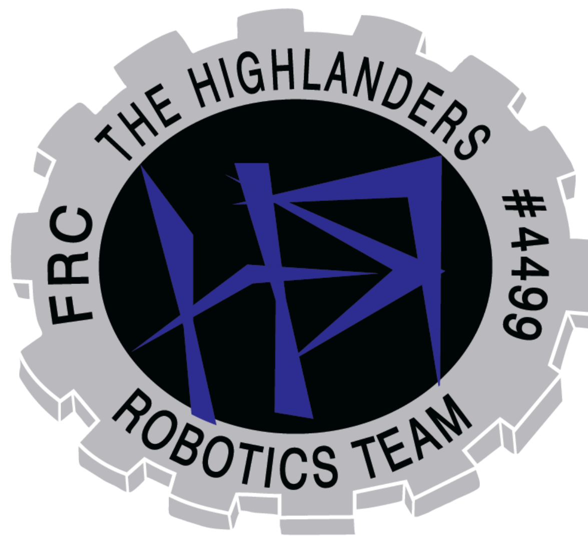
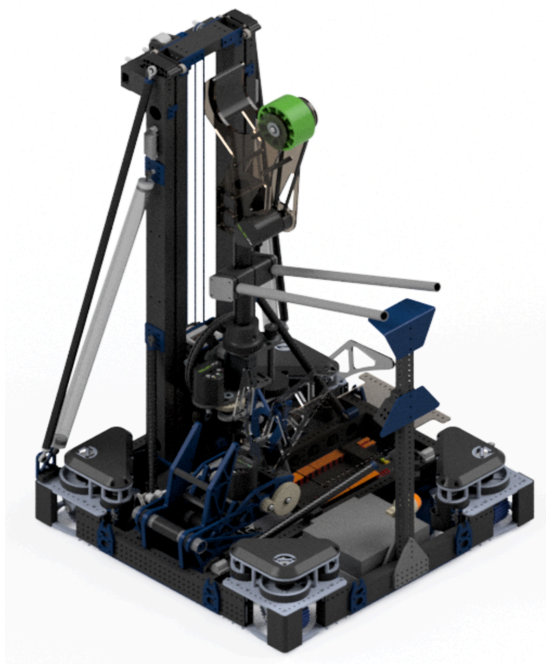




The Highlanders

#4499

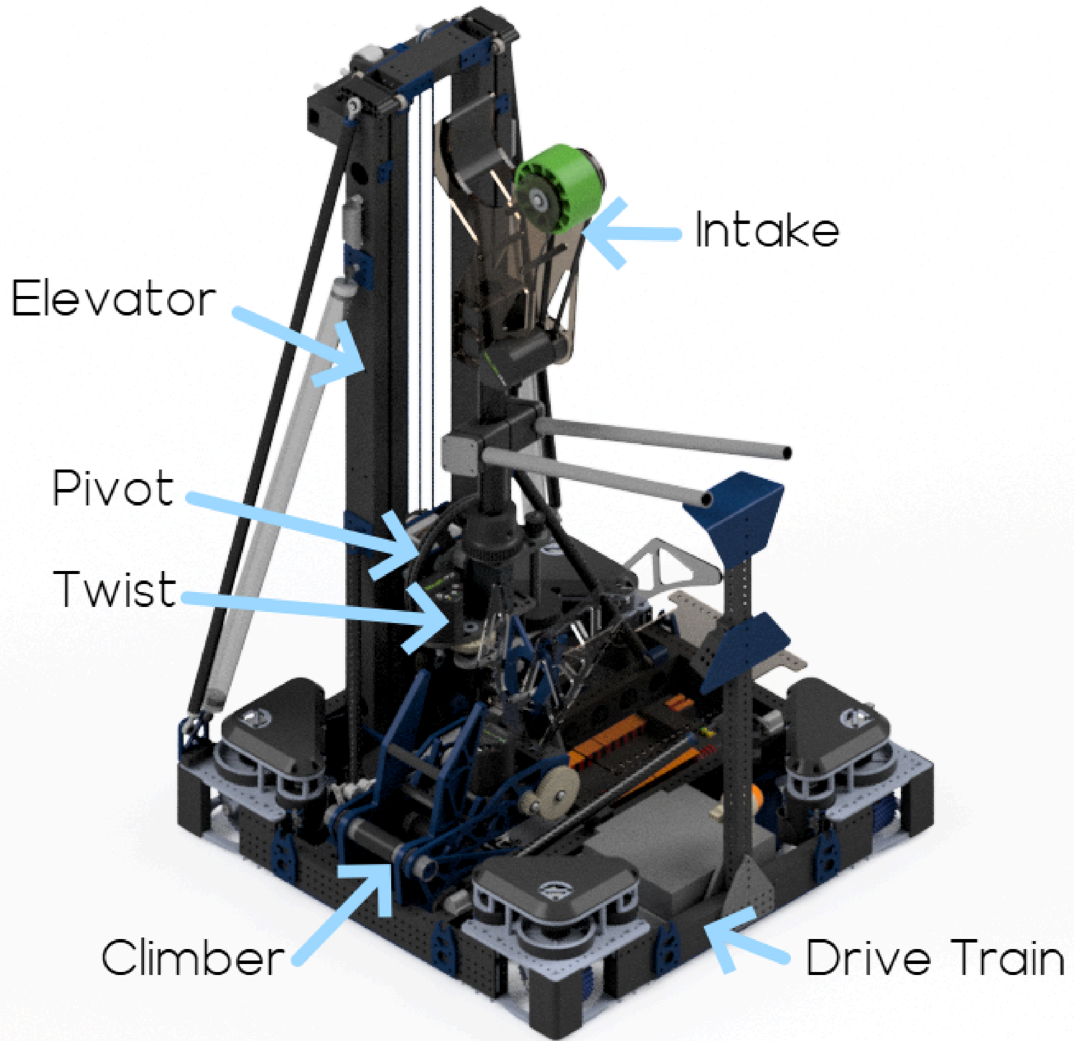
2025

Technical Binder

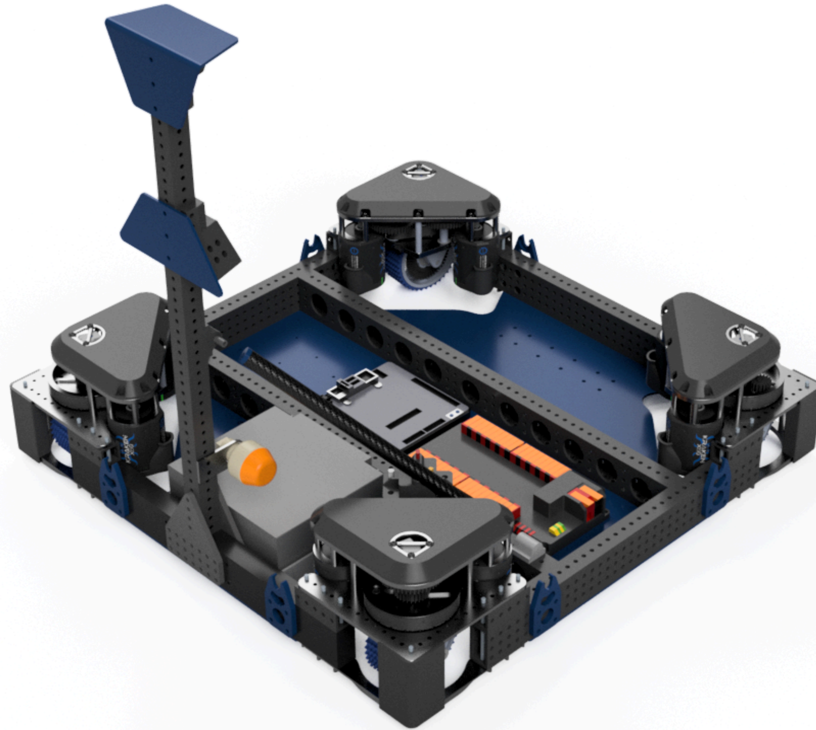
Table of Contents

ROBOT DESIGN **2**
Drive Train 3
Bumpers 4
Elevator 5
Pivot/Carriage 6
Twist 7
Intake 8
Climber 9
Machining 10
ELECTRICAL **12**
Elevator 12
Orange Pi 13
Cancoder 15
Battery 15
LED Lights 16
PROGRAMMING **17**
Strategy 17
Controls 17
Autonomous Procedures 18
Localization 18
Polar Forecast 40
Groups and Alliances 40
Pit Scouting 42
Auto Scouting 43
Match Scouting 44
Data Display 46
Event Page 46
Stats 48
Data Analysis 49
Data Simulation 49
Backend and Database 54

ROBOT DESIGN



Drive Train



The swerve drive train allows us to move quickly and precisely across the field omnidirectionally. To prevent tipping, the frame height was dropped 1”, and an 1/8" aluminum belly pan was added to provide a lower CG.

- Built with 2x1x0.125” aluminum tubing with a regular hole pattern, allowing for easy mechanism mounting.
- 26.5x26.5” frame
- MK4i-L3 modules (16.5ft/s) with black neoprene tread
- Powered by 4 FOC (Field Oriented Control) Kraken X60’s.
- Steel ballast and battery placed opposite to the Elevator to center CG for climb.

BUMPERS

Image 1

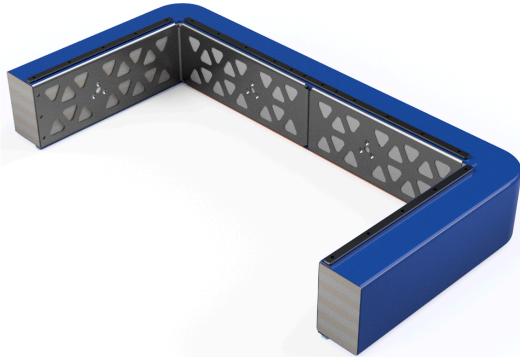


Image 2

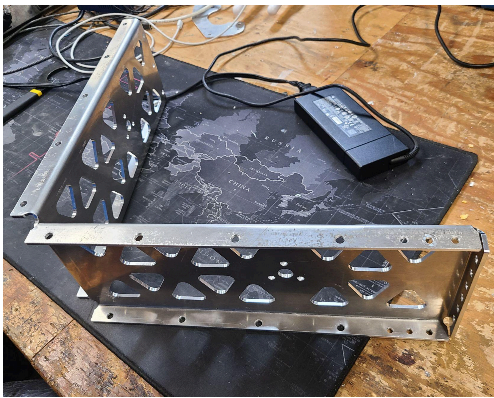


Image 3

Image 4

With the bumper changes this year, our bumper design changed away from pool noodles. The bumpers this year used foam tiles with a metal backing. Testing many types of foam and stacking methods were used to find and decide this solution.

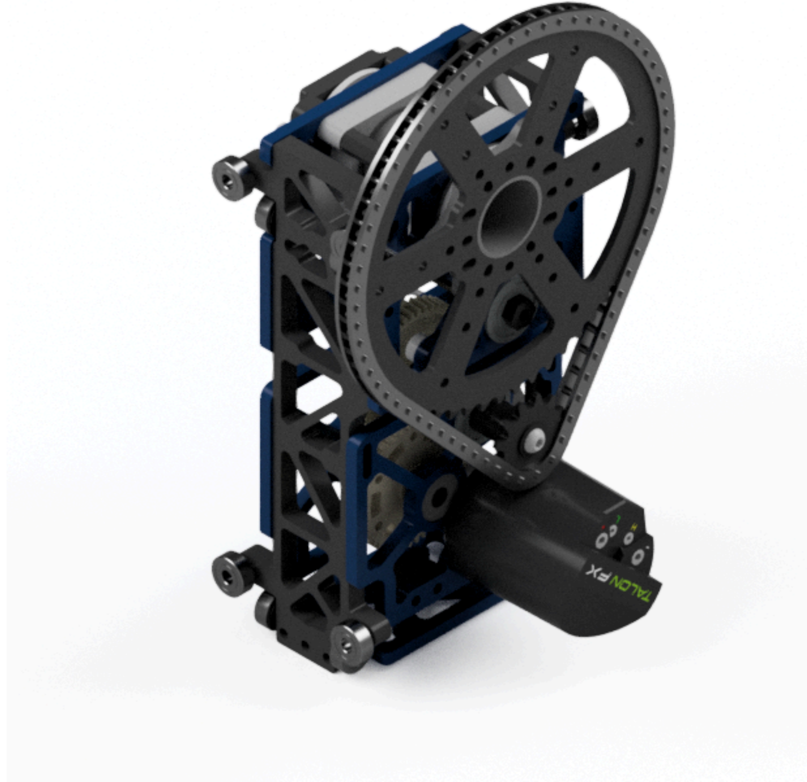
- EVA floor tiles, bought from AndyMark as FIRST Tech Challenge Field Soft Tiles (Image 4)
- Laser cut foam tiles horizontally stacked and glued 5" tall and a minimum of 3" thick (Image 4)
- Bent metal backing to ensure consistent structure along our whole frame (Image 3)
- Heat pressed team number on cordura fabric (Image 2)
- Attached to the chassis with 5/16-18 threaded studs on the bumper that fasten with wingnuts clamping on a plate.

Elevator

The elevator is a two-stage continuous design enabling the ability to score both on L4 and Algae in the Net, and contains the arm pivot as its carriage. It is designed to be lightweight and have fast deployment and retraction to minimize cycle times.

