Engineering Design Journal

5839B

2024 - 2025

High Stakes

Table of Contents

Entries

Appendix

Our Team

Davis Bodami

The main designer of the team and enjoys no-lifeing robotics

Praful Adiga

The main programmer of the team and makes code to operate the robot's systems

Brandon Lewis

Keeps the team organized and punctual, and assists with coding and building when needed

Cole Stephan

The main drive team coordinator to ensure correct execution during events

Jerrick Chen

Designs and constructs various components on the robot

Madison Moreno

Assembles assorted parts and coordinates information with the public

Our Team

Jayden Htwe

Journalist and editor for the notebook and assists with building

? Pre-Rulebook Reveal Thoughts 2024/04/17

After watching the conclusion of the World Competition, three of us—Davis, Praful, and Andrew—began to theorize ideas and strategies for the new game. These ideas were created before the release of the rulebook, and new ideas will be documented upon its release.

Initially, the similarities to the previous games of Round Up and Tipping Point provide a wide variety of strategies to draw from, such as:

- Pneumatically grabbing a mobile goal (MOGO)
- Using an intake to place rings on a held MOGO

In addition, some unique ideas were discussed, such as:

- Using a D4Rb or similar lift to raise and lower the intake to deposit at the variable elevations where the stakes reside
- Having a pneumatic grip on the intake to grab the top bar and pull the robot off
- Creating a descoring device to remove rings from a goal
- Having at least 4 or possibly 6 motor drive to have enough power to hold the corners when needed

Multiple ideas relating to strategy were also discussed:

- Climbing is far more valuable compared to previous games due to the reduced point values of the rings compared to previous objects.
- Securing the corners is critical, as they could flip a game in an instant.
- Having a way to only put your rings or the enemy's rings onto a goal quickly could prove valuable.
- One could make a MOGO with only enemy rings just for it to be put in the descoring corner.
- The top ring seems pointless if it is only worth three points, as going for it would require precision and a good amount of time that could be better used elsewhere.

Some ideas for decorating the robot and field were also discussed:

- One could draw the popular video game character Kirby or a face in general on one of the rings.
- The top ring is the "One Ring to Rule Them All," allowing for multiple Lord of the Rings references, such as calling the robot "The Eye of Sauron."

Overall, we see this game as being far more complex than previous games, requiring more advanced mechanisms and a higher level of quality for a robot to be competitive. We shall begin prototyping some models in Inventor the following day and adjust when the rules are released. We are also interested to see if there will be any big differences in skills compared to the base game.

? Post-Rulebook Reveal Thoughts 2024/05/14

To the entire team's surprise, the rulebook completely changes how the game is played compared to our initial thoughts from the video. Davis, Brandon, and Praful discussed these on Discord the night it was released.

New Rules

- The vertical expansion limit makes the previous lift models too tall, but not out of the picture, as elevating the intake could still help with climbing and depositing rings.
- It is required to go rung by rung to climb, greatly increasing the challenge in doing so, but the point values still make it worthwhile.
- The expansion limits do make the high stake worth far less, as scoring it would require climbing to the top rung and then having a mechanism to score it for only 3 more points.

Skills

- The rules regarding the different rings make it valuable for the robot to be able to sort out colors in its intake to ensure no rings are wasted.
- The corners are still valuable for the 5 points per MOGO, but not as crucial for the game.
- Descore mechanisms could help remove the pre-scored blue rings for more points.

Final Thoughts for the First Robot

- 6-motor drive
- Intake on a lift
- Pneumatic clamp to grab MOGOs
- Clamp on intake/lift to grab the first rung
- Passive clamps to hold onto the climbing rungs so that the lift can raise and grab the next rung
- Descore mechanism to help in skills and matches

Post Season Build Analysis

For the building of our previous robot even though it unfortunately did not make worlds there were a lot of aspects executed properly and poorly to be identified. This is important as it allows the team to know what works and to keep doing as well as what to change in order to improve for the next season.

Drive Train The Drive Train is the base of any robot. The previous seasons drive train used 6 11w motors with the blue cartidge and a 48:60 gear ratio giving an RPM of 480. This spun 3 4" omni wheels with the middle one on each side being locked.

Pros	Cons
• Speed • Pushing power • Strong Wedge	• Turning • Reliability • Over Heating • Field Traversal

Intake The intake was powered by 1 5.5w motor spining a series of 2" 45A flexwheels to interact with the game objects. The intake was also allowed to float so that it could raise over the goal to score the triball.

Pros	Cons
• Holding ability • Scoring • Reliability • Effectiveness	• Chain Broke once

Wings These were pneumatically activated flaps that would extend 9in on either side of the robot. These allowed for a large amount of game objects to be pushed into the goal at one time.

Pros	Cons
• Never failed • Reached corners • Simple Design	• Bent after multiple matches

Flywheel Arm this was a 4" Flexwheel with a ratchet spining at 3600 RPM off of a blue motor. Game obejcts were placed and laumched off of the flywheel. It could also be raised by a 5.5w motor assisted by rubber bands to shoot over other robots.

Pros	Cons
• Consistent firing • Fast firing • Height • Ratchet persrved motor health	• Unable to use the arm for climbing • Flywheel got jammed on a standoff in 2 matches

Odometry Modules These are 3 modules 2 vertical and 1 horizantal that are used to track the robots position. They are jointed to always be in contact with the ground, and have a 3.25" omni wheel spin an encoder to track movements.

Pros	Cons
• Simple Design	• Bent over time • Large • Unreliable gorund contact

What to do now?

- With the biggest problem being the drive a variety of drives should be modeled and tested in order to have a better idea of what works for the next season
- Work to create new odometry module desings that are stronger and more compact
- Take inventory of the parts available to our team so that when designing we know what parts we can use and how many of them are available
- Put together an order list of parts that the team wants to asses the needed funds
- See what funding is available to the team and what amount should be allocated to new parts
- Look into making a functional PTO (power take off) as they can allow for more powerful drives while still having all the desired mechanisms to manipulate game objects
- Look for or model our own paddels for the controller that suite the needs of our driver

Before any models for possible drive trains or new odometry sensors can be made a decision on what software to use is necessary. In the previous season we adpoted Autodesk inventor. There exists 3 main options other then inventor for vex which include Solid Works, Fusion 360, and Onshape. Each has a variety of favorable aspects to be considered when choosing.

The best way to comapre these is through a decision matrix of various aspects of each

	Familiarty	Acessability	Availble Part Libraries	Availble Help Recoruces	Vex Compata-bility	Ease of Use	Total
Inventor	150	100	20	25	8	6	309
Fusion 360	90	60	25	15	10	8	208
Solid Works	60	80	15	10	6	6	177
Onshape	30	100	20	15	8	8	181

Note

Inventor scored far higher in some categories compared to other teams due to Eastern Tech's Engineering Program. This program which the majority of the team is in teaches Inventor and provides us with a browser version of it. This ngeates Onshapes main advantage and gives us 3 teachers who can help fix any problems we run into.

Final Decision

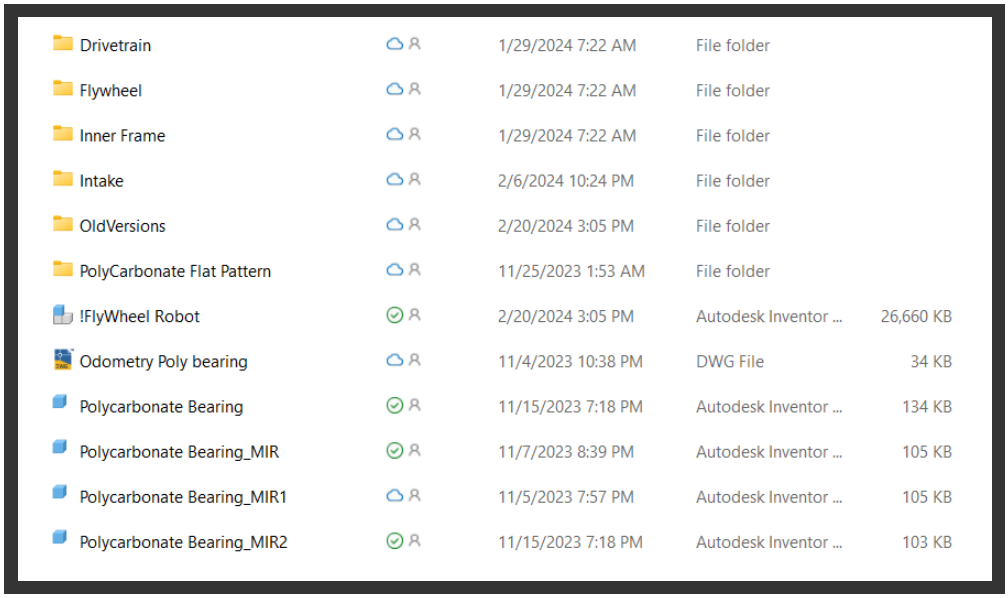
Due to a variety of reasons the main one being a lack of time to learn a new software and familiarty with Inventor it will continue to be our primary design software.

To effectivley use any software whether for coding or modeling, file organization is key. To share these files the team makes use of Microsoft OneDrive since it works best with windows device which the majority of the team has.

For the Inventor model, the main robot assembly is made from a variety of sub-assemblies. These sub-assemblies are each for a key system such as the drive train or intake. There can exist further sub aseemblies within these as well such as that for the odometry sensors within the assembly they are attachetd to. To organize the many file for this a variety of naming conventions are used based off of the guide lines from Perdu Sig Robotics.

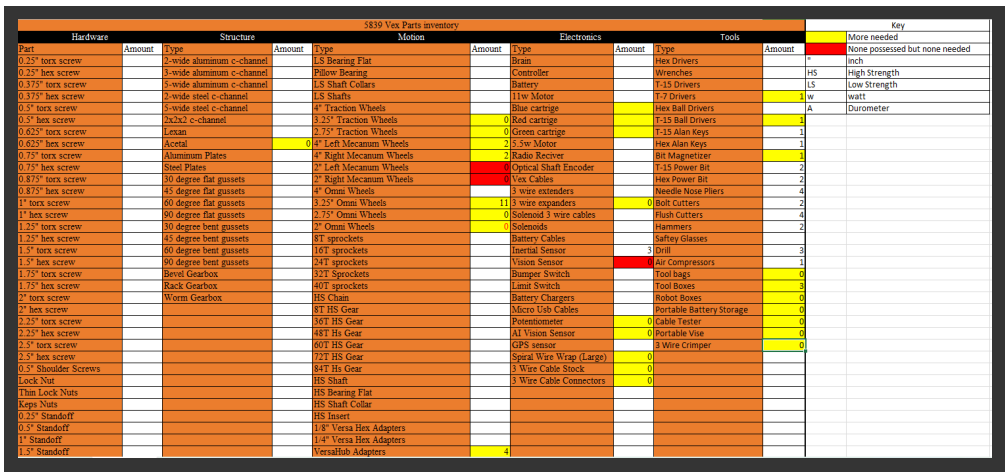
- Robots are marked with a folder labeled “! Robot Name”
- Within in that folder is the main assembly of the same name
- Assemblies or cut parts used throughout the model are stored in this high level folder
- There are also folders with each of the sub-assemblies labeled “Assembly Name”
- Within the sub-assembly folders are all the special cut parts and sub-assemblies of that Assembly.
- All other parts are stored in a default parts folder with every vex part organized by type

This system comes with a variety of advantages for the team. Indivudal axles or mechanisms can be edited without opening the whole robot which allows team members to interact with the models without a powerful computer. It also allows for big changes to be made easier as the major sub-assemblies can be removed or changed without having to edit a bunch of indivudal parts. This also helps during the build process as we can divide the many sub-assemblies between team members without any other work to do. In the past, this has proven to allow us to build full redesigns in just a week and half such as what we did after the Dulaney competition last year.



Inventor Files from the Over Under Season

Before any designing can take place it is key to know the constraints one is placed under. For vex a key way of doing this other then reading the rules is too see what parts are available to your team. You may have the best idea for a design but without hte parts to build that idea is jsut waisted time.In order to see what parts we had an excel spread sheet was created with all Vex Parts that were in our Inventor parts library as well as newer ones found the Vex website. Additionally Tools and other accesories from the Robosource website were included that we deemed may prove useful.

