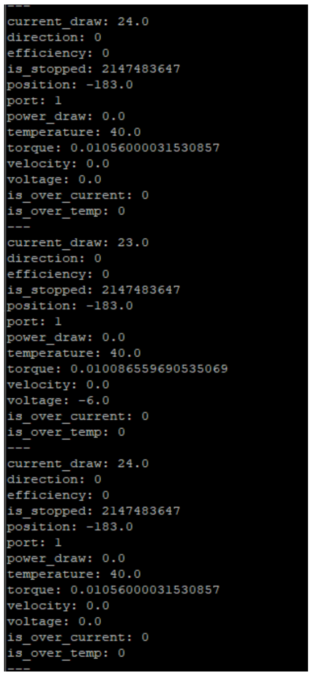
Another issue we ran into was linked to some random header file imports located within the serial code. For instance, we found two files within the serial driver implementation (V5Hardware.h and RingBuf.h) which included "main.h", causing a cyclical dependency on build. This didn't become an issue until we started adding more than one node for testing (for reasons that we are still unsure of). We believe that in our additional sensor nodes, we had utilized the V5Hardware class differently which helped us find this issue, allowing us to change it in the rosserial implementation.

After fixing some problems with our usage of pointers in the original node structure, fixing the errors in NodeManager that resulted in undesired behavior, and re-writing some of the serial driver files to fix the cyclcial dependencies and other read/write errors (such as disabling COBS), we were able to get the ROS master to successfully see and register multiple motor nodes, being hosted off of the V5 hardware! We were able to check the state of each node by monitoring the data it was publishing, specifically watching for the position field in the message to change while rotating the motor. This was a very positive sign, and we were able to scale this up to support 4 motors through our testing, which brings a potential chassis ROS implementation much closer.



```
ubuntu@ubuntu:~$ rostopic list
/diagnostics
/helloWorldNode
/leftFrontDrive
/rosout
/rosout_agg
ubuntu@ubuntu:~$
```

Figure 126: Robot Topics Registered with ROS Master Node



```
---
current_draw: 24.0
direction: 0
efficiency: 0
is_stopped: 2147483647
position: -183.0
port: 1
power_draw: 0.0
temperature: 40.0
torque: 0.01056000031530857
velocity: 0.0
voltage: 0.0
is_over_current: 0
is_over_temp: 0
---
current_draw: 23.0
direction: 0
efficiency: 0
is_stopped: 2147483647
position: -183.0
port: 1
power_draw: 0.0
temperature: 40.0
torque: 0.010086559690535069
velocity: 0.0
voltage: -6.0
is_over_current: 0
is_over_temp: 0
---
current_draw: 24.0
direction: 0
efficiency: 0
is_stopped: 2147483647
position: -183.0
port: 1
power_draw: 0.0
temperature: 40.0
torque: 0.01056000031530857
velocity: 0.0
voltage: 0.0
is_over_current: 0
is_over_temp: 0
---
```

Figure 127: Readouts on the MotorNode Topic

Currently, the main blocker we still need to pursue is a subscriber implementation. As it stands now, our nodes only have the ability to publish data to the ROS environment, not read in and act on data. Without this ability, we won't actually be able to control our robot. We held some additional research into this topic, and came across ROS ActionLib, which is a ROS package that supports control of feedback loops on another processor.

According to the wiki, ActionLib follows a general structure as defined in various action messages:

```
# Define the goal
uint32 dishwasher_id  # Specify which dishwasher we want to use
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

In this case, the robot acts as an ActionServer, and our coprocessor acts as an ActionClient. At the beginning of a specified action, our co-processor will send a goal message to our robot, telling it what to accomplish. Over time, the robot relays feedback messages on the performance of the action, and when complete, the robot can send one final, unique message back to the co-processor. This allows for the robot to asynchronously handle the feedback loop on the robot without needing constant supervision by an off-board processor. The message file above can be defined in such a way to support custom data fields as well, allowing us to define our own commands to send back and forth.

After running into issues with our custom action messages failing to be picked up by the CMake build, we did some searching and came across a GitHub issue thread that clarified that ROSSerial doesn't natively support actionlib, and if we wished to go down this route, we'd need to implement our own sort of actionlib structure. Instead of doing this, we did some additional research and decided upon implementing simple subscribers, and instead of using the actionlib structure, we would keep with using custom messages to pass what we needed. We didn't see much of a downside in this as well, as the same actions could be queued with both an actionlib message and a standard subscriber, and any feedback we need to send could be specified by an additional publisher on a new topic.

The generic code for a subscriber is as follows, provided from the ROS documentation:

```
#include <ros.h>
ros::NodeHandle nh;

void messageCb(const std_msgs::Float64& msg)
  if(msg.data > 1.0)
    digitalWrite(13, HIGH-digitalRead(13));   // blink the led
}

ros::Subscriber<std_msgs::Float64> sub("your_topic", &messageCb);

void setup()
```

```
{
    ...
    nh.subscribe(sub);
    ...
}
```

This code allows for a node to create a callback on a specific topic to a function reference. The callback receives the contents of the message that was published on the specified topic, and can execute data off of it. We think this should be a much more simple way of listening into topics in comparison to actionlib, and plan on focusing our efforts in getting MotorNode to subscribe to messages. Thanks to various ROS utilities available to us, we can publish our own messages on topics from our co-processor, without needing to link multiple nodes together. We hope to use this for integration testing in the future, allowing us to test systems without needing hardware and/or needing all systems active.

We quickly realized that our object-oriented approach to our nodes caused some issues with the subscriber sample code we were attempting to use. We found out that we could also template out the Subscriber with the class instance we were using (since we needed to use a method reference within a class instance instead of a function instance), along with passing a reference of the class that was to receive the callback on. The revised subscriber instantiation for MotorNode is as follows:

```
m_move_motor_voltage_sub = new ros::Subscriber<std_msgs::Int8, MotorNode>
        (m_sub_move_motor_voltage_name.c_str(),
         ↪  &MotorNode::m_moveMotorVoltage, this);
```

The method requires passing in a char* for the string, and since we were originally instantiating our nodes with a C++ std library string, we had to convert this over using the c_str() method (which returns a char* to the string).

While getting our subscribers to work, we also started exploring different possibilities for co-processors to run on our robot. We previously got access to both a Raspberry Pi 4, as well as an NVIDIA Jetson Nano. Seeing that the VEX AI Platform utilizes the Jetson as a part of it's control system (even powered from the V5 hardware), we started doing some research into hosting our ROS system on it. However, we found out that the current build of Jetpack, the GPU-enabled OS running on the Jetson, is built off of Ubuntu 18.x, which isn't compatible with the ROS version (Noetic) we were running on our development machines. We decided to instead go with the Raspberry Pi for now, until we could spend more time looking for a custom port of Jetpack based off Ubuntu 20.x, or created the docker infrastructure to compile the necessary ROS code on the Jetson. In the future we hope to expand back to the Jetson for GPU and CUDA capabilities for more computationally complex tasks, but for now, we aren't running enough load to make it viable.

Once the simple subscriber in MotorNode was created, we were able to run the default launch file on the co-processor to start up the ROS environment, and were able to successfully send commands to turn motors on and off. With this implemented, We were able to branch our software team out into two different areas:

- Working on a co-processor based state machine for motor control
- Adding subscribers to other nodes that required them

The subscriber implementation should all be straight forward, as the MotorNode code all works correctly. The state machine as well is built off of these same ideas, but since it's all running on the co-processor, we can write each Node on the co-processor as it's own file to be started on a different thread when the ROS server starts. This means that the co-processor implementation doesn't require any object oriented implementation, making the state machine very similar to the sample code provided previously.

After implementing a larger state machine on the co-processor, we were able to get motors to be controlled successfully by the input of a controller. However, we started running into problems as more nodes were added to the state machine. With larger numbers of nodes, the system wouldn't be able to sustain itself, and would start losing connection. We ran various tests with different nodes and messages, and have found that the primary problem is us reaching our serial implementation bandwidth limit, due to how the system handles incoming and outgoing messages through a temporary buffer. This buffer takes IO cycles to interact with (both incoming and outgoing), which tends to be too much for the system to handle when we're sending lots of data through.

We noticed that many of our nodes were set to publish extremely frequently, and could be slowed down slightly. This change allowed us to retain communications for longer periods of time (with some intermittent communication crashes), but also added intermittent latency to the system between the controller sending data, and the motor receiving data. We also realized that the custom messages we had created for most of our nodes were excessively large; we had built them off the default return types of the PROS API functions, which often return int32 values which can be contained within int8 or even bool types. Other messages, such as our custom message for MotorNode, included every piece of data returned by the motor. While convenient for the future, a large majority of this data will likely never be utilized, and can be scrubbed from the messages. In order to make the system easier to work with, we moved away from custom messages in many areas and transitioned to using the default message types in most cases. This added some slight stability improvements, but wasn't enough to stabilize the system entirely under low latency configurations.

After the problems we were facing with the serial interface on the V5 hardware, we did some research on other avenues of communication from the V5 hardware. Unlike the VEX Cortex that precedes it, the V5 hardware doesn't have many openly accessible communication protocols to use. Doing some digging in how others are communicating with the hardware, we came across a VEX Forum Post that discusses using the smart ports as generic communication devices. Each of the smart ports operate off of RS-485 protocol, which supports higher communication speeds than we're able to achieve over serial. Additionally, each of the ports has a hardware buffer that data is automatically stored into, and becomes accessible by the system without needing to use clock cycles to load it. This form of communication addresses many of the problems we're currently facing, and seems like an ideal solution to our problems.

In a paper published by both the BCUZ and QUEEN VEX U teams, they talk about communicating with the V5 hardware over RS-485. The RS-485 protocol is a simplex structure, where one line is needed for sending, and one for receiving. This would require the use of two separate V5 ports, and would function much differently than the full-duplex serial line we are currently using. We found this interesting, but for the time being needed to focus on implementing code for some initial swerve drive modules we were testing with. We passed this documentation onto Andrew (due to his electrical engineering background) and continued on with development.

Andrew ran some initial tests with some RS-485 to Serial boards he had on hand, and wasn't able to get communications to cleanly be handled by either side of our network. The boards we were using didn't support handling both an input and output channel, so we made some modifications to the PCB to convert both channels into a single serial out. However, we found that the data we were sending wasn't being properly taken from the hardware buffer on the V5, as our data would start overflowing before it was sent. This would cause an improper packet structure when sending to the ROS master node, which would subsequently force the entire packet to be thrown out. From these tests, we are looking at potentially making custom boards to handle this communication cleanly, to test if the problem was due to how we implemented the communications.

Since we weren't able to get this to quite work, we are moving forward with using serial communications for ROS. Another problem with our implementation that we've identified is that our nodes are always publishing data; regardless of if it's needed or not. A future fix for this would be to have nodes subscribe to a specific "publish" topic, which would invoke their publish method. This work around isn't ideal, but would make it easier to work with.

Some of the core problems we've experienced that need to be noted:

- We had issues with custom topic names. Using certain topic names resulted in an invalidated checksum when received by our co-processor, which then denied to read the message
- Even with null terminated strings as the message handle, the checksum didn't match up
- Problem with the logging on the co-processor not having the correct topic name
- Problems with C++ string scoping since topics need to reference the topic name even after initialization (takes in a char*; since c++ string went out of memory it didn't have a valid name after initialization)
- Multiple topics working and updating live
- Use a higher frequency to diminish impact of failed communications (bad messages will be replaced with good data in one of the next cycles that works)

Phase 2 Overview

With the basic framework of ROS implemented, we are able to start working on robot-specific implementations. In this phase, we will focus on implementing code for our competition robots, and adding other nodes for autonomous.

Phase 2 Development

On our first competition robot, we had decided to go with differential swerve due to it's high maneuverability and power output. The code behind differential swerve has decent complexity, and in order to align our software team on how to implement it correctly, we held several discussions on the math required to compute motor powers for the swerve module.

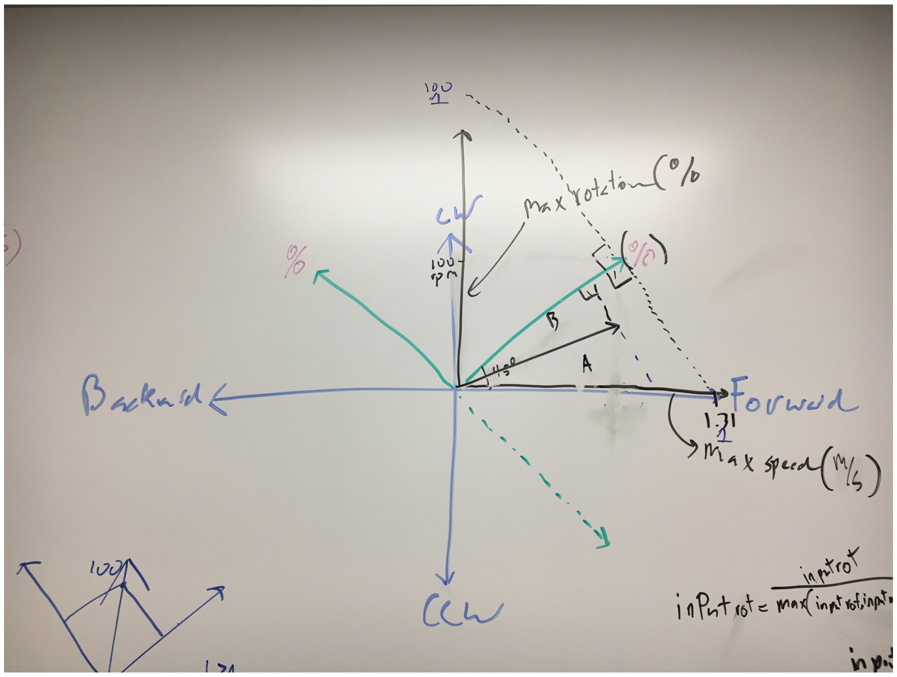


Figure 128: Initial Swerve Math Discussions

Differential swerve is a unique drivetrain, as not only does each wheel spin independently, but each motor can also contribute towards linear movement of the robot and/or rotational movement of the wheel pod. For more information about the differential swerve, please reference our drivetrain design section.

The figure above shows a plot of how our motion exists in 2D space. Essentially, depending on how each motor is rotating relative to the other, a single motor can have 100% of it's power contributing towards module rotation, 100% of it's power contributing towards linear wheel rotation, or some proportional combination of the two. In order for this to work properly, we need to utilize custom kinematics code to map from specified x, y, and theta velocities to specific motor powers. This process is called Inverse Kinematics, and in our specific implementation, it requires the following steps:

- Find the position of each module with respect to the center of the robot
- Calculate a rotation vector of each module by finding a vector tangent to the module, multiplied by the target rotation velocity
- Sum the target velocity and rotation vectors for each module to determine the overall target movement vector
- Find the angle of the target vector and calculate the error from the current module position
- If the angle is larger than 180 degrees, roll over to make the module turn in the opposite direction
- If the angle is larger than 90 degrees, exclusively send power to turn the module (to correct faster)
- Calculate motor power from a PID based on the error in angle
- Break the desired power into rotation and linear components, and scale them to the maximum motor values
- Send the values to the motors

As mentioned above, there is a lot of vector math that's being utilized in this system. Vector math allows us to easily incorporate multitudes of different drive configurations (such as using 3 swerve modules instead of 4). For this to work properly, we started looking for libraries that could help us accomplish this math easily, which led us to the Eigen C++ library. We were able to add this library into compilation as a header-only include, allowing it to easily compile into our robot code on the co-processor.

We quickly realized through testing that the internal feedback loop on each module (the embedded PID above) caused lots of issues due to the latency still present in the ROS system. When testing, compounding latency errors between the sensors, motor inputs, and motor outputs made the module almost unusable, as we had nearly no reliable control of the module. We were able to analyze what it was doing and confirm that the logic made sense, be lag in the system caused the module to constantly over correct and failed to perform as we needed it to. These problems with the system invoked us to do some additional research on how to handle feedback loops similar to this.

Through much of our research, we've found that implementing ROS on larger robotics systems utilizes sets of external processors, such as Arduinos, ESP-32s, or Raspberry Pis to handle much of the hardware control, while allowing the master node to handle larger tasks such as storing and updating information from a robot's environment. This style of implementation is also a similar mirror to the control structure in FRC, where the primary controller (the roboRIO) facilitates much of the communication and higher level control, while allowing specific motor controllers to run integrated feedback loops on the devices that they're controlling.

In this sense, we can use ROS to handle more abstract controls (such as state machines and environment-dependent control), while delegating specific motion control tasks (such as PID motion control) to the V5 in order to reduce latency in the feedback loop.

Due to this, we decided to offload the swerve control code onto the V5 hardware. Since Eigen is a header only library, it was very easy to move over and just required adding it in our includes. The same logic was able to be ported over nearly identically from our co-processor implementation, and initial tests showed a vast improvement in the system. We were able to run three modules with little latency in the system, and were able to see the logic working correctly.

We worked to incorporate ROS plotting tools into the system so we could accurately tune the PID on the system, using the messages published by the robot to allow it to plot on a computer nicely. Through our testing, we also found that due to internal inefficiencies in the swerve modules, and due to a non-ideal PID, more of our energy was being spent turning the module than powering it linearly. This caused our robot to be excessively slow and hard to control, and with our tournament approaching quickly, we jumped to a backup plan of implementing a tank drive on our robot.

During this time, we also had some issues with code quality in our repository, especially due to the quick change in direction of the robot. To help rectify this, we added a short CI check to confirm our code could compile before allowing it to push to the repository. Since all of our code is open-source, our CI runs through GitHub are entirely free to run, allowing us to use it regularly in our repository.

Due to our ROS architecture, a new drivetrain node for our Tank Drive was extremely quick to implement, using our MotorNode class as a wrapper to the pros Motor class. Our first implementation of drive code was developed concurrently to some updates we were making to our hardware nodes, which resulted in the code conforming to older node formats and creating a highly coupled system. For instance, this was our constructor to the original class:

```
DriverControlNode::DriverControlNode(NodeManager* node_manager, MotorNode*
 ↪  left_swerve_1, MotorNode* left_swerve_2,
        ADIAnalogInNode* left_swerve_pot, MotorNode* right_swerve_1,
         ↪  MotorNode* right_swerve_2,
        ADIAnalogInNode* right_swerve_pot, MotorNode* rear_swerve_1,
         ↪  MotorNode* rear_swerve_2,
        ADIAnalogInNode* rear_swerve_pot, MotorNode* left_intake, MotorNode*
         ↪  right_intake, MotorNode* bottom_rollers,
        MotorNode* ejection_roller, MotorNode* top_rollers,
         ↪  InertialSensorNode* inertial_sensor, ControllerNode*
         ↪  controller_primary) : Node(node_manager, 20),
    swerveController(left_module_location, right_module_location,
     ↪  rear_module_location, rotation_angle_threshold,
    max_velocity, max_rotation_velocity, kP, kI, kD),
     ↪  left_swerve_1(left_swerve_1), left_swerve_2(left_swerve_2),
    left_swerve_pot(left_swerve_pot), right_swerve_1(right_swerve_1),
     ↪  right_swerve_2(right_swerve_2),
    right_swerve_pot(right_swerve_pot), rear_swerve_1(rear_swerve_1),
     ↪  rear_swerve_2(rear_swerve_2),
    rear_swerve_pot(rear_swerve_pot), left_intake(left_intake),
     ↪  right_intake(right_intake), bottom_rollers(bottom_rollers),
    ejection_roller(ejection_roller), top_rollers(top_rollers),
     ↪  inertial_sensor(inertial_sensor),
     ↪  controller_primary(controller_primary->getController()) {
    left_module_location(0) = left_module_location_x;
    left_module_location(1) = left_module_location_y;
    right_module_location(0) = right_module_location_x;
    right_module_location(1) = right_module_location_y;
    rear_module_location(0) = rear_module_location_x;
    rear_module_location(1) = rear_module_location_y;
    m_navx_sub = new ros::Subscriber<v5_hal::RollPitchYaw,
     ↪  DriverControlNode>
        ("/navx/rpy", &DriverControlNode::m_navxDataCallback, this);
}
```

As is quite obvious, this is terrible. Our DriverControlNode became the main source of everything on the robot, which completely defeats the purpose of our Node architecture. To start fixing this, we broke this code into a TankDriveNode to handle the drivetrain (replacing the swerve code above), and a ConveyorNode to handle the ball feeding/processing system. This cut the node nearly in half, making it much more manageable for multiple people to work concurrently. This shift wasn't super hard, and the benefits of the node structure were finally becoming apparent.

While working on the ConveyorNode, we worked with the design team to add several line follower sensors along the conveyor to act as proximity sensors for balls. When a ball passes nearby a sensor, it will detect a change and allow us to detect the presence of the ball. Using three different followers, we were able to fully automate the storage in our conveyor. Using two sensors at the bottom and one at the far top, we were able to queue balls in one at a time, until one reached the top sensor. When the top sensor was activated, all other balls were stopped to prevent us from spitting out the ball early. This automation both helped our driver, as well as made it easier to control in autonomous.

During this time, we also realized that our drivetrain wrappers didn't support velocity control (and only voltage control), which prompted Dylan to add pass-through methods to the underlying pros Motor classes.

With the new drivetrain in place, we once again split our software team apart to work on several core pieces of need. First, we delegated several people to start working on an auton scheduling mechanism Andrew had explained based on some of his previous FIRST experience. Additionally, we had several others start working on implementing a NavX node on our co-processor, to send NavX inertial data from the Raspberry Pi (over USB) to the V5. The NavX normally isn't compatible with the VEX system, but due to our ROS implementation, we believe it will be possible to pass these values to the robot. Since our bandwidth usage is near to nothing now with most code being off loaded onto the V5 hardware, we don't see any bandwidth issues with the sensor.

Our auton scheduling system works off the concept of node trees. Essentially, every autonomous routine starts with the primary trunk node, and continues branching out into one or more nodes of execution. The autonomous is based down a single "main line" of nodes that are executed sequentially, and parallel "leaves" can be added to any single node on the main line to be executed concurrently. For instance, if our main line nodes include a "drive forward" and "turn" node, we can make our intake turn on concurrently to driving forward, and turn off concurrently to turning. This allows for more complex structures of scheduling that would be very hard to manually control. This system also handles automatic memory management based on the first node, where deleting the first node will automatically delete all linked nodes in the tree. Finally, the autonomous system has automatic node scheduling under the surface, with incorporated node timeouts and expirations based on what actions are being performed, and what the constraints on each are.

A decent way into our testing, we realized that our robot was jerking during drive movements due to sudden acceleration and deceleration. This was causing our routine to get misaligned over time, and prompted us to add basic trapezoidal motion profiling to our drive movements. This trapezoidal motion profiling was used to increase and decrease the velocity in accordance with the max acceleration of the robot, and where it was along it's motion. This includes slowing the robot down before reaching the end of it's path, looking ahead to see when the robot is supposed to stop in order to slow it down in time.

As we were working on the programming skills, we also ran into problems with dissimilar code between our teleop and autonomous code causing our nodes to become overly complex. The original node structure we had implemented a single-execution initialize loop, and a continuous periodic loop to execute in the node. This single periodic loop was called both in teleop and autonomous, causing complex logic to form to dictate whether the node should be in driver control state or autonomous state. We found a more robust fix to this problem to be implementing both an autonPeriodic and teleopPeriodic loop to the system, allowing each to only be called when in their respective mode. This made it much easier to manage and control state machines separately in autonomous and teleop, and made automation easier to handle.

Phase 3 Overview

Our first tournament went well, but due to some issues with recording and a battery running out mid-run, we weren't able to submit a programming skills score. Our final routine was estimated to score over 50 points, and while low, was decent for only working on the routine for several hours.

Reflecting on the tournament also helped us identify a few areas of weakness in our system, including:

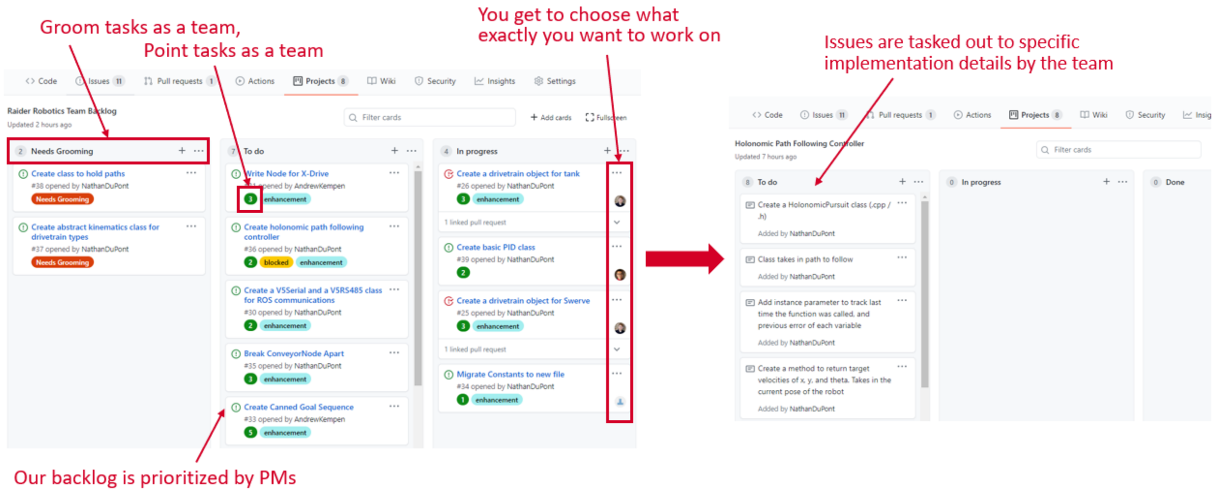
- Lack of sequencing of AutonNodes made for massive programming skills files. Since all of the auton nodes operate off pointers, we can create sequences in linked lists or structs, as we only need access to the first and last pointers
- Lack of sideways movement complicated the programming extremely. We weren't able to generate paths for the robot to follow due to not having enough time to fix our adaptive pure pursuit controller implementation, and being off on position was hard to correct
- While breaking out TankDriveNode and ConveyorNode was good, ConveyorNode is becoming bloated and should have all front-intake features broken out

Phase 3 Development

Working with the mechanical team, we decided that we wanted to iterate our robot and build a second version with holonomic capabilities. This feature should allow us to have much more control over the robot, and make it easier to use kinematics and odometry to figure out where we are on the field. Since our holonomic drive (in specific, X-drive) would also require forward and inverse kinematics, we plan on breaking out the kinematics class into reusable modules that can be added into any drivetrain object.

We also spent lots of time reworking our development process. Our previous approach mirrored a waterfall mentality, which caused bigger issues down the line as we got further into development. Over the first week of this phase, we started moving away from this waterfall base and from Trello as a whole, instead moving to GitHub issues. We started holding "grooming"/"refinement" meetings to get everyone aligned on what needed to be done for our issues, and started to add story points to our issues to make it easier for new members to jump in and work on stuff.

During this time, we also held a 45 minute presentation to our school's Society of Software Engineering chapter, focused on our development technology, robot software, and system architecture. In this presentation, we revealed our new project management tools, as discussed above:


Groom tasks as a team,
Point tasks as a team
You get to choose what exactly you want to work on
Issues are tasked out to specific implementation details by the team
Our backlog is prioritized by PMs
Code
Issues 11
Pull requests 1
Actions
Projects 8
Wiki
Security
Insights
Settings
Filter cards
Add cards
Fullscreen

Raider Robotics Team Backlog
Updated 2 hours ago

2 Needs Grooming
- Create class to hold paths — #38 opened by NathanDuPont — Needs Grooming
- Create abstract kinematics class for drivetrain types — #37 opened by NathanDuPont — Needs Grooming

7 To do
- Write Node for X-Drive — #31 opened by AndrewKempen — 3 enhancement
- Create holonomic path following controller — #36 opened by NathanDuPont — 2 blocked enhancement
- Create a V5Serial and a V5RS485 class for ROS communications — #30 opened by NathanDuPont — 2 enhancement
- Break ConveyorNode Apart — #35 opened by NathanDuPont — 3 enhancement
- Create Canned Goal Sequence — #33 opened by AndrewKempen — 5 enhancement

4 In progress
- Create a drivetrain object for tank — #26 opened by NathanDuPont — 3 enhancement — 1 linked pull request
- Create basic PID class — #39 opened by NathanDuPont — 2
- Create a drivetrain object for Swerve — #25 opened by NathanDuPont — 3 enhancement — 1 linked pull request
- Migrate Constants to new file — #34 opened by NathanDuPont — 1 enhancement

Holonomic Path Following Controller
Updated 7 hours ago

8 To do
- Create a HolonomicPursuit class (.cpp / .h) — Added by NathanDuPont
- Class takes in path to follow — Added by NathanDuPont
- Add instance parameter to track last time the function was called, and previous error of each variable — Added by NathanDuPont
- Create a method to return target velocities of x, y, and theta. Takes in the current pose of the robot — Added by NathanDuPont

0 In progress

0 Done

Figure 129: GitHub Backlog from Phase 3

While the design team is working on the robot, we started getting ahead on some work we will need to do. First, we had some programmers pick up tasks to break the ConveyorNode apart in order to separate the intake out, and make it easier to work with. We additionally started work on the kinematics for the holonomic drive, and the base drive node. For this tournament, we also started ramping up development on our WebDashboard, a JavaScript web application that allows for dynamic path generation. These paths would then be saved on our robot's SD card, and parsed into various point objects.

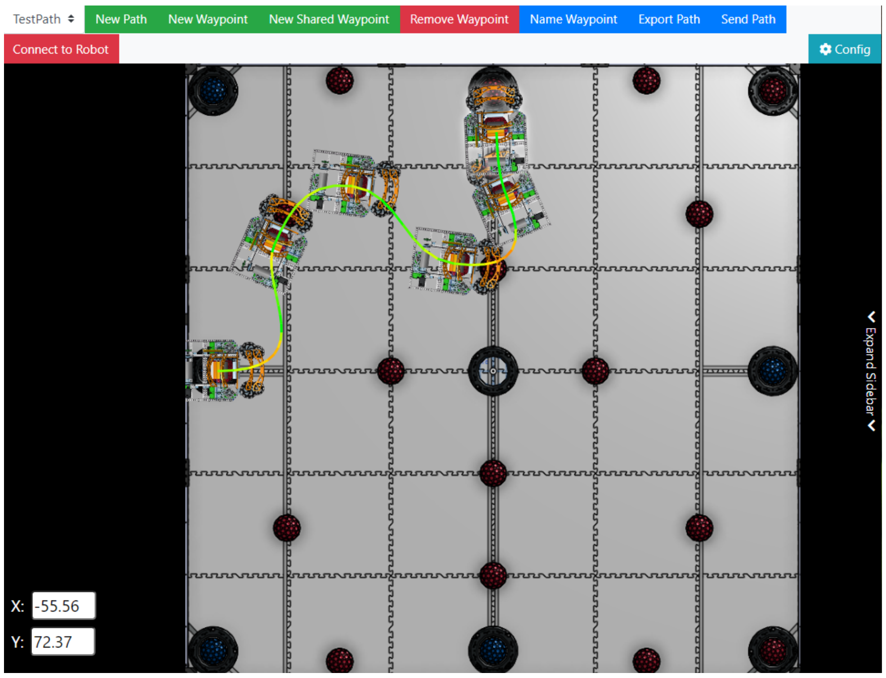


Figure 130: WebDashboard Application

The web interface works for basic paths, but doesn't fully support holonomic drivetrains. In addition, the paths are currently entirely distance parameterized, which isn't ideal for a time-based control loop. In order for the generated paths to be usable on the robot, we need to add rotational control of the robot, fix the motion profiling done on the dashboard, as well as add some speed configurations for different waypoints.

During the last week, we focused our efforts between adding the required holonomic nodes on the robot, and modifying the dashboard. We spent a few hours converting the dashboard to Typescript using a custom Webpack toolchain, but ran into some issues near the end that didn't allow it to be completely usable. We continued development on the JavaScript version, and are currently working through some math problems with calculating angular change over time. The time parameterization of points was completed successfully, and base functionality for tank drives should be complete.

During this time, the holonomic kinematics were also completed with the holonomic drivetrain, which now allows us to specify drivetrain movements based on x, y, and theta values, which is then parsed by the kinematics into motor movements. This is a great step towards higher abstraction in drivetrains, and we will soon be revisiting our old drivetrains to refactor them to match an interface similar to this.

The ConveyorNode was also broken apart into separate classes (namely adding IntakeNode), making it much easier to work with).

9.8 CI

Through some of our member's experiences in software companies in the area, we identified that we wanted to create some basic CI workflows to allow us to maintain high code quality, as well as give our new members confidence in pushing to our repository. In such a fast-paced environment, we understand that it can be stressful to push code with the fear of breaking something further down the line. Because of this, we want to create a set of CI workflows that run basic checks on the code, making sure that any code that gets pushed onto our main branch(es) won't break when testing, requiring other developers to go out of their way to hunt down bugs.

CI, or Continuous Integration, is a philosophy for fast-moving software development focused on continuous merging and progress of software systems. In order to sustain this fast-paced progress, software checks and tests need to make sure that a minimum standard of code is upheld when merging code, preventing other programmers from being blocked by bugs introduced on different branches. Read more about CI here.

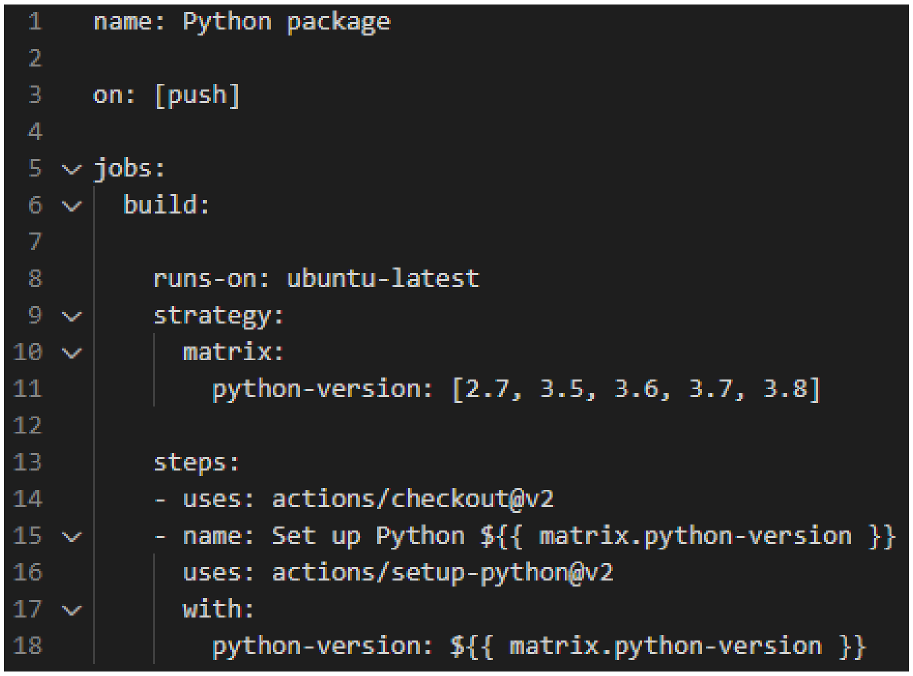
There are many forms of CI workflows that can be used to improve the quality of software development for engineers, as well as code quality committed to a repository. To help our repository become better, we brainstormed a list of potential usages of CI that we have experiences with, for use in picking what we want to invest time towards bringing into our code base.

- CI can be useful in making sure that code can be built and compiled without issues. While it seems like a simple check, it can be extremely easy to forget to make sure recent changes, especially if they are small, still compile correctly. However, even a small misspelling can cause a compilation error that will break on our main branch, costing more time down the line to trace the problem down and fix.
- CI can help in running automated tests on subsystems; if the code is structured correctly, we can write tests for a specific subsystem, and have tests for every subsystem run whenever code is to be pushed into the repository. This can help developers know if their changes affected certain subsystems, or even made them break without their knowledge.
- CI can help us produce executable packages or libraries for use in the code. Using utilities such as Conductor Packages for PROS, we could have CI workflows automatically create libraries for our code base, which just need to be updated on the user end to get the latest code.
- CI can help enforce good style in code, and perform basic linting checks on the code. In order for people to push code to the repository, they could be required to pass all linting checks (to make sure they follow a common style guide), working to make our code much cleaner, readable, and easier to work in.
- In many software platforms, CI can integrate with code reviews and merge requests as well. Using CI can help make code reviews quicker (make it so we don't need to look for compile issues when reviewing - only style and logical issues), and give us additional checks for confidence that we are producing good code.

Since our program has all of our code hosted in GitHub, we started doing research into GitHub Actions; a system produced by GitHub to allow for CI and CD (Continuous Deployment) workflows to be automatically run. GitHub Actions allows for usage of public actions (produced by other users and posted in a marketplace) as well as custom-defined actions, placed within a ".github/workflows" directory in the repository and written in YAML (.yml or .yaml) files. Full Documentation on GitHub Actions is located here.

Actions are run on "runners", which are small compute instances spun up on GitHub's servers, or on a machine you own and provide entry-points to within GitHub (so the program can start runners on your machine). Normally, runner compute time is one of the largest costs for hosting CI/CD workflows. However, GitHub allows for all public repositories to host unlimited workflows with unlimited time; since we are running a public repository, this seems like an excellent route for us to continue down.

As mentioned before, Actions are defined within a YAML file. Similar to other "semi structured" text representation languages (such as JSON or XML), YAML defines data through sets of defined tags, and associated indentations. An example YAML file (provided by GitHub) is located below:



