Date: March 4th, 2026 Event: Illinois V5RC HS Push Back State Championship

Instructions for team: Please fill out all information, printing clearly. This form should be included either behind the front cover, or in a clearly marked section in your Engineering Notebook. Teams may only submit **one** aspect of their design to be considered for this award at each event. Submission of multiple aspects will nullify the team's consideration for the award.

Full Team Number: **355Y**

Brief Description of the novel aspect of the team's design:
Our team developed a **completely custom made autonomous library** from scratch to replace the standard, built-in movement functions or the pre-coded functions from VexPros or LemLib that most teams use. This custom framework was designed to bypass the limitations of generic movement functions, providing us with granular control over the robot's low-level kinematics and high-level decision-making. Our system of a **three-wheel odometry tracking engine** that uses two parallel wheels and one perpendicular wheel, combined with a 6-axis Inertial Measurement Unit (IMU) allows the robot to calculate its exact global position (x, y, theta) on the field at all times. By using this data and custom library, we can use our **Pure Pursuit algorithms**, which allow the robot to follow smooth, curved paths rather than just driving in straight lines and making jerky turns.

Identify the page numbers and/or the section(s) where documentation of the development of this aspect can be found:
For an in-depth analysis of our custom motion framework and overall control strategy, please refer to our Programming Notebook, which can be found at the end half of this document. Page numbers: 144-227

Explain why your submission is unique from other approaches to the problem it solves or task it performs:
While many teams opt for a simple solution by configuring pre-built templates like LemLib or PROS, which often act as a black box where mathematical logic is hidden from the user, our team rejected this shortcut in favor of a **completely proprietary, full-stack motion library** engineered from first principles. By moving beyond mere parameter tuning, we replaced the generic, often jerky movement functions of standard libraries with a custom-coded framework that integrates 3-wheel odometry and a 6-axis IMU for absolute global positioning (x, y, theta). This vertical integration allowed us to bypass the inherent limitations and overhead of off-the-shelf wrappers, giving us the transparency to implement a high-frequency **Pure Pursuit path-following algorithm** that calculates smooth, non-linear trajectories in real-time. Our transition from a configured to a created system demonstrates **a deeper level of software engineering and control theory**, providing a level of precision and reliability that generic templates simply cannot match.

355Y YETIS ENGINEERING NOTEBOOK PUSH BACK

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Table Of Contents

Table of Contents

Category	Page	Entry
	1	Innovate Form
	2	Authors / Team Members
	3	Table of Contents
	4	Table of Contents
	5	Table of Contents
	6	Table of Contents
	7	Letter to Judges – State
	8	Team Introduction
	9	The Engineering Design Process
	10	Season Goals
	11	Season Goals (Signature Event Impact)
	12	5/11/25 – Game Analysis: Push Back
	13	Autonomous Win Point Explanation
	14	5/11/25 – Field Layout
	15	5/12/25 – Game Elements
	16	5/12/25 – Scoring Chart
	17	5/12/25 – Scoring Rules
	18	5/12/25 – Autonomous Strategy
	19	5/12/25 – Driver Control Strategy (Early Season)
	20	Driver Strategy Implementation
Design Iteration 1	21	Early Season Strategy
Design Iteration 1	22	6/18/25 – V1 Design Brief
Design Iteration 1	23	6/18/25 – V1 Drivetrain Criteria
Design Iteration 1	24	6/19/25 – Tank Drive Brainstorm

Design Iteration 1	25	6/19/25 – H-Drive Brainstorm
Design Iteration 1	26	6/19/25 – Wheel Layout Brainstorm
Design Iteration 1	27	6/19/25 – Drivetrain Layout
Design Iteration 1	28	6/20/25 – Drivetrain Gear Ratio
Design Iteration 1	29	6/20/25 – Drivetrain CAD
Design Iteration 1	30	8/2/25 – Parking Strategy
Design Iteration 1	31	8/2/25 – V1 Intake Criteria
Design Iteration 1	32	9/22/25 – Ruiguan Intake Brainstorm
Design Iteration 1	33	9/22/25 – S-Shaped Intake Brainstorm
Design Iteration 1	34	9/22/25 – C-Bot Brainstorm
Design Iteration 1	35	9/22/25 – Hopper Bot Brainstorm
Design Iteration 1	36	9/22/25 – Intake Design
Design Iteration 1	37	9/22/25 – Intake CAD
Design Iteration 1	38	9/22/25 – Matchloading Criteria
Design Iteration 1	39	9/30/25 – Tongue Mechanism Brainstorm
Design Iteration 1	40	Tongue Mechanism Criteria
Design Iteration 1	41	9/30/25 – Side Intake Brainstorm
Design Iteration 1	42	Cycle Speed Analysis
Design Iteration 1	43	Side Intake Conclusion
Design Iteration 1	44	Motor Mapping
Design Iteration 1	45	9/30/25 – De-Score Mechanism Criteria
Design Iteration 1	46	9/30/25 – Double Wing Mechanism Brainstorm
Design Iteration 1	47	10/10/25 – Matchload Mechanism Build
Design Iteration 1	48	10/13/25 – Intake Mechanism Build
Design Iteration 1	49	10/15/25 – Drivetrain Build
Design Iteration 1	50	Early Season Progress
Design Iteration 1	51	9/6/25 – Meeting #1 Drivetrain Structure
Design Iteration 1	52	9/13/25 – Meeting #2 Intake Structure

Design Iteration 1	53	Team Notes
Design Iteration 1	54	9/20/25 – Meeting #3 Drivetrain Screwjoints & Axles
Design Iteration 1	55	Wheel Testing
Design Iteration 1	56	9/27/25 – Meeting #4 Intake Stages
Design Iteration 1	57	10/4/25 – Meeting #5 Intake Rollers
Design Iteration 1	58	10/11/25 – Meeting #6 S-Shaped Curve
Design Iteration 1	59	10/18/25 – Meeting #7 Matchloading
Design Iteration 1	60	Matchloading Assembly Notes
Design Iteration 1	61	10/24/25 – Meeting #8 Pneumatics Installation
Design Iteration 1	62	10/25/25 – Meeting #9 Friction Testing
Design Iteration 1	63	Friction Optimization
Design Iteration 1	64	10/31/25 – Meeting #10 Autonomous Programming
Design Iteration 1	65	Tournament Recap #1
Design Iteration 1	66	Post-Lake Park Goals
Design Iteration 1	67	Film Analysis – Lake Park Matches
Design Iteration 1	68	Autonomous Match Analysis
Design Iteration 1	69	11/7/25 – Meeting #11 De-Score Mechanism
Design Iteration 1	70	De-Score Decision Matrix
Design Iteration 1	71	11/9/25 – Meeting #12 Matchloading Redesign
Design Iteration 1	72	Matchloading Improvement Notes
Design Iteration 1	73	Matchloading Optimization
Design Iteration 1	74	11/10/25 – Meeting #13 Programming
Design Iteration 1	75	Tournament Recap #2
Design Iteration 1	76	Competition Performance Analysis
Programming Iteration 1	77	Early Season Programming Explanation
Programming Iteration 1	78	Hardware Architecture
Programming Iteration 1	79	Sensor Polling Rate

Programming Iteration 1	80	Autonomous Sensor Usage
Programming Iteration 1	81	Odometry Technical Specifications
Programming Iteration 1	82	Odometry Mathematics (Part 2)
Programming Iteration 1	83	Arc Length Approximation
Programming Iteration 1	84	Average Orientation Method
Programming Iteration 1	85	Continuous Position Tracking
Programming Iteration 1	86	PID Control Reference
Programming Iteration 1	87	PID Control – Proportional Behavior
Programming Iteration 1	88	PID Control – Derivative Term
Programming Iteration 1	89	PID Final Tuning Values
Programming Iteration 1	90	PID Mathematical Summary
Programming Iteration 1	91	Sensor Integration – Optical System
Programming Iteration 1	92	Optical Sensor Operation
Programming Iteration 1	93	Teleoperated Control
Programming Iteration 1	94	Input Filtering (Cubic Control)
Programming Iteration 1	95	Annotated Code – main.cpp
Programming Iteration 1	96	PID Code Constants
Programming Iteration 1	97	Autonomous Code Section
Programming Iteration 1	98	Scoring Code Example
Programming Iteration 1	99	V1 Programming Conclusion & Summary
Design Iteration 2	100	Mid-Season Rebuild
Design Iteration 2	101	12/18/25 – V2 Design Brief

Design Iteration 2	102	12/20/25 – V2 Intake Criteria
Design Iteration 2	103	Multi-Block Possession Strategy
Design Iteration 2	104	Intake Performance Strengths
Design Iteration 2	105	Mobility Advantages
Design Iteration 2	106	Mobility Advantages (continued)
Design Iteration 2	107	12/22/25 – Tray Bot Brainstorm
Design Iteration 2	108	12/22/25 – Intake Selection & Design Plan
Design Iteration 2	109	12/23/25 – Meeting #1 Drivetrain Build
Design Iteration 2	110	12/24/25 – Meeting #2 Build Drivetrain Screwjoints
Design Iteration 2	113	12/25/25 – Meeting #3 Build Intake Structure
Design Iteration 2	115	12/26/25 – Meeting #4 Build Intake Structure p2
Design Iteration 2	117	12/27/25 – Meeting #5 Add Chain + Flexwheels
Design Iteration 2	121	12/28/25 – Meeting #6 Build Counter Roller
Design Iteration 2	123	12/29/25 – Meeting #7 Build Tongue Structure
Design Iteration 2	125	12/30/25 – Meeting #8 Build Wing Brainstorm & Structure
Design Iteration 2	127	Tournament Recap #3
Design Iteration 2	128	Goals After GL2 Competition
Design Iteration 2	129	Film Analysis of Great Lakes 2 Matches 1/05/26
Design Iteration 2	131	1/08/26 – Meeting #9 Redo Matchloader
Design Iteration 2	132	Tournament Recap #4
Design Iteration 2	133	Film Analysis of Mundelein Matches 1/11/26

Design Iteration 2	135	12/01/25 – Final Thoughts on Design Iteration 2
Design Iteration 3	137	1/12/26 – V3 Design Brief
Design Iteration 3	138	1/12/26 – V3 Intake Criteria
Design Iteration 3	139	1/12/26 – Brainstorm: Why Are We Rebuilding?
Design Iteration 3	140	1/15/26 – Re-Adjusting Drivetrain
Design Iteration 3	141	3/8/26 - Robot Evolution

Letter to Judges- State

Thank you so much for being here! We know that volunteering your time to judge is a huge commitment, and our team truly appreciates the expertise and encouragement you bring to this competition.

Making Your Review Easier

We know that diving into a technical notebook can be an overwhelming task. To respect your time and help you navigate our process, we've included a Table of Contents. You'll also find color-coded tabs at the top of the pages; these correspond to our design process chart below, allowing you to quickly jump to specific iterations or milestones.

What We're Proud of This Season: While we've learned a thousand small lessons this year, here are three major areas where we've pushed ourselves to grow:

1. **A Transparent Design Process:** We believe that the why is just as important as the what. To make our logic easy to follow, we've added a progress bar at the bottom of every page. This serves as a visual "map," showing exactly where we were in the design cycle during that specific entry.

2. **Precision Through CAD:** This season marked our transition into the world of digital twin modeling. By using CAD (Computer-Aided Design), we've learned to "fail fast" in a virtual space—saving us precious materials and time while helping us understand the intricate math behind our mechanisms before they ever hit the field.

3. **Building from the Ground Up In programming:** We decided to take the "scenic route" this year. Rather than relying on pre-built libraries, we've been developing our own Odometry and PID control systems and libraries from scratch. It's been a challenge to learn the underlying calculus and logic ourselves, but it has given us a much deeper connection to how our robot actually "thinks." **Check out our programming notebook for a detailed review of our innovative programming!**

What awards we are aiming for at state: Think Award, Innovate Award, Create Award, Build Award.

Thank you again for supporting student builders like us. We hope you enjoy reading through our season!

Sincerely,
Team 355Y

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Team Introduction

Ediz (10)- Programmer, Strategist

Mithil (10)- Designer

Naisha (11)- Mechanic, Strategist

Sathvik (10)- Driver, Mechanic, Designer

Shaurya (10)- Mechanic

Shreyas (8) - Mechanic

Sonit (11)- Programmer

Team Background and Chemistry- Each of us have multiple years of experience competing in the **VEX Robotics Competition**, and most of us have multiple years of competing at the **state level**. However, this is the first year we are competing together as a group. We realize that **team chemistry** is a vital aspect towards our success as a team. As we formed our team, we created **group conversations** on multiple networking platforms to make sure that we are **always in contact** with each other, so that everyone is aware of the progress that occurs as a team. As we are a team made up of students from **various schools**, we realize that if we can't openly communicate with our teammates to keep everyone in check, we will **not make any progress** throughout the season.

The Engineering Design Process

Overview- The engineering design process is a **systematic, iterative method** engineers use to solve problems and create **effective** solutions. It involves several steps, usually in the following order (though we will often loop back and repeat steps as needed). We will be using a **seven step** design process as noted:

Identify	Understand the need or challenge. Define what the problem is and the constraints involved.
Brainstorm	Generate a range of possible ideas or approaches. Creativity and collaboration are key here.
Select	Choose the most promising solution based on criteria such as cost, materials, time, and effectiveness. This involves creating a design matrix to find the best solution.
Designing/CAD	Show the process for incorporating the selected solution into a full design using CAD (Computer Aided Design) on Fusion, sketches, and/or prototypes.
Build	Build a working model or prototype of the chosen solution.
Program	Show the process as well as the final code used in the robot/solution. Explanations of techniques/concepts will be also under this step.
Test	See how well the prototype performs under real conditions. Measure effectiveness, durability, and other factors.

After iterating through this process, our robot should end up with a reliable mechanism to manipulate the game elements as necessary, which is achieved through thorough building and testing of prototypes until an effective solution is reached.

Season Goals

In order to define a clear vision for what we hope to achieve as a team throughout the 2025-2026 V5RC season, Push Back, we created the following set of goals:

1. Team Culture & Operations

- **Foster a Collaborative Environment:** Successfully integrate our new members by prioritizing mentorship and clear communication, ensuring every voice contributes to our robot's evolution.
- **Professionalism & Excellence:** Maintain a high standard of organization within our workspace and documentation to ensure consistency from the lab to the competition floor.
- **Sustainability through Partnership:** Secure a local sponsorship to build lasting community ties and support the team's technical needs.
- **Strategic Time Management:** Implement a consistent, balanced meeting schedule that respects team members' academic commitments while maximizing our build time.

2. Technical & Programming Milestones

- **Advanced Navigation (MCL):** Elevate our programming capabilities by implementing Monte Carlo Localization (MCL) to achieve high-precision robot positioning during Skills matches.
- **Custom Library:** This season, we challenged ourselves to push the boundaries of standard V5RC programming. By developing our own custom libraries for PID control and Odometry from the ground up, we've gained a deeper mathematical understanding of our robot's movements. This transition from pre-built solutions to custom-coded logic allows for significantly higher precision and a more reliable autonomous routine.
- **Autonomous Mastery:** Refine and "perfect" our autonomous routines through rigorous testing, aiming for nearly 100% reliability in a live competition setting.
- **Global Collaboration:** Engage actively in online alliances and scouting networks to exchange technical insights and learn from the diverse strategies of teams worldwide.

3. Competitive Performance Goals

- **Drive for Excellence:** Increase participation in local scrimmages to provide our drive team with the "seat time" necessary for peak performance under pressure.
- **Early Qualification:** Secure a spot at the State Championship during our very first event of the season through high-level performance and sportsmanship.
- **Signature Event Impact:** Attend a few Signature Events to challenge ourselves against world-class competition, aiming to rank **1st in Skills** and earn a direct qualification to the World Championship.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

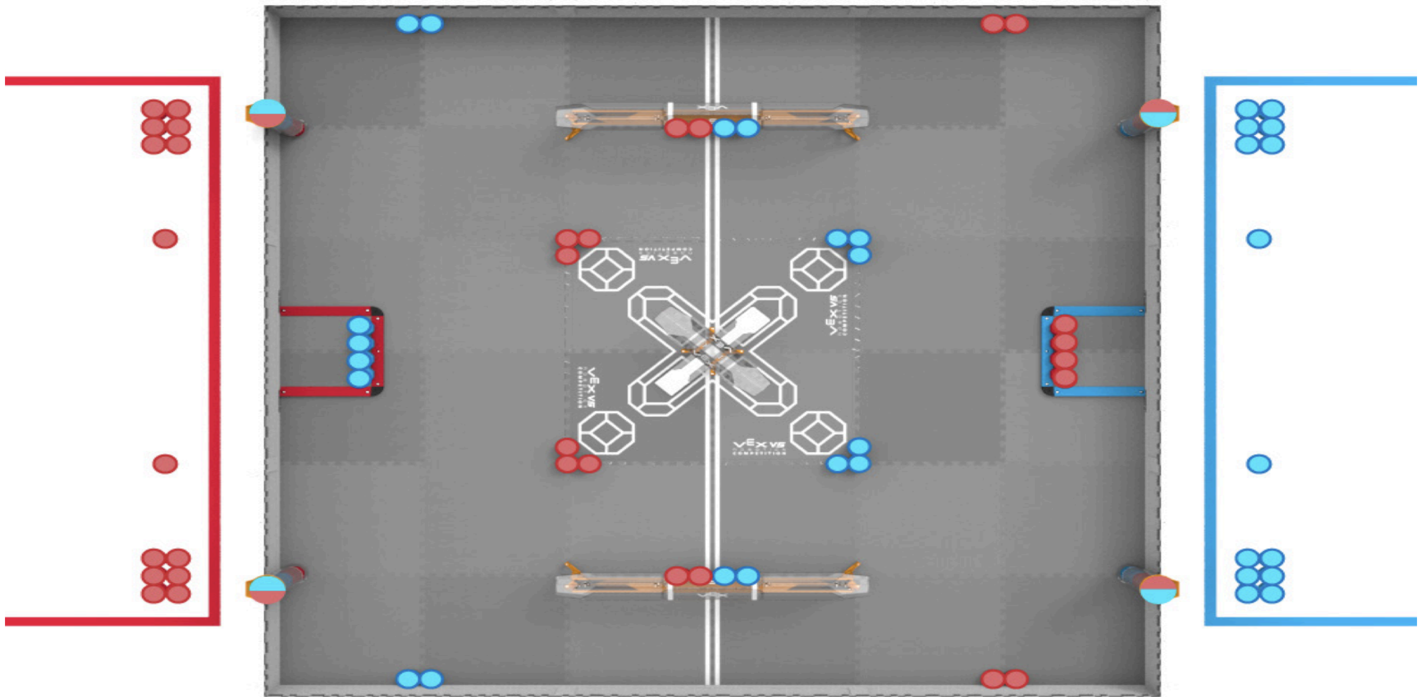
- **The "Triple Crown":** Aim to earn the Excellence, Tournament Champion, and Robot Skills awards (the Triple Crown) at at least one local event.

4. Season Benchmarks & World Championship

- **Awards:** Document our journey so thoroughly that we earn 10+ judged or performance awards by the end of the season.
- **State Championship:** Become tournament champions at State and win a notebook award.
- **The World Stage:** Qualify for the VEX World Championship, with the ultimate goal of winning a division award

5/11/25- Game Analysis: Push Back

Identify



Push Back Top View (VEX Game Manual 0.1)

Push Back is played on a 12' x 12' field, played with two alliances: one red alliance, and one blue alliance, made up of two individual teams each. The objective of the game is to score as many points within a two minute time period: 15 seconds in autonomous mode, and 1:45 minutes in driver control.

The main way of scoring points in this year's game is by placing octakaidecagon-shaped blocks in plastic scoring zones, and by parking your robot in the allocated parking zone.

Push Back resembles a rapid **back-and-forth** game, similar to VRC Change Up ('20-'21) and VRC Turning Point ('18-'19). In these games, the objective is to maintain possession of a certain scoring status in a scoring zone.

During the Autonomous Period, robots must remain on their side of the autonomous line splitting the field into half, while attempting to complete **four** different tasks to earn an Autonomous Win Point (AWP), and while scoring as many points as possible. Our autonomous strategy will attempt to **earn as many Autonomous Win Points as possible** as this helps increase our qualification rank in-tournament. The AWP can be earned by either alliance, independent of each alliance's end of autonomous result.
This year's Autonomous Win Point is earned by completing the specified tasks:

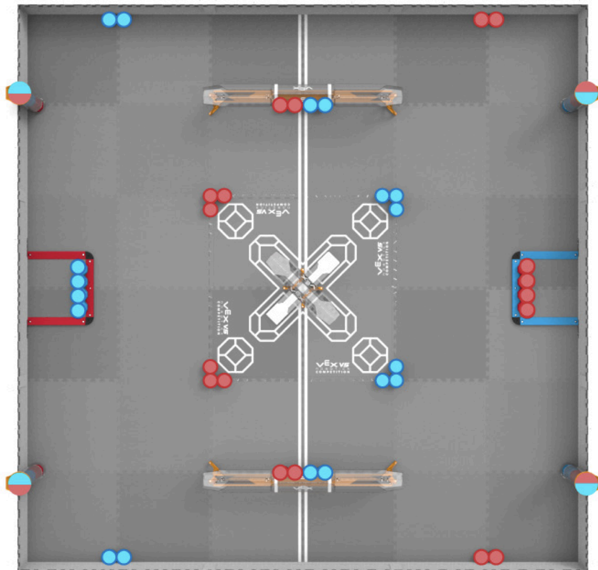
> At least seven (7) Blocks of the Alliance’s color are Scored.
>
> At least three (3) different Goals include at least one (1) Scored Block of the Alliance’s color.
>
> At least three (3) Blocks of the Alliance’s color have been removed from Loaders adjacent to the Alliance’s Alliance Station.
>
> Neither Robot is contacting the Park Zone barrier.

Additionally, the alliance that has scored the most points at the end of the autonomous period will be awarded with a **6 point bonus**.

At the end of the match, robots can park in a parking zone. As of 5/8/25, one robot within the parking zone awards **eight** points. If both robots are parked within the zone, the alliance is awarded **thirty** points.

5/11/25- Field Layout

Identify



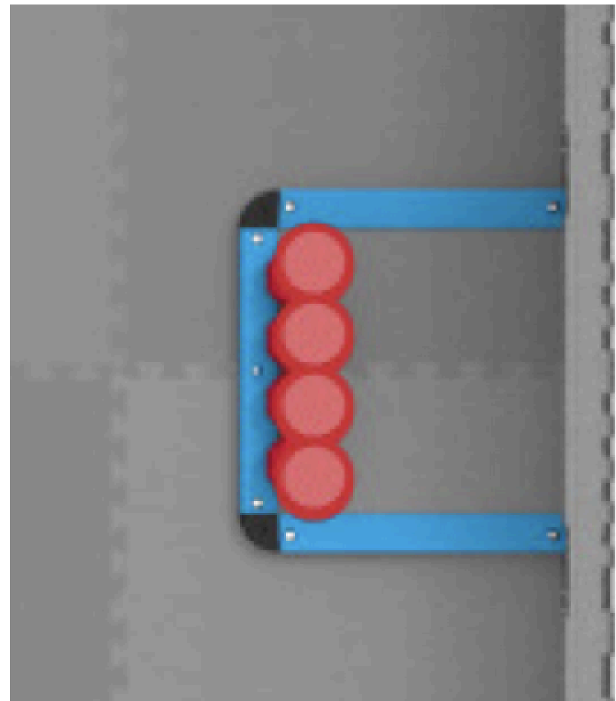
Push Back Top View (VEX Game Manual 0.1)

VRC Push Back is made of the following:

- 12' x 12' field, made up of 36 anti-static tiles
- 88 blocks, 44 per each alliance
  - 12 match loads
  - 12 within match-loading tubes found at the field parameter
  - 2 preloads
  - 18 found on the field in predetermined positions on the field
- 4 match loading sites
- 4 scoring zones
  - Two long goals
  - Two center goals, one upper and one lower
- Two parking zones, one per each alliance

At the **beginning** of the match, robots must start in predetermined locations. Robots must **not be contacting any blocks** other than the allocated preload given to them, and they must not be in contact with another robot. Additionally, Robots must be contacting the **barrier** that constitutes the **alliance parking zone** (pictured on the right).

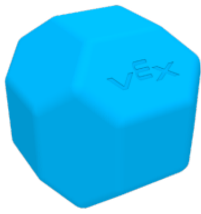
During the 15-second Autonomous Period, robots may **not** cross over the Autonomous line (2 parallel lines of white tape splitting the middle of the field). If any robot does cross over, the autonomous bonus goes to the **other alliance**, given the other alliance also didn't cross the autonomous line. After the Autonomous Period is over, and everything has been scored, the Driver Control Period begins. During Driver Control, teams may cross the autonomous line with **no violations**.



Parking Zone (VEX Game Manual 0.1)

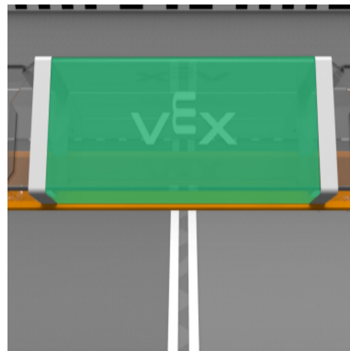
5/12/25 - Game Elements

Identify



Push Back Blue Block (VRC Game Manual 0.1)

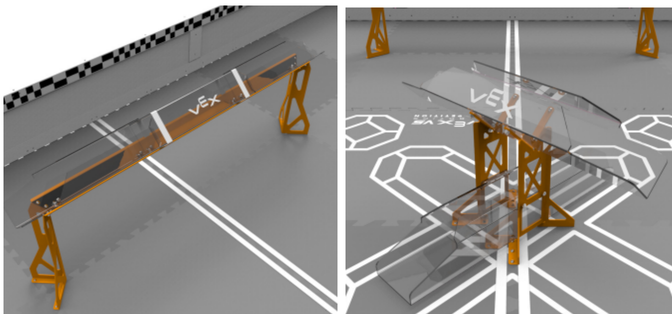
Block: A blue or red 18-sided hollow plastic polygonal object with flat faces and a weight of approximately 40 grams. Each cross-section measures approximately 3.25” (82mm) between pairs of opposing flat faces, and 3.85” (98mm) between pairs of opposing corners.



Push Back Control Zone (VRC Game Manual 0.1)

Control Zone - A defined section of a Goal that can be Controlled by an Alliance at the end of a Match

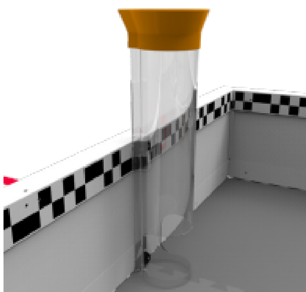
Goal - A Field Element that is constructed out of plastic and metal components into which Blocks can be Scored. Each Long Goal has a completely enclosed center section between two open sections. Each Goal includes a defined Control Zone.



Push Back Long and Center Goals (VRC Game Manual 0.1)

Long Goal - Each Long Goal is 48.8” (1239 mm) in length, with a 13.33” (339mm) enclosed center section. Each Long Goal can hold up to 15 Scored Blocks. The Control Zone consists of a zone highlighted by white tape.

Center Goal, Upper and Lower – Each Center Goal is 22.6” (574mm) in length. Each Center Goal can hold up to seven (7) Scored Blocks. The entire goal would be considered the Control Zone.



Push Back Loader (VRC Game Manual 0.1)

Loader - One of four 21.34” (542mm) tall plastic and rubber structures each attached to the Field Perimeter. Robots may remove Blocks from Loaders during a Match, and Drive Team Members may add Match Load Blocks to Loaders during the Match. Each Loader can hold up to six (6) Blocks.

Push Back Park Zone (VRC Game Manual 0.1)

Park Zone - A Field Element that marks a location where Blocks begin a Match and Robots can be Parked at the end of the Match. Park Zones are made of red or blue plastic extrusions and black plastic connectors. Each Park Zone is 18.87“ (479mm) wide x 16.86” (428mm) deep

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

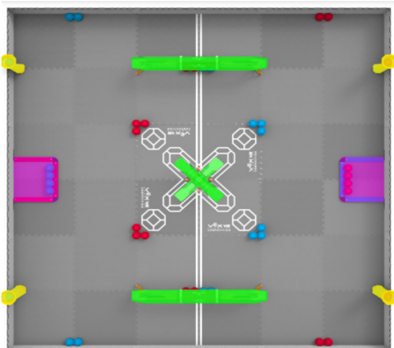
5/12/25 - Scoring Chart

Identify

	Point Value	Maximum Value
Autonomous Bonus	6 points	6 points
Each Block Scored	3 points	134 points
Controlled Zone in a Long Goal	10 points	20 points
Controlled Upper Goal	8 points	8 points
Controlled Lower Goal	6 points	6 points
1 Parked Robot	8 points	8 points
2 Parked Robots	30 points	30 points
	Total Possible Score:	212 points

After analyzing the number of ways to score, we concluded that prioritizing the control zones would be most beneficial to help us maintain momentum in each match by awarding us 10 points. We also concluded that utilizing the end game bonus's are beneficial for our matchplay strategy, potentially earning **thirty** points for our alliance.

At first thought, the initial strategy would be to quickly collect and score as many blocks as possible, defend your scoring zones, and use the last 5 seconds to park both robots in the parking zones. **However, this idea is subject to numerous changes as the Meta continues to change.** Our team will constantly watch game film on experienced teams to mold our strategy and dominate games.



Hot Zones in Push Back (VRC Game Manual 0.1)

Presumed **hot zones** are highlighted in the diagram shown at the left. We believe that as the season goes on, match loading will be an effective part of game strategy, causing opponents to contest an alliance's match loading zone **(highlighted in yellow)**. Additionally, scoring zones **(highlighted in green)** will be contested throughout the match, as alliances work to hold possession of the control zone. Endgame parking zones **(highlighted in purple)** will also be prioritized during the endgame, as teams can earn up to thirty points.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

5/12/25 - Scoring Rules

Identify

SC1: "All Scoring statuses are evaluated after the Match ends. Scores are calculated 5 seconds after the Match ends, or once all Scoring Objects, Field Elements, and Robots on the Field come to rest, whichever comes first." - The five second grace period is intended to be for allowing "benefit of the doubt" for scoring which takes place last second.

For example:

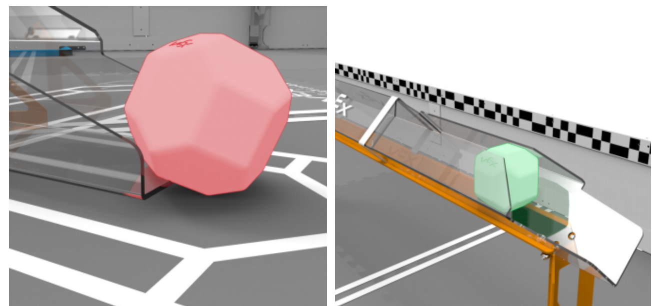
- A Robot which has Parked in a Park Zone but slowly droops down and is in contact with the top of the Field Perimeter at five (5) seconds would not be considered Parked.
- A Block which slowly falls out of a Goal at five (5) seconds would not be considered Scored.

(Game Manual V0.1, Page 22)

SC2: Blocks are considered scored if it meets the following criteria:

- The Block is in contact with the inside surface(s) of a Goal.
- The Block is not in contact with a Robot of the same color as that Block.
- The Block is not in contact with the Floor.

(Game Manual V0.1, Page 22)

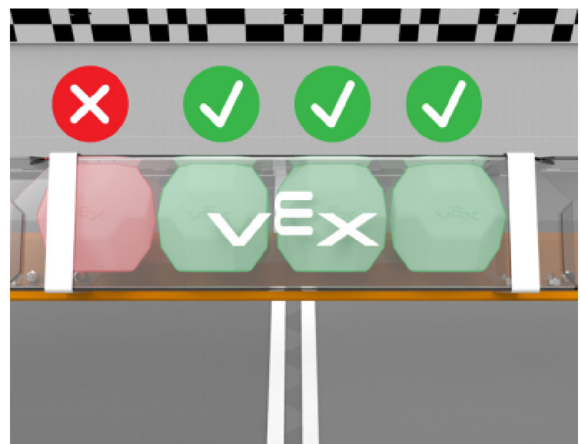


Not-Scored Block (Left), Scored Block (Right) (VRC Game Manual 0.1)

SC3: A Control Zone is considered Controlled by an Alliance if a majority of the Blocks Scored in that Control Zone are the same color as the Alliance.

- For Long Goals, a Scored Block is considered Scored in the Control Zone if it is entirely contained within that Control Zone.
- A Block must be considered Scored in a Goal (see) to also be considered Scored in a Control Zone.

(Game Manual V0.1, Page 23)



Example of Scored Blocks in a Control Zone (VRC Game Manual 0.1)

5/12/25 - Autonomous

Identify

SC5: Scoring of the Autonomous Bonus is evaluated immediately after the Autonomous Period ends (i.e., once all Blocks, Field Elements, and Robots on the Field come to rest).

- Points for Parked Robots are not included in the calculation of an Alliance's score for the purposes of determining the Autonomous Bonus.
- If the Autonomous Period ends in a tie, including a zero-to-zero tie, each Alliance will receive an Autonomous Bonus of five (5) points.
- Any Violations, Major or Minor, committed during the Autonomous Period will result in the Autonomous Bonus being automatically awarded to the opposing Alliance.
- If both Alliances commit Violations during the Autonomous Period that would have affected the outcome of the Autonomous Bonus, then no Autonomous Bonus will be awarded.

(Game Manual V0.1, Page 25)

SC6: An Autonomous Win Point is awarded to any Alliance that ends the Autonomous Period with all of the following tasks completed, and that has committed no Violations during the Autonomous Period:

- At least seven (7) Blocks of the Alliance's color are Scored.
- At least three (3) different Goals include at least one (1) Scored Block of the Alliance's color.
- At least three (3) Blocks of the Alliance's color have been removed from Loaders adjacent to the Alliance's Alliance Station.
- Neither Robot is contacting the Park Zone barrier.

Potential Autonomous Strategy 1:

- Low Goal
  - Pick up the balls from the low goal corner on the alliance side and score them in the low goal
  - Unload the entire matchloader and try to score it into the high goal

Potential Autonomous Strategy 2:

- Middle Goal
  - Pick up the balls from the middle goal corner on the alliance side and score them in the middle goal
  - Unload the entire matchloader and try to score it into the high goal

Potential Autonomous Strategy 3:

- High Goal
  - Go directly and unload the entire matchloader into a high goal
  - Place the robot in a strategic position for the driver control portion of the match, this would most likely be readying the bot to score another full high goal.

5/12/25 - Driver Control Strategy- Early Season

Identify

Potential Offensive Strategy 1:

- Prioritize the Autonomous Bonus/AWP
  - Descoring blocks is very easy in this game, therefore you are never guaranteed points in driver control until the game timer is complete. This stresses the importance of taking control of the autonomous bonus, as those are guaranteed points even if all of your blocks are descored.
- Collecting match loads during the match, and then scoring them in the scoring zones
  - This type of gameplay resembles late-season **VRC Over Under strategies**, where teams would utilize match loads throughout the driver-controlled period. In High Stakes, gameplay could include loading one of your **fourteen** match loads into the match loader, intaking it, and scoring it in a zone, making Push Back into a **back-and-forth** game.
- Collecting blocks throughout the match, and scoring everything late match
  - This strategy would utilize a **compartment** to store alliance blocks

Potential Offensive Strategy 2:

- Get Into a Scoring Position
  - Instead of trying to win AWP, **collect** as many blocks as possible and get in a position to score them from driver control, leading to more points in the long run. This would likely consist of emptying the matchloader during autonomous and (maybe) scoring them. Another possible way to do this would be picking up the four balls in the corners in the middle and low goals and trying to score those.
- Score in high goals
  - Focus mainly on the high goals instead of low and middle goals, use the de-scoremech to try and get these blocks into the control zone, a lot of potential points sit in high goal control zones.
- Score in middle/low goals
  - Later in the game, focus completely on controlling the low and middle goals. Even more specifically, focus on the low goals, since they are harder to de-score and aren’t noticed as much, especially in the endgame, when everyone is rushing to score/de-score high goals.

Potential Defensive Strategy 1:

- Camping one side of a scoring zone that you maintain control of
  - This strategy would help by preventing any of your blocks from falling out of the side of the goal when your opponent attempts to score one of their blocks in the zone.

Potential Defensive Strategy 2:

- Collecting blocks that belong to the opponent’s side and saving them in a compartment
  - To implement this strategy, we would need to create a **backpack** that is accessible using a **redirect** method.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Potential Defensive Strategy 3:

- Dedicating one of our alliance members to chasing after an opponent's bot
    - This involves trying to disturb their driving by setting up **screens** that block any **access** to their blocks.

Potential Endgame Strategy 1:

- Descoring by ramming into the scoring zones
    - **With moderate speed and force, ramming into the scoring zones clears out every block.** We could use this to our advantage by ramming into any zone that the opposing alliance holds possession of around the end of the match.

Potential Endgame Strategy 2:

- Dumping any blocks collected during the match that belong to the opponents
    - This would also disturb the opponents from utilizing the parking zones, **without getting disqualified**. Opponents would be forced to clear the zone, taking time off of their endgame.

Our team will be utilizing these strategies to maximize our alliance's score while hindering the progress of our opponents.

Early Season

6/18/25 - v1 Design Brief

Identify

Game Constraints:

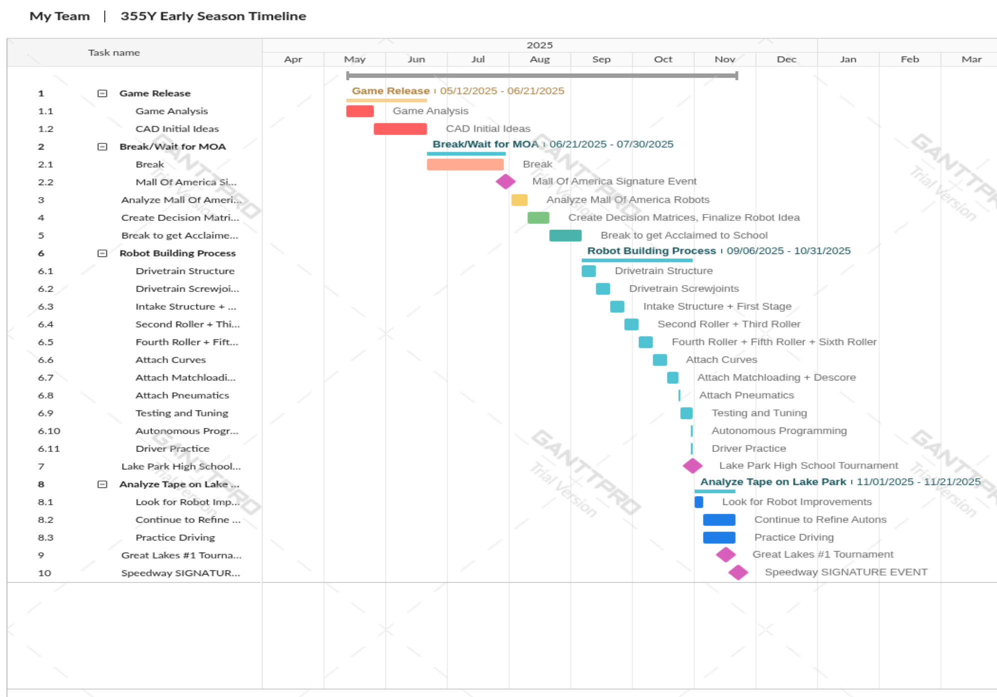
- Robot must be able to fit in an 18” by 18” by 18” cube at the start of the match
- Total combination of motors must not exceed 88 watts
- Must be built only from VEX-approved parts
- Custom plastic is limited to twelve pieces, each piece fitting within a 4” x 8” area

Personal Goals and Strategy:

- Score more than 7 blocks in 3 scoring zones during autonomous period
- Strategize with alliance team to get the autonomous bonus and win point
- Work with alliance team to score as many points as possible
- Control one long goal and one center goal during matchplay
- Be able to park both robots in the parking zone at the endgame without getting your scoring zone descored
- Descore opponent scoring zones before end of match

Robot Subsystems:

- Drivetrain
- Intake mechanism to load blocks into the robot
- Scoring mechanism to score blocks into goals
- Mechanism to punch out blocks from a scoring zone
- Mechanism to smoothly collect match loads from match loading zones



6/18/25 - v1 Drivetrain Criteria

Identify

Problem Statement: Design a drivetrain to move a full weight robot around the field to efficiently interact with Blocks, Scoring Zones, Match Loading Sites, and Parking Zones. The robot also needs to possess multiple blocks, so the drivetrain needs to be able to move effectively with an extra few pounds. The Drivetrain is the foundation of all robot movement, so having an efficient and robust solution is important.

Solution Constraints:

- Must start within an 18” by 18” square at the start of the match
- 22” footprint once the match starts in both dimensions
- Has to work effectively on a combined 66W or less

Solution Goals:

The drivetrain must traverse the entire 12-foot field in under 2 seconds.

- A high-speed drivetrain is essential for rapid acquisition of Blocks and other game elements, directly increasing scoring efficiency. Enhanced speed also optimizes autonomous routines, reducing transit time and allowing more time for additional tasks.

Maximum power consumption per drivetrain side must not exceed 0.2W.

- Low wattage values indicate minimized drivetrain friction and improved energy efficiency, allowing for extended motor operation without overheating and reducing battery drain during matches.

The drivetrain must generate sufficient torque to transport a mobile goal (3.5 lb).

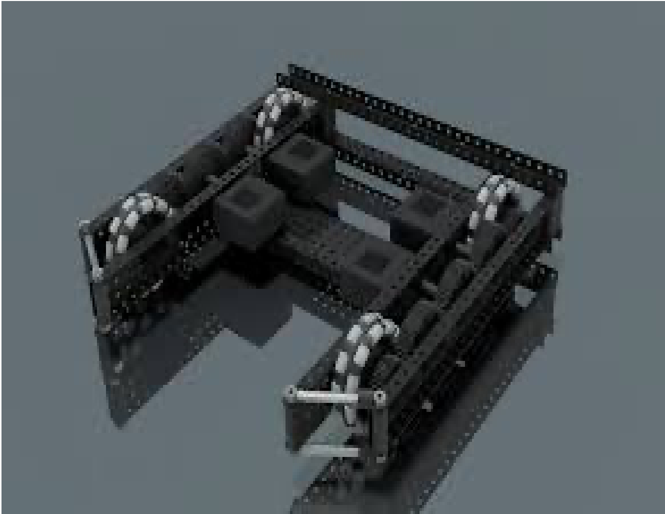
- The system must support both the robot and a mobile goal, maintaining the ability to cross the 12-foot field in under 3 seconds while carrying the additional load. This ensures reliable possession and movement of mobile goals throughout gameplay, supporting strategic positioning and corner play.

The drivetrain must provide sufficient force and agility to resist defensive interactions.

- The system should withstand or counteract opposing robot forces, enabling either a pushing stalemate or the ability to push adversaries. Lateral maneuverability must be maintained to escape defensive pinning, ensuring the robot does not become immobilized and can recover position efficiently.