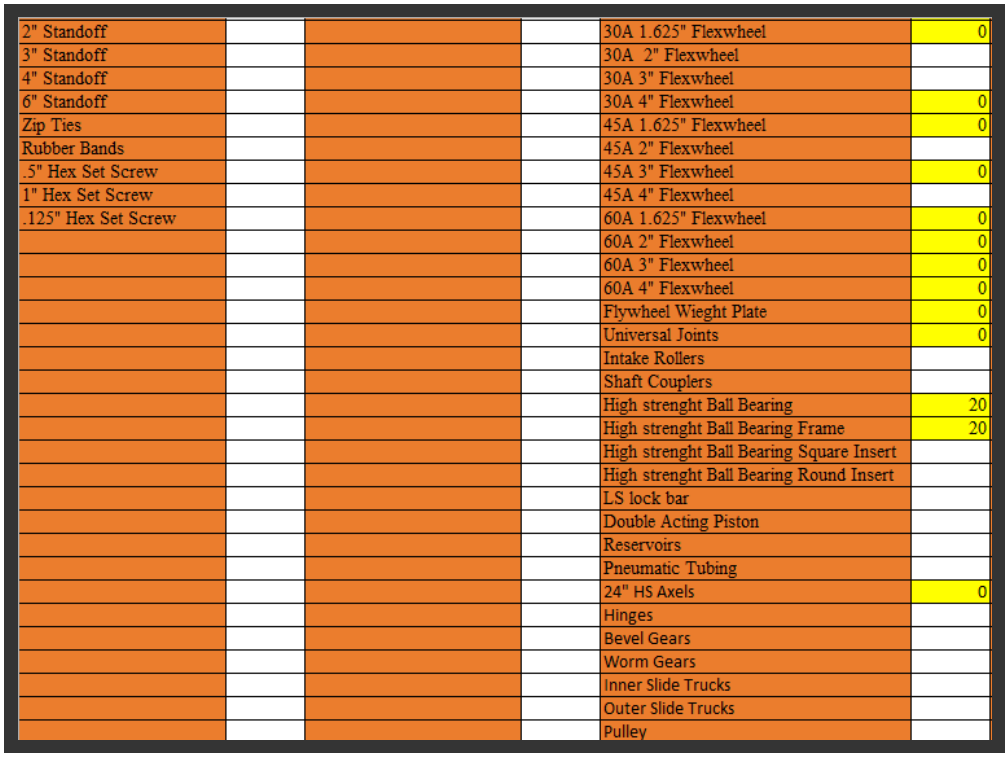


5839 Vex Parts inventory

Hardware		Structure		Motion		Electronics		Tools	
Part	Amount	Type	Amount	Type	Amount	Type	Amount	Type	Amount
0.25" torx screw		2-wide aluminum c-channel		LS Bearing Flat		Brain		Hex Drivers	
0.25" hex screw		3-wide aluminum c-channel		Pillow Bearing		Controller		Wrenches	
0.375" torx screw		5-wide aluminum c-channel		LS Shaft Collars		Battery		T-15 Drivers	
0.375" hex screw		2-wide steel c-channel		LS Shafts		11w Motor		T-7 Drivers	1
0.5" torx screw		5-wide steel c-channel		4" Traction Wheels		Blue cartrige		Hex Ball Drivers	
0.5" hex screw		2x2x2 c-channel		3.25" Traction Wheels	0	Red cartrige		T-15 Ball Drivers	1
0.625" torx screw		Lexan		2.75" Traction Wheels	0	Green cartrige		T-15 Alan Keys	1
0.625" hex screw		Acetal	0	4" Left Mecanum Wheels	2	2.5w Motor		Hex Alan Keys	1
0.75" torx screw		Aluminum Plates		4" Right Mecanum Wheels	2	Radio Reciver		Bit Magnetizer	1
0.75" hex screw		Steel Plates		2" Left Mecanum Wheels	0	Optical Shaft Encoder		T-15 Power Bit	2
0.875" torx screw		30 degree flat gussets		2" Right Mecanum Wheels	0	Vex Cables		Hex Power Bit	2
0.875" hex screw		45 degree flat gussets		4" Omni Wheels		3 wire extenders		Needle Nose Pliers	4
1" torx screw		60 degree flat gussets		3.25" Omni Wheels	11	3 wire expanders	0	Bolt Cutters	2
1" hex screw		90 degree flat gussets		2.75" Omni Wheels	0	Solenoid 3 wire cables		Flush Cutters	4
1.25" torx screw		30 degree bent gussets		2" Omni Wheels	0	Solenoids		Hammers	2
1.25" hex screw		45 degree bent gussets		8T sprockets		Battery Cables		Saftey Glasses	
1.5" torx screw		60 degree bent gussets		16T sprockets		Inertial Sensor	3	Drill	3
1.5" hex screw		90 degree bent gussets		24T sprockets		Vision Sensor	0	Air Compressors	1
1.75" torx screw		Bevel Gearbox		32T Sprockets		Bumper Switch		Tool bags	0
1.75" hex screw		Rack Gearbox		40T sprockets		Limit Switch		Tool Boxes	3
2" torx screw		Worm Gearbox		HS Chain		Battery Chargers		Robot Boxes	0
2" hex screw				8T HS Gear		Micro Usb Cables		Portable Battery Storage	0
2.25" torx screw				36T HS Gear		Potentiometer	0	Cable Tester	0
2.25" hex screw				48T Hs Gear		AI Vision Sensor	0	Portable Vise	0
2.5" torx screw				60T HS Gear		GPS sensor		3 Wire Crimper	0
2.5" hex screw				72T HS Gear		Spiral Wire Wrap (Large)	0		
0.5" Shoulder Screws				84T Hs Gear		3 Wire Cable Stock	0		
Lock Nut				HS Shaft		3 Wire Cable Connectors	0		
Thin Lock Nuts				HS Bearing Flat					
Keps Nuts				HS Shaft Collar					
0.25" Standoff				HS Insert					
0.5" Standoff				1/8" Versa Hex Adapters					
1" Standoff				1/4" Versa Hex Adapters					
1.5" Standoff				VersaHub Adapters	4				

Key	
	More needed
	None possessed but none needed
-	inch
HS	High Strength
LS	Low Strength
w	watt
A	Durometer

Top half of the Inventory Spread Sheet

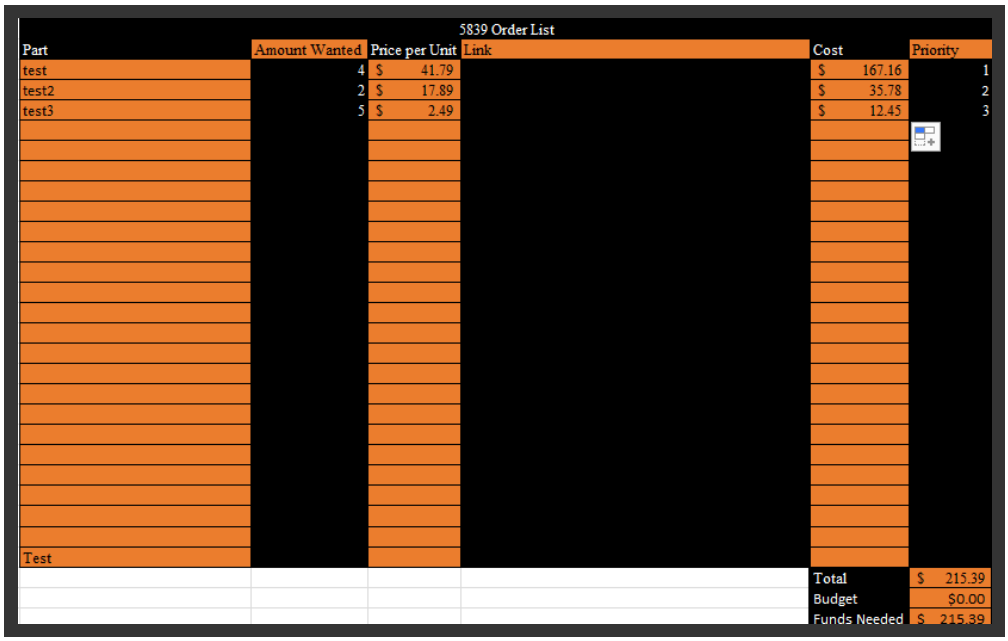


2" Standoff				30A 1.625" Flexwheel	0
3" Standoff				30A 2" Flexwheel	
4" Standoff				30A 3" Flexwheel	
6" Standoff				30A 4" Flexwheel	0
Zip Ties				45A 1.625" Flexwheel	0
Rubber Bands				45A 2" Flexwheel	
.5" Hex Set Screw				45A 3" Flexwheel	0
1" Hex Set Screw				45A 4" Flexwheel	
.125" Hex Set Screw				60A 1.625" Flexwheel	0
				60A 2" Flexwheel	0
				60A 3" Flexwheel	0
				60A 4" Flexwheel	0
				Flywheel Wieght Plate	0
				Universal Joints	0
				Intake Rollers	
				Shaft Couplers	
				High strenght Ball Bearing	20
				High strenght Ball Bearing Frame	20
				High strenght Ball Bearing Square Insert	
				High strenght Ball Bearing Round Insert	
				LS lock bar	
				Double Acting Piston	
				Reservoirs	
				Pneumatic Tubing	
				24" HS Axels	0
				Hinges	
				Bevel Gears	
				Worm Gears	
				Inner Slide Trucks	
				Outer Slide Trucks	
				Pulley	

Bottom half of the Inventory Spread Sheet

The Spreadsheet will take a while to fill out, but for now parts with known quanitities such as zero have been filled out. Those were marked with yellow to indicate more were needed. These were then taken

to a second spreadsheet. This lays out all the parts and tools we want, their price, quanity, link to purchase, and priority. The priority is key as it allows us to make decisions on what to get within our budget.



5839 Order List

Part	Amount Wanted	Price per Unit	Link	Cost	Priority
test	4	$ 41.79		$ 167.16	1
test2	2	$ 17.89		$ 35.78	2
test3	5	$ 2.49		$ 12.45	3
Test					
				Total	$ 215.39
				Budget	$0.00
				Funds Needed	$ 215.39

Order Spread Sheet with a calculation test

> **Note**
>
> Currently the budget is at zero as dues are yet to be collected and no fund raisers have been planned. It also important to consider that a small percentage of the school's 15,000 dollar engineering budget is randomly allocated to us so we will have that to work with.

Inventory Results 2024/03/10

After a week of work we were able to complete all rows of the spread sheet and figure out what parts the team was in need of. A variety of methods were used to measure the various parts. Large parts like Wheels and Motors were counted but other parts required a different aproac=h. String wires, and tubing were measured in feet, and metal strcuture by its weight. For parts like screws and nuts a single unit was weighed as well as the container and the total amount we had. The weight of the container was then subtracted from the total and then divided by the unit to find the total quanity.

> **Note**
>
> We found our time managment to be extremely poor during this endevor which greatly increased its length. To adress this we may work to change how the team meets to allow for not only more time, but better uses of that time.

Drive Train Types

2024/03/17

New Drive Models

There exists a variety of drive models both practical and impractical that can be made with the VRC legal parts. It is important to judge where each one can shine to see which is the most practical for the next game release. A decision cannot yet be made for which drive is best, but the strengths and weaknesses of each one can be assesed as well as models for the more practical ones generated. These models can give us a head start on the next seasons bot if they prove adequate for the next game as well as allow the team to test various ideas.

Tank Drive

A large variety of what can be considered a Tank Drive or differential drives exist within Vex. These work by having two sides where each sides' wheels all spin together. This allows for linear motion (Both sides spin the same direction), turning (Both sides spin opposite), and arcing (One side spins slower then the other in the same direction). These drives are often the simplist and provide a wide range of motion while remaining able to push back against other robots. This drive can also be achieved in a variety of ways with varying numbers and sizes of wheels that augment their performance.

4in wheels provide greater speed as per each rotation the robot moves farther, however they give the robot less torque. Additionally since less of them can fit onto a drive with the 18*18*18 size limit there is also less points of contact. Additionaly, with the older 4in wheels the team currently owns, the traction versions are .125in smaller then the omni versions.

3.25in wheels provide slower speeds, but are able to give the robot more pushing power as they have more torque and points of contact. These wheels are also easier to work with, as the traction and omni versions are the same size, unlike with the older 4in wheels.

Omni wheels have rollers that allow the wheel to move side to side as well as forwards and backwards. This makes them great for turning, but poor for traction.

Traction wheels wheels are all rubber and provide exceptional ground adherence for any robot, however they greatly limit turning, making them impractical unless used as the middle wheel where their effects on turning are mitigated.

From our teams experience, a 3.25in drive with 2 traction wheels in the middle and 2 omni wheels on either end appears to be the optimal way to execute this drive. Our previous drive with three 4in omniwheels failed to push back against other robots that were using 3.25in Tank Drives with the same amount of motors. These robots were also just as fast and maneuverable as ours showing little trade off for this design.