- Driven by 2 Kraken x60’s with 9:44 gear ratio on a 1.5” aluminum drum
- Uses 3/16” Dyneema Rope
- Max vertical travel of 60.5”
- 1”x2”x1/16” tubing used for upright, saving weight
- 20mm Carbon tie rods to increase stability
- LED light tubes to communicate with the driver
- Constant force spring to force carriage to deploy first
- E-Chain and rigging protected by sponsor panel

Pivot/Carriage



The pivot, residing in the carriage of the elevator, allows us to be symmetrical in function and abilities. Wires feed through the hollow pivot tube, giving way for a safe 270° of motion without putting the wires under much stress. 3/8" weight-saved aluminum was used as the side walls and bearing blocks of the carriage to increase the available room and save weight.

- Single Kraken x60 on an 18.3:1 gear reduction and 5:1 chain reduction using #35 chain
- Gearbox slides relative to the carriage by two 10-32 bolts to tension the chain
- CANcoder geared off of the main pivot to provide position
- 1.5”x0.08” steel tube with an OD lathed to 38mm to provide a clean contact to IGUS bushings that saved weight and complexity to traditional bearings

Twist

The twist is a mechanism that rotates the intake around a carbon fiber tube in the center. It ensures a seamless and easy transition between intake and placement on either side of the robot with no pass-off.

- 31.5:1 ratio powered by 1 Kraken x60 for high-speed rotation with adequate torque
- Custom-machined 10mm hardened steel shaft for extreme force transfers
- Uses a custom-machined pillow block to encase 30mm OD angular contact bearings
- 180° degrees of mobility for continuous motion with no interrupts

Intake

The Intake is designed to be able to pick coral off of the ground and the feeder and score on any reef level on both sides of our robot. Additional 18mm carbon fiber forks layered with silicon allow for the control of Algae on the reef and ground and subsequently place it into the Net or Processor.

- 3D printed clamps to 36mm OD carbon rod, allowing for easy replacement
- Single 4” 35A compliant wheel for manipulation of game elements
- Belt hidden within polycarbonate plates for protection
- Driven by a Kraken x60 with a 12:42 belt reduction

Climber

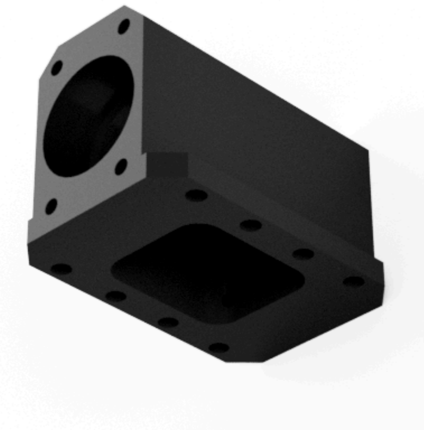
The climber allows the robot to deep climb ~4” off the ground within 3 seconds of acquisition of the Cage without a large weight and size footprint. Passive spring hooks grab onto one bar of the Cage, keeping it simple yet effective at various angles.

- A 200N gas spring deploys the climber
- Bottom forks sticking out of frame are held static with a locking mechanism and hold bottom of the cage, holding the chain above CG
- Large polycarbonate forks guide the Cage to the spring hooks
- Kraken x60 with a 90:1 worm gear reduction drives a 1” drum to winch down

Machining

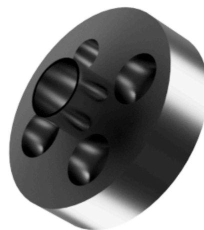
To achieve our ambitious design for our robot, many complex and unique parts of our mechanisms emerged, demanding high precision and tight tolerances. With high-quality parts machined in-house, we could fabricate a robot that accomplishes our goals. The process starts with designing the parts and then creating the sequences that are later run on the CNC machines.

Pillow Block



Machined to retain steel shaft with minimal wiggle room, preventing backlash in the system

Custom Steel Hub Interface

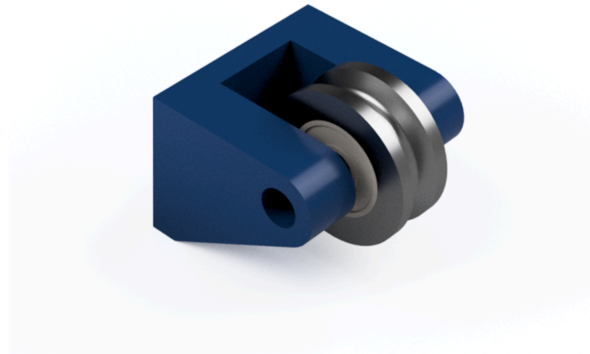


- Originally 2 separate parts, switched to one single steel hub integrating spacer to improve strength
- Made initially on a CNC mill and later finished on the manual lathe
- Hardened and tempered post-machining

Constant Force Spring Holder

- Originally 3d printed, but switched to aluminum to improve durability
- Machined on a Tree Journeyman 310, a CNC mill
- Clamping the part proved to be very difficult yet vital to the overall quality, leading to multiple attempts and orientations to machine this part successfully

Pulley Holder



- Machined on a Tree Journeyman 310, a CNC mill
- Starts with billet aluminum that is processed through multiple sequences
- A similar process to the previous Constant Force Spring Holder

ELECTRICAL

Going into 2025, we had a few main electrical goals that we wanted to keep as guiding principles through the season. We realized throughout the 2024 offseason that our lack of a detailed battery regiment was restricting us from competing at our fullest capacity, the amount of degrees of freedom we wanted on our robot this year would require very detailed wiring, and our programming teams switch from Limelights to photon vision would require a new way of communication and power for that subsystem. The way we wired the chassis, elevator, cameras, strain relieved our wires, and managed our batteries were all derived from these goals.

Elevator

From the start of the season, we immediately knew that our elevator and arm would hold many degrees of freedom, requiring multiple motors and encoders on moving elements. We wanted to minimize the volume of wires to ensure there was space for all necessary power and CAN and to save weight at the top of the elevator. Because of our electronics, weight, and space requirements, we came up with the following solution:

- A polycarbonate plate on the back of the elevator acts as an e-chain with wires zip-tied to it.
  - We reasoned this would give us a tighter bend radius, necessitating less space behind the elevator for wires.

- Run CAN up the elevator through an ethernet cable.
  - As ethernet is a group of wires inside another insulation, we thought this would give us convenient packaging for our CAN and the wires for any sensors we wanted to implement on the elevator or arm.
- Use the minimum possible wire gauges and breaker amperes to ensure adequate torque to all our motors.
  - We theorized this would be the best way to save weight up the elevator, as we'd previously run 10-gauge wires to all motors when some didn't require the torque allowed by such a low wire gauge.

Additionally, we knew that routing wires to our twist would be a struggle, as it rotated through such a range of motion that any wires would undergo significant stretching or bunching based on our current arm positioning. To solve this, we came up with the following idea:

- We would route all wires under the elevator carriage and then direct them under our pivot sprocket, where they could be attached to the center of the pivot's rotation. This

would allow us to fasten the wires there, minimizing the deformation of our wires to the intake, as they would be attached to the center of rotation so as to not undergo any significant length changes.

Our practice robot suffered significant electrical failures due to some of these choices, specifically:

- The polycarbonate E-chain was prone to fatigue and eventual snapping due to prolonged bending in our preset positions.
- Our CAN connectors frequently failed to hold on the internal ethernet wire, as the 26 AWG wire was much too high for our crimps to get a strong hold on.
- The wires under the elevator would often become loose, running the risk of being crushed by the carriage after becoming unstuck due to high velocities.

Because of these failures, when wiring our comp bot, we came to the following decisions:

- We switched back to the normal E-chain to ensure protection of the wires from all sides and to provide more structure and strength, supporting our wires through the E-chain motion.
- We returned to standard 22 AWG CAN wire. We decided not to use any sensors atop the elevator, which made all advantages of using ethernet obsolete as it took up too much room to warrant only supporting connection for one device.
- We hollowed out the rotating axle of the pivot and pillow block; wires could be run directly through the center of rotation of the pivot. This way, our wires would maintain the advantages of being anchored at the center of rotation (minimal deformation), and they would now be internal to the robot, providing them more protection and a more reliable path of movement.

Because of this iteration process on our first elevator of the season, we have been able to eliminate nearly all electrical issues throughout the incorporated subsystems and have come to a product with clean, reliable wiring.

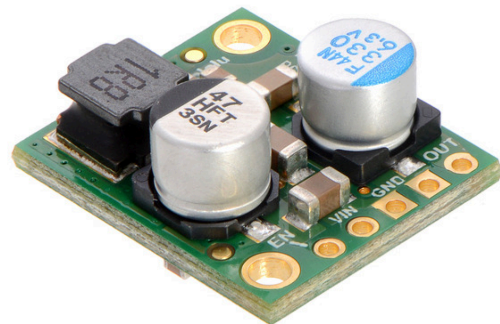
Orange Pi

After transitioning from Limelight to photon vision, our biggest challenge was ensuring adequate power supply for the Orange Pis.

- Connectors for power
    - We started with Dupont connectors but quickly transitioned away due to their fragility and loose connections.

- We changed to USB-C cables with power poles on the ends, which gave us a more durable and secure connection.

- VRM power
    - Initially, the Orange Pis were powered by the VRM, receiving 5V and 2A.
    - Our consistent loss of power led us to do more digging, ultimately revealing that the Pis were not receiving adequate power.
- Buck converter power
    - Our quick fix for this problem was using a buck converter. Powering the Orange Pis off a bus bar (terminal bar), which was powered by the buck converter, didn’t solve our problems.
    - As the battery dropped below 12V, the buck converter was no longer able to supply enough power, causing us to lose our vision.
- PDH power
    - Our last resort was to power the Pis directly off the PDH.
    - This worked temporarily while we waited for the Pololus to arrive.
- Pololu power
    - The ultimate solution was purchasing Pololu voltage regulators. With 5V and 3A, this tiny buck-boost provided us with consistent power for the Orange Pis.
    - After CADing and 3D printing a case for the Pololus, we integrated these onto our robots, ensuring sufficient power supply for our vision system.



CANcoder

Going into our season, we planned to incorporate a CTRE cancoder for our twist; however, problems quickly arose the more we ran our elevator.

- Hardware damage
  - Our biggest problem was our own elevator constantly hitting the cancoder and its mount, resulting in many broken cases and mounts.
  - This caused external hardware damage; some of our cancoders were missing components that we presume to have fallen after critical hits.
- Mount
  - We started with a 3D print to mount to the back of the elevator. We went through many iterations of mounting techniques, as the 3D print restricted the cancoder from reading accurate values since the magnet was blocked by the print.
- Wiring and soldering
  - The strong elevator was not a match for the weak solder joints.
  - After wires had been ripped out time after time, we decided to focus on higher attention to detail whilst soldering and more efficient strain-relieving techniques.
- New location
  - After many failed iterations, we decided to locate the cancoder on the inside of the carriage to eliminate hardware damage.
  - With this new location, we were also able to get Kraken PowerPole adapter boards on our motors. This gave us a direct way to receive power and a signal for our power and better strain relieving.

Battery

During the offseason, we encountered a recurring power issue that traced back to our batteries. Initially, we suspected the problem stemmed from old, worn-out batteries. However, even after purchasing multiple new ones for an off-season tournament, the issue persisted. Ultimately, we realized that poor battery maintenance at home had cost us, leaving us with mediocre batteries during competitions and limiting our performance. The robot draws a significant amount of power, forcing us to purchase new batteries.

- We changed the protocols for battery usage at home to avoid failures.
  - We reserved 10 batteries to be used only at tournaments, the rest were allocated for testing purposes. This can increase the longevity of our batteries with higher quality, increasing our performance at competitions.
- Battery testing

- Through battery testing, we track each battery's quality in an organized spreadsheet, making it easy to determine whether a battery is suitable for competition, testing, or recycling.
- When looking at each battery's test, we focus on their test amp hour. Our threshold for a competition battery is 14aH, while an at-home battery ranges from 10aH to 12aH. Anything below 10aH is put aside to be recycled later.

- Battery leads are created with higher attention to detail to increase longevity
    - This includes proper crimping, torquing the leads to spec on the battery, and sufficient use of heat shrink and electrical tape to avoid exposed connectors and lugs.

LED Lights

Our lights play a vital role in providing driver feedback. With two tie rods and swerve lights, visibility from the opposite side of the field is essential. To achieve this, each tie rod is equipped with its own CTRE CANdle, along with a separate CANdle dedicated to the swerve lights.

- Light Tie Rods
    - The light strip is hot glued, spiralling down a wooden rod.
    - The light sticks are put in tubes with vinyl light diffusers.
- Swerve lights
    - With clear 3D printed strips on our swerve covers to make lights more visible, the light strip is constrained within the cover.
    - Wires for lights are run through chassis tubes.
- To have quality feedback, we have set easy-to-understand and memorable light codes.
    - Disabled: Red or blue lights bounce up and down the tie rods or circle through the swerves based on our alliance color.
    - Teleop:
        - ➢ The back tie rods will be either white or green, based on coral or algae mode.
        - ➢ The front tie rods will be either purple for auto mode or alliance color for manual mode.
    - Placement sequences: Lights will flash purple for intaking, yellow for lining up, or green for scoring.

PROGRAMMING

Strategy

Through early game analysis, it became apparent that fast coral cycle times were going to be crucial in the Reefscape game. The main ways to enable fast cycles are auto-feeding, auto-placement, and other driver assists, all of which were implemented this year.