```yaml
name: Python package

on: [push]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [2.7, 3.5, 3.6, 3.7, 3.8]

    steps:
    - uses: actions/checkout@v2
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v2
      with:
        python-version: ${{ matrix.python-version }}
```

Figure 131: Example YAML File for Python

The file contains several components. They will be detailed below:

name: This tag is located at the top of the file, as it names the workflow/action you are creating. This name will be displayed when running it in the GitHub UI.

on: This tag takes in an array of *triggers*. These triggers are actions that will be performed by programmers with the repository, such as pushing code to a branch, creating pull requests, etc. A complete list of triggers can be found here.

jobs: Most of the workflow will be defined underneath this tag. Specifically, the jobs tag it used to define what will be run, and on what platform. This is defined based on the tags that are indented below it; all tags that are shown in the picture, and are indented below the "jobs" tag are used to parameterize and configure the jobs being run.

build: The naming of this tag isn't as important as how it's defined. The tag(s) at the first indentation indicate one or more subsequent jobs to run on a runner. These jobs can consist of any actions you define, such as building code, running tests, or deploying to remote servers/locations.

It is important to note that jobs have **minimal** communications with each other, as each job is designed to be completely isolated. There are ways to pass information between jobs, but highly coupled tasks should often be put into steps of a single job.

runs-on: This tag defines which platform, or image to create the runner in. One large benefit to CI is it's quick; being quick, it requires setting up and tearing down run-time environments for short bursts, with high repeatability. GitHub has several different tools to aid in this; runners can be quickly started in multiple Linux distribution versions, as well as creating runners from Docker Images available in Docker Hub.

This will be very important for us. Our ROS infrastructure requires lots of dependencies, and may be a problem for quick runner execution. Docker should allow us to quickly start up runners for testing using the official ROS Docker Images (located here), but will need to be tested and custom configured to our exact environment requirements.

steps: The steps tag defines multiple subsequent steps that run in the same runner environment. This allows for sharing resources between different steps, in a way that isn't available across multiple jobs. Steps are primarily defined with a **name:** tag to tell the user what the step is doing, an **uses:** tag to define a public action to use (from the GitHub actions marketplace), or a **runs:** tag to run shell commands directly inside of the environment.

This will become useful when running our build commands inside of an environment. Since we have custom build commands defined for our project, we should be able to call them from within the project directory in the environment to simulate building the project.

While some of the tags in the image weren't discussed, these can all be found in some detail within the GitHub Actions Documentation, along with other examples and links to external sources. The list above describes most of the tags we feel we may need, as well as ideas and concerns for different steps of the implementation.

It is also worthwhile to note that many current actions seem to run within TypeScript files, running in Node.JS. These allow for a higher level of complexity than is available within the YAML files, and is very helpful in defining custom actions. While it will become beneficial to create our own actions in the future to easily package these workflows into single commands, it seems best to start out with defining everything within the basic YAML structure for simple workflows, for now.

9.9 Code

This section will include the main code files running on our robot. It isn't feasible to include all of the code running on our robot in this section, so instead we will just include some of the main files responsible for controlling our robot, to share how a ROS implementation of VEX code looks:

NodeManager

NodeManager is responsible with scheduling, creating, and handling all of our ROS nodes under the covers. It features an abstract Node implementation to handle all communications with the NodeManager.

```
#NodeManager.h
#pragma once

#include <vector>

#include "api.h"
#include "ros_lib/ros.h"
#include "util/Constants.h"

class Node;

// The NodeManager class handles multiple things:
// 1) What nodes need to be called
// 2) When each node needs to be called
//
// Nodes are automatically added to the manager on creation as long as they
// inherit from the Node class below. This means you never should be
 ↪  calling
// addNode() explicitly!
class NodeManager {
private:
    struct NodeStructure {
        Node* node;
        uint32_t trigger_millis;
        uint32_t last_executed_millis;
    };

    std::vector<NodeStructure> m_node_structures;

    uint32_t(*m_get_millis)(void);
```

```
protected:
    ros::NodeHandle* m_handle;

public:
    NodeManager(uint32_t(*get_milliseconds)(void));

    ros::NodeHandle* addNode(Node* node, uint32_t interval_milliseconds);

    void initialize();

    void reset();

    void executeTeleop();

    void executeAuton();

    ~NodeManager();
};

// The Node class is the parent object of all Nodes on the robot. It
↪  outlines
// what a node should have, and gives us a common interface on how to
↪  interact
// with nodes.
//
// The constructor of the node object takes in a pointer to the node
↪  manager,
// which AUTOMATICALLY ADDS IT to the manager on creation. This means that
↪  you
// don't need to add nodes on your own!
//
// The interval at which a node is called is set within the Node's CPP file,
↪  in
// the superclass constructor (should look like :Node([manager], [time]))
class Node {
public:
    Node(NodeManager* node_manager, uint32_t interval_milliseconds) {
        m_handle = node_manager->addNode(this, interval_milliseconds);
    }

    ros::NodeHandle* m_handle;

    virtual void initialize() = 0;
    virtual void teleopPeriodic() {}
    virtual void autonPeriodic() {}
    virtual ~Node() {}
};


#NodeManager.cpp
#include "nodes/NodeManager.h"

NodeManager::NodeManager(uint32_t(*get_milliseconds)(void)) {
    m_get_millis = get_milliseconds;
    m_handle = new ros::NodeHandle();
}

ros::NodeHandle* NodeManager::addNode(Node* node,
    uint32_t interval_milliseconds) {
    NodeManager::NodeStructure node_structure = { node,
     ↪  interval_milliseconds, 0 };
    m_node_structures.push_back(node_structure);

    return m_handle;
}

void NodeManager::initialize() {
    m_handle->initNode();

    for (auto node_structure : m_node_structures) {
        node_structure.node->initialize();
    }
}

void NodeManager::reset() {
    for (auto& node_structure : m_node_structures) {
        node_structure.last_executed_millis = 0;
    }
}

void NodeManager::executeTeleop() {
    m_handle->spinOnce();

    for (auto& node_structure : m_node_structures) {
        auto current_time = m_get_millis();
        if (current_time - node_structure.last_executed_millis >=
            node_structure.trigger_millis) {
            node_structure.node->teleopPeriodic();
            node_structure.last_executed_millis = current_time;
        }
    }

    pros::c::delay(DELAY_TIME_MILLIS);
}

void NodeManager::executeAuton() {
    m_handle->spinOnce();

    for (auto& node_structure : m_node_structures) {
        auto current_time = m_get_millis();
        if (current_time - node_structure.last_executed_millis >=
            node_structure.trigger_millis) {
            node_structure.node->autonPeriodic();
            node_structure.last_executed_millis = current_time;
        }
    }

    pros::c::delay(DELAY_TIME_MILLIS);
}

NodeManager::~NodeManager() { m_node_structures.clear(); }
```

MotorNode

MotorNode is the original ROS node we created, and has served as the basis for many of the other Actuator and Sensor nodes in our system. It serves as the primary wrapper to the pros Motor class.

```
#MotorNode.h
#pragma once

#include "nodes/NodeManager.h"
#include "api.h"
#include "ros_lib/ros.h"
#include "ros_lib/v5_hal/V5Motor.h"
#include "ros_lib/std_msgs/Int8.h"
#include "ros_lib/std_msgs/Empty.h"

class MotorNode : public Node {
private:
    pros::Motor m_motor;
    v5_hal::V5Motor m_motor_msg;
    std::string m_handle_name;
    std::string m_sub_publish_data_name;
    std::string m_sub_move_motor_voltage_name;
    ros::Publisher* m_publisher;
    ros::Subscriber<std_msgs::Int8, MotorNode>* m_move_motor_voltage_sub;
    ros::Subscriber<std_msgs::Empty, MotorNode>* m_publish_data_sub;

    void m_populateMessage();

    void m_publishData(const std_msgs::Empty& msg);

    void m_moveMotorVoltage(const std_msgs::Int8& msg);

public:
    MotorNode(NodeManager* node_manager, int port_number, std::string
     ↪  handle_name,
        bool reverse=false, pros::motor_gearset_e_t
         ↪  gearset=pros::E_MOTOR_GEARSET_18);

    void initialize();

    void resetEncoder();

    int getPosition();

    void move(int value);

    void moveVoltage(int voltage);

    void moveVelocity(float velocity);

    void moveAbsolute(double position, int max_velocity);

    void teleopPeriodic();

    void autonPeriodic();

    ~MotorNode();
};

#MotorNode.cpp
#include "nodes/actuator_nodes/MotorNode.h"

// By default, this constructor calls the constructor for the Node object
 ↪  in
// NodeManager.h
```