6/19/25 - Tank Drive Brainstorm

Brainstorm



(Tank Drive Computer-Assisted Design, BLRS/BLRS2)

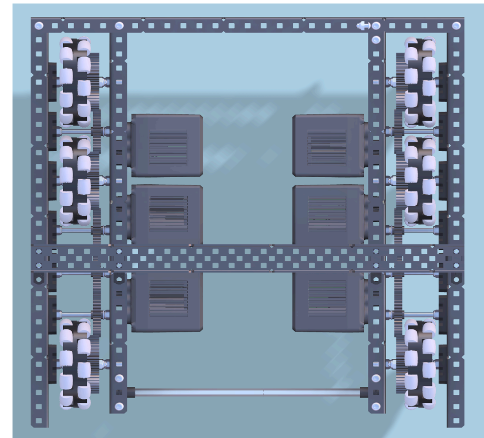
The Tank Drive utilizes two parallel sets of wheels, all oriented in the same direction, and can be powered by anywhere from 2 to 8 motors. The parallel wheel configuration greatly simplifies the process of mechanically linking all wheels via gearing, offering flexibility in the use of idler gears between the driver and driven gears for efficient power transmission.

The internal layout of the Tank Drive is highly optimized for space efficiency. Motors can be mounted in parallel along the inner channels on each side of the chassis, resulting in a long, unobstructed central area. This arrangement not only maximizes available internal space but also maintains a low vertical profile, allowing for a compact and streamlined drivetrain.

The Tank Drive design provides extensive options for structural reinforcement. Cross bracing can be implemented between the wheels (either above or below the chassis), at the front and rear of the chassis, or between the inner channels where the motors are mounted. This structural versatility, combined with the drive's thin profile, creates ample mounting space for additional subsystems and mechanisms, facilitating modular robot design and integration.

Controlling the drive is straightforward because there are only two sets of wheels, making it easy to use different joystick setups. You can use Tank Drive, where each joystick controls one side; Split Arcade, where one joystick turns and the other moves forward and backward; or **Single Arcade**, where one joystick handles all movement, leaving the other hand free for other controls.

The simple wheel layout also makes programming easy, since the code closely matches the way you control the robot. Even with this simplicity, the drive can still be finely tuned for precise movement and high performance.



(Tank Drive 355Y)

Although we acknowledge other types of drivetrains, our team has decided that we will **build a tank drive** for our early season iteration due to our familiarity with the drivetrain type and us having limited time to build during the summer. In future push back iterations or seasons, we will consider using other types of drivetrains.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

6/19/25 - “H” Drive Brainstorm

Brainstorm

The H drive is a variation of the Tank drive with one addition: a perpendicular wheel in the center of the drivetrain. The center wheel and motor used to power it are used to strafe the robot sideways without rotation. To keep the functionality of the tank drive, the wheel has to be an omnidirectional wheel so when it isn’t active, the robot can move without resistance. All other wheels on the drive have to be Omni wheels as well, for the same reasons. This gives the H Drive extra mobility, but it also makes it susceptible to being pushed easily from the side.

Due to the middle wheel being powered by an independent motor, this takes away some potential functionality from the rest of the robot. While strafing, this makes the center wheel a one-motor drivetrain, which places a lot of strain on the center motor, which could make it burn out faster, leading to limited strafing movement during the match.

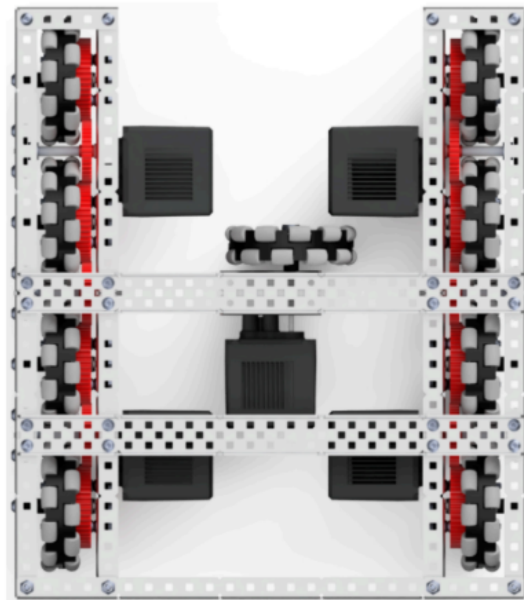


Figure 1: 5-Motor H Drivetrain (by 8995B)

....

Figure 2: 5-Motor H Drivetrain (by 355Y)

To strafe, the center wheel has to be at the point of center of mass. Otherwise, the wheel will just be rotating the robot while moving sideways, which can misalign the robot while autonomous, and be a nuisance during the match. The wheel also has to have a lot of torque to push the entire robot, so with only 1 motor it will have to be slow compared to the rest of the drive. Such torque can be achieved through gearing a motor down to the wheel, or by using a torque-geared motor cartridge like the 100 RPM.

For programming, this extra movement option can be beneficial to align the robot and make very precise corrections to its position, due to the slow center wheel. This extra precision will take some extra time though: a trade-off of sorts .

6/19/25 - Wheel Layout Brainstorm

Brainstorm

Omni Wheels: Omni wheels have small rollers around their edge, set perpendicular to the main wheel, which lets them move both forward/backward and side-to-side with very little friction. This makes it easy for a robot to turn and slide in any direction, but also means the robot can be pushed sideways easily. Omni wheels are often used for tracking wheels in odometry because they don't interfere with movement when mounted sideways, allowing for accurate position tracking.

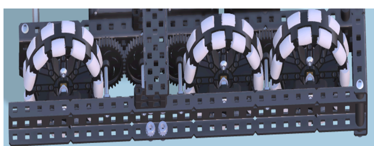
(Anti-static and Legacy Omni Wheels, VEX)

(Anti-Static Traction Wheels, VEX)

Traction Wheels: Traction wheels have a rubber layer that gives them strong grip and helps prevent the robot from sliding sideways, especially during defense or when another robot is pushing. This high traction makes the robot harder to push but can reduce turning speed compared to omni wheels. Traction wheels are often used in the middle of a drivetrain with omni wheels on the outside, creating a turning center that reduces drift when turning and helps resist unwanted sideways movement. Flex wheels can also be used for similar purposes.

Wheel Layouts Being Considered:

4 Wheel, all Omni:
The wheels are positioned at each corner of the robot, sinking the robot heavily into the tiles, and having a lot of idle gears in between wheels. This creates a lot of opportunity for bracing space, but also wasted space.



(Wheel Layout 355Y)

6 Wheel, all Omni:
4 of the wheels are at corners, with the other 2 between 2 corners, which creates 2 parallel lines of wheels. Less tile sink for more bottom brace clearance, while keeping brace options open. Lots of gearing options.

6 Wheel, 4x Omni, 2x Traction:
Similar to the all-omni variant, but all the wheels are in a row, and the middle traction wheel has to be in the middle. The center traction wheel will become the turning axis, so the middle is needed.

8 Wheel, all Omni:
4 wheels on 2 parallel robot sides, positioned in a line with usually even spacing. Has less bracing/spacing customization, but the robot can smoothly move around the field even with close bottom bracing.

8 Wheel, 4-6x Omni, 2-4x Traction:
Similar to the all omni variant, but the traction wheels can be either 1 per side, or 2 per side. 4 total tractions: they are in-between omni wheels, and the turning center is between them. With 2 total, they are asymmetrical, and the turning center is displaced in the direction where the tractions are placed.

6/19/25 - Drivetrain Layout

Select

Decision Chart for Drivetrain Layout:

	Weight	Maneuvering	Resistance	Traction	Total
4 wheel all omni	5	1	1	1	8
6 wheel all omni	3	3	1	1	8
6 wheel with 2 traction	4	4	4	4	**16**
8 wheel with 4 traction	1	2	5	5	13
8 wheel all omni	2	5	1	1	9

How we gave our point values: Each layout in consideration was ranked based on the four aforementioned categories, with 5 being the highest and 1 being the lowest. Our selection was based on which drivetrain had the highest point value.

Decision: Our final decision for Drivetrain Layout is a 6 wheel drivetrain with 2 traction wheels in the middle. We would also have an idler gear in place of the wheels that we are choosing not to utilize on a standard 8 wheel drivetrain. This allows us to insert a full-length crossbrace directly over the idler gear, allowing us to mount our intake in a favorable position around the front of the robot.

Drawbacks: Although the drivetrain is ideal for weight-saving purposes, it lacks the maneuvering ability similar to a 8 wheel omnidirectional drivetrain, or even a **10 wheel** all omni drivetrain.

Important: The presence of two traction helps to not be pushed around during a match. This is vital as from early strategy analysis, robots will face heavy contention when approaching the scoring zones to load blocks.

6/20/25 - Drivetrain Ratio

Select

Decision Chart for Drivetrain Ratio:
As a team we had identified early that there is a need for speed in these games, especially for driver skills. Adding in our weight-saving goals, the two ratios we are considering for our drivetrain are

- 450RPM (36 driving : 48 driven)
- 600RPM (36 driving : 36 driven)

	Speed	Acceleration	Torque	Space	Strategy	Totals
450RPM, 2.75 inch wheels	1	**3**	**3**	**3**	1	**11**
450RPM, 3.25 inch wheels	2	2	2	2	**3**	**11**
600RPM, 2.75 inch wheels	**3**	1	1	1	2	8

How we gave our point values: Each layout in consideration was ranked based on the four aforementioned categories, with 5 being the highest and 1 being the lowest. Our selection was based on which drivetrain had the highest point value.

The decision was a tie between the two 450RPM drivetrains. The decision was between whether we wanted a drivetrain with more speed or a drivetrain with more acceleration. Based on early impressions of the game, we decided to go with the **450RPM using 3.25 inch wheels**. We felt that we needed more speed for this game as this helps us with our skills runs. Part of our strategy involves traversing the whole field in such quick times, requiring more speed than acceleration.

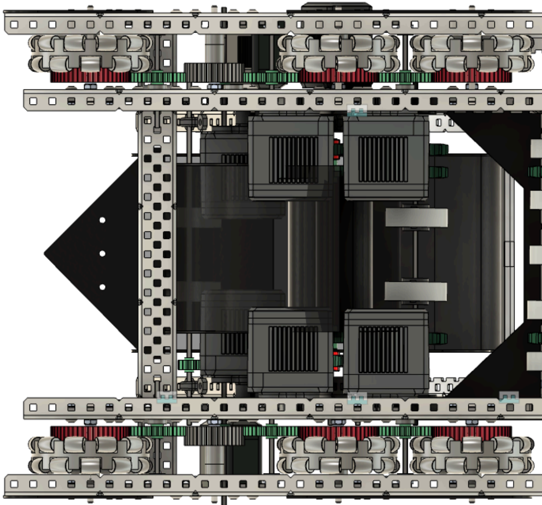
Drawbacks: The spacing in between wheels will require us to use out of the ordinary spacers (0.25OD spacers), as the clearance between the axle joints and wheels isn't a comfortable space.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

6/20/25 - Drivetrain Computer-Assisted Design

Designing/CAD

Designing our robot on a computer-assisted design software is essential to our team's progress as it makes the construction steps clear and flawless. It allows us to properly plan out our crossbrace position on the drivetrain, where our motors fall, and where we can build our manipulator structures off of without having to go through a trial and error process during our limited building time.



Bottom View of Drivetrain Design (355Y)

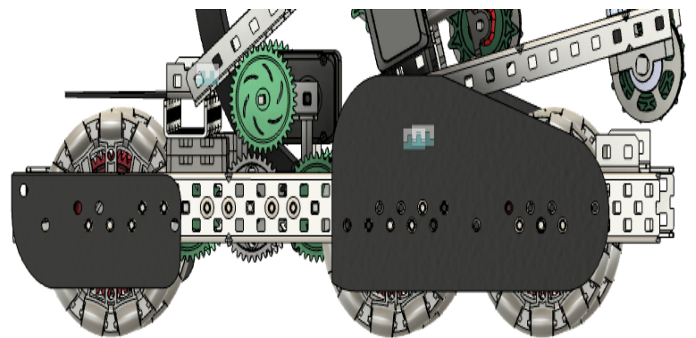
Our drivetrain's ideal layout supports 8 wheels, however we decided that 8 wheels wasn't necessary and also utilizing only 6 wheels allowed us to find a convenient spot for our drivetrain's cross brace to build off of.

Our robot dimension is 29 holes long by 25 holes wide, equating to a 14.5" * 12.5" drivetrain. The small width aligns with the width of the ball, and was strategically designed to reduce jamming within the robot's mechanisms as only one ball could flow at a time.

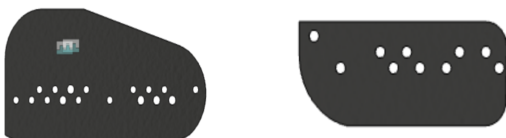
The outer drivetrain c-channels are 29 holes long, while the inner drivetrain c channels are 27 holes long.

We are implementing a stacked motor setup so that we can utilize more space on the bottom of our robot for important aspects like odometry sensors and a potential spot to place our battery.

We are utilizing 6 motors on our drivetrain, with a 450 RPM ratio, on 3.25 inch wheels, giving us the necessary speed needed to play a fast-paced game.



Side View of Drivetrain CAD (355Y)



Our drivetrain will include two sets of sleds on the front and back, allowing us to easily cross the parking zone's border with its curved design. This will play heavily into our strategy, as discussed in the next section of our design notebook.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

8/2/25 - Parking

Identify

Overview of Parking: As shown below, the scoring guidelines in the Vex PushBack Game Manual introduce another scoring mechanic: Parking. This is in the same category as the hang/climb from last year's High Stakes, an endgame robot-based scoring opportunity. Parking mainly consists of a robot entering its alliance-colored parking zone. It has to do this without touching the field outside of the parking zone after the end of the game.

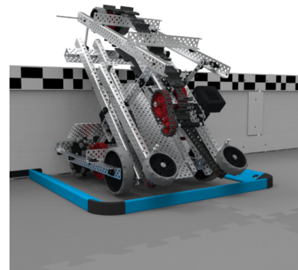
SC4: A Robot is considered Parked if it meets all of the following criteria:

- The Robot is not contacting the Floor outside of its Alliance-colored Park Zone.
- The Robot is not contacting any Field Elements other than the inside face of the Field Perimeter, the floor inside of its Alliance-colored Park Zone, and/or the plastic extrusions and connectors that are part of the Park Zone. Contact with these allowed elements is not required.
- The Robot is at least partially within the vertical projection of its Alliance-colored Park Zone.

Significant Q&As:

• Q&A 2672 - Contact with Blocks doesn't affect a Parked status

(Game Manual V0.1, Page 27)



This Robot is at least partially within the vertical projection of their Alliance-colored Park Zone, and would be considered as Parked.

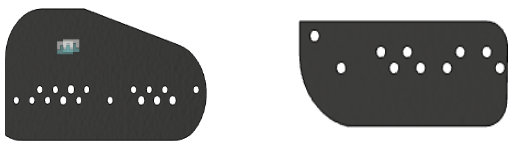
Both of these Robots would be considered as Parked, as they satisfy all the criteria listed above.

For example:

- A Robot which has Parked in a Park Zone but slowly droops down and is in contact with the top of the Field Perimeter at five (5) seconds would not be considered Parked.

(Game Manual V0.1, Page 27)

Our Solution:



For our solution, we took inspiration from many reveals such as Snacky Cakes (16610A), where we noticed they had sleds made of plastic attached to the drivetrain. This eased additional height for the bot and lifted it up in such a way that it didn't get stopped abruptly or slowed down too much.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

8/2/25 - v1 Intake Criteria

Identify

Problem Statement: Design an intake to interact with blocks by collecting and storing them within the mechanism. Intake should at least be able to transport a block from ground to goal to enable scoring opportunities.

Solution Constraints:

- Must work efficiently on a 22W maximum
- Must not extend past the 18" size constraint on all three dimensions

Solution Goals:

Maximum power consumption for intake must not exceed 0.5W.

- Given that most of our intake will be linked by chain, the motors will most likely be working harder to power the entire mechanism. However, the amount of power needed to work the mechanism must remain low so that the intake works efficiently and doesn't overheat the motor.

Intake must be able to score on long goals and both center goals (medium + low heights)

- Being able to utilize all types of scoring methods could prove handy in a close match where long goals are filled to the maximum. Being able to score on both of the center goals could give both our team and our alliance an advantage over others.

Intake should be able to possess around 8-9 blocks within its mechanism

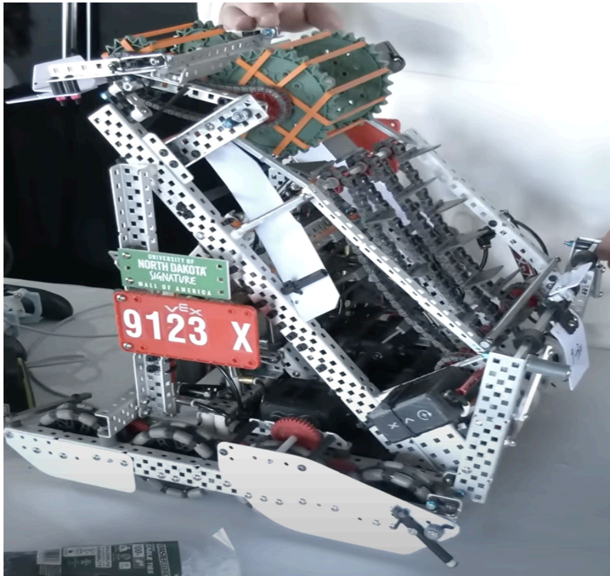
- Being able to possess multiple blocks at once will make it effortless to create splash plays of point gain, rather than possessing around 5 blocks at a time and having to go back to retrieve 5 more blocks in another rotation.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

9/22/25 - 9123C + X "Ruiguan Intake" Brainstorm

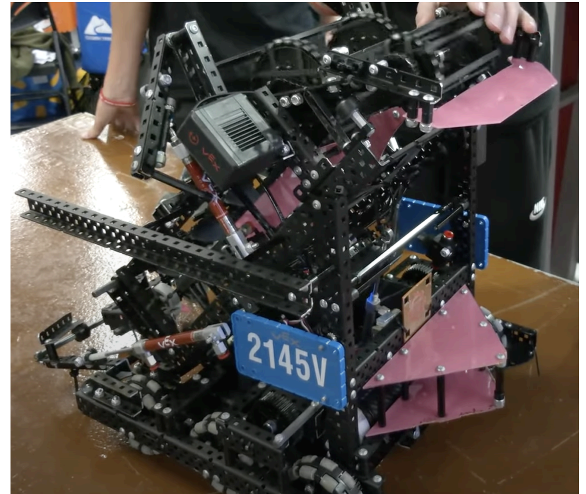
Brainstorm

<u>Tournament Review:</u> Mall of America Signature Event **Champions** + **Tournament Finalists** 9123C and 9123X showed up with intriguing "front to back" intake designs.



Ruiguan Intake (Pits and Parts Interview with 9123X)

A **"front to back"** intake design is one where a ball travels from the front and bottom of the robot and exits the mechanism and is eventually scored through the top and back of the robot.



2145V Intake (Pits and Parts Interview with 2145V)

<u>Strengths:</u> The intake is able to score blocks at a fast pace. It is also able to carry many blocks at the same time through an inbuilt "storage", due to the length of the system. In addition, the top aligner can be rammed into the long goals to descore game blocks.

<u>Weaknesses:</u> Easily prone to jamming if not made correctly. Not easily able to score on the middle goals.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

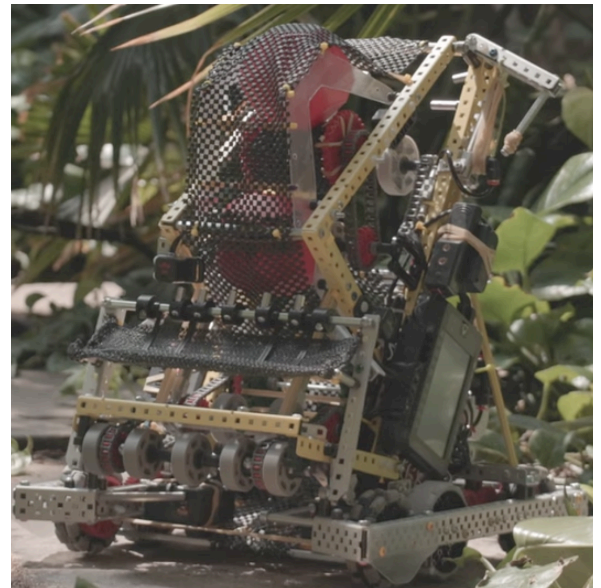
9/22/25 - 16610A + 16099A "S-Shaped" Brainstorm

Brainstorm

Tournament Review: Mall of America Signature Event **Champions** 16610A showed up with a unique front to back intake design in the shape of an S. More teams across the world started implementing the S-Shaped design, such as Overclock Robotics (2x World Qualifier at Highlander Summit)

S-Shaped Intake (Pits and Parts Interview with 16099A)

An **"S-Shaped"** intake design is one where a ball travels from the front and bottom of the robot in an S- Shape and exits the mechanism and is eventually scored through the top and back of the robot.



16610A Intake (16610A Youtube Mall of America Reveal)

Strengths: The intake is able to score blocks at a fast pace. It is also able to carry more blocks than the Ruiguan intake at the same time through an inbuilt "storage", due to the length of the system. Additionally, scoring on the middle goal and high goal becomes easier through one set of flexwheels.

Weaknesses: Necessary to create an arm for descoring blocks, which is prone to damage, and not as easy to utilize as a plastic piece that a front to back intake would utilize.

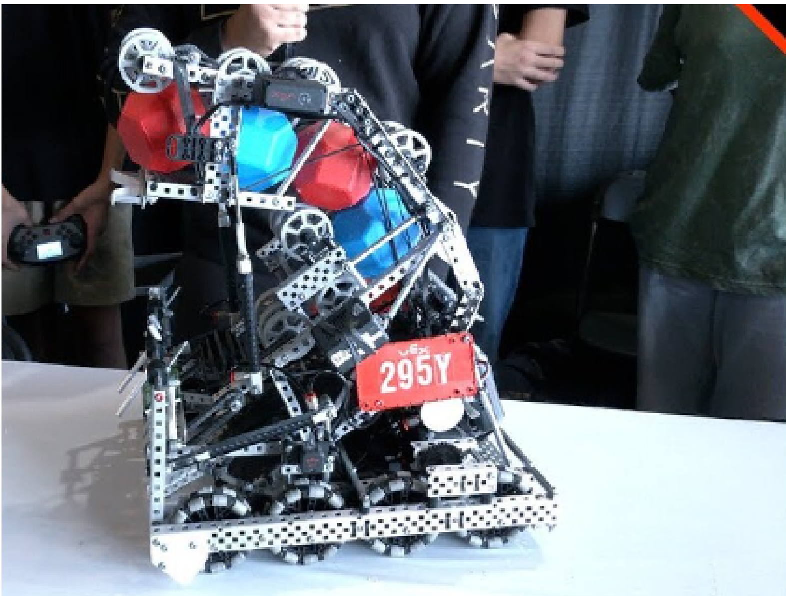
Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

9/22/25 - C-Bot Brainstorm

Brainstorm

No significant tournament was found with this bot in use proving to be clearly not in the meta but being off meta is not always bad as you can be the innovator however after further analysis and research we decided to go against the idea of using this bot

295Y C bot (Pits and Parts Interview/Vex Forum)
C intake is where the ball intakes and outtakes from the same side



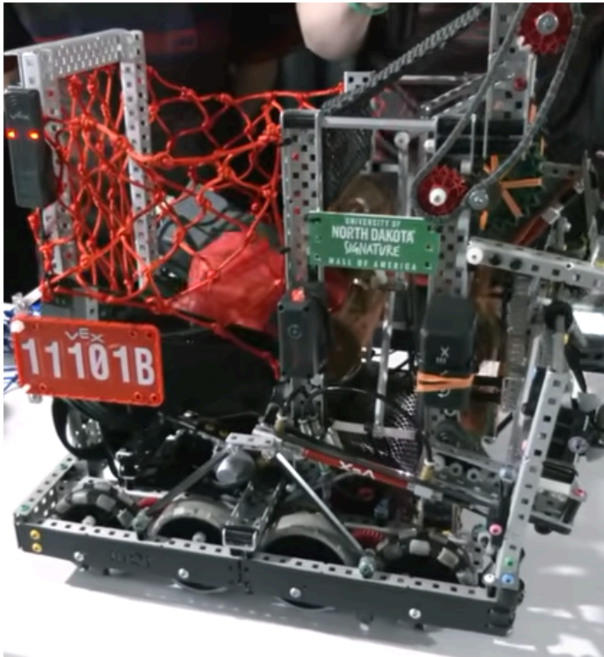
C Bot vs. S Bot

The main difference between the 2 is that a c bot intakes and outtakes from the same side whereas a S bot is shaped like an S as the name might imply but that also means that it intakes and outtakes from opposite sides which can allow quick scoring from match loader to long goals. However a C bot is faster at cycling but as a result leads to smaller max capacity compared to a S bot. Ultimately we realized that a C bot was not worth the major cons that comes with having to turn around to score and negated its quick cycling abilities as well as its small storage capacity which we planned to utilize as much as possible since this year there is no rule in amount of blocks you can hold along with holding opposing blocks as well.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

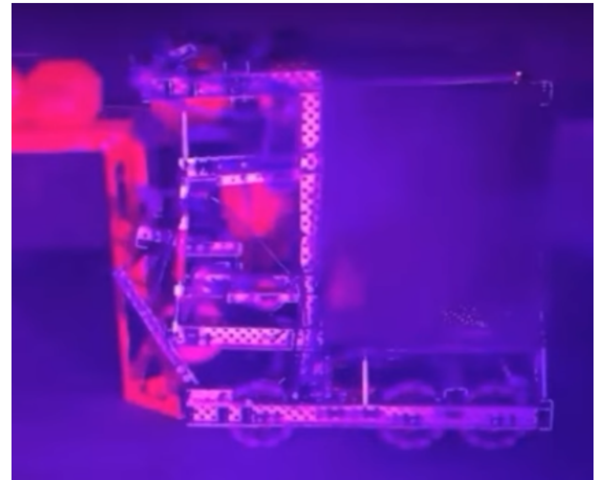
9/22/25 - Hopper Bot Brainstorm Brainstorm

Tournament Review: Mall of America Signature Event **Finalists** 11101B showed up with a unique basket-robot intake design. Other teams, including 10B,



Hopper Bot(aka Basket Bot) (Pits and Parts Interview with 11101B)

A hopper bot aka Basket Bot is characterized by its basket hence the name, which allows it to carry upwards of 15 blocks at a time



6842V Robot (6842V Youtube Mall of America Reveal)

Strengths: It is able to carry tons and tons of blocks a key aspect this year as there is no possession limit and no limit towards carrying other teams blocks as well. It also incorporates color sort which helps for scoring and for scoring in the middle having intake and outtake on the same side increases that speed.

Weaknesses: It is not able to score quickly as the blocks go from intake to the basket and outtake which has unnecessary time increase. Additionally its intake and outtake on the same side hurts the match loader to long goal speed which makes scoring much slower. This addition in speed hurts its autonomous performance as it needs additional movements and actions prior to being able to score.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

9/22/25 - Intake Design

Designing/CAD

Decision Chart for Intake Design:

	Speed	Ball Capacity	Build Time	Defense	Total
S-Shaped Robot	3	3	2	**4**	**12**
Ruiguan Intake	2	2	**4**	3	11
C Bot	**4**	1	3	1	9
Basket Bot	1	**4**	1	2	8

How we gave our point values: Each layout in consideration was ranked based on the four aforementioned categories, with 4 being the highest and 1 being the lowest. Our selection was based on which intake had the highest value.

How we created the categories: Speed is an important factor because the ball has to be scored fast. Ball capacity is important because it's better to be able to score around 6-7 blocks fast rather than having to score 3 at a time. The time needed to build the mechanism is an important factor as the time our team meets to build is limited given our conflicting schedules. Finally, popularity is an important factor to secure top-seeded alliances, ultimately contributing to the team's success in competitions.

Decision: Our final decision for Intake Layout is an S-Shaped intake. The benefits of this are that we have a higher ball-holding capacity and we have the capability to score on the middle goals.

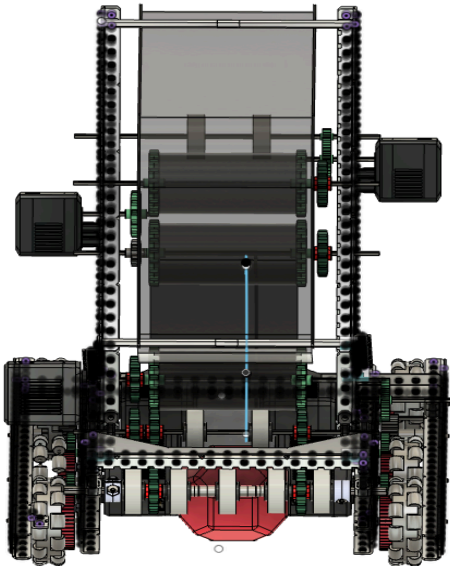
Drawbacks: Although the potential of the s-shaped robot is strong, teams tend to prefer a Ruiguan intake as it's specifically designed for quick de-scoring opportunities. This could have a slight effect in terms of alliance selections, however it shouldn't be an issue if we perform with excellence.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

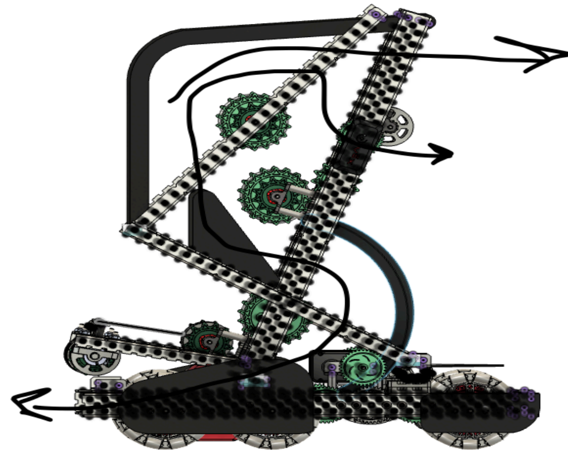
9/22/25 - Intake Computer-Assisted Design

Designing/CAD

Our first two stages of the intake consist of flexwheels, guiding the ball into the s curve.

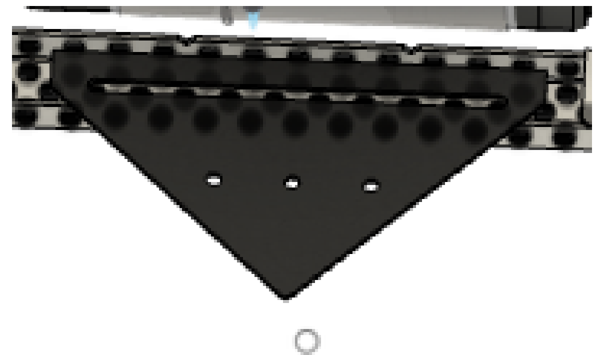


Front View of Intake Design (355Y)

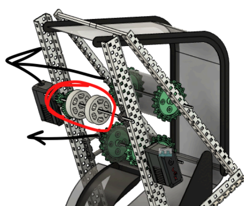


Drawn out pathways for the S-Shaped Intake (355Y)

An intake aligner is handy in this game. If the driver had to take time to align perfectly into the goal to score goals, it would take quite a bit of time, making it inefficient and wasting valuable seconds.



Top View of Intake Aligner (355Y)



End-Stage Flexwheels (355Y)

Highlighted in red are two flexwheels crucial for controlling the ending pathway of a block. When the axle spins in the positive direction, it sends the blocks upwards, allowing us to score on the high goals. However, if the axle spins in the other way, it sends the ball downwards, the perfect height for scoring blocks in the middle goal.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

9/22/25 - v1 Matchloading Criteria

Identify

Problem Statement: Design a piston-powered arm that can be utilized to dig under blocks in the matchloading zone and clearing them out.

Solution Constraints:

- Must work efficiently on a pneumatic cylinder maximum
- Must not extend past the 24” size constraint in one dimension

Solution Goals:

Be able to clear out 6 blocks in under 2 seconds.

- Our game strategy in push back involves utilizing matchloading sites to introduce more blocks into the playing field. Being able to quickly introduce a set of blocks can play a big role in splash plays and rapid scoring.

9/30/25 - Tongue-Mech Brainstorm

Brainstorm

The Problem with Our Current Design: Up until now, we've used static standoff guides (metal posts) to help us load blocks. While they are sturdy, they are very "unforgiving." If the block is even slightly off-angle during a matchload, it hits the metal and jams. In a 2-minute match, every second lost to a jam is a second we aren't scoring.

Tournament Review: Highlander Summit **Signature Event Champions** 16099A (Overclock) showed up with a unique matchloading mechanism, utilizing a rotating piece of plastic to guide blocks into their intake.

Matchloading Mechanism Regular Position (16099A Pits & Parts Interview)

Matchloading Mechanism Curved Position (16099A Pits & Parts Interview)

Strengths: Rotatability of the plastic piece allows for precise placement of the plastic piece allowing for rapid loading into the intake

Weaknesses: Time needed to tune the mechanism.

Decision Matrix: Evaluating the Change

We used this table to decide if the upgrade was worth the time it would take to build and tune. We scored each on a scale of 1–5.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Criteria	**Static Standoffs (Old)**	**Tongue-Mech (New)**	**Explanation**
Consistency	2	5	The Tongue-Mech "self-corrects" the blocks as they go in.
Loading Speed	3	5	We can load much faster without worrying about jams.
Durability	5	3	Plastic can wear down over time compared to metal.
Ease of Build	5	3	The Tongue-Mech requires custom cutting and fine-tuning.
TOTAL	**15**	**16**	**Tongue-Mech Wins**

Our Reasoning: We realized that "Ease of Build" shouldn't stop us from making a better robot. Yes, the Tongue-Mech is harder to make and might need more maintenance (Durability), but the **Speed** and **Consistency** it provides are what win matches. We are willing to put in the extra hours to tune the mechanism because we know it gives us a higher "ceiling" for performance.

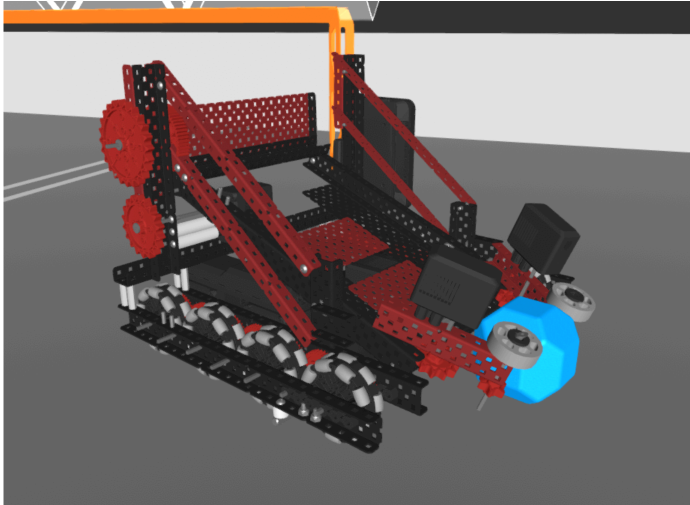
Action Plan:

1. **Prototyping:** Cut a 2-inch wide strip of polycarbonate and mount it to a pivot point.
2. **Tensioning:** Add a small rubber band so the "Tongue" always snaps back to the center after a block passes through.
3. **Testing:** Run 10 practice matchloads with the old system vs. the new system and record the time difference.

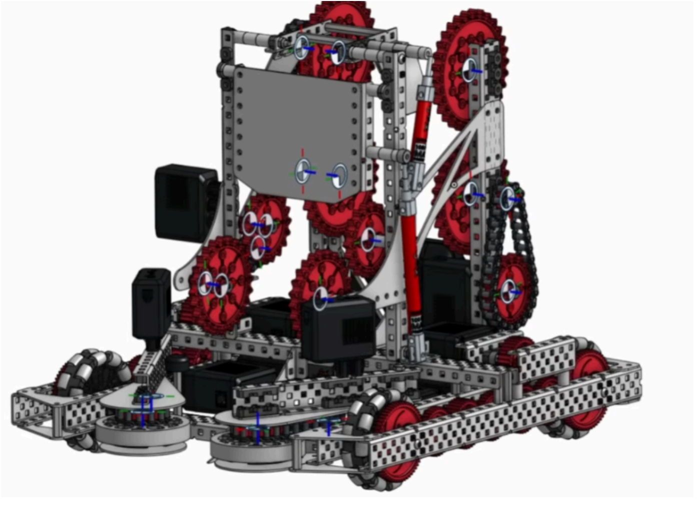
9/30/25 - Side Intake Brainstorm

Brainstorm

Tournament Review: Previous State Excellence Award Winners 44252A (Arsenic), although not an active team, presented an interesting type of matchloader with another major use: it also works as the intake.



Vex Forum: Push Back Robot Predictions



1346S Early Season CAD (Luke does robotics)

Strengths: Allows for both the intake and the matchloader to be the same contraption, simplifying maintenance and building. It is also an extremely fast matchloader, which is useful in an extremely fast and rapidly-moving game such as PushBack.

Weaknesses: While this "all-in-one" solution offers incredible cycle speeds, it comes with a high "power tax." In the V5RC *Push Back* season, we are limited to **88 Watts** of motor power. An integrated side-intake typically requires two dedicated motors to ensure the torque is high enough to pull in blocks rapidly. We had to decide if the speed gained in matchloading was worth the loss of motor power in our drivetrain or scoring lift.

Weighted Decision Matrix: Side-Intake vs. Front-Intake

We evaluated the Integrated Side-Intake against our current front-facing design to determine if a total redesign was strategically sound.

Selection Criteria	Integrated Side-Intake	Front-Facing Intake	Engineering Rationale

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Cycle Speed	5	3	Side-intakes allow for near-instant loading during the matchload period.
System Simplicity	4	2	Combining two systems into one reduces the total number of moving assemblies.
Motor Efficiency	1	5	The side-intake "costs" 2 motors; the front intake is more power-efficient.
Ease of Tuning	2	4	Side-intakes require complex geometry to ensure blocks don't jam at high speeds.
TOTAL SCORE	**12**	**14**	**Front-Facing Intake Wins**

Final Decision: Rejection of Integrated Side-Intake

After a thorough team discussion and review of the matrix, we have decided **not to proceed** with the Integrated Side-Intake.

Justification for Rejection:

1. **The 88W Constraint:** Our primary concern is the motor budget. By choosing a more traditional front-intake, we can reallocate those two motors to our drivetrain or a high-torque scoring mechanism. We believe that having a stronger, faster drive is more valuable throughout the entire 2-minute match than having a specialized matchloader that is only used for a few seconds.
2. **Tuning Complexity:** As noted in our review of 44252A, these mechanisms are notoriously difficult to tune using only standoffs. We determined that the "ROI" (Return on Investment) for our time was too low; we would rather spend that time perfecting our autonomous code and driver practice.

Conclusion: While the Side-Intake is a brilliant piece of engineering, it does not align with our "Power-to-Performance" strategy for this season. We will stick with our front-facing design and focus on making it the most reliable intake in our region.

Thus, we created the following table to map out how many motors we will use for each function of our robot:

Motor Map:

88W Motor Allocation Map

This map outlines our strategic distribution of the 88-watt power limit. By opting for a simplified, 1-motor intake system rather than the 2-motor integrated side-intake, we have reallocated **22% of our total power** back into our drivetrain for better pushing power and defensive resistance.

Subsystem	Motor Count	Total Wattage	Strategic Justification
Drivetrain	6	66W	Essential for high-speed cross-field travel and winning "pushing battles" in the *Push Back* game.
Primary Intake	1	11W	Efficient 1-motor design (S-Shape) provides high capacity without draining the power budget.
Scoring / Lift	1	11W	High-torque gearing allows a single motor to handle vertical scoring requirements.
TOTAL	**8**	**88W**	**Maximum legal power utilized for peak performance.**

9/30/25 - v1 De-score Criteria

Identify

Problem Statement: Design a piston-powered arm that can be utilized to clear out blocks in the scoring zones

Solution Constraints:
- Must work efficiently on a pneumatic cylinder maximum
- Must not extend past the 24" size constraint in one dimension

Solution Goals:

Be able to clear out 6 blocks in under 1 seconds.
Be able to utilize de-score mechanism for both removing blocks and gaining control bonus
- One of our game strategies in push back is to keep as many opponent blocks away from the scoring zones as possible, while also maintaining a control bonus of each scoring zone

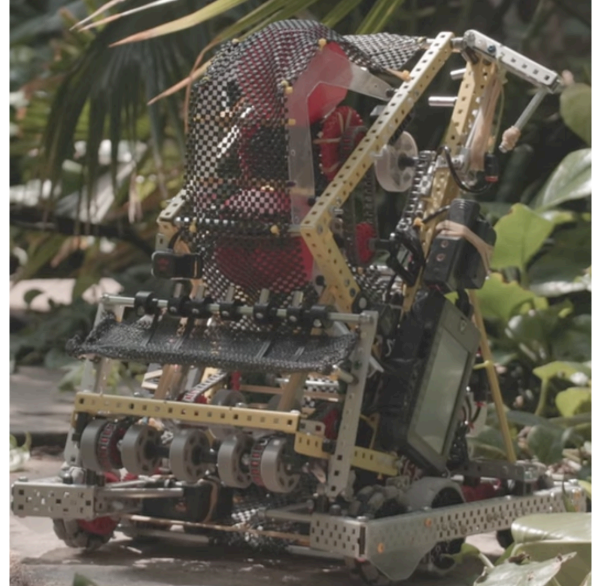
Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

9/30/25 - 16610A Double Wing Mech Brainstorm

Brainstorm

Tournament Review: Mall of America Signature Event **Champions** 16610A showed up with a unique wing design, powered by two pistons. Teams have started to utilize this wing mechanism more often.

The "**Wing**" extends out of the robot, powered by a piston and rotates off of a screw joint mounted at the top of an intake.



16610A Wing (16610A Youtube Mall of America Reveal)

Strengths: The design is compact, and not very heavy. Additionally, the mechanism is very durable if built well.

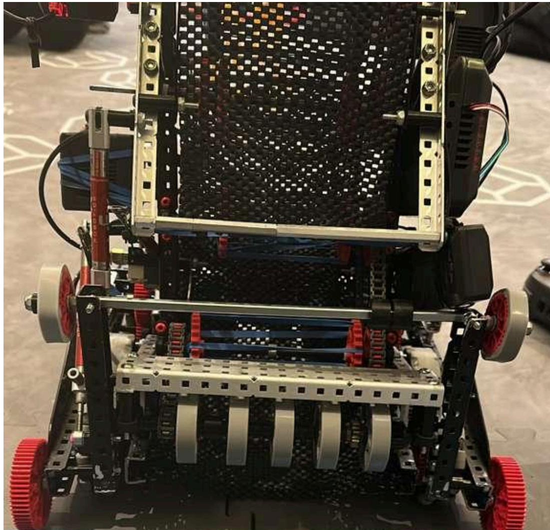
Weaknesses: Aligning to the long goal is really difficult since the wing's furthest is super close to the robot's drivetrain end.

As a team, we unanimously chose to build this design instead of a puncher.

