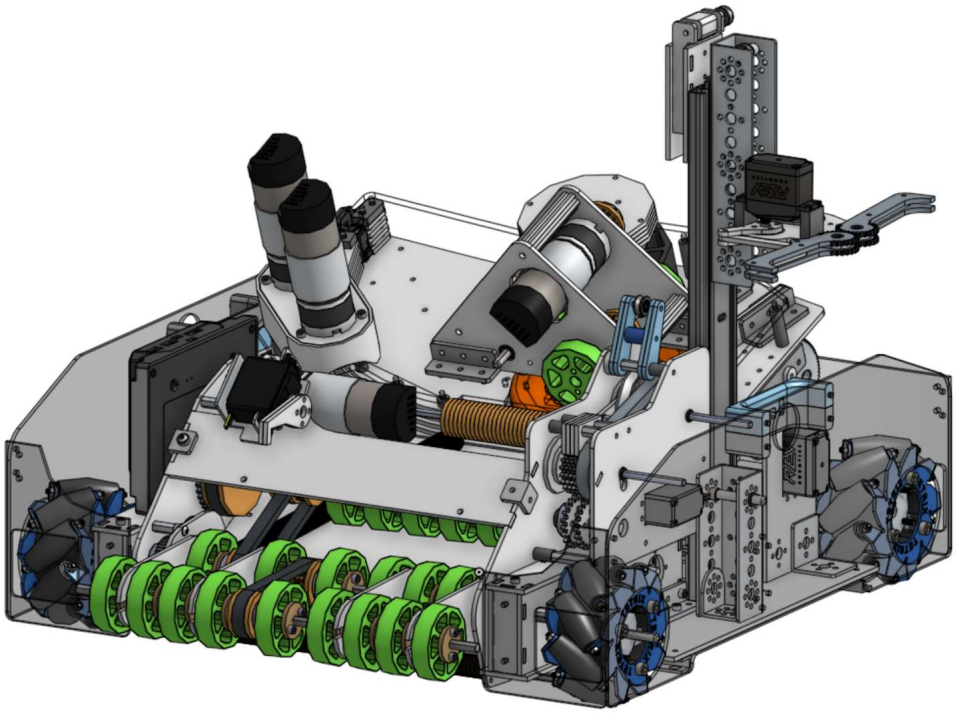


FTC Team 15317 Error Code 404
Presents:
Prometheus

Engineering Portfolio 2021

Team 15317 Error Code 404

Our Team

Error Code 404 was started in 2018 as the second team at Harrisonburg High School. We have Always been a small team, usually consisting of no more than five members. This year, our four members, two designers/builders, and two programmers, have spent countless hours to create our best robot yet - during a pandemic.

COVID-19 Meeting Plan

With a pandemic ongoing, our team shifted to heavily rely on virtual meetings to design our robot. This included a major switch to using computers to design our robot instead of doing it hands-on. This was a major stepping stone for our team because we were rarely using CAD beforehand. At the end, this greatly improved our robot. For the actual construction, we were restricted to meeting at school for two days a week, for three hours each. All of these considerations all factored into our approach to this year's challenge.

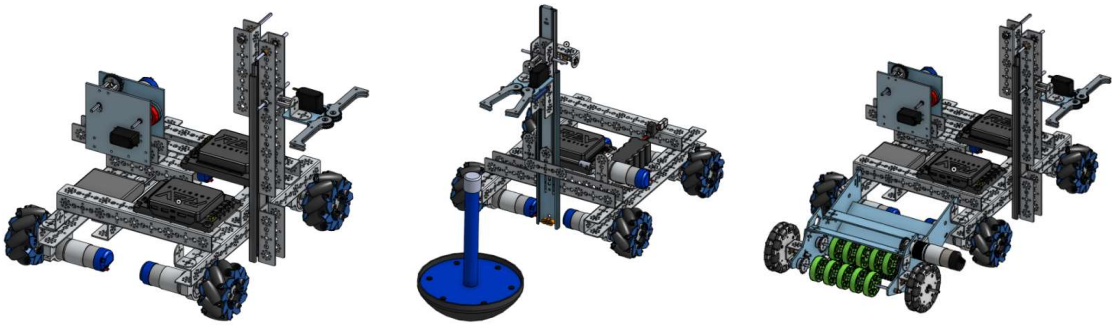
Constraints

For the build team, we had a couple major constraints. The first, was time. Because we were limited to six hours a week, we had to make effective use of our time at school. This is what led to our detailed planning in CAD. We also knew from the start that our most accurate tool was our laser-cutter. So, we constantly designed with that tool's capabilities in mind, and made our robot so that it was easy to manufacture with the laser.

