2024 Technical Documentation

Team 3082
“Chicken Bot Pie”
Minnetonka High School
Minnetonka, MN

-TABLE OF CONTENTS-

-ROBOT OVERVIEW-

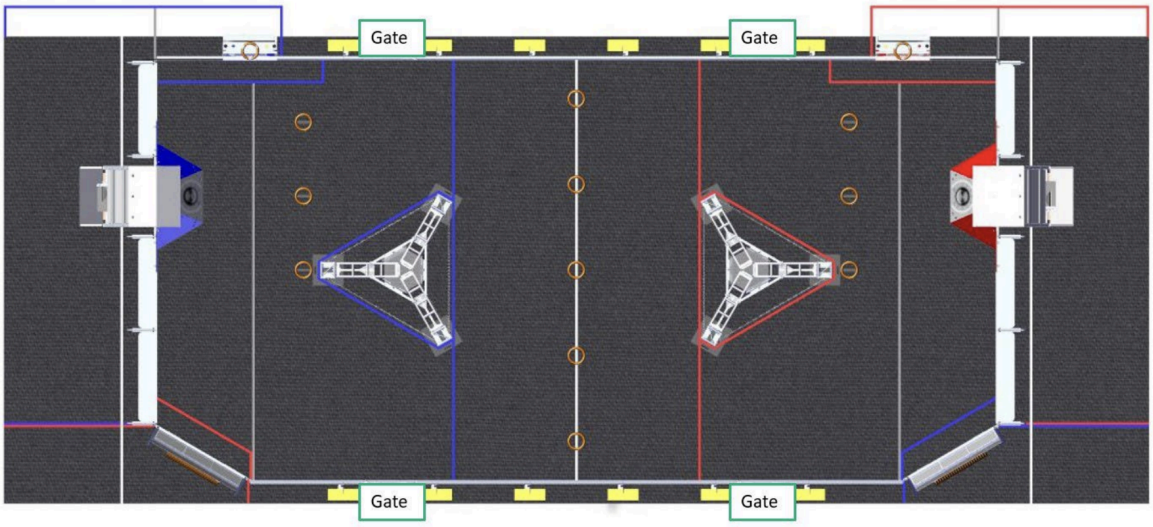
We are proud to present our 2024 robot: Front-Runner!

-STRATEGY-

Crescendo
Our Strategy
Priority List and Design Goals

STRATEGY Crescendo

After reviewing the 2024 game, *Crescendo*, we identified several key tasks that robots would have to accomplish. These are:
Driving the ROBOT to navigate the field
Collecting NOTES from the source or the ground
Scoring NOTES into the SPEAKER, AMP, or TRAP
Climbing onto the STAGE's chain



The 2024 Game Field

There are 107 total NOTES on the field. 3 are staged in each ALLIANCE's WING area, and 5 are staged on the CENTER LINE. One NOTE may also be preloaded in each ROBOT. The remaining NOTES are staged in each ALLIANCE's SOURCE area (45 each).

In addition to the 2 ranking points (RPs) given for a win, there are two other available RPs. One (the MELODY RP) is granted for scoring 18 SPEAKER and AMP NOTES, and one (the HARMONY RP) is granted for earning 10 STAGE points and at least 2 ONSTAGE ROBOTS. This year also brings the COOPERTITION point, one point granted to each team by activating the COOPERTITION bonus in the AMP. This bonus brings your required 18 SPEAKER and AMP notes down to 15.

STRATEGY: Our Strategy

We set our goal this season to compete in the playoffs as a high seed at both of the events that we attend (Northern Lights Regional and Seven Rivers Regional), as well as qualify for the FRC Houston Championship.

It seemed feasible that one robot would be able to control its destiny with an efficient shooter, bringing the potential of scoring the MELODY RP down to a single robot. Speaking narrowly in regards to a week one event (like Northern Lights), without a dedicated TRAP scoring mechanism from either your robot or an alliance mate, it would be comparatively difficult to score the ENSEMBLE RP. The absence of a low/floor goal is an interesting strategy move, and one that we believe will push the design process of lower-resource teams.

The ideal robot would be able to secure both ranking points consistently by shooting NOTES into the SPEAKER from anywhere on the field - including while moving, maintaining an effective climbing mechanism reliable enough to hang on one of the chains, and score into the TRAP during ENDGAME. With our resources, timeframe, and capabilities in mind, we realized that we would either have to prioritize a reliable shooter or the TRAP scoring mechanism.

We expect this to be a relatively common alliance at MN regional events:
Captain - Swerve robot, can score TRAP, AMP, and SPEAKER efficiently
First Pick - Swerve robot, can score SPEAKER and AMP efficiently
Second Pick - Low cycler, AMP or SPEAKER-only robot

Keeping in mind the updated eligibility requirements for advancing to the Houston Championship - as well as our constricted timeframe and resources, we decided to build a robot that will be a very attractive first pick. We led our design process around creating a very effective and reliable shooter, crafting a balanced and fast climber, and foregoing a TRAP scoring mechanism. We believe that this delicate balance of speed, agility, and cycling power will make us a very attractive pick at MN regional events.

STRATEGY Priority List and Design Goals

As previously mentioned, our goal for this season was to build a highly competitive robot that could act as an alliance captain or first pick. We had to be efficient with our time to meet our Week 1 deadline, so we had to carefully choose what we did and didn't want our robot to do. This process started with a game action analysis, in which we determined the benefits of every action a robot could complete on-field, as well as how difficult they would be to accomplish. This process would later help us determine which features we did and didn't want on our robot.

