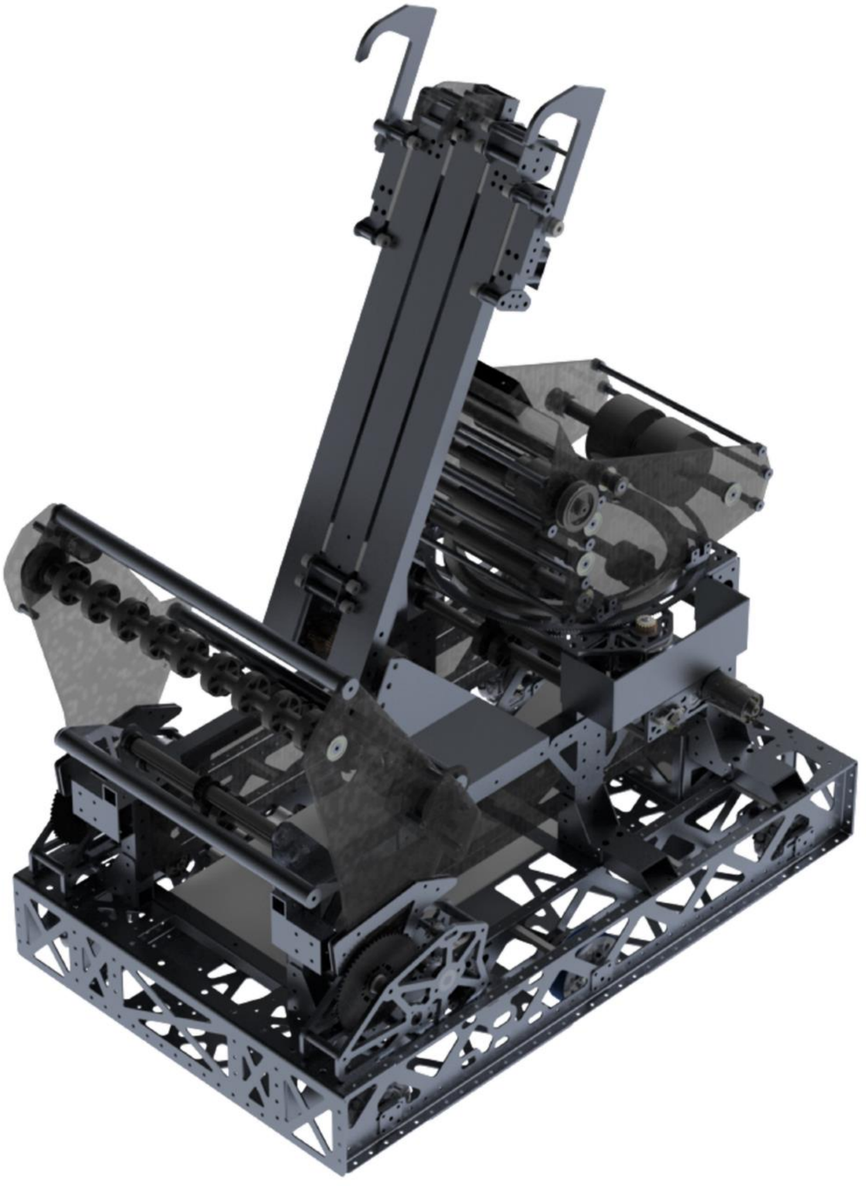


OVERTIME

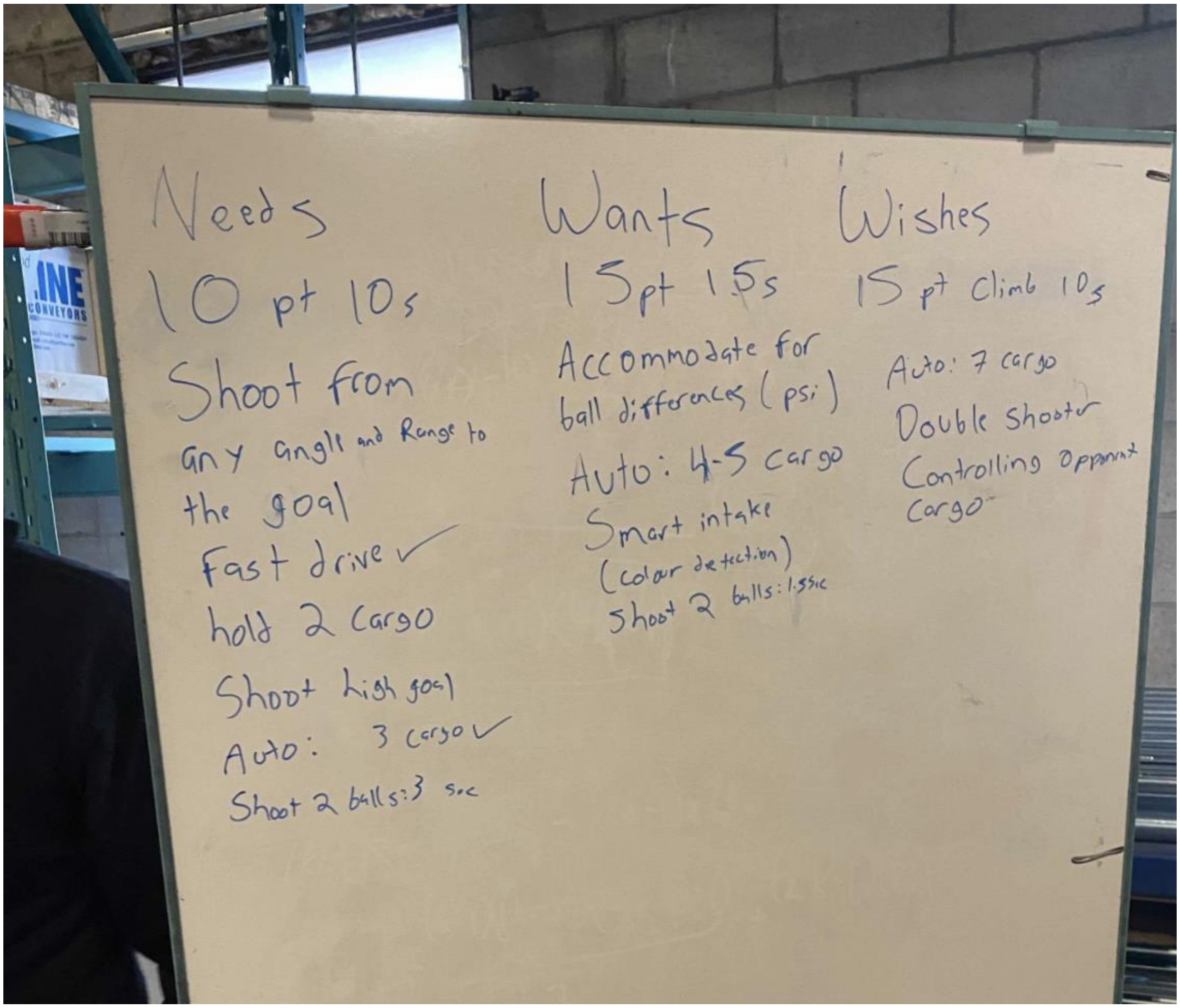
2021-2022 Technical Binder

Table of Contents

Design Process

Brainstorming

Once the game manual was released and carefully read, a list of priorities was made (this list is called our needs, wants and wishes list). This year we decided that we need to be able to climb to the traversal rung while also being able to shoot the cargo into the high goal from anywhere on the field. The team then split into small groups to brainstorm robot prototype ideas. Once all the groups were done, their ideas were tossed around with the rest of the team to decide what prototypes to try and what to avoid. By the end of kickoff, we had a list of what needed to be prototyped and who would lead the prototype process. The brainstorming stage is one of the most important stages as it essentially dictates our course of action for the rest of the season.



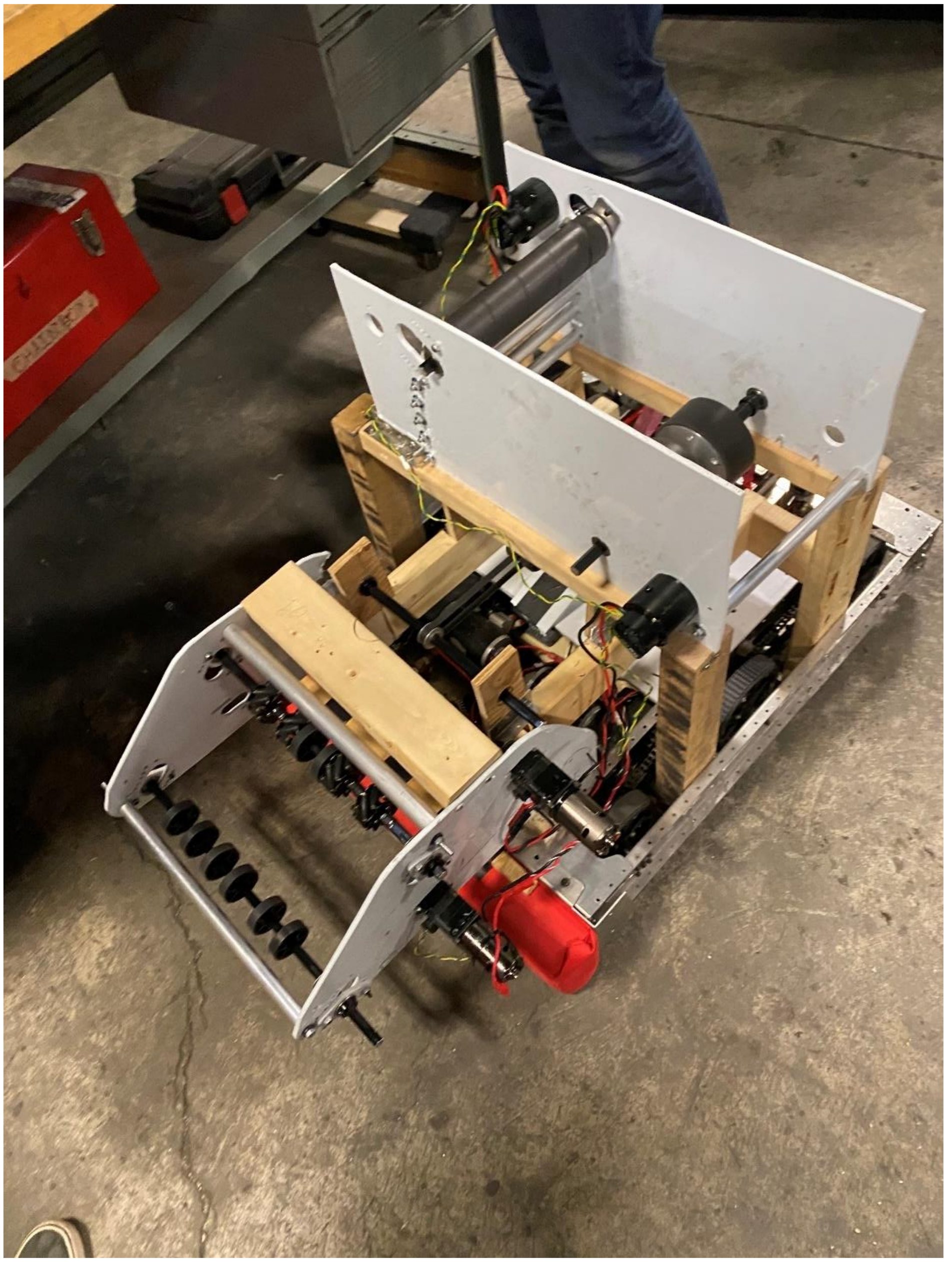
Needs
10 pt 10s
Shoot from
any angle and Range to
the goal
Fast drive ✓
hold 2 Cargo
Shoot high goal
Auto: 3 cargo ✓
Shoot 2 balls: 3 sec