Pros	Cons
• Simplicity • Versatility • Easier to Control	• Limited Mobility • Wheel incompatabilities

H/X Drives

These use either 4 or 5 omni wheels to achieve a robot that has the same range of motion as a Tank Drive, but with the additon of diagonal and horizontal movements. In the case of an X drive, they use

either 4 indivudally powered omni wheels in each corner at 90 degress from one another or 4 indivudallty powered omni wheels in a traditional tank drive setup with one horizantal omni wheel for the H drive. These drives can however, prove dificult to control and in the case of the H drive, it is impractical because the horizontal wheel rarley makes contact. They are also very easy to push around since all the wheels are omni. X drives can prove highly practical given the right game and design but in games such as over under the middle bar limits their use.

Pros	Cons
• Maneuverability • Complex Autonomous • Strafing	• Mechanical Complexity • Motor Usage • Practicality • Low Traction/Easy to push

Mecanum Drives

Mecanum drives are likley the most special as they use specialized mecanum wheels. Similar to omni wheels, these have rollers attached, but at an angle to provide uniquely augmented movement. When set up correctly, 4 indvidually powered mechnum wheels can provide the same movment as an X drive. However, since to go in any direction it directly turns the mecanum wheels, they are harder to push as the motors resist the pushing directly. This, along with other issues, can lead to faster overheating with mecanum drives. The Vex EDR 4in mecanum wheels are very bulky putting more strain on the motor. Additonally, the vex mecanum wheels unlike most designs have limited contact with the ground due to the irregular design of their rollers. It is also important to note that it is easier to gear and build a frame for a mecnum drive over an X drive as it does not require the 45 degree angles to achieve its unique motion. Though also possible with an X drive, an additional powered omni wheel could be put into the middle to provide additional drive power. Since this wheel is not needed when a successful PTO can be developed, it could allow for a very versatile robot and drive.

Pros	Cons
• Maneuverability • Complex Autonomous • Strafing	• Mechanical Complexity • Motor Usage • Practicality • Requires balanced weight

Swerve Drives

Previously considered impractical, Vex Swerve Drives involve either 3 or 4 independently steered and powered wheels. These focus around modules that can both spin and rotate the orientation of the wheels. This allows for the robot to turn rapidly as well as turn while moving. The wheels can be positioned in the manner of a tradtional tank drive for linear movement and then turned to go in the desired direction. However until the addition of the 5.5W motors, these would either use 6 or all 8 of the robots available motors. The 5.5W motors now allow for this drive to be possibly practical as a 3 wheel Swerve Drive could be made from three 11W motors and three 5.5W motors allowing for 38.5W of motors to be allocated to the robots mechanisms and manipulators. The advantages of swerve drives can be seen from other competitions like FRC, where they are often used to great success to create highly maneuverable robots. The use of a Swerve Drive within Vex would be highly dependent on the

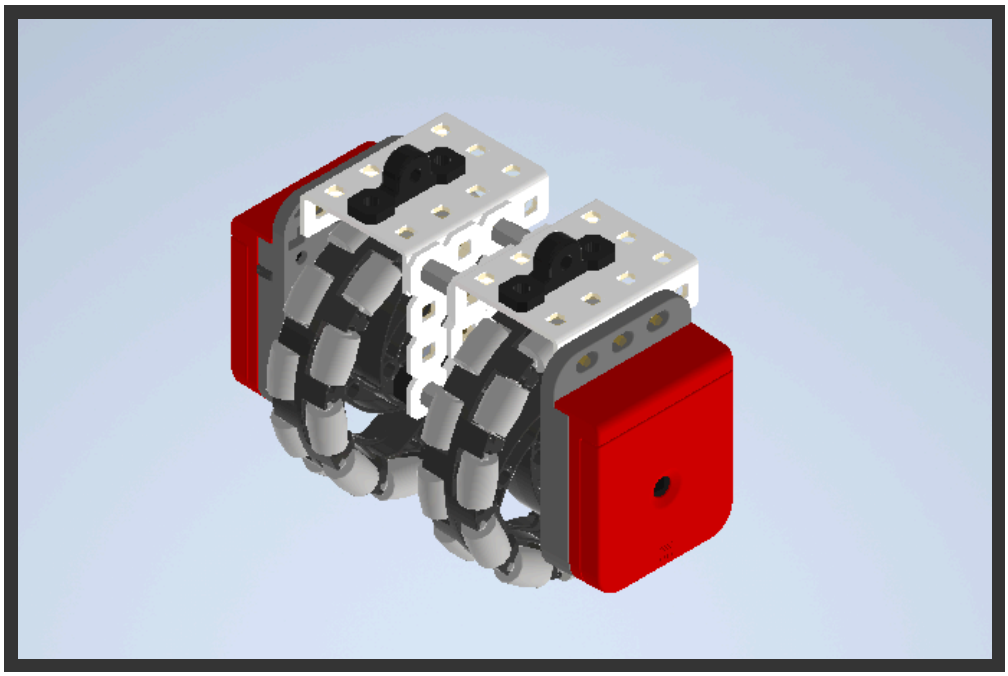
game as one with limited room to move, such as over under, takes away many of the Swerve Drives advantages. It is worth creating a model for a swerve drive module in case the next game is one that prioritizes movement. It would also provide practice using more complex gearing which the team has yet to experiment with.

Pros	Cons
• Maneuverability • Complex Autonomous	• Mechanical Complexity • Motor Usage • Practicality

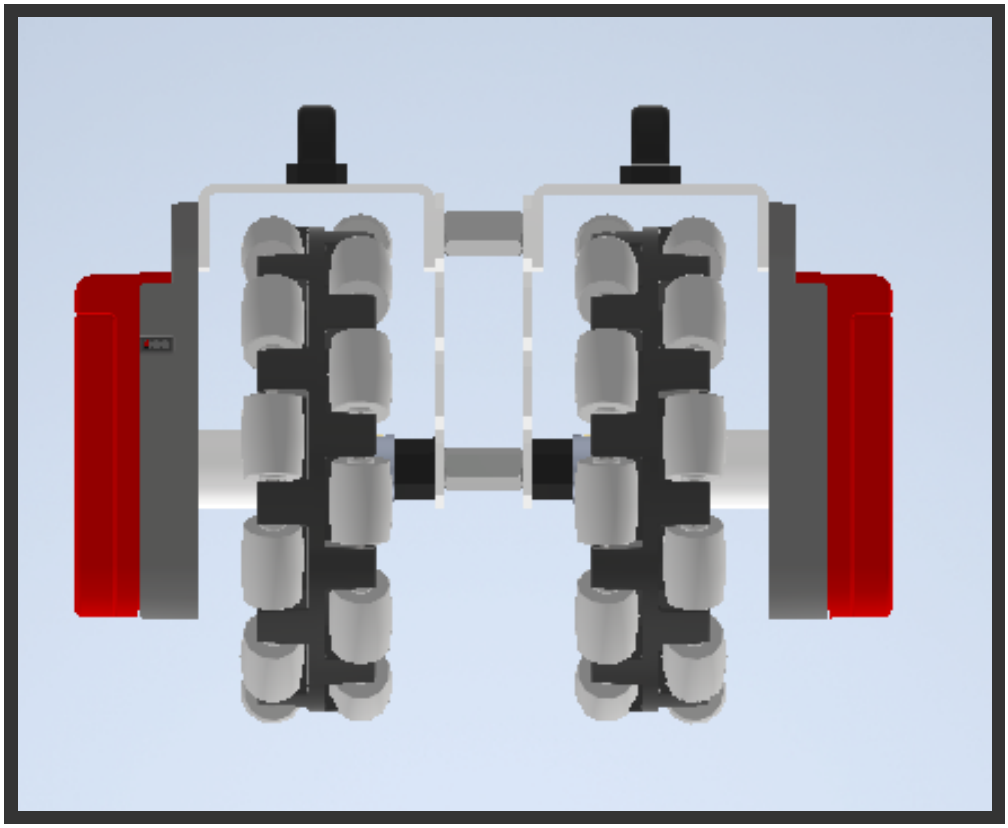
Odometry is a position tracking algorithm used by the coder to implement complex autons. It relies on three sensors 2 vertical and 1 horizontal. The failures of the previous design were compactness and resilience and the new designs makes a few improvements to this area. It is important to complete this first as any prototype drives made must be designed to fit the sensors. This along with a basic mechanum drive which can act as a tank drive when need be will allow the coder to begin making some basic frameworks for next year.

New Design:

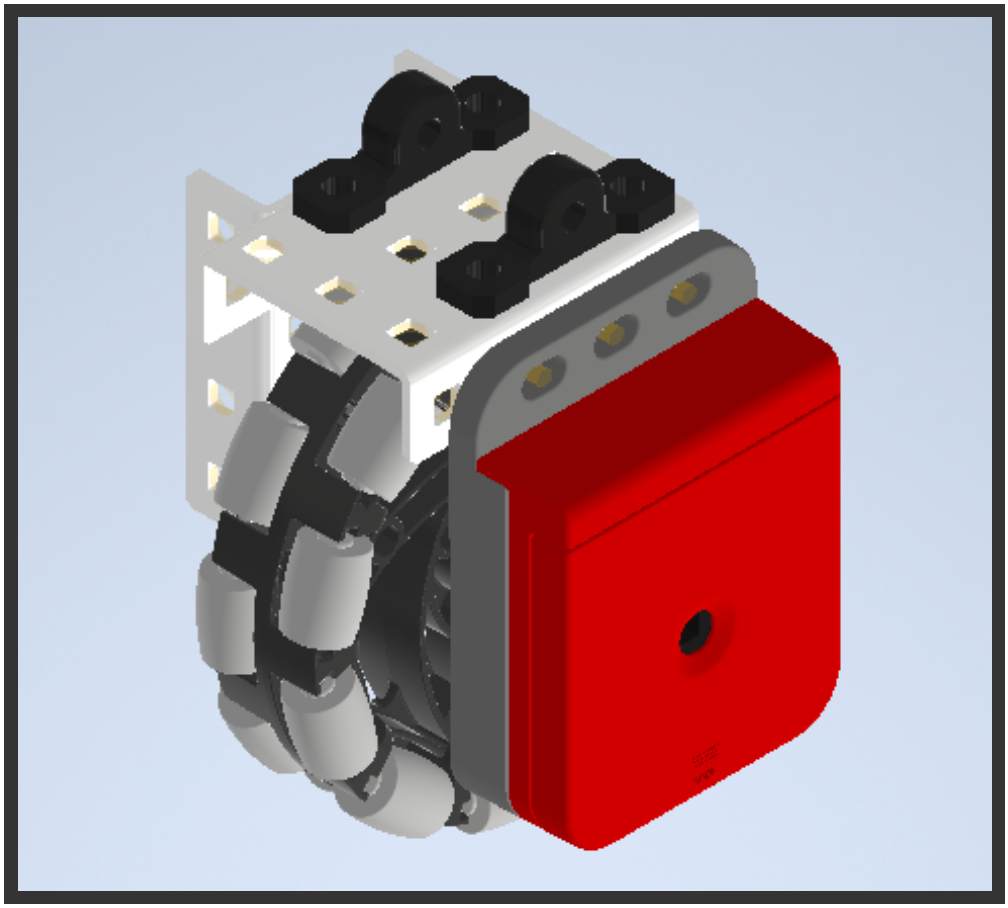
- Vertical Wheels save space by being in the same module
- No plate is used without being reinforced
- Pillow bearings used to simplify mounting
- Newer 3.25in wheels used for better traction