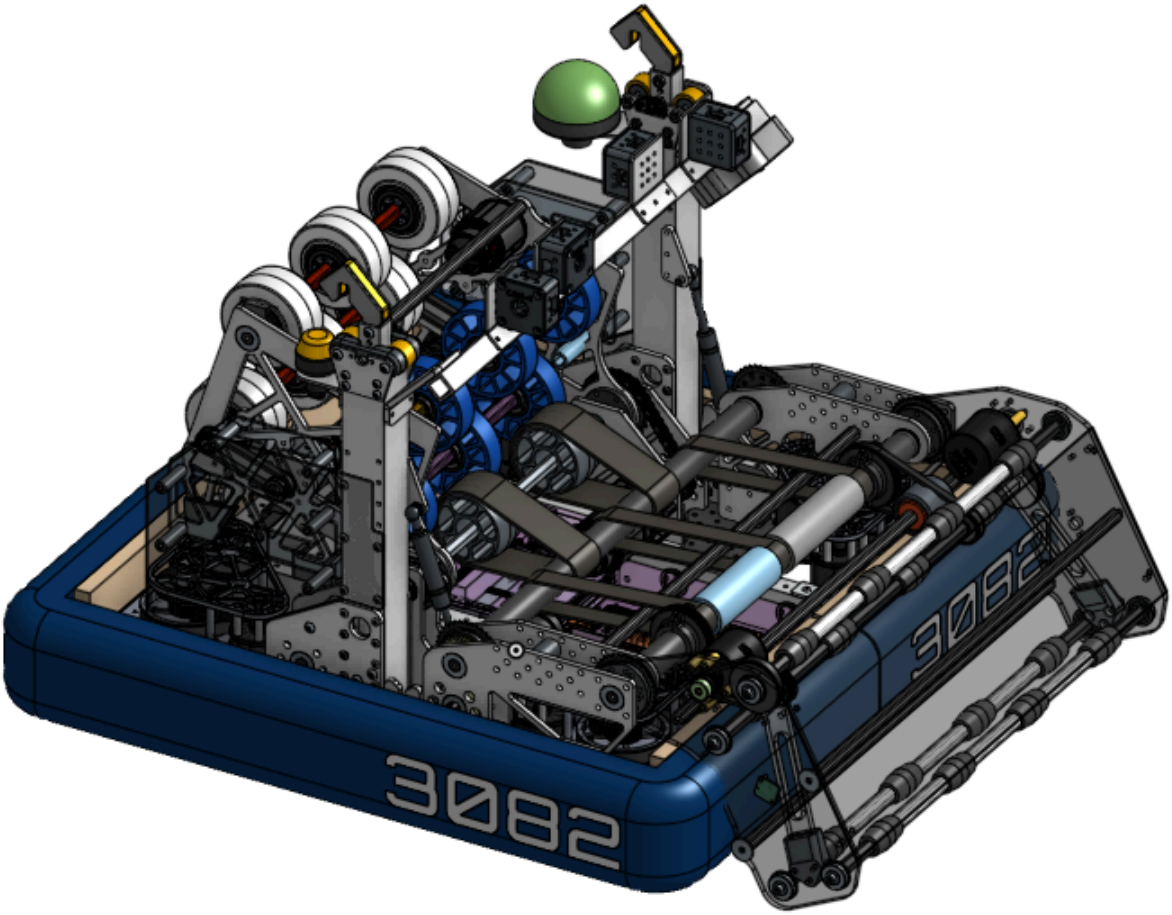
Game Action	Difficulty (Low, Med, High)	Benefit	Notes
Ground intake	Ground intake	required to score (or source intake), can pre-place game piece before intake	
Source intake	Low-Med	Quicker note loading	
Solo climb	Low-Med,	3 points	
Double climb	Med-High	3 per robot and 2 bonus so 8 total	
Speaker score	Med	2 points normal, 5 points amplified, 5 points auto	
Amp score	Med-High, awkward shooting/scoring angle	1 point normal, 2 points auto, 1/2 way to amplification	Lose 2 points when scoring here twice, but the amplification bonus probably makes up for it?
Driving under stage	Depends, just need short robot	Faster cycles, more direct line around field, less defendable	Should not be prioritized over others, could affect ability to have a good shooter
Trap score	High	5 points	Gets you half way to RP, may be difficult
High note score	N/A	1 point	helps with RP, literally just HP skill issue (recruit ultimate frizbee players)
1 piece auto	Low-Med	5 points for scoring, +2 for leaving, 7 total	
2 Piece auto	Med-High	10 points for scoring, +2 for leaving, 12 points total	

After we finished our game action analysis, we created a "want/don't want" list. We determined that we wanted our robot to be capable of scoring in the amp and speaker, but it wouldn't be worthwhile to score in the trap. We wanted to be able to intake notes from the ground and directly from the source, and we wanted to be able to climb on the stage.

In addition to these necessary features, we set some design goals. We needed the robot to be very light, as we had just purchased a set of L3 swerve modules. We also wanted it to be very compact, as it needed to fit under the stage and have a low center of gravity (for stability). Finally, we wanted the design to be simple so we could build and repair it quickly.

Beyond those criteria, we left quite a bit of freedom for the design team. The subsequent design decisions would be based on prototyping and testing, as well as an ease of integration into the final robot.