```
MotorNode::MotorNode(NodeManager* node_manager, int port_number,
    std::string handle_name, bool reverse,
    pros::motor_gearset_e_t gearset) : Node(node_manager, 10),
    m_motor(port_number, gearset, reverse) {
    m_handle_name = handle_name.insert(0, "motor/");
    m_sub_move_motor_voltage_name = m_handle_name + "/moveMotorVoltage";
    m_sub_publish_data_name = m_handle_name + "/publish";

    m_publisher = new ros::Publisher(m_handle_name.c_str(), &m_motor_msg);

    m_publish_data_sub = new ros::Subscriber<std_msgs::Empty, MotorNode>
        (m_sub_publish_data_name.c_str(), &MotorNode::m_publishData, this);

    m_move_motor_voltage_sub = new ros::Subscriber<std_msgs::Int8,
     ↪  MotorNode>
        (m_sub_move_motor_voltage_name.c_str(),
         ↪  &MotorNode::m_moveMotorVoltage, this);
}

void MotorNode::m_publishData(const std_msgs::Empty& msg) {
    m_populateMessage();
    m_publisher->publish(&m_motor_msg);
}

void MotorNode::m_moveMotorVoltage(const std_msgs::Int8& msg) {
    float voltage = (msg.data / 127.0) * (float) MAX_MOTOR_VOLTAGE;
    moveVoltage((int)voltage);
}

void MotorNode::initialize() {
    // Initialize the handler, and set up data to publish
    Node::m_handle->advertise(*m_publisher);
    Node::m_handle->subscribe(*m_move_motor_voltage_sub);
}

void MotorNode::resetEncoder() {
    m_motor.tare_position();
}

int MotorNode::getPosition() {
    return m_motor.get_position();
}

void MotorNode::move(int value) {
    m_motor.move(value);
}

void MotorNode::moveVoltage(int voltage) {
    m_motor.move_voltage(voltage);
}

void MotorNode::moveVelocity(float velocity) {
    m_motor.move_velocity(velocity);
}

void MotorNode::moveAbsolute(double position, int max_velocity) {
    m_motor.move_absolute(position, max_velocity);
}

void MotorNode::teleopPeriodic() {

}

void MotorNode::autonPeriodic() {

}

void MotorNode::m_populateMessage() {
    m_motor_msg.direction = m_motor.get_direction();
    m_motor_msg.position = m_motor.get_position();
    m_motor_msg.velocity = m_motor.get_actual_velocity();
    m_motor_msg.voltage = m_motor.get_voltage();
}

MotorNode::~MotorNode() {
    delete m_publisher;
    delete m_publish_data_sub;
    delete m_move_motor_voltage_sub;
}
```

ConveyorNode

ConveyorNode holds the logic to automate our conveyor system on the robot, and focuses on using a state machine to control how the conveyor acts.

```
#ConveyorNode.h
#pragma once

#include "nodes/NodeManager.h"
```

```
#include "api.h"
#include "nodes/actuator_nodes/MotorNode.h"
#include "nodes/sensor_nodes/ADIAnalogInNode.h"
#include "nodes/sensor_nodes/ControllerNode.h"
#include "nodes/actuator_nodes/ADIDigitalOutNode.h"
#include "util/Constants.h"
#include "util/Timer.h"

class ConveyorNode : public Node {
public:
    enum ConveyorState {
        STOPPED, HOLDING, SCORING, REVERSE, DEPLOY
    };

    ConveyorNode(NodeManager* node_manager, std::string handle_name,
     ↪  ControllerNode* controller, MotorNode* bottom_conveyor_motor,
        MotorNode* top_conveyor_motor, ADIAnalogInNode*
         ↪  bottom_conveyor_sensor, ADIAnalogInNode* top_conveyor_sensor);

    void setBottomConveyorVoltage(int voltage);

    void setTopConveyorVoltage(int voltage);

    void setConveyorVoltage(int voltage);

    void setTopConveyorRPMPercent(float percent);

    void setBottomConveyorRPMPercent(float percent);

    void setConveyorRPMPercent(float percent);

    void setConveyorState(ConveyorState conveyorState);

    int getNumBallsStored();

    void initialize();

    void teleopPeriodic();

    void autonPeriodic();

    ~ConveyorNode();

private:
    ConveyorState m_current_conveyor_state = STOPPED;
    pros::Controller* m_controller;
    MotorNode* m_bottom_conveyor_motor;
    MotorNode* m_top_conveyor_motor;
    ADIAnalogInNode* m_bottom_conveyor_sensor;
    ADIAnalogInNode* m_top_conveyor_sensor;

    std::string m_handle_name;

    bool m_enable_holding;
    bool m_previous_ball_on_top;
    bool m_capture_sequence;
    Timer m_ball_hold_timer;

    void m_updateConveyorHoldingState();
};


#ConveyorNode.cpp
#include "nodes/subsystems/ConveyorNode.h"

ConveyorNode::ConveyorNode(NodeManager* node_manager, std::string
 ↪  handle_name, ControllerNode* controller,
        MotorNode* bottom_conveyor_motor, MotorNode* top_conveyor_motor,
         ↪  ADIAnalogInNode* bottom_conveyor_sensor,
        ADIAnalogInNode* top_conveyor_sensor) : Node(node_manager, 5),
        m_controller(controller->getController()),
        m_bottom_conveyor_motor(bottom_conveyor_motor),
        m_top_conveyor_motor(top_conveyor_motor),
        m_bottom_conveyor_sensor(bottom_conveyor_sensor),
        m_top_conveyor_sensor(top_conveyor_sensor),
        m_enable_holding(false),
        m_previous_ball_on_top(false),
        m_capture_sequence(false) {
    m_handle_name = handle_name.insert(0, "robot/");
}

void ConveyorNode::m_updateConveyorHoldingState() {
    bool is_ball_at_top = (m_top_conveyor_sensor->getValue() <=
     ↪  BALL_PRESENT_THRESHOLD);
    bool is_ball_at_bottom = (m_bottom_conveyor_sensor->getValue() <=
     ↪  BALL_PRESENT_THRESHOLD);

    if (m_capture_sequence) {
        setTopConveyorVoltage(-1 * MAX_MOTOR_VOLTAGE);
        if (m_ball_hold_timer.Get() > 0.05) {
            m_capture_sequence = false;
            m_previous_ball_on_top = true;
            m_ball_hold_timer.Reset();
        }
    } else {
        if (is_ball_at_top && !m_previous_ball_on_top) {
            m_ball_hold_timer.Start();
            m_capture_sequence = true;
            setTopConveyorVoltage(-1 * MAX_MOTOR_VOLTAGE);
        } else if (is_ball_at_top && m_previous_ball_on_top) {
            setTopConveyorVoltage(0);
        } else {
            setTopConveyorVoltage(MAX_MOTOR_VOLTAGE);
            m_previous_ball_on_top = false;
        }
    }

    setBottomConveyorVoltage(is_ball_at_top && is_ball_at_bottom ? 0 :
     ↪  MAX_MOTOR_VOLTAGE);
}

void ConveyorNode::setBottomConveyorVoltage(int voltage) {
    m_bottom_conveyor_motor->moveVoltage(voltage);
}

void ConveyorNode::setTopConveyorVoltage(int voltage) {
    m_top_conveyor_motor->moveVoltage(voltage);
}

void ConveyorNode::setConveyorVoltage(int voltage) {
    setTopConveyorVoltage(voltage);
    setBottomConveyorVoltage(voltage);
}

void ConveyorNode::setTopConveyorRPMPercent(float percent) {
    m_top_conveyor_motor->moveVelocity(percent * 600.);
}

void ConveyorNode::setBottomConveyorRPMPercent(float percent) {
    m_bottom_conveyor_motor->moveVelocity(percent * 200.);
}

void ConveyorNode::setConveyorRPMPercent(float percent) {
    setTopConveyorRPMPercent(percent);
    setBottomConveyorRPMPercent(percent);
}

void ConveyorNode::setConveyorState(ConveyorState conveyorState) {
    m_current_conveyor_state = conveyorState;
}

int ConveyorNode::getNumBallsStored() {
    int ballsStored = 0;

    if (m_bottom_conveyor_sensor->getValue() <= BALL_PRESENT_THRESHOLD) {
        ballsStored++;
    }

    if (m_top_conveyor_sensor->getValue() <= BALL_PRESENT_THRESHOLD) {
        ballsStored++;
    }

    return ballsStored;
}

void ConveyorNode::initialize() {

}

void ConveyorNode::teleopPeriodic() {
    if (m_controller->get_digital(pros::E_CONTROLLER_DIGITAL_R1)) {
        setConveyorState(SCORING);
        m_enable_holding = false;
    } else if (m_controller->get_digital(pros::E_CONTROLLER_DIGITAL_R2)) {
        setConveyorState(REVERSE);
        m_enable_holding = false;
    } else if (m_controller->get_digital(pros::E_CONTROLLER_DIGITAL_DOWN)) {
        setConveyorState(DEPLOY);
        m_enable_holding = false;
    } else if (m_controller->get_digital(pros::E_CONTROLLER_DIGITAL_UP) &&
     ↪  !m_enable_holding) {
        setConveyorState(HOLDING);
        m_enable_holding = true;
        pros::delay(150);
    } else if (m_controller->get_digital(pros::E_CONTROLLER_DIGITAL_UP) &&
     ↪  m_enable_holding) {
        setConveyorState(STOPPED);
        m_enable_holding = false;
        pros::delay(150);
    } else if (!m_enable_holding) {
        setConveyorState(STOPPED);
    }

    switch(m_current_conveyor_state) {
        case STOPPED:
            setConveyorVoltage(0);
        break;
        case HOLDING:
            m_updateConveyorHoldingState();
        break;
        case SCORING:
            setConveyorVoltage(MAX_MOTOR_VOLTAGE);
        break;
        case REVERSE:
            setConveyorVoltage(-1 * MAX_MOTOR_VOLTAGE);
        break;
        case DEPLOY:
            setBottomConveyorVoltage(-1 * MAX_MOTOR_VOLTAGE);
            setTopConveyorVoltage(0);
        break;
    }
}

void ConveyorNode::autonPeriodic() {
    switch(m_current_conveyor_state) {
        case STOPPED:
            setConveyorVoltage(0);
        break;
        case HOLDING:
            m_updateConveyorHoldingState();
        break;
        case SCORING:
            setConveyorVoltage(MAX_MOTOR_VOLTAGE);
        break;
        case REVERSE:
            setConveyorVoltage(-1 * MAX_MOTOR_VOLTAGE);
        break;
        case DEPLOY:
            setBottomConveyorVoltage(-1 * MAX_MOTOR_VOLTAGE);
            setTopConveyorVoltage(0);
        break;
    }
}
```

```
ConveyorNode::~ConveyorNode() {

}
```

HolonomicDriveNode

The holonomic drive node controls all commands to the drivetrain, and implements the needed kinematics to control our drivetrain properly

```
#HolonomicDriveNode.h
#pragma once

#include "nodes/NodeManager.h"
#include "api.h"
#include "nodes/subsystems/drivetrain_nodes/IDriveNode.h"
#include "nodes/actuator_nodes/MotorNode.h"
#include "nodes/sensor_nodes/ControllerNode.h"
#include "kinematics/HolonomicDriveKinematics.h"
#include "eigen/Eigen/Dense"
#include <algorithm>
#include <initializer_list>

class HolonomicDriveNode : public IDriveNode {
public:
    struct HolonomicMotors {
        MotorNode* left_front_motor;
        MotorNode* left_rear_motor;
        MotorNode* left_rear_motor_2;
        MotorNode* right_front_motor;
        MotorNode* right_rear_motor;
        MotorNode* right_rear_motor_2;
    };

    HolonomicDriveNode(NodeManager* node_manager, std::string handle_name,
     ↪  ControllerNode* controller,
        HolonomicMotors motors, HolonomicDriveKinematics kinematics);

    void initialize();

    void resetEncoders();

    IDriveNode::FourMotorDriveEncoderVals getIntegratedEncoderVals();
    void setDriveVoltage(int x_voltage, int theta_voltage);

    void setDriveVoltage(int x_voltage, int y_voltage, int theta_voltage);

    void setDriveVelocity(float x_velocity, float theta_velocity);

    void setDriveVelocity(float x_velocity, float y_velocity, float
     ↪  theta_velocity);

    void teleopPeriodic();

    void autonPeriodic();