Wants
15pt 15s
Accommodate for
ball differences (psi)
Auto: 4-5 cargo
Smart intake
(color detection)
Shoot 2 balls: 1.5sec

Wishes
15 pt Climb 10s
Auto: 7 cargo
Double Shooter
Controlling Opponent
Cargo

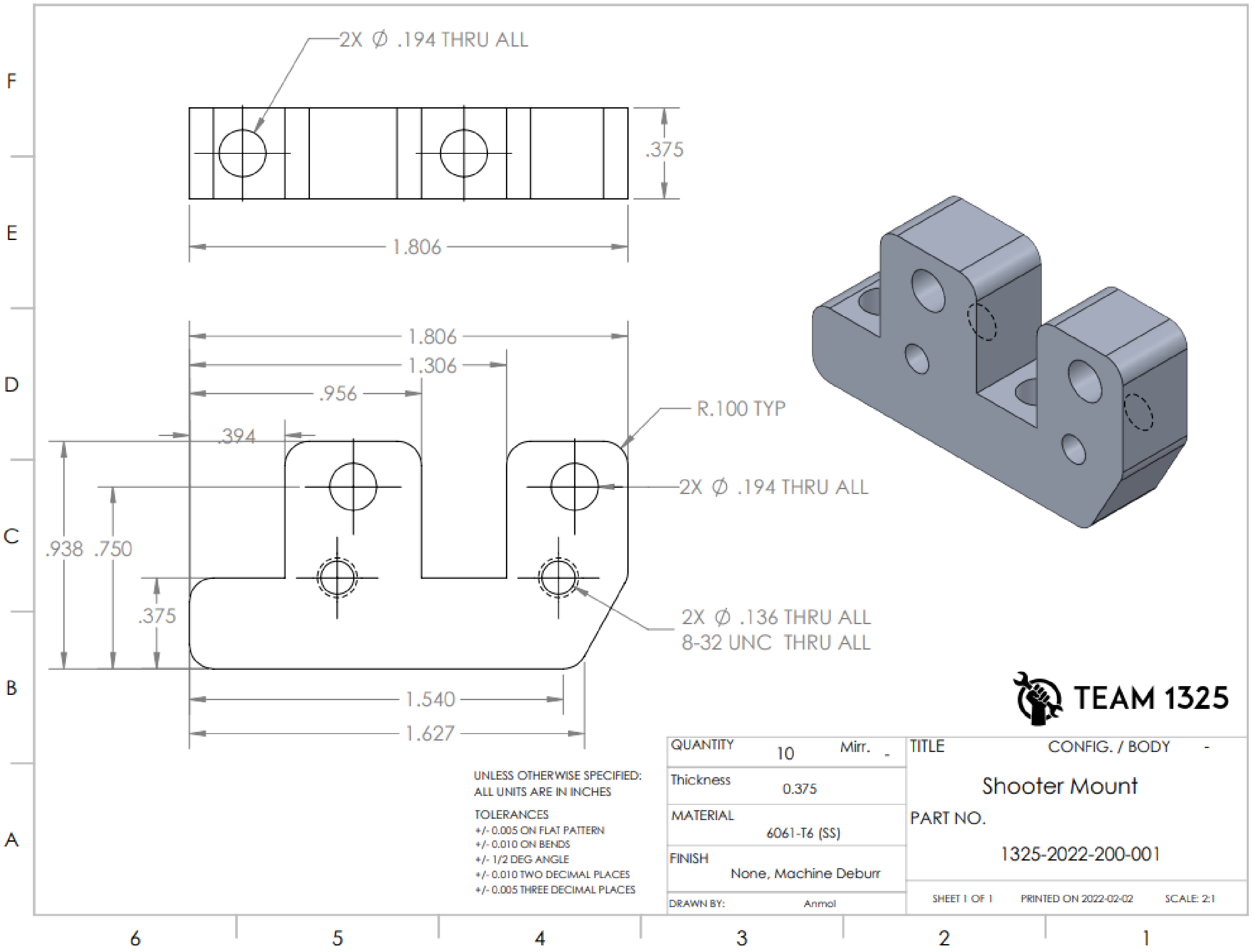
Prototyping

After the kickoff is over, the team worked to make the prototype ideas tangible. Over the course of a week, prototypes of all the subsystems were built and tested. This year we built a shooter, turret, intake, climber, and feeder prototype. One important part of the prototyping stage is collecting and recording data to ensure the CAD team can produce CAD models in accordance with the prototypes. Prototyping gives us an opportunity to plan out the major geometry of each subsystem on our robot.



CAD

During the prototyping stage, the general design of the robot was established and throughout the next 2 weeks, the design team made CADs of the prototypes to send to our manufacturing sponsors. The drivetrain was done within the first week to be sent out to one of our sponsors. The next week and a half were spent designing the rest of the robot. Using SolidWorks and GrabCAD, students and mentors can collaborate and communicate. By the end of week two our parts were sent out. 4 years ago, we introduced the use of milled aluminum extrusions which allowed us to make the design process more efficient while keeping the robot as robust as possible. This is an example of one the part drawings that would be sent to our manufacturing sponsors.



UNLESS OTHERWISE SPECIFIED:
ALL UNITS ARE IN INCHES

TOLERANCES
+/- 0.005 ON FLAT PATTERN
+/- 0.010 ON BENDS
+/- 1/2 DEG ANGLE
+/- 0.010 TWO DECIMAL PLACES
+/- 0.005 THREE DECIMAL PLACES

QUANTITY	10 Mirr. -	TITLE	CONFIG. / BODY -
Thickness	0.375		Shooter Mount
MATERIAL	6061-T6 (SS)	PART NO.	1325-2022-200-001
FINISH	None, Machine Deburr		
DRAWN BY:	Anmol	SHEET 1 OF 1 PRINTED ON 2022-02-02	SCALE: 2:1

Assembly

Once the in-house parts are made and all the sent-out parts came back, the assembly process began. CAD's and part drawings were printed out and followed for the assembly process. Aluminum extrusion is held together with gussets which are then bolted or riveted together. Occasionally, tolerances cause holes to misalign or there are factors that were not accounted for in CAD, during the assembly process we were able to address those issues and modify the parts.

Iterations

Once the parts were in and the robot was built, thorough testing and revisions were done to improve the efficiency of each subsystem. Often, through careful testing, we realize there are problems within different aspects of the robot. Solutions to these problems were tested on the practice robot before putting it on the matching competition robot.

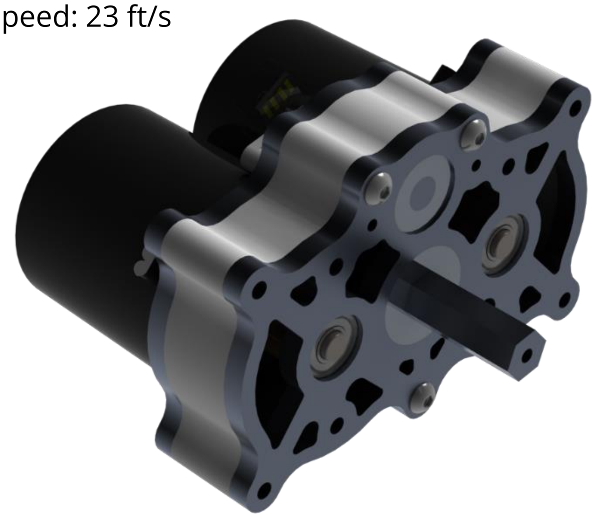
Drivetrain

This year's Drivetrain, while similar to previous years in its six-wheel tanks drives and sheet metal construction, has some key improvements and changes. An example of a design we stayed with from previous years is the S rail, which allows for easier mounting to the side rails of the drivetrain. This year's drivetrain has 1.125" of ground clearance to allow us to scale the berm in the middle of the field. This year's Drivetrain is 23" wide and 36" long. It also features adjustable chain tensioning mounting for gearboxes and wheels.