10/10/25 - Matchload Mechanism Build

Build

Designing the matchloader mechanism for our bot is a vital step to succeeding in Push Back, since it allows us to both clear out blocks in the matchloading area and more efficiently pick up blocks in our intake. Without an efficient matchloader/tongue mech, it becomes very difficult for our robot to intake blocks to score.



Matchloader Up (355Y)

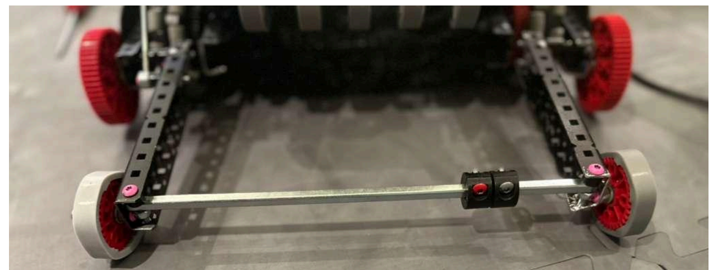
Our intake supports a tongue mech, but it becomes difficult to add a bulky matchloader on top of that, which is why we opted to stick with the sleeker and faster polycarbonate model instead.

Our robot dimension is 20 holes long by 11 holes wide, giving us about 10” to work with on the tongue mech. The small width aligns with the width of the ball, so we can strategically design the matchloader to reduce jamming at the beginning of the intake.

The inside space is about 10 inches x 5.5 inches, while the outside space is about 14.5 inches x 12.5 inches.

We are thinking of adding a small spinning polycarbonate piece to add to our matchloader to get under blocks that makes it smoother to unload blocks and intake them during games.

We are going to implement this design (according to the CAD) by laser cutting a piece of plastic and attaching it to our tongue using rubber bands and screws.



Matchloader Down (355Y)

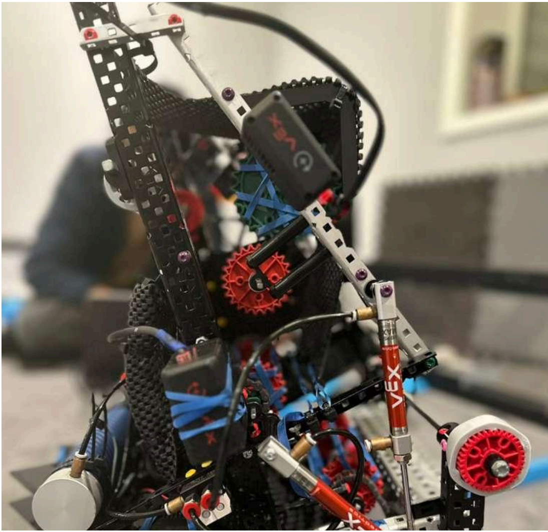
Our matchloader mechanism will be a rotational piece of 3 inches x 6 inches polycarbonate that adjusts its angle based on what angle the forces are coming from to reinforce the intake.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

10/13/25 - Intake Mechanism Build

Build

Making an intake is one of the most essential parts of making any bot, and without having one it becomes very difficult to score any points. Intakes don't just complete bots, they are bots. Because of this, the intake was one of the first parts we designed, and it was one of the most time consuming mechanisms on the robot.



Intake Side (355Y)

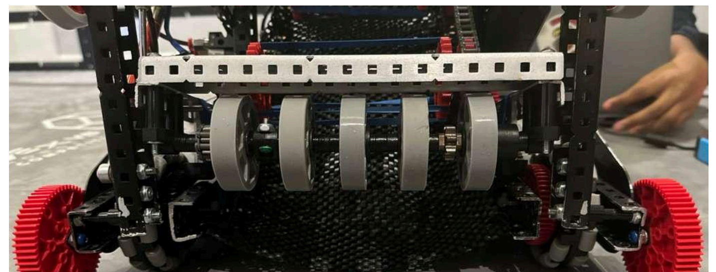
Our intake is extremely constrained due to the rollers, so we opted to use an S-Shaped intake, similar to 16610A(Snacky Cakes) or 16099A(Overclock, instead of one that's closer to 9123X (Ruiguaan), which goes straight through the bot.

Our robot dimension is 29 holes long by 25 holes wide, giving us about 12" to work with on the tongue mech. The small width aligns with the width of the ball, so we can strategically design the matchloader to reduce jamming at the beginning of the intake.

The inside space is about 12 inches x 6 inches, while the outside space is about 15 inches x 8 inches.

We are implementing a small plastic piece so that we can make sure there is more space on the bottom of our robot for important aspects like more odometry sensors and a potential spot to place our battery or gas cylinder.

We are going to implement this design (according to the CAD) by using our design to build a replica.



Intake Front (355Y)

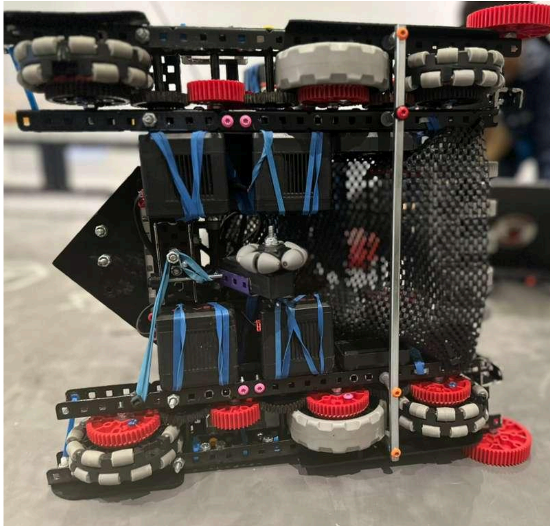
Our matchloader mechanism will be a rotational piece of 3 inches x 6 inches polycarbonate that adjusts its angle based on what angle the forces are coming from to reinforce the intake.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

10/15/25 - Drivetrain Build

Build

Designing the drivetrain took a long process, which required many hours to build. Our drivetrain has a 6 blue motor drivetrain which allows us to move around the field quickly. We also have an idler gear which is for stability on our robot.



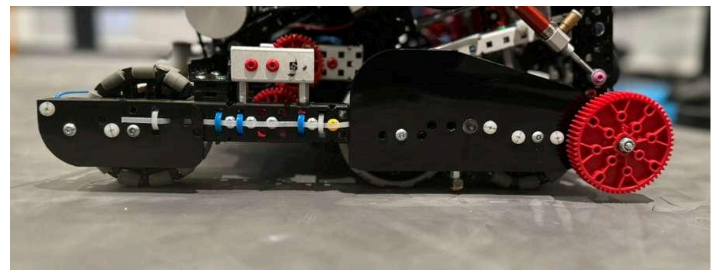
Drivetrain Bottom (355Y)

Our drivetrain uses blue motors which allows us to get around the field faster. This is helpful because if we need a ball that is far away, we can get it in a quick amount of time.

Our robot dimension is 14.5 inches by 12.5 inches which allows us to have more space for our intake.

The inside space is about 10 inches x 5.5 inches, while the outside space is about 14.5 inches x 12.5 inches.

We are implementing a small plastic piece so that we can make sure there is more space on the bottom of our robot for important aspects like more odometry sensors and a potential spot to place our battery or gas cylinder.
We are going to implement this drivetrain design (according to the CAD) by mounting the motors and attaching the wheels to the base for smooth and stable movement.



Drivetrain Side (355Y)

Our drivetrain will be 14.5 inches by 12.5 inches which helps us gain more space incase we need an intake or space to build..

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Early Season Meetings

9/6/25 Meeting #1- Drivetrain Structure

Goal: Start building robot by creating drivetrain structure.

This is our first meeting, and as a team, we had already finalized our design as mentioned above using Computer-Assisted Design Software. Our first step is to build a completed drivetrain.

The Drivetrain is the most fundamental aspect of the robot, as every other mechanism is mounted about it. Having a strong, durable, and fast but efficient drivetrain not only helps with proper maneuvering but also helps with being able to easily mount items without any issues. We planned out our drivetrain in a way that we could mount the intake structure's main c channels and braces in a proper and convenient way.

Our drivetrain wheel gap is 3 holes wide, allowing us to remain narrow which helps with maneuverability in push back. The overall drivetrain dimension is 14.5" by 12.5", with inner c channels of 13.5".

We were able to mount two full-length crossbraces that square the drivetrain, with the top being a full-length c-channel and the bottom one being a high-strength axle.

We started by taking a 35 hole c-channel with a 5 wide gap. This sets as the temporary foundation for squaring our drivetrain. Using shoulder screws, we mapped out where the c channels have to be arranged.

After attaching the two outer 29 hole c channels and the 2 27 hole inner c channels, we proceeded with the top crossbrace. We kept the top crossbrace at the back of the robot as this was where we would mount our intake bracing angles. Using bearings to square the brace, we were able to attach 8 screws without problem to secure it in.

From there, we took our pre-drilled high strength axle and proceeded to attach it to the bottom and front of the drivetrain, connecting it via 0.5" standoffs.

When checking for sturdiness after removing the original, temporary squaring c-channel, the drivetrain remained sturdy and unable to bend around. This meant that we were ready to proceed to creating drivetrain screwjoints.

Was goal reached? **YES**

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

9/13/25 Meeting #2- Creating Intake Structure

Goal: Create all 8 drivetrain screwjoints

Now that all of the Drivetrain structure has been completed, the next goal was to finish the drivetrain altogether. Unfortunately, we did not have the parts to create a screwjoint. We placed an order in for circular inserts and spacers to use to create the screwjoints, however **we basically failed to reach our original goal**. But we worked around this issue by instead resuming the robot structure on the intake. For the intake, we had to mount 7 pieces of metal, including two vertical c channels, 2 angles of bracing, and 3 more extensions to support all of the rollers needed on the robot.

The 2 bracing c channels and the two vertical c-channels will be mounted using angles, a low-weight but efficient and strong method of mounting large pieces of structure on a robot. We cut four small angles, two to be attached on the top crossbrace, and two to be attached around the middle of the inner c channels on the drivetrain. From there, we proceeded to add the two vertical and two bracing pieces of metal, creating our base structure. Using locknuts and shoulder screws, we securely and properly positioned all pieces of metal on the robot to be further built on.

After that, we attached 3 more pieces of metal to complete the base for the S shape on the intake. Using shoulder screws, angles, and locknuts, we fastened everything properly and made sure everything was sturdy. When checking to make sure nothing was lose on the robot, we held the drivetrain by a single piece of metal rather than from the base, and it proved strong as it picked up the entire robot structure with ease. This is a good sign that the base structure for the intake was built properly, however time will tell if this remains true once we attach every aspect of the robot on.

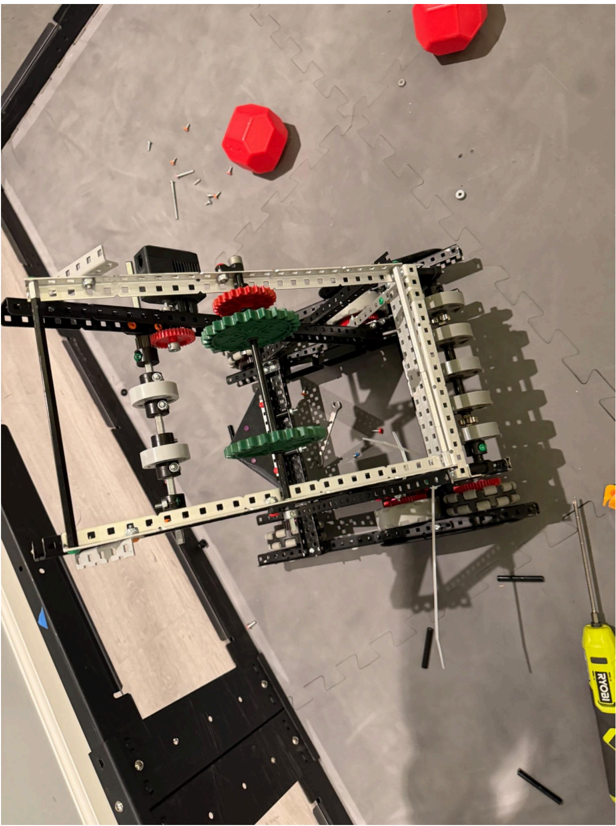
We proceeded to create the first flexwheel stage of the intake to attach onto the robot. When deciding how many flexwheels to use, we chose to use 5 wheels as that was an average estimate of flexwheels that teams across the globe are using. Two flexwheels don’t sound realistic for this game as it would be difficult to grab blocks from the corner of your intake. Using a 3 wide c channel to hold the two angles together, we attached the final contraption to the robot via the vertical c channels, using screwjoints to give it play and rubber bands to make sure that the flexwheels are always applying the right amount of pressure on a block. When testing to check for changing the height of the intake stage, the mechanism was able to smoothly collect the block without any error or jam or other problem, so we kept it at our estimated height.

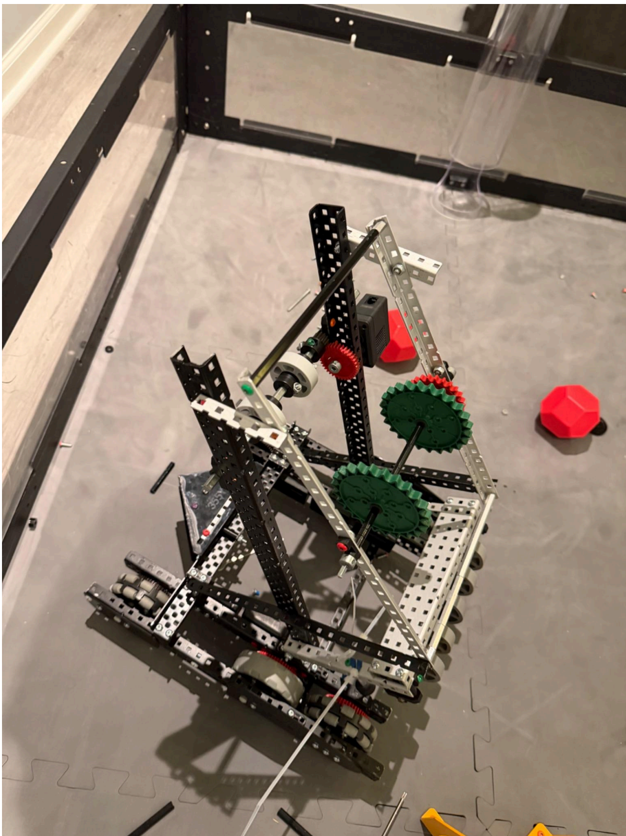
Front-Stage Intake Test-

Test #	1	2	3	4	5
Result (Yes or No)	Yes	Yes	Yes	Yes	Yes

Was goal reached? **NO**

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke


RYOBI



9/20/25 Meeting #3- Drivetrain Screwjoints + Axles

Goal: Finish all 8 Drivetrain Screwjoints + Axles

We received all of the parts necessary to construct the drivetrain screwjoints today. Our plan is to attach all 8 of them, 6 of them being for actual wheels and two of them being for idler gears. Given that our screwjoints have to be properly spaced out to give wheels as little slop room as possible, we calculated the amount of spacing we would need to successfully add the screwjoint.

In a 2 inch gap for screwjointed wheels, we used:
Bearing - 0.25 inch
Kepsnut- 0.1875 inch
Spacer- 0.25 inch
Spacer- 0.125 inch
Washer- <0.0625 in
Gear- 0.25 inch
Wheel- 0.5 in
Washer- <0.0625 inch
Locknut- 0.25 in

In total, this process took us around 4 hours. We individually tested each wheel for how long they spun for after installing each joint to see if any wheel experienced a friction force. Ideally, each wheel must be able to spin for around 10 seconds, preferably more.

Wheel 1-

Test #	1	2	3	4
Duration of Spin (seconds)	10 seconds	12 seconds	15 seconds	11 seconds

Wheel 2-

Test #	1	2	3	4
Duration of Spin (seconds)	13 seconds	12 seconds	13 seconds	11 seconds

Wheel 3-

Test #	1	2	3	4
Duration of Spin (seconds)	12 seconds	17 seconds	16 seconds	10 seconds

Wheel 4-

Test #	1	2	3	4
Duration of Spin (seconds)	10 seconds	12 seconds	12 seconds	12 seconds

Wheel 5-

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Test #	1	2	3	4
Duration of Spin (seconds)	11 seconds	13 seconds	12 seconds	10 seconds

Wheel 6-

Test #	1	2	3	4
Duration of Spin (seconds)	10 seconds	13 seconds	15 seconds	10 seconds

Next, we did our two idler gears. The layout on a 2 inch gap:
Bearing - 0.25 inch
Kepsnut- 0.1875 inch
Spacer- 1 inch
Washer- <0.0625 in
Gear- 0.25 inch
Washer- <0.0625 inch
Locknut- 0.25 inch

When testing (target seconds: 6):

Idler 1-

Test #	1	2	3	4
Duration of Spin (seconds)	6 seconds	seconds	7 seconds	6 seconds

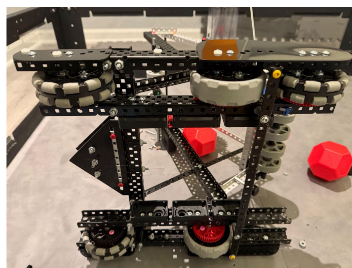
Idler 2-

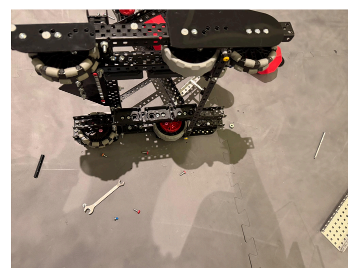
Test #	1	2	3	4
Duration of Spin (seconds)	6 seconds	8 seconds	7 seconds	7 seconds

All of our tests being above the target duration is a good sign, however further testing with the powered motors are needed to confirm that everything is good.

We proceeded to attach all of the axles afterwards, which didn't take long. By the end of the day, we finished our drivetrain.

Was goal reached? **YES**





9/27/25 Meeting #4- 2nd + 3rd Stage of Intake

Goal: Finish the second and third rollers of the intake

Now we needed to start on what turned out to be one of our most difficult meetings yet. As the rollers were being added, we quickly understood that we needed to adjust a lot of the spacing between the rollers to get rid of **blind spots or overexposed areas in the intake**.

We eventually realized that this was because of some **inconsistencies within the CAD**, which we promptly fixed. We moved both of the rollers up a few inches up compared to our original design in our CAD because after the ball was taken in we noticed that it dug completely into the rubber bands we are using in the rollers. Because of this CADding and spacing problem, we took about 30 minutes trying to find the correct spacing to avoid the blind areas we experienced sometimes at the beginning of the intake due to the spacing changes.

After about 30 minutes of trial and error on both the bot and the CAD to install the second roller, we moved forward and started working on the third roller in accordance with what we CADded for the second roller.

Finally, we proceeded to add the mesh “ramp” to our intake to push the blocks up, to keep them away from the motors, towards the rollers, and towards the rest of the mesh we added later. We played around with the mesh for a little bit, because we couldn’t predict its path on the CAD. We used trial and error and attached the mesh directly on the robot. After we found the proper tightness and areas to hold back, we reinforced our mesh “ramp” path with zip ties.

When reinforcing the mesh, we also added some small pieces of metal we knew we needed anyway to tie the mesh to, further strengthening the desired channel in the mesh.

Was goal reached? **YES**

Second + Third-Stage Intake Test-

Test #	1	2	3	4	5
Result (Yes or No)	Yes	Yes	Yes	Yes	Yes

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

10/4/24 Meeting #5- 4th + 5th Rollers of Intake

Goal: Attach the rest of the rollers in the intake

Finally, in this meeting we get to finish up the second stage of our intake in order to be able to get the ball from the ground into the middle and high level goals. This is a crucial step, because making sure that the gaps are accurate allow for the ball to be able to be redirected **over or under the flex wheels to score on different goal levels**.

In this portion, we were able to use the CAD almost directly, because we were able to measure the distance in CAD; however, a discrepancy occurred because we were unable to test the elasticity of the rubber bands that are in between the sprockets, hence we had to adjust the distancing by ±1 hole in order to make sure that the ball has fluid motion.

Once those adjustments were made, we conducted **several trials** to ensure that the ball maintained consistent velocity as it transitioned between the stages. Through this process, we identified **minor variations in compression** that influenced how smoothly the ball transferred into the upper path. By **optimizing the distance between the rollers**, we were able to achieve a much more stable passageway for the ball.

Was goal reached? **YES**

Fourth + Fifth-Stage Intake Test-

Test #	1	2	3	4	5
Result (Yes or No)	Yes	Yes	Yes	Yes	Yes

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

10/11/25 Meeting #6- S-Shaped Curve

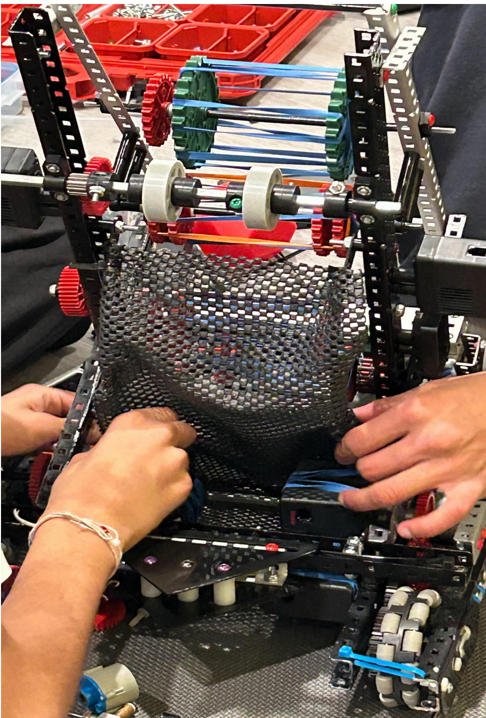
Goal: Installing the s-shaped curve

In this meeting, we had to attach both sets of plastic curves and attach the overlapping mesh, giving us the needed **“s-shape”** for the blocks to travel along the intake.

We started by adding the bottom two sets of curves. We mounted them along the vertical c channels using locknuts and screws. From there, we wrapped mesh around the curves. When testing with the rollers, we found that the ball consistently passes through the bottom curve.

Our next step was to attach the top curve. Using screws and locknuts, we fastened the two plastic pieces, and wrapped both pieces with mesh. When testing, the ball was able to pass through this curve effectively.

Was goal reached? **YES**



Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

10/18/25 Meeting #7- Matchloading

Goal: Prototype the Matchloading Mechanism and Attach the Descoring Mechanism

Problem Statement

During early gameplay simulations for Push Back, we recognized that our robot needed a more reliable and controlled way to push game objects efficiently during matchloading situations.

Our initial matchloading structure lacked:

- Surface area for consistent pushing
- Stability during contact
- Efficient transfer of force to game elements

This resulted in inconsistent pushes and occasional slipping.

Constraints

- Must remain within size limits
- Must not interfere with intake system
- Must remain lightweight
- Must use existing structural frame

Proposed Improvement

Instead of building a complex mechanism, we chose to modify the existing matchloading structure by:

- Attaching a 1x1 metal sheet to the front face
- Increasing surface area for improved contact
- Improving structural rigidity
- Allowing more consistent pushing during driver control

This was chosen because it:

- Required minimal additional weight
- Did not require extra motors
- Preserved mechanical simplicity

Build Process

- Mounted two c channels off the insides of drivetrain, mounted on screwjoints to allow both channels to pivot.
- Attached a drilled high-strength axle to connect both c channels, also allowing us a place to mount a 1x1 metal sheet

---

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

- Mounted 1x1 metal sheet to the front of matchloading mechanism
- Secured using screws and lock nuts
- Ensured flush alignment with pushing surface
- Reinforced connection points to prevent bending

Spacing and alignment were adjusted to prevent interference with intake and drivetrain components.

Testing Procedure

Success Criteria:

- Push at least 3 objects consistently in one motion
- No structural bending
- No negative effect on intake performance

Testing Results

Test Metric	Before Metal Sheet	After Metal Sheet	Result
Max Objects Pushed	2	4	Improved
Slipping During Push	Frequent	Rare	Reduced
Structural Flex	Moderate	Minimal	Improved
Driver Feedback	Inconsistent	Stable & Controlled	Improved

Results & Reflection

The addition of the 1x1 metal sheet significantly improved pushing consistency without increasing mechanical complexity.

This iteration demonstrated that small structural adjustments can create meaningful performance gains.

Rather than overcomplicating the design, we prioritized:

- Simplicity
- Stability
- Force distribution

Was goal reached? **YES**

10/24/25 Meeting #8- Attaching all Pneumatics

Goal: Attach all the pneumatics to the pistons

In this meeting the team wanted to focus on attaching all of the pneumatics. So far, on the bot we had no pneumatics attached but during this meeting we added pistons and a tank.

For our tank we decided to use zipties to secure it into place at a reasonable place we found. We then attached the valves to the tank and filled it with air.

Then, we attached the pistons to our matchloading mech so that we can control when it goes up vs. down; and also to our intake roller, so that it can stiffen up when we need to park at the end of the game.

How this works is that, because we have sleds, we can park on top of the boxes and when we get on, we can angle the roller towards the ground and make it rigid, effectively parking our bot.

At the end, we finally added the air tubes to the tank and connected them to the pistons. Then, we coded the pneumatics controls and tested them, making sure they work.

Was goal reached? **YES**

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

10/25/25 Meeting #9- Friction Testing Drivetrain, Intake

Goal: Reduce drivetrain and intake friction to improve overall efficiency and motor performance

Problem Statement

During driving tests, we observed that the drivetrain required more power than expected and felt resistant during acceleration. The robot showed:

- Slower response time
- Increased motor temperature
- Slight battery drain increase
- Audible gear resistance

We hypothesized that internal gear friction was reducing drivetrain efficiency.

Root Cause

After inspection, we identified that:

- Stacked motors were creating alignment stress
- Gears showed slight bending under torque

Additionally, we discovered minor friction in the intake sprocket connection due to improper spacing.

Constraints

- Must maintain compact drivetrain structure
- Must not redesign entire chassis
- Must stay within time limitations before next scrimmage
- Must preserve gear ratio

Implemented Solution

Drivetrain Stack Motor Fix:

- Replaced standard gear with high-strength gear
- Re-adjusted spacing using spacers and collars

Intake Fix:

- Re-aligned intake sprockets together
- Adjusted spacing to reduce unnecessary friction

Spacing adjustments required multiple small iterations to find optimal mesh alignment, including playing around with thin washers, and extra thin washers.

Testing Procedure

Success Criteria:

- Reduced motor temperature after 2-minute drive
- Smoother free spin test
- No audible grinding
- Improved acceleration time

Drivetrain Friction Testing

Test Metric	Before Fix	After Fix	Improvement
2-Min Motor Temp (°C)	Really hot	Barely any heat	Present
0–Full Speed Time	1.4 sec	1.1 sec	0.3 sec faster
Free Spin Duration (manual push)	1.8 sec	3.2 sec	+1.4 sec
Audible Grinding	Present	None	Eliminated

Results & Analysis

After switching to high-strength gears and optimizing spacing, drivetrain friction was significantly reduced. The free spin duration nearly doubled, indicating less mechanical resistance.

Motor temperature dropped, confirming reduced workload on the drivetrain.

Intake gear realignment eliminated minor binding, improving consistency and reliability.

The spacing issue required multiple micro-adjustments to balance between:

- Excess compression (binding friction)
- Excess looseness (gear slippage)

This reinforced the importance of precise gear alignment in stacked motor configurations.

Was Goal Reached? **YES**

10/31/25 Meeting #10- Autonomous Programming

Goal: Coding the autonomous algorithm for our robot

In this meeting, we worked on finally hooking up our custom autonomous library with the robot, which is a difficult step because we needed to debug some of the syntax.

The first order of business was setting up our proportional integral derivative (PID) controller. This allows the robot to smoothly move or turn without under- or oversteering.

Unfortunately, the process of tuning PID took a lot of trial and error to find the best set of parameters. This lost us a lot of time, but it worked out in the end.

We ended up with a decent tuning, though there are still some issues with oscillations that we'll have to iron out soon.

Additionally, we set up our first basic autonomous routine for our qualification matches, allowing us to theoretically score 4 balls into the side goal.

Was the Goal Reached? **YES**

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Tournament Recap #1

Competition 11/01/2025: Lake Park

Results:
Round of 16 (Lost by 16 Points)

Immediate Action:
As the team was reviewing how the tournament was, we were reflecting on future events. We realized one of the main problems yesterday was that our color sort on the red alliance sorted the wrong color and scored blue blocks instead of blue, but when we realized this the competition was already over. Additionally, we realized the importance of our driver, and we held driver trials shortly after to make sure we have a backup driver ready in case of problems, along with fixing the color sort. Another good work item we did was decide on the rest of the competitions we are attending, mainly in the mid-season, around January and onwards. The fourth and final thing that we adjusted during that meeting was fixing up some of the mesh, since there was a gap in the intake mesh that caused the balls to keep jamming before they could get scored.

Team Reflections:

Mithil (Designer) - I feel that in this competition we were on a big time crunch, as a team we started later than most; however, on the good side we had a majority of the bot built. On the other hand, this lack of time hindered us from tuning our bot which is crucial when optimizing ball outtake speed. I feel that with a little more time and tuning we should be able to get a competition winning robot.

Naisha (Mechanic) - Today was a great learning experience for everyone, understanding how the competition environment is, getting a look at early season strategy, and also it's good to know that there is lots of work to do to reach the top of the top in the state.

Sathvik (Driver, Mechanic) - We started really well in the competition going 2-1 but once our robot's matchloading mechanism had to be restricted gameplay got really difficult. It was a great learning experience and should definitely factor into our rebuild.

Shaurya (Mechanic) - I think we definitely could have done better than we ended up doing but I'm still proud of what we achieved and the flaws we found which are vital to the improvement of our robot.

Shreyas (Mechanic) - We were really good at the beginning of the competition, but once we got those violations to not use our tongue mechanism that really impacted us as we scored very low after that incident. However, we used this knowledge to improve our robot for the future.

Sonit (Programmer) - I'm proud of our team despite our setback of not having enough time to have a tuned robot. Programming needs a lot of work.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Goals After Lake Park Competition

Build Outtake Stopper - Takes advantage of capacity in the robot to hold and score in bursts rather than scoring 1 or 2 at a time. Ideally, we can do this by either using a standoff that stands in the way of our scoring path to restrict the balls, or we could create a plastic hood that is powered by a piston cylinder.

Rebuild Matchloading Mechanism - One of our biggest issues at the competition was that we got called for three minor violations that included the 6th block flying out of the matchloading cylinder because of the speed and force our matchloading mechanism put on the blocks. This factored into our gameplay as we started to lose matches when we realized that we had to disable **for future design cycles**, we need to have a reliable tongue that does not cause minor violations. Instead of a metal plate, we should create our matchloader using a rotating plastic piece that can change its position as it enters the matchloading station.

Watch Game Film - Part of becoming a better driver includes watching game film to analyze missed opportunities **AND** to scout our opponents for future matches. Sathvik plans on taking a dedicated hour to watch his game film and find spots where he can make obvious alternate choices to further our result.

Build De-score Mechanism - One disadvantage to our robot was that we didn't have any means of removing blocks in an already-scored position within the field. Not only does this make our robot less capable, but it also reduces our appeal to other teams for potential alliances. **We anticipate having a consistent, reliable de-score mechanism will be important come mid-season.** The solution to this would be to create a wing that is powered by a piston cylinder that reaches into the goal and pushes them out or inside the control zone via a standoff sticking out, similar to how 16610A built their wing.

Film Analysis of Lake Park Matches 11/2/25

Q4	Score
38535K + 2360N	39
355Y + 2360C	37

We started this match with zero autons a piece. Our alliance, 2360C, had only built their bot on the bus ride to the tournament and didn't have scoring ability. Sathvik was able to score 8 balls but was outplayed by control zone bonuses, and lost 39-37.

Important takeaway: Control Zones swing matches
Minor Violation: Tongue mech can't hit the matchloading station fast enough where a block flies out.

Q14	Score
12543E + 60441Y	19
60172W + 355Y	52

Zero auto, again. Since neither of the three teams at this match outside of us were able to score at a decent rate, we were able to take this match simply. We earned yet another minor violation in this match. **Same minor violation earned again.**

Q29	Score
355Y + 355U	93
499G + 60172V	13

355U had a really great auton, scoring 7 blocks to ultimately dominate that area. As Sathvik aligned to the long goal, our robot ultimately disconnected (unknown as to whether its a field issue or code issue). This was a comfortable win with 355U on our side.

Q46	Score
355Y + 60441X	27
2360A + 2360S	115

Autos were lost to Blue alliance, and from there our untuned and lack of helpful alliance ultimately led to our loss in this match. **Third minor violation earned in this match**

Strategy: Since we earned our third minor violation in this match, unfortunately we were stuck with **avoiding** the matchloader until the elimination round.

Q57	Score
355P + 333Y	118
355Z + 355Y	26

On our side, we were able to score one ball on autons, but unfortunately 355z's autons did not hit. The red alliance had a really strong auton, and it set them up for a comfortable victory.

Q77	Score
60172X + 23880A	16
321H + 355Y	5

Zero autos in this match. We had a comfortable lead until Sathvik got pushed around and hit the middle goal, de-scoring all of our blocks and ultimately our control bonus.

Eliminations: Due to a collection of unfortunate factors, we ended up 2-4. However, we convinced the 12th seeded 2360N for an alliance, securing our spot as the 8th seed.

R16 2-1	Score
2360N + 355Y	35
2360C + 2360S	51

We won autons off the get-go scoring our singular block, and scored around 9 blocks, securing one control bonus and dominated the one long goal we scored on. However, due to a double park by the blue alliance, we lost that match.

Takeaway: A double park can flip the result of a game.

11/7/25 Meeting #11-De-Score Mech

Goal: Design and implement a mechanism for rapid field control

Problem Identification

During match simulations, we observed that our robot lacked an effective method to quickly clear multiple game objects and control space. This limited our cycle speed and reduced scoring efficiency.

We determined we needed a wing that:

- Deploys instantly
- Clears multiple objects in one motion
- Retracts reliably
- Does not interfere with intake or drivetrain
- Stays within size constraints

Constraints:

- Limited motor ports available
- Must comply with expansion rules
- Must not reduce drivetrain stability

Brainstorming Solutions

We evaluated three deployment methods:

Option 1 – Motor-Driven Hinged Wing

- Controlled movement
- Uses motor port
- Slower deployment
- Adds drivetrain load

Option 2 – Passive Spring-Loaded Wing

- Lightweight
- No motor required
- Hard to control retraction
- Risk of inconsistent deployment

Option 3 – Pneumatic-Driven Wing (Selected)

- Instant deployment
- High force output
- Does not use motor port
- Clean driver control via toggle

Decision Matrix:

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Criteria	Motor	Passive	Pneumatic
Deployment Speed	3	2	5
Reliability	4	2	4
Resource Efficiency	2	5	4
Driver Control	4	1	5
Total	29	21	**38**

Build Process

- Mounted pneumatic piston to tall c channel
- Connected piston to hinged wing arm using screw joint
- Installed mechanical stop to prevent overextension
- Routed tubing internally to avoid entanglement

Wing opens outward at approximately 90° when activated. It reached inside the goal and was able to remove scored blocks quickly. To ensure reliability, we ran several tests of deployment:

Test Number	1	2	3	4	5
Result	Descore!	Descore!	Descore!	Descore!	Descore!

Result: Consistent enough to proceed with. The pneumatic system was powered by a regulated air tank and controlled through a driver toggle button.

Afterwards, we monitored air consumption, as our tank holds at max 100psi and it is imperative that we use as less psi as possible per each deployment. After testing, we noticed that our wing only consumed up to 3psi per deployment, deeming it a reliable and efficient wing to proceed with for **SPEEDWAY.**
Was today's goal reached: **YES**

11/9/25 Meeting #12 - Redesigning Matchloading Mech

Goal: Redesign the tongue/matchloading mechanism to eliminate minor violations and improve consistency

After competing at the Lake Park competition, we conducted a full performance review of our robot.

One of our biggest issues during the event was receiving three minor violations. These occurred when the 6th block exited the matchloading cylinder due to excessive speed and force from our matchloading mechanism.

This directly impacted match outcomes because:
- We were forced to disable matchloading in later matches
- Our scoring potential decreased
- Driver confidence was reduced
- Strategy had to be adjusted mid-tournament

This revealed that our current tongue mechanism lacked controlled interaction with the matchloading station.

Problem Statement
The existing tongue design (metal plate structure) caused:
- Excessive force transfer
- Poor compliance when entering the matchloading station
- Lack of controlled positioning
- Inconsistent block containment

This led to rule violations and unreliable gameplay.

We determined that future design cycles required:
- A controlled, compliant tongue mechanism
- Reduced direct force on blocks
- Improved alignment when entering the station
- Elimination of minor violations

Design Decision

Instead of using a rigid metal plate, we redesigned the tongue using:
- A rotating polycarbonate piece
- Mounted on a pivot to allow positional adjustment
- Designed to flex slightly under load

Polycarbonate was chosen because:
- It is lightweight
- It has slight flexibility (reduces force spikes)
- It provides smoother contact with blocks
- It reduces risk of launching blocks outward

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

The rotating design allows the tongue to naturally adjust its angle as it enters the matchloading station, reducing aggressive impact forces.

Build Improvements

In addition to material change, we significantly improved build quality:
- Chose better pivot mounting points
- Improved spacing and alignment
- Reduced structural flex
- Secured all fasteners with locking hardware
- Balanced weight distribution to avoid tilting

The rotating polycarbonate plate was mounted using a controlled pivot joint, allowing it to shift position under contact instead of acting as a rigid pushing surface.

Testing Procedure

Success Criteria:
- No 6th block ejection
- Smooth entry into matchloading station
- No minor rule violations
- Consistent 5-block containment
- No structural failure after 20 cycles

Testing Table

Test Metric	Before Redesign (Metal Plate)	After Redesign (Polycarbonate)	Result
6th Block Ejection	Occasional	None Observed	Eliminated
Minor Violations	3 at competition	0 in testing	Eliminated
Entry Smoothness	Aggressive/Forceful	Smooth & Controlled	Improved
Containment Reliability (20 Trials)	16/20	20/20	Improved
Structural Stability	Moderate Flex	Minimal Flex	Improved

Engineering Analysis

The rigid metal plate transferred excessive force directly into the blocks.

The rotating polycarbonate tongue:

- Absorbs impact energy
- Reduces sudden force spikes
- Self-adjusts during entry
- Improves compliance with matchloading station geometry

This redesign transformed the mechanism from rigid and aggressive to controlled and compliant.

This redesign was directly influenced by competition performance. Instead of ignoring minor violations, we treated them as critical engineering feedback.

We learned that:

- Compliance is essential in matchloading mechanisms
- Material choice significantly affects force behavior
- Rule compliance must be engineered into the design

This iteration greatly improved reliability and restored driver confidence.

Was Goal Reached? **YES**

11/10/25 Meeting #13- Code

Goal: Fix the Color Sort + Work on the Tongue Mech

Today, the team was working on installing color sort on the code, we also were working on installing the tongue mech.

To install the tongue mech, the first thing we did was cut a piece of plastic, and attached it to our structure that we already built. This consists of a high strength axle attached to two angles on the edges which will then be screw-jointed to the front of the robot. Using some of the ideas we previously brainstormed as well as the inspiration from 16099A, we were able to create our own.

The color sort was inconsistent, it had issues with the sensor detecting the color but the roller was not spinning outwards fast enough. This caused wither the wrong color ball going in or the entire intake jamming.

The color sort is designed as an **OR Logic Gate**:

- "If the ball is RED OR BLUE, spin the front flex wheel in a certain way."

Although we were able to get it functioning, it still showed inconsistency during testing. The color sensor occasionally detected the ball color correctly, but the roller didn't spin outward fast enough, causing either the wrong-colored ball to be accepted or the entire intake to jam.

Table: Outcomes of Color Sort Test Runs

Ball Color	Sensor Result	Roller Response	Outcome
Red	Correctly detected	Spun outward quickly	Ball Rejected
Red	Inconclusive	Ball going out jammed with ball coming in	Intake Jam
Blue	Correctly detected	Spun outward quickly	Ball Rejected
Blue	Incorrectly Detected	Spun outward too slowly	Wrong Ball Accepted
Red	Incorrectly Detected	Spun inward	Wrong Ball Accepted
Blue	Correctly detected	Spun outward quickly	Ball Rejected

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Tournament Recap #2

Competition 11/21/2025 - 11/23/2025: Speedway Signature Event

Results:
Ranked 63 (1-3-0 on Day 1; 3-1-0 on Day 2)

Immediate Action:
As we were at the Speedway Signature Event, we realized during Day 1 that our outtake stopper wasn't working well, so we tried fixing it, and eventually, we got it fixed right before the last match of the day. This was the start of a 4-game win streak for our team, which boosted our morale high and our rank to the Top 20 teams before the last match of the event. Unfortunately, we lost our last game, which broke our streak and plummeted our rank to 63. This became the end of the road for us, but it was an interesting and incredible event that provided us with a lot of reflection and excitement for the future of our team. After Speedway, the whole team collectively decided on a full rebuild in the mid season, the reasons being elaborated on in the notebook.

Team Reflections:

Mithil (Driver, Designer) - I felt that Speedway was a great learning experience. Understanding the balance between defensive and offensive plays was a crucial learning element. One thing I was proud of was the strategy making skills our team possessed when allianceing with other teams, and by sticking with these strategies we were able to guide ourselves to victory. I believe that a faster scoring, better de-score arm, and a better match loading mechanism would have helped us reach an even higher ranking.

Naisha (Mechanic) - I enjoyed the speedway atmosphere a lot, I got to see many new robot ideas and am looking forward to implementing some of these ideas on our new robot!

Sathvik (Driver, Mechanic) - Speedway was such a thrilling experience, but also was ultimately a successful competition for our design iteration as even though it wasn't the strongest bot, with some great driving and some great alliances, we ended up .500 at what is ultimately a competition that is regarded harder than a worlds division tournament. I'm extremely proud but am looking forward to rebuild.

Shaurya (Mechanic) - I personally was extremely proud of our performance at Speedway, because we made a 40 rank comeback at the end of the day from Day 1 to Day 2, and also because after Match 7 we were ranked in the Top 20 teams, which unfortunately didn't last when we lost the last match. I am still extremely proud of our performance and all of the strategic and technical aspects of robotics that we learned at Speedway, and it truly was a one-of-a-kind event to attend.

Shreyas (Mechanic) - The competition was a very exciting experience, and showed us how some of the best teams in the entire world built and drove their robot along with their game strategy. The first day wasn't as well with mediocre alliances, but the second day we had really good alliances which helped us win.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Sonit (Programmer) - From a coding perspective, Speedway revealed that our autonomous routines were too 'rigid.' On Day 2, I had to make quick adjustments to our PID constants to account for the specific friction of the event tiles. My goal for the rebuild is to create more adaptable code that can be tweaked in seconds between matches.

12/01/25 - final Thoughts on Design Iteration 1

Strengths:

- Communication with teammates
- Driver Strategy with weaker alliances

Weaknesses:

- Overall build quality
- Lack of driver practice
- Lack of consistency and reliability
- Bad tuning
- Lack of capacity.