-MECHANICAL-



Prototyping and Iteration

Drivetrain

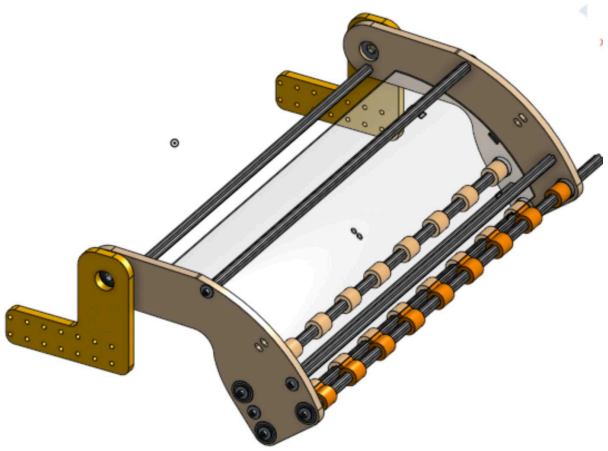
Intake & Indexer

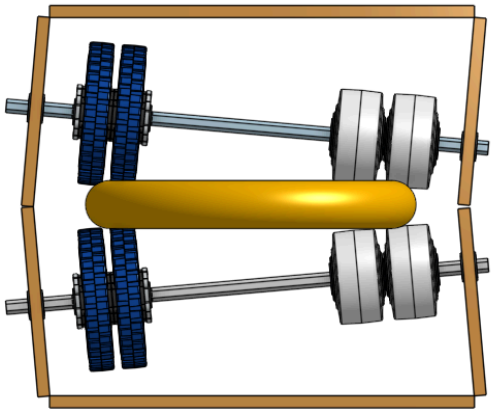
Shooter

Climber

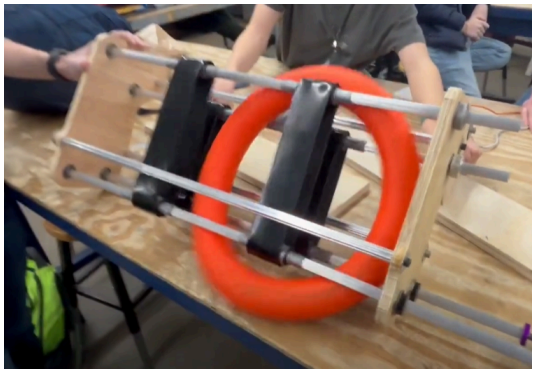
MECHANICAL: Prototyping & Iteration

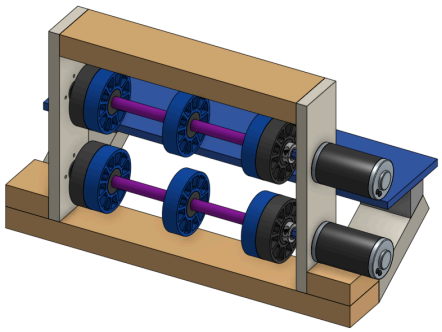
We spent the majority of the first week after kickoff working on prototypes. We focused most of our efforts on our shooter and intake. We tested multiple concepts for both mechanisms.
For our intake, we built a roller intake, as well as a belt intake. For our shooter, we built both a horizontal and vertical roller shooter. We ultimately decided to use the belt intake and horizontal roller shooter.



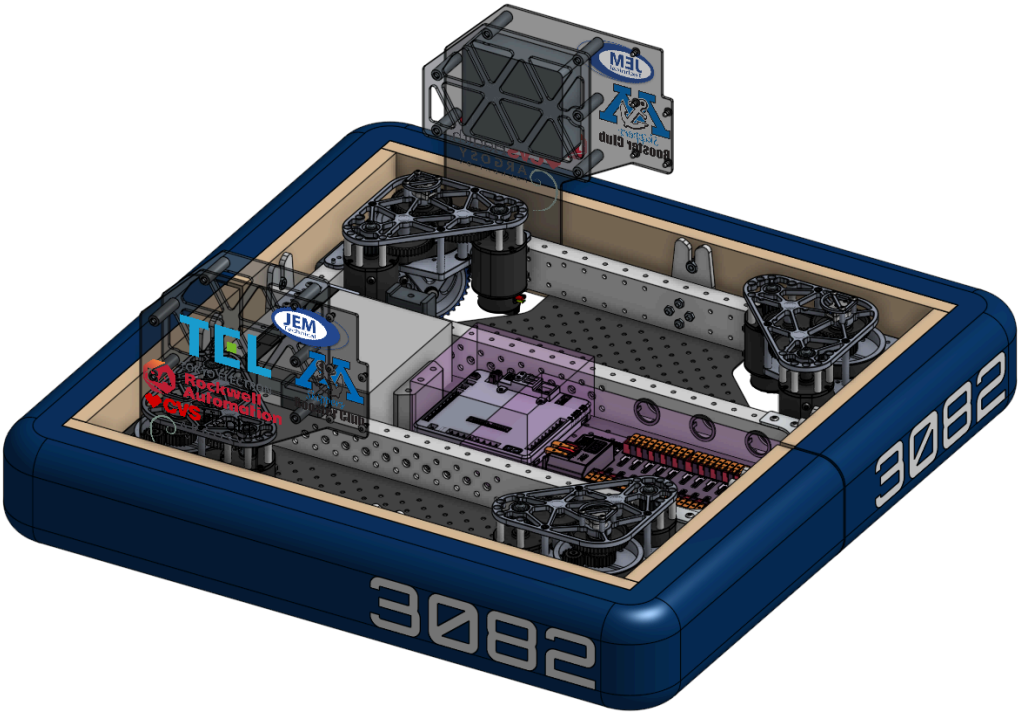


We also tested several variations of the mechanisms we chose, including a spin-inducing shooter (as pictured above). However, this concept was not promising enough to be tested further. In total we created about 9 distinct prototypes which were far more than we had done the previous year.