To learn more about our team, including our outreach initiatives, Engineering Notebook, and code, please visit our website: ftc15317.weebly.com

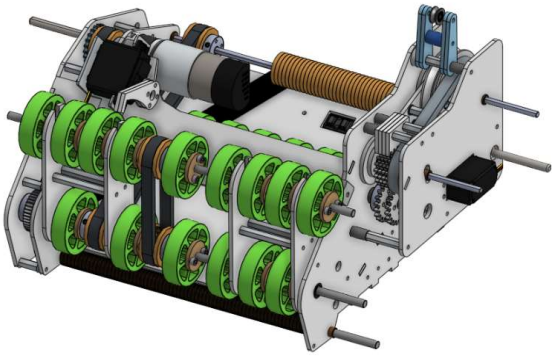
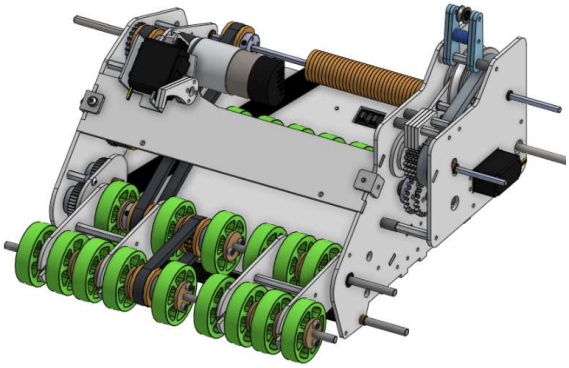
Mechanical Design:

Test Chassis



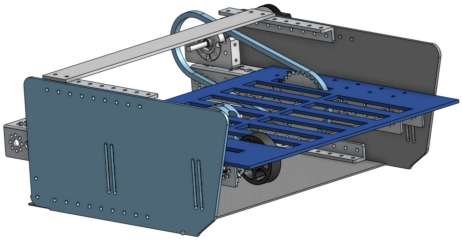
One prototype we created that greatly benefited the programmers was the test chassis. This was a super simple design made mostly out of tetrix parts that was used by the programmers to get the drive program tested as early as possible. The chassis was also used to test other prototypes such as the claw, PTO, lift, and collector.

The Collector

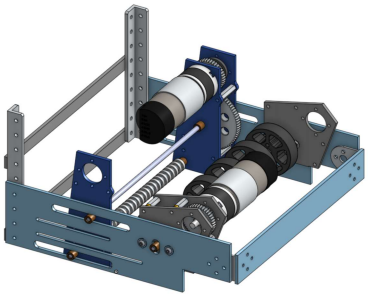
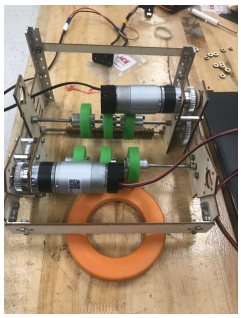



The collector on our robot was designed with speed and reliability in mind. This collector consists of five separate rollers, powered by a single motor, and a ramp. Three of these rollers consist of two inch compliant wheels rotating to pull the ring inwards from the top. The other two rollers consist of many textured wooden rings stacked next to each other. One of these wooden rollers is located just above the playing surface and is used to help lift the rings off the floor and up the ramp. This collector features a drop down front roller in order to give us more room in the robot itself. This drop down roller is held back before the match with a servo.