    ~HolonomicDriveNode();

private:
    pros::Controller* m_controller;

    std::string m_handle_name;

    HolonomicMotors m_motors;

    HolonomicDriveKinematics m_kinematics;

    void m_setLeftFrontVoltage(int voltage);

    void m_setLeftRearVoltage(int voltage);

    void m_setRightFrontVoltage(int voltage);

    void m_setRightRearVoltage(int voltage);

    void m_setLeftFrontVelocity(float velocity);

    void m_setLeftRearVelocity(float velocity);

    void m_setRightFrontVelocity(float velocity);

    void m_setRightRearVelocity(float velocity);
};

#HolonomicDriveNode.cpp
#include "nodes/subsystems/drivetrain_nodes/HolonomicDriveNode.h"
```

```
HolonomicDriveNode::HolonomicDriveNode(NodeManager* node_manager,
 ↪  std::string handle_name, ControllerNode* controller,
    HolonomicMotors motors, HolonomicDriveKinematics kinematics) :
        IDriveNode(node_manager),
        m_controller(controller->getController()),
        m_motors(motors),
        m_kinematics(kinematics) {
    m_handle_name = handle_name.insert(0, "robot/");
}

void HolonomicDriveNode::m_setLeftFrontVoltage(int voltage) {
    m_motors.left_front_motor->moveVoltage(voltage);
}

void HolonomicDriveNode::m_setLeftRearVoltage(int voltage) {
    m_motors.left_rear_motor->moveVoltage(voltage);
    m_motors.left_rear_motor_2->moveVoltage(voltage);
}

void HolonomicDriveNode::m_setRightFrontVoltage(int voltage) {
    m_motors.right_front_motor->moveVoltage(voltage);
}

void HolonomicDriveNode::m_setRightRearVoltage(int voltage) {
    m_motors.right_rear_motor->moveVoltage(voltage);
    m_motors.right_rear_motor_2->moveVoltage(voltage);
}

void HolonomicDriveNode::m_setLeftFrontVelocity(float velocity) {
    m_motors.left_front_motor->moveVelocity(velocity);
}

void HolonomicDriveNode::m_setLeftRearVelocity(float velocity) {
    m_motors.left_rear_motor->moveVelocity(velocity);
    m_motors.left_rear_motor_2->moveVelocity(velocity);
}

void HolonomicDriveNode::m_setRightFrontVelocity(float velocity) {
    m_motors.right_front_motor->moveVelocity(velocity);
}

void HolonomicDriveNode::m_setRightRearVelocity(float velocity) {
    m_motors.right_rear_motor->moveVelocity(velocity);
    m_motors.right_rear_motor_2->moveVelocity(velocity);
}
```

```
void HolonomicDriveNode::initialize() {
    resetEncoders();
}

void HolonomicDriveNode::resetEncoders() {
    m_motors.left_front_motor->resetEncoder();
    m_motors.left_rear_motor->resetEncoder();
    m_motors.left_rear_motor_2->resetEncoder();
    m_motors.right_front_motor->resetEncoder();
    m_motors.right_rear_motor->resetEncoder();
    m_motors.right_rear_motor_2->resetEncoder();
}

IDriveNode::FourMotorDriveEncoderVals
 ↪  HolonomicDriveNode::getIntegratedEncoderVals() {
    return FourMotorDriveEncoderVals {
        m_motors.left_front_motor->getPosition(),
        m_motors.left_rear_motor->getPosition(),
        m_motors.right_front_motor->getPosition(),
        m_motors.right_rear_motor->getPosition()
    };
}

void HolonomicDriveNode::setDriveVoltage(int x_voltage, int theta_voltage) {
    setDriveVoltage(x_voltage, 0, theta_voltage);
}

void HolonomicDriveNode::setDriveVoltage(int x_voltage, int y_voltage, int
 ↪  theta_voltage) {
    IDriveKinematics::FourMotorPercentages motor_percentages =
        m_kinematics.inverseKinematics(x_voltage, y_voltage, theta_voltage,
         ↪  MAX_MOTOR_VOLTAGE);

    m_setLeftFrontVoltage(motor_percentages.left_front_percent *
     ↪  MAX_MOTOR_VOLTAGE);
    m_setLeftRearVoltage(motor_percentages.left_rear_percent *
     ↪  MAX_MOTOR_VOLTAGE);
    m_setRightFrontVoltage(motor_percentages.right_front_percent *
     ↪  MAX_MOTOR_VOLTAGE);
    m_setRightRearVoltage(motor_percentages.right_rear_percent *
     ↪  MAX_MOTOR_VOLTAGE);
}
```

```
void HolonomicDriveNode::setDriveVelocity(float x_velocity, float
 ↪  theta_velocity) {
    setDriveVelocity(x_velocity, 0, theta_velocity);
}

void HolonomicDriveNode::setDriveVelocity(float x_velocity, float
 ↪  y_velocity, float theta_velocity) {
    IDriveKinematics::FourMotorPercentages motor_percentages =
        m_kinematics.inverseKinematics(x_velocity, y_velocity,
         ↪  theta_velocity, MAX_VELOCITY);

    m_setLeftFrontVelocity(motor_percentages.left_front_percent *
     ↪  MAX_VELOCITY);
    m_setLeftRearVelocity(motor_percentages.left_rear_percent *
     ↪  MAX_VELOCITY);
    m_setRightFrontVelocity(motor_percentages.right_front_percent *
     ↪  MAX_VELOCITY);
    m_setRightRearVelocity(motor_percentages.right_rear_percent *
     ↪  MAX_VELOCITY);
}

void HolonomicDriveNode::teleopPeriodic() {
    int y_voltage =
     ↪  (m_controller->get_analog(pros::E_CONTROLLER_ANALOG_LEFT_Y) / 127.0)
     ↪  * MAX_MOTOR_VOLTAGE;
    int x_voltage =
     ↪  (m_controller->get_analog(pros::E_CONTROLLER_ANALOG_LEFT_X) / 127.0)
     ↪  * MAX_MOTOR_VOLTAGE;
    int theta_voltage =
     ↪  (m_controller->get_analog(pros::E_CONTROLLER_ANALOG_RIGHT_X) /
     ↪  127.0) * MAX_MOTOR_VOLTAGE;

    setDriveVoltage(x_voltage, y_voltage, theta_voltage);
}

void HolonomicDriveNode::autonPeriodic() {

}

HolonomicDriveNode::~HolonomicDriveNode() {
    delete m_motors.left_front_motor;
    delete m_motors.left_rear_motor;
    delete m_motors.right_front_motor;
    delete m_motors.right_rear_motor;
}
```

Odometry

The Odometry class is the parent class to all other generic forms of odometry we use on the robot, providing quick swapping functionality based on how we want to compute our odometry, or if we want to add redundant systems.

```
#Odometry.h
#pragma once

#include "api.h"
#include "math/Pose.h"
#include "math/Math.h"
#include "util/Encoders.h"

class Odometry {
protected:
    Rotation2Dd m_gyro_initial_angle;

    Pose m_robot_pose = Pose(Vector2d(0, 0), Rotation2Dd());

    bool m_pose_reset = true;

    double m_encoder_1_ticks_to_dist;
    double m_encoder_2_ticks_to_dist;

    double m_last_encoder_1_dist;
    double m_last_encoder_2_dist;

public:
    Odometry(EncoderConfig encoder_1_config, EncoderConfig encoder_2_config,
     ↪  Pose current_pose=Pose());

    void ResetEncoderTicks(double encoder_1_ticks=0, double
     ↪  encoder_2_ticks=0);

    Pose GetPose();

    void SetCurrentPose(Pose current_pose);

    virtual void Update(double encoder_1_raw_ticks, double
     ↪  encoder_2_raw_ticks, double track_width) {}
    virtual void Update(double encoder_1_raw_ticks, double
     ↪  encoder_2_raw_ticks, Rotation2Dd gyro_angle) = 0;
    ~Odometry();
};


#Odometry.cpp
#include "odometry/Odometry.h"

Odometry::Odometry(EncoderConfig encoder_1_config, EncoderConfig
 ↪  encoder_2_config, Pose current_pose):
        m_encoder_1_ticks_to_dist((encoder_1_config.wheel_diameter * PI) /
            encoder_1_config.ticks_per_wheel_revolution),
        m_encoder_2_ticks_to_dist((encoder_2_config.wheel_diameter * PI) /
            encoder_2_config.ticks_per_wheel_revolution),
        m_last_encoder_1_dist(0),
        m_last_encoder_2_dist(0) {
    SetCurrentPose(current_pose);
}

void Odometry::ResetEncoderTicks(double encoder_1_ticks, double
 ↪  encoder_2_ticks) {
    m_last_encoder_1_dist = encoder_1_ticks * m_encoder_1_ticks_to_dist;
    m_last_encoder_2_dist = encoder_2_ticks * m_encoder_2_ticks_to_dist;
}

Pose Odometry::GetPose(){
    return m_robot_pose;
}

void Odometry::SetCurrentPose(Pose current_pose) {
    m_robot_pose = current_pose;
    m_pose_reset = true;
}

Odometry::~Odometry() {

}
```

FollowerOdometry

FollowerOdometry is a specific odometry class implemented to handle the math of Follower odometry wheels (one on both the x and y axes)

```
#FollowerOdometry.h
#pragma once
```

```
#include "odometry/Odometry.h"
#include "util/Constants.h"

class FollowerOdometry : public Odometry {
private:

public:
    FollowerOdometry(EncoderConfig x_encoder_config, EncoderConfig
     ↪  y_encoder_config, Pose current_pose=Pose());

    void Update(double x_encoder_raw_ticks, double y_encoder_raw_ticks,
     ↪  Rotation2Dd gyro_angle);

    ~FollowerOdometry();
};

#FollowerOdometry.cpp
#include "odometry/FollowerOdometry.h"

FollowerOdometry::FollowerOdometry(EncoderConfig xEncoderConfig,
 ↪  EncoderConfig yEncoderConfig,
    Pose currentPose): Odometry(xEncoderConfig, yEncoderConfig, currentPose)
     ↪  {

}

/**
 * This function is used to update the internal calculated position of the
 ↪  robot
 *
 * To maximize accuracy in robot position estimation, this function should
 ↪  be called as frequently as possible, ideally
 * at a rate of 50 Hz or greater.
 *
 * @param x_encoder_raw_ticks The distance in ticks measured by the X
 ↪  encoder since when the tracker was last run
 * @param y_encoder_raw_ticks The distance in ticks measured by the Y
 ↪  encoder since when the tracker was last run
 * @param gyro_angle The current yaw of the robot as measured by the gyro
 */
void FollowerOdometry::Update(double x_encoder_raw_ticks, double
 ↪  y_encoder_raw_ticks, Rotation2Dd gyro_angle) {
    // Convert the current position in ticks to a position in distance
     ↪  units
    double x_dist = x_encoder_raw_ticks *
     ↪  Odometry::m_encoder_1_ticks_to_dist;
    double y_dist = y_encoder_raw_ticks *
     ↪  Odometry::m_encoder_2_ticks_to_dist;

    // Reset the current position of the robot
    if (m_pose_reset) {
        Odometry::m_last_encoder_1_dist = x_dist;
        Odometry::m_last_encoder_2_dist = y_dist;
        Odometry::m_gyro_initial_angle = gyro_angle;
        Odometry::m_pose_reset = false;
    }

    // Calculate the change in position since the last check
    double x_delta = x_dist - Odometry::m_last_encoder_1_dist;
    double y_delta = y_dist - Odometry::m_last_encoder_2_dist;

    // Convert the x and y deltas into a translation vector
    Vector2d robot_translation(x_delta, y_delta);

    // Update the current angle of the robot position
    Odometry::m_robot_pose.angle = gyro_angle *
     ↪  Odometry::m_gyro_initial_angle.inverse() *
     ↪  Eigen::Rotation2D(GYRO_OFFSET);

    // Rotate the translation vector by the current angle rotation matrix
    robot_translation = Odometry::m_robot_pose.angle * robot_translation;

    // Add the current translation onto the robot position vector
    Odometry::m_robot_pose.position += robot_translation;

    // Update the previous values of the encoders for the next iteration
    Odometry::m_last_encoder_1_dist = x_dist;
    Odometry::m_last_encoder_2_dist = y_dist;
}
```

SwerveModule

The SwerveModule class contains the kinematics to map abstract movements to specific motor speeds in a differential swerve module.