Controls

To cut down on inefficiency with multiple drivers, fitting all controls onto one controller was essential. Because buttons are needed for many different placement heights and directions, the controls were broken down into 2 modes: coral and algae. To reduce the amount of thinking the driver must do during the match, the robot automatically selects which side to feed coral, place coral, remove algae, place net algae, and place processor algae on. The robot also automatically chooses which height the algae to be removed is. A full controller layout is shown below.

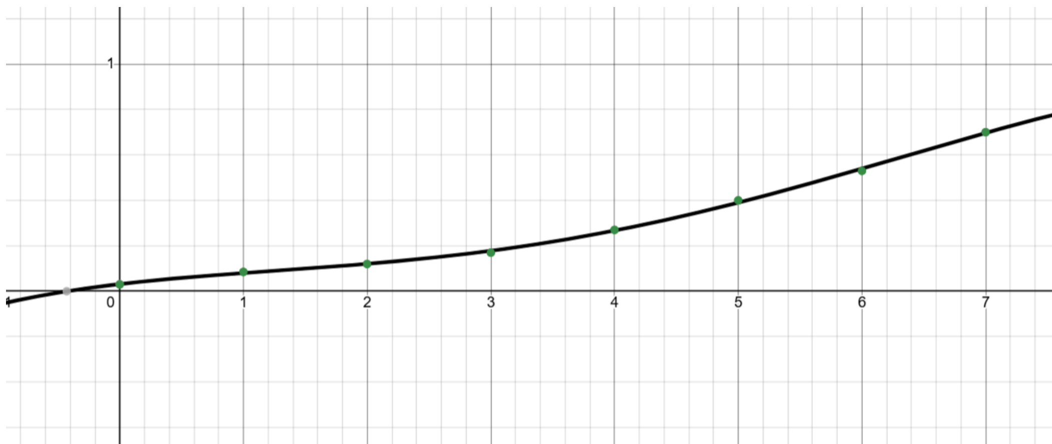
Autonomous Procedures

The goal during autonomous was to be able to configure only one auto but flip them in 3 directions so they could run on both the red and blue on both the processor and net sides. This would allow the robot to easily synergize with other robots that have autos that are more constrained. Another goal was to be able to consistently hit

- 3-piece coral auto that scores preload and two more from the feeder station. 1 is placed L1 and the others L4. Runs on both processor and non-processor sides.
- 3-piece coral auto that scores the preload and two more from the feeder station on L4 branches. Runs on both processor and non-processor sides.

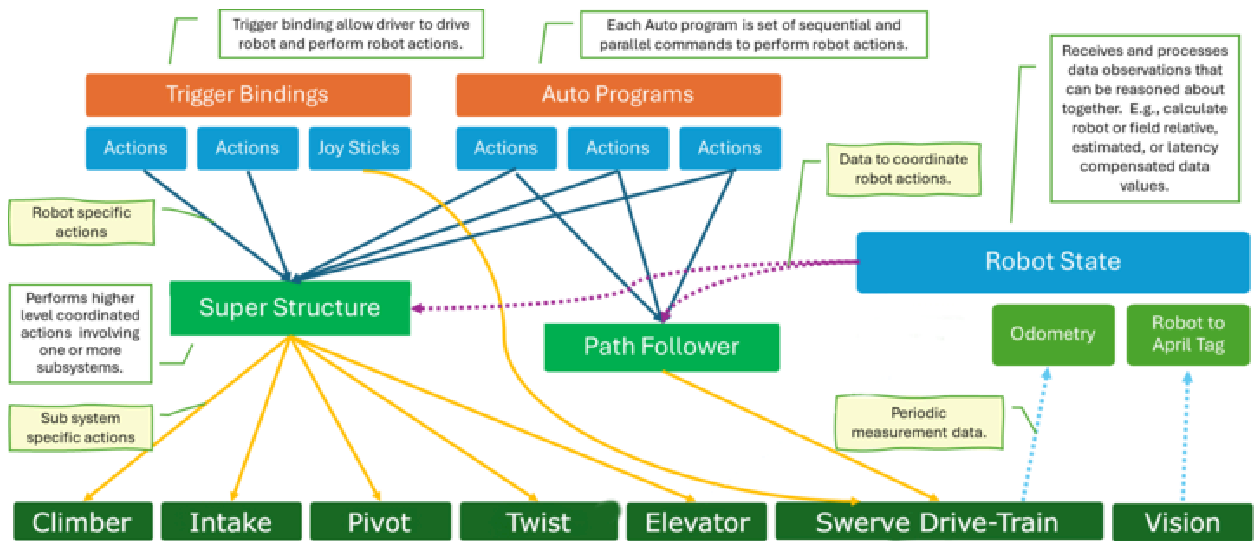
Localization

Camera localization was implemented to enhance robot control and field positioning using Orange Pis and Arducams. It allows the robot to automatically line up and place on any reef branch with an accuracy of less than 0.8 inches. This system works when the cameras view April tags around the field. Using its position and the corners of the tag in the frame, the robot can calculate where it is on the field. Localization also helps with autonomous pathing and optimizations to ensure the robot is in the right spot. It utilizes a Kalman filter with other basic filtering to reduce noise. The graph below shows a tuned quartic regression with different standard deviation values to change the vision measurement trust rating at different distances from tags. This figure of merit adjusts when more tags are being tracked, further increasing the accuracy of the pose estimation. A camera stick enabled the robot to see reef and feeder tags from both sides. This was done by setting up a high-angle and a low-angle camera facing either side of the robot.



State-Machining

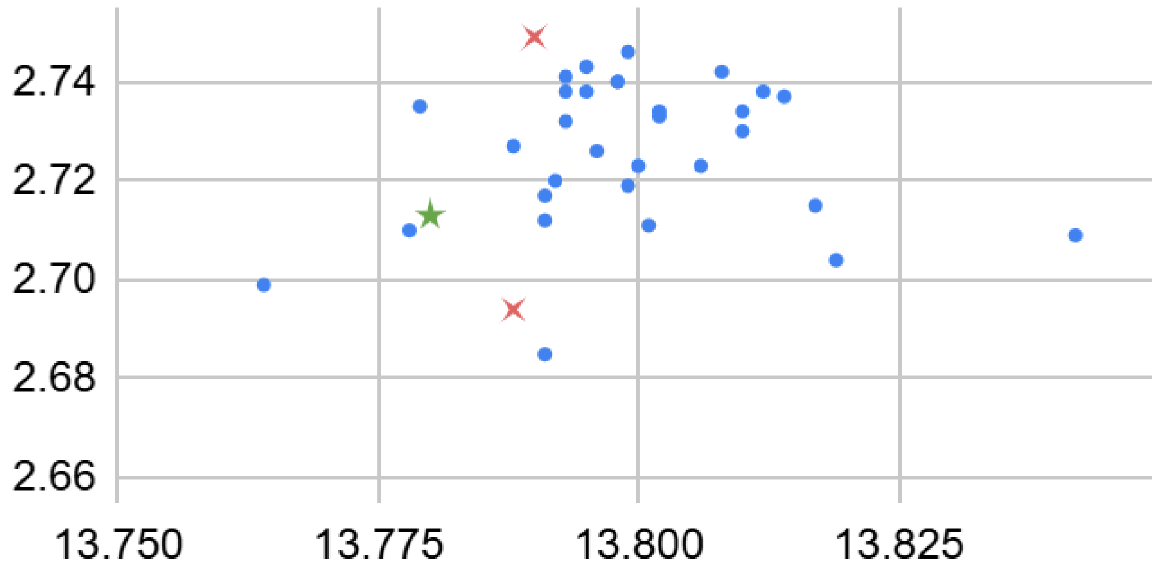
A state machine architecture integrated with WPILIB subsystems and command-based programming allows the code to break down complex tasks into states for the robot itself and each subsystem. Each state then has a transition to move to a different state, allowing for the robot to easily complete any task. The below flowchart showcases how the codebase is organized.



Auto Placement

	X	Y	Theta (radians)	X Error	Y Error	Magnitude Error (XY)	Theta Error	Successful	Battery Voltage
Meters/Radians	13.788	2.694	2.124	=B2-L	=C2-$M	=SQRT(POW(E2, 2)+F	=D2-N2	Position was off	12.3
Inches/Degrees	=B2*39.37	=C2*39.37	=(D2*180)/3.141	=E2*39.3	=F2*39.	=G2*39.37	=(H2*180)/3.		
	X	Y	Theta (radians)	X Error	Y Error	Magnitude Error (XY)	Theta Error	Successful	Battery Voltage
Meters/Radians	13.788	2.727	2.087	=B6-L	=C6-$M	=SQRT(POW(E6, 2)+F	=D6-N2	Yes	12.4
Inches/Degrees	=B6*39.37	=C6*39.37	=(D6*180)/3.141	=E6*39.3	=F6*39.	=G6*39.37	=(H6*180)/3.		
	X	Y	Theta (radians)	X Error	Y Error	Magnitude Error (XY)	Theta Error	Successful	Battery Voltage
Meters/Radians	13.778	2.71	2.092	=B10-$L	=C10-$	=SQRT(POW(E10, 2)	=D10-N2	Yes	12.4
Inches/Degrees	=B10*39.37	=C10*39.37	=(D10*180)/3.14	=E10*39	=F10*3	=G10*39.37	=(H10*180)/		
	X	Y	Theta (radians)	X Error	Y Error	Magnitude Error (XY)	Theta Error	Successful	Battery Voltage
Meters/Radians	13.793	2.732	2.091	=B14-$L	=C14-$	=SQRT(POW(E14, 2)	=D14-N2	Yes	12.4
Inches/Degrees	=B14*39.37	=C14*39.37	=(D14*180)/3.14	=E14*39	=F14*3	=G14*39.37	=(H14*180)/		
	X	Y	Theta (radians)	X Error	Y Error	Magnitude Error (XY)	Theta Error	Successful	Battery Voltage
Meters/Radians	13.795	2.738	2.084	=B18-$L	=C18-$	=SQRT(POW(E18, 2)	=D18-N2	Yes	12.4
Inches/Degrees	=B18*39.37	=C18*39.37	=(D18*180)/3.14	=E18*39	=F18*3	=G18*39.37	=(H18*180)/		
	X	Y	Theta (radians)	X Error	Y Error	Magnitude Error (XY)	Theta Error	Successful	Battery Voltage
Meters/Radians	13.791	2.712	2.107	=B22-$L	=C22-$	=SQRT(POW(E22, 2)	=D22-N2	Yes	12.4
Inches/Degrees	=B22*39.37	=C22*39.37	=(D22*180)/3.14	=E22*39	=F22*3	=G22*39.37	=(H22*180)/		

A pathing PID was used to drive the robot to the specified position for coral placements. The accuracy of the control loop was tested 33 times while recording the X, Y, and heading (shown as theta) of the robot once it was placed. The graph of this data and its Gaussian distribution were used to determine the degree of accuracy of the autonomous placement sequence.



The green star is the setpoint, and the 2 red Xs are the times the robot missed the placement. The blue dots are successful placements. A majority of the points lie within a 5 by 5 centimeter square, which shows a high precision but a lower accuracy. The data is up and to the right of the setpoint, meaning the robot overshoots the setpoint. This result meant that a new PID tuning for the autonomous placement sequence was required.

Pathing Tool

Polar Pathing

3piece

part1 - 2.75 seconds

part2 - 4.5 seconds

part3 - 4.5 seconds

part4 - 1.0 seconds

Add Branched Path

Add Path

0:01

0:12

Polar Pathing

part2

Commands

Elevator Down
1.0 - 1.2

Intake Coral
1.5 - 3.5

Raise 2in
3.0 - 3.1

Elevator L2
3.5 - 4.5

Elevator Mid
4.5 - 4.7

Outake
4.7 - 4.8

Add Command

0:03

0:04

Draw

Edit

Commands

Design Requirements

The set of design requirements to be met with a new pathing tool were as follows:

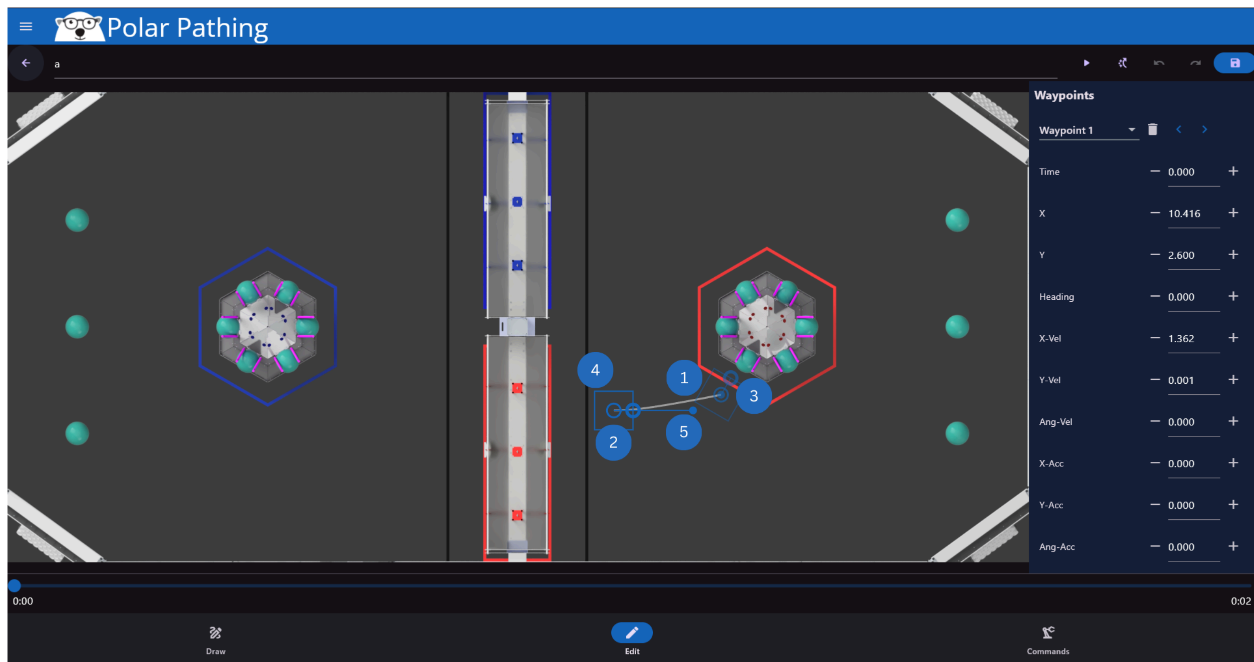
- Have a user-friendly UI/UX (intuitive controls/displays).
- Control paths with key point position, velocity, and acceleration.
- Upload and download autos to and from the robot.
- Visual feedback in path creation.
- Flexible layout to create room for updates.
- Incorporation with state machine.
- Different optimizations at different parts of the path.
- Branching based on conditions.
- Command management.
- Configurable for multiple games/robots.