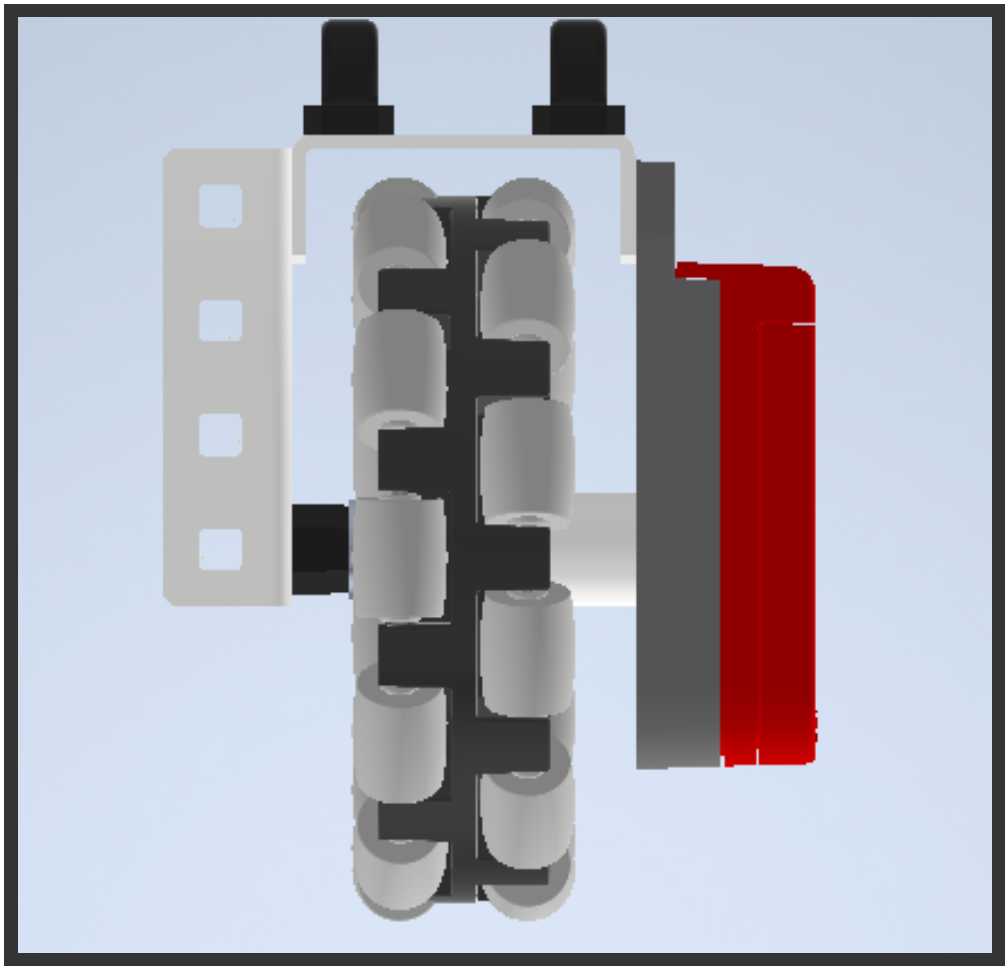
Isometric View of the New Vertical Odometry Sensor



Front View of the New Vertical Odometry Sensor



Isometric View of the New Horizontal Odometry Sensor



Front View of the New Horizontal Odometry Sensor

]

With inventory taken we can now begin to make some designs. To expirment with more complex Drives a Mechnum drive and a Swerve Drive module were completed. These should provide good practice for designing before the next season as well possibly giving us a head start if we choose to use these drives.

The first model made was that of a Mecanum Drive:

- 4 Mecanum wheels geared to 300 RPM with a 72:48 ratio driven by an 11w motor with the 600rpm cartrige.
- 24in HS axels with holes drilled in them as the main frame to ensure it was stronger then our previous drive trains.
- Center Omni Wheel for additional power
- PTO to allow for the cneter omni wheel to power other system while not in use
- Battery and Air Tanks kept low to ensure a proper center of gravity
- New Odometry Sensor fitted in the rear

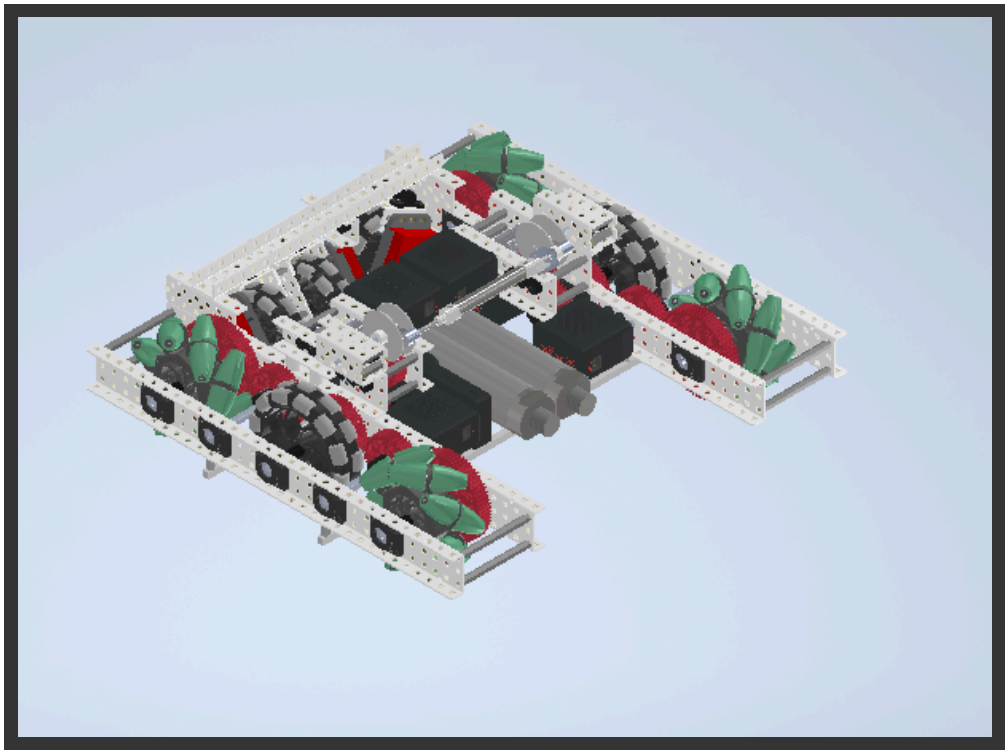


Figure 9: Isometric View of the Prototype Mecanum Drive

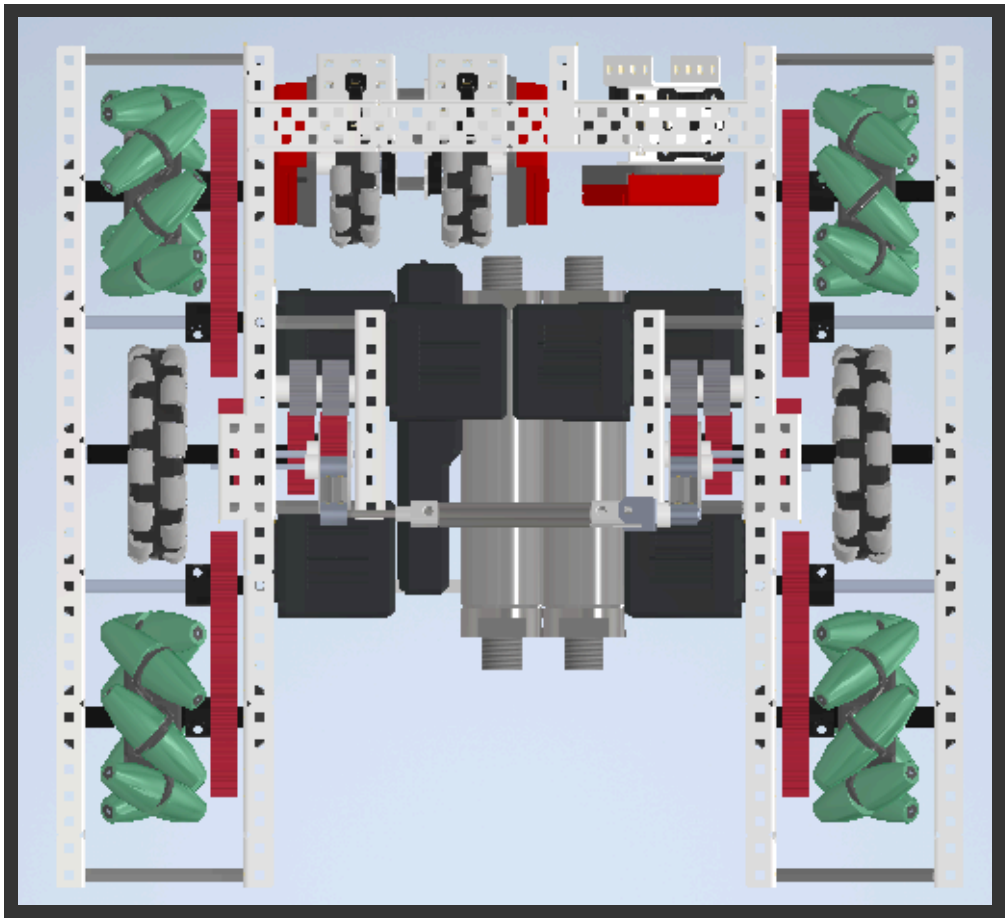


Figure 10: Top View of the Prototype Mecanum Drive

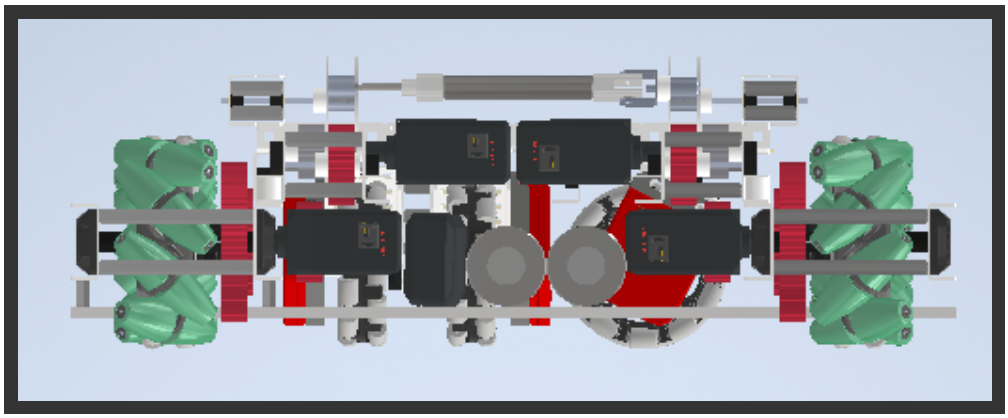


Figure 11: Front View of the Prototype Mecanum Drive

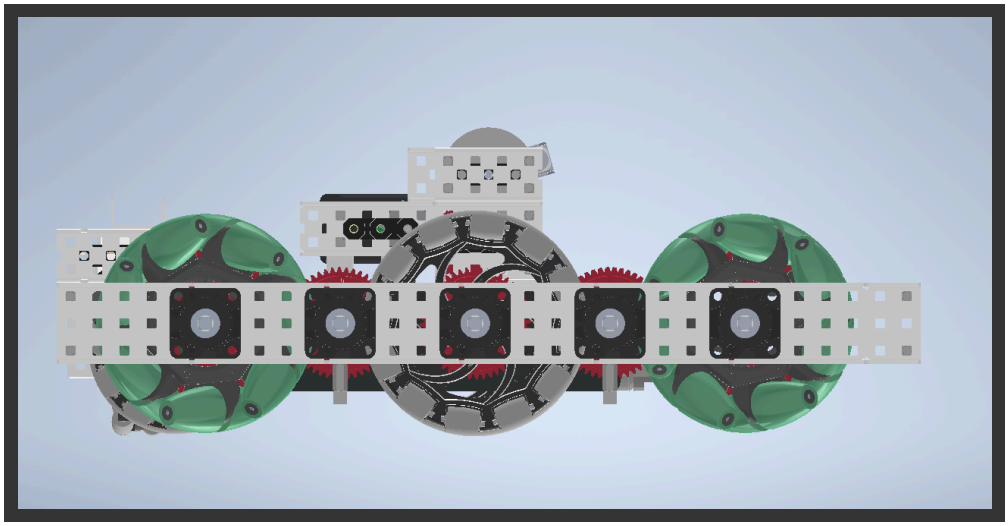


Figure 12: Side View of the Prototype Mecanum Drive

Before we could enter school to test this I created a Model for a Swerve Drive module:

- Keeps design compact with motors below the frame
- 72 tooth gear is screwed to the frame so they spin together
- Circular insert within the gear to allow the drive shaft to turn
- Chain runs to connect the Drive Shaft to the 11w motor
- 5.5w motor used to turn module