```
#SwerveModule.h
#pragma once
```

```
#include "eigen/Eigen/Dense"
#include "eigen/Eigen/Geometry"
#include "math.h"
#include "api.h"
#include "util/Constants.h"

struct MotorPowers {
    int8_t left_motor_power;
    int8_t right_motor_power;
};

class SwerveModule {
private:
    Eigen::Vector2d m_module_location;
    double m_percent_error;
    double m_total_error;
    double kP;
    double kI;
    double kD;


public:
    SwerveModule (Eigen::Vector2d module_location, double kP, double kI,
     ↪  double kD);

    MotorPowers InverseKinematics(Eigen::Vector2d target_velocity, double
     ↪  target_rotation_velocity, Eigen::Rotation2Dd module_actual_angle);
};


#SwerveModule.cpp
#include "swerve/SwerveModule.h"

SwerveModule::SwerveModule(Eigen::Vector2d module_location, double kP,
 ↪  double kI, double kD) :
    m_module_location(module_location),
    m_percent_error(0),
    m_total_error(0),
    kP(kP),
    kI(kI),
    kD(kD) {

}
```

```
MotorPowers SwerveModule::InverseKinematics(Eigen::Vector2d target_velocity,
 ↪  double target_rotation_velocity, Eigen::Rotation2Dd module_actual_angle)
 ↪  {
    // ROS_INFO("Target Velocity - x:%.2f y:%.2f", target_velocity(0),
     ↪  target_velocity(1));
    // ROS_INFO("Target Rotation Velocity: %.2f",
     ↪  target_rotation_velocity);

    // std::cout << "Target Velocity: " << target_velocity(0) << " " <<
     ↪  target_velocity(1) << "\n" << std::endl;
    // std::cout << "Target Rotation Velocity: " << target_rotation_velocity
     ↪  << "\n" << std::endl;

    // If you aren't trying to move, make sure to send no velocity to the
     ↪  motors
    if ((target_velocity(0) == 0) && (target_velocity(1) == 0) &&
     ↪  (target_rotation_velocity == 0)) { //not sure if this works, might
     ↪  need to be reworked
        double scaled_motor_1_mag = 0;
        double scaled_motor_2_mag = 0;

        MotorPowers motor_powers;

        motor_powers.left_motor_power = scaled_motor_1_mag;
        motor_powers.right_motor_power = scaled_motor_2_mag;

        return motor_powers;
    }

    // Create a maximum power vector for translating motor into percent
     ↪  motor output power
    Eigen::Vector2d max_motor_vector(MAX_VELOCITY, MAX_ROTATIONAL_VELOCITY);

    // Find the length of the maxium power vector, but divide by sqrt(2) to
     ↪  account for the fact that later
    // we will be projecting vectors onto this vector and due to the
     ↪  45-45-90 nature of this conversion need to
    // divide this vector by sqrt(2)
    float max_motor_power = max_motor_vector.norm() / sqrt(2);

    // Take the vector from the origin to the module (module_location) and
     ↪  rotate it to
    // make it orthogonal to the current (module_location) vector
    Eigen::Vector2d rotated_module_location = Eigen::Rotation2Dd(M_PI / 2) *
     ↪  m_module_location;
```

```
// Multiply the orthogonal vector (rotated_module_location) by the
↪ target angular velocity
// (target_rotation_velocity) to create your target rotation vector
Eigen::Vector2d target_rotation_vector = target_rotation_velocity *
↪ rotated_module_location;

// Add the target velocity and rotation vectors to get a resultant
↪ target vector
Eigen::Vector2d target_vector = target_velocity +
↪ target_rotation_vector;

// std::cout << "Target Velocity Vector (x, y): " << target_velocity(0)
↪ << " " << target_velocity(1) << "\n" << std::endl;
// std::cout << "Target Rotation Vector (x, y): " <<
↪ target_rotation_vector(0) << " " << target_rotation_vector(1) <<
↪ "\n" << std::endl;

// ROS_INFO("Target Vector - x:%.2f y:%.2f", target_vector(0),
↪ target_vector(1));

// std::cout << "Target Vector (x, y): " << target_vector(0) << " " <<
↪ target_vector(1) << "\n" << std::endl;

// Get the angle of the target vector by taking tangent inverse of y and
↪ x components
// of the vector, and convert to a Rotation2D angle object
Eigen::Rotation2Dd target_vector_angle =
↪ Eigen::Rotation2Dd(atan2(target_vector(1), target_vector(0)));

// ROS_INFO("Current Angle: %.2f", module_actual_angle.angle());
// ROS_INFO("Target Angle: %.2f", target_vector_angle.angle());

// std::cout << "Current Angle: " << module_actual_angle.angle() << "\n"
↪ << std::endl;
// std::cout << "Target Angle: " << target_vector_angle.angle() << "\n"
↪ << std::endl;

// Subtract the actual module vector from the target to find the change
↪ in angle needed
double module_rotation_delta = (target_vector_angle *
↪ module_actual_angle.inverse()).smallestAngle();

// PID control for turning
Eigen::Vector2d motor_power_vector;
```

```
if(fabs(module_rotation_delta) > M_PI_2){
    module_rotation_delta = (target_vector_angle *
     ↪  module_actual_angle.inverse() *
     ↪  Eigen::Rotation2Dd(M_PI)).smallestAngle();

    // Set the power as the magnitude of the vector
    motor_power_vector(0) = -target_vector.norm() / MAX_VELOCITY;

    // std::cout << "Forward %: " << motor_power_vector(0) << "\n" <<
     ↪  std::endl;
} else{
    // Set the power as the magnitude of the vector
    motor_power_vector(0) = target_vector.norm() / MAX_VELOCITY;

    // std::cout << "Forward %: " << motor_power_vector(0) << "\n" <<
     ↪  std::endl;
}

double error;
double derivative;

// Proportional
error = module_rotation_delta;

// std::cout << "PID Error: " << module_rotation_delta << "\n" <<
 ↪  std::endl;

// Derivative
derivative = error - m_percent_error;

// Integral
m_total_error += error;

// The last value of error
m_percent_error = error;

// Set the turn as the error * a constant + the derivative * a constant
 ↪  + the integral * a constant
motor_power_vector(1) = error * kP + derivative * kD + m_total_error *
 ↪  kI;

// std::cout << "PID Output: " << motor_power_vector(1) << "\n" <<
 ↪  std::endl;
```

```
// ROS_INFO("Motor Power Vector - x:%.2f y:%.2f", motor_power_vector(0),
 ↪  motor_power_vector(1));

// std::cout << "Target Vector (x, y): " << motor_power_vector(0) << " "
 ↪  << motor_power_vector(1) << "\n" << std::endl;

// We are working in the (m/s Forward)-(rpm Speed of Rotation) plane
 ↪  now
// Project the target vector onto each max motor vector to get
 ↪  components
// This finds the projection magnitude onto the max motor vector
double scaled_motor_1_mag = motor_power_vector.dot(Eigen::Vector2d(1,
 ↪  1));
double scaled_motor_2_mag = motor_power_vector.dot(Eigen::Vector2d(-1,
 ↪  1));

// ROS_INFO("Max Motor Mag: %.2f", max_motor_power);

// std::cout << "Max Motor Magnitude: " << max_motor_power << "\n" <<
 ↪  std::endl;

// ROS_INFO("Scaled Motor 1 Mag (START): %.2f", scaled_motor_1_mag);
// ROS_INFO("Scaled Motor 2 Mag (START): %.2f", scaled_motor_2_mag);

// std::cout << "Scaled Motor 1 Magnitude (0): " << scaled_motor_1_mag
 ↪  << "\n" << std::endl;
// std::cout << "Scaled Motor 2 Magnitude (0): " << scaled_motor_2_mag
 ↪  << "\n" << std::endl;

float max_scaled_motor_mag = fmax(fabs(scaled_motor_1_mag),
 ↪  fabs(scaled_motor_2_mag));

// If the max of the two motor powers is more than we can ouput, scale
 ↪  both down so the max motor's power
// is equal to the max_motor_power
if (max_scaled_motor_mag > 1.0) {
    scaled_motor_1_mag /= max_scaled_motor_mag;
    scaled_motor_2_mag /= max_scaled_motor_mag;
}

// ROS_INFO("Scaled Motor 1 Mag (1): %.2f", scaled_motor_1_mag);
// ROS_INFO("Scaled Motor 2 Mag (1): %.2f", scaled_motor_2_mag);
```

```
    // std::cout << "Scaled Motor 1 Magnitude (1): " << scaled_motor_1_mag
    ↪   << "\n" << std::endl;
    // std::cout << "Scaled Motor 2 Magnitude (1): " << scaled_motor_2_mag
    ↪   << "\n" << std::endl;

    // Scale motors between -127 and 127
    scaled_motor_1_mag = scaled_motor_1_mag * 127.0;
    scaled_motor_2_mag = scaled_motor_2_mag * 127.0;

    // ROS_INFO("Scaled Motor 1 Mag (FINAL): %.2f", scaled_motor_1_mag);
    // ROS_INFO("Scaled Motor 2 Mag (FINAL): %.2f", scaled_motor_2_mag);

    // std::cout << "Scaled Motor 1 Magnitude (2): " << scaled_motor_1_mag
    ↪   << "\n" << std::endl;
    // std::cout << "Scaled Motor 2 Magnitude (2): " << scaled_motor_2_mag
    ↪   << "\n" << std::endl;

    // Set and return the motor powers
    MotorPowers motor_powers;

    motor_powers.left_motor_power = (int8_t)scaled_motor_1_mag;
    motor_powers.right_motor_power = (int8_t)scaled_motor_2_mag;

    return motor_powers;
}
```

Auton

The Auton class manages the creation, execution, and deletion of our node structure in autonomous. These classes are vital to making quick and efficient autonomous routines.