To meet these requirements, the following solutions were incorporated:

- Flutter was used to build a user-friendly UI/UX with customizable theming.
- Use quintic Hermite splines to generate path equations.
- Upload and download support directly from Roborio
- Path animation, waypoint markers, path curves, and command curve highlighting.
- Easily deployable with a single JSON file.
- Easy code-side configuration
- Multiple custom path-following algorithms to optimize for
- Branching support based on robot configuration
- Command support based on robot configuration
- Configurable with robot and field config files

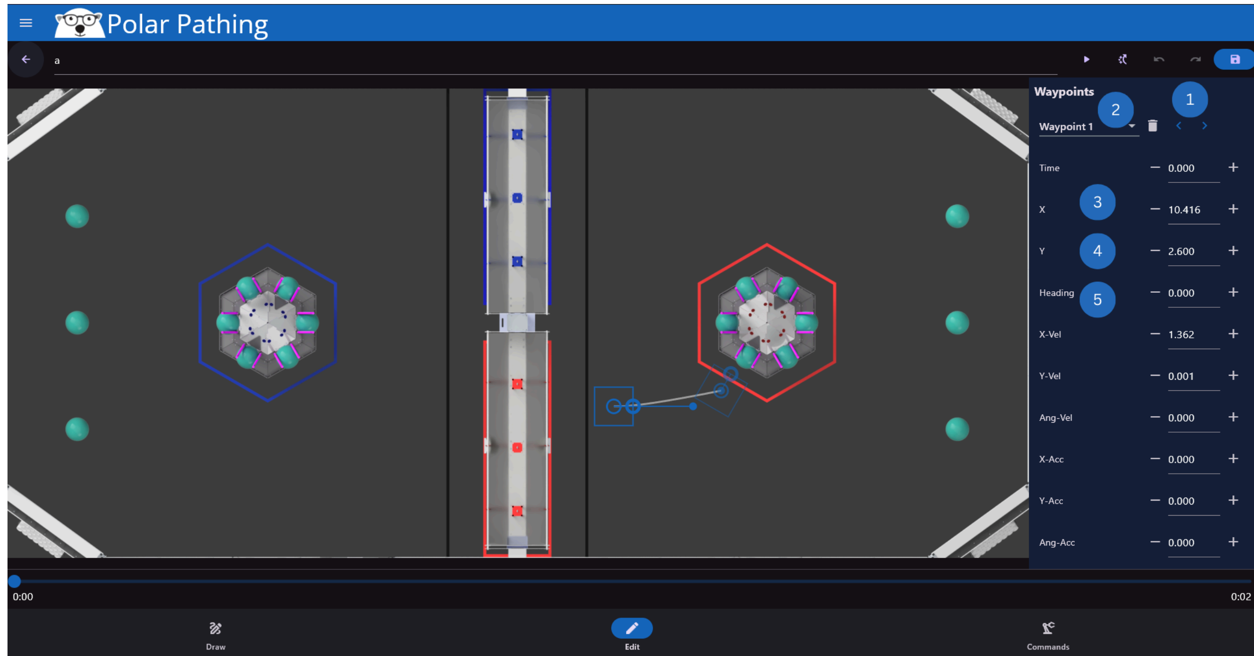
Labelled diagrams and Descriptions

Path display



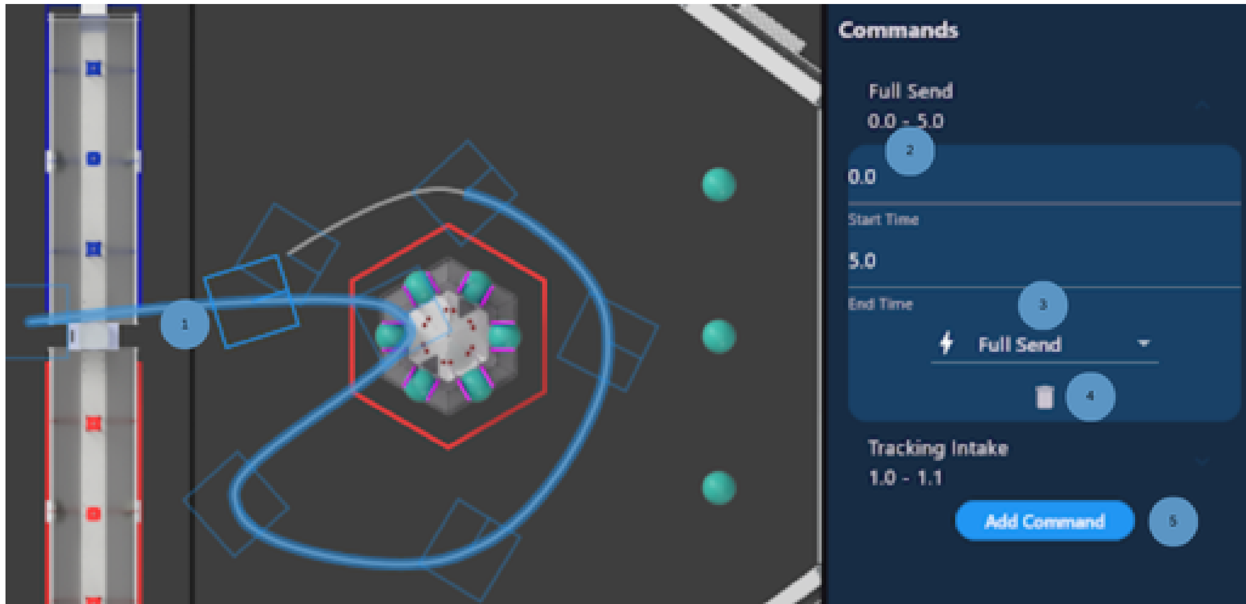
1 - Path Line	Shows what path the robot will follow.
2 - Robot Indicator	Shows where the edges of the robot should be at that key point.
3 - Robot Angle Indicator	Shows at what angle the robot should be facing at that key point.
4 - Selected Point Indicator	Indicates that that key point is currently selected.
5 - Linear Velocity Indicator	Indicates the magnitude and direction of the robot's linear velocity at that key point.

Key point menu



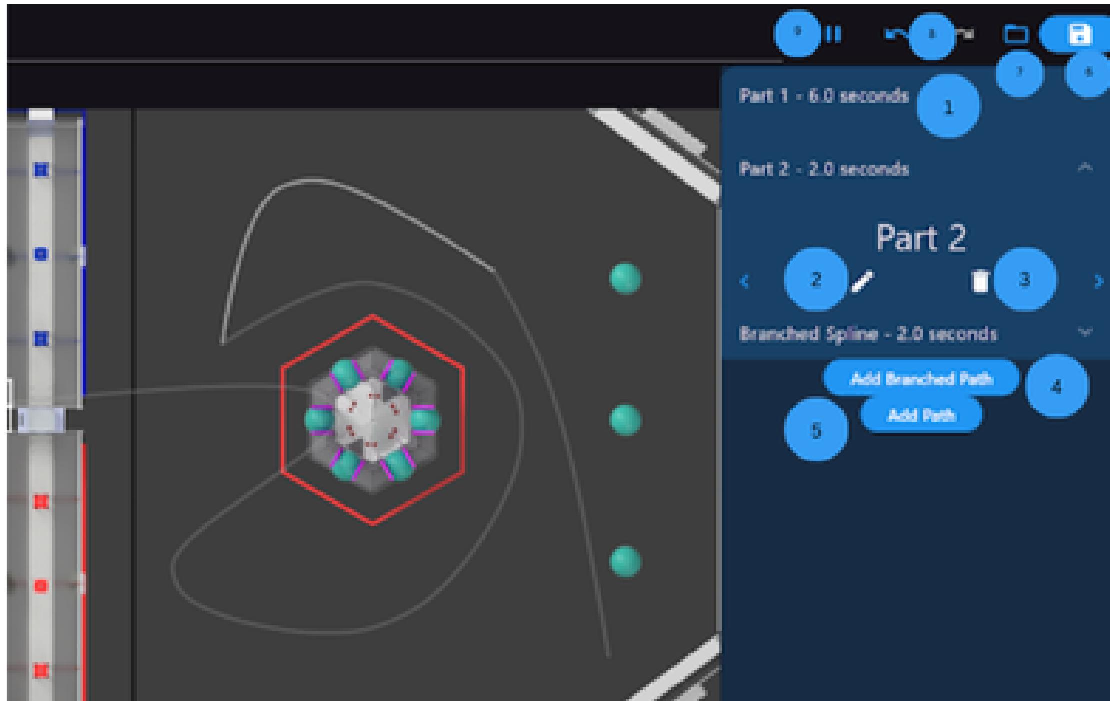
1 - Reindex Point Button	Switches the selected key point with the point before or after it, controlled with the buttons labeled “<” and “>”. This is useful for inserting key points into the middle of a path.
2 - Waypoint dropdown	Dropdown list to select any key point and edit the specifics of it.
3 - X edit field	Here the X value can be adjusted.
4 - Y edit field	Here the Y value can be adjusted.
5 - Angle edit field	Here the heading value can be adjusted.

Commands menu



1 - Highlighted Path	Indicates where on the path the command will run.
2 - Command Start/End Time	Allows users to edit where the command runs on the path.
3 - Command Selector	Allows users to select which command to run.
4 - Delete Command	Removes the command from the list of commands.
5 - Add Command	Adds a command to the list of commands.

Paths menu



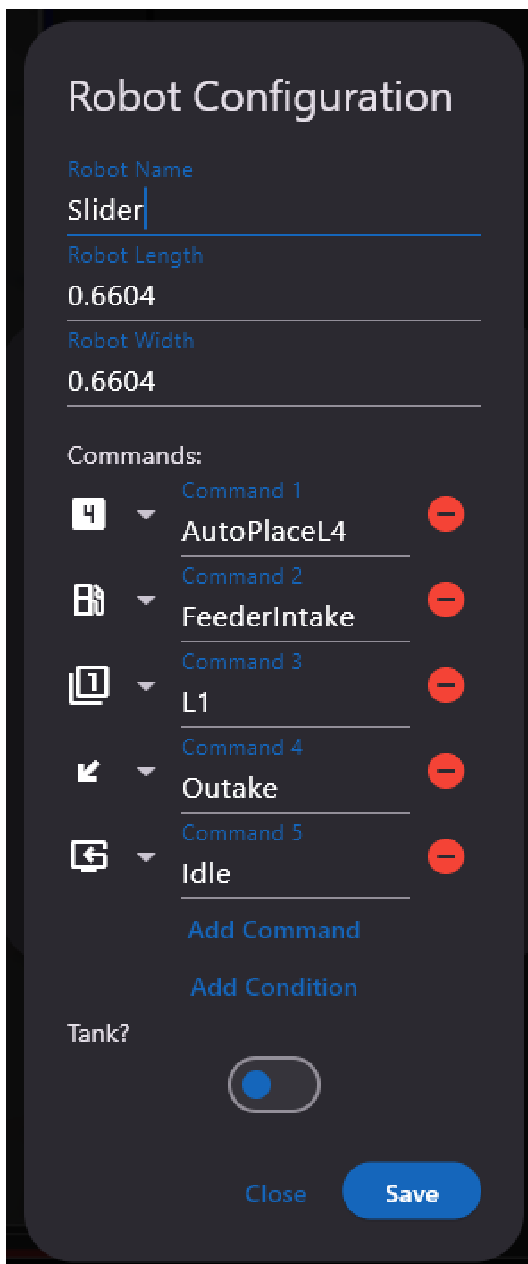
1 - Path Selector	Here users can select the path to use
2 - Path Editor	Clicking this will allow users to open the editor on the current path
3 - Path Deleter	Clicking this will allow users to delete the current path.
4 - Add Branched Path	Adds a branched path to the paths list
5 - Add Path	Adds a path to the paths list
6- Save Button	Opens a window to choose where to save the auto file to
7 - Robot Save Button	Saves the autonomous directly to the robot
8 - Redo and Undo Buttons	Users can redo or undo any action using this button
9 - Play/Pause Button	Clicking this will play or pause the animation

How to use

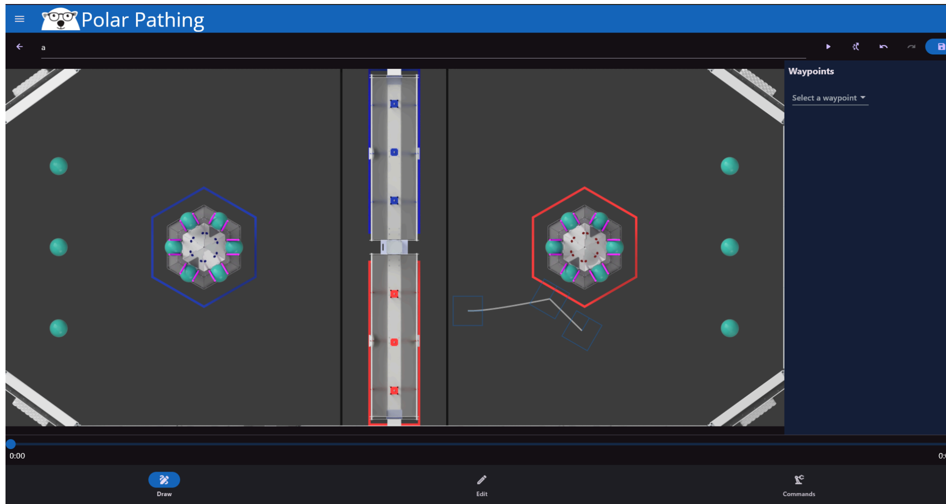
The 2025 Path Tool is capable of creating, refining, debugging, and transferring autonomous path files. Here is a guide that will explain how best to take advantage of the features it has to offer, shown with an example path.