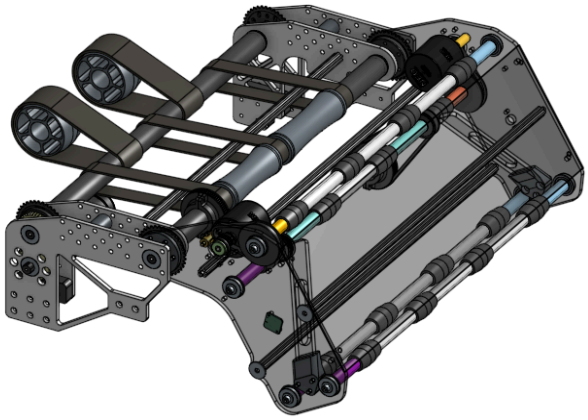


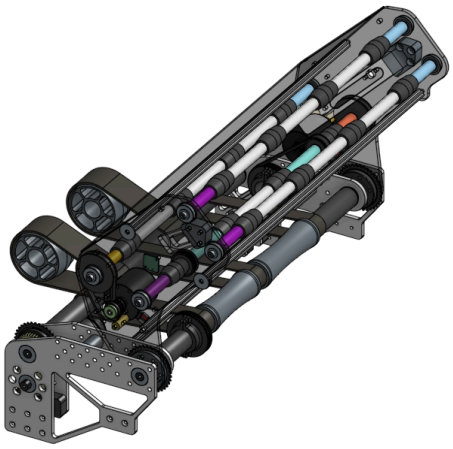
MECHANICAL: Drivetrain



Our drivetrain is a fairly standard swerve drivetrain. We used SDS MK4i swerve modules, equipped with L3 gear ratios and Falcon 500 motors. Our frame perimeter measures 26.5”x26.5” total, and 33”x33” with bumpers. This is our fourth year utilizing a Swerve drivetrain, and due to our experience, we have incorporated many lessons from our prior years. We switched to MK4i modules primarily to better protect our motors, as we had issues with our arm hitting our motors last year. We also have some benefits with these newer modules like a lower center of gravity, and a faster top speed.

MECHANICAL: Intake & Indexer



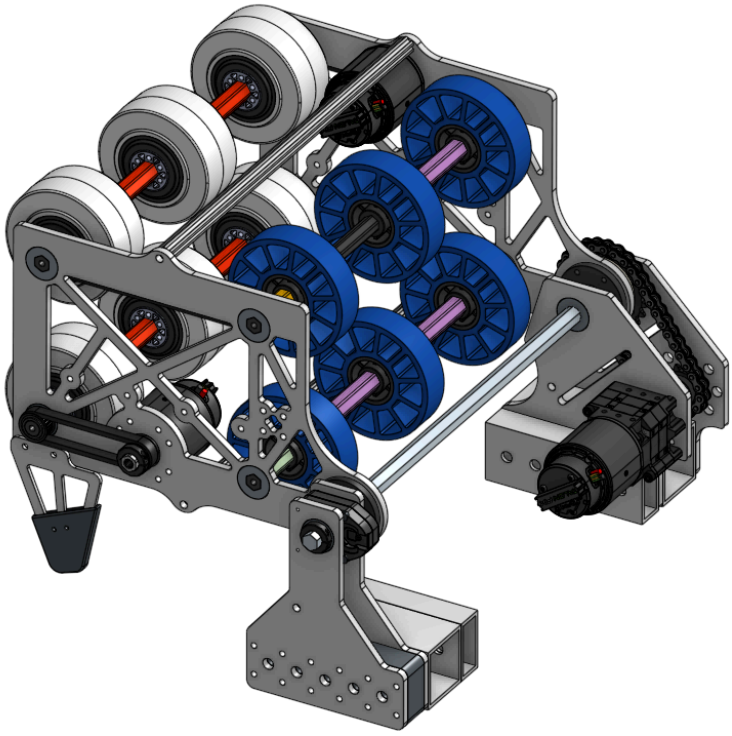


Our intake is constructed out of ¼” polycarbonate, with a ¼” aluminum mounting structure. We chose to use polycarbonate for the arms of the intake because of their resilience and light weight, this material has been used in FRC for many years and is good at getting hit and bending rather than snapping, and this same material is also used in the manufacturing of bulletproof glass.

The intake manipulates notes using 2” polyurethane belts on crowned pulleys. The top and bottom sets of belts are driven independently by two NEO motors. The whole intake is actuated using a Falcon 500 motor, paired to a 36:1 gearbox. We use the hardstop to zero the intake arms at the start of each match.

We use a laser distance sensor to detect if we have intaken the note piece, when we are ready to score a note, the intake moves into its stored (fully retracted) position, and the note is compressed between the intake belts and indexer belts where then it can be fed into the shooter.

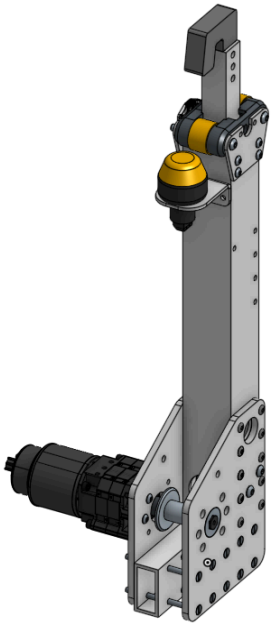
MECHANICAL: Shooter

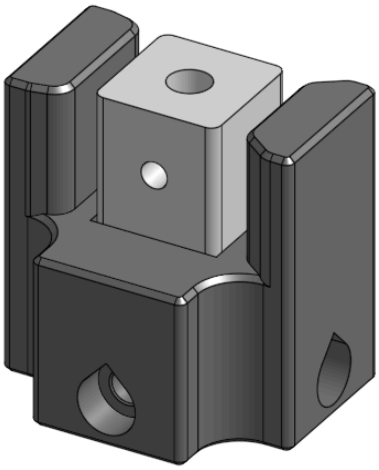


The shooter is the most important of our robot so it was prototyped 4 different times so that we could design an accurate and consistent mechanism. It is made out of ¼ inch aluminum which was used to have a very rigid and consistent feed for the game pieces; some weight reduction patterns have also been used to decrease the overall weight of the entire mechanism.