Areas of priority for next design:

- Quick design that is reliable and visually appealing
- Increased driver practice
- Reliable wing mech
- Increased Capacity

Early Season Programming Explanation:

Program

Note to judges: Some content in this programming section shows repetition with what is found in our programming notebook, linked here. This is a documentation of our early season Lemlib library, and we switched to a custom template where we coded everything ourselves after early season. Check out all of those

System Overview & Philosophy

Objective:

The primary objective of our programming for the "Push Back" season is to implement a robust positioning system that allows the robot to navigate the field using coordinate geometry rather than time-based dead reckoning. Our philosophy is **"Predictability over Raw Speed."** A robot that moves at 90% speed but hits its target 100% of the time is infinitely more valuable than a robot that moves at 100% speed but misses 20% of the time.

To achieve this, we moved away from standard time-based programming and adopted a **State-Space approach** utilizing Odometry for localization and PID (Proportional-Integral-Derivative) for closed-loop control.

The Problem with "Time-Based" Movement:

In standard drive code (e.g., drive_forward(2000ms)), the robot assumes that applying voltage for a set time results in a set distance. However, we identified three critical failure modes:

1. **Battery Voltage Variance:** As the battery drains from 100% to 80%, the voltage supplied to the motors drops. 2000ms at 12V moves the robot significantly further than 2000ms at 10V.
2. **Field Friction:** VEX foam tiles vary in friction. Old tiles are slippery; new tiles are grippy. Time-based code cannot adapt to these conditions.
3. **Defense & Slippage:** If the robot pushes a block or gets bumped by an opponent, the wheels spin but the robot doesn't move. The code has no way of knowing it hasn't reached the target.

The Solution: Odometry

Odometry is the practice of using sensors to track the robot's position relative to a starting point. We define our field as a Cartesian plane where the starting point is (0,0). By tracking the robot's physical location (X, Y, θ), we can command the robot to "Drive to point (24, 24)" regardless of battery level or field friction.

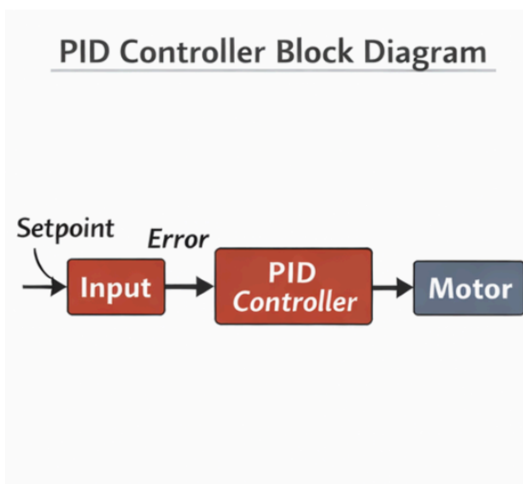
Hardware Architecture

To implement Odometry, we utilize specific sensors designed to bypass the mechanical backlash of the drivetrain.

- **Tracking Wheels ("Dead Wheels"):**
  - We utilize unpowered omni-wheels tensioned to the ground using a custom hinge mechanism.
  - *Why?* When the robot accelerates rapidly, the powered drive wheels may slip. Tracking wheels are independent; they only spin if the robot actually moves across the tiles.
- **Sensors:**
  - **V5 Rotation Sensors:** These are keyed to the tracking wheels to measure rotation with high precision (0.08 degrees per tick).
  - **Inertial Sensor (IMU):** We use the V5 Inertial Sensor to track our absolute heading (θ). While tracking wheels can calculate rotation, the IMU does not suffer from "scrub" (lateral friction) errors during turns.

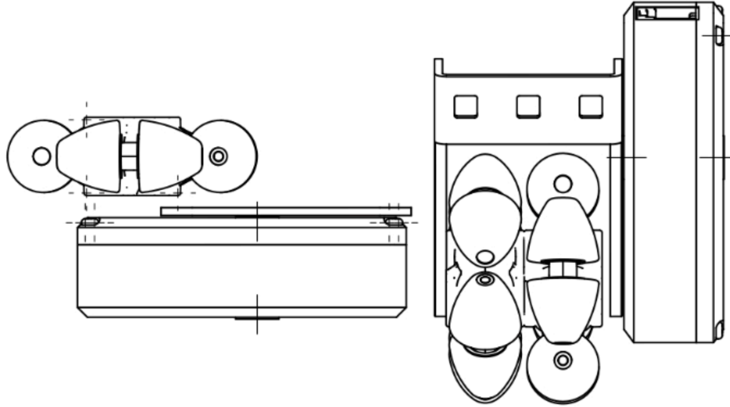
Signal Flow Diagram:

The below diagram explains the flow of signals, starting from our initial input in our autonomous code, and ending with rotation of a motor or piston.



Sensor Polling Rate: The V5 Brain updates motor ports every 10ms (100Hz). To ensure our math is accurate, our Odometry task runs in a dedicated thread at **priority level 15**, ensuring it is never interrupted by the driver control loop.

Below is a diagram of a Rotation Sensor with the “dead wheels” attached:



And below is an image of an IMU sensor:

Now that we have identified the sensors we used to create our autonomous routines, we will discuss the mathematics, physics, and more behind our programming.

The Mathematics of Odometry:

The core of our tracking system is converting raw wheel rotation into (X, Y) coordinates. This process happens in three distinct steps: **Data Acquisition**, **Local Displacement**, and **Global Transformation**.

Step 1: Calculating Distance (ΔS)

First, we convert the raw sensor data (ticks) into linear distance traveled. We must know the physical circumference of our tracking wheels.

- **Wheel Diameter:** 2.75 inches
- **Ticks Per Rev:** 360

$$\Delta S = (\text{Ticks}_{\text{current}} - \text{Ticks}_{\text{previous}}) \times \frac{\pi \times \text{Diameter}}{\text{Ticks per Revolution}}$$

This gives us the change in distance for the Left (**ΔL**) and Right (**ΔL**) sensors.

Technical Specification: Using a 2.75" wheel with a 360-tick encoder provides a resolution of approximately **0.024 inches per tick**, allowing for sub-inch precision during high-speed maneuvers.

Step 2: Calculating Heading (θ)

We prioritize the IMU for heading data because it is absolute.

We update our global heading using the IMU or the differential between left and right tracking wheels. This can be represented by:

$\theta_new = \theta_old + \boldsymbol{\Delta\theta}$

While we primarily rely on the V5 Inertial Measurement Unit (IMU), we calculate a secondary heading based on the differential between the Left (**Δ**L) and Right (**Δ**R) tracking wheels to cross-reference for "drift." This approach is known as **Sensor Fusion**.

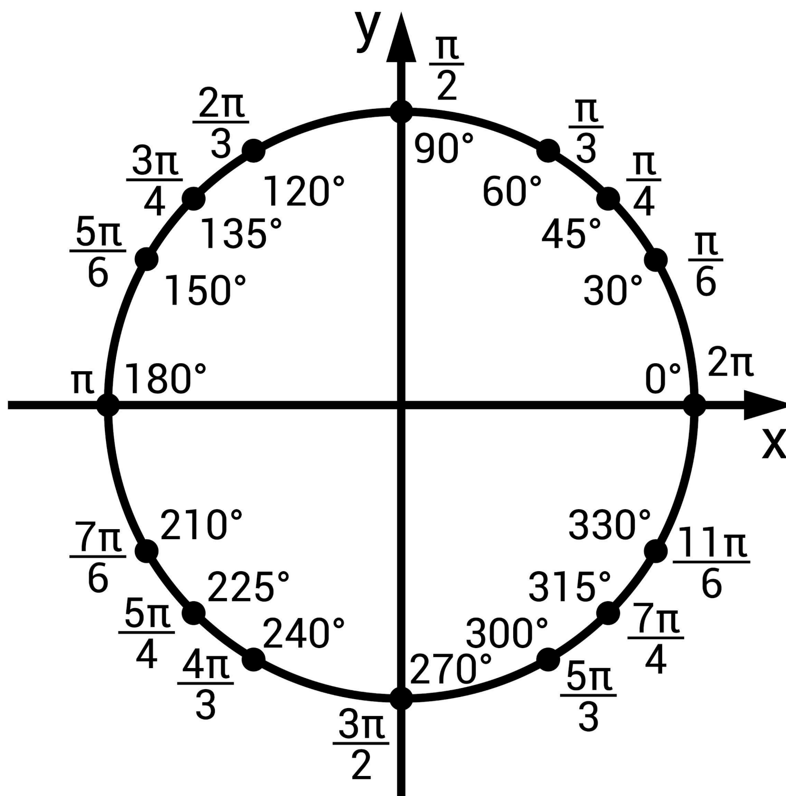
The change in heading ($\boldsymbol{\Delta\theta}$) is calculated as:

$$\Delta\theta = \frac{\Delta L - \Delta R}{S_L + S_R}$$

Where S_L and S_R are the distances from the tracking wheels to the center of the robot.

The new global heading is then updated.

The below unit circle displays some common theta values for heading:



The Mathematics of Odometry, continued

Local Displacement Vector:

At this stage, we know how much the wheels turned, but we need to know how the robot moved *relative to its own center*.

The following are three ways of identifying the robot's movement relative to its own center:

- If **Δ**L = **Δ**R, The robot moved in a perfectly straight line
- If **Δ**L = -**Δ**R, The robot rotated about its center point
- If **Δ**L ≠ **Δ**R. The robot is simultaneously translating and rotating (moving in an **Arc**)

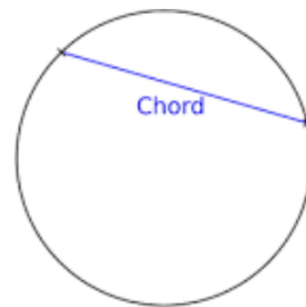
Because the majority of VEX movements are arcs, our math must prioritize calculating these curved paths to avoid "positional drift."

The "Chord" Approximation:

Calculating a true arc length using calculus in real-time can be computationally expensive. However, because our sensor loop refresh rate is so high (**100Hz / 10ms**), the arc traveled in a single cycle is microscopic.

We can therefore treat the arc as a **Chord** (a straight line connecting two points on a curve). This simplification maintains high accuracy while ensuring the V5 Brain can process the logic without lag.

To the right is a diagram of a **Chord** for visual interpretation:



Thus, the below formula, where **Δ**Local_Y represents the magnitude of the movement vector of the robot, allowing us to compute the latest **"chord"** that the robot traversed.

$$\Delta \text{Local_Y} = \frac{\Delta L + \Delta R}{2}$$

Note on Lateral Movement: If a horizontal tracking wheel is present, we calculate **Δ** Local_Y to account for "drift" or "sideways slip" caused by collisions or high-speed turns.

Orientation for Integration:

When the robot moves and turns simultaneously, it travels along an arc. If we used only the starting angle (θ_old), we would consistently undershoot the curve; if we used the ending angle (θ_new), we would overshoot it.

To solve this, we utilize the **Average Orientation (alpha (α))**. By calculating the angle at the midpoint of the 10ms movement window, we can approximate the tangent of the arc, turning a complex curve into a manageable linear vector.

The below formula is how we calculate alpha:

$$\alpha = \theta_{old} + \frac{\Delta\theta}{2}$$

The Mathematics of Odometry - Part 3

Global Transformation:

We now rotate the Local Displacement Vector by the Average Orientation (α) using standard trigonometry. This maps the robot-centric movement onto the field-centric grid.

Global Displacement components:

$$\Delta X = \Delta \text{Local_Y} \times \cos(\alpha)$$

$$\Delta Y = \Delta \text{Local_Y} \times \sin(\alpha)$$

Integration (Accumulation):

Because our tracking is continuous, we must "accumulate" these infinitesimal changes into our global position variables. This process is known as **Discrete Integration**. By summing these small steps every 10ms, we maintain a real-time "State Vector" of the robot's position.

Updated Global State:

$$X_{global} = X_{global} + \Delta X$$

$$Y_{global} = Y_{global} + \Delta Y$$

This calculation runs in a background task every 10ms, constantly integrating these tiny changes to update our global position vector [x, y, θ].

PID Control Theory:

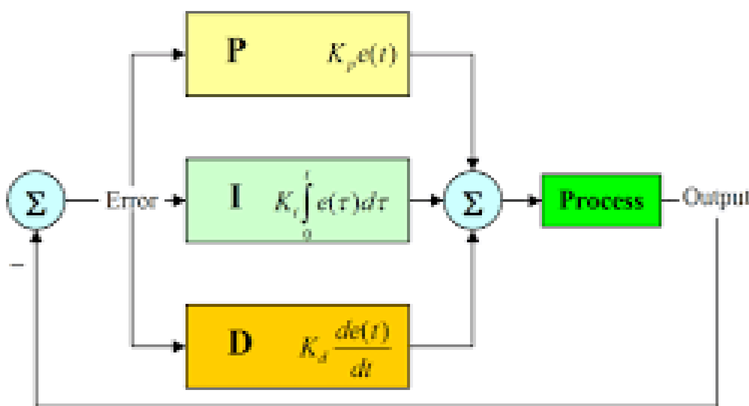
Objective:

Odometry tells us *where* we are. PID tells the motors *how* to get to where we want to be. The objective is to create smooth, accurate motion that corrects itself if disturbed by a block or an opponent.

Definition:

PID (Proportional-Integral-Derivative) is a closed-loop control feedback mechanism. It calculates an "error" value as the difference between a measured process variable (current position) and a desired setpoint (target position).

PID Control Diagram:



Mehta, Nikunj & Chauhan, Dharmendra & Patel, Sagarkumar & Mistry, Siddharth. (2017). Design of HMI Based on PID Control of Temperature. International Journal of Engineering Research and. V6. 10.17577/IJERTV6IS050074.

The Closed-Loop Feedback Concept:

A "Closed-Loop" system is one that constantly checks its own work. PID calculates the **Error**—the difference between where we are and where we want to be—and adjusts power in real-time to eliminate that gap.

The Error Function:

- **For Linear Movement:** $\text{Error} = \sqrt{(x_t - x_c)^2 + (y_t - y_c)^2}$ (Target minus Current)
- **For Rotational Movement:** $\text{Error} = \text{Target Angle} - \text{Current Angle}$

PID Terms - The Proportional (P)

We view PID as a way to look at the **Past**, **Present**, and **Future** of our robot's movement.

Concept: "The Present":

The P-term reacts to the current state of the robot. It provides the "raw power" needed to move toward a target.

Equation: $P_{\text{out}} = kP \times \text{Error}$

System Behavior:

- **Large Error:** When the robot is far from the target, P_out is large, causing the robot to drive fast.
- **Small Error:** As the robot gets closer, Error decreases, so P_out decreases, causing the robot to slow down.

Tuning Impact:

- If kP is too low: The robot will be sluggish and may not reach the target.
- If kP is too high: The robot will react too violently, overshoot the target, and potentially oscillate.

We will discuss our tuning process in more depth later, in a specific section of our notebook.

PID Terms - The Integral (I)

Concept: "The Past"

The Integral term accounts for accumulated error over time. It is used to fix "Steady-State Error."

Equation: $I_{\text{out}} = kI \times \sum(\text{Error} \times dt)$

Overcoming Static Friction:

If the robot stalls 0.5 inches from the target because the P-term isn't strong enough to overcome friction, the I-term begins to accumulate that "stuck" error. Eventually, the sum becomes large enough to give the motors the extra "nudge" needed to reach the target.

Safety Constraint: Integral Windup:

To prevent the robot from accelerating uncontrollably if it is physically blocked, we implement an **Integral Zone**. The I-term only begins accumulating when the robot is within 2 inches of its target.

PID Terms - The Derivative (D)

Concept: "The Future"

The Derivative term measures the *rate of change* of the error. It predicts where the robot will be in the next instant.

Equation: $D_{\text{out}} = kD \times \frac{\Delta\text{Error}}{\Delta t}$

.

Momentum Control: Our robot has high mass and significant momentum. The D-term detects when the robot is approaching the target too quickly and applies a counter-force, preventing overshooting and violent oscillations.

Importance for Heavy Robots:

Our robot is heavy. Once it gets moving, it has high momentum. The D-term is critical to prevent it from crashing through the target distance.

PID Tuning Strategy

Tuning is the process of finding the perfect "Weights" (kP, kI, kD) for our robot. Since friction varies by surface, we perform all tuning on official competition foam tiles.

The Tuning Process:

1. **Isolate Proportional:** Increase kP until the robot reaches the target but begins to "hunt" or oscillate back and forth.
2. **Add Damping:** Increase kD to "smooth out" the oscillations until the robot settles at the target in a single, fluid motion.
3. **Final Polish:** If the robot consistently stops 0.25" short, we introduce a very small kI to pull it to the final setpoint.

Final Tuning Values:

Tuning Log (Lateral Drive)

We performed iterative testing to arrive at our final constants.

Iteration	kP	kI	kD	Result
1	0.5	0	0	Too slow. Didn't reach the target.
2	1.5	0	0	Fast, but violent oscillation at the end.
3	1.5	0	0.5	Better, but still some overshoot.
4	1.0	0	0.5	Good speed, slight overshoot.
5	**1.0**	**0**	**0.8**	**Perfect.** Stops on a dime.

Note: We chose not to tune kI because it does not provide much usefulness.

Current Gains:

- **Lateral (Drive Straight):**

    - kP = 1.0: High enough for speed.
    - kD = 0.8: High damping required to stop the heavy chassis.
- **Turn (Rotate):**
    - kP = 1.0: Provides necessary torque to scrub wheels.
    - kD = 0.8: Prevents "whipping" past the target angle.

Mathematical Summary:

To help visualize how these three terms combine to create our final motor output (12.0V Max), we use the following summation:

$$\text{Output}_{total} = (kP \times E) + (kI \times \int E\,dt) + (kD \times \frac{dE}{dt})$$

Sensor Integration - The Optical System

Objective: Autonomous Block Sorting:

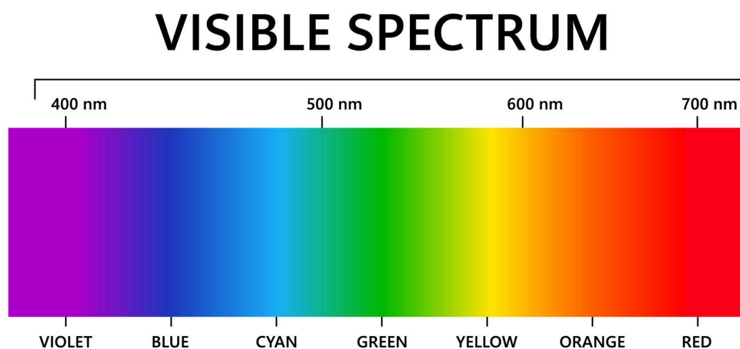
In the *Push Back* game environment, precision is paramount. Accidentally scoring an opponent's block results in severe penalties. Our objective was to create an **Automated Sorting System** that reduces driver cognitive load by programmatically identifying and ejecting "enemy" game elements.

HSV Color Space:

Data Normalization: HSV vs. RGB:

The V5 Optical Sensor provides raw RGB data, but lighting conditions on competition stages are notoriously inconsistent. To ensure reliability, we convert raw data into the **HSV (Hue, Saturation, Value)** color space.

- The Logic: Unlike RGB, where a shadow can change all three values, the Hue remains relatively constant regardless of brightness or shadows. This makes our detection algorithm robust under various stadium lighting conditions.



Essentially, we are converting the data from the optical sensor, which is the period of the waves of reflected light from the color. The above photo represents the visible spectrum of

light, but we are only dealing with the blue and the red portions of the visible spectrum, because these are the colors of the blocks in the “Push Back” game.

Color Definitions:

- Red Blocks: Hue 0 degrees to 20 degrees, as well as 340 degrees to 360 degrees
- Blue Blocks: Hue 200 degrees to 260 degrees

Proximity Check:

To prevent the sensor from accidentally "seeing" blocks outside the robot, we use a **Proximity Check**. The sorting logic only triggers if the **Proximity Value is > 200**, confirming the block is actually deep inside the intake.

Finite State Machine (Sorting Logic):

We managed our sorting logic using a **Finite State Machine (FSM)**. This ensures the robot transitions smoothly between identifying a block and ejecting it without getting "stuck" in a loop.

Sorting Execution (Blue Alliance Example):

1. **Idle:** The intake spins forward normally.
2. **Detection:** The sensor identifies a Red (Enemy) Hue and high proximity.
3. **Action:** The code instantly reverses the intake motors at full voltage (-12.0V).
4. **Recovery:** The motors stay in reverse for 500ms to ensure the block is cleared before returning to the Idle state.

Signal Hysteresis:

To avoid "sensor noise" (short flashes of color), we require the sensor to see the enemy color for **3 consecutive frames (30ms)** before triggering an ejection. This acts as a filter to ensure every ejection is intentional.

Teleoperated Control (Driver UX):

Objective:

Provide intuitive control that bridges the gap between human input and robot output.

Split Arcade Control:

We utilize an "Arcade Drive" control scheme:

- **Left Stick:** Controls the lateral movement for the drivetrain
- **Right Stick:** Controls turning

Subsystem Automation:

Instead of requiring the driver to manually reverse the intake for sorting, the code handles it automatically.

- **L1 Button:** Toggles "Intake Mode" (Forward).
- **L2 Button:** Toggles "Outtake Mode" (Reverse).
  The driver simply sets the state, and the robot manages the motors.

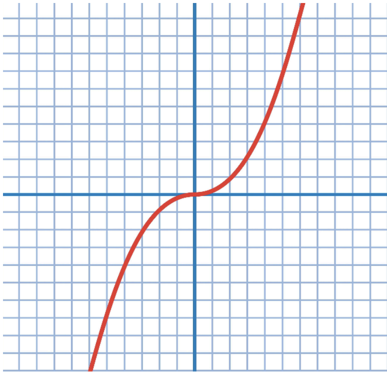
Input Filtering (Cubic Control):

Linearity Problem:

A standard joystick map is linear: 50% stick = 50% power. This makes small adjustments (like aiming at a goal) very difficult, as the robot moves too fast.

The Solution:

We applied a **Cubic Function** (diagram below) to our inputs.



The Effect:

- **Input 0.2 (20%):** 0.2^3 = 0.008 (0.8% Power). Ultra-fine precision.
- **Input 0.5 (50%):** 0.5^3 = 0.125 (12.5% Power). Soft control for maneuvering.
- **Input 1.0 (100%):** 1.0^3 = 1.0 (100% Power). Full speed available instantly.

This "Exponential Curve" creates a soft center for precision aiming while retaining maximum power at the edges.

Annotated Code - Main.cpp

```
#include "main.h"
#include "lemlib/api.hpp"
```

```
/* --- 1. MOTOR GROUP CONFIGURATION --- */
// We use a 6-motor drive for maximum pushing power.
// Negative numbers (e.g., -8) mean the motor is physically reversed on the
robot.
// Blue gearsets = 600 RPM for high speed.
pros::MotorGroup leftMotors({-8, 9, -10}, pros::MotorGearset::blue);
pros::MotorGroup rightMotors({16, -17, 18}, pros::MotorGearset::blue);

/* --- 2. DRIVETRAIN PARAMETERS --- */
// Configures the physical dimensions: 12" track width, 3.25" wheels, 450
actual RPM.
lemlib::Drivetrain drivetrain(&leftMotors, &rightMotors, 12,
lemlib::Omniwheel::NEW_325, 450, 2);

/* --- 3. SENSOR DEFINITIONS --- */
pros::Imu imu(10); // Inertial Measurement Unit for precise heading
pros::Rotation vertical_rotation_sensor(20); // Tracks forward/backward
movement
pros::Rotation horizontal_rotation_sensor(19); // Tracks sideways drift (scrub)

/* --- 4. PID CONTROLLER SETTINGS --- */
// Lateral PID: Handles forward/backward movement accuracy.
lemlib::ControllerSettings lateral_controller(
    10,  // kP: Raw power. The "aggression" of the movement.
    0,   // kI: Corrects steady-state error (friction).
    3,   // kD: The "Damper." Prevents the robot from overshooting.
    3,   // Anti-windup for the I-term.
    1,   // Small error range (inches).
    100, // Timeout for small error.
    3,   // Large error range (inches).
    500, // Timeout for large error.
    20   // Slew rate: Prevents the robot from "popping wheelies" by capping
acceleration.
);

// Angular PID: Handles turning accuracy.
lemlib::ControllerSettings angular_controller(
    2,   // kP: Lower than lateral because turning requires less torque.
    0,   // kI
    10,  // kD: High damping to ensure the turn settles instantly without
"wobbling."
    3, 1, 100, 3, 500, 0
);

/* --- 5. ODOMETRY & CHASSIS SETUP --- */
```

```
// Defines the physical offsets of our tracking wheels from the center of the
robot.
lemlib::TrackingWheel verticalWheel(&vertical_rotation_sensor,
lemlib::Omniwheel::NEW_2, -0.5);
lemlib::TrackingWheel horizontalWheel(&horizontal_rotation_sensor,
lemlib::Omniwheel::NEW_275, 0.5);
lemlib::OdomSensors odometry(&verticalWheel, nullptr, &horizontalWheel,
nullptr, &imu);
lemlib::Chassis chassis(drivetrain, lateral_controller, angular_controller,
odometry);

/* --- 6. SUBSYSTEMS --- */
pros::Motor intake(-18, pros::MotorGearset::blue); // High-speed intake
pros::Motor score(-17, pros::MotorGearset::blue);  // Scoring mechanism motor

// Pneumatics: Using air for tasks that don't need motor power (Saving the 88W
budget!)
pros::adi::Pneumatics intakePiston('A', false); // Shifts intake geometry
pros::adi::Pneumatics tongue('B', false);       // Deployment for our
matchloader
pros::adi::Pneumatics wing('C', false);         // Side wing for defensive
descoring

/* --- 7. INITIALIZATION --- */
void initialize() {
    pros::lcd::initialize();
    pros::lcd::set_text(1, "System Online: Push Back Edition");
}

void competition_initialize() {
    // Calibrating sensors at the start of the match to ensure (0,0,0) is
accurate.
    horizontal_rotation_sensor.reset_position();
    vertical_rotation_sensor.reset_position();
    imu.reset();
    chassis.setPose(0, 0, 0); // Sets our starting coordinates on the field map
}

/* --- 8. AUTONOMOUS --- */
void autonomous() {
      // 7 balls
     intake.move(127);
     score.move(127;
     chassis.setPose(0,0,180);
     chassis.turnToPoint(6,20.6,1000);
     chassis.moveToPoint(7,20,1000);
```

```
    chassis.waitUntilDone();
    tongue.extend();
    pros::delay(500);
    chassis.turnToPoint(32,18.5,1000);
    chassis.moveToPoint(32,18.5,1000);
    chassis.turnToPoint(43.5,-10,1000);
    chassis.moveToPoint(44.2,-10,1000);
    chassis.moveToPoint(42.5,45,3500);
    chassis.waitUntil(30);
    hood.retract();
    pros::delay(1000);
    chassis.moveToPoint(45,27,1000);
    chassis.turnToPoint(60,45,1000);
    chassis.moveToPoint(61,45,800);
    wing.extend();
    chassis.turnToHeading(0,800);
    chassis.moveToPoint(54.4,62.3,1500);
}

/* --- 9. DRIVER CONTROL (OPCONTROL) --- */
void opcontrol() {
    pros::Controller controller(pros::E_CONTROLLER_MASTER);
    chassis.setBrakeMode(pros::E_MOTOR_BRAKE_COAST); // Allows for smoother
movement

    while (true) {
        // Tank Drive: Left stick controls left side, right stick controls
right side.
        chassis.tank(controller.get_analog(pros::E_CONTROLLER_ANALOG_LEFT_Y),
                     controller.get_analog(pros::E_CONTROLLER_ANALOG_RIGHT_Y));

        /* --- SUBSYSTEM LOGIC --- */
        if (controller.get_digital(pros::E_CONTROLLER_DIGITAL_L1)) {
            // Standard Intake Mode
            intakePiston.retract();
            intake.move(127); // Full power
            score.move_absolute(0.0, 600); // Ensures score motor is reset
        }
        else if (controller.get_digital(pros::E_CONTROLLER_DIGITAL_L2)) {
            // Outtake/Eject Mode
            intake.move(-100);
        }
        else if (controller.get_digital(pros::E_CONTROLLER_DIGITAL_R2)) {
            // High Goal Scoring: Combined power of intake and score motors
            intakePiston.retract();
            intake.move(127);
```

```
            score.move(127);
        }
        else if (controller.get_digital(pros::E_CONTROLLER_DIGITAL_B)) {
            // Middle Goal Scoring: Changes geometry via piston extension
            intakePiston.extend();
            intake.move(127);
            score.move(127);
        }
        else {
            // Stop and Reset to prevent motor burnout
            intake.move(0);
            score.move(0);
            score.tare_position();
        }

        // Defensive Wing Logic: Hold R1 to keep the wing out
        if (controller.get_digital(pros::E_CONTROLLER_DIGITAL_R1))
wing.extend();
        else wing.retract();

        // Tongue Toggle Logic: Taps X to switch between Extended and Retracted
        if (controller.get_digital(pros::E_CONTROLLER_DIGITAL_X)) {
            if (tongue.is_extended()) tongue.retract();
            else tongue.extend();
            pros::delay(250); // Debounce delay to prevent rapid toggling
        }

        pros::delay(20); // Standard loop delay for V5 Brain stability
    }
}
```

Conclusion & Summary:

System Reliability Analysis:

The transition from time-based code to Coordinate-Based Odometry has revolutionized our consistency. In testing, our **"Drive to Point"** functions are accurate to within 0.5 inches over a 12-foot run, regardless of battery voltage.

Key Achievements:

1. **Odometry:** Successfully tracks position (X,Y) using dead wheels and IMU fusion.
2. **PID Tuning:** Achieved critical damping with specific gains (kP=1.0, kD=0.8), eliminating oscillation while maintaining speed.
3. **Automation:** The Cubic Drive and Auto-Sorters significantly reduce the driver's cognitive load, allowing them to focus on game strategy rather than basic mechanics.

Future Improvements:

- **Pure Pursuit:** We plan to implement "Pure Pursuit" algorithms to follow curved paths rather than just straight lines ("Turn-then-Move").
- **Motion Profiling:** We aim to implement trapezoidal motion profiles to further smooth out acceleration and protect our gearboxes.
- **Custom Library:** Although we currently utilize the LemLib library for our odometry and PID, we wish to, in the future, implement our own custom library in order to make our autonomous routines more precise.

Final Statement

This software architecture provides a solid, mathematical foundation for our robot's performance, ensuring that our hardware's potential is fully realized on the competition field. However, we still have several things to add to our programming to further improve our autonomous routines as the season progresses.

Mid-Season Rebuild

12/18/25 - v2 Design Brief

Identify

Game Constraints:

- Robot must be able to fit in an 18” by 18” by 18” cube at the start of the match
- Total combination of motors must not exceed 88 watts
- Must be built only from VEX-approved parts
- Custom plastic is limited to twelve pieces, each piece fitting within a 4” x 8” area
- **Bot must be built + programmed in two weeks**

Personal Goals and Strategy:

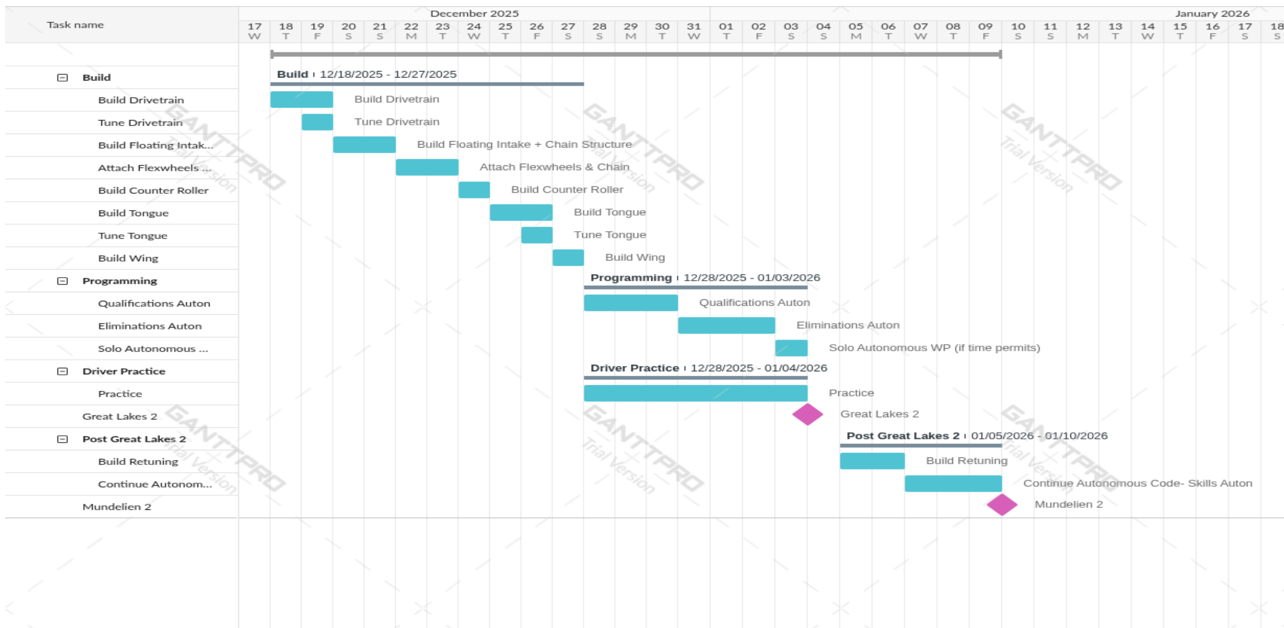
- Score more than 7 blocks in 3 scoring zones during autonomous period
- Strategize with alliance team to get the autonomous bonus and win point
- Work with alliance team to score as many points as possible
- **Control all Control Zones**
- Descore opponent scoring zones before end of match

Robot Subsystems:

- Drivetrain
- Intake mechanism to load blocks into the robot
- Scoring mechanism to score blocks into goals
- Mechanism to remove out blocks from a scoring zone
- Mechanism to smoothly collect match loads from match loading zones

Gantt Chart

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke



12/20/25 - v2 Intake Criteria

Identify

Problem Statement: Design an intake to interact with blocks by collecting and storing them within the mechanism. Intake should at least be able to transport a block from ground to goal to enable scoring opportunities.

Solution Constraints:

- Must work efficiently on a 22W maximum
- Must not extend past the 18” size constraint on all three dimensions

Solution Goals:

Maximum power consumption for intake must not exceed 1.0W.

- Given that most of our intake will be linked by chain, the motors will most likely be working harder to power the entire mechanism. However, the amount of power needed to work the mechanism must remain low so that the intake works efficiently and doesn’t overheat the motor.

Intake must be able to score on long goals and both center goals (medium + low heights)

- Being able to utilize all types of scoring methods could prove handy in a close match where long goals are filled to the maximum. Being able to score on both of the center goals could give both our team and our alliance an advantage over others.

Intake should be able to possess around 6-7 blocks within its mechanism

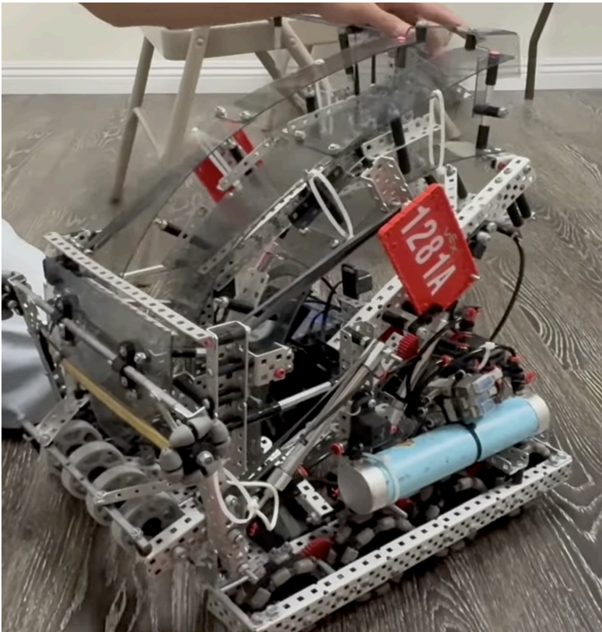
- Being able to possess multiple blocks at once will make it effortless to create splash plays of point gain, rather than possessing around 3 blocks at a time and having to go back to retrieve 5 more blocks in another rotation.
- Additionally, 6 blocks is the number of block storage needed to clear out a full matchloader, making it easier on the driver for skills.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

12/22/25 - Lever Bot Brainstorm

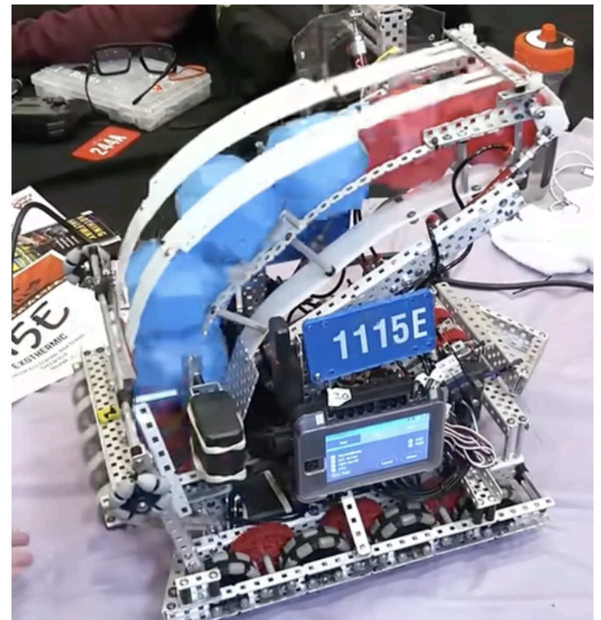
Brainstorm

__Overview:__ An intriguing design was released earlier in November, called the Lever robot, or the Mailbox robot. Multiple high-level teams have built this design as of today, and it could be a potential build for our winter rebuild



Lever Intake (1281A, Youtube)

An **"Lever"** intake design is one where a ball travels upwards and horizontally, being pushed by a c channel powered by a motor.



Lever Intake (1115E Pits and Parts Interview)

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Strengths: The intake is able to score blocks at the fastest pace compared to your standard Ruiguan/S-Bot/C-Bot/Basket. To score on the high goal and low goal, you just need a piston to set the state of your outtake height.
Weaknesses: Necessary to create an arm powered off of a motor that pushes blocks constantly, which is prone to damage and overheating. Additionally, a strategy disadvantage is that you are forced to score all blocks that you hold in your capacity instead of scoring only 1-2 blocks.

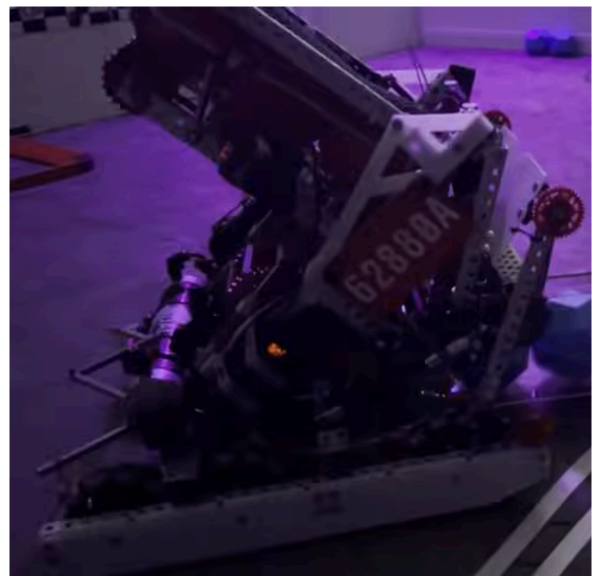
12/22/25 - Undergoal S-Bot Brainstorm

Brainstorm

Overview: Undergoal designs have potential to be a great defensive force in push back, especially when many drivers at the regional level struggle to score and de-score against heavy defense. Here, we take a look at a unique undergoal S-Bot design.

Undergoal S Intake (8995R Raiders)

Similar to how a lever bot works, the outtake height is set by two pistons that extend and retract to score on high goals and middle goals, respectively.



Undergoal S Intake (62880A, Youtube)

Strengths: The robot's ability to go under goal gives significantly improved mobility. This is especially useful when in need to drive away from opposition.

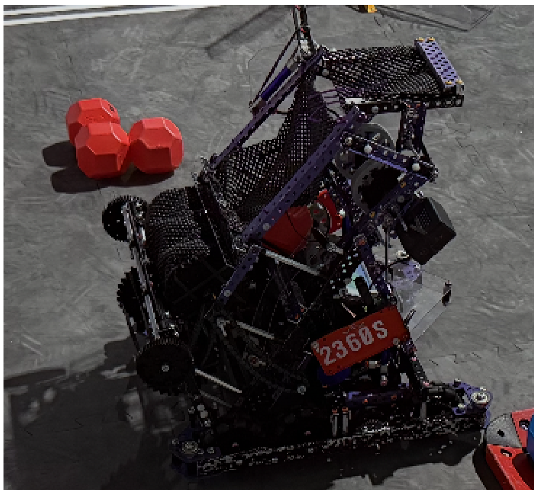
Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Weaknesses: Robot has not been built often, only a few teams have built this robot which poses concerns about its feasibility with the current meta in regional-level matches and world-level matches. Additionally, motor distribution on this robot would be very unique, and it is not worth having a 5.5W motor powering your scoring stage.

12/22/25 - Undergoal Ruiguan Brainstorm

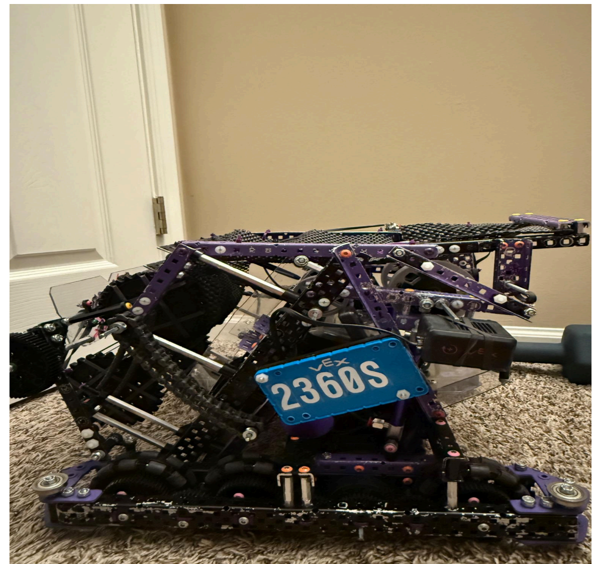
Brainstorm

Overview: Very similarly to the Undergoal S-Bot, many Undergoal designs have potential to be a great defensive force in push back, especially when many drivers at the regional level struggle to score and de-score against heavy defense and also attack because of its ability to quickly escape.



Undergoal Ruiguan Intake upper state (2360S Singularity)

Similar to how a lever bot and other undergoal bots works, the outtake height is set by two pistons that extend and retract to score on high goals and middle goals, respectively.