Final Design

Gearbox

- Custom Single Speed Gearbox with a 3.38:1 reduction
- 2 NEO motors
- Highest Speed: 23 ft/s

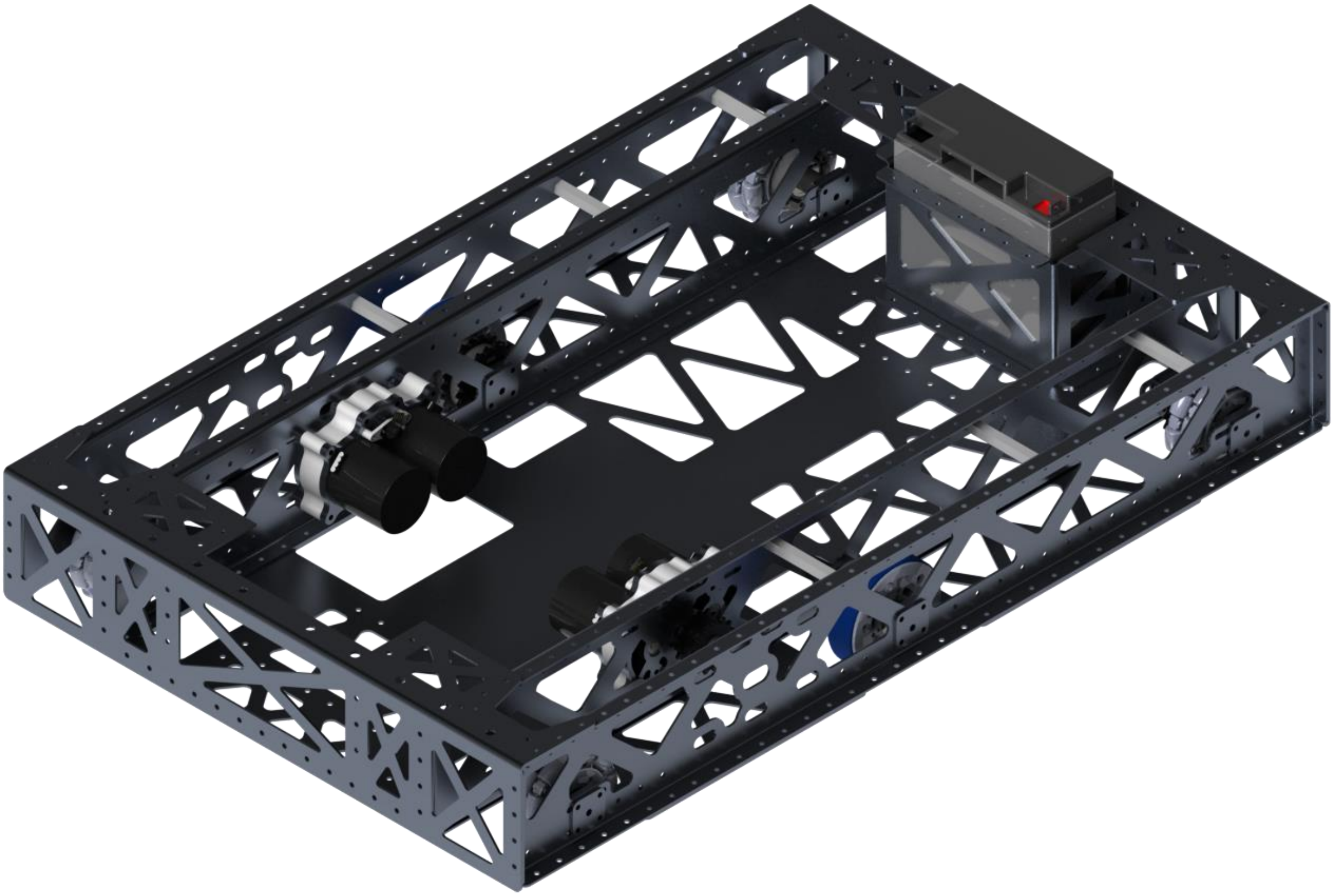


S Rail

- The inside side rail of the drivetrain creates an S shape, instead of a C shape like past years.
- Allows for direct mounting of tubing and other parts to the side of the drivetrain.
- In previous years spacers were used to mount to the side of the drivetrain. The spacers caused difficulty to mount and made service harder

Axle blocks

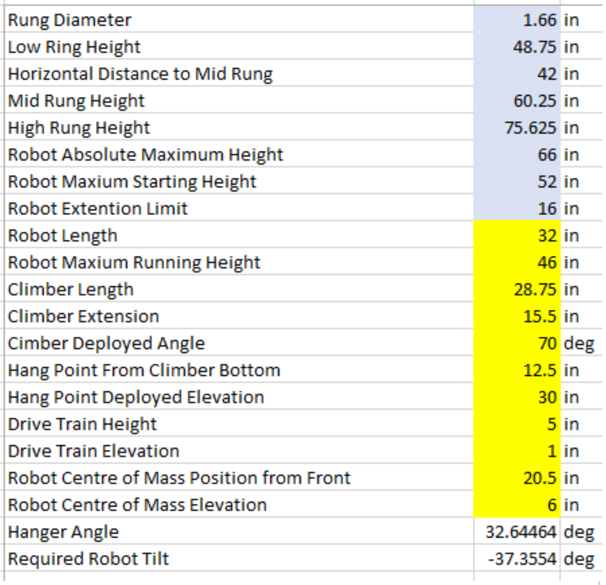
- Built in chain tensioners so that chain runs are tight and always perform at their best
- Chain fatigues over the course of the season and becomes looser, chain tensioning combats that problem



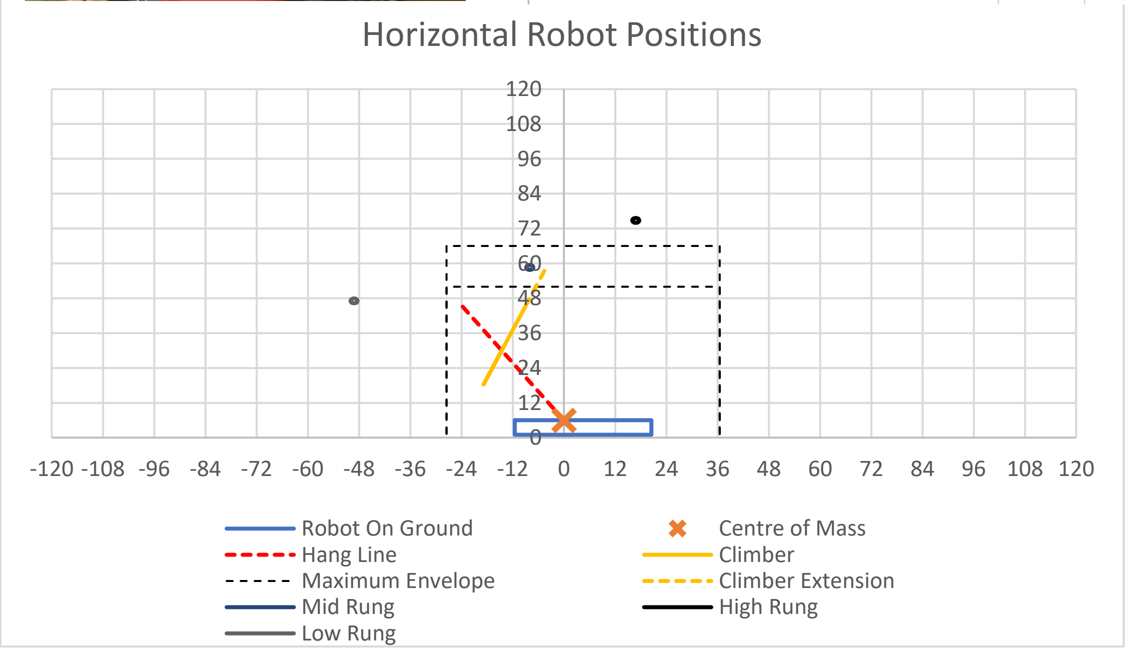
Climber

Much like our previous robots, this year’s robot is purpose-built. This entailed lifting the robot on to the Traversal Rung, hence the angled elevator system. Our climber is based on gravity and trigonometry, and we used the power of math to get our climber mechanism working. This year’s climber design centred around a two-stage cascading elevator and the custom bearing blocks designed for it. This climber has a total travel of 32”, 371.91 pound stall load and can traverse the rungs in ~11 seconds.

Prototype



Rung Diameter	1.66	in
Low Ring Height	48.75	in
Horizontal Distance to Mid Rung	42	in
Mid Rung Height	60.25	in
High Rung Height	75.625	in
Robot Absolute Maximum Height	66	in
Robot Maxium Starting Height	52	in
Robot Extention Limit	16	in
Robot Length	32	in
Robot Maxium Running Height	46	in
Climber Length	28.75	in
Climber Extension	15.5	in
Cimber Deployed Angle	70	deg
Hang Point From Climber Bottom	12.5	in
Hang Point Deployed Elevation	30	in
Drive Train Height	5	in
Drive Train Elevation	1	in
Robot Centre of Mass Position from Front	20.5	in
Robot Centre of Mass Elevation	6	in
Hanger Angle	32.64464	deg
Required Robot Tilt	-37.3554	deg