We use a 50:1 reduction to drive the angle of the shooter in a dead axle setup to reduce backlash, there are 2 additional motors used to drive the top and bottom wheel rows, the philosophy behind this was that we could vector the top and bottom rows to achieve maximum control over how we manipulated the piece during a shot, in this way we can tilt the piece in a vertical position by running the lower row faster to make the piece vertical upon its exit. Each motor can spin up to 6,000 RPM which gives us plenty of power to shoot the piece across the field.

MECHANICAL: Climber





The climber was one of the most difficult mechanisms to create which is why many teams choose to buy a premade kit instead of designing one themselves. We decided it would be best to design our own from scratch to maximize the learning outcomes and versatility of design, therefore our climbers are able to attach various brackets and attachments to the side which is possible because of our unique plunger design. Many kit climbers cannot do this, so we were able to utilize this principle to attach the RSL (robot signal light) and a camera bar which holds 4 cameras.

We are using a 3D printed plunger made of PP (Polypropylene) for a low coefficient of friction, which ensures smooth movement on the inside of the box tube. The clearance holes for the socket screws use a teardrop hole which is a lesser known 3d printing trick that lets the printer create a more accurate/circular hole, and it works by sacrificing the upper portion of the hole so that the plastic cannot droop down to create an oval shape which is what happens without the teardrop. We coupled the plastic part with a custom CNC'd 7075 aluminum core which easily pulls the compression load of our robot. While a plastic part could technically do this, the metal core proved to be more reliable when assembling.

We are using a 25:1 reduction on our two Falcon 500 motors, along with a dyneema cord rope rated for 2000 pounds that enables us to get a climb in about 1.5 seconds.

-Software-

Methodology

Trajectories

Autonomous

Structure Control

Vision & Driver Assist

Driver Station

SOFTWARE: Methodology

With our experience programming flywheels & swerve drivetrains, we kicked off our year with a list of ambitions that, if executed successfully, would be able to remove a lot of the strain off the driver. Our Swerve drive utilizes a Pigeon 2.0 as our gyroscope, enabling our driver to utilize Field-Oriented Control leading to a much simpler way of controlling the robot during the match. We use CANCoders to get the wheel's direction on the robot's startup and sync that to our steer motor, allowing for overall lower CAN utilization throughout the robot.

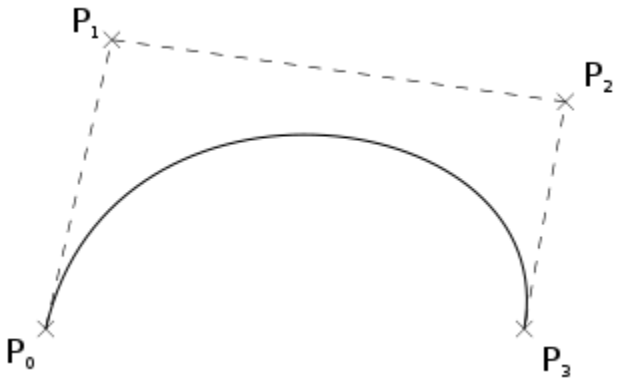
This year is our second year of vision-tracking with AprilTags. This allows us to get a really accurate position estimator running on a coprocessor, so our power isn't taken away from the core robot components. This allows us to run more advanced field localization strategies, helping execute complex autonomous routines & enabling our driver to take advantage of some autonomous driver aids.

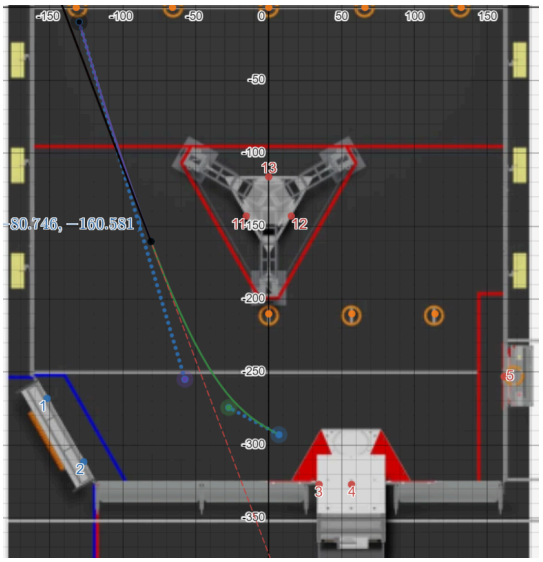
Our Shooter code is controlled by a state machine, with three major states that the driver can choose between: firing, revving & disabled. For simplicity, the shooter is always disabled when not actively firing or revving. Our angle for the shooter's adjustable pivot is also determined in the state machine, and it depends on the current position that we are relative to the speaker on our alliance's side. The shooter's pivot, intake + handoff & the flywheels are each in their own separate file, and it makes for a seamless integration into our existing driver inputs. During our tests, we shot notes at varied RPMs, as we initially thought that our exit velocity would vary depending on our distance away from the target. However as we continued to iterate, we found out that the structure of the shooter was more consistent shooting at 3,500 RPM, so we forgoed the varying RPMs and decided to stick with a consistent exit velocity.

SOFTWARE: Trajectories

This year we decided to completely overhaul our autonomous trajectory system, we have a couple of different pathing solutions that we implemented and tested.