Undergoal Ruiguan bot & Intake lower state (2360S Singularity)

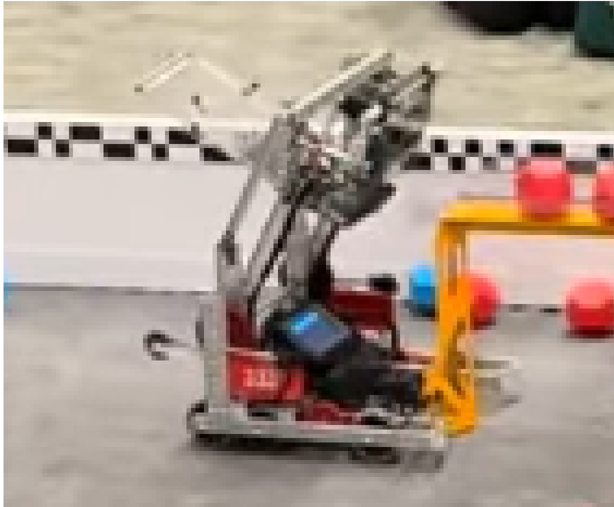
Strengths: The robot’s ability to go under goal gives significantly improved mobility. This is especially useful when in need to drive away from opposition to score or evade enemy robots.

Weaknesses: Robot has not been built often, only a few teams have built this robot which poses concerns about its feasibility with the current meta in regional-level matches and world-level matches. Additionally, its capacity isn’t as good as some other meta bots

12/22/25 - Tray Bot Brainstorm

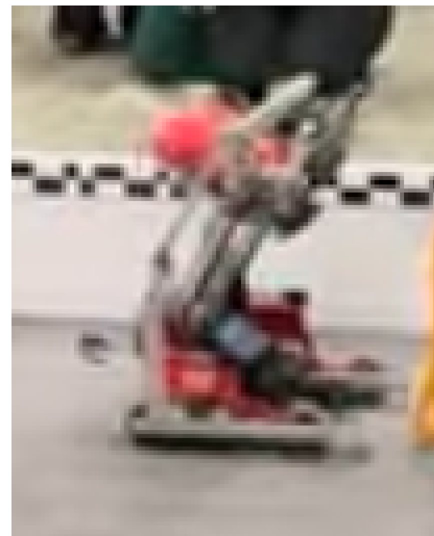
Brainstorm

Overview: Characterized by 333Y who won the Niles tournament, the Tray Bot has recently begun to pick up some steam though it may not be meta quite yet it is a new intriguing option for our rebuild.



333Y Tray Bot scoring (333Y during competition)

A tray bot is like a hybrid undergoal basket bot compromise with its "tray" used like a mobile basket.



333Y Tray bot (333Y Youtube Niles Finals)

Strengths: It has a decent scoring ability in open space as well as being able to escape quickly and go under the long goals. It can defend and move quickly as well with a good storage capacity.

Weaknesses: Not currently a meta bot at the moment which isn't always a bad thing. But it also is quite easy to defend as it does take some time to load up and score and is obvious when scoring as it has to bring up its tray which can be easily punished. It is also a more complex build to get right.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

12/22/25 - Intake Selection & Design Plan

Select

Decision Chart for Intake Design:

	Speed	Ball Capacity	Time needed	Popularity	Total
S-Shaped Robot	2	7	5	7	21
Ruiguan Intake Chain	7	4	**8**	**8**	**27**
Undergoal Ruiguan	5	2	3	4	14
Hopper Bot	3	**8**	6	6	23
Band Rollers Ruiguan	6	3	7	5	21
Tray Bot	4	1	4	2	11
Undergoal S	1	5	2	1	9
Lever Bot	**8**	6	1	3	18

How we gave our point values: Each layout in consideration was ranked based on the four aforementioned categories, with 8 being the highest and 1 being the lowest. Our selection was based on which intake had the highest value.

How we created the categories: Speed is an important factor because the ball has to be scored fast. Ball capacity is important because it's better to be able to score around 6-7 blocks fast rather than having to score 3 at a time. The time needed to build the mechanism is an important factor as the time our team meets to build is limited given our conflicting schedules. Finally, popularity is an important factor to secure top-seeded alliances, ultimately contributing to the team's success in competitions.

Decision: Our final decision for Intake Layout is an S-Shaped intake. The benefits of this are that we have a higher ball-holding capacity and we have the capability to score on the middle goals.

Drawbacks: Although the potential of the s-shaped robot is strong, teams tend to prefer a Ruiguan intake as it's specifically designed for quick de-scoring opportunities. This could have a slight effect in terms of alliance selections, however it shouldn't be an issue if we perform with excellence.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

12/23/25 -Meeting #1 Build Drivetrain Structure

Build

Goal:

Today, the team started working on the drivetrain for the new bot.

There were a few things that we had to keep in mind, flaws that we had in our old bot:

1. The distance between the channels where the wheels were placed were 3 holes, this hindered us from being able to put all the necessary elements needed for a structural drivetrain.
2. The drivetrain was 25 holes wide, this caused a lot of jamming, because it could not fit two balls coming in at the same time.
3. We had a high-strength axle as our cross brace under the bot, this was a problem when we tried to park as that axle blocked our rear wheels from crossing the parking barrier.

To counteract these problems we decided to implement these solutions through design:

1. We made the distance between the channels 4 holes, this allowed us to make sure that all the necessary elements (e.g. shaft collars) were easy to fit in.
2. The drivetrain was changed to 27 holes wide, now we are easily able to fit two balls side-by-side and prevent jamming.
3. We also changed the high strength axle to a L-Channel which sits above the wheels. This allows the wheels to come in contact with the parking barrier first, so that we are able to cross the barrier with much more ease.

The Drivetrain is the most fundamental aspect of the robot, as every other mechanism is mounted about it. Having a strong, durable, and fast but efficient drivetrain not only helps with proper maneuvering but also helps with being able to easily mount items without any issues. We planned out our drivetrain in a way that we could mount the intake structure's main c channels and braces in a proper and convenient way.

Our drivetrain wheel gap is 4 holes wide, allowing us to remain narrow which helps with maneuverability in push back. The overall drivetrain dimension is 14.5" by 13.5", with inner c channels of 13.5".

We were able to mount two full-length crossbraces that square the drivetrain, with the top being a full-length c-channel and the bottom one being a high-strength axle.

We started by taking a 35 hole c-channel with a 5 wide gap. This sets as the temporary foundation for squaring our drivetrain. Using shoulder screws, we mapped out where the c channels have to be arranged.

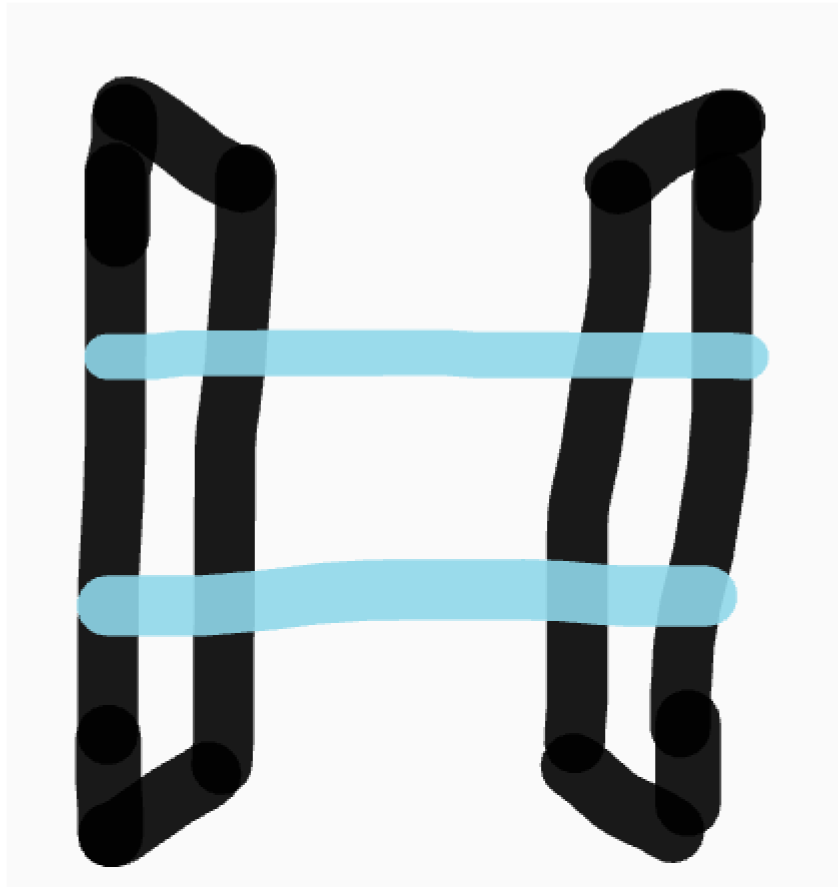
After attaching the two outer 29 hole c channels and the 2 27 hole inner c channels, we proceeded with the top crossbrace. We kept the top crossbrace at the back of the robot as this was where we would mount our intake bracing angles. Using bearings to square the brace, we were able to attach 8 screws without problem to secure it in.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

From there, we took our 1x1 angle, cut out flanges to fit the wheels, and attached it to the bottom of the drivetrain using 0.500” and 0.250” inch spacers.

When checking for sturdiness after removing the original, temporary squaring c-channel, the drivetrain remained sturdy and unable to bend around. This meant that we were ready to proceed to creating drivetrain screwjoints.

Attached to the image is a quick sketch of our drivetrain structure, with the blue lines indicating crossbraces.



Was the goal reached? **YES**

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

12/24/25 -Meeting #2 Build Drivetrain Screwjoints

Build

Goal: Finish all 8 Drivetrain Screwjoints + Axles

Our plan is to attach all 8 of them, 6 of them being for actual wheels and two of them being for idler gears. Given that our screwjoints have to be properly spaced out to give wheels as little slop room as possible, we calculated the amount of spacing we would need to successfully add the screwjoint.

In a 2 inch gap for screwjointed wheels, we used:
Bearing - 0.25 inch
Locknut- 0.25 inch
Spacer- 0.25 inch
Spacer- 0.125 inch
Washer- <0.0625 in
Gear- 0.25 inch
Wheel- 0.5 in
Washer- <0.0625 inch
Locknut- 0.25 in

In total, this process took us around 2 hours. We individually tested each wheel for how long they spun for after installing each joint to see if any wheel experienced a friction force. Ideally, each wheel must be able to spin for around 13 seconds, preferably more.

Wheel 1-

Test #	1	2	3	4
Duration of Spin (seconds)	15 seconds	15 seconds	15 seconds	15 seconds

Wheel 2-

Test #	1	2	3	4
Duration of Spin (seconds)	13 seconds	13 seconds	13 seconds	13 seconds

Wheel 3-

Test #	1	2	3	4
Duration of Spin (seconds)	16 seconds	17 seconds	16 seconds	16 seconds

Wheel 4-

Test #	1	2	3	4
Duration of Spin (seconds)	15 seconds	15 seconds	15 seconds	15 seconds

Wheel 5-

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Test #	1	2	3	4
Duration of Spin (seconds)	15 seconds	13 seconds	12 seconds	14 seconds

Wheel 6-

Test #	1	2	3	4
Duration of Spin (seconds)	15 seconds	13 seconds	15 seconds	14 seconds

Next, we did our two idler gears. The layout on a 2 inch gap:
Bearing - 0.25 inch
Locknut- 0.25 inch
Spacer- 1 inch
Washer- <0.0625 in
Gear- 0.25 inch
Washer- <0.0625 inch
Locknut- 0.25 inch

When testing (target seconds: 7):

Idler 1-

Test #	1	2	3	4
Duration of Spin (seconds)	7 seconds	7 seconds	7 seconds	7 seconds

Idler 2-

Test #	1	2	3	4
Duration of Spin (seconds)	7 seconds	8 seconds	7 seconds	8 seconds

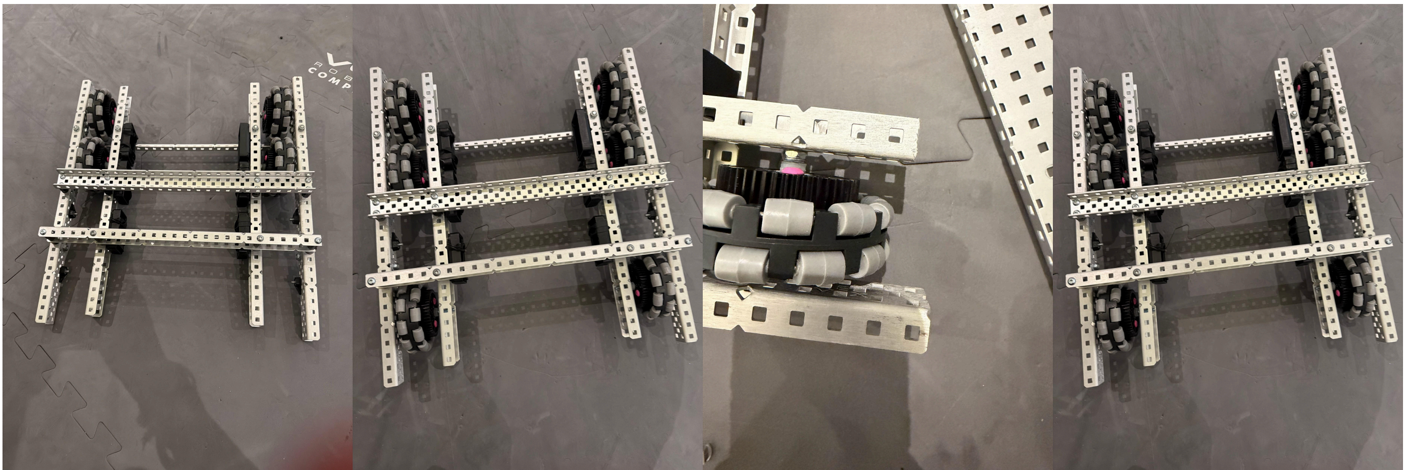
All of our tests being above the target duration is a good sign, however further testing with the powered motors are needed to confirm that everything is good, which is what we ended up testing today.

When testing with the motors, we found the below work values.

Side / Test	**1**	**2**	**3**	**4**
Right Side	0.2 W	0.2 W	0.2 W	0.1 W
Left Side	0.2 W	0.2 W	0.2 W	0.1 W

The amount of friction (in Watts) on both sides of the drivetrain throughout the tuning process. Was the goal reached? **YES**

355Y Drivetrain



12/25/25 -Meeting #3 Build Intake Structure Build

Goal: Create intake structure

Our next step in the design process was to create the intake structure for our robot.

Design Statement: Create a structurally strong intake ramp for blocks to be stored, which can also be built further upon to mount chain and motors/floating intake

Constraints

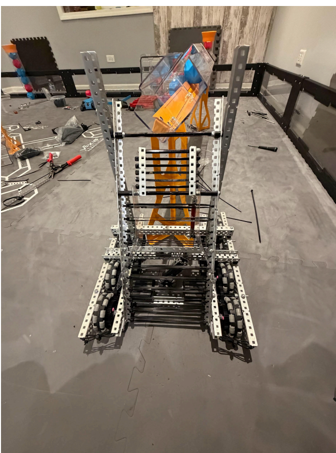
- Height must not exceed 18” high
- The width between the balls must be within two balls wide.

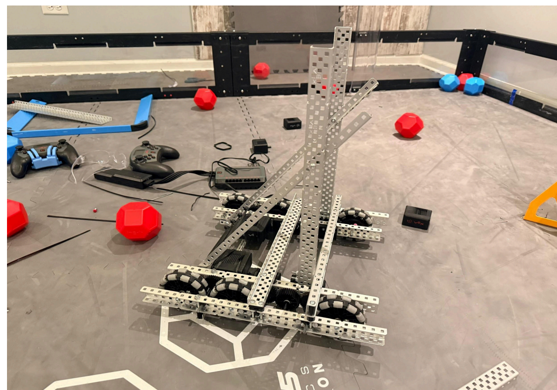
Implemented Solution

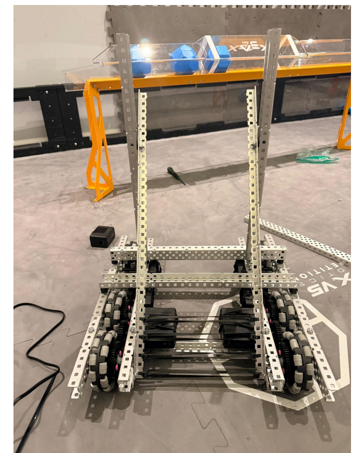
Our drivetrain is 27 holes wide, running on a 4 wide gap between each wheel side. Therefore, we have around 15 holes to work with.

Initially, our thought process was to use c channels as our diagonal ramp piece. However, we realized later on that two balls can’t fit in between, so we switched to half-cut 3 wides to create our intake ramp.

Additionally, we attached two vertical c channels that we can use for structural purposes, to build a wing off of, and to add a joint connection for our ramp, keeping it secure. Here is a photo of our skeleton:







As displayed in the first photo to the left, we created our ramp out of elastic forces (rubber bands). We found that an elastic force not only helps with capacity and ball slipping, but it also provides an upward force that, due to physics, would help with scoring speed. We can use vector physics to calculate this idea:

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

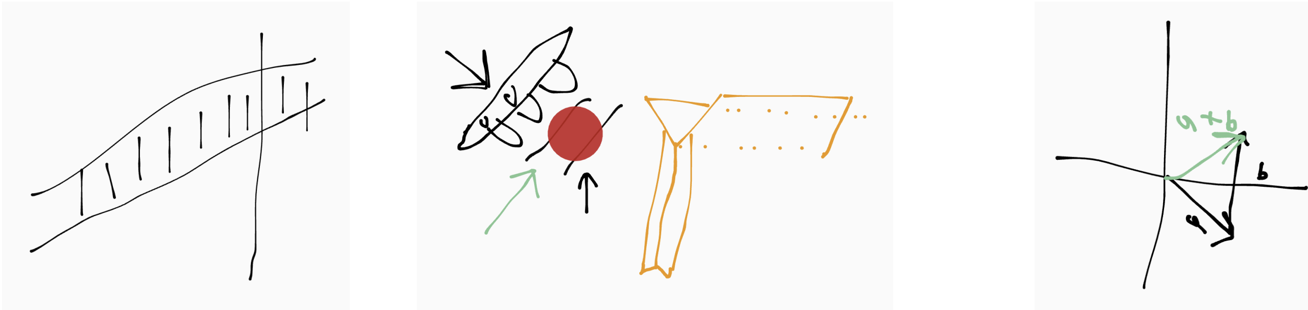


Figure 1 (Left): A quick sketch of our ramp intake. Figure 2 (Middle): Forces diagram of the system of a ball. A southeastern force is applied from the chain, a north force is applied from the ramp, and the resultant force is the green arrow, a northeastern force which is ultimately what we are looking for. Figure 3 (Right): Vector addition, the core principle explaining why rubber bands would be most efficient. When calculating the sum of vector a (the chain's force) and vector b (the rubber band's force), we get vector a+b, which shows that our intake speed will be faster.

Additionally, for our **middle goal control**, we built a **trapdoor** structure. This is built off of a screwjoint, and is controlled using a piston.

Results: We successfully built our intake ramp, braced it well, and successfully implemented our trapdoor. The rubber band's elastic force did not bend our angles, and we were able to drive around and play aggressively pretty well when checking driver control.

Was the goal reached? **YES**

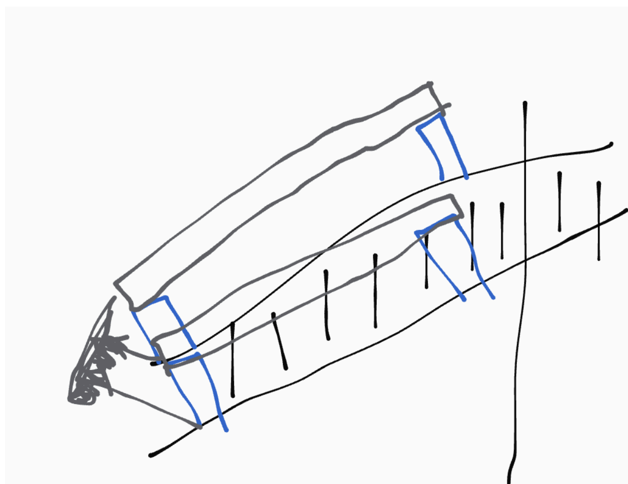
12/26/25 -Meeting #4 Build Intake Structure p2 Build

Goal: Continue to build off of the skeleton for intake chain structure.

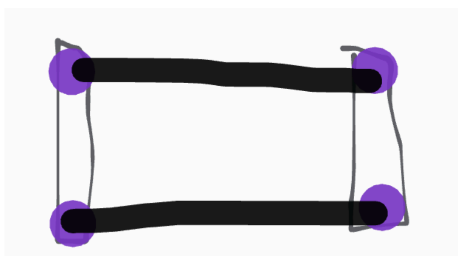
Design Statement: To use chains to transport blocks into our intake storage system, we need to find mounting points. We need to create structurally strong mounts, using shoulder screws to reduce workload on motor-axle joints, and find the perfect height for axles to be mounted on. Additionally, build floating intake structure

Constraints: The biggest constraint is to make sure that our chain mounts don't hit the natural movement of our floating intake when intaking balls. Other than that, the mounts can't extend past the floating intake's height.

Implemented Solution: We use c channels, and we connect them off the side of the diagonal ramp implemented in yesterday's meeting.



Attached here is a diagram of our hypothetical solution. The angled ramp is labeled in black, our work from yesterday. Blue c channels are mounted off of that angled ramp, and grey c channels connect the two blue c channels on each side. This helps shape our chain's path. **Shoulder screws** were used to make sure everything is square, helping us guarantee that our intake's workload is significantly reduced as one motor needs to power three rollers.



To further brace the two c channels, we used a coupler + standoff structure. Purple dots signify shaft collars, and using couplers, we attached standoffs, highlighted in black, to make sure that side of the intake is square.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Results: We successfully created the entirety of our intake structure. After trying to play around with the structure, we were unable to shift anything around, showing that our structure is not prone to structural bending and damage off of contact.

Was the goal reached? **YES**

12/27/25 -Meeting #5 Add Chain + Flexwheels

Build

Goal: Add Chain to our Intake, finish floating intake.

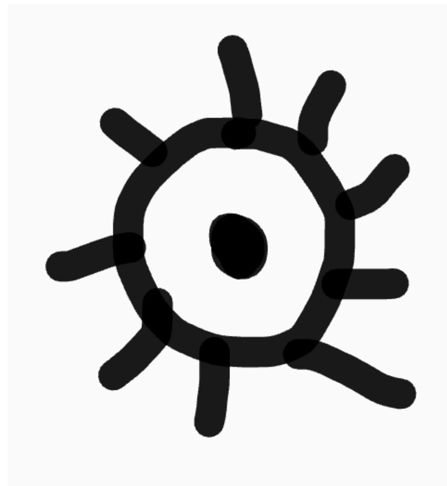
Design Statement: We need to build a reliable chain transport, built off of two joints, that can transport the ball from the bottom ground to the top where it will eventually be scored into the long goal.

Constraints:

- Subsystem must require 2W maximum off of the rollers, combined. When it comes to chain mechanisms, even if the individual axles spin at a low level of friction, the overall connected system tends to require a greater amount of force from the motor. If not tuned well, we would have to frequently deal with an overheating motor, as mechanisms that typically pull around 2W of friction when rotating tend to be unreliable and require lots of tuning to make it more efficient.
- Must not hit the floating stage of the intake
- Must not have a chain flap extend past 18” in height

Implementation

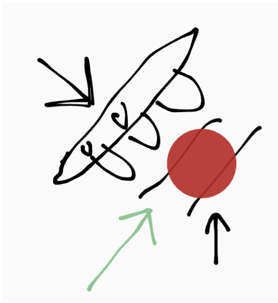
The first step of the build process was to add the axles to set the joints for where the chain would spin off of. To help us test the right spot, we added a single sprocket, covered in flaps. The figure attached below is a quick sketch of what we attached to the bottom and top axle joints.

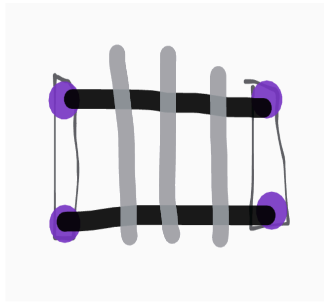


When we spun our axle and sprocket, we checked for two things:

- Did the flaps hit our floating intake? If this was the case, we would have to rethink our slot chosen for our axle. Fixes would include: moving our axle upwards, moving it inwards, and if all didn't work, either reducing the size of our flaps or cutting the flaps.
- Was the **CHAIN**, not the **FLAPS**, hitting the blocks? If so, we needed to move the axle upwards.

Here are quick sketches that we drew of what the intake would hypothetically look like:





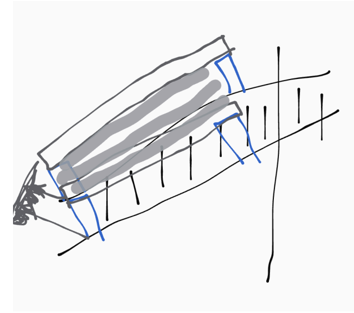


Figure 1 (Left): In this figure we bring back the sketch of intake forces. The chain was positioned over the ball in a diagonal line. **Figure 2 (Middle):** This would be a front view of our intake, bringing back the sketch from when we braced the Intake structure to keep it square. **Figure 3 (Right):** This image is a drawing of our side view of the ruiguan intake.

TESTING: This took a lot of iteration to find the perfect spot. Here is a look at our table:

Table 1: Testing position of bottom intake shaft.

Test #	Result Found	Resultant Action
1	Chain flaps hitting the floating intake	Move the axle slightly **inwards** to increase clearance from the floating intake
2	Chain flaps hitting the floating intake	Move the axle **upwards** to lift the chain path away from the intake
3	Block hitting the chain	Move the axle **upwards** to increase the clearance between the chain and the blocks
4	Block hitting the sprocket	Move the axle **upwards** and slightly **inwards** to prevent sprocket interference
5	Block not enough contact with chain	Move the axle slightly **downwards** to improve contact with the chain flaps
6	Chain flaps hitting the floating intake	Trim the **size of the flaps slightly** to reduce interference
7	Block hitting the chain	Move the axle **upwards** to raise the chain path
8	Block not enough contact with chain	Move the axle **slightly inward** to bring the chain closer to the block path
9	Block hitting the sprocket	Move the axle **upwards slightly** to avoid contact

10	No interference; block maintains consistent contact with chain	**Final position kept – configuration accepted**

After running several iterations, we were able to gradually refine the placement of our chain axle joints. Early tests revealed frequent interference issues, primarily with the chain flaps hitting the floating intake and the blocks contacting either the chain or the sprocket. To resolve these issues, we systematically adjusted the axle position by moving it upwards, inwards, or slightly downwards depending on the type of interference observed. In some cases, we also modified the mechanism itself by trimming the size of the chain flaps to reduce unwanted contact. Throughout testing, we focused on maintaining consistent contact between the blocks and the chain while ensuring that the mechanism stayed within the height constraint and did not interfere with the floating intake. By the tenth test, we reached a configuration where the chain rotated smoothly, the blocks maintained reliable contact with the flaps, and no collisions occurred with surrounding components. This final configuration was therefore selected as the design we kept for our intake chain transport system.

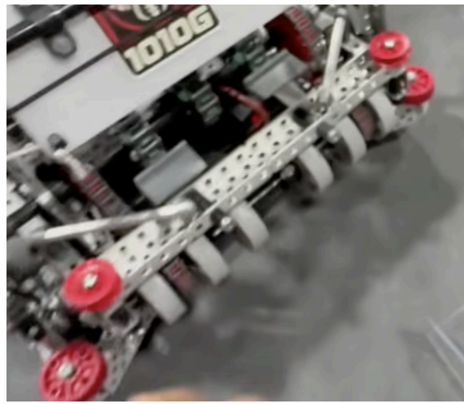
Attaching the **top axle** was rather easier, as when we checked with flaps, it did not hit anything. Additionally, blocks were able to pass through fairly easily.

To create our chain, we used **9p chain**, as we could use the chain links that you can attach flaps to. From there, we spaced out all of our chain flaps so that each flap was spaced out 7 links away from each other. We determined that this was the perfect spacing between flaps because we could fit exactly one block in between each flap when laying it out on the ground.

We then proceeded to attach the chains to our completed axle joints, which consisted of three 12t sprockets. **We chose 12p over 6p sprockets due to speed**, as that is our biggest priority for our intake.

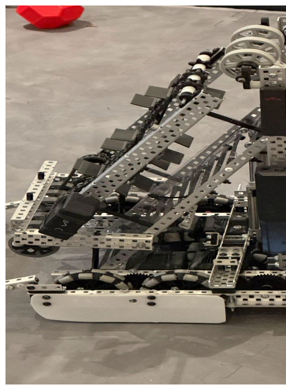
After finishing this, we proceeded to attach the high strength axle and flexwheels for our floating intake. We used 6 flexwheels, evenly spacing each other out. Inspired by team 1010G, we left space in between our middle two flexwheels, just big enough for a block to fit in between, so that our low goal process would become easier. A display of our inspiration is shown below:

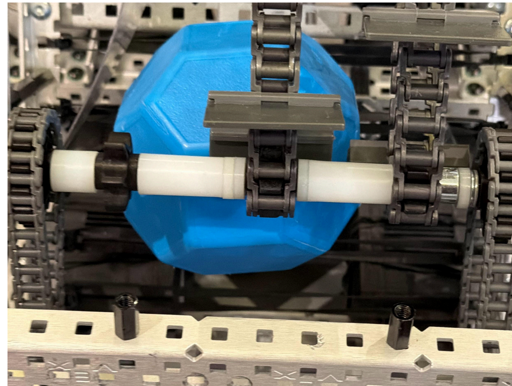
Here is what our intake looked like after the build process ended for this:



Spaced-Out Middle Flexwheels on Floating Intake (1010G, Youtube)

After Building, this is what our robot looked like:





Testing for Friction: When we linked all of the mechanisms we built together, and attached our motor, the first thing we tested was our capacity, and our natural intake-ball progression, which we found was good. When we checked for friction, we found that our entire intake spun at a combined force of **1.9W.**

As a team, we deemed tuning this friction not worth the effort for the time constraint we are under. There are aspects like driver practice and autonomous routines that were more important, and a 1.9W spinning intake was good enough to last us 4 minutes of continuous activity.

Was the goal reached? **YES**

12/28/25 -Meeting #6 Build Counter Roller

Build

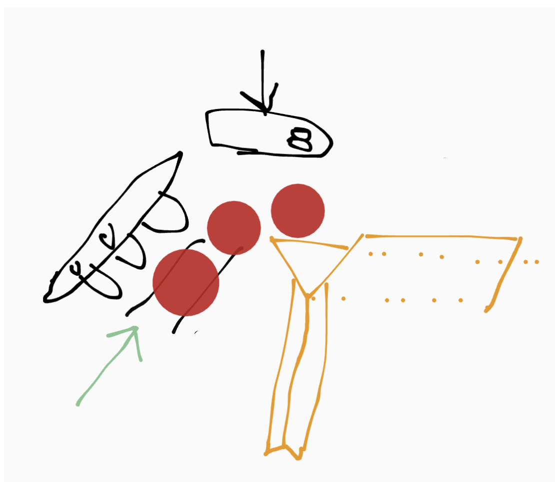
Goal: Finish Building Counter Roller

Design Statement: Create a counter roller to provide that final push into the long goal.

Constraints: Bot must not exceed a height of 18”. Outside of that, making sure the counter roller is light.

Implementation:

The structural design for a counter roller is straightforward: It requires the use of two angles, mounted off of two screwjoints and squared, that has a smaller roller which would add a force toward the long goal that makes our scoring faster. Attached is our hypothetical counter-roller design, displayed in a quick sketch:



However, there were various options that we could choose from: cut flexwheels, regular flexwheels, or sprockets + rubber bands.

We ended up testing all three of these ideas. Using a high strength axle, we mounted each of our counter roller options. We tested the scoring time of 6 balls using a stopwatch.

We acknowledge that our times can be off, and that could serve as a potential error in this experiment.

This table displays our results:

Option/Test Number	1	2	3	4	5	6

Regular Flexwheels	1.5	1.4	1.7	1.6	1.5	1.7
Sprocket + Bands	1.7	1.9	1.8	1.9	2.0	1.9
Cut Flexwheels	2.1	2.3	1.9	2.0	2.2	2.1

The test was really close. We ended up choosing the **regular flexwheel** option, 30A sprockets with a small radius. Our counter roller has no force pulling it down so that balls wont get stuck to the roller but it has a stopper so that it doesn't expand past its desired angle.

Was the goal reached: **YES**

12/29/25 - Meeting #7 Build Tongue Structure Build

Goal: Finish building the tongue mechanism, tune the rotation piece and rubber bands.

Design Statement: Create a matchloading mechanism that can clear out the matchloader as fast as possible. The matchloader should also be able to grab blocks during autonomous runs, making our mechanism versatile.

Constraints:

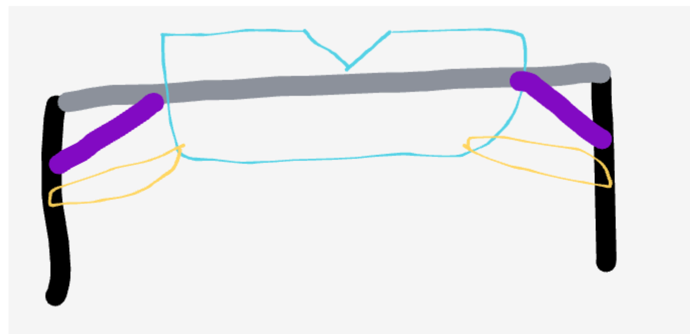
- The mechanism must be powered on one piston
- The starting position of the matchloading mech should not extend past 18”
- The extended position of the matchloading mech should not extend past 24”
- Plastic should not shatter

Implementation:

As indicated previously, we will build a tongue mech.

Our structure starts off with mounting two 3 hole angles to the sides of our drivetrain. From there, we mount two 13 hole angles and connect them with a high strength shaft. Instead of drilling out holes through our high strength shaft, we use high strength collars to mount it to the previously mounted 13 hole angles.

In between, we add two more shaft collars for triangle bracing, we add two pillow bearings, and high strength spacers to space out our tongue mech into the middle of the shaft. Attached here is a sketch of our hypothetical design.

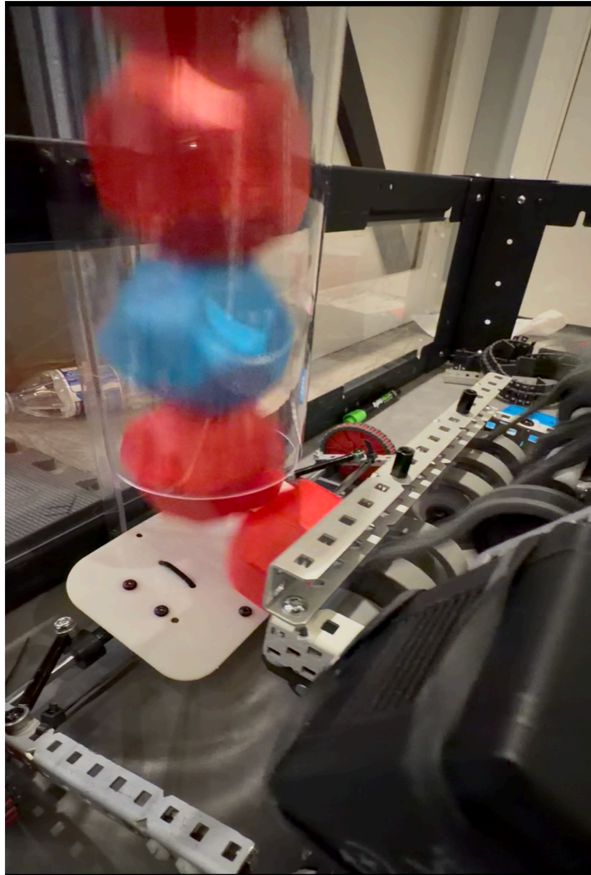


Black lines indicate the 13 hole angles. **Grey lines** indicate the high strength axle. **Cyan lines** indicate the plastic piece. **Purple lines** indicate secure triangle bracing. **Yellow lines** indicate rubber band placements to help tune the position of the plastic piece as it enters the matchloading station.

Using rubber bands to tune the height of the plastic piece’s starting position, we were able to find the right tension on the plastic piece to smoothly dig underneath the first block in the matchloading station, while also not hitting the rim of the matchloader.

Attached here is the finished tongue mech, as it digs underneath a goal:

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke



Testing: After constructing the mechanism, we ran several tests. Our target time for clearing out the matchloader had to be under 1.5 seconds:

Trial #	1	2	3	4	5
Time (seconds)	1.5	1.4	1.5	1.3	1.3

We used a stopwatch to test our mechanism, and we acknowledge that there is human error. However, with the times observed, we felt comfortable proceeding without tuning further.

Was the goal reached? **YES**

12/30/25 -Meeting #8 Build Wing Brainstorm & Structure

Build

Goal: Build a de-scoring wing

Design Statement: Create a structurally secure wing that can remove blocks out of a scoring zone. It should be able to stay intact amidst heavy defense and pressure.

Constraints:

- Powered by a single piston
- Must not have a starting position of over 18”
- Must not have an expansion width of over 24”
- Must not have an expansion height of over 24”
- Should not clamp onto the goal

Brainstorming:

Descore Mechanism - Flaws of Previous Robot:

1. We had two de-score wings, and in order to stay in the 22 inch limit we were forced to make both of them short. This caused a few problems:
    a. It was very hard to get the perfect angle to allow the de-score wings to reach inside the goal.
    b. We were called for clamping multiple times because the wing was so short it would make us clamp to the edges of the goal.

Implementation: We attached a c channel off of a screwjoint, and attached that c channel to a piston mounted off of a screwjoint too. We attached a standoff to the end of the channel to reach the goal.

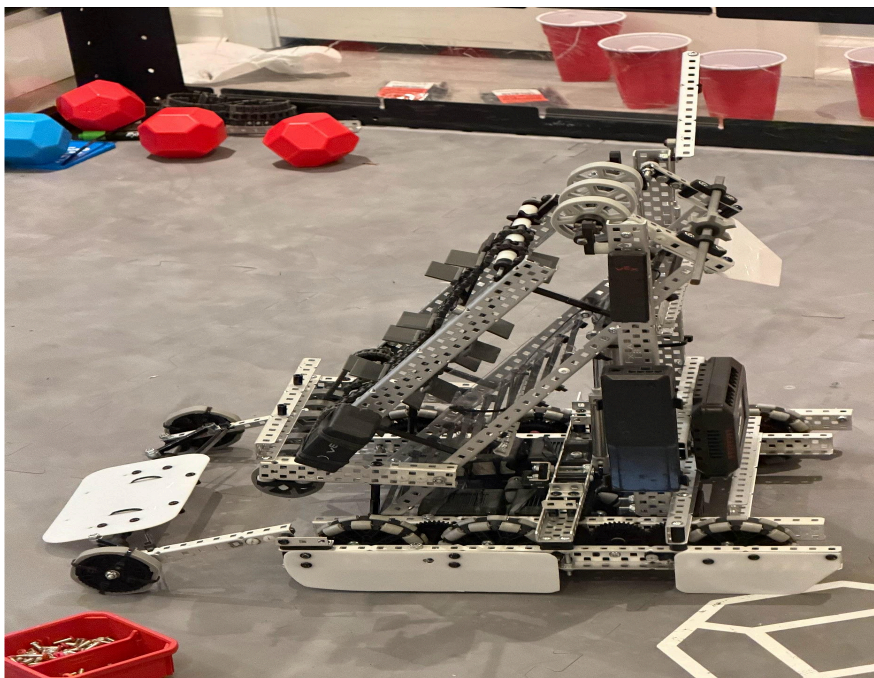
Improvements are listed here:

1. We changed to have one wing, this allowed us to have a longer wing, and this can allow easier access from a greater distance from the long goal, which gives us a few benefits:
    a. We have a lower risk of getting called for clamping.
    b. We have a higher centrifugal force, which can allow us to push out more balls.
        i. We calculate centrifugal force through the formula:
            1. F = mass*(angular velocity)^2 * radius
        ii. Given the radii:
            1. **Case 1:** r_1 = 6.5 inches
            2. **Case 2:** r_2 = 5 inches
        iii. Assume:
            1. Same mass
            2. Same angular velocity

iv. Compare the Forces:
    1. F_1 / F_2 = mass*(angular velocity)^2*r_1 / mass*(angular velocity)^2*r_2
v. Cancel Common Terms:
    1. $F_1 / F_2 = r_1 / r_2$
vi. Substitute Values:
    1. $F_1 / F_2 = 6.5 / 5 = 1.3$
vii. Final Result:
    1. $R_{6.5 \text{ inches}} = 1.3 * R_{5 \text{ inches}}$

c. This shows that our new wing (6.5 inches) has a very high chance of being better than our old wing (5 inches).

Attached here is a photo of our wing on the robot:



Testing:

Green = Met Requirements
Orange = Close to Meeting Requirements
Red = Not Meeting Requirements

Length of Wing	Balls Flung Out	Clamping Violation
4.5 inches	2 Balls	True
6 inches	4 Balls	False

Was the goal reached? **YES**

Tournament Recap #3

Competition 01/04/2026: Great Lakes II (at Batavia High School)

Results:
Quarter-Finals (Lost to Tournament Champions)

Immediate Action:
While reflecting on the recent tournament, the team was proud of the new bot made during the rebuild, since we realized that, even though we didn't even make semifinals, the robot performed exceptionally well, being one of the stronger robots at the competition.

Team Reflections:

Mithil (Designer) - This competition was a great learning experience, as we were able to make quarter finals again but we lost to not having an optimized descore mechanism which made it insanely difficult for our driver, Sathvik, to align quickly and effectively to the long goal in order to be able to descore the balls which if he had been successful in, could have led us to winning our match.

Naisha (Mechanic) - The tournament was very good and I was very proud on how the team overall performed. This was a really great learning experience and if we were able to descore the balls better than we would have done even better. It was a great competition and helped us improve our robot for the future.

Sathvik (Driver, Mechanic) - GL2 was such a thrilling experience, but also was ultimately a successful competition for our design iteration as even though it wasn't the strongest bot, with some great driving and some great alliances, we ended up doing very well and I am proud of our team.

Shaurya (Mechanic) - At GL2, I realized how important autonomous is to our success in games as a team. Although it took a while to work, I appreciate the work and effort Sonit put into the autonomous program during the competition and at home. I strive to make our autonomous even stronger, on both sides, and also help create an autonomous skills program. About the competition itself, I was extremely proud of our robot's performance, since it was a brand new bot and it still ended up being one of the strongest bots at the competition, even ranking 4th and being undefeated after the lunch break.

Shreyas (Mechanic) - GL2 was a great competition where we unfortunately lost in the Quarter Finals against the Tournament Champions since we didn't have a fully functioning descore mechanism. It was a great learning experience and helped our robot improve in the future.

Sonit (Programmer) - This tournament was very good and I am proud of how the team performed. It was a great learning experience and we did very well overall in the matches. Using our knowledge on how we did, we improved our robot to make sure not to repeat the same mistakes in the future.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Goals After GL2 Competition

Rebuild Matchloading Mechanism - Our matchloading mechanism was unable to dig from different angles on the block during the competition, which really prevented Sathvik from playing a smooth driver role.

Watch Game Film - Part of becoming a better driver includes watching game film to analyze missed opportunities **AND** to scout our opponents for future matches. Sathvik plans on taking a dedicated hour to watch his game film and find spots where he can make obvious alternate choices to further our result.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Film Analysis of Great Lakes 2 Matches 1/05/26

Q7	Score
99371C + 355Y	72
8995M + 355T	25

This was a great start to the tournament and a real confidence booster, since we know our robot is actually good now. We won autos, played it easy in driver, and secured our victory.