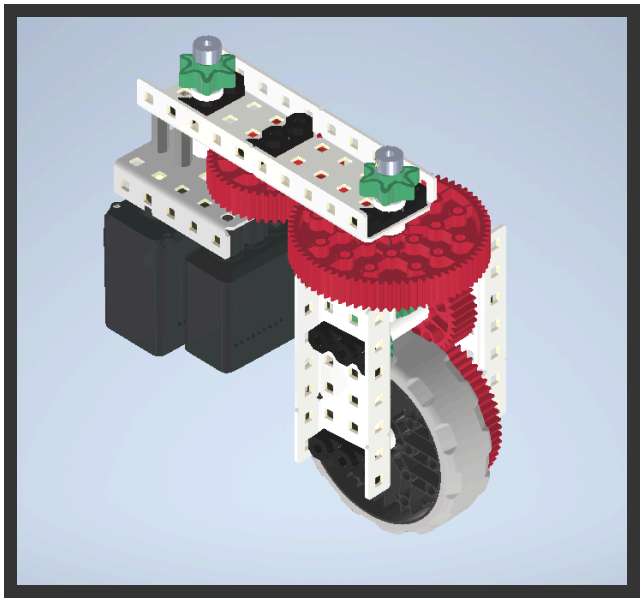


Figure 13: Isometric View of the Prototype Swerve Drive Module

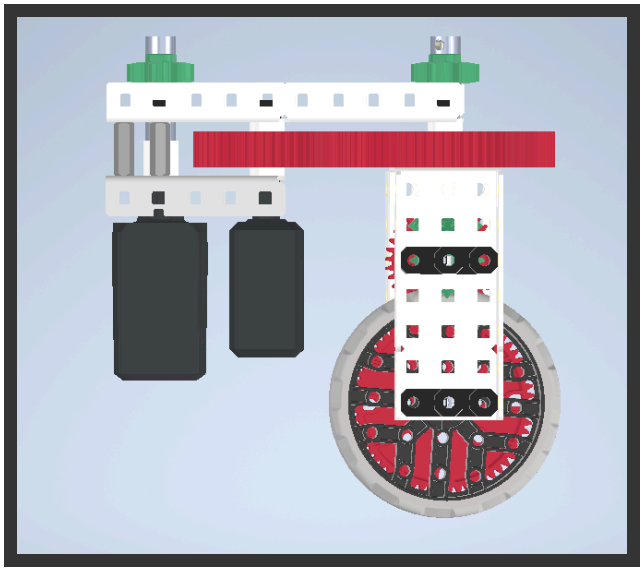


Figure 14: Side View of the Prototype Swerve Drive Module

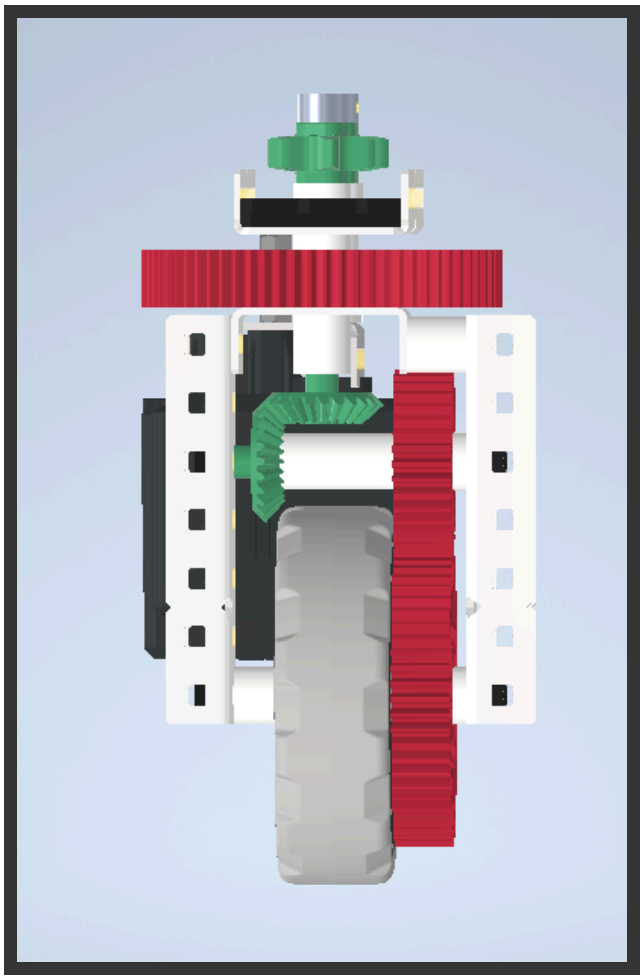
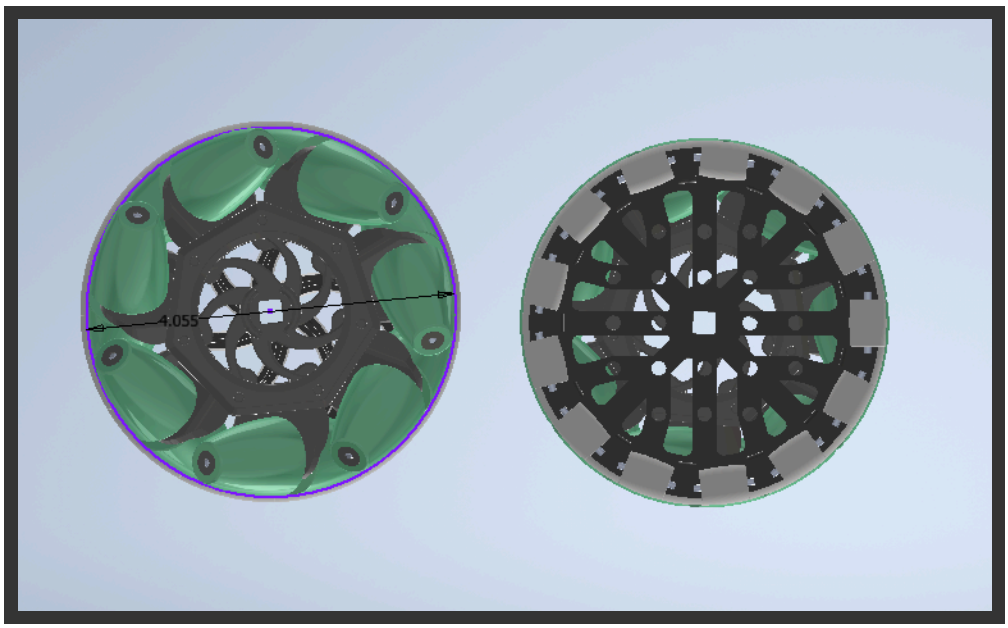


Figure 15: Front View of the Prototype Swerve Drive Module

> **Note**
>
> It is unlikley any of these drives will be used as tank drives have proved superior for many games in a row. They simply serve as a way to practice building and design techniques and mechanisms before the next season. However, cataloging them is still important as the ideas learned from them could prove very important.

While Building the Mecanum Drive a large oversight was made. Vex wheels may be advertised in standard sizes like 4in but that is seldom the case. For the wheels various forum posts and the Perdue Vex Sig Robotics website placed both the older 4 in omni wheels and 4in mecanum wheels as having a 4.125in diameter. However once we built the drive we found the omni wheel to be slightly bigger. Going into inventor confirmed these as the Mecanum wheels measured .0625in smallers then the omni Wheels.



Comparison of Mecanum Wheels and both the new (right) and old (left) omni wheel diameters

This means that for now, until we can get the newer omni wheels that should be compatible with the mecanum wheels and align with the PTO attached to them, they will be left out of the build. The older wheels can not be used as they cause only one of the mecanum wheels to be in contact with the ground which defeats the purpose of the drive.

Additionally weight plates needed to be screwed into the front to ensure the drive was balanced allowing it to properly strafe.

Overall the Drive was a success after a few minor tweeks and will serve as a good test base until the season starts.

Since the most important task of inventorying was completed, most of the team agreed to slow down until the next game was revealed in order to study for upcoming MCAP/AP/Final exams and other schoolwork.

However, when we had time to stop by Robotics, we worked on disassembling the previous game's field and repairing the field border. This was necessary because the field border had been severely weakened from numerous rams during our autonomous practices in the previous season.

During a group meeting today, we discussed a variety of fundraising methods. All ideas were listed on the right side of the board (even jokes), and the potentially viable ideas were written on the left. A fundraiser would provide us with early access to the field, allowing us to get more practice. During the previous year, the team worked with a 3D-printed tri-ball and a makeshift field made from old parts until around November. To avoid this, a fundraiser that raises at least six hundred dollars would allow us to secure the field months before the county might decide to purchase it for us.

Out of the many options we considered, we decided to try a simple GoFundMe campaign and send it to our families. This would require minimal input while offering the possibility of a large output. We promised that any donors would receive a resin-printed Eastern Tech keychain to encourage more donations. All team members sent out the link to friends and family to raise funds.

To start, an intake with a lift was modeled along with initial ideas for the drive. This was done so that we had a good starting point to begin building when we met again later that week. Additionally, the drivetrain allows us to start making considerations for motor distribution and the sizing of our manipulators.

The intake is designed to use a chain belt with standoffs screwed in that will hook onto the center of the rings and pull them up, as well as the lift to drop them off at various heights on the stakes.

The drivetrain includes a variety of features, some of which had been worked on during the preseason:

- 6-motor drive with a PTO to transfer two of the motors to a lift.
- 4 3.25-inch omni wheels for traction and maneuverability.
- Vertical motor mount to save space.
- HS axles used to secure the two halves of the drive.
- Pneumatics with Lexan washers used to shift the gears in the PTO.
- Odometry sensors to allow for an improved and more accurate autonomous routine.

Overall, this is just a quick model put together and will be further explored as the season continues and we begin to construct the robot.

Funraising Results

2024/05/01

The fundraiser is still ongoing, however 765 dollars was raised which is able to cover the field costs as well as shipping, including tax, whcih allowed us to puchase the field today. This should allow us to get started far sooner then in previous years and allow for the testing of more protypes before the first competitions.

We started by building the basic intake modeled in Inventor and worked to create various improvements as we encountered problems. (It is important to note that these tests were done with a 3D-printed ring, so the weight issues encountered may not be entirely accurate.)

Initial tests using a standoff intake similar to those in Tipping Point showed that the chain flexed and dropped the rings because they were too heavy. To adjust for this, flexwheels were tested and found to work very well. However, as we increased the incline of the intake, the flexwheels became less effective at picking the rings up from the ground, though they still carried them up once the intake secured them.

To alleviate this, standoffs were added to the center of the initial stage of the intake. This helped, but it occasionally caused jamming, which was fixed by making the first stage of the intake float. This allowed it to rotate and enable the disk to enter more smoothly. After multiple changes to the gripping devices, a final design was settled on. (Picture of different intake rollers/chains with descriptions and charts of each one's ability to intake.)

Odometry, also known as Dead Reckoning, is an algorithm, that allows us to calculate our global position by summing up small local movements. In the context of VEX Robotics, it is more commonly used to refer to a method in which the movement of wheels at a known distance away away from the tracking center is used to calculate the change in position of the robot, this is then transformed from local coordinates to global coordinates and accumulated for a live readout of the robot's global position.

The specific equations for odometry aren't too complicated. It is mainly derived from the equation of arc length, in particular, we use $l = \theta * r$ where we have l and are looking for θ and r by using each wheel. Doing some algebra gets us the following equations

$$\Delta\theta = \frac{\Delta L - \Delta R}{s_L - s_R}$$

$$\Delta\vec{d_l} = 2\sin\left(\frac{\theta}{2}\right) * \begin{pmatrix} \frac{\Delta S}{\Delta\theta} + s_S \\ \frac{\Delta R}{\Delta\theta} + s_R \end{pmatrix}$$

where ΔL, ΔR, and ΔS are the distances the left, right and back tracking wheel moved respectivly, the back tracking wheel being perpendicular to the other 2, s_L, s_R, and s_S being their distances from the tracking center, $\Delta\theta$ being the amount the robot turned, and $\Delta\vec{d_l}$ being the amount the robot moved, as a vector. However both $\Delta\theta$ and $\Delta\vec{d_l}$ are in local coordinates based of the robot, we need to transform them into global coordinates, which we can do using the following equation.

$$\vec{p}_{t+1} = \vec{p}_t + \Delta\vec{d_l} \begin{pmatrix} -\cos\left(\theta_t + \frac{\Delta\theta}{2}\right) & \sin\left(\theta_t + \frac{\Delta\theta}{2}\right) \\ \sin\left(\theta_t + \frac{\Delta\theta}{2}\right) & \cos\left(\theta_t + \frac{\Delta\theta}{2}\right) \end{pmatrix}$$

$$\theta_{t+1} = \theta_t + \Delta\theta$$

The equations above may seem complicated but really, all its doing is rotating the vector $\Delta\vec{d_l}$ by the average of θ_t and θ_{t+1} (we do the average because after some testing, we found it to be more accurate than just doing one or the other) and what this does is convert the local displacement into global displacement. We can then just add the new global displacement to the old global coordinates, and $\Delta\theta$ to the old theta, and that gets us the new global coordinates.

When implementing the odometry algorithm, we have a few options:

	Simplicity	Extensibility	Experience	Interface	Total
LemLib	15	4	3	8	30
OkapiLib	9	10	3	4	26
Custom	3	10	5	6	24

According to the decision matrix, using LemLib's odometry would make the most sense to use. And as Praful plans to contribute to that library, our familiarity with it will quickly grow, and we can more easily extend the library.

With LemLib we have an option to use an inertial sensor instead of 2 parallel wheels, and it is advertised as being more accurate. We didn't quite trust this so we decided to do our own testing. Using an unpowered drivetrain with the odometry sensors on the back, with similar positioning to our Boosted Mechanum sensors. We then measured the error over time between the true position, and the odometry reported position. During each test, we rotated the drivetrain in the following order.