Robot and Field configuration

Users can add different robot configs and field configs. The robot configs hold the robot dimensions and the commands for that robot. The field configuration takes the image, the image dimensions, and the field dimensions in meters.

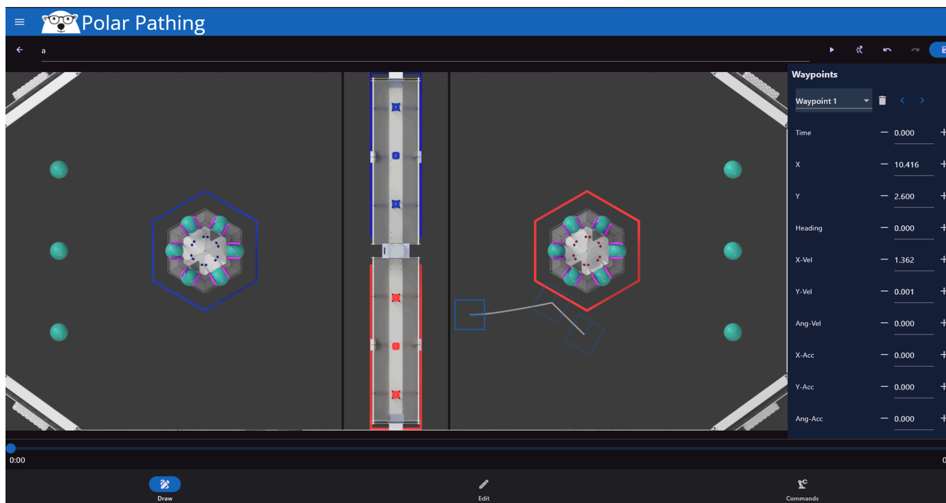


Creating Paths



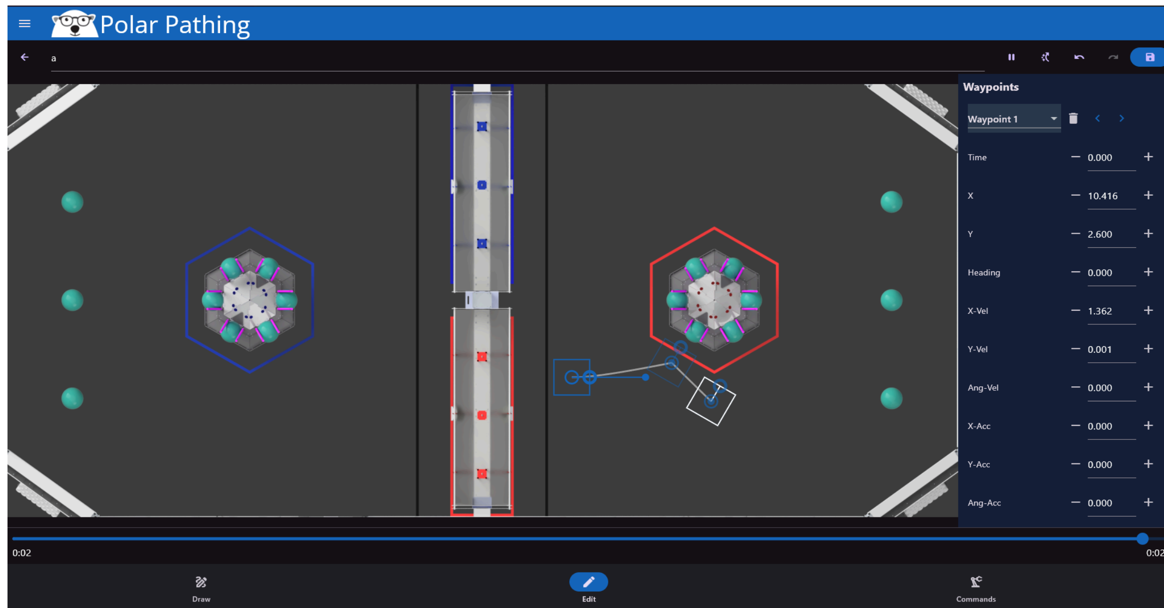
Left-click on the picture of the game field to create new key points. These key points are what define any given path.

Editing Way-points



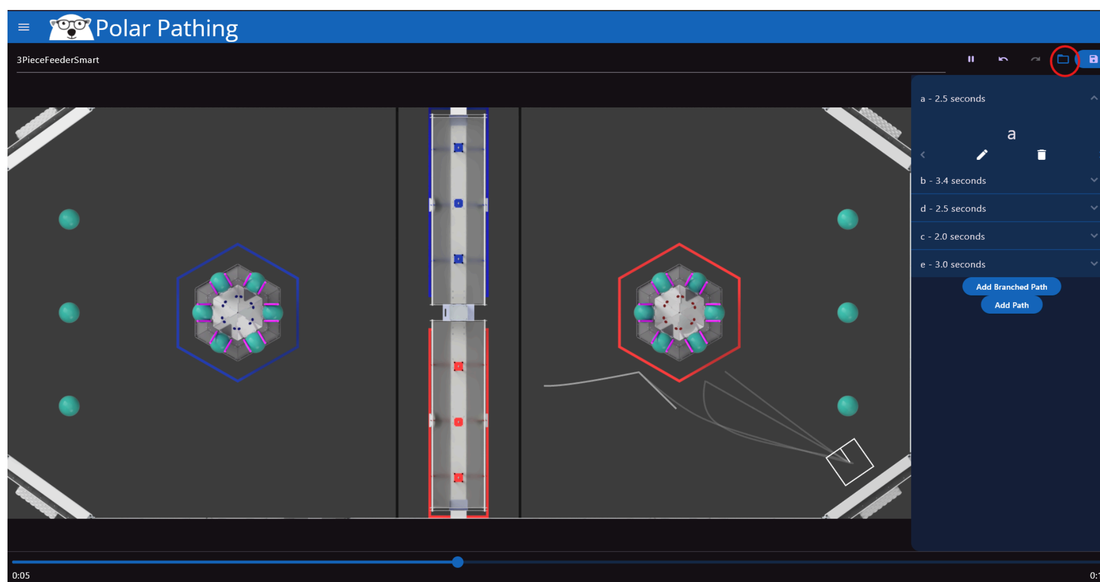
Controls in the editor and waypoint menu can be used to select each key point and edit its parameters to the user's liking. In the example above, each point is rotated, the timings are adjusted, and the whole path is optimized with the "Smoothen" button.

Make sure it looks right



The animation can be run to check that the path was generated correctly at all points.

Upload to RoboRIO



The upload path button uploads the path to the RoboRio (which must be connected to the robot).

Code structure

The structure of the pathing tool was written in a way to increase how intuitive each part of the pathing tool is. Each autonomous has a set of paths. Each of these paths can be edited on its pages. Each path has commands and waypoints. The overall app was given access to a set of global variables: the robot configuration, field image configuration, and preference configuration. These can dynamically change the information shown on each page, making for a simple UI.

Path-generation

The 2025 Path Tool uses quintic Hermite splines to interpolate between key points in a path. This has the benefit of smooth, continuous motion, which reduces robot error when following the path.

Quintic Hermit Splines

There are 3 separate piecewise position equations, which are for x, y, and heading, respectively. To generate these splines, equations formulated by Jan Hünermann were used. These involve multiplying a vector consisting of the waypoint data by a special matrix to get the coefficients to the quintic position function:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\ -10 & -6 & -\frac{3}{2} & 10 & -4 & \frac{1}{2} \\ 15 & 8 & \frac{3}{2} & -15 & 7 & -1 \\ -6 & -3 & -\frac{1}{2} & 6 & -3 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x_0 \\ \dot{x}_0 \\ \ddot{x}_0 \\ x_1 \\ \dot{x}_1 \\ \ddot{x}_1 \end{bmatrix}$$

The output of this matrix multiplication is the quintic function of position in the form: $x = c_0 + c_1 t + c_2 t^2 + c_3 t^3 + c_4 t^4 + c_5 t^5$. In this function, t represents time in the path minus the time of the first point[1].

[1]Jan Hünermann, "Quintic Hermite Splines," February 12, 2020, https://janhuenermann.com/paper/spline2020.pdf.

Velocity and Acceleration Equations

The velocity and acceleration of these functions can be easily found by taking the first and second derivatives of the position function above with respect to time:

$$v = c_1 + 2c_2 t + 3c_3 t^2 + 4c_4 t^3 + 5c_5 t^4$$

$$a = 2c_2 + 6c_3 t + 12c_4 t^2 + 20c_5 t^3$$

It is important to note that these functions return the position, velocity, and acceleration of only one dimension (x, y, or heading), not all of them. This means that to calculate magnitudes, the distance formula must be used with the x and y:

$$D\left(t\right) = \sqrt{x(t)^2 + y(t)^2}$$

Code Implementation

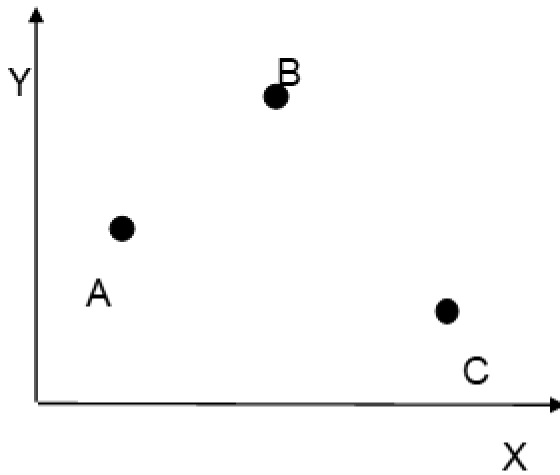
To implement these equations in code, the spline generator uses vector and matrix libraries in Dart to easily multiply the vectors and matrices. Using the calculated coefficients, position, velocity, and acceleration can be evaluated at different times:

```
Matrix position = Matrix.fromList([
          List.generate(6, (int index) {
            return pow(adjustedTime, index) as double;
          }),
        ]) *
        segmentCoefficients[segmentIdx];
    Matrix velocity = Matrix.fromList([
          List.generate(6, (int index) {
            return index == 0
                ? 0
                : index * pow(adjustedTime, index - 1) as double;
          }),
        ]) *
        segmentCoefficients[segmentIdx];
    Matrix acceleration = Matrix.fromList([
          List.generate(6, (int index) {
            return index <= 1
                ? 0
                : index * (index - 1) * pow(adjustedTime, index - 2) as double;
          }),
        ]) *
        segmentCoefficients[segmentIdx];
```

These vector dot products will result in a scalar with the desired value. To display these as continuous functions, points are chosen in 0.01-second intervals and graphed parametrically over the game field.

Optimization

The linear and/or angular components of a path can optionally be converted into Catmull-Rom splines, which helps to smooth out corners by optimizing the velocities of key points interior to the path (key points that are not the first or last point). The math to calculate these velocities is fairly simple: just set the velocity of the key point equal to the average velocity based on the key points before and after it.