Q12	Score
499R + 5741A	16
355Y + 99371A	19

This was a match that was closer than it should have been. 499R is a really strong team however 99371A played some incredible defense to help our team get the slight edge in scoring to win the game.

Q28	Score
355Y + 2360N	94
499G + 60172V	23

We won autos, played it easy in driver, filled a goal, and parked. This was a comfortable victory.

Q42	Score
355Y + 725K	58
2360X + 355R	9

We won autos, played it easy again in driver, and had zero competition from the opposing alliance to prevent us from scoring.

Q53	Score
355Y + 3695E	23

8995E + 1755N	74

This was our first loss of the day. We lost autons, had an alliance who didn't really help out much since their drivetrain didn't work, and faced an unfortunate no-call from the referee of a pinning that lasted around 12 seconds.

Q61	Score
99371G + 8995H	80
355J + 355Y	11

We won autons, however our alliance wasn't much help defending me from getting pushed around, and we lost due to 8995H's strong wing.

Q72	Score
1755K + 2360A	131
355S + 355Y	12

This match was just a disaster in total, our alliance's battery disconnected and our drivetrain wire somehow popped out of our motor.

We ended the qualifications round 4-3. 1755K picked us to form the 8th seed.

R16 2-1	Score
1755K + 355Y	103
725K + 14153M	29

This was a comfortable victory, as we won autons, and we dominated the control zones. Flawless execution.

QF 1-1	Score
2360A + 8995H	82
1755K + 355Y	8

This match didn't start well for us as 2360A filled the middle goal in auton, making us lose by 1 point. From that point on, Sathvik faced heavy defense from 2360A, and 1755K stuck to a defensive mode on 2360A, leaving 8995H a free chance to score and win the game.

1/08/26 -Meeting #9 Redo Matchloader

Build

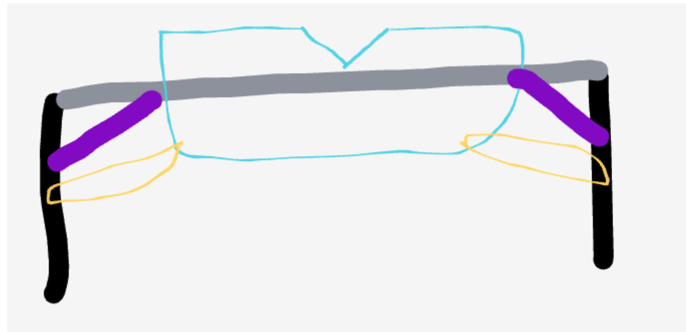
Unfortunately, during driver practice, Sathvik broke our matchloader.

Design Statement: Build a structurally stable matchloading mechanism that can successfully clear our matchloading stations in quick time.

Constraints:
- The mechanism must be powered on one piston
- The starting position of the matchloading mech should not extend past 18”
- The extended position of the matchloading mech should not extend past 24”
- Plastic should not shatter

Implementation:

Bringing back this image:



The goal was to make sure our triangle cut-out was precise. After constructing the mechanism, we ran tests on flat-angle blocks in the matchloader:

Test #	1	2	3	4	5
Result	yes	yes	yes	yes	yes

We determined our new matchloader was successful, and we are ready for Mundelein!

Was the goal reached? **YES**

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Tournament Recap #4

Competition 01/10/2026: Mundelein

Results:
Ranked 6th in Qualification Round
Lost in Quarterfinals
State Qualification through Skills

Immediate Action:
Most would agree that Mundelein was by far our best performance at a competition so far. As a team, we were a major competitor for the tournament champion title. We qualified for State through the skills ranking by placing 5th, and we are planning a rebuild for State. We realized a lack of capacity in our bot, so we probably will go back to an S-Curve.

Team Reflections:

Mithil (Designer) - Mundelein was our strongest competition so far. We were able to place top 5 in every scoring category, from Autonomous to Qualifications. This was a major breakthrough as this competition let the team understand that with the proper engineering process we would be able to achieve higher goals. We were also able to qualify to state which will now give us time to better execute the engineering process.

Naisha (Mechanic) - xxxxx

Sathvik (Driver, Mechanic) - xxxxx

Shaurya (Mechanic) - I'm very proud of our performance at Mundelein. Along with qualifying for State through skills, we placed extremely highly in every category. Although we didn't qualify through the bracket, we placed 5th in skills, and I'm extremely happy about our qualification.

Shreyas (Mechanic) - Mundelein was a very strong competition for us as we dominated, however unfortunately when it came to the quarter finals, our alliance unfortunately tipped over which led to Sathvik having to 1v2 which was very difficult and we ended up losing.

Sonit (Programmer) - xxxxx

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Film Analysis of Mundelein Matches 1/11/26

Q2	Score
499K + 60172T	34
355Y + 60172W	44

This was a comfortable victory but it was closer than it should have been. We won autons, and played it easy in driver.

Q16	Score
355Y + 4454X	64
499D + 60172S	21

This was another comfortable victory, where we won autons and secured control zones to play it easy in driver control. Our autons are really consistent.

Q32	Score
355Y + 499Z	19
1965C + 355P	46

For what it's worth, this was a wonderful defensive showing on Sathvik's part. At the start of the match, we realized that we forgot to plug in our intake wire, essentially making our intake dysfunctional. Both teams lost autos. Sathvik played heavy defense on 355P, forcing them to switch their strategy into defense while he cleared out the entire scoring zone. 355P's motor came out, essentially making it a 1v1 with 1965C and 499Z. However, 499Z easily had the weaker bot as they had a clawbot, and 1965C scored 7 blocks and pushed into the control zone to secure the match for blue alliance.

Q47	Score
6410M + 4454V	18
48652A + 355Y	50

We won autos, played it easy again in driver, and had zero competition from the opposing alliance to prevent us from scoring.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Q56	Score
862C + 499X	9
499R + 355Y	116

This was a clean match in total, flawless execution on both 499R and our part. Unfortunately, we could not secure the AWP but this was a real booster for our stats in the tournament.

Q64	Score
499G + 355Y	106
2827Y + 862D	21

This was a wonderful way to end our qualifications. 2827Y, at the time, was the 4th ranked team in the tournament, and being able to defeat them with a huge margin really secured us this victory. We won autons, and immediately told 499G to try to play as much defense on the other two teams as possible. We proceeded to fill up a long goal and a half and get a park to win the match while playing aggressive defense against 2827Y at the same time.

We ended the qualifications round 5-1, in the 6th seed.

Our original plan was to alliance with 2827A, a strong team with a great driver and a capable wing. However, after 499X picked us and we chose to decline, we were forced to become an alliance captain, securing 1755K.

R16 4-1	Score
355Y + 1755K	113
6410C + 99371C	24

This was a comfortable victory, as we won autons, and we dominated the control zones. Flawless execution.

QF 2-1	Score
2827A + 2018B	77
355Y + 1755K	21

Unfortunately, 1755K's autons did not hit and 2827A was able to win over the middle goal. On top of that, 1755K tipped over due to their center of gravity, essentially finishing this game before it even started. Sathvik held up his own part playing defense against both robots while trying to protect his long goal, but when he had to de-score the other long goal, the red alliance was able to de-score our filled goal, finishing the game.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

12/01/25 - Final Thoughts on Design Iteration 2

Strengths:
- Faster scoring than v1
- Better de-scoring than v1
- Parking clearance

Weaknesses:
- Middle goal and low goal scoring were not the greatest, especially a problem with ruiguan-styled robots.
- Heavy robot

Areas of focus for v3:
- Driver Practice
- Better middle goal scoring
- Better Parking Clearance
- Weight reduction on next robot

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

State Rebuild

New Teammate!

355Y welcomes **Ediz**, an experienced **programmer** and **strategist**, who can work with Sonit to help make our autons more consistent and as versatile as possible. We have seen from other teams that having two programmers can help split the workload, and also help keep each other in check, ultimately making this a great decision for us.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

1/12/26 - v3 Design Brief

Identify

Game Constraints:

- Robot must be able to fit in an 18” by 18” by 18” cube at the start of the match
- Total combination of motors must not exceed 88 watts
- Must be built only from VEX-approved parts
- Custom plastic is limited to twelve pieces, each piece fitting within a 4” x 8” area
- **Bot must be built + programmed before March 13th (State Competition)**

Personal Goals and Strategy:

- Score more than 10 blocks in 3 scoring zones during autonomous period
- Strategize with alliance team to get the autonomous bonus and win point
- Work with alliance team to score as many points as possible
- **Control all Control Zones**
- Descore opponent scoring zones before end of match

Robot Subsystems:

- Drivetrain
- Intake mechanism to load blocks into the robot
- Scoring mechanism to score blocks into goals
- Mechanism to remove out blocks from a scoring zone
- Mechanism to smoothly collect match loads from match loading zones

Gantt Chart

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

1/12/26 - v3 Intake Criteria

Identify

Problem Statement: Design an intake to interact with blocks by collecting and storing them within the mechanism. Intake should at least be able to transport a block from ground to goal to enable scoring opportunities.

Solution Constraints:

- Must work efficiently on a 22W maximum
- Must not extend past the 18" size constraint on all three dimensions

Solution Goals:

Maximum power consumption for intake must not exceed 0.5W.

- Given that most of our intake will be linked by chain, the motors will most likely be working harder to power the entire mechanism. However, the amount of power needed to work the mechanism must remain low so that the intake works efficiently and doesn't overheat the motor.

Intake must be able to score on long goals and both center goals (medium + low heights)

- Being able to utilize all types of scoring methods could prove handy in a close match where long goals are filled to the maximum. Being able to score on both of the center goals could give both our team and our alliance an advantage over others.

Intake should be able to possess around 8-9 blocks within its mechanism

- Being able to possess multiple blocks at once will make it effortless to create splash plays of point gain, rather than possessing around 5 blocks at a time and having to go back to retrieve 5 more blocks in another rotation.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

1/12/26 - Brainstorm: Why are we Rebuilding? Brainstorm

1. **Strength Of Previous Bot:**
    a. Drivetrain
        i. 450 RPM gave us a solid balance between speed and torque, allowing us to move in an agile manner across the field while not being out of control.
        ii. The width of the drivetrain was perfect, as it would prevent jamming the first stage of the intake if two balls were intake side-by-side.
2. **Weaknesses of Previous Robot:**
    a. Speed of Outtake / Intake:
        i. We used flaps to power our outtake which was easy to tune, but was much slower than rubber band rollers, which is important because when you are getting defended, having the agility and speed to unload at maximum speed will help get most points.
    b. Middle-Goal Scoring:
        i. Our middle goal scoring was very weak, because the flaps didn't give enough power for the ball to fall out.
        ii. Now with the S-Bot we are able to control the outtake between middle and high goal through a state mechanism which allow for us to effectively and quickly score in both goals.
    c. Intake:
        i. Our intake was a floating intake and got stuck in positions very easily which causes the balls to not flow in smoothly.
        ii. For our S-Bot our intake will be lower and hence it will make sure that even at a higher angle it will be able to intake the balls.

Robot Version	Single Lane	Ball Capacity	Middle Goal Scoring	Intake	Outtake Mechanism
S-Bot v1	True (Lots of Jamming, not tuned properly)	4 Balls	Fast	2 inches off of the ground	Rollers (Not Tuned)
Linear Bot	False	7 Balls	Slow	2.25 inches off of the ground	Flaps
S-Bot v2 (hypothetical)	True	8-9 Balls	Fast	1.5 inches off of the Ground	Rollers (Tuned)

We have seen multiple teams switch to an S-styled robot. We believe this robot is the most versatile to give us a great chance of winning at state, hence we have decided to proceed with it.

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

1/15/26 Re-Adjusting Drivetrain

Build

Goal: To finish redoing our drivetrain

Design Statement: Create a structurally stable drivetrain that can help move around multiple subsystems.

Constraints:

- Must work on 66W of power maximum
- Must not exceed 18” in length
- Must not exceed 18” In width
- Must have enough room to cross the parking barrier
- Must have at least two crossbraces on the robot.

Brainstorm: Why have we decided to change the width of our robot?
The problem with a 27 hole robot on a 4 wide drivetrain is that a ball can’t fit in between the drive motors, as it just barely can’t squeeze through. Our intention with this robot was to create a fast, single-lane robot so that funnelling won’t slow down our scoring. Having the balls pass through the motors helps us keep everything streamlined in one lane.

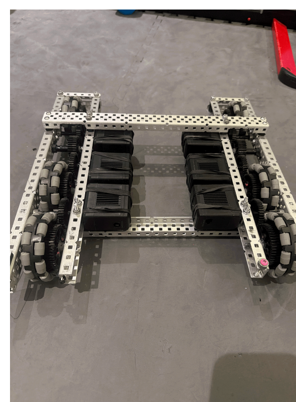
Implementation:
We started by de-attaching our crossbraces from the previous build. When Sathvik examined the metal, we noticed that our bottom crossbrace was bent, showing that our drive was not structurally stable enough.

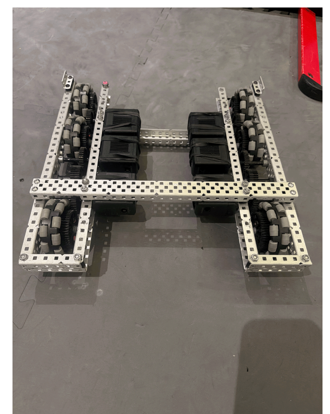
Once we were left with two sides of the drivetrain, we attached them to a 5 wide to help square the drivetrain as we attach the new two crossbraces.

Our next step was to find the placement that is most optimal for our drivetrain. When we found our intended spot, we noticed that we had to cut a few flanges on our metal to allow for wheels to pass. After dremelling those off the metal, we checked for structural stability, and there was no sign of weakness in the metal.

We then proceeded to attach two of our crossbraces, and made sure we boxed everything properly. We friction tested the drivetrain afterwards for testing purposes, and everything was perfect;y normal, spinning at 0.2W of friction per side.

Attached is a photo of our new drivetrain:





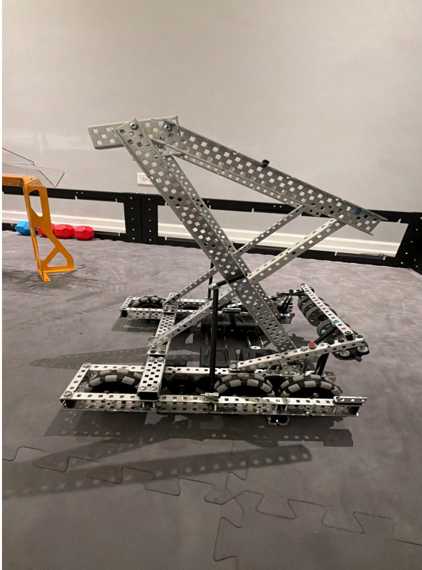
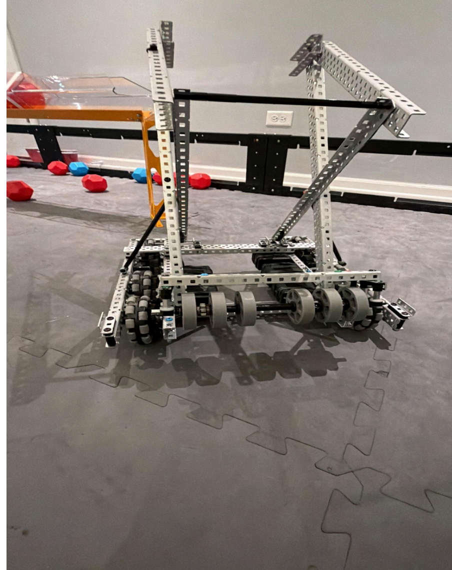
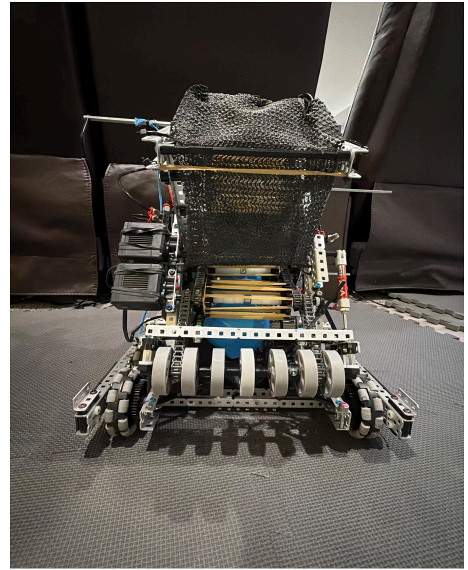
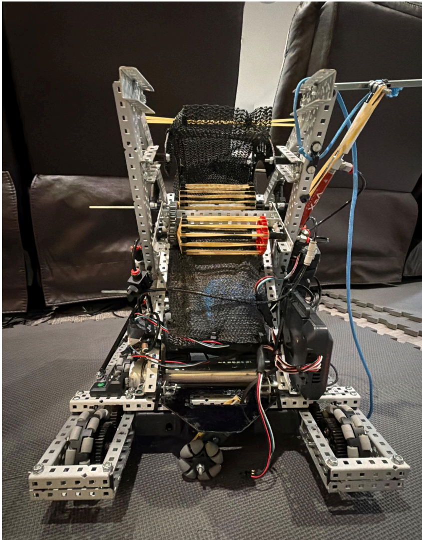
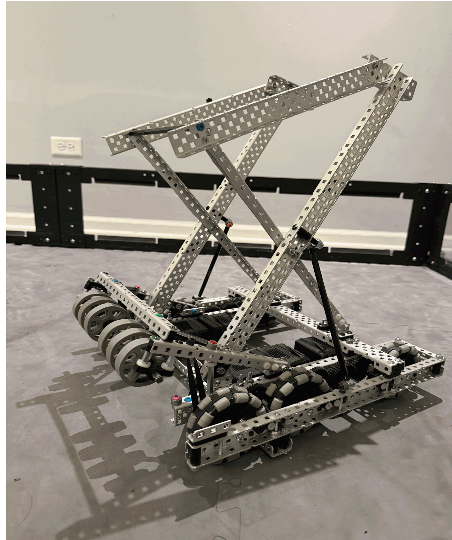
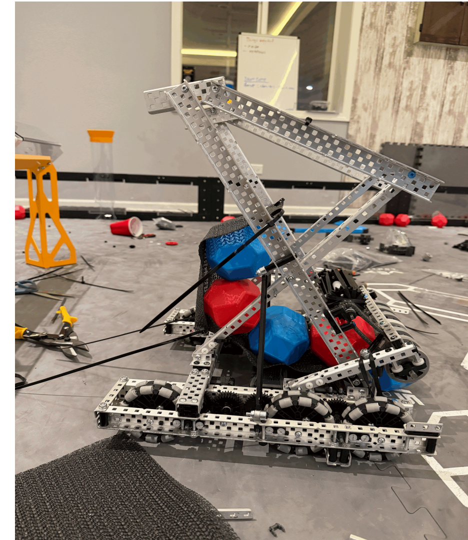
Was the goal reached? **YES**

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

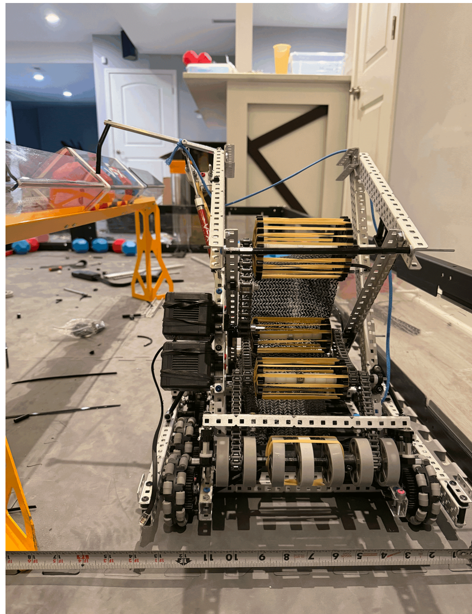
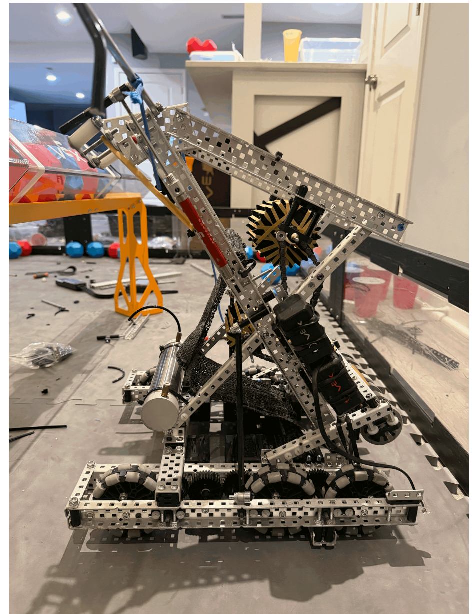
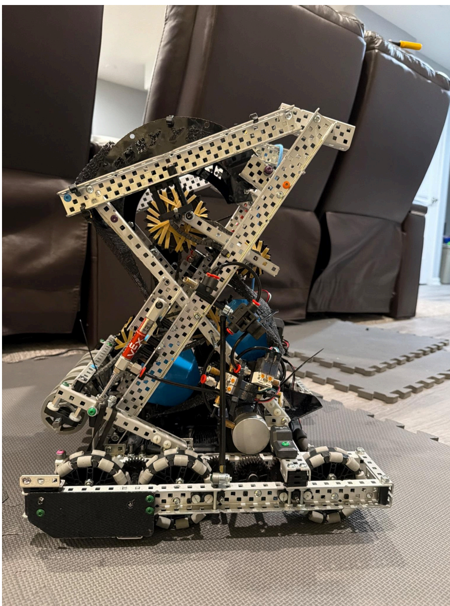
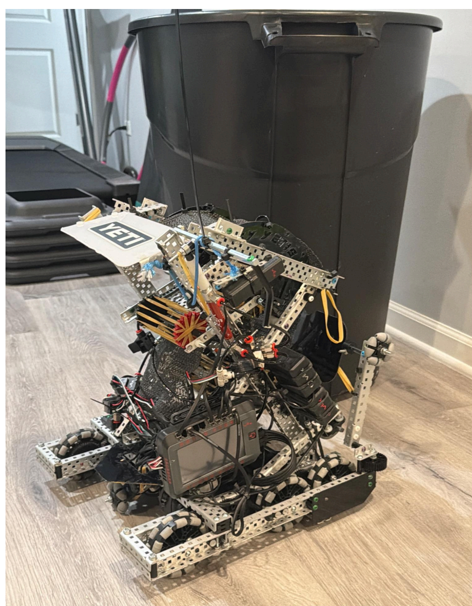
3/8/26 Rest of Robot Evolution Photos

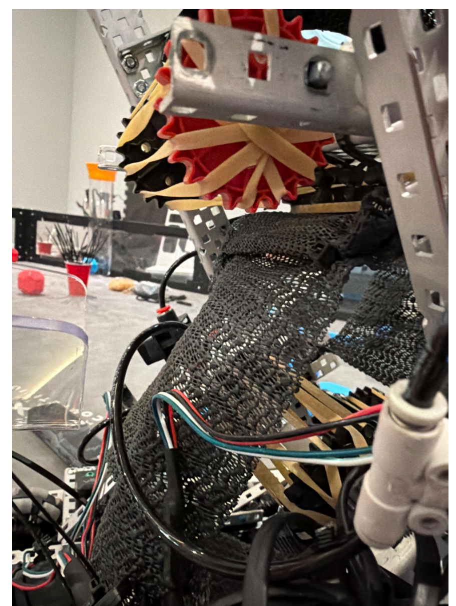
Build

We unfortunately did not have enough time to translate all of our meeting notes before the deadline for judging, but here is a photo of the rest of our robot's progress until its finish. Using the Engineering Design Process, heavy iteration, and strict testing, we have build a structurally sound robot with effective function in manipulating every aspect of push back for the upcoming **Illinois V5RC High School State Competition.**

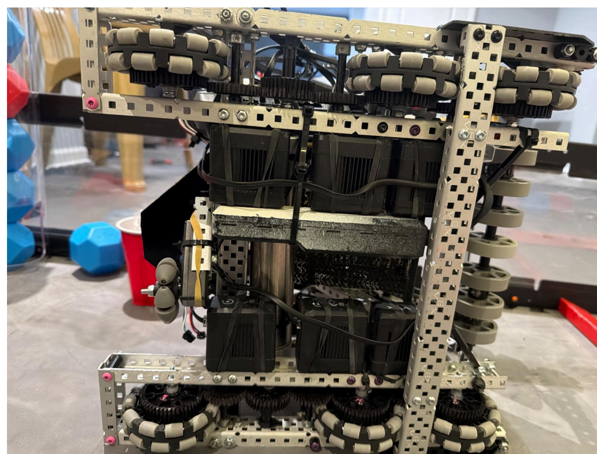
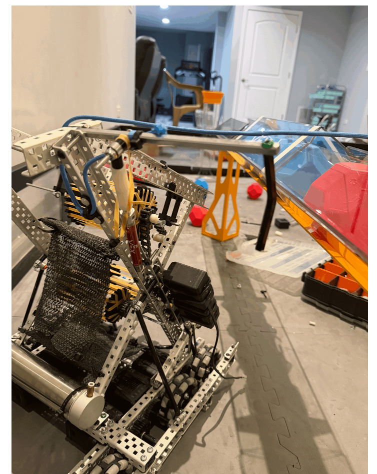







Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke



Crazy Drivetrain Optimization →

Written by: Sathvik Loke, Shaurya Yadav, Ediz Guzey, Shreyas Loke

Programming Notebook

We have dedicated the rest of our notebook to programming in order to better explain all of our iterations of our programming, specifically our Custom Library for Odometry and PID that we created.

Please take some time to read the the rest of our notebook, which we have called our specific "Programming Notebook"

355Y PROGRAMMING NOTEBOOK

PUSH BACK '25-'26

Written by: Sonit Sahoo, Ediz Guzey

Table of Contents

Table of Contents **145**
Glossary **148**
Why Custom Translate **149**
Pros vs Cons 149
LemLib: 149
Custom: 149
Codebase Philosophy **150**
Driver Control Driver 151
Objective 151
Control Scheme 151
Code Modularity 151
Voltage Curves 152
Cubic Control: 152
Turn Tuning 153
Odometry Overview Translate 155
Hardware Architecture Translate 155
The Mathematics of Odometry: Theory 157
Step 1: Calculating Distance (ΔS) 157
Step 2: Calculating Heading (θ) 157
Our Coordinate System 158
Local Displacement Vector: 159
The "Chord" Approximation: 159
Global Transformation: 161
Vector3D & Pose Custom 162
Autonomous Algorithms Theory **163**
PID Control Theory: 163
The Closed-Loop Feedback Concept: 164
PID Terms - The Proportional (P) 164
PID Terms - The Integral (I) 165
PID Terms - The Derivative (D) 166
PID Tuning Strategy 167
The Tuning Process: 167
Final Tuning Values: 167
Mathematical Summary: 168
Code Implementation Custom 168
TurnTo, Move, and PointTowards 169

Algorithm...169
Feedforward...169
Code Implementation Custom...170
MoveToPoint Custom...174
Algorithm...174
Code Implementation...174
MoveToPose Theory...179
Algorithm...179
Code Implementation Custom...180
Path Following Theory...186
Pure Pursuit vs RAMSETE...186
Pure Pursuit...186
RAMSETE...187
Algorithm...187
Code Implementation Custom...188
Adaptive Monte Carlo Localization Theory...194
Algorithm...194
Why is this needed...194
How we tested...195
Code Implementation...195
Sensor Integration - The Optical System...208
Data Normalization: HSV vs. RGB:...208
Finite State Machine (Sorting Logic):...209
Sorting Execution (Blue Alliance Example):...209
Signal Hysteresis:...210
Auton Manager...211
Why it Exists...211
Code Implementation Code...211
Solenoid Class Translate... 214
Why it Exists... 214
Code Implementation Code...214
Conclusion & Summary:...216
Autonomous Pathing Brainstorm ...218

Glossary

Below, you can find the labelling system for this notebook.

Label	Description
Translate	Thoughts on the implementation of various subsystems including compare and contrast.
Theory	Deeper insights into the mathematics and physics behind programming decisions.
Driver	Programming regarding driver-control.
Auton	Programming regarding autonomous control and function.
Custom	Custom software and programming beyond robot control, including autonomous path planning and vision.
Code	Live examples of code for various systems and versions thereof.

```
cout << "Let's begin." << endl;
```

Written by: Sonit Sahoo, Ediz Guzey

Why Custom

Translate

To start off the season, we had to decide whether to build our own custom library or to use an existing one like LemLib. Both options have their own advantages and disadvantages, and we had to weigh them carefully to determine which would be the best fit for our team.

Pros vs Cons

LemLib:

Pros	Cons
- LemLib is built specifically for autonomous motion and includes odometry, so it is much faster and easier to get competitive auton running - It is open source and actively maintained, so if any bugs emerge, they can be fixed. - There is a strong, preexisting ecosystem of project examples, documentations, and support channels. - Lots of the minute details, including motion helpers and certain constants, have already been figured out.	- We must work under someone else's design choices, so it may be difficult to debug issues or add new functionality. - LemLib is not completely plug-and-play - it still requires tuning on our end. - Changes in the API can quickly mess up an existing project. - Using prebuilt methods makes it difficult to learn and understand the core concepts.

Custom:

Pros	Cons
- We have full control over the architecture, abstractions, motion model, naming, and debugging tools. - Building our own odometry, controllers, and motion routines teaches a lot more about control systems and software design.	- It takes a lot more upfront time and can detract from other parts of the robot like building or design. - There's a higher chance of having subtle bugs which only appear in rare circumstances.

Written by: Sonit Sahoo, Ediz Guzey

- By being aware of the full codebase, it is much easier to find and fix issues.	- It requires more maintenance and upkeep of documentation in the long run.

Overall, we decided that the benefits of building our own custom library, **`vractolib`,** outweighed the drawbacks. We wanted to have full control over our codebase and learn as much as possible about control systems and software design. Additionally, it allows us to implement novel algorithms like Monte Carlo Localization. While it may take more time upfront, we believe that it will ultimately lead to a stronger and more competitive robot in the long run.

Codebase Philosophy

There are a few general rules that we kept in mind when creating this library:

1. **Everything in its own class**: We wanted to make sure that each component of the library was modular and could be easily modified by users. This means that we created separate classes for different functionalities, such as `drivetrain`, `solenoid`, `tracking_wheel`, each with easily adjustable parameters.
2. **Generalizable to all robots**: We wanted to make sure that our library could be used by any VEX team, regardless of their specific robot design. This means that we avoided hardcoding any values or assumptions about the robot's hardware, and instead made everything configurable through parameters.
3. **Understandable and modular for a beginner**: We wanted to make sure that our library was accessible to beginners who may not have a lot of programming experience. This means that we wrote clear and concise documentation, used simple language, and organized the code in a way that is logical.

Driver Control

Driver

Objective

Driver code, that is code used to control the robot during driver control, is one of the most important parts of a robot's codebase. However, it is also one of the most difficult parts to modularize due to how tailored it is. To solve this, we implemented quickswap drive controls via a `Drivetrain` class, simple loops for intakes, and a solenoid class that allows for easy control of solenoids.

Control Scheme

We utilize a "Arcade Drive" control scheme:

- **Left Stick:** Controls lateral movement
- **Right Stick:** Controls the turns.

Instead of requiring the driver to manually reverse the intake for sorting, the code handles it automatically.

- **L1 Button:** Toggles "Intake Mode" (Forward).
- **L2 Button:** Toggles "Outtake Mode" (Reverse).
  The driver simply sets the state, and the robot manages the motors as well as relevant pneumatics.

Code Modularity

The `Drivetrain` class allows users to supply motor groups which represent the left and right sides of the drivetrain. In the main loop, users can then call `Drivetrain::arcade`, `Drivetrain::tank`, or any custom drive control function that they create, and supply it with the controller object. This allows for quick swapping of drive controls without having to change any of the underlying code.

```
void Drivetrain::arcade(pros::Controller controller,
std::function<int(int)> mappedVolt) {
    int forward = controller.get_analog(pros::E_CONTROLLER_ANALOG_LEFT_Y);
    int turn = controller.get_analog(pros::E_CONTROLLER_ANALOG_LEFT_X);

    arcade_handle_input(forward, turn, mappedVolt);
}
```

The implementation of the arcade function

Written by: Sonit Sahoo, Ediz Guzey

```cpp
void Drivetrain::tank(pros::Controller controller, std::function<int(int)>
mappedVolt) {
    int leftY = controller.get_analog(pros::E_CONTROLLER_ANALOG_LEFT_Y);
    int rightY = controller.get_analog(pros::E_CONTROLLER_ANALOG_RIGHT_Y);

    if (abs(leftY) > vconfig::forwardDeadzone) {
        leftMotors.move_voltage(mappedVolt(leftY));
    } else {
        leftMotors.move_voltage(0);
    }

    if (abs(rightY) > vconfig::forwardDeadzone) {
        rightMotors.move_voltage(mappedVolt(rightY));
    } else {
        rightMotors.move_voltage(0);
    }
}
```

The implementation of the tank function

Voltage Curves

As you can see in the implementation of the arcade and tank functions, they both take in a `mappedVolt` function which maps the controller inputs to voltage values. This is needed because the controllers return values from -127 to 127, but the motors need voltage values from -12000 to 12000. By allowing users to supply their own mapping function, we allow for easy customization of the drive controls, such as implementing a deadzone or a custom voltage curve.

Cubic Control:

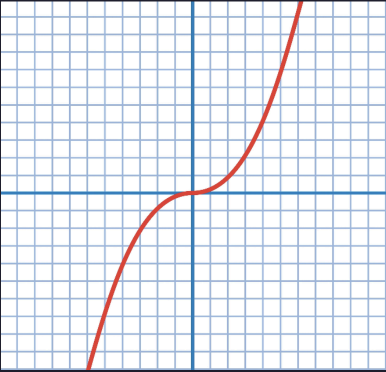
Linearity Problem:

A standard joystick map is linear: 50% stick = 50% power. This makes small adjustments (like aiming at a goal) very difficult, as the robot moves too fast.

The Solution:

We applied a **Cubic Function** (diagram below) to our inputs.

Code



The Effect:

- **Input 0.2 (20%):** 0.2^3 = 0.008 (0.8% Power). Ultra-fine precision.
- **Input 0.5 (50%):** 0.5^3 = 0.125 (12.5% Power). Soft control for maneuvering.
- **Input 1.0 (100%):** 1.0^3 = 1.0 (100% Power). Full speed available instantly.

This "Exponential Curve" creates a soft center for precision aiming while retaining maximum power at the edges.

```
int Drivetrain::linearVoltMap(int input) {
    // [-127, 127] -> [-12000, 12000]
    return std::min(input / 127.0 * 12000.0, vconfig::maxVolt * 1.0);
}
```

The implementation of the default linear voltage mapping function

Turn Tuning

There is additional tuning in the `arcade_handle_input` function, which is called by variations of the arcade function. We multiply the turn input by a multiplier to slow down turning, and allow for forwards/backwards movement while turning.

```
void Drivetrain::arcade_handle_input(int forward, int turn,
std::function<int(int)> mappedVolt) {
    if (abs(turn) > vconfig::turnDeadzone || abs(forward) >
```

```
vconfig::forwardDeadzone) {
        rightMotors.move_voltage(mappedVolt(0.67 * turn - forward));
        leftMotors.move_voltage(-mappedVolt(0.67 * turn + forward));
    } else {
        leftMotors.move_voltage(0);
        rightMotors.move_voltage(0);
    }
}
```

Some parts of the mapping is also handled by the `vconfig` namespace, which contains configuration variables for the robot. For example, the `forwardDeadzone` variable is used to determine the deadzone for the forward/backward movement of the drivetrain. By adjusting this variable, users can easily change the deadzone without having to modify any of the underlying code.

```
namespace vconfig {
    const int maxVel = 1.2 * maxPercent;
    const int maxVolt = 120 * maxPercent;
    const int forwardDeadzone = 10;
    const int turnDeadzone = 5;
}
```

Odometry Overview

Translate

Objective:

The primary objective of our programming for the "Push Back" season is to implement a robust positioning system that allows the robot to navigate the field using coordinate geometry rather than time-based dead reckoning. Our philosophy is **"Predictability over Raw Speed."** A robot that moves at 90% speed but hits its target 100% of the time is infinitely more valuable than a robot that moves at 100% speed but misses 20% of the time.

To achieve this, we moved away from standard time-based programming and adopted a **State-Space approach** utilizing Odometry for localization and PID (Proportional-Integral-Derivative) for closed-loop control.

The Problem with "Time-Based" Movement:

In standard drive code (e.g., drive_forward(2000ms)), the robot assumes that applying voltage for a set time results in a set distance. However, we identified three critical failure modes:

1. **Battery Voltage Variance:** As the battery drains from 100% to 80%, the voltage supplied to the motors drops. 2000ms at 12V moves the robot significantly further than 2000ms at 10V.
2. **Field Friction:** VEX foam tiles vary in friction. Old tiles are slippery; new tiles are grippy. Time-based code cannot adapt to these conditions.
3. **Defense & Slippage:** If the robot pushes a block or gets bumped by an opponent, the wheels spin but the robot doesn't move. The code has no way of knowing it hasn't reached the target.

The Solution: Odometry

Odometry is the practice of using sensors to track the robot's position relative to a starting point. We define our field as a Cartesian plane where the starting point is (0,0). By tracking the robot's physical location (X, Y, θ), we can command the robot to "Drive to point (24, 24)" regardless of battery level or field friction.

Written by: Sonit Sahoo, Ediz Guzey

Hardware Architecture

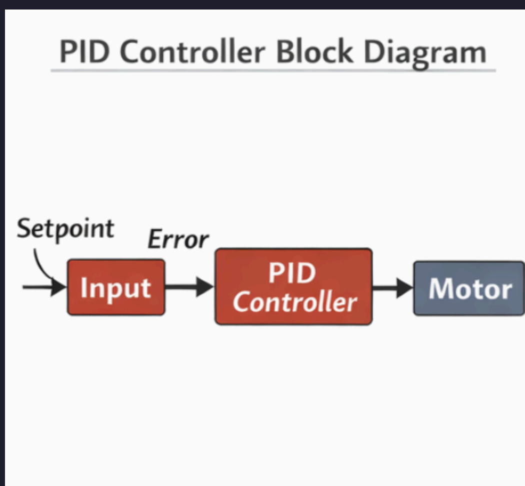
Translate

To implement Odometry, we utilize specific sensors designed to bypass the mechanical backlash of the drivetrain.

- **Tracking Wheels ("Dead Wheels"):**
  - We utilize unpowered omni-wheels tensioned to the ground using a custom hinge mechanism.
  - *Why?* When the robot accelerates rapidly, the powered drive wheels may slip. Tracking wheels are independent; they only spin if the robot actually moves across the tiles.
- **Sensors:**
  - **V5 Rotation Sensors:** These are keyed to the tracking wheels to measure rotation with high precision (0.08 degrees per tick).
  - **Inertial Sensor (IMU):** We use the V5 Inertial Sensor to track our absolute heading (θ). While tracking wheels can calculate rotation, the IMU does not suffer from "scrub" (lateral friction) errors during turns.

Signal Flow Diagram:

The below diagram explains the flow of signals, starting from our initial input in our autonomous code, and ending with rotation of a motor or piston.



Sensor Polling Rate: The V5 Brain updates motor ports every 10ms (100Hz). To ensure our math is accurate, our Odometry task runs in a dedicated thread at **priority level 15**, ensuring it is never interrupted by the driver control loop.

The Mathematics of Odometry:

Theory

The core of our tracking system is converting raw wheel rotation into (X, Y) coordinates. This process happens in three distinct steps: **Data Acquisition**, **Local Displacement**, and **Global Transformation**.

Step 1: Calculating Distance (ΔS)

First, we convert the raw sensor data (ticks) into linear distance traveled. We must know the physical circumference of our tracking wheels.

- **Wheel Diameter:** 2.75 inches
- **Ticks Per Rev:** 360

$$\Delta S = (\text{Ticks}_{\text{current}} - \text{Ticks}_{\text{previous}}) \times \frac{\pi \times \text{Diameter}}{\text{Ticks per Revolution}}$$

This gives us the change in distance for the Left (**ΔL**) and Right (**ΔL**) sensors.

Technical Specification: Using a 2.75" wheel with a 360-tick encoder provides a resolution of approximately **0.024 inches per tick**, allowing for sub-inch precision during high-speed maneuvers.

Step 2: Calculating Heading (θ)

We prioritize the IMU for heading data because it is absolute.

We update our global heading using the IMU or the differential between left and right tracking wheels. This can be represented by:

$$\theta_new = \theta_old + \Delta\theta$$

Written by: Sonit Sahoo, Ediz Guzey

While we primarily rely on the V5 Inertial Measurement Unit (IMU), we calculate a secondary heading based on the differential between the Left (**Δ**L) and Right (**Δ**R) tracking wheels to cross-reference for "drift." This approach is known as **Sensor Fusion**.

The change in heading (**Δ**θ) is calculated as:

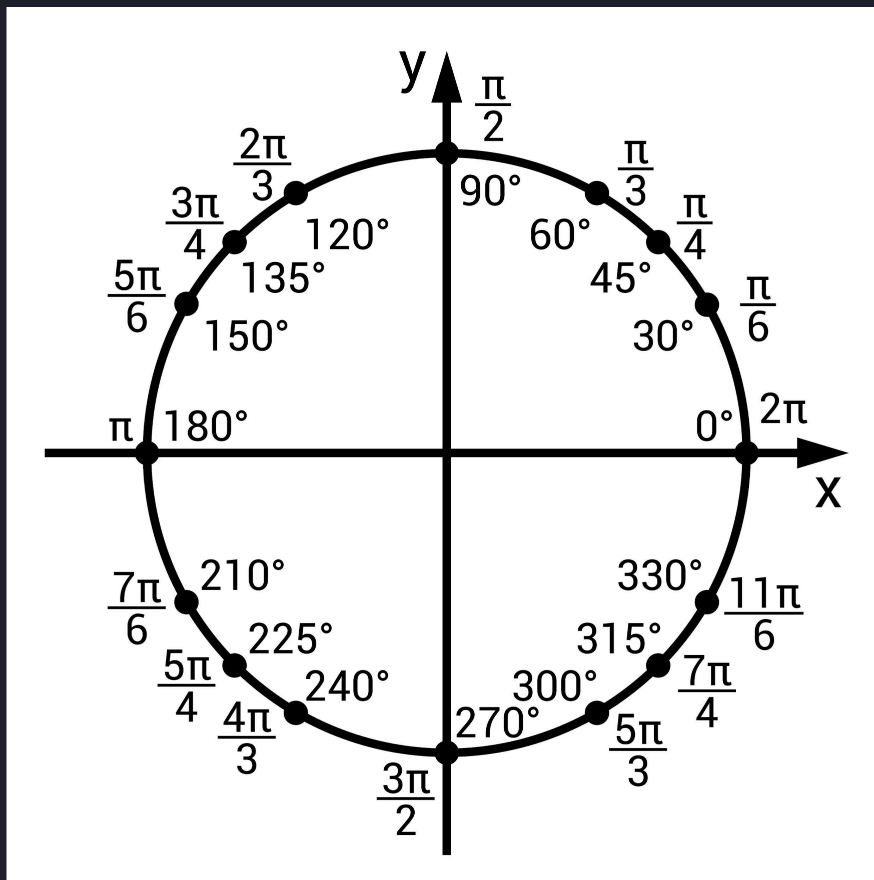
$$\Delta\theta = \frac{\Delta L - \Delta R}{S_L + S_R}$$

Where S_L and S_R are the distances from the tracking wheels to the center of the robot.