```
#Auton.h
#pragma once

#include <vector>
#include <map>
#include <queue>
#include <string>
#include <memory>
#include <iostream>
#include <functional>

#include "util/Timer.h"
```

```
using namespace std;

class AutonAction {
public:
    enum actionStatus {
        CONTINUE,
        END
    };

    virtual void ActionInit() {}

    virtual void ActionEnd() {}

    virtual actionStatus Action() = 0;

    virtual ~AutonAction() {}
};

class AutonNode {
public:
    AutonNode(double timeout, AutonAction* action1 = nullptr, AutonAction*
     ↪  action2 = nullptr, AutonAction* action3 = nullptr);
    void AddNext(AutonNode* childNode);
    void AddAction(AutonAction* leaf);
    void AddCondition(function<bool()> startCondition);
    bool Complete();
    void Reset();
    void SetTimeout(double timeout);
    void Act(bool lastNodeDone);
    ~AutonNode();
private:
    vector<AutonNode*> m_children;
    vector<AutonAction*> m_actions;
    bool m_startConditonGiven = false;
    bool m_actionsInitialized = false;
    function<bool()> m_startCondition;
    double m_timeout;
    Timer m_timer;
};

class Auton {
public:
    Auton(string name, bool defaultAuton = false);

    inline Auton* GetInstance() {
        return this;
    }

    inline bool GetDefault() {
        return m_defaultAuton;
    }

    string GetName();
    void AutonPeriodic();
    void AutonInit();
    bool Complete();
    void Reset();
    virtual ~Auton();
protected:
    void AddFirstNode(AutonNode* firstNode);
    virtual void AddNodes() = 0;
private:
    string m_name;
    bool m_defaultAuton = false;
    vector<AutonNode*> m_firstNode;
};

class WaitAction : public AutonAction {
public:
        WaitAction(double duration);
        void ActionInit();
        actionStatus Action();
private:
        Timer m_timer;
        double m_duration;
};

class PrintAction : public AutonAction {
public:
    PrintAction(string toPrint);
    void ActionInit();
    actionStatus Action();

private:
    string m_toPrint;
};


#Auton.cpp
#include "eigen/Eigen/Dense"
```

```
#include "eigen/Eigen/Geometry"
#include "math.h"
#include "api.h"
#include "util/Constants.h"

struct MotorPowers {
    int8_t left_motor_power;
    int8_t right_motor_power;
};

class SwerveModule {
private:
    Eigen::Vector2d m_module_location;
    double m_percent_error;
    double m_total_error;
    double kP;
    double kI;
    double kD;


public:
    SwerveModule (Eigen::Vector2d module_location, double kP, double kI,
     ↪  double kD);

    MotorPowers InverseKinematics(Eigen::Vector2d target_velocity, double
     ↪  target_rotation_velocity, Eigen::Rotation2Dd module_actual_angle);
};
```

AutonSequence

The AutonSequence class allows us to sequence multiple Auton nodes together, making our routines much more compact and understandable. Sequences are created and returned by static functions, and patched into routines with the beginning and end pointers.

```
#AutonSequence.h
#pragma once

#include "Auton.h"

class AutonSequence{
private:
    struct AutonSequenceList{
        AutonNode* headNode;
        AutonNode* tailNode;
    };
    AutonSequenceList m_auton_sequence;
public:
    AutonSequence(AutonNode* initialNode);
    void AddNext(AutonNode* node);
    void AddAction(AutonAction* action);
    AutonSequenceList GetSequence();
    ~AutonSequence();
};


#AutonSequence.cpp
#include "auton/AutonSequence.h"

AutonSequence::AutonSequence(AutonNode* initialNode) {
    m_auton_sequence = { initialNode, initialNode };
}

void AutonSequence::AddAction(AutonAction* action) {
    m_auton_sequence.tailNode->AddAction(action);
}

void AutonSequence::AddNext(AutonNode* node) {
    m_auton_sequence.tailNode->AddNext(node);
    m_auton_sequence.tailNode = node;
}

AutonSequence::AutonSequenceList AutonSequence::GetSequence() {
    return m_auton_sequence;
}

AutonSequence::~AutonSequence() {

}
```

Path

The path class manages dispatching positions along a path based on time, and allows for time-based feedback loops to run from a parsed path object.

```
#pragma once

#include "eigen/Eigen/Dense"
#include "pathing/PathPoint.h"
```

```
#include "math/Pose.h"
#include <vector>

using namespace std;

class Path {
public:
    Path();

    Path(vector<PathPoint> pathPoints);

    Pose update(float time);

    vector<PathPoint> getPathPoints();

    bool isComplete();

private:
    vector<PathPoint> m_pathPoints;
    PathPoint m_last_point;
    bool m_is_complete;
};


#Path.cpp
#include "pathing/Path.h"

Path::Path() :
        m_is_complete(false),
        m_last_point(0, Pose(), Vector2d(0., 0.), 0.) {

}

Path::Path(vector<PathPoint> pathPoints) :
        m_pathPoints(pathPoints),
        m_is_complete(false),
        m_last_point(pathPoints.back()) {
}

Pose Path::update(float time) {
    // Remove any points that have been passed
    for (auto it = m_pathPoints.begin(); it != m_pathPoints.end(); it++) {
        if(time > (it + 1)->getTime()) { // Point has been passed
            m_pathPoints.erase(it);
        } else { // Point not yet reached
```

```
            break;
        }
    }

    Pose next_pose;

    if (m_pathPoints.size() == 0) {
        next_pose = m_last_point.getPose();
        m_is_complete = true;
    } else {
        //Interpolate between the first point and the next point
        auto path_point =
         ↪  m_pathPoints.begin()->interpolateTo(*(m_pathPoints.begin() + 1),
         ↪  time);
        next_pose = path_point.getPose();
    }

    return next_pose;
}

vector<PathPoint> Path::getPathPoints() {
    return m_pathPoints;
}

bool Path::isComplete() {
    return m_is_complete;
}
```

PathPoint

PathPoint is the main structure to a path, consisting of methods to interpolate between way-points in order to get an exact position at a certain point of time

```
#PathPoint.h
#pragma once

#include "eigen/Eigen/Dense"
#include "math/Pose.h"

class PathPoint {
public:
    PathPoint(float time, Pose pose, Vector2d linear_velocity, float
     ↪  rotational_velocity);
    float getTime();

    Pose getPose();

    Vector2d getLinearVelocity();

    float getRotationalVelocity();

    PathPoint interpolateTo(PathPoint other, float time);
private:
    Pose m_pose;
    float m_time;
    Vector2d m_linear_velocity;
    float m_rotational_velocity;
};


#PathPoint.cpp
#include "pathing/PathPoint.h"

PathPoint::PathPoint(float time, Pose pose, Vector2d linear_velocity, float
 ↪  rotational_velocity) {
    m_time = time;
    m_pose = pose;
}

float PathPoint::getTime() {
    return m_time;
}

Pose PathPoint::getPose() {
    return m_pose;
}

Vector2d PathPoint::getLinearVelocity() {
    return m_linear_velocity;
}

float PathPoint::getRotationalVelocity() {
    return m_rotational_velocity;
}

PathPoint PathPoint::interpolateTo(PathPoint other, float time) {
    float t = time - getTime();
    float deltaTime = other.getTime() - getTime();
    Vector2d acceleration = (other.getLinearVelocity() +
     ↪  getLinearVelocity()) / deltaTime;
    Vector2d position = 0.5 * acceleration * (t * t) + getLinearVelocity() *
     ↪  t + getPose().position;
    Vector2d velocity = acceleration * t + getLinearVelocity();

    float alpha = (other.getRotationalVelocity() - getRotationalVelocity())
     ↪  / deltaTime;
    Rotation2Dd theta = Rotation2Dd(0.5 * alpha * (t * t) +
     ↪  getRotationalVelocity() * t) * getPose().angle;
    float omega = alpha * t + getRotationalVelocity();

    return PathPoint(time, Pose(position, theta), velocity, omega);
}
```

V5 Serial Driver

Our V5 Serial Driver class, or V5Serial, is used to template the NodeHandle class for ROS to send messages to the ROS master over serial USB through the USB port.

```
#V5Serial.h
/*
 * Software License Agreement (BSD License)
 *
 * Copyright (c) 2011, Willow Garage, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *  * Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *  * Redistributions in binary form must reproduce the above
 *    copyright notice, this list of conditions and the following
 *    disclaimer in the documentation and/or other materials provided
 *    with the distribution.
 *  * Neither the name of Willow Garage, Inc. nor the names of its
 *    contributors may be used to endorse or promote prducts derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef _ROSSERIAL_VEX_V5_V5_SERIAL_H_
#define _ROSSERIAL_VEX_V5_V5_SERIAL_H_

#include "ros_lib/rosserial_vex_v5/utils/RingBuf.h"

 // for the mutex.
#include "pros/apix.h"

#define SERIAL_CLASS int
#define ROSVEX_BUFFER_INPUT_SIZE 32

using RB = RingBufCPP<char, ROSVEX_BUFFER_INPUT_SIZE>;

// load the serial reading into a buffer.
inline void vexRosBufferInput(void* arg) {

  void** arglist = (void**)arg;
  RB* inputBuffer = (RB*)arglist[0];
  __FILE* streamOut = (__FILE*)arglist[1];

  int readcount = 0;
  while (1) {
    char c = fgetc(streamOut);
    inputBuffer->add(c);
  }
}

class V5Serial {

public:
  V5Serial() : rosvexMutex(), inputBuffer(rosvexMutex), failCount(),
   ↪  successCount() {
  }
  // any initialization code necessary to use the serial port
  // note: the serial port initialization for rosserial for VEX Cortex must
   ↪ be implemented in `src/init.cpp`
  // see that file for more information.
  void init() {
    pros::c::serctl(SERCTL_DISABLE_COBS, NULL);
    rosFile = fopen("/ser/sout", "r+");
    pros::c::fdctl(fileno(rosFile), SERCTL_DEACTIVATE, NULL);

    // not typesafe, be careful!
    void** taskArgs = (void**)malloc(sizeof(void*) * 2);
    taskArgs[0] = &inputBuffer;
    taskArgs[1] = rosFile;

    pros::Task reader(vexRosBufferInput, taskArgs);
  }

  // read a byte from the serial port. -1 = failure
  int read() {
    char c;
    // pull serial reading out of the buffer.
    if (inputBuffer.pull(&c)) {
      char sucmsg[16];
      return c;
    }

    return -1;
  }

  // write data to the connection to ROS
  void write(uint8_t* data, int length) {
    for (int i = 0; i < length; i++) {
      vexroswritechar(data[i]);
    }
  }
  // returns milliseconds since start of program
  unsigned long time() {
    return pros::c::millis();
  }
private:
  int failCount;
  int successCount;
  pros::Mutex rosvexMutex;
  __FILE* rosFile;
  RB inputBuffer;

  // writing helper.
  void vexroswritechar(uint8_t data) {
    fputc(data, rosFile);
    fflush(rosFile);
  }

  // reading helper.
  char vexrosreadchar() {
    return fgetc(rosFile);
  }
};

#endif
```

V5 Smart Port Driver

Our V5 Smart Port Driver class, or V5RS485, is used to template the NodeHandle class for ROS to send messages to the ROS master over RS-485 through the smart ports.

```
#V5RS485.h
/*
 * Software License Agreement (BSD License)
 *
 * Copyright (c) 2011, Willow Garage, Inc.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *  * Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *  * Redistributions in binary form must reproduce the above
 *    copyright notice, this list of conditions and the following
 *    disclaimer in the documentation and/or other materials provided
 *    with the distribution.
 *  * Neither the name of Willow Garage, Inc. nor the names of its
 *    contributors may be used to endorse or promote prducts derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

#ifndef _ROSSERIAL_VEX_V5_V5_V5RS485_H_
#define _ROSSERIAL_VEX_V5_V5_V5RS485_H_

#include "pros/apix.h"

class V5RS485 {
   public:
    V5RS485(int readPortNum = 19, int writePortNum = 20, int baud = 115200)
        : readPort(readPortNum, baud),
          writePort(writePortNum, baud) {}

    void init() {
        pros::delay(10);
        readPort.flush();
        writePort.flush();
    }

    // read a byte from the serial port. -1 = failure
    int read() {
        int read = readPort.read_byte();
        return read;
    }

    // write data to the connection to ROS
    void write(uint8_t* data, int length) {
        int freeBytes = writePort.get_write_free();

        if(freeBytes > length) { // Enough bytes free in buffer
            writePort.write(data, length);
        } else {
            printf("Serial buffer full!\n");
            writePort.write(data, freeBytes);
            for(int i = freeBytes; i < length; i++) {
                while(writePort.get_write_free() == 0) {
                    pros::delay(5);
                }
                writePort.write_byte(data[i]);
            }
        }
    }

    // returns milliseconds since start of program
    unsigned long time() { return pros::c::millis(); }

   private:
    pros::Serial readPort;
    pros::Serial writePort;
};

#endif
```