Programming

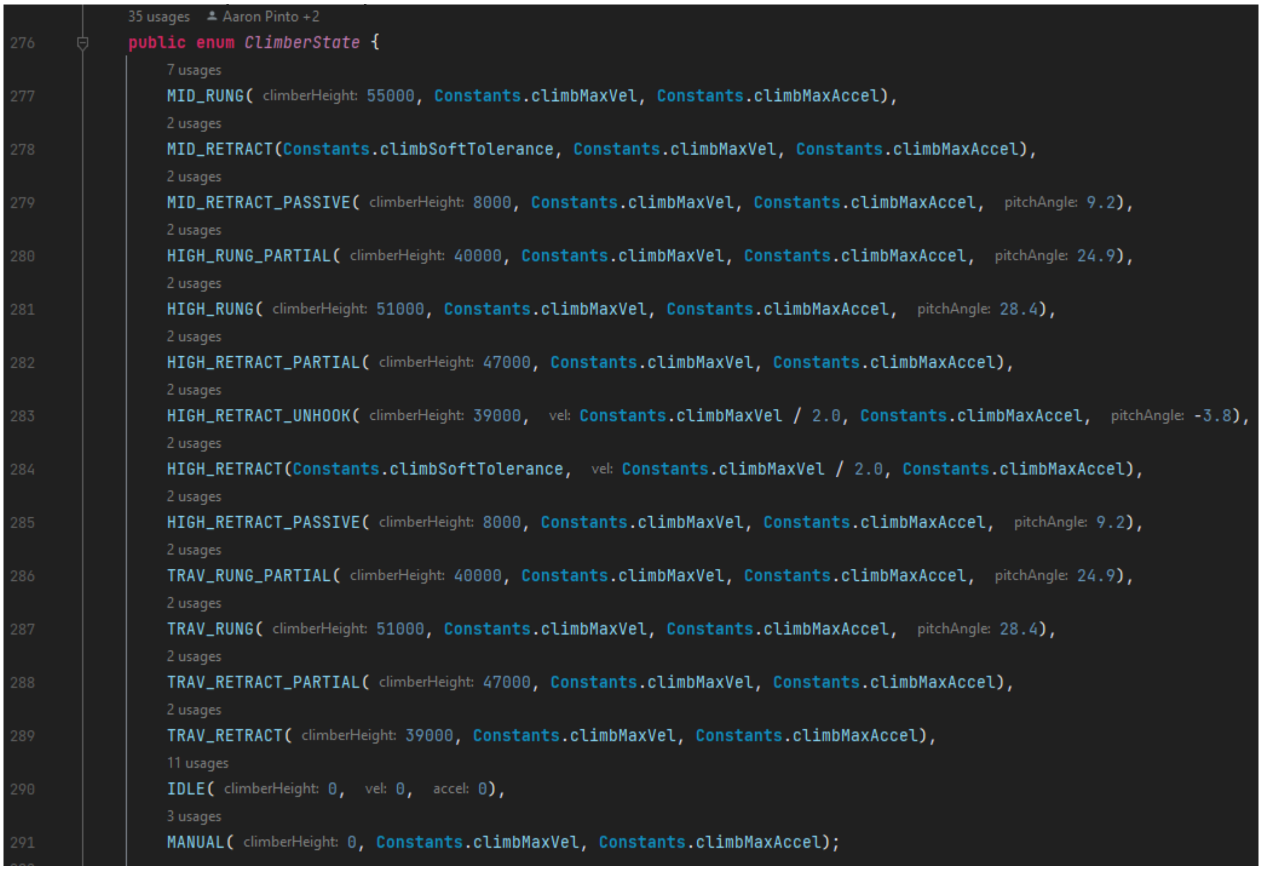
To make the climber programming as simple as possible, we created a state machine, with each state representing a different motion in the entire climb sequence. For example, our first state is MID_RUNG, which extends the climber carriage to the mid rung height, followed by MID_RETRACT which retracts the climber carriage all the way down to very slightly above the bottom soft stop position. We added these soft stops so that we were not continuously powering the carriage into its hard stops, and potentially damaging the mechanism.

To prevent accidental climber extensions during teleop, we made climb mode 2 separate and simultaneous button presses (the left trigger and Y button), which adds another level of safety to our operator controls. When we enter climb mode, the climber goes into the aforementioned MID_RUNG state, and we automatically turn the turret to -90 degrees to get the shooter cables out of the way of the climber chain run, raise the intake to its UP position, and set all the other subsystems except for drive to IDLE.

The entire climbing sequence is fully automated, and requires only 1 button press from the operator, depending on which bar they want to go to. We have one button for mid (B), one for high (A), and one for traversal (X). Whenever any of those buttons are pressed in climb mode, we set a latch for that press so that we retain confirmation that the operator wants to transition between all the states to get to that bar. In order to safely transition between states, we have a tolerance on the position of the climber and the robot checks if the climber position is within range of that tolerance before transitioning to the next state, which is its next motion.

We also use the pitch angle of the robot from the NavX to automate transition between specific states. For specific transitions, like going from hanging on the high bar with our passive hooks (HIGH_RETRACT_PASSIVE) to extending the carriage for the traversal bar (TRAV_RUNG_PARTIAL) we check if we have passed a set pitch angle in a specific direction, going from either greater than the set angle to less than it, or vice versa. This allows us to be sure that we're not going to smash any part of the robot that we don't want to into a climber bar.

In the case that the operator sees something that has gone wrong with the climber, we have a manual override that engages whenever the right joystick y-axis is outside of a deadband range (10%), so the operator just moves that joystick past that deadband and the climber will automatically stop moving, and then will be able to be controlled by the



```
35 usages  Aaron Pinto +2
public enum ClimberState {
    7 usages
    MID_RUNG( climberHeight: 55000, Constants.climbMaxVel, Constants.climbMaxAccel),
    2 usages
    MID_RETRACT(Constants.climbSoftTolerance, Constants.climbMaxVel, Constants.climbMaxAccel),
    2 usages
    MID_RETRACT_PASSIVE( climberHeight: 8000, Constants.climbMaxVel, Constants.climbMaxAccel,  pitchAngle: 9.2),
    2 usages
    HIGH_RUNG_PARTIAL( climberHeight: 40000, Constants.climbMaxVel, Constants.climbMaxAccel,  pitchAngle: 24.9),
    2 usages
    HIGH_RUNG( climberHeight: 51000, Constants.climbMaxVel, Constants.climbMaxAccel,  pitchAngle: 28.4),
    2 usages
    HIGH_RETRACT_PARTIAL( climberHeight: 47000, Constants.climbMaxVel, Constants.climbMaxAccel),
    2 usages
    HIGH_RETRACT_UNHOOK( climberHeight: 39000,  vel: Constants.climbMaxVel / 2.0, Constants.climbMaxAccel,  pitchAngle: -3.8),
    2 usages
    HIGH_RETRACT(Constants.climbSoftTolerance,  vel: Constants.climbMaxVel / 2.0, Constants.climbMaxAccel),
    2 usages
    HIGH_RETRACT_PASSIVE( climberHeight: 8000, Constants.climbMaxVel, Constants.climbMaxAccel,  pitchAngle: 9.2),
    2 usages
    TRAV_RUNG_PARTIAL( climberHeight: 40000, Constants.climbMaxVel, Constants.climbMaxAccel,  pitchAngle: 24.9),
    2 usages
    TRAV_RUNG( climberHeight: 51000, Constants.climbMaxVel, Constants.climbMaxAccel,  pitchAngle: 28.4),
    2 usages
    TRAV_RETRACT_PARTIAL( climberHeight: 47000, Constants.climbMaxVel, Constants.climbMaxAccel),
    2 usages
    TRAV_RETRACT( climberHeight: 39000, Constants.climbMaxVel, Constants.climbMaxAccel),
    11 usages
    IDLE( climberHeight: 0,  vel: 0,  accel: 0),
    3 usages
    MANUAL( climberHeight: 0, Constants.climbMaxVel, Constants.climbMaxAccel);
```

Final Design

This climber fits 32 inches of travel into just a two-stage elevator, which is done to save weight from the bearing blocks and additional aluminum extrusions while also being a simpler solution to ensure easier maintenance over the course of the season. The elevator is supported by a pair of ¼ aluminum plates allowing for an unobtrusive superstructure that is just as resilient as other moderate superstructure designs.

Material

- 1” x 1” 3/32 wall aluminum extrusion
- 6061-T6 Material
- Bearing block gussets
- Mounting gussets
- 2 types of bearing
- 0.25” radial bearing

- Custom 0.25” HTPE sliding blocks

Power

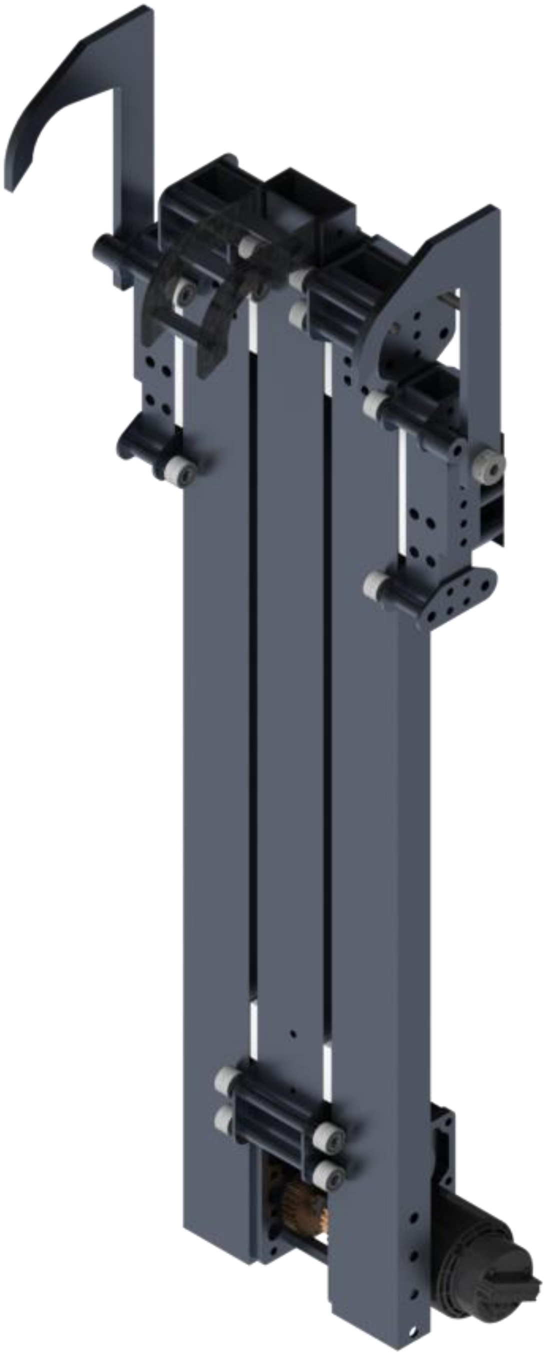
- 1 custom gearbox sporting 2 Falcon 500 motors on a 7:1 overall reduction resulting in 371.91 pounds until stall and a loaded travel speed of 71.2 inches per second
    - 371.91 pounds of stall load allows for the climb to never be affected by stall since it will only lift a maximum of ~150 pounds.