The new global heading is then updated.

Our Coordinate System

Our coordinate system places the origin at the center of the field with units in inches and radians. The positive x-axis points from the center to the blue alliance, and the positive y-axis points towards the long goal on the right of the blue side. Angles are measured counterclockwise from the positive x-axis, with 0 radians pointing towards the blue alliance. Furthermore, the angle is bounded between -π and π.




```
//wrap to [-\pi, \pi)
inline double wrapToSignedRadians(double angleRad) {
    angleRad = std::fmod(angleRad + PI, TAU);
    if (angleRad < 0) angleRad += TAU;
    return angleRad - PI;
}
```

Helper function to keep theta bounded

Local Displacement Vector:

At this stage, we know how much the wheels turned, but we need to know how the robot moved *relative to its own center.*

The following are three ways of identifying the robot's movement relative to its own center:

- If **Δ**L = **Δ**R, The robot moved in a perfectly straight line
- If **Δ**L = -**Δ**R, The robot rotated about its center point
- If **Δ**L ≠ **Δ**R. The robot is simultaneously translating and rotating (moving in an **Arc**)

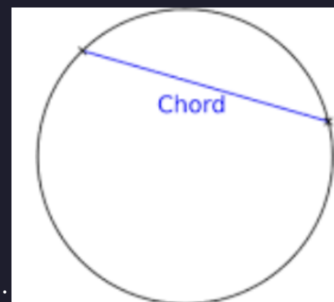
Because the majority of VEX movements are arcs, our math must prioritize calculating these curved paths to avoid "positional drift."

The "Chord" Approximation:

Calculating a true arc length using calculus in real-time can be computationally expensive. However, because our sensor loop refresh rate is so high (**100Hz / 10ms**), the arc traveled in a single cycle is microscopic.

We can therefore treat the arc as a **Chord** (a straight line connecting two points on a curve). This simplification maintains high accuracy while ensuring the V5 Brain can process the logic without lag.

To the right is a diagram of a **Chord** for visual interpretation:



Thus, the below formula, where **Δ**Local_Y represents the magnitude of the movement vector of the robot, allows us to compute the latest **"chord"** that the robot traversed.

$$\Delta\text{Local_Y} = \frac{\Delta L + \Delta R}{2}$$

Note on Lateral Movement: If a horizontal tracking wheel is present, we calculate **Δ** Local_Y to account for "drift" or "sideways slip" caused by collisions or high-speed turns.

Orientation for Integration:

When the robot moves and turns simultaneously, it travels along an arc. If we used only the starting angle (θ_old), we would consistently undershoot the curve; if we used the ending angle (θ_new), we would overshoot it.

To solve this, we utilize the **Average Orientation (alpha (α))**. By calculating the angle at the midpoint of the 10ms movement window, we can approximate the tangent of the arc, turning a complex curve into a manageable linear vector.

The below formula is how we calculate alpha:

$$\alpha = \theta_{old} + \frac{\Delta\theta}{2}$$

Global Transformation:

We now rotate the Local Displacement Vector by the Average Orientation (α) using standard trigonometry. This maps the robot-centric movement onto the field-centric grid.

Global Displacement components:

$$\Delta X = \Delta \text{Local_Y} \times \cos(\alpha)$$

$$\Delta Y = \Delta \text{Local_Y} \times \sin(\alpha)$$

Integration (Accumulation):

Because our tracking is continuous, we must "accumulate" these infinitesimal changes into our global position variables. This process is known as **Discrete Integration**. By summing these small steps every 10ms, we maintain a real-time "State Vector" of the robot's position.

Updated Global State:

$$X_{global} = X_{global} + \Delta X$$

$$Y_{global} = Y_{global} + \Delta Y$$

This calculation runs in a background task every 10ms, constantly integrating these tiny changes to update our global position vector [x, y, θ].

Vector3D & Pose

Custom

To keep track of position, we define a `Vector3D` class which represents a point in 3D space `(x, y, z)`. It contains standard vector operations such as addition, subtraction, and scaling. Derived from this, we have a `Pose` class that represents the robot's pose with `(x, y, theta)`. This contains additional helper functions for working with pose, such as calculating the angle between two poses, or applying a rotation to a pose.

```
Vector3D operator=(const Vector3D<T> &v1) {
    if (this != &v1) {
        x = v1.x;
        y = v1.y;
        z = v1.z;
    }
    return *this;
}
```

Equals implementation for Vector3D

```
Pose rotatedBy(const double &angle) const {
    double cosTheta = std::cos(angle);
    double sinTheta = std::sin(angle);
    double newX = cosTheta * x - sinTheta * y;
    double newY = sinTheta * x + cosTheta * y;
    return Pose(newX, newY, theta);
}
```

Rotating a pose by a given angle, used in transforming from local to global position

Autonomous Algorithms

Theory

Autonomous algorithms are essential for efficient and effective autonomous routines in robotics. They allow for precise control of the robot's movements, enabling it to navigate complex environments and complete tasks with high accuracy. By using well-designed algorithms, we can easily adjust and optimize our autonomous routines to achieve better performance. Simple functions allow us to create complex auton routines quickly.

PID Control Theory:

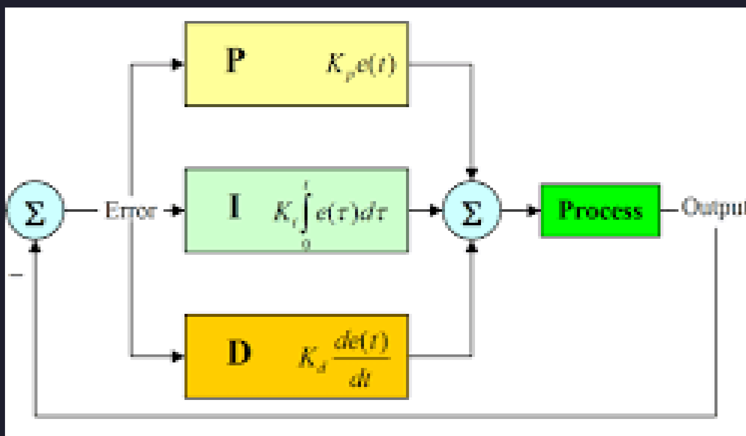
Objective:

Odometry tells us *where* we are. PID tells the motors *how* to get to where we want to be. The objective is to create smooth, accurate motion that corrects itself if disturbed by a block or an opponent.

Definition:

PID (Proportional-Integral-Derivative) is a closed-loop control feedback mechanism. It calculates an "error" value as the difference between a measured process variable (current position) and a desired setpoint (target position).

PID Control Diagram:



Mehta, Nikunj & Chauhan, Dharmendra & Patel, Sagarkumar & Mistry, Siddharth. (2017). Design of HMI Based on PID Control of Temperature. International Journal of Engineering Research and. V6. 10.17577/IJERTV6IS050074.

The Closed-Loop Feedback Concept:

A "Closed-Loop" system is one that constantly checks its own work. PID calculates the **Error**—the difference between where we are and where we want to be—and adjusts power in real-time to eliminate that gap.

The Error Function:

- **For Linear Movement:** $\text{Error} = \sqrt{(x_t - x_c)^2 + (y_t - y_c)^2}$ (Target minus Current)
- **For Rotational Movement:** $\text{Error} = \text{Target Angle} - \text{Current Angle}$

PID Terms - The Proportional (P)

We view PID as a way to look at the **Past**, **Present**, and **Future** of our robot's movement.

Concept: "The Present":

The P-term reacts to the current state of the robot. It provides the "raw power" needed to move toward a target.

Equation: $P_{\text{out}} = kP \times \text{Error}$

System Behavior:

- **Large Error:** When the robot is far from the target, P_out is large, causing the robot to drive fast.
- **Small Error:** As the robot gets closer, Error decreases, so P_out decreases, causing the robot to slow down.

Tuning Impact:

- If kP is too low: The robot will be sluggish and may not reach the target.
- If kP is too high: The robot will react too violently, overshoot the target, and potentially oscillate.

We will discuss our tuning process in more depth later, in a specific section of our notebook.

PID Terms - The Integral (I)

Concept: "The Past"

The Integral term accounts for accumulated error over time. It is used to fix "Steady-State Error."

Equation: $I_{\text{out}} = kI \times \sum(\text{Error} \times dt)$

Overcoming Static Friction:

If the robot stalls 0.5 inches from the target because the P-term isn't strong enough to overcome friction, the I-term begins to accumulate that "stuck" error. Eventually, the sum becomes large enough to give the motors the extra "nudge" needed to reach the target.

Safety Constraint: Integral Windup:

To prevent the robot from accelerating uncontrollably if it is physically blocked, we implement an **Integral Zone**. The I-term only begins accumulating when the robot is within 2 inches of its target.

PID Terms - The Derivative (D)

Concept: "The Future"

The Derivative term measures the *rate of change* of the error. It predicts where the robot will be in the next instant.

Equation: $D_{\text{out}} = kD \times \frac{\Delta\text{Error}}{\Delta t}$

.

Momentum Control: Our robot has high mass and significant momentum. The D-term detects when the robot is approaching the target too quickly and applies a counter-force, preventing overshooting and violent oscillations.

Importance for Heavy Robots:

Our robot is heavy. Once it gets moving, it has high momentum. The D-term is critical to prevent it from crashing through the target distance.

PID Tuning Strategy

Tuning is the process of finding the perfect "Weights" (kP, kI, kD) for our robot. Since friction varies by surface, we perform all tuning on official competition foam tiles.

The Tuning Process:

1. **Isolate Proportional:** Increase kP until the robot reaches the target but begins to "hunt" or oscillate back and forth.
2. **Add Damping:** Increase kD to "smooth out" the oscillations until the robot settles at the target in a single, fluid motion.
3. **Final Polish:** If the robot consistently stops 0.25" short, we introduce a very small kI to pull it to the final setpoint.

Final Tuning Values:

Tuning Log (Lateral Drive)

We performed iterative testing to arrive at our final constants.

Written by: Sonit Sahoo, Ediz Guzey

Iteration	kP	kI	kD	Result
1	0.5	0	0	Too slow. Didn't reach the target.
2	1.5	0	0	Fast, but violent oscillation at the end.
3	1.5	0	0.5	Better, but still some overshoot.
4	1.0	0	0.5	Good speed, slight overshoot.
5	**1.0**	**0**	**0.8**	**Perfect.** Stops on a dime.

Current Gains:

- **Lateral (Drive Straight):**
  - kP = 1.0: High enough for speed.
  - kD = 0.8: High damping required to stop the heavy chassis.
- **Turn (Rotate):**
  - kP = 1.0: Provides necessary torque to scrub wheels.
  - kD = 0.8: Prevents "whipping" past the target angle.

Mathematical Summary:

To help visualize how these three terms combine to create our final motor output (12.0V Max), we use the following summation:

$$\text{Output}_{total} = (kP \times E) + (kI \times \int E\,dt) + (kD \times \frac{dE}{dt})$$

Code Implementation

Custom

```
double PID::update(double err) {
    const uint32_t curTime = pros::millis();

    double dt = (curTime - lastTime) / 1000.0;
    lastTime = curTime;

    if (dt <= 0) {
        dt = 1e-6;
    }

    integral += err * dt;

    double deriv = 0.0;
    if (!starting) { // avoid large spike on first cycle
        deriv = (err - prevError) / dt;
    }

    prevError = err;
    starting = false;

    return gains.kP * err + gains.kI * integral + gains.kD * deriv;
}
```

Our implementation of the PID controller

TurnTo, Move, and PointTowards

Algorithm

The turnTo, move, and pointTowards functions are implemented very similarly. All of them follow the same basic process:

1. Calculate the error between the current state and the target state (angle for turnTo and pointTowards, distance for move).

2. Until reaching an acceptable error around the topic for a certain amount of time, or running for long enough without reaching the target, keep moving.
3. Use a PID controller to calculate the necessary output to reduce the error, and add feedforward to ensure the robot overcomes "stiction" (static friction) in the right direction even when the error is small.

Feedforward

Since the robot is a physical system with mass, it can not move under arbitrarily small PID outputs, as the force generated by the motors will be less than the force of friction. So, we add a feedforward term to ensure that the robot can overcome this "stiction" and start moving in the right direction as well as reach the precise target position even when the error is small.

```
int Drivetrain::applyFeedforwardFloor(int pidVolt, double err, double acceptableErr, double fullFloorErr, int feedforward, int minNearVolt, int nearDivisor) const {
    const int minFloorVolt = std::abs(feedforward) * 100;
    if (minFloorVolt <= 0 || nearDivisor <= 0 || fullFloorErr <= acceptableErr) return pidVolt;

    const double absErr = std::fabs(err);
    if (absErr <= acceptableErr || std::abs(pidVolt) >= minFloorVolt) return pidVolt;

    const int nearFloorVolt = std::max(minNearVolt, minFloorVolt / nearDivisor);
    const double rawScale = (absErr - acceptableErr) / (fullFloorErr - acceptableErr);
    const double floorScale = (rawScale < 0.0) ? 0.0 : ((rawScale > 1.0) ? 1.0 : rawScale);
    const int taperedFloor = static_cast<int>(std::lround(minFloorVolt * floorScale));
    const int appliedFloor = (absErr >= fullFloorErr) ? minFloorVolt : std::max(nearFloorVolt, taperedFloor);
    return (err > 0 ? 1 : -1) * appliedFloor;
```

```
}
```

Our feedforward function

As you can see, the feedforward is applied as a floor to the PID output, ensuring that the robot receives enough power to overcome stiction and start moving towards the target even when the error is small. The feedforward is scaled based on how close the robot is to the target, providing a smooth transition from low to high power as the error increases.

Code Implementation

Custom

```
void Drivetrain::turnTo(double angle, int timeout, int settleTime, int
maxVolt, int feedforward) {
    const double targetHeading =
vunits::wrapToSignedRadians(vunits::degToRad(angle));

    const double aErr = vunits::degToRad(0.5); // acceptable error
    const double minVoltErr = vunits::degToRad(2.0); // apply stiction
floor only when farther from target
    int settledTicks = 0;
    int elapsedTicks = 0;

    turnPID.reset();

    while (elapsedTicks * vconfig::updateRate < timeout && settledTicks *
vconfig::updateRate < settleTime) {
        const double curHeading = odom->getPose().theta;
        const double err = vunits::angleDiffRadians(curHeading,
targetHeading);

        const double pidOut = turnPID.update(err);
        int volt = applyFeedforwardFloor(
            static_cast<int>(std::lround(pidOut * 100.0)),
```

```
            err,
            aErr,
            minVoltErr,
            feedforward
        );

        if (volt > maxVolt) volt = maxVolt;
        if (volt < -maxVolt) volt = -maxVolt;

        rightMotors.move_voltage(volt);
        leftMotors.move_voltage(-volt);

        if (std::fabs(err) <= aErr && std::abs(volt) <= maxVolt * 0.25) {
            settledTicks += 1;
        } else {
            settledTicks = 0;
        }

        pros::delay(vconfig::updateRate);
        elapsedTicks += 1;
    }

    rightMotors.brake();
    leftMotors.brake();
}
```

Our turnTo function, pointTowards calculates the valid angle and then calls turnTo

```
void Drivetrain::move(double distance, int timeout, int settleTime, int
maxVolt, int feedforward) {
    const vunits::Pose startPose = odom->getPose();
    const double targetDist = distance;
```

```
    const double aErr = 0.25; // acceptable error
    const double minVoltErr = 4.0; // apply full floor at and beyond this
error

    int settledTicks = 0;
    int elapsedTicks = 0;

    latPID.reset();

    while (elapsedTicks * vconfig::updateRate < timeout && settledTicks *
vconfig::updateRate < settleTime) {
        const vunits::Pose curPose = odom->getPose();

        double curDist = vunits::Pose::distance(startPose, curPose);
        if (distance < 0) {
            curDist = -curDist;
        }

        const double err = targetDist - curDist;
        const double pidOut = latPID.update(err);

        int volt = applyFeedforwardFloor(
            static_cast<int>(std::lround(pidOut * 100.0)),
            err,
            aErr,
            minVoltErr,
            feedforward
        );
        if (volt > maxVolt) volt = maxVolt;
        if (volt < -maxVolt) volt = -maxVolt;

        rightMotors.move_voltage(volt);
        leftMotors.move_voltage(volt);
```

```
        if (std::fabs(err) <= aErr && std::abs(volt) <= maxVolt * 0.15) {
            settledTicks += 1;
        } else {
            settledTicks = 0;
        }

        pros::delay(vconfig::updateRate);
        elapsedTicks += 1;
    }

    rightMotors.brake();
    leftMotors.brake();
}
```

Our move function

As you can see, both functions are very similar, only having variations in error calculation.

MoveToPoint

Custom

Algorithm

`moveToPoint` is the first function where we are doing translation and turning at the same time. Instead of only caring about raw distance like `move`, it continuously looks at the robot's current pose and the target point, and recomputes what direction the robot should be traveling in.

The first step is to find the vector from the robot to the target point. From this, we can compute both the total distance to the point and the angle the robot should be facing to drive toward it. If we are driving backwards, we flip that target angle by 180 degrees so the robot intentionally approaches the point in reverse.

We then split the problem into 2 separate errors:

1. A lateral/forward distance error, which is the distance to the point projected onto the robot's current heading.
2. A heading error, which is how far the robot still has to turn to face the target direction.

This is important because if the robot is facing far away from the target point, we do not want to command full forward motion immediately. By projecting the distance using cosine, the robot

only drives hard when it is actually pointed mostly toward the goal. This prevents the robot from trying to take a wide sloppy arc when it should be turning first.

After calculating these 2 errors, we run the lateral PID on the distance error and the turn PID on the heading error. The lateral side still uses our feedforward floor for the same reason as `move`: we need enough voltage to overcome friction. The turn side does not directly use the same floor here, because while moving to a point we care more about smooth combined motion than about forcing a minimum turn voltage at every moment.

One of the most important details in this function is that the turning output is gradually faded out as the robot gets close to the target. If we keep turning aggressively all the way until the end, the robot tends to do a late heading snap right before settling. By scaling the turn PID down as distance decreases, we let the robot smoothly stop on the point instead of over-correcting.

Finally, the linear and turning commands are mixed together into left and right motor voltages. However, we prioritize turning authority by limiting how much lateral command can be used once turn voltage is already taking up part of the voltage budget. This keeps the robot responsive when it still needs to rotate significantly.

Code Implementation

```
void Drivetrain::moveToPoint(vunits::Pose pose, bool isBackwards, int
timeout, int settleTime, int maxVolt, int feedforward) {
    const double distAErr = 0.20;
    const double distFullFloorErr = 1.00;
    const double settleDistErr = 0.50;
    const double turnRampDist = 10.0;
    const double minTurnScale = 0.0;
    const double endTurnFadeDist = 3.0;
    const double turnDisableDist = 0.50;

    int settledTicks = 0;
    int elapsedTicks = 0;

    latPID.reset();
    turnPID.reset();

    while (elapsedTicks * vconfig::updateRate < timeout && settledTicks *
vconfig::updateRate < settleTime) {
```

```
        const vunits::Pose curPose = odom->getPose();

        const double dx = pose.x - curPose.x;
        const double dy = pose.y - curPose.y;
        const double dist = std::hypot(dx, dy);
        double targetAngle = std::atan2(dy, dx);

        if (isBackwards) {
            targetAngle = vunits::wrapToSignedRadians(targetAngle +
vunits::PI);
        }

        double distErr = dist *
std::cos(vunits::angleDiffRadians(curPose.theta, targetAngle));

        if (isBackwards) {
            distErr = -distErr;
        }

        double headErr = vunits::angleDiffRadians(curPose.theta,
targetAngle);

        const double latPIDOut = latPID.update(distErr);
        double turnPIDOut = turnPID.update(headErr);

        // gradually reduce turning as we approach target to prevent late
heading snap
        const double turnScaleRaw = dist / turnRampDist;
        const double turnScaleBase = std::clamp(turnScaleRaw,
minTurnScale, 1.0);
        const double endTurnScale = std::clamp(
            (dist - turnDisableDist) / std::max(1e-6, endTurnFadeDist -
turnDisableDist),
```

```
            0.0,
            1.0
        );
        const double turnScale = turnScaleBase * endTurnScale;
        turnPIDOut *= turnScale;

        const int latVolt = applyFeedforwardFloor(
            static_cast<int>(std::lround(latPIDOut * 100.0)),
            distErr,
            distAErr,
            distFullFloorErr,
            feedforward
        );
        int turnVolt = static_cast<int>(std::lround(turnPIDOut * 100.0));
        if (dist <= turnDisableDist) {
            turnVolt = 0;
        } else {
            const int maxNearTurnVolt =
static_cast<int>(std::lround(maxVolt * (0.25 + 0.75 * endTurnScale)));
            if (turnVolt > maxNearTurnVolt) turnVolt = maxNearTurnVolt;
            if (turnVolt < -maxNearTurnVolt) turnVolt = -maxNearTurnVolt;
        }
        double left = 0.0;
        double right = 0.0;
        mixLatTurnPrioritizeTurn(latVolt, turnVolt, maxVolt, left, right);
        const double max = std::max(std::fabs(left), std::fabs(right));

        rightMotors.move_voltage(static_cast<int>(right));
        leftMotors.move_voltage(static_cast<int>(left));

        if (dist <= settleDistErr && std::fabs(max) <= maxVolt * 0.15) {
            settledTicks += 1;
        } else {
```

```
            settledTicks = 0;
        }

        pros::delay(vconfig::updateRate);
        elapsedTicks += 1;
    }

    rightMotors.brake();
    leftMotors.brake();
}
```

Our movetoPoint function

Like the earlier functions, this still uses the same timeout/settle-time loop structure. The main difference is that the error is no longer one-dimensional. We repeatedly recompute both distance and heading based on the live odometry pose, which makes the function much more adaptive while the robot is moving.

```
const double dx = pose.x - curPose.x;
const double dy = pose.y - curPose.y;
const double dist = std::hypot(dx, dy);
double targetAngle = std::atan2(dy, dx);
```

This is the geometric core of the algorithm. `dx` and `dy` define the vector from the robot to the target point, `dist` gives the magnitude of that vector, and `targetAngle` gives the direction of travel.

```
double distErr = dist * std::cos(vunits::angleDiffRadians(curPose.theta,
targetAngle));
```

This line is especially important. Rather than using raw distance directly, we project the remaining distance onto the robot's current heading. If the robot is facing away from the target, this projected error gets smaller, which naturally reduces how hard the robot tries to drive forward before it has turned enough.

```cpp
const double turnScaleRaw = dist / turnRampDist;
const double turnScaleBase = std::clamp(turnScaleRaw, minTurnScale, 1.0);
const double endTurnScale = std::clamp(
    (dist - turnDisableDist) / std::max(1e-6, endTurnFadeDist -
turnDisableDist),
    0.0,
    1.0
);
const double turnScale = turnScaleBase * endTurnScale;
turnPIDOut *= turnScale;
```

This is what makes the controller feel smoother in practice. Far from the point, turning is allowed at full strength. Near the point, turn output is faded down, and very close to the target it is disabled entirely. This reduces oscillation and helps the robot settle more cleanly.

```cpp
static void mixLatTurnPrioritizeTurn(double latCmd, double turnCmd, int
maxVolt, double &leftOut, double &rightOut) {
    const double turn = std::clamp(turnCmd, -static_cast<double>(maxVolt),
static_cast<double>(maxVolt));
    const double maxLat = std::max(0.0, static_cast<double>(maxVolt) -
std::fabs(turn));
    const double lat = std::clamp(latCmd, -maxLat, maxLat);
    leftOut = lat - turn;
    rightOut = lat + turn;
}
```

Instead of just adding the two outputs blindly, we mix them while preserving turn authority. If turning already uses most of the available voltage, the lateral command is clipped to fit the remaining budget. That means the robot can still correct its heading instead of saturating both sides and pushing through the turn.

The settle condition is also intentionally based only on distance and output magnitude, not final heading. This makes sense because `moveToPoint` is intended to get the robot to a location, not to a fully specified pose. If we also required a final exact heading here, it would start behaving more like `moveToPose`.

MoveToPose

Theory

Algorithm

While `moveToPoint` only cares about getting the robot to a location, `moveToPose` cares about both position and final heading. This is a much harder problem, because if the robot drives straight at the final point and only tries to fix its heading at the very end, it often arrives awkwardly and needs a large correction. To solve this, we use a boomerang-style controller.

The main idea of a boomerang controller is that instead of driving directly at the final pose the entire time, the robot aims for a temporary target point called a **carrot point**. This carrot point lies along the final heading of the target pose, not necessarily at the final pose itself. As the robot gets closer, the carrot point moves, so the robot follows a smoother path that naturally lines it up for the final orientation.

The carrot point is placed some distance away from the target pose along the target heading:

1. When moving forward, the carrot point is placed behind the final pose.
2. When moving backward, the carrot point is placed in front of the final pose.

This means the robot is not just trying to hit the endpoint. It is trying to approach that endpoint from the correct direction. That is what makes the motion feel much cleaner than a plain point-to-point controller.

The `lead` value directly controls how far away this carrot point is placed. Internally, the lookahead distance is chosen as a fraction of the remaining distance to the target, then clamped to a maximum value. So:

1. A larger `lead` value places the carrot point farther away, making the path smoother and more gradual.
2. A smaller `lead` value keeps the carrot point closer to the final pose, making the robot drive more directly and aggressively.

In practice, a higher lead usually gives a smoother arc and better approach angle, while a lower lead makes the controller feel tighter but can also make it more abrupt. So `lead` is one of the main tuning values for how "curved" the boomerang path feels.

From there, the controller works similarly to `moveToPoint`. It computes the angle from the robot to the carrot point, projects the distance error onto that direction, and runs both lateral and turning PID. The turning output is still scaled down as the robot gets close, since aggressive turning near the end tends to cause oscillation.

Once the robot is close enough to the target position, the controller switches modes. At that point it stops trying to chase the carrot point and instead performs an in-place heading correction to match the target pose's final angle exactly. This two-stage structure is what makes `moveToPose` much more reliable than trying to solve both position and orientation with one behavior all the way through.

Code Implementation

Custom

```
void Drivetrain::moveToPose(vunits::Pose pose, double lead, bool
isBackwards, int timeout, int settleTime, int maxVolt, int feedforward) {
    const double distAErr = 0.20;
    const double distFullFloorErr = 1.00;
    const double settleDistErr = 0.25;
    const double settleHeadErr = vunits::degToRad(3.0);
    const double turnRampDist = 10.0;
    const double minTurnScale = 0.20;
    const double headingAErr = vunits::degToRad(1.0);
    const double headingFullFloorErr = vunits::degToRad(4.0);
    const double maxLookahead = 18.0;
    const double turnFadeStartDist = 6.0;
    const double turnDisableDist = 0.60;
    const double finishTurnMaxFrac = 0.30;

    lead = std::clamp(lead, 0.0, 1.0);
```

```
    int settledTicks = 0;
    int elapsedTicks = 0;

    latPID.reset();
    turnPID.reset();

    while (elapsedTicks * vconfig::updateRate < timeout && settledTicks *
vconfig::updateRate < settleTime) {
        const vunits::Pose curPose = odom->getPose();

        const double dx = pose.x - curPose.x;
        const double dy = pose.y - curPose.y;
        const double dist = std::hypot(dx, dy);
        const double finalHeadErr =
vunits::angleDiffRadians(curPose.theta, pose.theta);

        const bool atPosition = dist <= settleDistErr;
        const bool atHeading = std::fabs(finalHeadErr) <= settleHeadErr;
        const double endTurnScale = std::clamp(
            (dist - turnDisableDist) / std::max(1e-6, turnFadeStartDist -
turnDisableDist),
            0.0,
            1.0
        );

        double left = 0.0;
        double right = 0.0;

        if (!atPosition) {
            //target a point along final heading:
            //forward: behind final pose, backward: ahead of final pose.
            const double hX = std::cos(pose.theta);
```

```
            const double hY = std::sin(pose.theta);
            const double lookahead = std::clamp(lead * dist, 0.0,
maxLookahead);

            const double carrotSign = isBackwards ? 1.0 : -1.0;
            double carrotX = pose.x + carrotSign * hX * lookahead;
            double carrotY = pose.y + carrotSign * hY * lookahead;
            if (dist <= 1e-6) {
                carrotX = pose.x;
                carrotY = pose.y;
            }

            const double travelAngle = std::atan2(carrotY - curPose.y,
carrotX - curPose.x);
            const double targetAngle = isBackwards
                ? vunits::wrapToSignedRadians(travelAngle + vunits::PI)
                : travelAngle;
            const double headErr = vunits::angleDiffRadians(curPose.theta,
targetAngle);
            double driveErr = dist * std::cos(headErr);
            if (isBackwards) {
                driveErr = -driveErr;
            }

            const double latPIDOut = latPID.update(driveErr);
            double turnPIDOut = turnPID.update(headErr);

            const double turnScale = std::clamp(dist / turnRampDist,
minTurnScale, 1.0) * endTurnScale;
            turnPIDOut *= turnScale;

            const int latVolt = applyFeedforwardFloor(
                static_cast<int>(std::lround(latPIDOut * 100.0)),
```

```
                driveErr,
                distAErr,
                distFullFloorErr,
                feedforward
            );
            int turnVolt = static_cast<int>(std::lround(turnPIDOut *
100.0));
            const int movingTurnCap = static_cast<int>(std::lround(
                maxVolt * (finishTurnMaxFrac + (1.0 - finishTurnMaxFrac) *
endTurnScale)
            ));
            if (turnVolt > movingTurnCap) turnVolt = movingTurnCap;
            if (turnVolt < -movingTurnCap) turnVolt = -movingTurnCap;
            if (dist <= turnDisableDist) turnVolt = 0;

            mixLatTurnPrioritizeTurn(latVolt, turnVolt, maxVolt, left,
right);
        } else {
            //if position is good, finish heading in place
            const double turnPIDOut = turnPID.update(finalHeadErr);
            int turnVolt = applyFeedforwardFloor(
                static_cast<int>(std::lround(turnPIDOut * 100.0)),
                finalHeadErr,
                headingAErr,
                headingFullFloorErr,
                feedforward
            );
            const int finishTurnCap = static_cast<int>(std::lround(maxVolt
* finishTurnMaxFrac));
            if (turnVolt > finishTurnCap) turnVolt = finishTurnCap;
            if (turnVolt < -finishTurnCap) turnVolt = -finishTurnCap;
            if (std::fabs(finalHeadErr) <= settleHeadErr) turnVolt = 0;
            left = -turnVolt;
```

```cpp
            right = turnVolt;
        }

        const double max = std::max(std::fabs(left), std::fabs(right));

        rightMotors.move_voltage(static_cast<int>(right));
        leftMotors.move_voltage(static_cast<int>(left));

        if (atPosition && atHeading && max <= maxVolt * 0.15) {
            settledTicks += 1;
        } else {
            settledTicks = 0;
        }

        pros::delay(vconfig::updateRate);
        elapsedTicks += 1;
    }

    rightMotors.brake();
    leftMotors.brake();
}
```

Our movetoPose function

The first major difference from `moveToPoint` is the carrot point construction:

```cpp
const double hX = std::cos(pose.theta);
const double hY = std::sin(pose.theta);
const double lookahead = std::clamp(lead * dist, 0.0, maxLookahead);

const double carrotSign = isBackwards ? 1.0 : -1.0;
double carrotX = pose.x + carrotSign * hX * lookahead;
double carrotY = pose.y + carrotSign * hY * lookahead;
```

This takes the target pose heading and builds a point somewhere along that heading line. The `lookahead` value is determined by `lead * dist`, so the carrot point automatically moves farther away when the robot is far from the target and collapses inward as the robot approaches. That is the core of the boomerang controller.

Because `lead` is multiplied by the remaining distance, it scales naturally with how far the robot still has to go:

1. If `lead` is larger, the carrot point stays farther from the final pose for longer, so the robot follows a more sweeping path.
2. If `lead` is smaller, the carrot point stays closer to the final pose, so the robot cuts in more directly.

```
const double travelAngle = std::atan2(carrotY - curPose.y, carrotX -
curPose.x);
const double targetAngle = isBackwards
    ? vunits::wrapToSignedRadians(travelAngle + vunits::PI)
    : travelAngle;
const double headErr = vunits::angleDiffRadians(curPose.theta,
targetAngle);
double driveErr = dist * std::cos(headErr);
```

Once the carrot point exists, the robot treats it similarly to a temporary target point. It computes the angle toward the carrot, computes heading error relative to that angle, and then projects the remaining distance onto that direction. So even though the final goal is a pose, the robot is really following a moving intermediate target until it gets close enough.

```
if (!atPosition) {
    ...
} else {
    //if position is good, finish heading in place
    ...
}
```

This mode switch is what makes the controller robust. During most of the movement, the robot is following the boomerang path defined by the carrot point. Once the position is good enough, it

stops caring about path shape and just rotates in place to clean up the final heading error. That separation avoids a lot of endgame instability.

```
const int movingTurnCap = static_cast<int>(std::lround(
    maxVolt * (finishTurnMaxFrac + (1.0 - finishTurnMaxFrac) * endTurnScale)
));
```

While the robot is still translating, turning is capped and faded near the end so it does not whip around too hard while approaching the final location. Then in the final heading-only stage, the controller uses a smaller capped in-place turn with feedforward to overcome stiction and settle precisely.

So overall, `moveToPose` can be summarized as:

1. Use a boomerang controller with a carrot point to approach the pose from the correct direction.
2. Let the `lead` value control how far ahead that carrot point sits, which controls how curved or direct the path is.
3. Once the robot is close enough in position, switch to a final in-place heading correction.

Path Following

Theory

Pure Pursuit vs RAMSETE

Path following is a more advanced form of autonomous control where the robot is given a full path to follow, rather than just a single target point or pose. The robot then continuously looks at the path and tries to drive along it as closely as possible. This allows for much more complex and flexible autonomous routines, as the robot can navigate through multiple waypoints and adjust its trajectory on the fly.

There are two main methods of implementing it. Pure Pursuit is a path following algorithm that works by constantly looking ahead along a predefined path and trying to drive towards a point on that path, while RAMSETE is a nonlinear feedback controller that drives a robot towards an end pose along a precalculated trajectory.

Pure Pursuit

Pros	Cons
- Pure Pursuit is much simpler to implement, and does not require as much computation.	- A lookahead point that is too close can cause oscillation, while a lookahead point that is too

- Since it does not rely on precise trajectory generation, it can be more robust to changes in the robot's performance or the environment.	far can cause the robot to cut corners and deviate from the path. - Pure Pursuit does not explicitly account for the robot's dynamics, which can lead to less optimal paths and slower convergence to the desired trajectory. - It is difficult to tune timing.

RAMSETE

Pros	Cons
- RAMSETE is designed to account for the robot's dynamics, which can lead to more optimal paths and faster convergence to the desired trajectory. - It can handle more complex trajectories, including those with varying speeds and curvatures.	- RAMSETE is more complex to implement and requires more computational resources, which can be a challenge for real-time control on a robot. - It relies heavily on accurate trajectory generation, which can be difficult to achieve in practice, especially in dynamic environments or with imperfect robot models.

We chose Pure Pursuit for its simplicity and robustness, as well as the fact that it is easier to implement and tune for our specific use case. While RAMSETE can provide better performance in some cases, the added complexity and computational requirements were not justified for our needs.

Algorithm

Pure Pursuit works by replacing the idea of "drive to the final point right now" with "drive to a point a little farther ahead on the path." At every update, the robot looks at the path, finds a **lookahead point** on that path, and then drives toward that point. As the robot moves, the lookahead point also moves, so the robot gets pulled along the path instead of only aiming at the very end.

The most important part of the algorithm is choosing that lookahead point. Conceptually, we draw a circle centered on the robot with radius equal to the lookahead distance. We then find where that circle intersects the path. The most useful intersection is usually the one farther along the path, because that point keeps the robot moving forward through the route instead of snapping backward to an earlier segment.

Once we have the lookahead point, the control side becomes much more straightforward. We compute the vector from the robot to that point, compute the angle to it, and then command the drivetrain to turn and drive toward it. So in a lot of ways, Pure Pursuit can be thought of as repeatedly running a "smart moveToPoint" where the target is not fixed, but slides along the path.

The lookahead distance is one of the most important tuning values:

1. A smaller lookahead keeps the robot very close to the path, but can make it oscillate and turn too aggressively.
2. A larger lookahead smooths everything out, but can cause the robot to cut corners and miss tight geometry.

That tradeoff is the core of Pure Pursuit. The controller is simple and effective, but it does not directly model timing or robot dynamics. Instead, it relies on choosing a good moving target and trusting the low-level drivetrain controller to chase it smoothly.

Code Implementation

Custom

Our pure pursuit uses the same odometry-based control loop, the same PID objects, and the same voltage mixing/feedforward ideas as the rest of the library.

```
struct PathPoint {
    double x;
    double y;
};

static bool findLookaheadPoint(
    const std::vector<PathPoint>& path,
    const vunits::Pose& robotPose,
    double lookaheadDist,
    PathPoint& outPoint
) {
    bool found = false;
    double bestProgress = -1.0;

    for (size_t i = 0; i + 1 < path.size(); i++) {
        const double x1 = path[i].x - robotPose.x;
        const double y1 = path[i].y - robotPose.y;
        const double x2 = path[i + 1].x - robotPose.x;
        const double y2 = path[i + 1].y - robotPose.y;
```

```
        const double dx = x2 - x1;
        const double dy = y2 - y1;

        const double a = dx * dx + dy * dy;
        const double b = 2.0 * (x1 * dx + y1 * dy);
        const double c = x1 * x1 + y1 * y1 - lookaheadDist *
lookaheadDist;

        const double discriminant = b * b - 4.0 * a * c;
        if (discriminant < 0.0 || a <= 1e-6) {
            continue;
        }

        const double sqrtDisc = std::sqrt(discriminant);
        const double t1 = (-b - sqrtDisc) / (2.0 * a);
        const double t2 = (-b + sqrtDisc) / (2.0 * a);

        auto consider = [&](double t) {
            if (t < 0.0 || t > 1.0) return;

            const double px = path[i].x + dx * t;
            const double py = path[i].y + dy * t;
            const double progress = static_cast<double>(i) + t;

            if (!found || progress > bestProgress) {
                found = true;
                bestProgress = progress;
                outPoint = {px, py};
            }
        };

        consider(t1);
```

```cpp
        consider(t2);
    }

    if (!found && !path.empty()) {
        outPoint = path.back();
    }

    return found;
}

void Drivetrain::followPurePursuit(
    const std::vector<PathPoint>& path,
    double lookaheadDist,
    int timeout,
    int settleTime,
    int maxVolt,
    int feedforward
) {
    if (path.empty()) return;

    const double distAErr = 0.30;
    const double distFullFloorErr = 1.50;
    const double settleDistErr = 0.75;
    const double turnRampDist = 12.0;

    int settledTicks = 0;
    int elapsedTicks = 0;

    latPID.reset();
    turnPID.reset();

    while (elapsedTicks * vconfig::updateRate < timeout &&
           settledTicks * vconfig::updateRate < settleTime) {
```

```
        const vunits::Pose curPose = odom->getPose();
        const PathPoint finalPoint = path.back();

        PathPoint lookahead = finalPoint;
        findLookaheadPoint(path, curPose, lookaheadDist, lookahead);

        const double dx = lookahead.x - curPose.x;
        const double dy = lookahead.y - curPose.y;
        const double lookaheadAngle = std::atan2(dy, dx);
        const double lookaheadDistNow = std::hypot(dx, dy);
        const double headErr = vunits::angleDiffRadians(curPose.theta,
lookaheadAngle);
        const double driveErr = lookaheadDistNow * std::cos(headErr);

        const double finalDist = std::hypot(finalPoint.x - curPose.x,
finalPoint.y - curPose.y);

        const double latPIDOut = latPID.update(driveErr);
        double turnPIDOut = turnPID.update(headErr);

        const double turnScale = std::clamp(finalDist / turnRampDist,
0.15, 1.0);
        turnPIDOut *= turnScale;

        const int latVolt = applyFeedforwardFloor(
            static_cast<int>(std::lround(latPIDOut * 100.0)),
            driveErr,
            distAErr,
            distFullFloorErr,
            feedforward
        );
        int turnVolt = static_cast<int>(std::lround(turnPIDOut * 100.0));
        if (turnVolt > maxVolt) turnVolt = maxVolt;
```

```cpp
        if (turnVolt < -maxVolt) turnVolt = -maxVolt;

        double left = 0.0;
        double right = 0.0;
        mixLatTurnPrioritizeTurn(latVolt, turnVolt, maxVolt, left, right);
        const double maxOut = std::max(std::fabs(left), std::fabs(right));

        leftMotors.move_voltage(static_cast<int>(left));
        rightMotors.move_voltage(static_cast<int>(right));

        if (finalDist <= settleDistErr && maxOut <= maxVolt * 0.15) {
            settledTicks += 1;
        } else {
            settledTicks = 0;
        }

        pros::delay(vconfig::updateRate);
        elapsedTicks += 1;
    }

    leftMotors.brake();
    rightMotors.brake();
}
```

Our Pure Pursuit function

```cpp
const double a = dx * dx + dy * dy;
const double b = 2.0 * (x1 * dx + y1 * dy);
const double c = x1 * x1 + y1 * y1 - lookaheadDist * lookaheadDist;
```

This is the standard line-circle intersection setup. Each path segment is treated like a line segment, and the robot-centered lookahead circle is intersected with that segment. Solving this quadratic gives possible points where the path crosses the lookahead radius.

```
const double t1 = (-b - sqrtDisc) / (2.0 * a);
const double t2 = (-b + sqrtDisc) / (2.0 * a);
```

These are the 2 possible intersection parameters on the segment. If either value lies between `0` and `1`, that means the intersection is actually on the segment rather than somewhere on the infinite extension of the line.

```
if (!found || progress > bestProgress) {
    found = true;
    bestProgress = progress;
    outPoint = {px, py};
}
```

If multiple intersections are valid, we keep the one farther along the path. That makes the robot continue advancing through the route instead of chasing an earlier point and getting stuck.

```
const double lookaheadAngle = std::atan2(dy, dx);
const double headErr = vunits::angleDiffRadians(curPose.theta,
lookaheadAngle);
const double driveErr = lookaheadDistNow * std::cos(headErr);
```

Once the lookahead point is found, the rest of the controller looks very similar to `moveToPoint`. The robot turns toward the lookahead point and projects distance error along its heading so it does not command excessive forward motion while facing the wrong way.

So the overall Pure Pursuit loop is:

1. Find the current lookahead point by intersecting the lookahead circle with the path.
2. Choose the intersection farther along the path.
3. Drive toward that moving point using the same kinds of drivetrain controls we already use elsewhere.
4. Stop once the robot is close enough to the final path point and the output has settled.