Bezier Curves





The first trajectory that we tested and implemented was the simple cubic bezier, this is the same type of spline that the program PathPlanner uses, however we created our own implementation of this into our code. Because we don't have a UI developed for this, we decided to create our trajectories on a Desmos calculator. First we define the curve using the 4 points P0-P3 and this equation:

$$B_x(t) = (1-t)^3 \cdot P_0.x + 3(1-t)^2 \cdot t \cdot P_1.x + 3(1-t) \cdot t^2 \cdot P_2.x + t^3 \cdot P_3.x$$

$$B_y(t) = (1-t)^3 \cdot P_0.y + 3(1-t)^2 \cdot t \cdot P_1.y + 3(1-t) \cdot t^2 \cdot P_2.y + t^3 \cdot P_3.y$$

Then we create a lookup table of the nth size depending on our desired resolution. Using another equation we can create a 2D Vector of the desired translation direction with this equation:

$$c_x(t) = -3P_0.x(1-t)^2 + 3P_1.x\left(3t^2 - 4t + 1\right) + 3P_2.x\left(2t - 3t^2\right) + 3P_3.xt^2$$

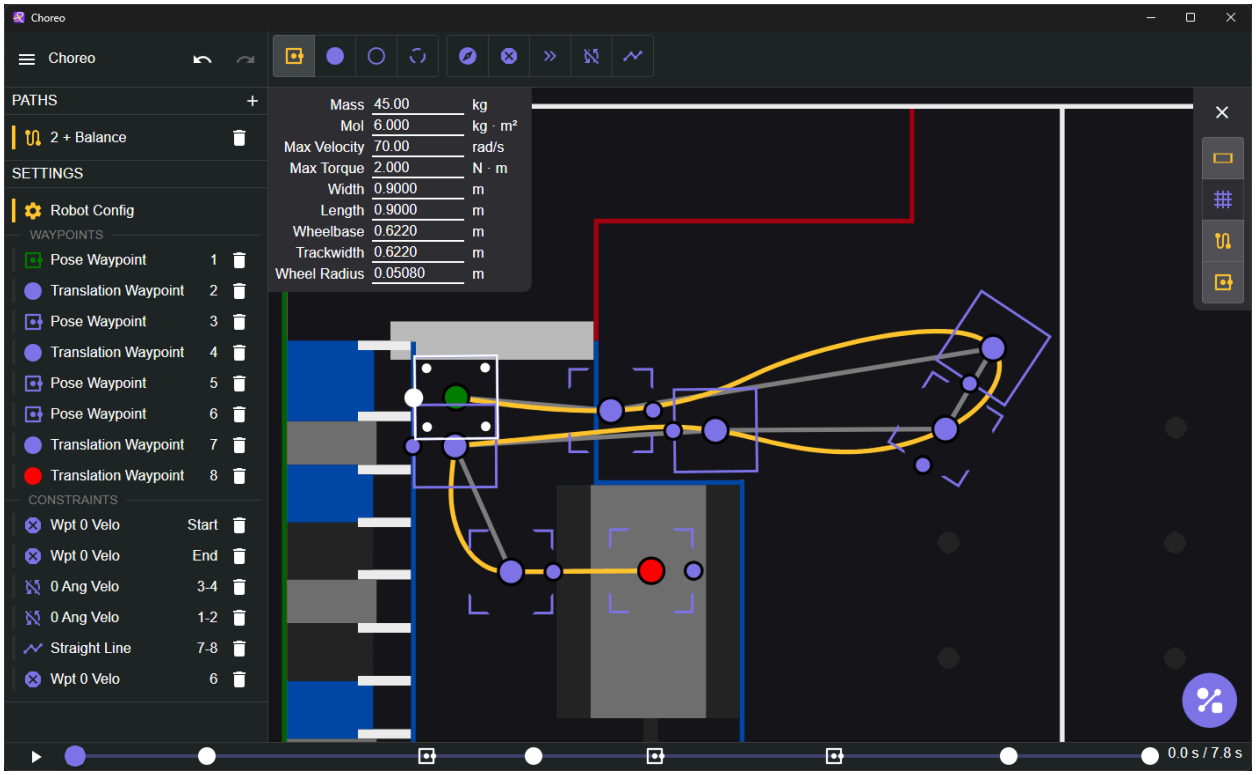
$$c_y(t) = -3P_0.y(1-t)^2 + 3P_1.y\left(3t^2 - 4t + 1\right) + 3P_2.y\left(2t - 3t^2\right) + 3P_3.yt^2$$

```
public Vector2 getPoint(double t) {
    double x = (Math.pow(1 - t,b:3) * a.x) +
    (3 * Math.pow(1 - t,b:2) * t * b.x) +
    (3 * (1 - t) * Math.pow(t,b:2) * c.x) +
    (Math.pow(t,b:3) * d.x);

    double y = (Math.pow(1 - t,b:3) * a.y) +
    (3 * Math.pow(1 - t,b:2) * t * b.y) +
    (3 * (1 - t) * Math.pow(t,b:2) * c.y) +
    (Math.pow(t,b:3) * d.y);

    Vector2 r = new Vector2(x, y);
    return r;
}
```

```
private void getEvenlySpacedPoints() {
    int n = Tuning.CURVE_RESOLUTION;
    int i;
    Vector2 pos;
    double t;
    for (i = 0; i < n; i++) {
        pos = getPoint((double) i / (double) n);
        t = (double) i / (double) n;
        this.curvePoints[i] = new CurvePoint(pos, t);
    }
}
```

The equation of c(t) is used to determine a vector of the desired translation. After we get this vector we use a PID loop using the final overall length of the curve and input each frame our current position on the curve, updating the PID loop, and multiplying the normalized vector of our translation by the output of the PID loop. Overall this solution was very successful, and produced great results, still we decided to further investigate other solutions to see if we could get even better results.

Choreo



A new piece of software called Choreo came out this season, Choreo generates optimal trajectories given the robots physical constraints such as weight, motors, and the robot's drive gear reduction to produce an optimal path that the robot can almost always follow, this approach takes a lot less tuning compared to more common spline paths. We created a script to interpret the json data returned by the program and convert the data into something usable in our custom swerve code. In the end we decided to continue with Choreo for our competition robot and the decision has made our lives a lot easier, as it has allowed us to more quickly tune our autonomous routines.