Bearing Blocks

- This year’s climber utilizes bearing block designs that are similar to the designs used in previous years
- Prior to that, we used ball bearings as a sliding surface for the elevator
- This year we opted to use HTPE sliding blocks.
    - This allowed us to save weight, as ball bearings are heavier.
    - Allowed for more space in the middle of the robot for our feeder subsystem, because the sliding blocks are thinner.

Hook

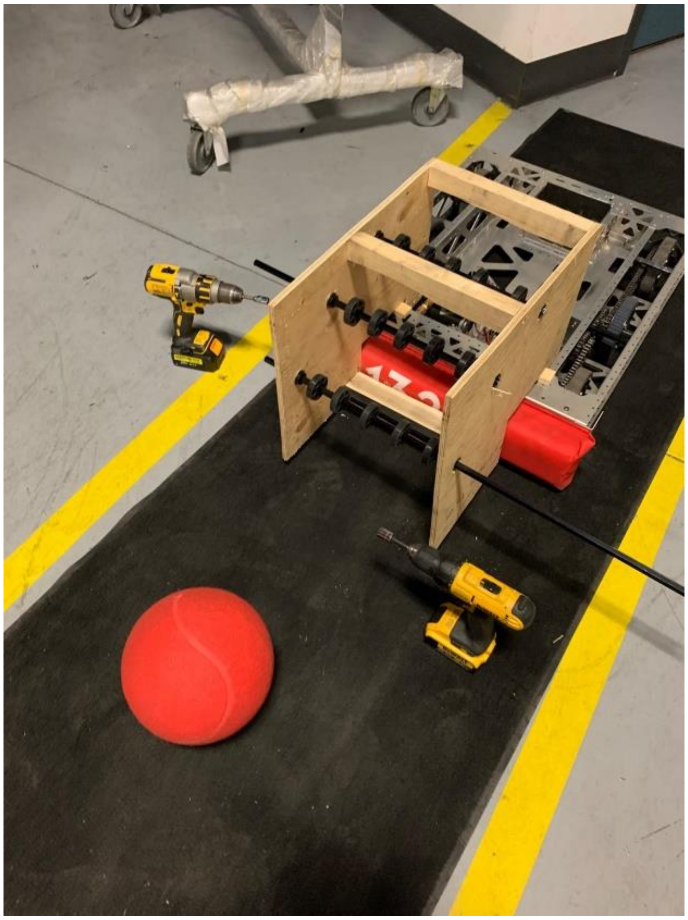
- This year's hooks are machined out of ¼” aluminum and 1/8” polycarbonate.
- The top of the outer hook required a steep slope to be able to move out of the way of the bar when switching between the inner hooks and the outer hooks.
- To make the switch, a spring mechanism was used, which allows for the outer hook to grab onto the bar while the inner hook extends out for the next rung.



Intake

The intake subsystem is one of the more important subsystems, the design and function affect the performance and effectiveness of the feeder and shooter alike. With this in mind, special care was taken to ensure the success of the intake. The intake was made of 2 rollers, powered by NEO-550s, and polycarbonate side plates, with 4" compliant wheels for the lower roller, and mecanums to vector the ball for the upper roller. The two-roller design enables a particularly unique feature on our robot, reject. The intake can automatically reject opposing-alliance-coloured balls, ejecting them through the space between the two rollers. This spacing was determined and tuned in our prototype. The vectored intake roller simultaneously intakes and centers the balls so that they feed into the feeder in a single stream, again, simplifying the overall robot design. The pivot consists of a custom 1:49 gearbox powered by 775 Pros with a final reduction in chain to produce a pivot that is not only fast but strong, reliable, and not subject to losing position.

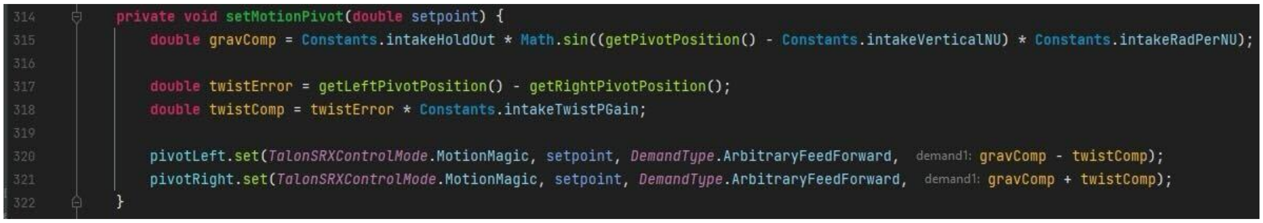
Prototype



Programming

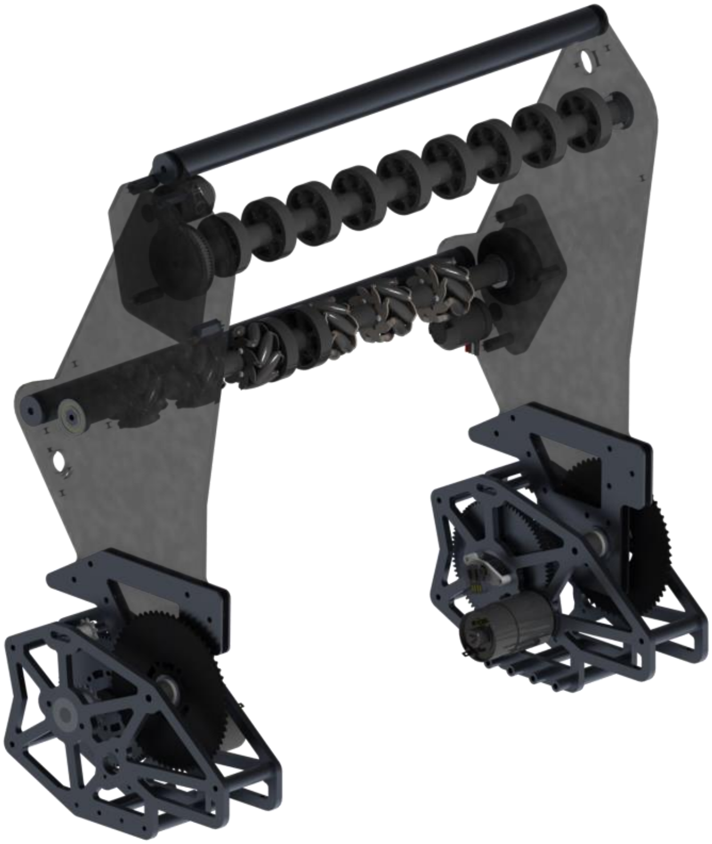
The roller programming is relatively simple, the outer roller spins in the same direction regardless of whether we want to intake or reject the ball. The inner roller matches the outer roller in direction while intaking and opposes it while rejecting. A phone mounted to the climber runs an Android app developed for colour detection using OpenCV. It sends the ball data to the RoboRIO via USB and a TCP socket where the RIO then chooses whether to intake or reject the closest ball. This enables our reject system to be completely automatic.

Two main challenges were encountered when programming the intake pivot, firstly, each side is independently powered, and secondly, gravity has a different effect on the pivot depending on the angle of the intake relative to the ground. The fact that each side is independently powered introduces the risk of the entire intake twisting in the case that one side gets jammed or stuck against a ball. The effect of gravity on the intake pivot additionally introduces a non-linearity on the system. The solution came in the form of a feed-forward term that outputs enough motor power to negate the effect of gravity on the intake, depending on the angle of the pivot. We measured the percentage output of the pivot motors that was required to hold the intake parallel to the ground and then measured the position of the intake when it is in its vertical position. Through simple trigonometry, the percent motor output needed to hold the motor at any angle was determined to be equal to the sine of the angle of the pivot from vertical multiplied by the percent output required to hold the intake parallel to the ground. We also added our own twist compensation in the form of a simple P control loop, to the arbitrary feed-forward demand.



```java
private void setMotionPivot(double setpoint) {
    double gravComp = Constants.intakeHoldOut * Math.sin((getPivotPosition() - Constants.intakeVerticalNU) * Constants.intakeRadPerNU);

    double twistError = getLeftPivotPosition() - getRightPivotPosition();
    double twistComp = twistError * Constants.intakeTwistPGain;

    pivotLeft.set(TalonSRXControlMode.MotionMagic, setpoint, DemandType.ArbitraryFeedForward,  demand1: gravComp - twistComp);
    pivotRight.set(TalonSRXControlMode.MotionMagic, setpoint, DemandType.ArbitraryFeedForward,  demand1: gravComp + twistComp);
}
```