In the example above, the x-velocity at key point B could be determined with the following equation:

$$x'_B = \frac{x_C - x_A}{t_C - t_A}$$

Angle Optimization

Because quintic Hermite splines interpolation angles between key points as a continuous quantity, the direction of robot rotation between key points is often not optimal because the 180/-180 degree flipping point is not taken into account. To fix this, the angles of the key points are optimized sequentially by determining which direction requires the least amount of rotation to reach the next angle. If rotation in the positive direction is optimal, the angle of the next key point will increase from the current angle, and if negative rotation is optimal, the next angle will decrease. This ensures that the generated path equations will not cause the robot to rotate more than 180 degrees between key points.

```
void optimizeRotation() {
    for (int i = 1; i < points.length; i++) {
      var p1 = points[i - 1];
      points[i] = points[i].copyWith(theta: points[i].theta % (2 * pi));
      while (true) {
        if (points[i].theta - p1.theta > pi) {
          points[i] = points[i].copyWith(theta: points[i].theta - 2 * pi);
        } else if (points[i].theta - p1.theta < -pi) {
          points[i] = points[i].copyWith(theta: points[i].theta + 2 * pi);
        } else {
          break;
        }
      }
    }
  }
```

File Handling

Paths are saved as JSON files in the format shown to the right:

```json
{
    "meta_data": {
        "path_name": "Path_Name",
        "sample_rate": 0.0
    },
    "commands": [
        {
            "branched": true,
            "branchedCommand": {
                "condition": "right_trigger",
                "on_true": {
                    "name": "arm_up",
                    "start": 0.5,
                    "end": 1.0
                },
                "on_false": {
                    "name": "arm_down",
                    "start": 0.5,
                    "end": 1.0
                }
            },
            "start": 0.5,
            "end": 1.0
        },
        {
            "branched": false,
            "command": {
                "name": "arm_up",
                "start": 1.0,
                "end": 1.5
            },
            "start": 1.0,
            "end": 1.5
        }
    ],
    "key_points": [
        {
            "index": 0,
            "delta_time": 0.0,
            "time": 0.0,
            "x": 0.0,
            "y": 0.0,
            "angle": 0.0,
            "x_velocity": 0.0,
            "y_velocity": 0.0,
            "angular_velocity": 0.0,
            "x_acceleration": 0.0,
            "y_acceleration": 0.0,
            "angular_acceleration": 0.0
        }
    ],
    "sampled_points": [
        {
            "time": 0.0,
            "x": 0.0,
            "y": 0.0,
            "angle": 0.0,
            "x_velocity": 0.0,
            "y_velocity": 0.0,
            "angular_velocity": 0.0,
            "x_acceleration": 0.0,
            "y_acceleration": 0.0,
            "angular_acceleration": 0.0
        }
    ]
}
```

There are 3 components of a save file: meta-data, key points, and sampled points.

The meta-data specifies the name of the path and the sample rate (in seconds) at which interpolated points are sampled at.

The key points are only included to allow paths to be downloaded and reconstructed by the path tool.

The sampled points are interpolated in a list for the robot code to follow (in this case, with a PID on the drivetrain).

Full autos are saved with their child paths all in one JSON file in the format shown below:

```json
{
    "meta_data": {
        "path_name": "Auto_Name",
        "sample_rate": 0.0
    },
    "schedule": [
        {
            "branched": false,
            "path": 0
        },
        {
            "branched": true,
            "condition": "right_trigger",
            "branchedPath": {
                "onTrue": 1,
                "onFalse": -1
            }
        }
    ],
    "paths": [ ...
    ]
}
```

There are three main parts to the autos: the metadata, schedule, and paths. The metadata is used to identify which auto is chosen. The schedule is used to communicate which paths need to be run and under which conditions. The paths to be run are given by the index; they are in the 'paths' array and have the same schema as path files.

Robot-side Code

Parsing the Autonomous JSON

In an optimal codebase, all of the robot code should be configurable from the RobotContainer.java and Constants.java files. To accomplish this optimization, a series of files for parsing the autonomous files were written. The main file is a PolarAutoFollower.java, which can use the parsed JSON to recursively schedule PolarPathFollower commands.

The PolarPathFollower.java file has two main sub-items: The AutoFollower and the list of commands. A recursive function was used to parse out all sub-commands:

```java
private Command addCommandsFromJSON(JSONObject command, HashMap<String, Supplier<Command>> commandMap,
    HashMap<String, BooleanSupplier> conditionMap) {
  if (command.has("command")) {
    return singleCommandFromJSON(command, commandMap);
  } else if (command.has("branched_command")) {
    BooleanSupplier startSupplier = () -> command.getDouble("start") < getPathTime();
    BooleanSupplier endSupplier = () -> command.getDouble("end") <= getPathTime();
    JSONObject onTrue = command.getJSONObject("branched_command").getJSONObject("on_true");
    JSONObject onFalse = command.getJSONObject("branched_command").getJSONObject("on_true");
    BooleanSupplier condition = conditionMap.get(command.getJSONObject("branched_command").getString("condition"));
    return new TriggerCommand(
        startSupplier,
        new ConditionalCommand(
            addCommandsFromJSON(onTrue, commandMap, conditionMap),
            addCommandsFromJSON(onFalse, commandMap, conditionMap),
            condition),
        endSupplier);
  } else if (command.has("parallel_command_group")) {
    ArrayList<Command> commands = new ArrayList<Command>();
    for (int i = 0; i < command.getJSONObject("parallel_command_group").getJSONArray("commands").length(); i++) {
      commands.add(addCommandsFromJSON(
          command.getJSONObject("parallel_command_group").getJSONArray("commands").getJSONObject(i), commandMap,
          conditionMap));
    }
    return new TriggerCommand(() -> command.getDouble("start") < getPathTime(),
        new ParallelCommandGroup(
            commands.toArray(new Command[0])),
        () -> command.getDouble("end") <= getPathTime());
  } else if (command.has("parallel_deadline_group")) {
    Command deadlineCommand = addCommandsFromJSON(
        command.getJSONObject("parallel_deadline_group").getJSONArray("commands").getJSONObject(0), commandMap,
        conditionMap);
    ArrayList<Command> commands = new ArrayList<Command>();
    for (int i = 1; i < command.getJSONObject("parallel_deadline_group").getJSONArray("commands").length(); i++) {
      commands.add(addCommandsFromJSON(
          command.getJSONObject("parallel_deadline_group").getJSONArray("commands").getJSONObject(i), commandMap,
          conditionMap));
    }
    return new TriggerCommand(() -> command.getDouble("start") < getPathTime(),
        new ParallelDeadlineGroup(
            deadlineCommand,
            commands.toArray(new Command[0])),
        () -> command.getDouble("end") <= getPathTime());
  } else if (command.has("parallel_command_group")) {
    ArrayList<Command> commands = new ArrayList<Command>();
    for (int i = 0; i < command.getJSONObject("parallel_race_group").getJSONArray("commands").length(); i++) {
      commands.add(addCommandsFromJSON(
          command.getJSONObject("parallel_race_group").getJSONArray("commands").getJSONObject(i), commandMap,
          conditionMap));
    }
    return new TriggerCommand(() -> command.getDouble("start") < getPathTime(),
        new ParallelRaceGroup(
            commands.toArray(new Command[0])),
        () -> command.getDouble("end") <= getPathTime());
  } else if (command.has("sequential_command_group")) {
    ArrayList<Command> commands = new ArrayList<Command>();
    for (int i = 0; i < command.getJSONObject("sequential_command_group").getJSONArray("commands").length(); i++) {
      commands.add(addCommandsFromJSON(
          command.getJSONObject("sequential_command_group").getJSONArray("commands").getJSONObject(i), commandMap,
          conditionMap));
    }
    return new TriggerCommand(() -> command.getDouble("start") < getPathTime(),
        new SequentialCommandGroup(
            commands.toArray(new Command[0])),
        () -> command.getDouble("end") <= getPathTime());
  } else {
    throw new IllegalArgumentException("Invalid command JSON: " + command.toString());
  }
}
```

This function first checks which type of command it is by checking for each overarching command type.

If the chosen command has a set of subcommands, the function will recursively call itself to schedule more commands. If the chosen command is not a set of subcommands, the

function will add the command by using the HashMap of commands set up in the RobotContainer.java file:

```
HashMap<String, Supplier<Command>> commandMap = new HashMap<String, Supplier<Command>>() {
        {
                put(key:"AutoPlaceL4", () -> new AutoPlaceL4Follower(superstructure, drive, timeout:2.3));
                put(key:"AutoFeeder", () -> new FeederPickupFollower(superstructure, drive));
                put(key:"FeederIntake", () -> new SetRobotState(superstructure, SuperState.FEEDER));
                put(key:"Outake", () -> new SetRobotStateSimple(superstructure, SuperState.OUTAKE));
                put(key:"L1", () -> new SetRobotStateSimple(superstructure, SuperState.AUTO_L1_PLACE));
                put(key:"Idle", () -> new SetRobotStateSimple(superstructure, SuperState.IDLE));
                put(key:"Full Send", () -> new FullSendFollower(drive, pathPoints:null, record:false));
        }
};
```

The command is then set to run with a start time and end time to ensure the command times out when needed.

When scheduling a command, the PolarPathFollower also checks if the command is a subtype of AutoFollower. If it is a subtype of AutoFollower, then the PolarPathFollower, for the allotted time, lets the command take control of the drivetrain, replacing the job of the default until the command finishes. When the command finishes, the default follower is applied once again. This allows the use of different types of path followers, which are each optimized for different objectives.

The default follower is optimized to follow the path with accurate velocities and command timings. The other followers implemented include: FullSendFollower, AutoPlaceFollower, and FeederPickupFollower. The FullSendFollower is optimized to maximize speed along a path, which can be useful when spanning large sections of the field. The AutoPlaceFollower and FeederPickupFollower commands are optimized for accuracy and end upon placement and coral intake, respectively.

Path Following Algorithms

In previous years, time-based followers were used in which if the robot got behind in a path, it would try to cut corners to reach the path's desired position at that time. To reduce this issue, a pure pursuit path following was implemented. Instead of choosing which point to run PID control to using time, this algorithm finds the next point in the path that has not been reached. This way, the path follower never has to cut a corner.

The problem with choosing just the next point, though, is that with 1 centisecond of granularity and 2-3 centiseconds of schedular loop runtime, the autonomous will function at a rate 2-3 times slower than expected. To counteract this, the algorithm chooses a point that is farthest within a specified radius from the robot. This extra "look-ahead distance" allows the robot to run its PID control loops on regular distance setpoints, leading to a constant speed of the robot.

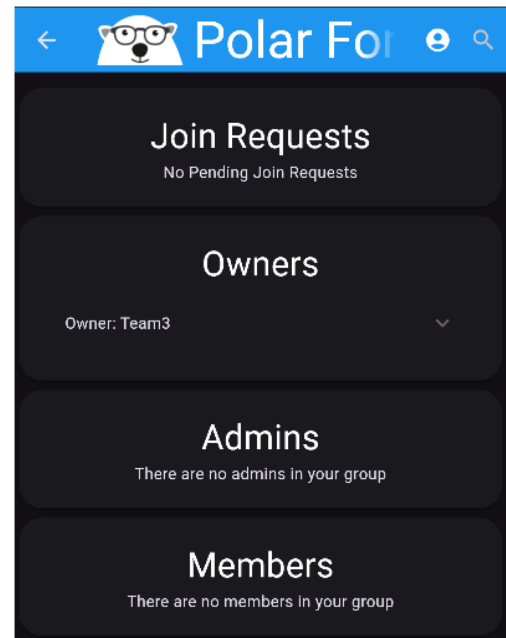
Even this path-following system is not perfect, however, because if a change in robot speed is required in a path, this form of following cannot provide it. This makes it useful in the FullSendFollower because of the need to go at a consistently fast rate, no matter how many points are within the look-ahead radius.

To allow variable speeds, the default follower scales the look-ahead distance dynamically by multiplying it by the desired velocity at that point. This allows the robot to slow down because as the look-ahead distance gets smaller, the robot moves slower. This also does not cut corners because it checks that it has crossed every point on the path. One issue with this algorithm is that if the velocity at a certain point within a path was set to 0, the look-ahead distance would also be 0, leading to the robot standing still and never moving. To stop this, the look-ahead distance was calculated by adding a small feed-forward look-ahead to the variable look-ahead.

Polar Forecast

Scouting is a small but crucial part of team 4499. It is the process of choosing alliance partners who synergize effectively. Although this task may sound easy, digging any deeper than the bare minimum is a complex act, which is why the Highlanders have implemented Polar Forecast this year. Polar Forecast is a web app that can display and collect scouting data. This data is sent to a backend, where complex analysis using machine learning is performed.

Groups and Alliances

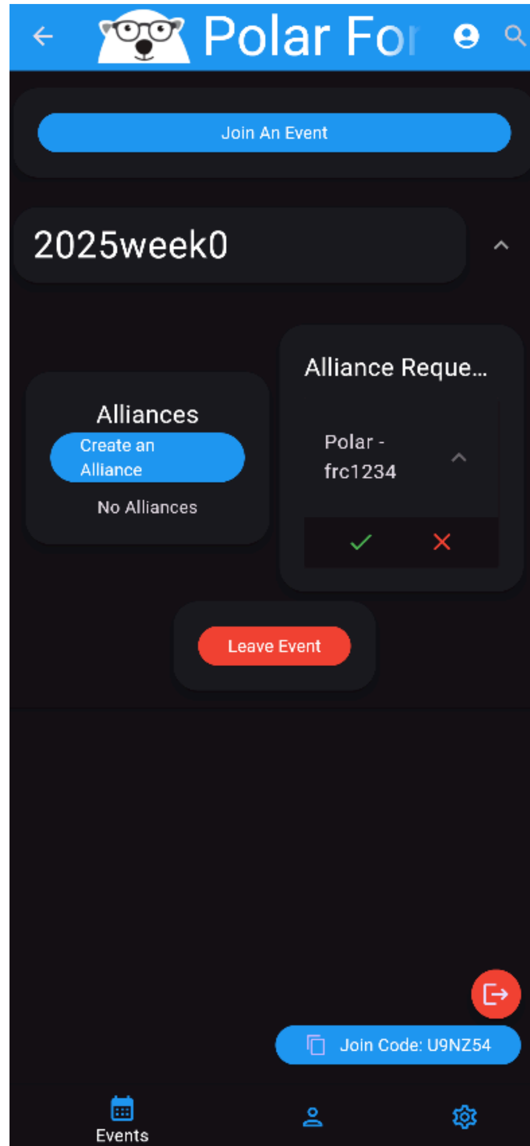


Each account can create a group. Users can only join a group associated with the same team number that the owner has. To create a group, click the profile icon and then click on groups. Once created, the join code will appear in the bottom right corner. On click, it will open up a QR code to join and automatically copy the join link to the clipboard. This link can be shared to add other users to the group. Once the user clicks on the link, the system will send a join request. Users can then click on the "Members" tab to see join requests and all members in the group, sorted by their permission level. There can only be one owner in a group, but the owner has full control of everything in the group. Admins can manage data (pit scouting, match scouting, pictures), join events, join alliances, kick, invite, or accept members. Members can submit data and view the analyzed data.