SOFTWARE: Structure Control

Shooter

Our shooter contains two main parts: a pivot and the flywheels. To control the pivot we use a Falcon 500 and TalonFX with MotionMagic to create fast, reliable, and efficient positional control. We zero the pivot's position at the start of the match using a CTRE CANCoder to get the shooter's absolute position. For the Flywheels we use 2 Falcon 500s one for the top and one for the bottom rollers, and we use TalonFX Velocity Control Mode to get more accurate motor velocity control than a simple DutyCycle Output. We then measure the velocity of each motor and if they are in a certain deadband we allow the shooter to shoot.

Intake

Our Intake includes four main parts, a motorized pivot, top and bottom belt rollers, a belted conveyor, and the new LaserCAN time of flight sensor. The pivot, similar to the shooter, uses a Falcon 500 using MotionMagic for positional control, but instead of a CANCoder for zeroing the internal rotor encoder we have a back hard-stop for a consistent starting position. This allows us to simplify our wiring, and reduce complexity both in design and the code. The conveyor and rollers are powered by two NEO motors, as this application with the belts creates a fairly high friction system and we required a fairly powerful motor to power it all. We use simple DutyCycle percent output on the SparkMAXes for control, but we also use the LaserCAN sensor as a beambreak to sense when we have a game piece to both stop the intake motors, and automatically flip up the intake.

SOFTWARE: Structure Control

Climbers

The Climbers use 2 main parts each, a Falcon 500 powering a winch, and a Thrifty Hall Effect Sensor which is used for zeroing the climbers. We use the TalonFX DutyCycle as well as the internal rotor position to prevent any damage, for when we pull down it will check the hall sensor each frame and if it is tripped then we stop all pulling input, we also measured the max extension in motor ticks and have another check to stop overextension.

Swerve Drive

Our swerve code is broken up into four sections: modules, kinematics, odometry and closed-loop PID based feedback control. Much like other teams, we define a class that represents one “module” (steering motor, external absolute encoder & a drive motor) with functions that propel the robot forward or steer in a certain direction. Our top-level class, called “SwerveManager” handles our kinematics & various other equations to get our wheels in the right direction depending on driver stick output. We opted to use our custom Swerve codebase as it allows extra flexibility and better fine-tuning of the robot’s drivetrain output. Our closed-loop feedback control is handled through SwervePID, a class that represents our custom implementation of a PID Controller for more advanced autonomous routines, but also for smoothing output between the driver & the robot.

An important part of swerve control is knowing the current state of the robot on the field, called odometry. This is typically represented as an (x, y) coordinate on the field, and used for knowing the position of the robot for, say, an auto routine. We have a custom odometry implementation, and we went with this route because it interfaces really nicely with our custom vision processing software, and allows us to implement a rudimentary Kalman Filter for smoothing the (x, y) output of odometry and the output that the vision system gives us.

SOFTWARE: Vision & Driver Assists

Custom Vision Software (ChickenVision)

During the offseason we started development on a custom vision solution that we named ChickenVision. Written purely in Python, this implementation is able to detect retroreflective tape as well as AprilTags, and gather either 2D or 3D data from said AprilTags. ChickenVision is primarily built for expansion, as we designed this system around the potential to have four global shutter cameras mounted around the robot.

PhotonVision

Currently due to time constraints we are utilizing PhotonVision to run all of the vision pipelines on our cameras, it's a very simple and easy way to integrate vision into a robot system. We are using only the Aruco 3D AprilTag pipeline and the simple (x, y, z) coordinate data that is returned into NetworkTables for our own custom pose estimation code.

Vision Coprocessor - high-power Mini PC

For our vision co processor we wanted to simplify our system compared to what many other teams have been running, instead of running for example 4 Limelights, Raspberry Pis, or Orange Pis we decided to run one centralized beelink mini pc running a Ryzen 5 5600U processor and 16GB of RAM, with this configuration we are able to run all 4 cameras at 30 - 40 FPS concurrent at 720P each, on the 3D AprilTag using the Aruco pipeline.

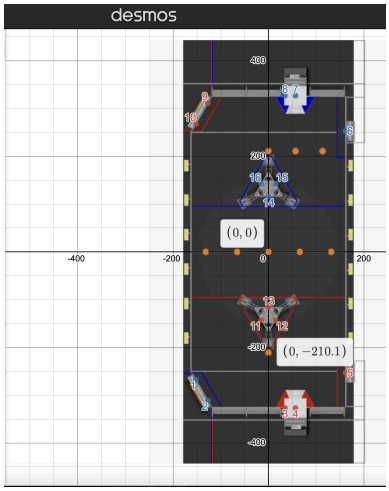
Cameras - 4 Cameras

As previously stated we are running 4 cameras on our robot for vision this year, they are all identical 70 degree FOV 720P 120FPS Global Shutter Cameras from ArduCam. With this configuration, running them in the 4 cardinal directions we can essentially see a close to 360 degree view around the robot. There is some deadband close up but past that point we have a full view. This makes it so that no matter where we are on the field we can always have a good view of at least one AprilTag on the field.

SOFTWARE: Vision & Driver Assists

Position Estimation (Math)

For our position estimation math we started by interpreting what the numbers mean from PhotonVision, then we built a simple desmos calculator to show the math needed to calculate a field position from the data we are given. First we start by taking the x (forward distance) and y (horizontal distance) given to us from the 3D AprilTag pipeline. We then multiply that vector by cos(camPitch) to account for the pitch of the cameras. After that, we add the positions of the cameras relative to the robot's origin to that vector, and then take the robot's current rotation - given to us by the Pigeon 2.0 - and rotate the vector by the robot's rotation. The final step is adding this vector to the AprilTag's position on the field, and that (x, y) coordinate is the position of our robot on the field. To increase accuracy, we take the robot position that we estimate from each camera and we average it out to reduce error. The method we use to determine the robot's position is a bit unorthodox compared to other teams - the origin of the field is defined squarely in the middle (0, 0) of the real-life field, whereas other teams typically have their origin on the blue alliance's source wall. This requires us to flip autonomous paths & the velocity of the drivetrain's wheels to line up with our side of the field. However this makes it really easy for us to measure out the locations of the game pieces for autonomous, and it is also really easy to visualize!