Final Design



Feeder

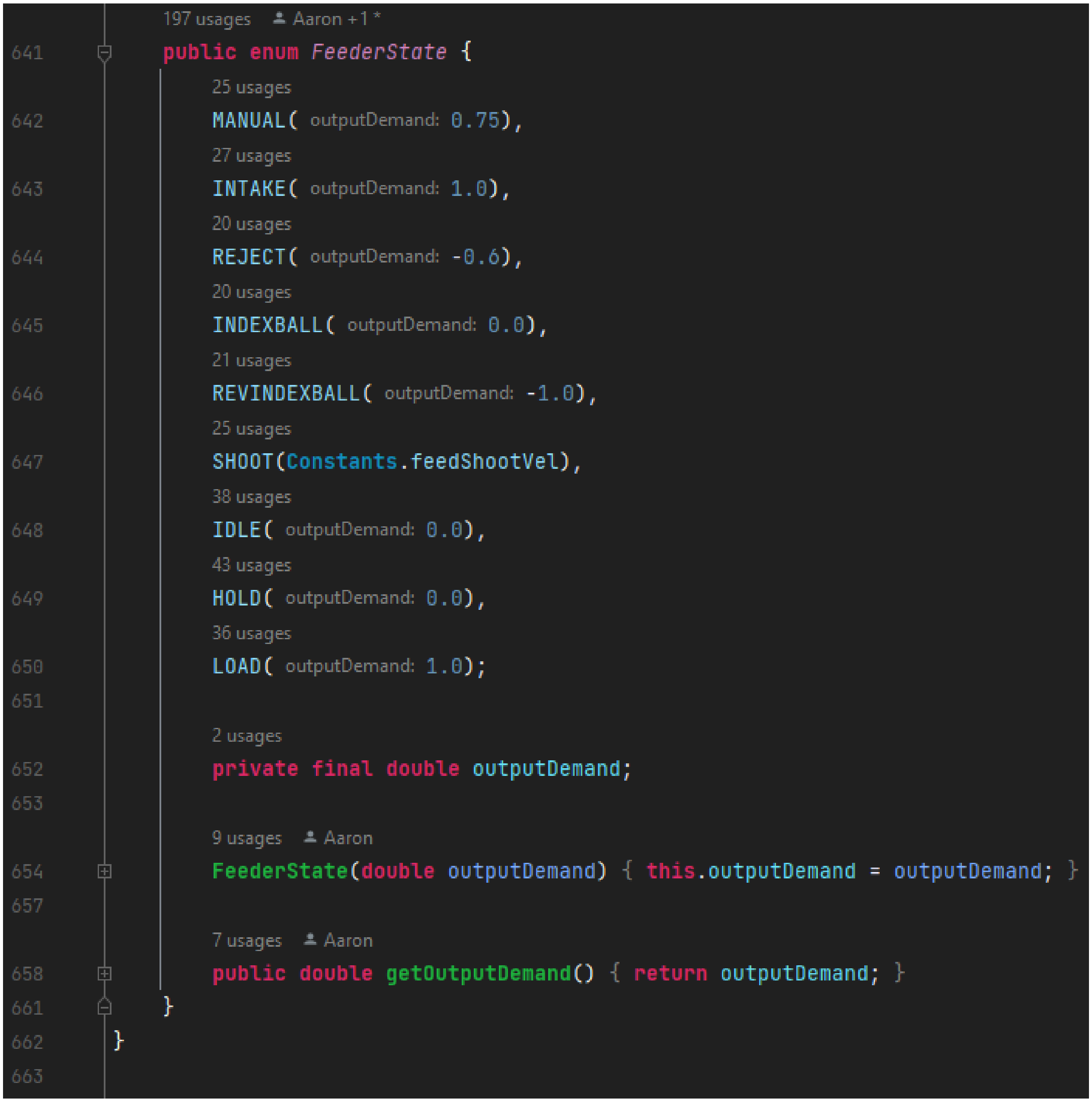
The feeder is used to store the cargo in the robot. The feeder can hold 2 cargo in the robot at a time. The main goal of the feeder is to transport cargo from the intake to the shooter. The feeder utilizes belts and pulleys to move the balls through the system.

Programming

The feeder programming was another state machine, with different states to handle different output percentages, speeds, and positions for the subsystem. The transitions between states only happened when it was safe to do so, and usually on a beam break rising or falling edge. For example, when we want to shoot and are not on target, the feeder will move the balls towards the shooter (LOAD), and then stop when the top beam break is broken and transition into the HOLD state, which makes the feeder actively hold its position, so that if we get hit, a ball isn't going to randomly get shot.

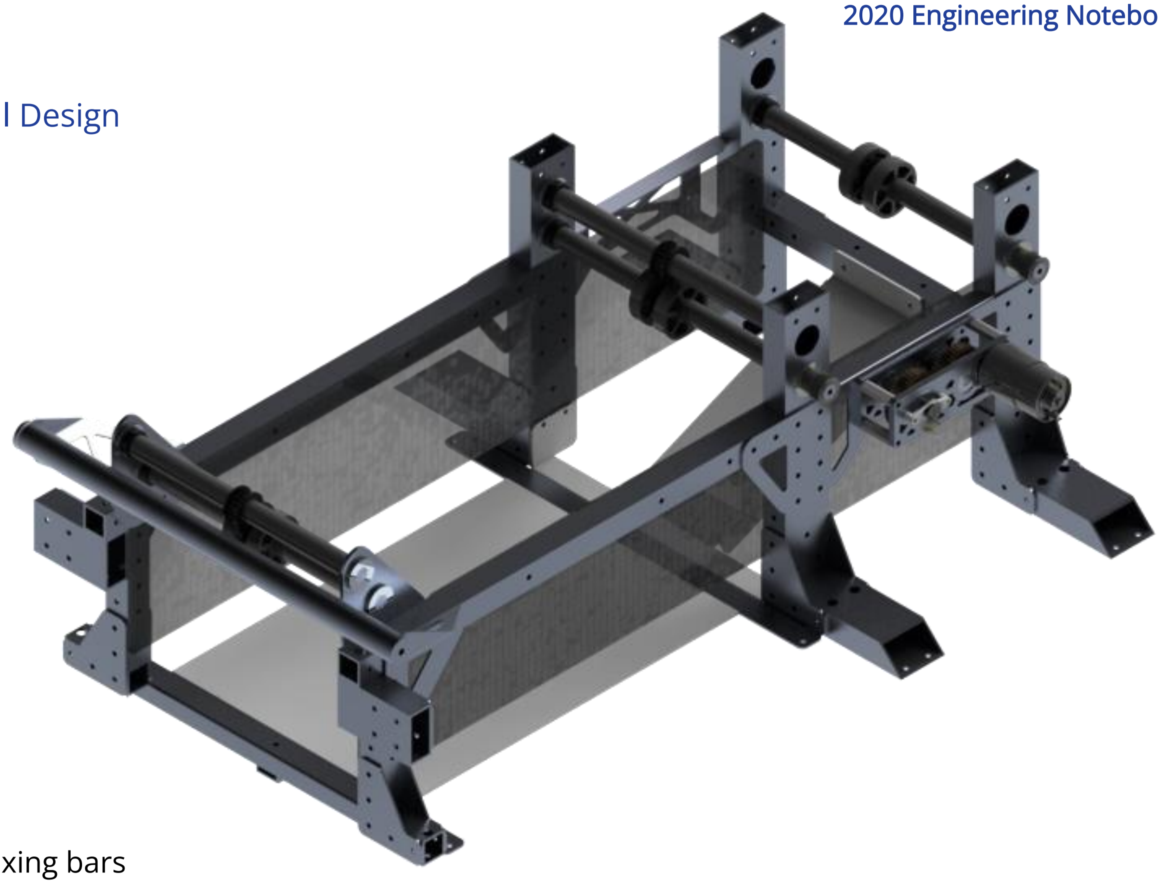
We also have a ball counter that actively keeps track of the number of balls in our robot. We have 2 beam breaks at the front of the robot, and whenever we intake a ball, the beams are broken and not broken in a specific order, which we keep track of and if that sequence matches what we expect, we increment a counter. We can also reject a ball through the feeder, which is the same beam break sequence as intaking, but just in reverse, so we also handle decrementing the counter when that occurs. Finally, we can shoot a ball to decrement the counter, which requires the top beam to go from broken to not broken, and a drop in RPM of the shooter flywheel & rollers to occur, so we check if that has happened at any time, and decrement accordingly.

In case some of the sequence triggers don't match, and yet we still managed to intake a ball, we have a clear safety, where if the feeder is moving for a certain amount of time and none of the beams have been broken, we automatically set the ball counter to 0. In case any of the beam break sensors fail, we have a manual ball counter clear via a button press on the operator controller.



```
    197 usages   Aaron +1 *
    public enum FeederState {
        25 usages
        MANUAL( outputDemand: 0.75),
        27 usages
        INTAKE( outputDemand: 1.0),
        20 usages
        REJECT( outputDemand: -0.6),
        20 usages
        INDEXBALL( outputDemand: 0.0),
        21 usages
        REVINDEXBALL( outputDemand: -1.0),
        25 usages
        SHOOT(Constants.feedShootVel),
        38 usages
        IDLE( outputDemand: 0.0),
        43 usages
        HOLD( outputDemand: 0.0),
        36 usages
        LOAD( outputDemand: 1.0);

        2 usages
        private final double outputDemand;

        9 usages   Aaron
        FeederState(double outputDemand) { this.outputDemand = outputDemand; }

        7 usages   Aaron
        public double getOutputDemand() { return outputDemand; }
    }
}
```

Final Design



Indexing bars

- Towards the front of the feeder, as well as on the intake, there are metal bars sitting close to the rollers to make sure the balls are vectored into the feeder structure and not out of it.

Tension block

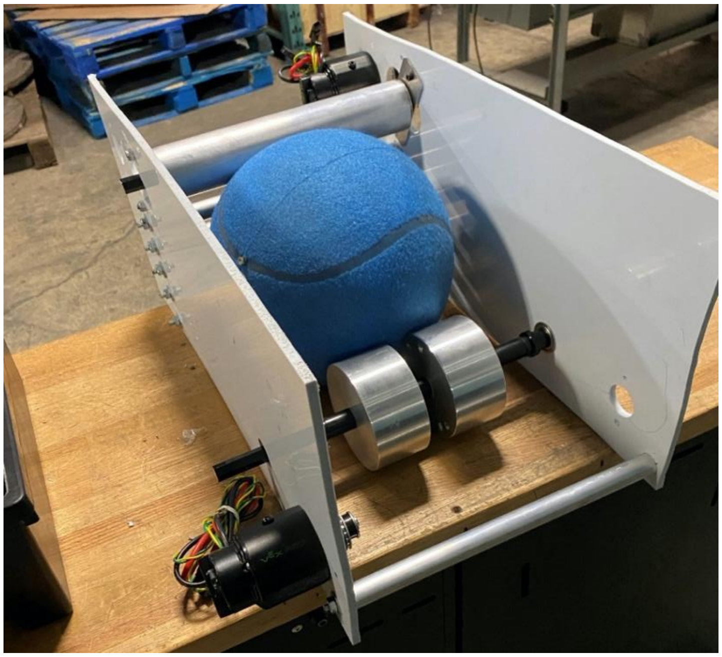
- As a result of the prototype geometry, we were unable to find an exact belt size
- In order to compensate for this, we added idlers.