1. 90° CW
2. 180° CCW
3. 45° CW
4. 135° CCW
5. 360° CW
6. 720° CCW
7. 540° CW

After each turn we measured the deviation from the actual position and recorded in the following table

Trial	Expected	Inertial	Parallel Wheels
1	90°	89.8°	92.1°
2	−90°	−91.4°	−86.4°
3	−45°	−45.2°	42.1°
4	−180°	−180.5°	−171.9°
5	180°	180.7°	191.4°
6	−540°	−541.0°	560.4°
7	0°	0.2°	24.4°

Table 1: The test results for comparing the inertial measurement unit vs. the parallel tracker wheels' accuracy

As we can see, the inertial measurement unit is significantly more accurate than the parallel tracking wheels. They both start out with around the same amount of error, however, the parallel tracking wheels accumulate in error, while the IMU, has a static amount of error. Therefore we can conclude that

LemLib's claim is in fact correct and using an IMU is more accurate than parallel tracking wheels. However there is some room for argument as the distance from the tracking wheels and tracking center may not be as accurate as possible, which might account for the error.

LemLib also has an existing PID class, so implementation is not strictly necessary, however understanding how PID loops work will greatly improve our ability to tune them, and generally understand what the robot does. PID loops stand for Proportional, Integral, and Derivative, which represent how the PID loop provides feedback.

As input, the PID loop gets the desired position, or setpoint, and the current position of the motor or system. The PID loop then uses the following formula to provide feedback to the motor or system, where K_p, K_i, and K_d are tunable parameters, and $e(t)$ is error or $\text{setpoint} - \text{position}$.

$$K_p e(t) + K_i \int e(t)\,\mathrm{d}t + K_d \frac{\mathrm{d}e}{\mathrm{d}t}$$

This formula may look complicated but when we break it down, its actually pretty simple. Lets look at each part separately.

Proportional

This first part, $K_p e(t)$, is relativity simple, we simply take $(\text{target value}) - (\text{sensor reading})$, call it error, and multiply it by a constant, K_p, to get the output that gets passed into the motor. This term is the main driver of the PID loop, so it's constant will be the largest. The more of the proportional you add to the loop, the quicker the loop will get to the target, however the loop will also overshoot the target more and oscillate around the target.

Integral

The next part, $K_i \int e(t)\,\mathrm{d}t$ looks a lot more complicated, however all its really doing is just accumulating the error, it looks much simpler in code:

```py
while (error > 0.5):
    error = target - sensor
    totalError += error
    motor.move(Ki * totalError)
```

As you can see all this term is doing is just keeping track of how much error the loop has had throughout its runtime and using that to drive the motor, instead of just using the error.

However this has a couple of problems as is, the first problem is that when we go to a point that's far away, the integral term will have a lot of time to accumulate, so when it reaches the point, the integral term will cause a lot of overshoot, we can fix this, by resetting the integral term when the accumulation gets too high or we reach the target point, as we don't want to overshoot. This term still increases the overshoot though, and increases the time it takes to settle to the setpoint, however it greatly improves steady state error, so you want to use it if the PID loop it settling on a point that's not the setpoint.

Derivative

This term, $K_d \frac{\mathrm{d}e}{\mathrm{d}t}$ also seems complicated however like the integral term its better thought of in code:

```py
while (error > 0.5):
    error = target - sensor
    derivitive = error - prevError
```

```
    prevError = error
    motor.move(Kd * derivitive)
```

As you can see, all this term is doing is just getting the change in error since the last iteration, and using that to drive the motor.

This term is most useful when it comes to correcting sharp disturbances that happen, for example, if the robot is moving, and it gets pushed, the derivative controller will quickly correct that. This term decreases the overshoot and settling time, but if it gets too large it can harm the stability of the loop. So its best to keep the derivative term small.

Put it All Together

To bring all of the terms together, we can add them together, and this will bring their respective strengths and weaknesses to the loop, creating a versatile control algorithm. We do need to tune the PID though, and that can be complicated, especially when you don't have a graph showing error.

Tuning

Tuning the constants for the PID loop can seen complicated, but if you have the right tools, its suprisingly easy. Figure 17 shows the procees i deveised for tuning my PID constants.

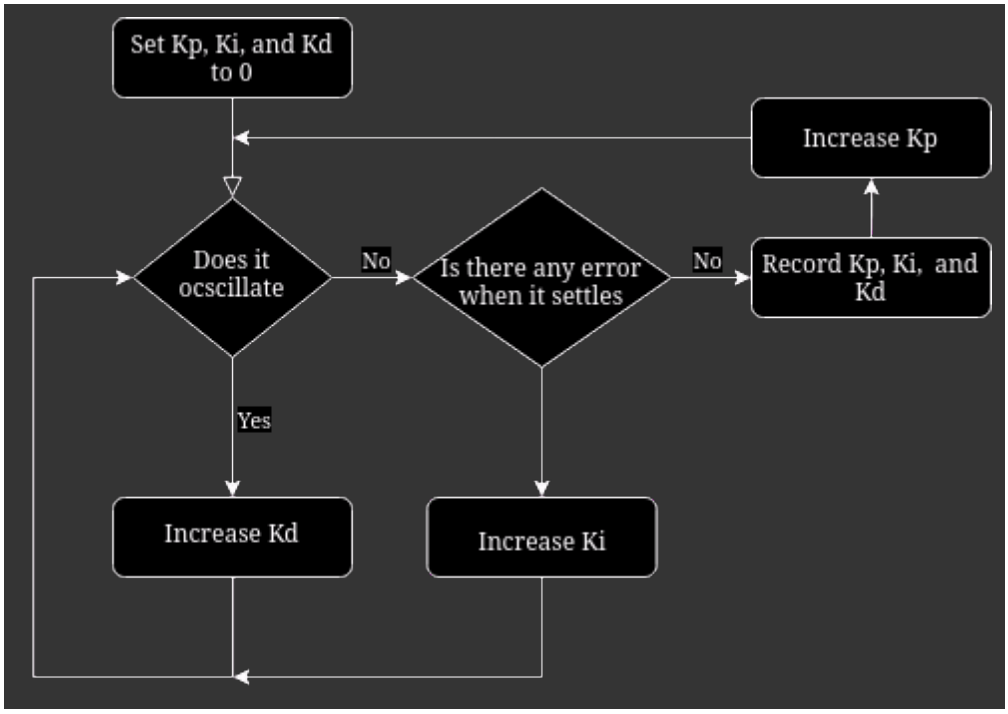


Figure 17: A Flowchart for tuning PID

Answering the questions in the flowchart becomes much easier if you have a graph of error vs time, as the characteristics in the graph are very distinctive, if you are just looking at the robot, it can be very unclear at times if the PID is properly tuned or not, which is why we decided to make a GUI that will help us tune the PID controller.

Pure Pursuit is one of the more advanced control algorithm that we'll be using. It is an algorithm that can follow a curved path, using both odometry and PID. The algorithm does this by chasing after a point on the path that is in front of the robot, this is to allow the robot to smoothly follow the path, making it much more versatile than basic autons where you first move in a straight line then turn, and repeat. Also moving in a curved line is more efficient than the start and stop motions because a curved motion allows you to smoothly move around obstacles without having to stop and turn multiple times, which would make the motion significantly faster. Curved motions also allow for better error correction as both turning and moving are handled at the same time, with stop and start motions, you may drift in angle while moving straight which cant be corrected with stop and start motions, but can easily be corrected with curved motions like pure pursuit.

LemLib just so happens to also have this algorithm pre-implemented, so I just have to understand how it works,or at least I have to relearn it, as I made a working pure pursuit algorithm in the previous competition year.At a high level, Pure pursuit works by taking the robot's current position, and finding a point that's further along the path, at a specified distance away, often called the lookahead distance. The robot then drives towards that point using normal differential drive kinematics, and repeats the whole process of finding the next point.

Now to go more into depth. The control flow of pure pursuit looks something like Figure 18

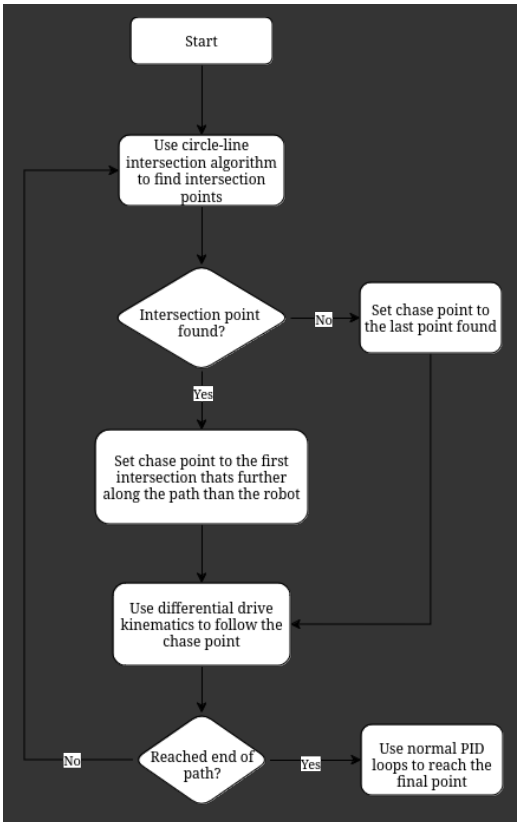


Figure 18: The control flow diagram for pure pursuit

Ok so lets go through each box, first, we have the circle line intersection formula, according to *this article* the formula is

$$\begin{pmatrix} \frac{D\Delta_y \pm \operatorname{sgn}(\Delta_y)\Delta_x\sqrt{r^2d_r^2-D^2}}{d_r^2} \\ \frac{-D\Delta_x \pm |\Delta_y|\sqrt{r^2d_r^2-D^2}}{d_r^2} \end{pmatrix}$$

where

$$D = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = x_1y_2 - x_2y_1$$

$$\Delta_x = x_2 - x_1$$

$$\Delta_y = y_2 - y_1$$

$$d_r = \sqrt{\Delta_x^2 + \Delta_y^2}$$

This may look complicated, and it actually is 😔. So we're not going to explain it, we'll just explain how its used. We first check the discriminant, which is the part under the sqrt: $r^2d_r^2 - D^2$ and check if its ≥ 0. Then we run the formula and get the points of intersection, we then test each point and check if they are within the line segment's bounds.

In the next part, if there is an intersection, we record the intersection that is the next point to come after the point the robot followed previously, or the closest point to the robot if no such point exists. For example in Figure 19, if the point the robot followed previously was the green point, the robot would now follow the red point, the robot being the black dot and the circle around it being the lookahead distance.

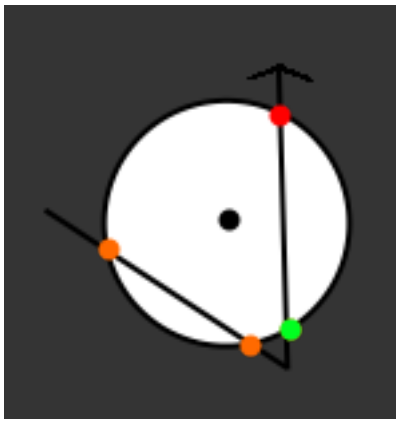


Figure 19: An example scenario that would result in having to choose between multiple points

However if there is no intersection, we follow the point the robot followed previously, or if there is none, the point closest to the robot. The only condition where there is no intersection, is if the robot somehow strayed from the path, in which case, the best option is to move towards the point that was last calculated to the in front of the robot.

After we get the point to follow we now need to follow it, to do this, we can use the kinematics of a differential drive and derive the inverse kinematics:

$$v_l = \frac{l - r}{2}$$

$$v_r = \frac{l + r}{2}$$

Where v_l and v_r are the left and right side's wheel velocities respectively, l is the linear velocity, and r is the rotational velocity. We then move the wheels using this velocity, which is calculated using the path, and it's curvature beforehand.

We then check if we are at the end of the path, and if we aren't, we go back and find the next intersection. But if we are, we stop the pure pursuit algorithm and switch to a PID loop, where we run both an angular PID and a linear PID loop to accurately get the robot to the final position.

The boomerang control algorithm is one of the most useful algorithms, that we will use in our autonomous. It is also one of the more simpler algorithms. This algorithm is used when you want to move from one point to another, but you also want to have a specific heading at the end of the motion, so just using a normal PID loop won't work. You also won't want to use pure pursuit when we use boomerang, as although the replacement could be made, the boomerang control algorithm is mainly used when we have just a small motion to make, so it wouldn't make sense to create a pure pursuit path solely for that one small path, this would clutter up the file structure, make iterating on the autonomous more tedious, and also make the code look more opaque as the code of pure pursuit does not make it clear where the robot will end up.