(Our handy-dandy Desmos calculator containing field positions)

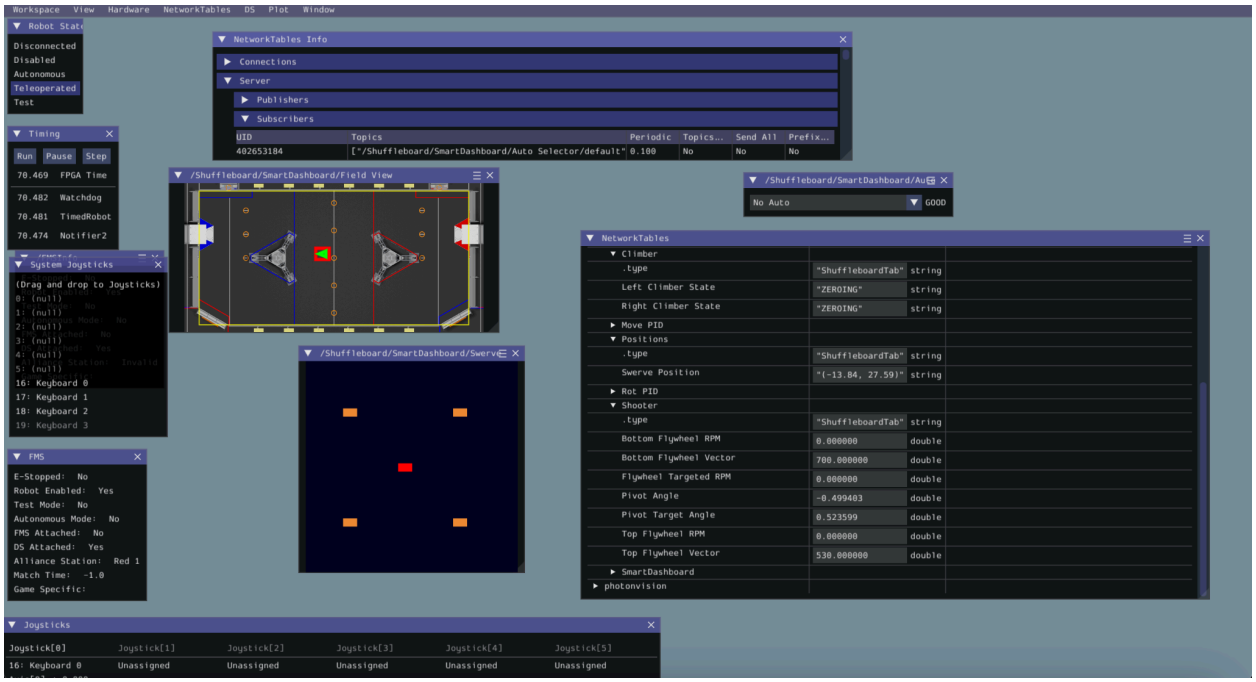
SOFTWARE: Driver Station

Advantage Scope

AdvantageScope is a robot telemetry application used to relay data from the robot back to the driver. This application is primarily used by us to get an accurate visualization of the robot's position on the field. This is very useful for both the driver station - the operator will be able to ensure the robot believes it is in the correct position for some of our autonomous routines in teleoperated, as well as creating autonomous paths without having a physical robot present.

Glass

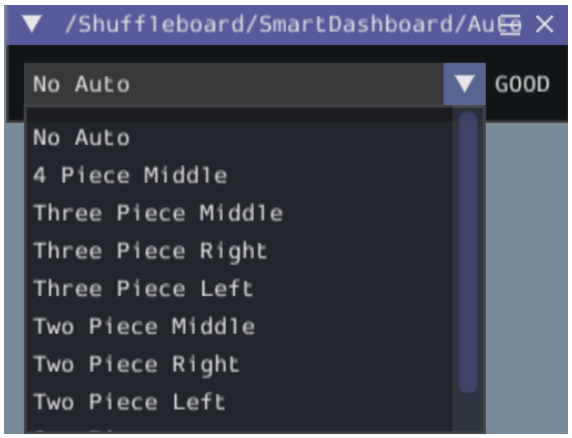
Glass is another robot telemetry application, and one of the newest ones introduced. Glass is seen more as a programmer's friend rather than for the drive team, as some of the UI is mainly meant for debugging various robot attributes through NetworkTables. Because of that, we use Glass for a majority of our telemetry & data monitoring needs instead of AdvantageScope because it gives us more flexibility over choosing values we want to see from the robot.



SOFTWARE: Driver Station

Auto Selector

One of our uses of Glass is a drop-down menu that we use to select the robot's autonomous routine. Because of our very flexible & custom autonomous framework, we are able to mix and match actions and craft them into one single routine that the drive team is able to select at the start of the match.



SOFTWARE: Driver Station

Swerve Visualizer

Because of our custom implementation of a Swerve drivetrain, we were able to utilize the potential of our library & create an easy visualization of the current state of the robot. We account for the translational velocity of the robot, as well as the rotational velocity so it's always a mirrored version of what the actual robot is running. We use this as a part of our pre-match checklist to ensure that the drivetrain wheels line up with what the joystick output is, the drivetrain mirrors the visualizer & our autonomous paths follow what we've programmed in Choreo.