Shooter & Turret

The shooter is an important subsystem on this years' robot, it is the primary method of scoring and will ultimately decide the success of the season. Our goal for this shooter was to be able to shoot consistently into the upper goal from anywhere on the field, this would enable us to be undefended and unblocked. Unlike previous years, the shooter only consists of fly wheel and back roller assemblies, each with specific criteria that needed to be met in order to remain competitive. The flywheel is a 4" neoprene wheel on a 1:1 reduction powered by a Falcon 500 motor. The hood for this year is static, meaning there is only one release angle. However, by varying the rpm in the active rollers within the hood, the trajectory of the ball is easily modified without a second hood position. These rollers are also driven by a Falcon 500 motor on a 2.5:1 reduction.

Early in the season, we decided to mount the shooter onto a turret that would allow the shooter to aim independent of the drivetrain. This has some innate advantages, it makes lining up easier for the drivers and limits the effects of drivetrain position on shooting, which is useful when playing under defence. The turret rotates on a custom HDPE bushing and is powered by a custom 77.78:1 gearbox capable of 2 rotations per second, this simple construction limits failure points and increases overall reliability over other turret designs.

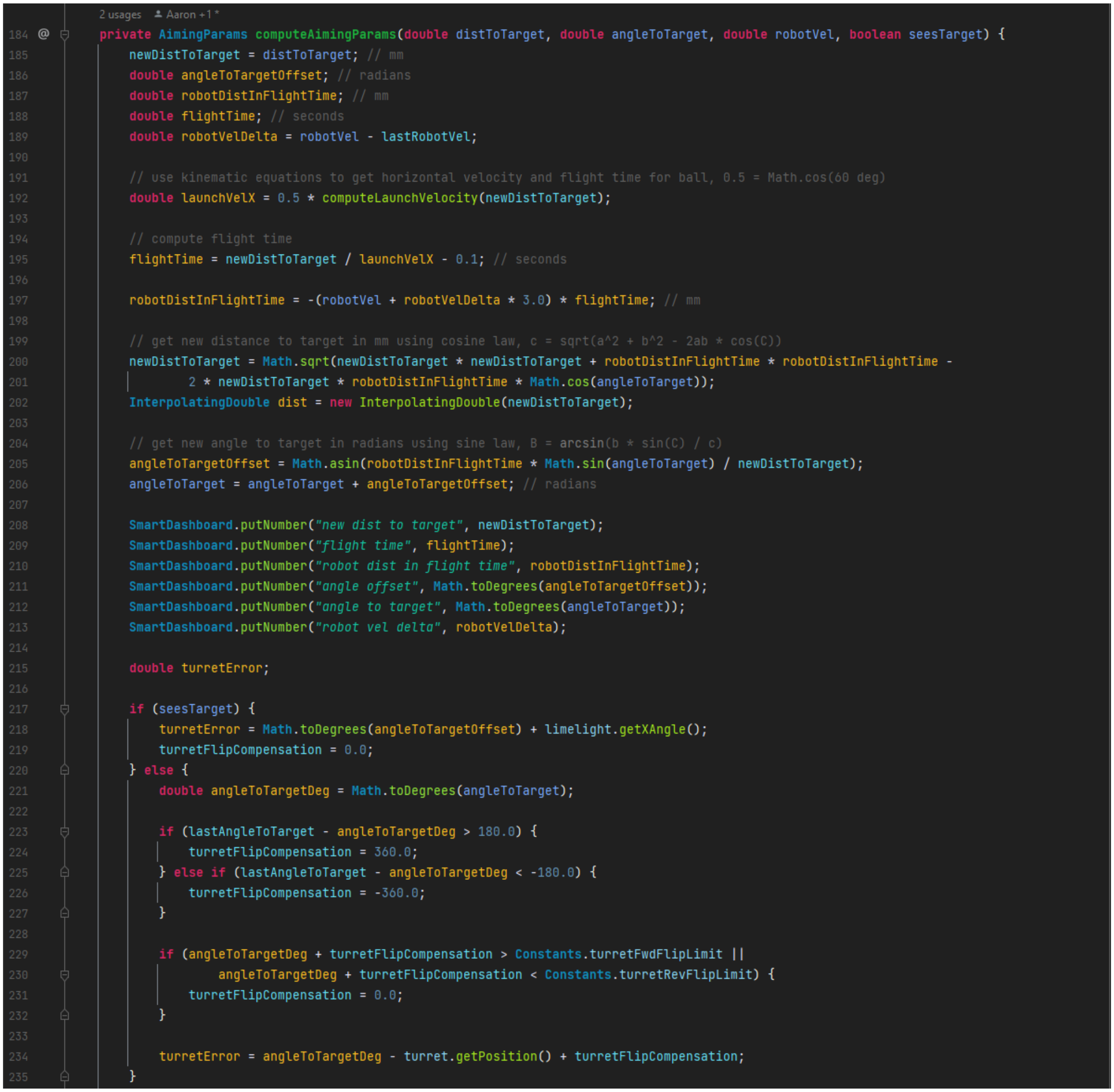
Prototype



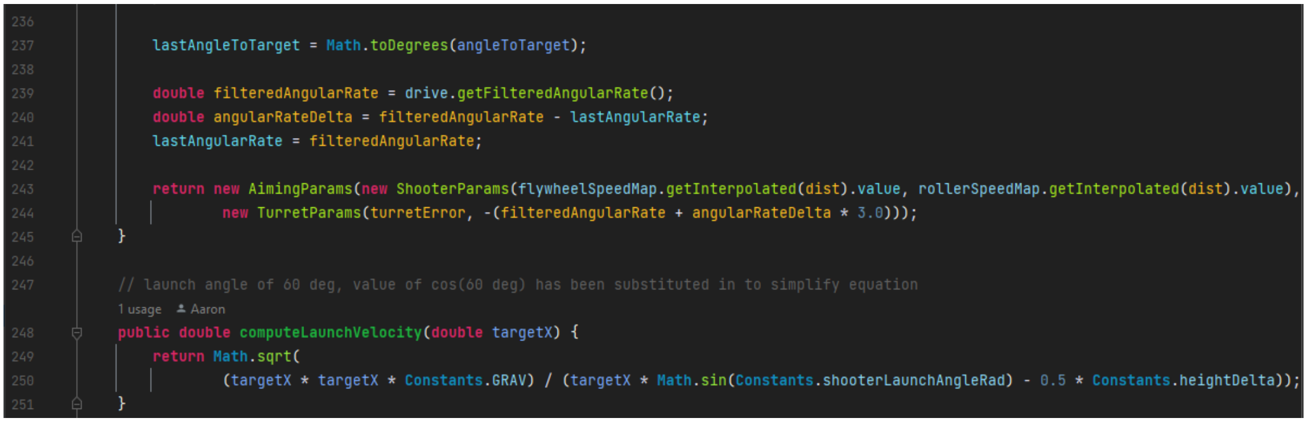
Programming

We created a prototype of the shooter using calculations from a spreadsheet to determine shooter arc. We wanted to make a shooter with only one release angle that could shoot from anywhere on the field. To achieve this, we use varying rpm calculated by the spreadsheet. Iterating from 2020, we had a static hood with 2 positions, which proved to create extra friction and slowed down the release of the ball. This year we went with active rollers in the hood to aim, so a second position was not needed. The prototype also showed that the active hood helped with shooting low psi balls, because we were able to transfer enough energy in the balls to propel them into the goal.