Alliances

At each event, groups can choose to collaborate their data with other teams using alliances. Groups can invite other groups at each event to join a scouting alliance. To create an alliance on the group page, users can go to the "events" tab. Here, the group can be added to an event. Once the group joins an event, click the "Create an Alliance" button to start inviting other groups to join. Alliances can be

accepted in the same place where invitations are sent. Once in an alliance, all data from these teams is visible to each other, but if a team decides to withdraw from the alliance, their data will be hidden once again.



Pit Scouting

To reduce communication errors and duplicate entries, a way to check this information beforehand is built into Polar Forecast. The status system is built so that whenever a team has been pit scouted, their pit scouting status goes from “Not Started” to “Done”. If the form was left incomplete, then it shows “Incomplete”. A similar thing is done with picture collection, and upon upload, the backend changes the picture status to “Done”. To access the input form for the data, click on its status. For example: to pit scout a team, click on their status in the “Pit Scouting” tab of the event page.

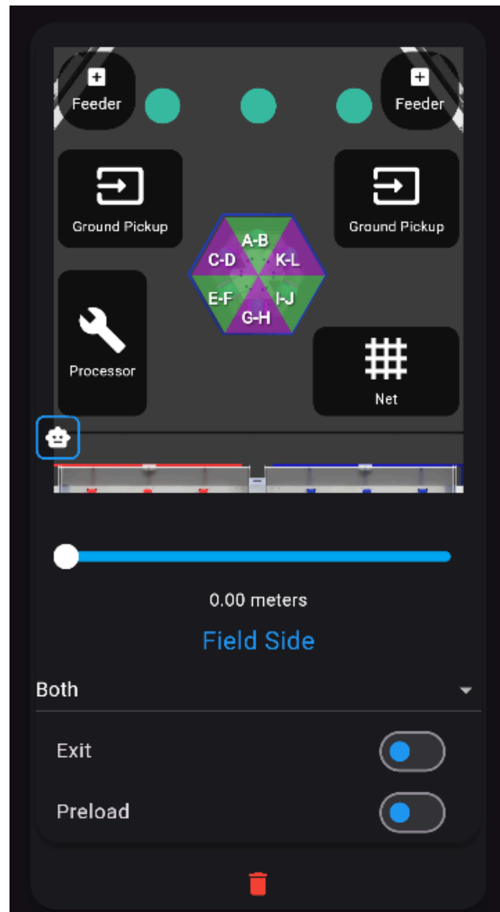
Polar Forecast 2025 Week 0 [week0]

Team	Pit Scouting	Pictures	Follow Up
88	Not Started	Not Started	Done
97	Not Started	Not Started	Done
166	Not Started	Not Started	Done
190	Not Started	Not Started	Done
238	Not Started	Not Started	Done
246	Not Started	Not Started	Done
501	Not Started	Not Started	Done
509	Not Started	Not Started	Done
811	Not Started	Not Started	Done
1058	Not Started	Not Started	Done
1073	Not Started	Not Started	Done
1119	Not Started	Not Started	Done
1153	Not Started	Not Started	Done
1721	Not Started	Not Started	Done
1729	Not Started	Not Started	Done
1768	Not Started	Not Started	Done

Pit Scouting

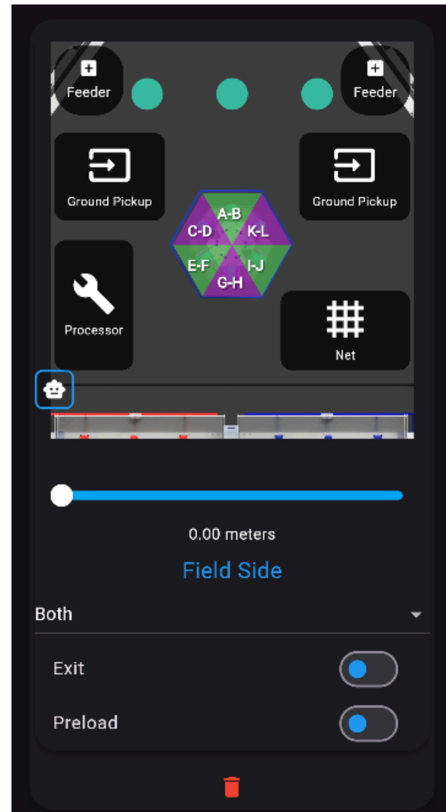
Auto Scouting

When a user is trying to scout an auto, clicking on the team's pit scouting area and then using the "Add Auto" button will open a card as shown below:

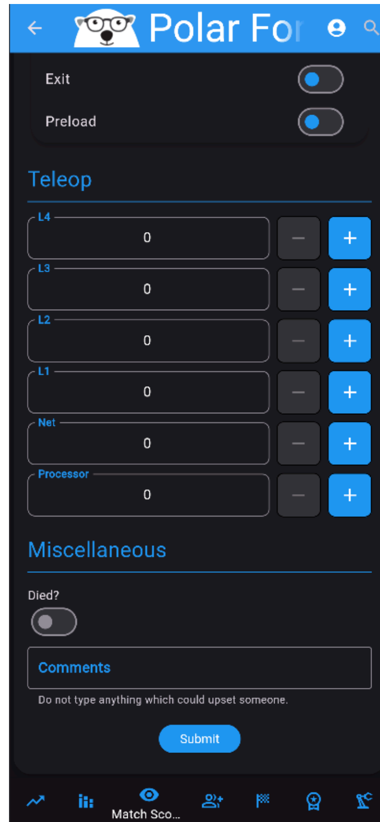


Clicking on each different button will add a step to the list. Steps can then be deleted or reordered by holding and dragging. This data will help plan autos with teams during a match and will help pick teams with compatible autos in a scouting meeting.

Match Scouting



The match scouting form has the same form for autonomous as pit scouting, but with extra buttons to indicate if the team succeeds on the piece they are attempting. For tele-op, the scouting is just a counter for each game element that is expected to be updated as elements are scored. It is important to note that climb data is not recorded, as perfect climb data can be calculated using data from The Blue Alliance.



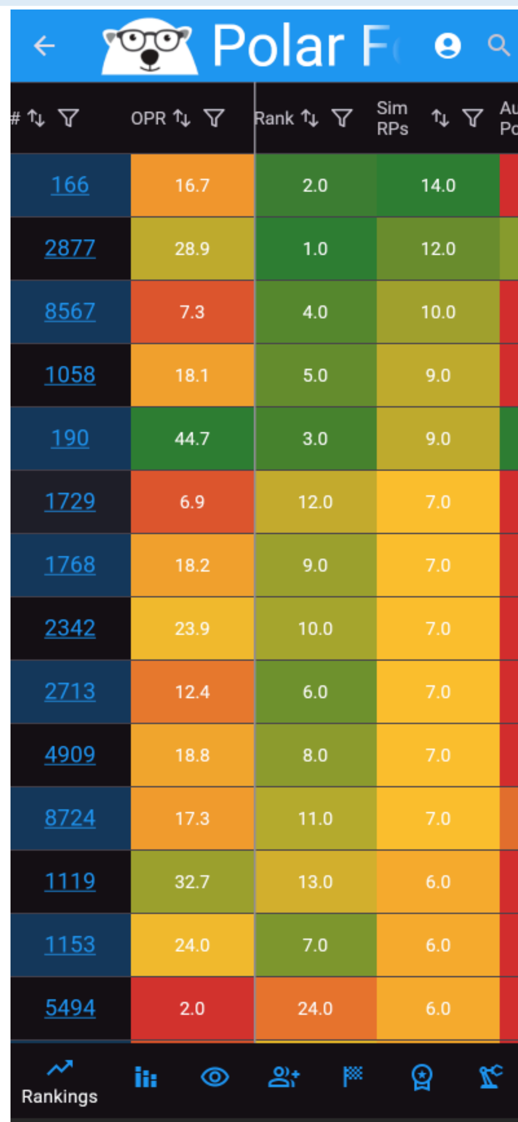
Data Verification

To ensure that scouts are entering good data, simple checks are run on the metadata fields. The first check is making sure that the scout has inputted a name. The next check is to see if the given match number exists. The backend next checks if the given match has the given team number. If the scout passes these checks, then the system will submit the entry instantly. If the scout fails any of these checks, the scouting data form will display the error so that the scouts can fix it. The last implementation is making sure that a scout didn't submit more than once. To stop this, the database checks if there is another entry with the same scout name, match number, and team number. If any entry matches, then the system will allow the scout the option to overwrite that entry. This set of checks minimizes the risk of duplicate, and incorrectly labeled data.

Data Display

When displaying information, it is all about making the information easy to access and view, but at the same time making it functional. The list of requirements for data displays is rigid, as they are the most vital part of the scouting app. 5 pages in the scouting app display data: Event, Team, Match, Pit Scouting, and Picture Scouting.

Event Page



#	OPR	Rank	Sim RPs
166	16.7	2.0	14.0
2877	28.9	1.0	12.0
8567	7.3	4.0	10.0
1058	18.1	5.0	9.0
190	44.7	3.0	9.0
1729	6.9	12.0	7.0
1768	18.2	9.0	7.0
2342	23.9	10.0	7.0
2713	12.4	6.0	7.0
4909	18.8	8.0	7.0
8724	17.3	11.0	7.0
1119	32.7	13.0	6.0
1153	24.0	7.0	6.0
5494	2.0	24.0	6.0

Rankings

When displaying rankings, only relevant information was kept on the event page. Each team's rank, simulated ranking points, auto score, auto coral points, teleop coral points, net, processor, climb rate, OPR, and death rate are shown on this page. This data is organized into a table and can be easily sorted for quick analysis. To see more granular stats of a team, users can click on each team's team number.

Charts

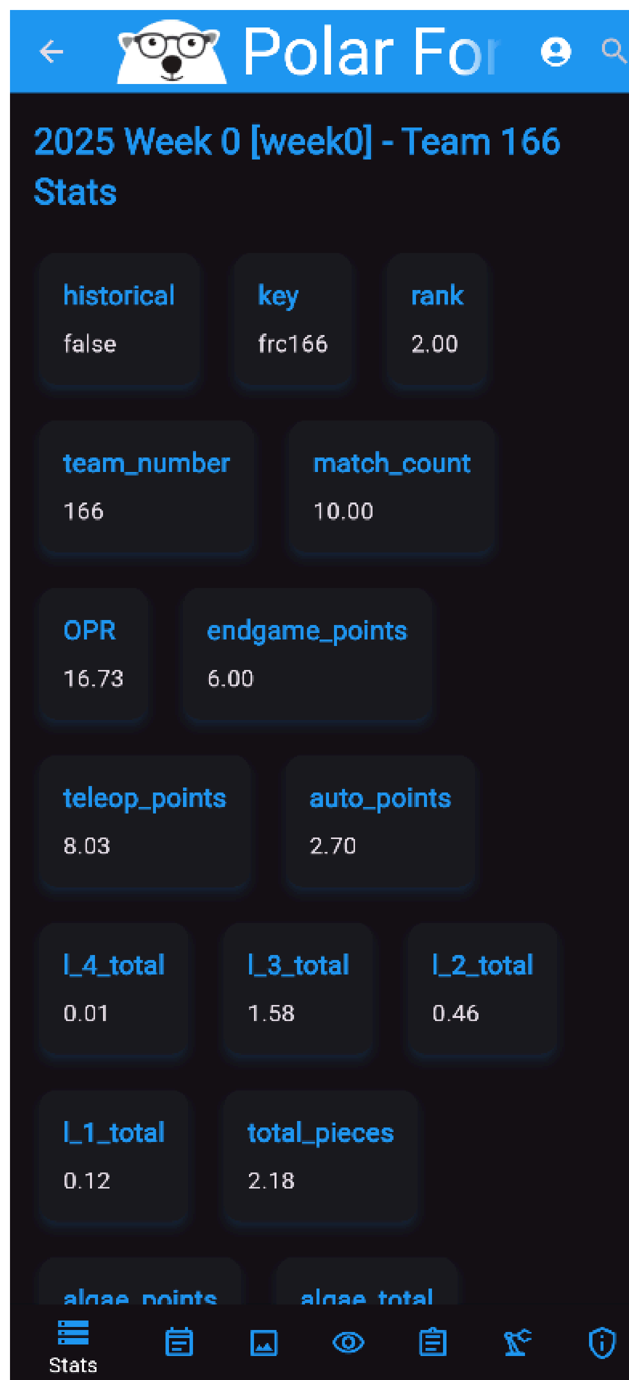
To display data more visually, a charts page was added. The match-by-match chart allows direct comparison between two different robots by clicking the "Select a Team" dropdown. The compare button on this chart allows users to directly compare two teams' stats. This chart page contains column graphs that have detailed breakdowns of OPR and cycles. These sub-fields, such as autonomous scoring, can be assigned a weight to increase or decrease the value that a field has. This is so that teams can be ranked based on metrics other than just points. These charts are displayed in the "Charts" tab on the scouting website.