The boomerang control algorithm is also pretty simple, however it is very obscure, only being used in the LemLib codebase as far as we could find. It is simply a modification to the PID loop, the point the PID loop is chasing moves in order to make the robot reach the point facing the direction that is wanted. Lemlib provides a nice *desmos graph* which demonstrates the algorithm's inner workings.

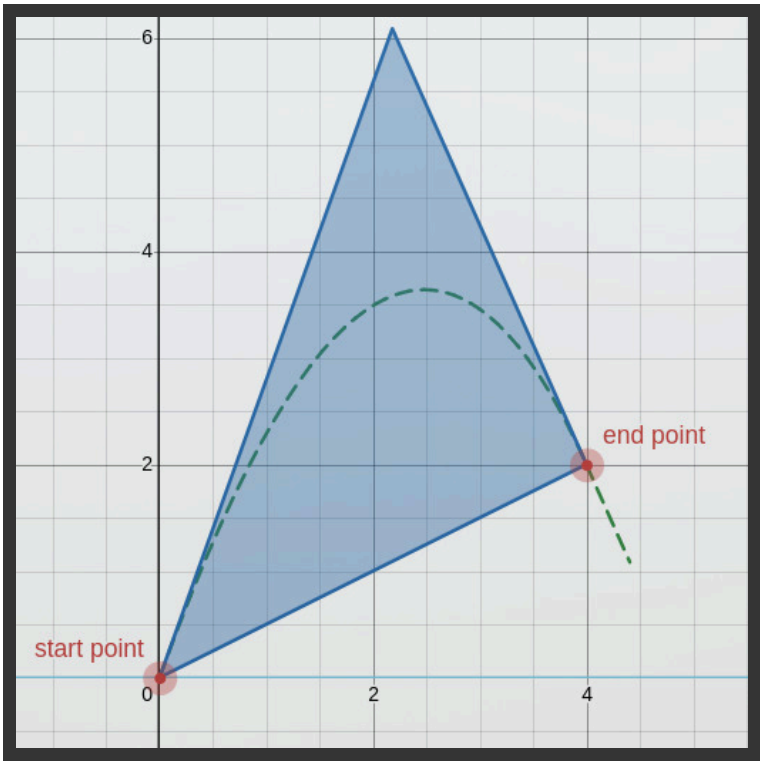


Figure 20: an example boomerang path with $g_{\text{Lead}} = 1$

During the summer months, our main programmer, Praful, had a lot of free time, which he used in developing 2 main things, a GUI for easier interactions with the robot, and a series of control algorithms for a better overall autonomous. And as he was travelling for the first part of the summer, the GUI seemed like a better option to do without a physical robot because, for one the library that PROS uses to display things on the screen, LVGL, has a simulator that we can run on my laptop and quickly iterate on the GUI. Well at least that's what we thought when we started to work on the GUI.

We started by first trying to design our own autonomous selector which didn't end up too well with our limited knowledge of LVGL. Our results were mediocre at best which can be seen in Figure 21 and the code wasn't very extensible.

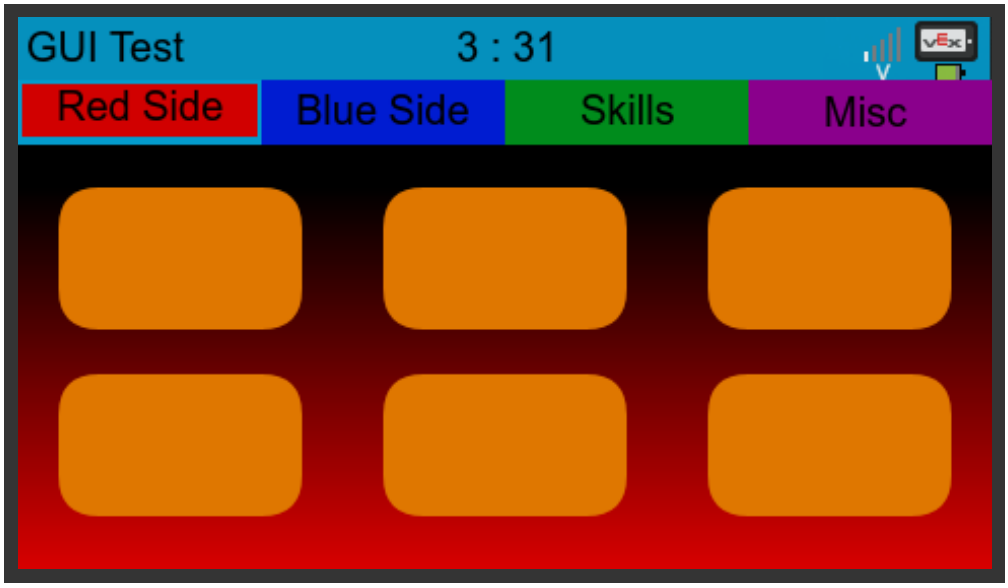


Figure 21: Our first attempt at an autonomous selector

So we started researching into existing auton selectors and we found a library that is extensible and perfectly fits our needs: *robodash* , it already has a debug console, alert system, image display, a well made view switcher, and a autonomous selector that looks good which can be seen in Figure 22.

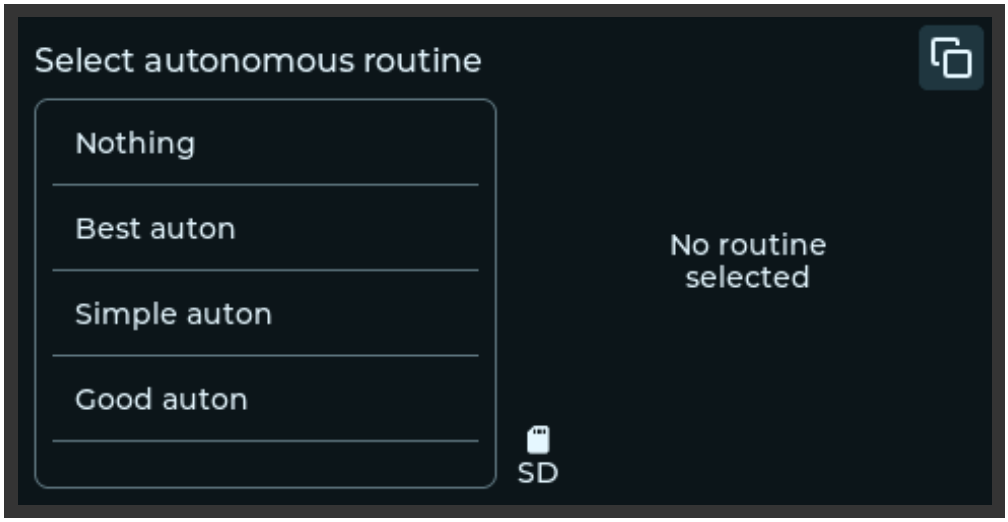


Figure 22: Robodash's default auton selector

We liked a lot of Robodash's auton selector, namely its clean UI and straightforward interface. Category there was one thing we didn't like, and that was that all the autons were dumped together in one category, it would be nice if each side had its own category like our original attempt. So we decided to add that ourselves. This was not without its own difficulties, but before that, a quick lesson on LVGL. So LVGL is written in C, so it's not object oriented, however the structure of the code is such that it would make sense for it to be written with an object oriented structure, which can be seen in the example below

```cpp
// This creates a pointer to the tabview which is a "child" of the screen
lv_obj_t * tabview = lv_tabview_create(lv_scr_act(), LV_DIR_TOP, 50);

// this sets the background color for the tabview
lv_obj_set_style_bg_color(tabview, lv_palette_lighten(LV_PALETTE_RED, 2), 0);

// this creates each tab of the tabview
lv_obj_t * tab1 = lv_tabview_add_tab(tabview, "Tab 1");
lv_obj_t * tab2 = lv_tabview_add_tab(tabview, "Tab 2");
lv_obj_t * tab3 = lv_tabview_add_tab(tabview, "Tab 3");

// now to add some content to the tabs
lv_obj_t * label = lv_label_create(tab1);
lv_label_set_text("Hello World");
label = lv_label_create(tab2);
lv_label_set_text("Hello World... Again?");
label = lv_label_create(tab3);
lv_label_set_text("Goodbye World");
```

As you can see there is a lot of structure and repetition in the code which could be easily avoided if LVGL was object oriented. Alas it is not, and its the only option, so we continue. Notice how all of the `lv_obj_t` are objects, regardless of what kind of object, a tabview or label, and this lead to a lot of errors initially, before i started labeling the type of object in the name of the variable, `tabview_autonType` instead of just `autonType`. The most common error we encountered however

was a null pointer dereferencing error, and as you can see all of the object are pointer, and there are a lot of objects, so this occurred most often when the object was not initialized and we tried to use it as a parent of another object, or set it's text or the like.

So now back to the topic, updating the robodash auton selector, to do this it was pretty straightforward after i got though all of the errors. I just convert the container into a tabview then duplicate the list of routines, so we have one for each of the red side, blue side, skills and misc, then we add an enum to the interface to indicate where each auton is going to go, and then we sort each of the autons into their respective tab. Really my inexperience is the only thing that prevented me from doing it faster. And you can see the result in Figure 23

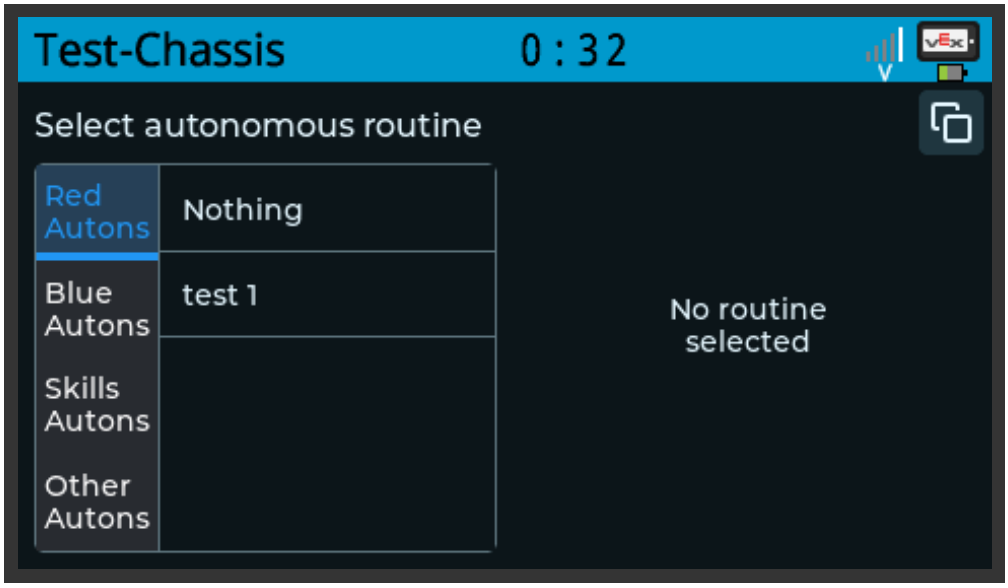


Figure 23: Our custom fork of Robodash

So how useful is this? In our opinion it is very useful, last year, we ran out of slots on the brain to store our autons and we had to scramble to quickly reupload the right auton before our match, our auton selector would completely elimineate the need for more than one auton slot as all of the autons are stored in one program.

And for our next steps, we still need to create other GUIs for different things, but regarding the auton selector, the next step is to create an auton builder, where we can toggle parts of autons. This would give us immense amounts of flexibility when it comes to working with our alliance partner, as we can build autons to optimize the number of points we get while still getting the Autonomous Win Point.

Another one of the GUI programs that we wanted to make was a way to intuitively see the odometry data, this would make it significantly easier to create and debug autons. We wanted the program to display a picture of the field, or at least a simplified version of it, and have a toggleable robot that is to scale, shown moving around the field. We also want it to display paths written in the *path.jerryio* path creator. Also having a function to record and save the robot's motion would be a very helpful feature.

To do this we first planned out the GUI with the same color theme as the auton selector, and orange as accents. We used an online image editor to do this and used layers to represent different modes. Our finished prototype can be seen in .

hi

To code the basic functionality, i referenced theo0403's *odomDebug* library. And for the interface I decided to look for the images in a specified path on the SD card with an optional parameter to change that path, and a similar idea for the path.jerryio paths and recorded paths. The only required input is a function to get the current position, or a reference to the LemLib odometry. I allowed a reference to the LemLib odometry class as I plan to release this GUI publicly after the season's over and I contribute to LemLib so I can advertise my library to the people there and gain a not insignificant userbase.