For lining up our turret we use a combination of encoders and computer vision through the limelight. We use the data from the limelight and some trigonometry involving known variables to determine our distance used to calculate an interpolated speed map, varying our rpm to change our projectile motion. We also use the limelight to cache data involving our positioning on the field, allowing us to always track the target even if the limelight fails. We also have functions to assist the driver and automate driver functions, such as a button to zero the position of the turret.



```java
private AimingParams computeAimingParams(double distToTarget, double angleToTarget, double robotVel, boolean seesTarget) {
    newDistToTarget = distToTarget; // mm
    double angleToTargetOffset; // radians
    double robotDistInFlightTime; // mm
    double flightTime; // seconds
    double robotVelDelta = robotVel - lastRobotVel;

    // use kinematic equations to get horizontal velocity and flight time for ball, 0.5 = Math.cos(60 deg)
    double launchVelX = 0.5 * computeLaunchVelocity(newDistToTarget);

    // compute flight time
    flightTime = newDistToTarget / launchVelX - 0.1; // seconds

    robotDistInFlightTime = -(robotVel + robotVelDelta * 3.0) * flightTime; // mm

    // get new distance to target in mm using cosine law, c = sqrt(a^2 + b^2 - 2ab * cos(C))
    newDistToTarget = Math.sqrt(newDistToTarget * newDistToTarget + robotDistInFlightTime * robotDistInFlightTime -
            2 * newDistToTarget * robotDistInFlightTime * Math.cos(angleToTarget));
    InterpolatingDouble dist = new InterpolatingDouble(newDistToTarget);

    // get new angle to target in radians using sine law, B = arcsin(b * sin(C) / c)
    angleToTargetOffset = Math.asin(robotDistInFlightTime * Math.sin(angleToTarget) / newDistToTarget);
    angleToTarget = angleToTarget + angleToTargetOffset; // radians

    SmartDashboard.putNumber("new dist to target", newDistToTarget);
    SmartDashboard.putNumber("flight time", flightTime);
    SmartDashboard.putNumber("robot dist in flight time", robotDistInFlightTime);
    SmartDashboard.putNumber("angle offset", Math.toDegrees(angleToTargetOffset));
    SmartDashboard.putNumber("angle to target", Math.toDegrees(angleToTarget));
    SmartDashboard.putNumber("robot vel delta", robotVelDelta);

    double turretError;

    if (seesTarget) {
        turretError = Math.toDegrees(angleToTargetOffset) + limelight.getXAngle();
        turretFlipCompensation = 0.0;
    } else {
        double angleToTargetDeg = Math.toDegrees(angleToTarget);

        if (lastAngleToTarget - angleToTargetDeg > 180.0) {
            turretFlipCompensation = 360.0;
        } else if (lastAngleToTarget - angleToTargetDeg < -180.0) {
            turretFlipCompensation = -360.0;
        }

        if (angleToTargetDeg + turretFlipCompensation > Constants.turretFwdFlipLimit ||
                angleToTargetDeg + turretFlipCompensation < Constants.turretRevFlipLimit) {
            turretFlipCompensation = 0.0;
        }

        turretError = angleToTargetDeg - turret.getPosition() + turretFlipCompensation;
    }
```



```java
        lastAngleToTarget = Math.toDegrees(angleToTarget);

        double filteredAngularRate = drive.getFilteredAngularRate();
        double angularRateDelta = filteredAngularRate - lastAngularRate;
        lastAngularRate = filteredAngularRate;

        return new AimingParams(new ShooterParams(flywheelSpeedMap.getInterpolated(dist).value, rollerSpeedMap.getInterpolated(dist).value),
                new TurretParams(turretError, -(filteredAngularRate + angularRateDelta * 3.0)));
    }

    // launch angle of 60 deg, value of cos(60 deg) has been substituted in to simplify equation
    1 usage  Aaron
    public double computeLaunchVelocity(double targetX) {
        return Math.sqrt(
                (targetX * targetX * Constants.GRAV) / (targetX * Math.sin(Constants.shooterLaunchAngleRad) - 0.5 * Constants.heightDelta));
    }
```

Final Design

Purpose

- To shoot cargo consistently and with high accuracy
- To shoot cargo from anywhere on the field
- To shoot multiple cargo in quick succession with minimal loss to accuracy.

Materials

- The shooter side plates, and hood assembly is constructed out of ¼ Poly Carb
- Shafts and milled parts are constructed out of 6061 aluminum

Wheels

- Began with aluminum flywheel and roller wrapped in neoprene tape on prototype, replaced with 4” 60A neoprene flywheel and 1.5” 60A neoprene rollers for the robot.

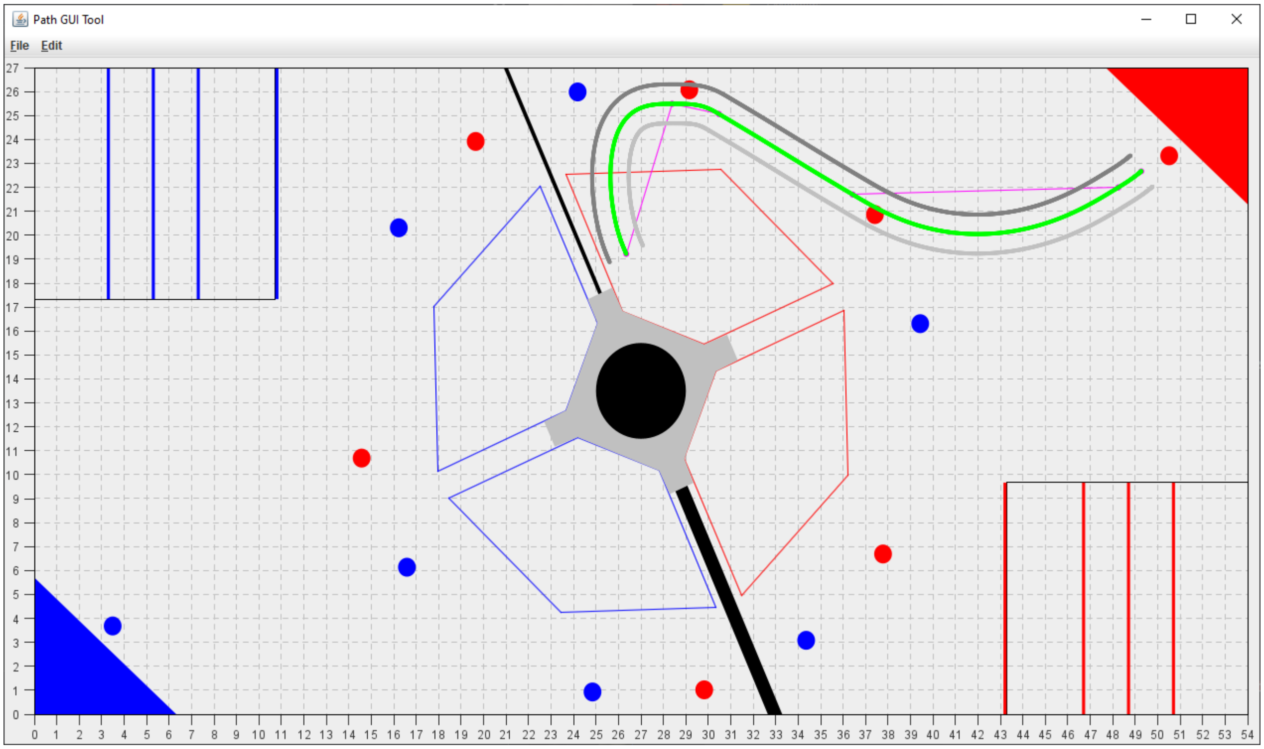
Power

- Powered by two Falcon motors. One controlling the set of smaller rollers in the hood (2.5:1 reduction) and the other motor running the flywheel (1:1 reduction)

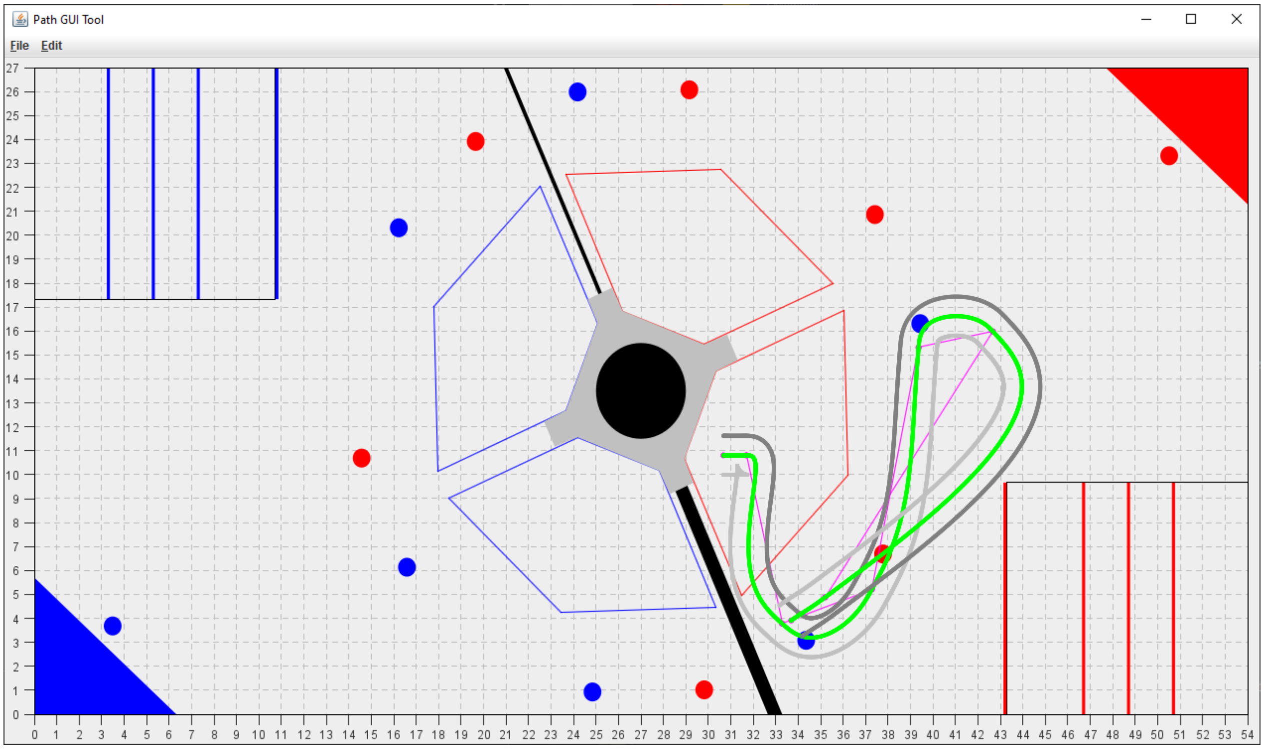
Autonomous

For the autonomous period of the game, there are pre-set paths for 2 and 5 cargo autos' as shown in the images below. Using encoder and gyro feedback from the drivetrain, as well as the pre-programmed automatically aiming shooter, **OVERTIME** can shoot as its moving towards the next piece of cargo. In the case of our five cargo auto, we rely on the human player to roll a ball into the robot from terminal and intake our 4th cargo as we proceed towards the 5th. Our second auto path scores 3 cargo of our alliance and rejecting 2 cargo of the other alliance into our Hanger zone. This makes it harder for the other alliance to score those 2 cargos during teleop. Using path following we were able to keep the robot on its course throughout the auto despite disruptions.

(Image of the 5 cargo autonomous path in the gui developed by the programming team)



(Image of the 3 cargo + 2 reject auto path)


Path GUI Tool
File Edit
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54