Schedule

The event page has two tabs for displaying schedules, "quals" and "eliminations". These tabs contain a table that shows the results or predictions for the match. When a user clicks on a match, it takes them to the match page.

Stats

The team page has a "Stats" tab in which all granular metrics are displayed. This has a simple list of each of the stats and attributes the team has.



Data Analysis

The data analysis used in Polar Forecast has two key areas: data simulation and algorithm creation.

Data Simulation

Simulating a tournament can be useful when testing analysis algorithms. 3 main sources of data must be simulated to test analysis algorithms: true data, The Blue Alliance data, and scouting data. True data records exactly what happened in every match, including who, what, and where it was scored. The Blue Alliance (TBA) data is a simulation based on the true data, which does not say which team on each alliance scored. Scouting data is simulated to try and mimic the nature of fallible human scouts, recording most data, but with error added as well.

True Data

To simulate true data, many factors must be randomized. The first step was to choose a schedule that would tell us how many teams would compete at the tournament and which matches they would be playing in. Instead of making a custom schedule, a pre-generated schedule created for cheesy-arena by FRC team 254 was used. Their work can be found here: https://github.com/Team254/cheesy-arena/tree/main/schedules.

The next step in the process was to randomize the stats of the teams while still keeping things realistic. To do this, the team stats were randomized with archetypes. These archetypes of robots could score in different places and have different abilities. They each have a set of autonomous schedules that they could choose from to run, and had some baseline amount of scoring, with some amount of random increment or decrement to make it more random. These robot archetypes and their given randomizations create a realistic simulation for true data. The true data came in JSON format, following closely with The Blue Alliance’s format, but with additional fields recording which teams scored.

The Blue Alliance Data

The next step was to translate true data into the language of TBA. To do this, additional fields relating to which teams scored were removed. Errors were not simulated because this data is assumed to be perfect.

Human-Input Scouting Data

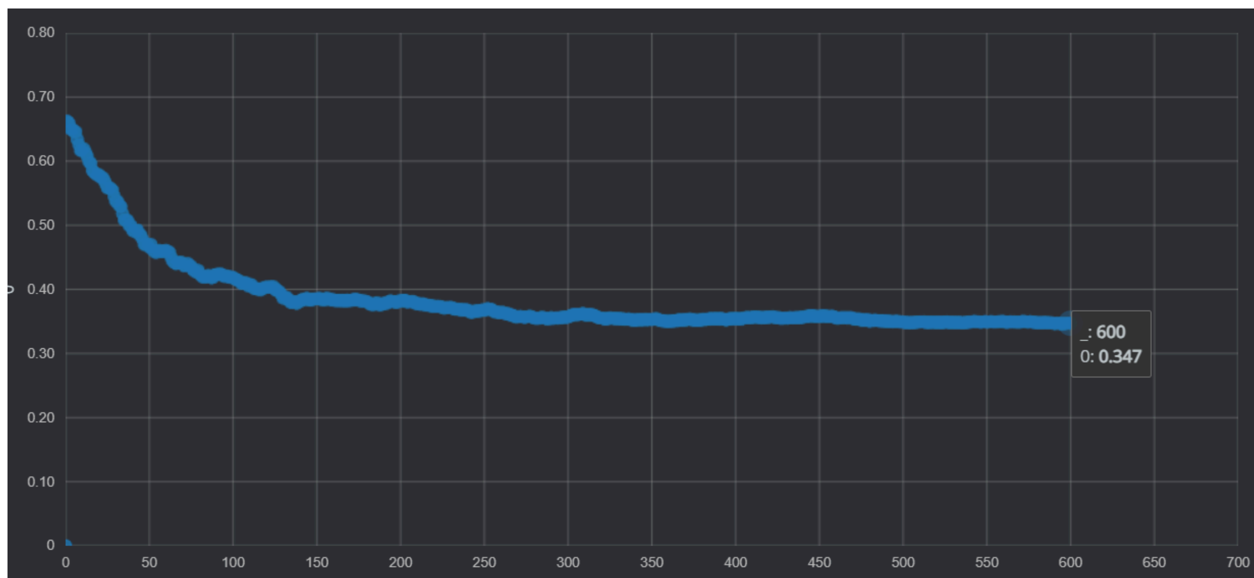
The last step was to simulate the human-entered scouting data. This was crucial because it shows exactly which team scored, data that cannot be extracted from TBA. This would not be as simple as the blue alliance data, as there is a different format for this set of data, and scouts are human, and have errors.

Scout archetypes were created to randomize this error. The simulation had a set number of scouts, each with an archetype that denoted the number of matches they would scout and the amount of error they would have when measuring each of the metrics. Each scout additionally had a probability of scouting each match to randomize which matches they decided to scout. This data was stored in JSON format.

```
class Scout: ...
class Vaclav(Scout): ...
class Vishnu(Scout): ...
class Dmitri(Scout): ...
class Ivan(Scout): ...
class Adarsh(Scout): ...
class Rishabh(Scout): ...
class Ben(Scout): ...
class Roarke(Scout): ...
class Deepesh(Scout): ...
class Rwad(Scout): ...
```

Simulation Usage

By using this simulation, the algorithm could be run multiple times to test different aspects of an event. The most important metric from simulation is the error, which gives insight into how well each algorithm performs. It can be calculated by finding the difference between calculated averages and true averages. This is a graph showing the error of data (game pieces of error per field) as the number of scout entries changes in a scouting data-only algorithm:



TBA-Only Regression

At first, multivariate regression with TBA data was used to calculate the OPRs. In the regression, each team is a variable, whose value is 0 or 1 based on if the team is playing in the given match. Each of these variables has a coefficient which is exactly what is being calculated – the contribution of each team. If the team is not in the match, then their contribution will be 0 for that match, which follows the way it works in the robot game. According to simulation data, after using multivariate regression on TBA data, the estimate was off by a margin of about 3.0 game pieces per team (in all fields, so about 0.6 pieces error per field).

Scouting and TBA Regression

Next, scouting data was incorporated into the algorithms. This was done by setting the coefficient of the scouted team to 1 in the matrix and filling in the rest of the values from the scouting data. It may seem that this data would decrease the accuracy of the regression data. However, it turns out that even though the data was flawed, it helped guide the regression closer to the true data because as more entries are submitted, the mean tends towards the true data. This resulted in about 2.6 game pieces of error per team, reducing the error by about 15%.

Machine-Learning based Regression

The next step towards great data was creating a machine learning algorithm to further improve the accuracy of the data. With pure regression, some teams had impossible contributions, such as negative or greater than the maximum number of game pieces. To solve this, a machine learning algorithm was created using a genetic algorithm. To simplify, it simulates evolution in a population using mutation, reproduction, and natural selection. In this case, “genes” are the team’s contribution, and the “organisms” are the arrays of data holding each of the individual contributions. The way this “Genetic Algorithm” works is complex, but it can be broken into 3 parts: mutation, reproduction, and natural selection.

Mutation

The mutation algorithm works by picking random genes, and giving them a random value using a normal distribution. The two factors in mutation are the mutation rate, or how often a mutation occurs in a gene, and the mutation distance, which is how large each mutation is.

Reproduction

The reproduction algorithm takes an input of two “parent organisms” to generate “offspring”. The algorithm randomly chooses some genes from one parent, and then some from the other parent. The offspring then goes under mutation to introduce new genes to the population.

Natural Selection

Although these two forces will tend toward better data, natural selection must be used to remove bad genes from the population. The algorithm needs to be able to rank these organisms based on their fitness and choose only the best of the population to use for reproduction to create the next population. This means that a singular metric to rank the fitness of these organisms is required.

This calls for an error function, which returns an error value for each organism. The error is calculated by finding the difference between the actual outcomes of the matches and the outcomes predicted by the organism. If a genetic algorithm with only this in the error function was run, then the values would end up the same as in the multivariate regression. To make the calculations more accurate, other custom algorithms must be added to the error function. In Polar Forecast, the error was artificially increased if the value was outside of the possible boundaries, such as if a team was scoring 19 net algae (max of 18) or -1 net algae (min of 0). This error custom error function better tunes the algorithm to create more realistic values than with pure regression.

Shifting gears back to natural selection, after ranking the population by their error, the best individuals in the population are chosen to become parents. The organisms that don't make it to become parents are removed from the population. Each of these parents has an offspring with every other parent, to create a new population. Before adding each offspring into the population, their genes are mutated to introduce new genes to the population. This is because, in evolution, fitness can be thought of as a set of mountains and valleys. Evolution always wants to go upward, but will never go down a slope to try and get on a taller mountain. This is why mutation is used to try and jump across valleys by introducing genes that could be part of a taller mountain. This is a dangerous game, however, because mutated genes could also decrease the accuracy of the data, so to make sure the population is always improving, the parents are kept as part of the next population. This is called "Elitism" as it keeps the elite genes as part of the population.

Populations are created until either no improvement is observed or a "generation timeout" is met. Once the algorithm ends, the singular best organism, out of its thousands of ancestors, becomes the output shown on Polar Forecast.

Algorithm Optimization

When optimizing a genetic algorithm, there are multiple factors that can be optimized: the number of parents, the maximum number of generations, the number of generations since improvement, the gene mutation rate, and the gene mutation distance.

Before optimizing the algorithm, the algorithm already had a low error of 2.1 game pieces but had a slight problem with time efficiency. This algorithm took 2 minutes and 10 seconds on average to run. After optimization, incredible results were achieved: an average error hovering around 1.6 game pieces, and a runtime of 6 seconds. In total, error was reduced by almost 50% by using scouting data and machine learning.

Updating to 2025

Updating the algorithms for 2025 was simple because the algorithms were accurate, only the inputs and outputs were changed. There were three types of inputs that needed to change: the data that can be directly extracted from TBA, the data that can be directly extracted from scouting data, and the data that requires both TBA and scouting data. This year, net scores for each alliance were adjusted for the processor scores made by the opponent alliance.

Predictions and Simulations

To get a better idea of which teams will end up at the top of the leaderboard, the algorithm simulates the remaining matches and compiles a simulated ranking based on the simulated ranking points (RPs) earned by each team. This is helpful because in most tournaments, the night before alliance selection is not the end of quals, but is the time when teams like to create picklists.

Backend and Database

To keep the database private, the app has a backend as a messenger between the database and the front end. This is also where analysis is periodically run.

Periodic Data Updates

To minimize overloading of the lightweight backend, the algorithm is set up to not run constantly. Every 20 minutes, the app runs the analysis algorithm. This alone is not enough to reduce the continual stress of analyzing every tournament in the world, as well as analyzing separate data for each group.

Another strategy used is tracking updates. The TBA API makes this simple because it has an ETag system that tracks changes for us. If there are no updates, the API will send a 304 error. The other set of updates checking needed is scouting data. To keep track of updates in the scouting data, it stores an "up_to_date" field. Whenever the backend receives scouting data, it changes the value of this field from true to false. Whenever the system updates, the value is set to true. This way, the algorithm can check for updates, and won't run analysis on up-to-date data.

REST API Endpoints

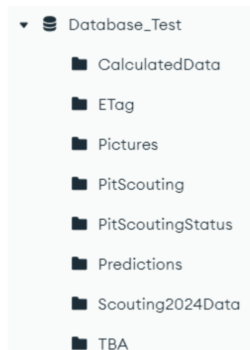
The entire scouting app is based on these API calls. The backend will send information as JSON to the front end, from where the front end can display the data. The front end also sends back data as JSON to be stored in the MongoDB database. This results in a mesh of GET, POST, PUT, and DELETE endpoints. The backend is built in Python using FastAPI.

scouting All of the scouting data Get and Post endpoints.

Method	Endpoint	Description
GET	/{year}/{event}/{team}/PitScouting	Get Pit Scouting Data
GET	/{year}/{event}/PitScoutingStatus	Get Pit Scouting Status
POST	/PitScouting/	Post Pit Scouting Data
PUT	/MatchScouting/	Update Match Scouting
POST	/MatchScouting/	Post Match Scouting
GET	/Pictures/PutURL	Get Picture Post Url
POST	/Pictures/ConfirmUpload	Confirm Picture Upload
GET	/{year}/{event}/{team}/getPictures	Get Pit Scouting Pictures
GET	/{year}/{event}/getPictures	Get Event Pictures
DELETE	/Pictures/Delete	Delete Pit Scouting Pictures
GET	/{year}/{event}/{team}/ScoutEntries	Get Scout Team Entries
GET	/{year}/{event}/ScoutEntries	Get Scout Event Entries
POST	/FollowUp	Post Team Follow Up
GET	/{year}/{event}/{team}/FollowUp	Get Team Follow Up
DELETE	/MatchScouting/Delete	Delete Match Scouting

users Manage users.

Method	Endpoint	Description
GET	/User/Groups	Get User Groups
GET	/User/Groups/Detailed	Get User Groups Detailed
GET	/User/GroupJoinRequests	Get User Join Requests



Database

MongoDB is used to store the data. This is extremely easy because it accepts documents of JSON type. This is important because all of the data used in the algorithm is in JSON format. On top of MongoDB, to reduce cloud costs, images are stored in an Azure blob storage, which can inexpensively store large files.

Login system

Keycloak was used to manage the login system. This is because it has easy user and group management utilities, which allow the backend to easily analyze data. Keycloak is attached to a Postgres database to ensure that all of the auth information persists across deployment cycles.