While coding, I followed the style of robodash's existing views to minimize errors and maintain readability. This allowed me to realtivily quickly code the GUI and make it work, which can be seen in .

hi

At this point I was getting more and more ideas on things I want as a GUI, but I decided to stop after the PID Tuner and Simple Text Display to prevent feature creep.

Last year, we had a working odometry system, and a working PID loop, however, our PID loop was inadequately tuned so the main advantage of using PID, better accuracy and speed, was effectively canceled out by the sub par tuning. So the purpose of the PID tuner is to create a GUI that allows us to tune the PID significantly faster than without, and have a better PID tune overall. To do this we want a GUI that lets us change the PID values without having to reupload the program, and also give us graphical feedback, ideally in the form of an error vs time graph.

After some planning we decided that out interface with the GUI would look something like this

```cpp
struct PIDToTune {
  public PIDToTune(std::string name, std::function<void(float,float,float,float)>
runFunc, std::function<float(void)> errorFunc)
}

PIDTuner(std::vector<PIDToTune> PIDs)
```

and using it would look something like this

```cpp

float linearError() { return 24 - linearTest->getPose().y; }

void linearRun(float kP, float kI, float kD, float slew) {
    lemlib::ControllerSettings testPID(kP, kI, kD, 3, 1, 100, 3, 500, slew);
    linearTest = std::make_shared<lemlib::Chassis>(drivetrain, testPID,
angularController, sensors);

    linearTest->setPose(0, 0, 0);
    linearTest->moveToPoint(0, 24, 5000);
}

// We are tuning the PID controller for linear movement, using a 24 inch forward
movement.
PIDTunerView pidTuner({{"Linear Drivetrain PID", linearRun, linearError}});
```

This design allows a lot of flexibility in what PID's we tune as it is very abstract asking only for what it needs to function.

To actually design the GUI we found that converting the design from an image to code is very time consuming so we looked for a way to automatically do this, as we figured, LVGL, a popular GUI program had to have at least one program for this purpose. So we started looking, and we found *SquareLine Studio* , which fit most of our requirements. So we designed the PID tuner inside SquareLine Studio, and we developed something like .

hi

Unlike the previous GUI's, developing this was significantly harder than just making it look good, however SquareLine Studio has a helpful function to export the UI to code, so all we had to do was implement the logic. Switching PID's was implemented by just changing the reference to the current `PIDToTune` and the text on the GUI. And the graph updating was done by first calling the run function,

and then creating a task that periodically calls the error function, resizes and updates the graph, which we'll talk about in more detail, and stops automatically when the error settles.

The task that creates the graph, is specifically a task, because we want it to run in the background and not block any interactions with the GUI, it should also stop if we interact with the other parts of the GUI, or press the panic button on the controller, so the task should look something like this:

```cpp
pros::Task graphError; // this is exposed to the rest of the class so it can be stopped from outside the task.

graphError = pros::Task([&]() {
    int settleTime = 0;
    while (true) {
        if (std::abs(error - this->currentPID->getError()) < 0.3) settleTime++;
        else settleTime = 0;

        float error = this->currentPID->getError();
        lv_graph_add_data_point(this->errorGraph, pros::millis(), error);
        graph_resize(this->errorGraph);

        if (settleTime >= 100) return;
        pros::delay(20);
    }
});
```

The resize function is the most interesting, what it does is it gets the bounds of the x and y axis and then adds a bit of padding. However after the size is set the graph has to be flagged as "dirty" which tells LVGL that the graph needs to be redrawn, this caused not a small amount of pain when coding this.

The final result, with a test function (shown in Listing 1), is shown in .

```cpp
external float startTime;
float testError() {
    float t = pros::millis() - startTime;
    return -0.2 * std::sin(4 * (t + 5.29)) * std::exp(3, 3.29 - t) + 5.48568; // https://www.desmos.com/calculator/234cudbevw
}

void testRun(float kP, float kI, float kD, float slew) { startTime = pros::millis(); }


PIDTunerView pidTuner({{"Test PID", testRun, testError}});
```

Listing 1: Our test code for the PID Tuner GUI

hi

The GUI also saves the values of each PID to a file on the SD card so we can easily resume tuning after a small (or large) break.

One last GUI before we start on the actual control algorithms. PROS has a default GUI that people use for debugging their code, It is a very helpful multi-purpose GUI that we would like, but with some improvements. PROS default GUI can be seen in .

hi

This has a couple of problems,

- there is a limited number of lines
- there are unused buttons on the bottom taking up space
- it looks ugly

So we decided to improve on them, we used the same color palette as Robodash similar to the rest of out GUI's. We again designed it SquareLine Studio, which is shown in . This time the implementation was pretty straightforward, we just have a vector of strings, one for each line, and when the user prints to a specific line it just changes the label for that line. The constructor has no arguments, and there is one method that lets you print to a line and another to clear it. The code was also made to be thread safe, by using mutexes to lock the variables when they are being accessed, so two threads don't change the same line at the same time.

hi

The code in Listing 2 give you .

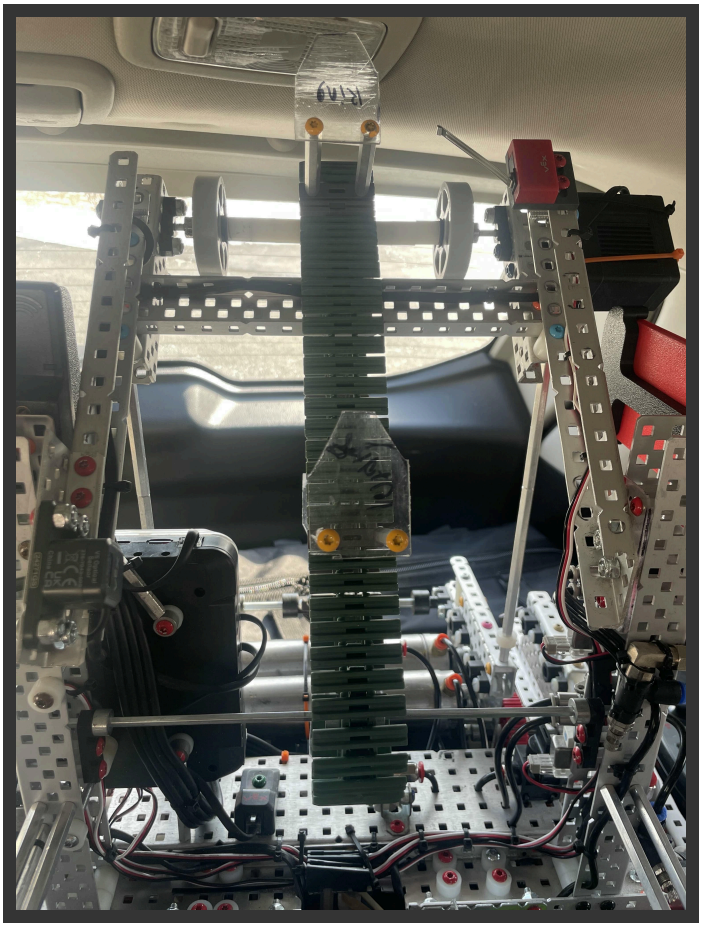
```cpp
DebugScreen lcd();

void initialize() {
  pros::Task test([&]() {
    while (true) {
      lcd.print(0, "hello %s", "world");
      lcd.print(1, "its been %f seconds", pros::millis() / 1000);
      lcd.print(4, "this would be on line 2");
      lcd.print(6, true, "this would be on line 6 with 3 empty lines before");
      lcd.print(3, "this is to test text wrapping and overriding lines, well i guess that was already tested as this is in a loop and run asynchonously from the rest of the program.");
    }
  })
}
```

Listing 2

hi

This GUI is very helpful when it comes to debugging random processes that don't have a designated GUI for them.

The usage of color sensors has been a prominent idea for quite a while now. When the games first came out we wanted to use the color sensors to differentiate between the different colored rings when running our intake. After completing most of the robot, we decided to attach it on as it would not require much modification. There are two segments to our color ejector, there's the actual color sensor itself which can track the color of the ring passing through the intake. Next there's a limit switch, near the end of the intake which acts as a trigger to determine when to eject the ring.



Top View of Color Sensor System on the Physical Robot

The image above is the color sensor equipped onto the intake. The gray bit on the left is the actual color sensor and the red piece on the top of the image is the limit switch.

We faced a couple of challenges when building this system, out first idea was to use a pneumatic to eject the rings. However after a rough calculation, we measured a 0.5 psi drop per piston toggle, and found that the psi drops to below usable pressures after only 10 ring sorts, so we discarded the idea and it never got build. Our first actual attempt lacked the limit switch and attempted to count the rotation of the motor powering the intake, however we found that this was very inconsistent and often failed. After adding the limit switch we still had to change the position of the switch to tune the ejection mechanism, and also change the position of the color sensor to make sure the ring was detected.

On the coding side of the mechanism, we decided to use a state machine. A state machine is a way to keep track of the robot's state, and apply different behaviors based on that. For example when the ring has been detected by the color sensor, we can change the robot's state to `toEject` if the color is not

the desired one, or to `toMogo` if it is. Before we started coding, we made a flowchart to help keep track of the different states, and how they change between each other which can be seen in Figure 32

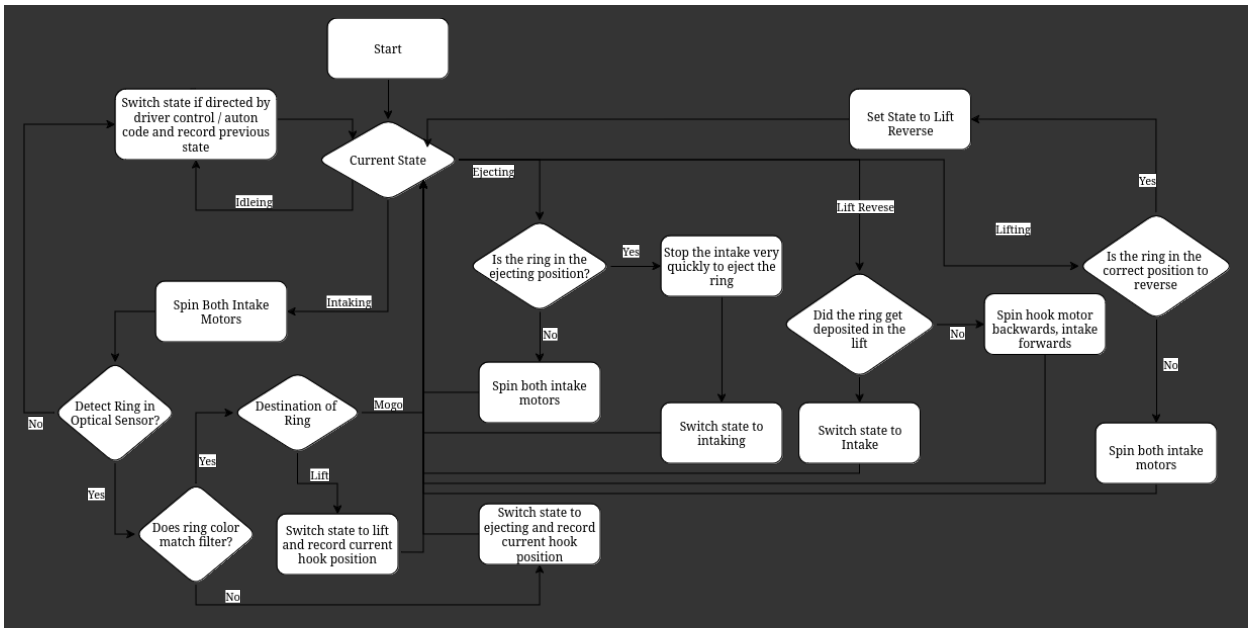


Figure 32: A flowchart describing the state machine for the intake

We decided to build a lot of extra features into the program, such as the wall stake mechanism which wasn't complete at that time.

Thanks to the flowchart, coding was relativity easy, with testing going very smoothly, and ejecting the ring relativity consistently, usually 4 times out of 5. When driving however this was reduced significantly, with the mechanism only succeeding 3 times in 10 trials. We tracked this down to the color sensor not sensing the ring, and failing to activate the color sort. We fixed this by moving the color sensor to the base of the intake, right where the flex wheels bring it up to the hooks.

Glossary

Odometry

An algorithim which determines the robots position based on the movement of 3 sensor wheels.

Omni Wheel

A shortened form of omni-directional wheel which is a wheel with rollers allowing it to be pushed side to side

PTO

Power take off device which takes the rotation of a motor and transfers it from one mechanism to another. Thye are often pneumatically powered, but can work with motors.