Adaptive Monte Carlo Localization

Theory

Algorithm

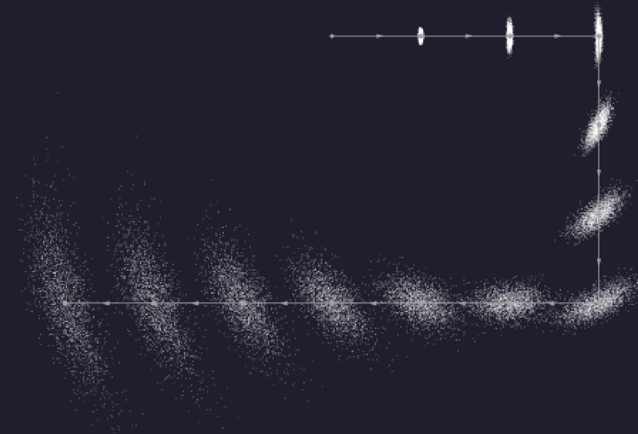
Adaptive Monte Carlo Localization, or AMCL, is a particle-filter based localization algorithm. Instead of assuming the robot is in exactly one place, AMCL keeps many possible robot poses at once. Each of those possible poses is represented by a particle. As the robot moves and takes sensor measurements, some particles become more believable and some become less believable. Over time, the particle cloud converges around the robot's true pose.

The easiest way to think about it is this:

1. Start with a cloud of guesses for where the robot might be.
2. Move all of those guesses forward using odometry.
3. Compare what each guess would "see" using the distance sensors to what the real robot actually sees.
4. Give higher weight to the guesses that match reality better.
5. Throw away bad guesses and keep / copy good ones.
6. Repeat this many times per second until the cloud tightens around the true pose.

So if the robot is somewhere on the field and the front distance sensor sees a wall 10 inches away while another sensor sees a goal edge a certain distance away, only some poses on the field are consistent with that combination of measurements. The particles near those poses get high weight, and particles in impossible places get low weight. That is what lets MCL recover from drift or uncertainty in a way that plain odometry cannot.

The reason this version is **adaptive** is that it does not always keep the exact same number of particles. When the belief is spread out and uncertain, it can keep more particles. When the belief has converged tightly, it can get away with fewer. That matters a lot on VEX hardware, where compute is limited and particle count directly affects runtime.



Belief after moving several steps for a 2D robot using a typical MCL implementation.

Written by: Sonit Sahoo, Ediz Guzey

Why is this needed

Odometry is still the main source of short-term motion tracking in our system. It is fast, smooth, and always available. The problem is that odometry accumulates error, especially when the robot goes through situations where the wheels or tracking system do not behave ideally. Going over the parking barrier is a good example of this, because it creates exactly the kind of disturbance that can throw off a pure dead-wheel based estimate.

A simple distance sensor reset can sometimes help, but it is much more limited. A reset usually assumes:

1. We know exactly which wall or object the sensor is looking at.
2. One measurement is enough to recover the pose.
3. The measurement should directly override the odometry estimate.

That can work in controlled situations, but it is fragile. AMCL is stronger because it does not treat localization as a one-shot correction. Instead, it continuously fuses odometry with multiple sensor measurements and a known field map. That means it can recover more gracefully, handle ambiguity better, and avoid hard-snapping the robot to a possibly incorrect pose from one noisy reading.

In other words, odometry tells us how the robot thinks it moved, while AMCL checks whether that motion still makes sense relative to the actual field. That combination is much more robust than either one by itself.

How we tested

Because Monte Carlo localization has so many moving parts, it is much harder to iterate directly on the robot than something like a simple PID controller. Particle count, sensor placement, confidence thresholds, map layout, motion noise, and measurement noise all interact with each other. If we tried to tune all of that only by flashing code and driving the robot, iteration would be far too slow.

That is why we built **https://mcl.vracto.xyz/test_vjeoivi.html**. The website lets us test the general algorithm and different sensor configurations much more quickly than we could on the robot itself. It gives us a way to visualize the particle cloud, see whether it actually converges, and identify cases where it converges too slowly or incorrectly. It also lets us experiment with things like:

1. How many particles are needed.
2. Where distance sensors should be mounted.
3. How noisy motion and measurements should be modeled.
4. Whether the filter is converging stably or just collapsing by accident.

This was especially important because the VEX brain has limited compute. A configuration that is mathematically reasonable is not automatically practical if it is too expensive to run. The site let us test both correctness and performance before relying on the algorithm on the robot.

Code Implementation

In our code, the MCL system is built out of 3 main pieces:

1. `DistanceSensor`, which wraps a physical distance sensor and keeps track of where it is mounted on the robot.
2. `MCLMap`, which stores a simplified geometric map of the field and can ray-cast against it.
3. `MCL`, which owns the particle set and performs the actual particle filter update.

The setup in `main.cpp` looks like this:

```
Distance dist1(1);
Distance dist2(2);
Distance dist3(3);
Distance dist4(4);

const vunits::Pose localizationStartPose{0.0, 0.0, 0.0};
const int localizationStartParticles = 200;

vractolib::OdomManager odom(horizWheels, vertWheels, imu);
std::vector<vractolib::DistanceSensor> mclSensors = {
    vractolib::DistanceSensor(dist1, vunits::Pose{5.0, 12.0, 5.0}),
    vractolib::DistanceSensor(dist2, vunits::Pose{-5.0, 12.0, 5.0}),
    vractolib::DistanceSensor(dist3, vunits::Pose{5.0, 12.0, -5.0}),
    vractolib::DistanceSensor(dist4, vunits::Pose{-5.0, 12.0, -5.0})
};
vractolib::MCL mcl(&odom, mclSensors, localizationStartPose,
localizationStartParticles);
```

Each distance sensor has a mount pose relative to the robot. That matters because AMCL cannot just ask "what did the sensor read?" It also needs to know where that sensor is physically located and what direction it is facing on the robot. Otherwise it cannot predict what that sensor *should* see for any candidate particle.

The field geometry lives in `MCLMap`. Right now, the map is intentionally simple as it just contains the Push Back map and data for height at 12 inches; it is made of line segments for the field perimeter and goal geometry. That is enough for the algorithm to raycast against and predict what a sensor should measure from any hypothesized pose.

`DistanceSensor` itself is also kept small and practical. It converts the raw PROS distance reading into inches and filters out bad sensor data using a minimum confidence threshold. That way the MCL update ignores readings that are likely too noisy or invalid to trust.

The core MCL loop lives in `MCL::update()`:

Custom

```
void MCL::update() {
    mutex.take();
    if (paused) {
        mutex.give();
        return;
    }

    if (particles.empty()) {
        seedParticles(startCenter, startParticleCount);
    }

    const vunits::Pose odomDelta = odom->getPoseDelta();
    applyMotionUpdate(odomDelta);

    const int validMeasurements = applySensorModel();
    normalizeWeights();
    if (validMeasurements > 0) {
        resampleParticles();
    }

    EstimateStats stats{};
    estimatePose(stats);

    const bool converged = validMeasurements > 0 &&
        stats.positionStdDev <= config.localizedPositionStdDev &&
        stats.headingStdDev <= config.localizedHeadingStdDev;

    if (converged) {
```

```
        localizedUpdates += 1;
    } else {
        localizedUpdates = 0;
    }

    localized = localizedUpdates >= config.localizedUpdateThreshold;
    const vunits::Pose estimate = poseEstimate;
    mutex.give();

    if (localized) {
        odom->setPose(estimate);
    }
}
```

This function is the full particle filter cycle and is updated after odometry every 10ms. Every update, it:

1. Gets the odometry delta since the last cycle.
2. Applies that motion to every particle with added noise.
3. Uses the distance sensors to score how believable each particle is.
4. Resamples particles based on those scores.
5. Estimates the current pose from the weighted particle cloud.
6. Decides whether the cloud has converged tightly enough to trust.
7. If it has converged, pushes that corrected pose back into odometry.

That last step is especially important. MCL is not replacing odometry every single frame. Instead, it waits until it is confident enough that the particle cloud has localized well, and then it corrects the odometry pose. This makes the system much more stable than hard-resetting pose from a single sensor event.

The first stage of the algorithm is particle initialization:

```
void MCL::seedParticles(const vunits::Pose &center, int particleCount) {
    particles.clear();
    particles.reserve(std::max(particleCount, 1));
```

```
    for (int i = 0; i < std::max(particleCount, 1); ++i) {
        Particle particle{
            vunits::Pose{
                center.x + sampleNormal(config.startPosStdDev),
                center.y + sampleNormal(config.startPosStdDev),
                vunits::wrapToSignedRadians(center.theta +
sampleNormal(config.startHeadingStdDev))
            },
            1.0
        };
        particles.push_back(particle);
    }

    setUniformWeights();
    poseEstimate = center;
    localized = false;
    localizedUpdates = 0;
}
```

This seeds the particle cloud around a known or assumed starting pose using Gaussian noise. So instead of pretending the robot is exactly at the start pose, it begins with a cloud around that pose. That is a much more realistic representation of uncertainty and prevents the position from converging too quickly.

Then comes the motion update:

```
void MCL::applyMotionUpdate(const vunits::Pose &odomDelta) {
    for (auto &particle : particles) {
        particle.pose.x += odomDelta.x +
sampleNormal(config.motionPosStdDev);
        particle.pose.y += odomDelta.y +
sampleNormal(config.motionPosStdDev);
```

```
        particle.pose.theta = vunits::wrapToSignedRadians(
            particle.pose.theta + odomDelta.theta +
sampleNormal(config.motionHeadingStdDev)
        );
    }
}
```

This is how odometry is fused into the particle filter. Every particle is moved by the same odometry delta, but with some added random noise. That noise matters because if we moved every particle exactly the same way, the cloud would unrealistically stay too certain even when the robot's motion was imperfect. The noise lets uncertainty grow when motion is uncertain, which is exactly what a good localization filter should do.

The sensor model is where the particles are actually judged against reality:

```
int MCL::applySensorModel() {
    int validMeasurements = 0;

    if (particles.empty()) {
        return validMeasurements;
    }

    std::vector<double> logWeights(particles.size(), 0.0);
    for (auto &sensor : sensors) {
        if (!sensor.isValid()) {
            continue;
        }

        double measuredDistance = sensor.getDistanceInches();
        if (measuredDistance <= 0.0) {
            continue;
        }
        if (measuredDistance > config.maxSensorRange) {
```

```
            measuredDistance = config.maxSensorRange;
        }

        validMeasurements += 1;

        for (std::size_t i = 0; i < particles.size(); ++i) {
            const vunits::Pose sensorPose =
sensorPoseForParticle(particles[i], sensor);
            double predictedDistance = config.maxSensorRange;
            const bool hit = map.findNearestIntersection(sensorPose,
sensorPose.theta, config.maxSensorRange, predictedDistance);
            if (!hit) {
                predictedDistance = config.maxSensorRange;
            }

            logWeights[i] += gaussianLogLikelihood(measuredDistance -
predictedDistance, config.measurementStdDev);
        }
    }

    if (validMeasurements == 0) {
        setUniformWeights();
        return 0;
    }

    const double maxLogWeight = *std::max_element(logWeights.begin(),
logWeights.end());
    double totalWeight = 0.0;
    for (std::size_t i = 0; i < particles.size(); ++i) {
        particles[i].weight = std::exp(logWeights[i] - maxLogWeight);
        totalWeight += particles[i].weight;
    }
```

```cpp
    if (totalWeight <= epsilon) {
        setUniformWeights();
        return validMeasurements;
    }

    for (auto &particle : particles) {
        particle.weight /= totalWeight;
    }

    return validMeasurements;
}
```

This is the heart of the correction step. For each valid sensor, we:

1. Read the real measurement from the robot.
2. Compute where that sensor would be for each particle.
3. Raycast into the map to predict what that particle should see.
4. Compare predicted distance to actual measured distance.
5. Increase or decrease the particle's weight based on how well it matches.

The map ray-casting is done by `MCLMap::findNearestIntersection()`, which checks the sensor ray against all field line segments and returns the nearest hit. That gives the predicted distance measurement for that particle. The comparison is then scored with a Gaussian likelihood, so small measurement errors are rewarded much more than large ones.

```cpp
bool MCLMap::findNearestIntersection(const vunits::Pose &origin, double
heading, double maxDistance, double &distanceOut) const {
        const double dx = std::cos(heading) * maxDistance;
        const double dy = std::sin(heading) * maxDistance;
        const double epsilon = 1e-9;

        bool found = false;
        double bestDistance = std::numeric_limits<double>::max();

        for (const auto &segment : segments) {
```

```
            const double sx = segment.end.x - segment.start.x;
            const double sy = segment.end.y - segment.start.y;
            const double qpx = segment.start.x - origin.x;
            const double qpy = segment.start.y - origin.y;

            const double denominator = cross2D(dx, dy, sx, sy);
            if (std::fabs(denominator) <= epsilon) {
                continue;
            }

            const double t = cross2D(qpx, qpy, sx, sy) / denominator;
            const double u = cross2D(qpx, qpy, dx, dy) / denominator;

            if (t < 0.0 || t > 1.0 || u < 0.0 || u > 1.0) {
                continue;
            }

            const double distance = t * maxDistance;
            if (distance < bestDistance) {
                bestDistance = distance;
                found = true;
            }
        }

        if (!found) {
            return false;
        }

        distanceOut = bestDistance;
        return true;
    }
```

Function which finds potential intersection between pose and stored map

The implementation uses **log weights** first, and only exponentiates at the end. That is a practical numerical-stability detail. If we multiplied many tiny Gaussian probabilities directly, the weights could underflow very easily. Summing log-likelihoods avoids that problem.

After the sensor model, the algorithm resamples:

```
void MCL::resampleParticles() {
    if (particles.empty()) {
        return;
    }

    std::vector<double> cumulativeWeights;
    cumulativeWeights.reserve(particles.size());

    double cumulative = 0.0;
    for (const auto &particle : particles) {
        cumulative += particle.weight;
        cumulativeWeights.push_back(cumulative);
    }

    if (cumulativeWeights.empty() || cumulativeWeights.back() <= epsilon)
{
        setUniformWeights();
        return;
    }

    std::vector<Particle> resampledParticles;
    resampledParticles.reserve(config.maxParticles);
    std::vector<vunits::Pose> occupiedBins;
    occupiedBins.reserve(config.maxParticles);

    std::uniform_real_distribution<double> offsetDist(0.0, 1.0 /
static_cast<double>(config.maxParticles));
    const double r = offsetDist(rng);
```

```
    std::size_t index = 0;
    double threshold = cumulativeWeights[0];
    int requiredSamples = config.minParticles;

    for (int sample = 0; sample < config.maxParticles; ++sample) {
        const double u = r + static_cast<double>(sample) /
static_cast<double>(config.maxParticles);
        while (index + 1 < cumulativeWeights.size() && u > threshold) {
            index += 1;
            threshold = cumulativeWeights[index];
        }

        Particle particle = particles[index];
        particle.weight = 1.0;
        resampledParticles.push_back(particle);

        if (updateBinOccupancy(particle.pose, occupiedBins)) {
            requiredSamples =
requiredSampleCount(static_cast<int>(occupiedBins.size()));
        }

        if (static_cast<int>(resampledParticles.size()) >= requiredSamples
&&
            static_cast<int>(resampledParticles.size()) >=
config.minParticles) {
            break;
        }
    }

    particles = std::move(resampledParticles);
    setUniformWeights();
}
```

This is where the "adaptive" part really shows up. Better particles are copied more often, while bad particles gradually disappear. But unlike a fixed-particle implementation, this one does not always keep exactly the same number of samples. Instead, it tracks how many bins in state space are occupied and uses a Kullback–Leibler divergence (KLD) bound to decide how many particles are needed. If the belief is spread out, more particles are kept. If it is already tight and confident, fewer particles are needed.

That is a very important optimization for a real robot because it balances accuracy against runtime. We want enough particles to localize well, but not more than necessary.

After resampling, the code estimates the robot pose from the weighted cloud:

```
void MCL::estimatePose(EstimateStats &stats) {
    if (particles.empty()) {
        poseEstimate = startCenter;
        stats = {std::numeric_limits<double>::max(),
std::numeric_limits<double>::max()};
        return;
    }

    double meanX = 0.0;
    double meanY = 0.0;
    double meanSin = 0.0;
    double meanCos = 0.0;

    for (const auto &particle : particles) {
        meanX += particle.pose.x * particle.weight;
        meanY += particle.pose.y * particle.weight;
        meanSin += std::sin(particle.pose.theta) * particle.weight;
        meanCos += std::cos(particle.pose.theta) * particle.weight;
    }

    poseEstimate = vunits::Pose{
        meanX,
```

```
        meanY,
        std::atan2(meanSin, meanCos)
    };
    ...
}
```

Position is estimated with a weighted average, while heading is estimated using weighted sine and cosine averages. That heading trick is important because angles wrap around; a plain arithmetic average would behave badly near the `-\pi / \pi` boundary.

The code also computes position and heading spread, and only marks the filter as localized when that spread stays small for several consecutive updates. So the system is not just asking "what is the average particle pose?" It is also asking "has the cloud actually converged tightly enough that we trust this estimate?"

Finally, the integration with the rest of the robot is handled in the localization task:

```
static Task localizationTask{[] {
    while (true) {
        odom.update();
        mcl.update();

        const bool useMclPose = mcl.isLocalized();
        const vunits::Pose pose = useMclPose ? mcl.getPose() :
odom.getPose();
        lcd::set_text(4, std::to_string(pose.x)+",
"+std::to_string(pose.y)+",
"+std::to_string(vunits::radToDeg(pose.theta)));
        lcd::set_text(6, std::string(mcl.isPaused() ? "MCL paused " : "MCL
running ") + std::to_string(mcl.getParticleCount()));
        delay(vconfig::updateRate);
    }
}};
```

This is a good summary of how the system is intended to be used. Odometry is always updating. MCL is also updating. If MCL has not converged yet, the system can still rely on odometry alone. Once MCL is localized, its estimate becomes trustworthy enough to use as the robot pose, and it can also correct the odometry estimate itself.

So the full code path can be summarized like this:

1. Seed a cloud of particles around a starting pose.
2. Move every particle using odometry plus noise.
3. Predict sensor readings for every particle by ray-casting into the field map.
4. Weight particles according to how well they match the real sensor readings.
5. Resample adaptively so good particles survive and particle count stays practical.
6. Estimate the final pose and wait until the particle cloud is consistently tight.
7. Once localized, use that estimate to correct the robot's odometry pose.

That is why this system is so useful. It does not just measure distance to a wall and reset the robot. It continuously reasons about where the robot could be, compares those hypotheses against the field, and gradually converges to a more reliable pose estimate.

Sensor Integration - The Optical System

Objective: Autonomous Block Sorting:

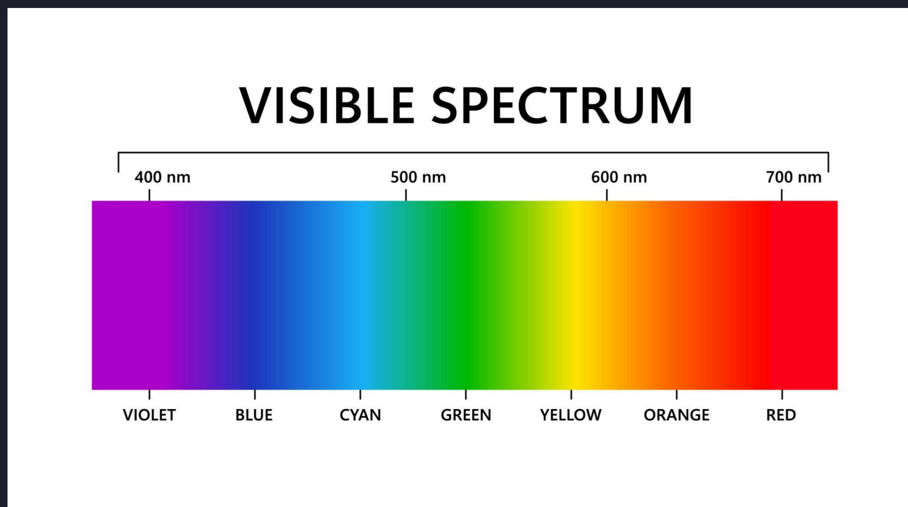
In the *Push Back* game environment, precision is paramount. Accidentally scoring an opponent's block results in severe penalties. Our objective was to create an **Automated Sorting System** that reduces driver cognitive load by programmatically identifying and ejecting "enemy" game elements.

HSV Color Space:

Data Normalization: HSV vs. RGB:

The V5 Optical Sensor provides raw RGB data, but lighting conditions on competition stages are notoriously inconsistent. To ensure reliability, we convert raw data into the **HSV (Hue, Saturation, Value)** color space.

- The Logic: Unlike RGB, where a shadow can change all three values, the Hue remains relatively constant regardless of brightness or shadows. This makes our detection algorithm robust under various stadium lighting conditions.



Essentially, we are converting the data from the optical sensor, which is the period of the waves of reflected light from the color. The above photo represents the visible spectrum of

light, but we are only dealing with the blue and the red portions of the visible spectrum, because these are the colors of the blocks in the “Push Back” game.

Color Definitions:

- Red Blocks: Hue 0 degrees to 20 degrees, as well as 340 degrees to 360 degrees
- Blue Blocks: Hue 200 degrees to 260 degrees

Proximity Check:

To prevent the sensor from accidentally "seeing" blocks outside the robot, we use a **Proximity Check**. The sorting logic only triggers if the **Proximity Value is > 200**, confirming the block is actually deep inside the intake.

Finite State Machine (Sorting Logic):

We managed our sorting logic using a **Finite State Machine (FSM)**. This ensures the robot transitions smoothly between identifying a block and ejecting it without getting "stuck" in a loop.

Sorting Execution (Blue Alliance Example):

1. **Idle:** The intake spins forward normally.
2. **Detection:** The sensor identifies a Red (Enemy) Hue and high proximity.
3. **Action:** The code instantly reverses the intake motors at full voltage (-12.0V).
4. **Recovery:** The motors stay in reverse for 500ms to ensure the block is cleared before returning to the Idle state.

Signal Hysteresis:

To avoid "sensor noise" (short flashes of color), we require the sensor to see the opposing color for **3 consecutive frames (30ms)** before triggering an ejection. This acts as a filter to ensure every ejection is intentional.

Auton Manager

Why it Exists

`AutonManager` is a very small helper class, but it solves one of the most annoying structural problems in competition code: choosing which autonomous routine to run, while also keeping track of which side and alliance that routine should be run on.

Without something like this, the autonomous code usually ends up with a lot of scattered booleans, ad-hoc variables, or duplicated routines for left and right starts. That gets messy very quickly, especially once multiple autonomous options exist. By wrapping those decisions into one class, it becomes much simpler to set the selected autonomous routine and account for starting side in a clean and consistent way.

The class stores 3 pieces of state:

1. The selected autonomous routine.
2. The starting side.
3. The alliance color.

This manager is then paired with our custom **LVGL** brain display that allows for step-by-step selection of autonomous routines once the robot is already running.

Code Implementation

Code

```
enum Alliance {
    RED,
    BLUE
};

enum Side {
    LEFT,
    RIGHT
};

enum AutonType {
    NONE,
    QUALS,
    SAWP,
    ELIMS,
```

```
    SKILLS
};
```

These enums make the rest of the code much easier to read. Instead of passing around raw integers or strings, the code can directly say what routine, side, or alliance is being used.

The most useful part of the implementation is the side sign:

```
Side AutonManager::setSide(Side newSide) {
    side = newSide;
    s = (side == LEFT) ? 1 : -1;
    return side;
}
```

This means the class does not just remember whether the robot is starting on the left or right. It also converts that into a simple `+1` or `-1` multiplier that can be reused throughout autonomous code. That is especially helpful for mirrored routines, because many turns and x-coordinates only need to be flipped in sign when swapping sides.

So instead of writing 2 separate versions of the same routine, we can often write one routine and mirror parts of it with `autonManager.sign()`.

For example:

```
dt.turnTo(-90 * autonManager.sign());
dt.turnTo(135 * autonManager.sign());
dt.moveToPoint(vunits::Pose{-11.0 * autonManager.sign(), 30.0,
vunits::degToRad(0.0)}, false, 3000, 150, 4500);
```

This is much cleaner than having separate hardcoded values for left and right autonomous starts.

The other major use of `AutonManager` is selecting which routine should run at all. In `main.cpp`, the selected routine is used in a switch statement:

```cpp
void autonomous() {
    switch (autonManager.getAutonType()) {
        case vractolib::AutonType::NONE:
            return;
        case vractolib::AutonType::QUALS:
            // qual auton
            dt.move(29.5);
            tongue.enable();
            dt.turnTo(-90 * autonManager.sign());
            break;
        case vractolib::AutonType::SAWP:
            // solo awp auton
            dt.move(30.75);
            dt.turnTo(-90);
            break;
        case vractolib::AutonType::ELIMS:
            // elims auton
            intakeLow.move_voltage(vconfig::maxVolt);
            dt.moveToPoint(vunits::Pose{-11.0 * autonManager.sign(), 30.0,
vunits::degToRad(0.0)}, false, 3000, 150, 4500);
            break;
        case vractolib::AutonType::SKILLS:
            // skills auton
            ...
            break;
    }
}
```

How the selected autonomous routine is actually used

This structure keeps the autonomous entry point simple. The selection logic is separated from the actual motion logic, so the `autonomous()` function just asks the manager what was chosen and runs the corresponding routine.

That is why this class is useful even though it is small:

1. It centralizes autonomous selection.
2. It makes mirrored left/right routines much easier to write.
3. It keeps the rest of the code cleaner by replacing scattered states with one small abstraction.

Solenoid Class

Translate

Why it Exists

The `Solenoid` class is a very small wrapper around `pros::adi::DigitalOut`, but it makes pneumatic code much more logical to work with. The main reason it exists is that the raw digital state of a pneumatic solenoid does not always match the behavior we actually care about.

For example, depending on how the pneumatic system is plumbed, a solenoid being "off" might actually mean the mechanism is extended. A tongue mechanism is a good example of this: electrically disabled does not necessarily mean physically retracted. If we write code directly against raw `true` and `false` values, the meaning becomes confusing very quickly.

By wrapping the raw digital output in a `Solenoid` class, we can instead think in terms of mechanism behavior:
1. `enable()` means put the mechanism into its active state.
2. `disable()` means return it to its default state.
3. `toggle()` means switch to the other state.

That makes the code much more readable, because the logic now matches what the mechanism is supposed to do rather than what voltage level happens to be sent to the valve.

The class stores:
1. A reference to the underlying digital output.
2. A `defaultState`, which represents the state the solenoid should return to when disabled.
3. The current logical `state`.

Code Implementation

Code

```
class Solenoid {
    private:
        pros::adi::DigitalOut &solenoid;
        bool defaultState; //default state for disabled
        bool state;
```

```cpp
    public:
        Solenoid(pros::adi::DigitalOut &solenoid);
        Solenoid(pros::adi::DigitalOut &solenoid, bool defaultState);

        void setDefault(bool defaultState);
        void enable();
        void disable();
        void toggle();
        void toggleOn(bool cond);
};
```

The public interface of the solenoid wrapper

The key design idea is the `defaultState`:

```cpp
Solenoid::Solenoid(pros::adi::DigitalOut &solenoid, bool defaultState) :
solenoid(solenoid) {
    this->state = defaultState;
    Solenoid::setDefault(defaultState);
}
```

When the wrapper is created, we define what "disabled" should mean for that mechanism. From then on, the class takes care of converting logical mechanism state into the correct raw output value.

That means the rest of the implementation becomes very simple. If the default electrical state is `false`, then `enable()` drives it to `true`. If the default electrical state is `true`, then `enable()` drives it to `false`. So the calling code never has to care how that mechanism is wired or plumbed.

So, in the main robot code, we can simply write `tongue.enable();` when actuating a mechanism. That is much clearer than trying to remember whether `set_value(true);` extends or retracts each individual pneumatic subsystem.

The other practical convenience this class adds is button-based toggling:

```
void Solenoid::toggleOn(bool cond) {
    if (cond) toggle();
}

//main.cpp
arm.toggleOn(master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_Y));
```

This seems small, but it makes controller code cleaner. Without it, every button press toggle would need an extra if statement around the solenoid call. With this helper, the code can directly express "toggle this mechanism when this button is newly pressed”.

So overall, this class exists for 2 main reasons:

1. It makes pneumatics logically sensible even when the raw solenoid state does not match the physical mechanism state in an intuitive way.
2. It provides quick helpers like `toggle()` and `toggleOn()` so pneumatic controls are easier to write and easier to read.

Like `AutonManager`, this is not a complicated abstraction. It is just a small wrapper that removes repetitive boilerplate and makes the rest of the robot code more understandable.

Conclusion & Summary:

System Reliability Analysis:

Overall, this library does a lot of things well, especially for a fully custom system made for a competition robot. One of its biggest strengths is that it stays understandable. Most parts of the code are broken into small, focused classes, so each subsystem has a clear responsibility. That makes it much easier to debug, modify, and explain, which is especially valuable for a team environment where newer programmers need to be able to understand what is happening.

This library is also strong because it was designed around real team needs rather than around trying to be a massive general-purpose framework. The code is modular, but still lightweight. It contains the pieces we actually found useful: drivetrain control, odometry, autonomous movement primitives, auton selection helpers, and pneumatics abstractions. That makes it easier to maintain and easier to adapt from one robot iteration to the next.

More broadly, this library shows that writing a custom robotics library can be valuable even when public alternatives already exist. It gave us more control over the design, helped us understand the math and logic behind our robot more deeply, and let us build systems in a way that matched how our team thinks about software.

Key Achievements:

1. **Custom Library:** Built a fully custom robotics library that combines drivetrain control, odometry, autonomous movement, utility abstractions, and robot-specific helpers into one modular and understandable system tailored to our team's needs.
2. **Adaptive MCL:** Added Adaptive Monte Carlo Localization to improve localization robustness, especially in situations where standard odometry can drift or become unreliable
3. **Pure Pursuit:** Developed and used a Pure Pursuit path-following approach, allowing the robot to follow smoother curved paths instead of relying only on simple turn-then-drive movement.

Future Improvements:

- **Motion Chaining:** We want to chain autonomous movements together more smoothly so the robot can flow between actions without fully stopping between each command.
- **MCL Improvements:** We aim to improve and better tune our Adaptive Monte Carlo Localization system, especially with stronger handling of distance sensors and field disturbances like crossing the parking barrier.

Written by: Sonit Sahoo, Ediz Guzey

- **Better Driver Signaling Via Vibrations:** It would likely be helpful if the controller automatically vibrates to signal the driver. For instance, it could vibrate once all balls have been outtaked, or when there are 15 seconds remaining.
- **More "Character" in Brain Display:** The current brain display is purely utilitarian. It would be interesting to add a dynamic face system onto it that would react to what's going on, the current state of the match, etc.

Final Statement

There is still room to improve, but even in its current form, this library already provides a strong base for both driver control and autonomous programming. More importantly, it is a codebase that we understand deeply, can change quickly, and can continue building on as our robots and programming experience improve.

(^.ㅗ.^)ʃ

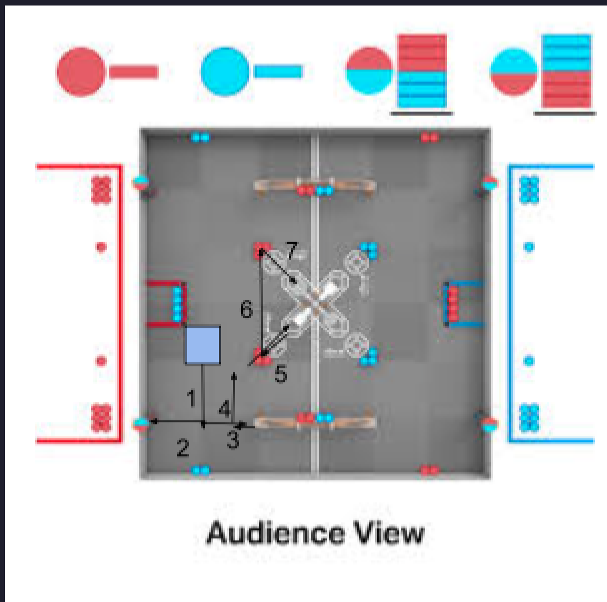
(^▨-▨^)ʃ

(^. .^)ʃ

Autonomous Pathing (skills, & matches)

10 Ball Solo AWP

Goals: Achieve AWP without needing an alliance to score and earn the 10 point autonomous bonus as well as securing an early lead without needing a strong alliance



Steps:

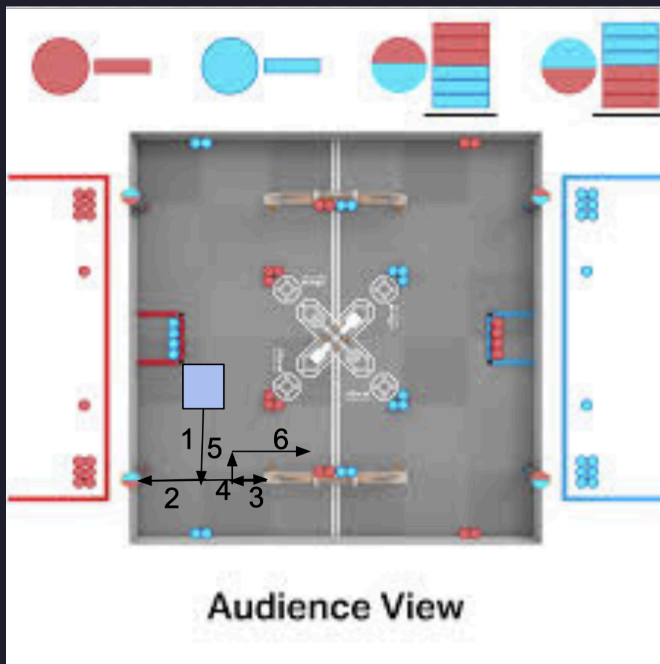
1. Move down between the match loader and long goal
2. Turn and release the tongue mech to secure 3 balls, as anymore would be scoring opposing blocks
3. Drive directly into the long goal, scoring a match load + 3 match loader blocks
4. Drive out and towards the middle goal
5. Score the 3 middle balls into the middle goal
6. Drive towards the high middle goal

Pros:
- Achieves solo AWP
- Able to score in most goals
- Achieves a strong auton without alliance intervention

Cons:
- Can interfere with alliance auton
- Hard to tune to perfection
- Vulnerable to long goal autons that use the wing to push out opposing blocks

4 Block long goal wing auton

Goals: To achieve control zone in the long goal and prevent opposition from scoring in the long goal



Steps:

1. Go down directly to position the bot in between match loader and long goal
2. Go to matchloader and get 3 balls out of it
3. Next go to long goal and score the match load + those 3 balls from the loader
4. After go back slightly to set up #5
5. Go up and turn to get in wing position
6. Drive forward with wing down to push those 4 blocks in control bonus and also prevent opposing blocks from being scored

Pros:

- Quite simple to tune
- Achieve extra points through the long goal
- Block opposition from scoring and long goal
- Better for decent bots rather than top level ones
- Good positioning for defending opposing bots

Cons:

- Requires help from alliance to achieve AWP
- Doesn't score a ton so susceptible to better autons
- Bad position to start cycling match loader to long goal

Written by: Sonit Sahoo, Ediz Guzey

Solo AWP max points (13 balls)

Goals: To achieve the highest possible auton score without alliance intervention

Steps:

1. Go down to prep for match loader
2. Go into match loader for 3 blocks
3. Score 4 blocks in long goal
4. Go out and drive towards middle goal
5. Intake the 3 blocks and score into middle low goal
6. Drive backwards
7. Drive up intake 3 blocks in front of high middle goal
8. Score into middle high goal
9. Drive out and into middle of match loader and long goal like step 1
10. Use match loader for 3 blocks
11. Finally score the blocks

Pros:

- Scores in all 4 aspects
- Max blocks scored at 13 with potential addition from alliance
- Very strong setup and very good chance at AWP

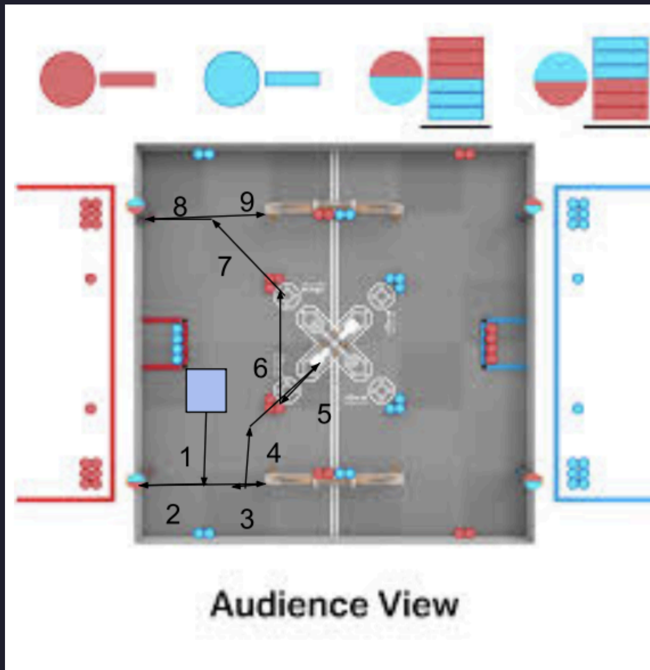
Cons:

- Might not have enough time
- Very hard to tune and will never be 100%
- Might potentially be blocked by alliance

Written by: Sonit Sahoo, Ediz Guzey

Double long goal AWP

Goals: achieve AWP with the slight variation of 2 long goals which allows better position to cycle between match loader and long goal



Steps:

1. Down like every other setup so far
2. Go back and gather 3 blocks from match loader
3. Score all 4 blocks into long goal
4. Go backwards and sideways for easy path to score
5. Drive forward collecting 3 blocks and scoring all 3 into low middle goal
6. Drive out and collect the other 3 blocks that are right in front of the other middle goal
7. Drive out setting up the typical cycle position for auton
8. Like set 2 collect match load blocks
9. Score 3 blocks from match loader

Pros:

- Good for AWP
- Prime cycle positioning
- Can be obtained easily during auton period

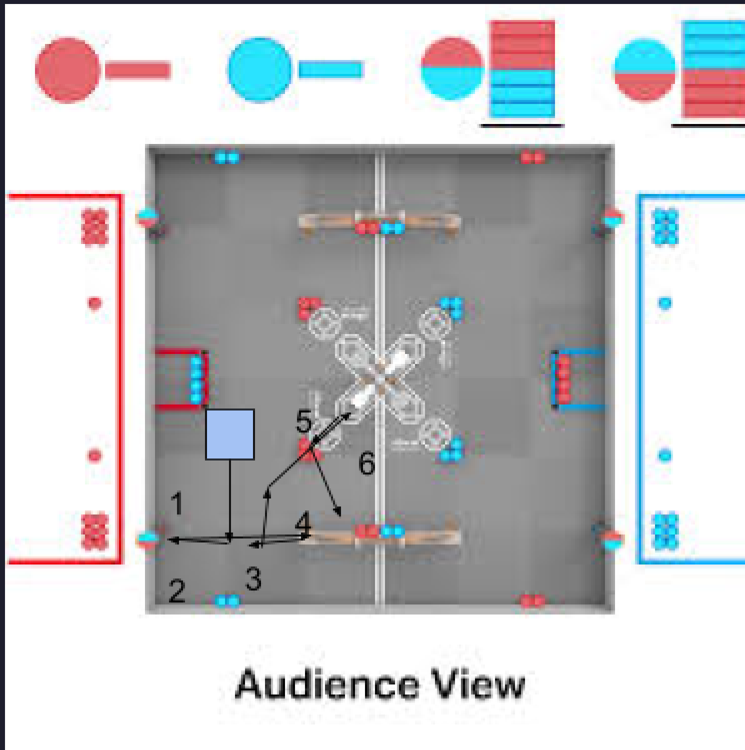
Cons:

- Will get in the way of teammate's auton
- Needs tuning
- Decently complex so it may just not hit without the timing and also external factors

Written by: Sonit Sahoo, Ediz Guzey

7 Block Auton

Goals: To set up for easy AWP if alliance can score 1 block in the long goal and easy tuning for quick quality auton



Steps:

1. Go down like all the other autons
2. Collect 3 blocks from loader
3. Score into long goal with 4 blocks
4. Go out and line up the blocks on the field and the short middle goal
5. Drive with intake forward collecting blocks and quickly outaking
6. Drive out and deploy wing to push long goal blocks into control zone

Pros:

- strong route for potential AWP
- good for gaining control of the long goal early
- not that difficult to tune with easy pathing
- gives alliance plenty of room to use their auton and achieve AWP

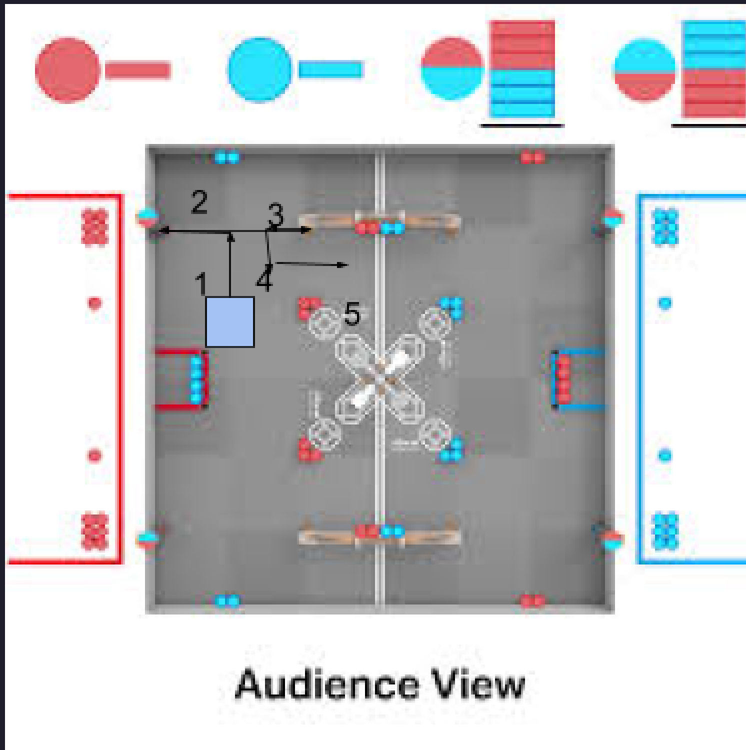
Cons:

- relies on alliance to achieve AWP
- the wing may not always be precisely into the long goal

Written by: Sonit Sahoo, Ediz Guzey

Left side 4 block long goal wing

Goals: To achieve total control of the long goal and push out the opposing bot to allow our alliance an easier time.



Steps:

1. Go directly straight up like all other auton's with the only difference being the side
2. Gather 3 blocks from the match loader
3. Score 4 blocks with the match load included
4. Go back and turn sideways a bit
5. Lower the wing drive straight the push the blocks forward and into the control zone

Pros:

- Achieves control of the long goal while pushes out opposing blocks
- Easy to tune compared to the other options

Cons:

- Does require alliance to achieve long goal and middle goal for AWP
- Does not score many points by itself
- Chance at accidentally crossing the middle if the wing does not hit

Written by: Sonit Sahoo, Ediz Guzey