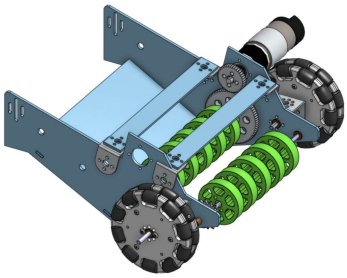
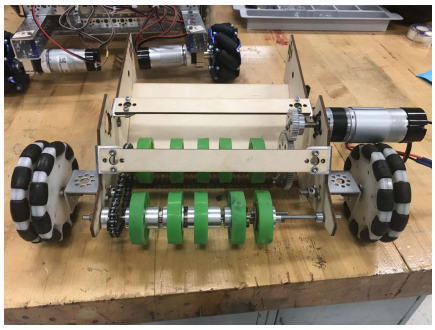
Collector prototypes:




The first prototype was a system of rollers with an adjustable compression and adjustable ramp angle. After testing this prototype we wanted to add a bottom roller to help the rings get lifted off the floor.

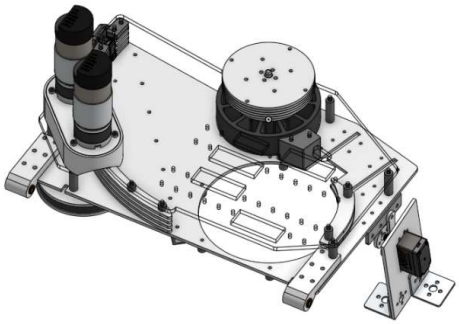
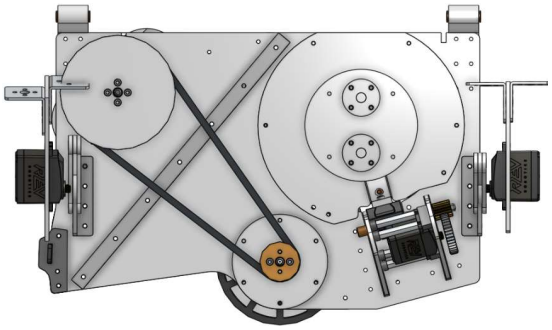



The second prototype consisted of three rollers with adjustable distances between them and adjustable compression with the ring.

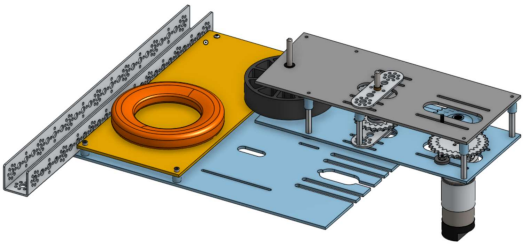
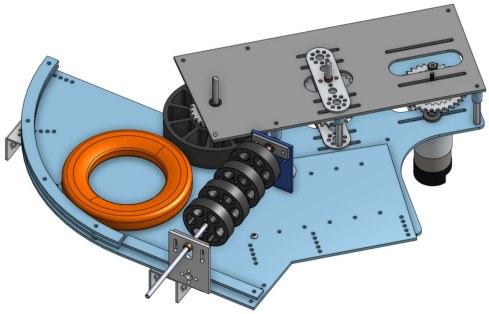
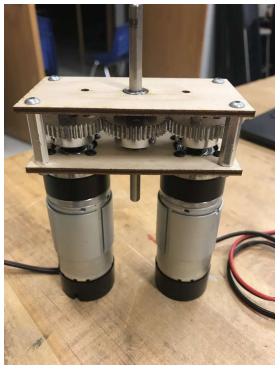
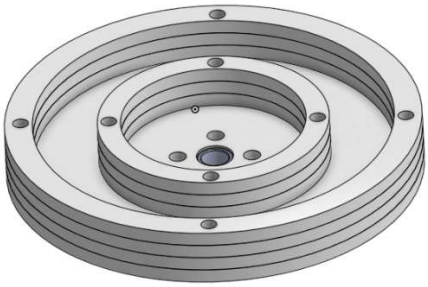
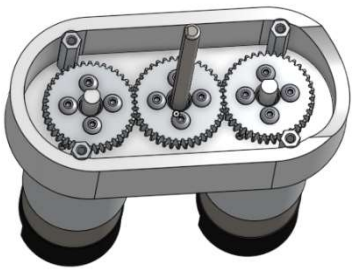
The last prototype was created using the test results of the previous prototype and was built in a solid configuration. This prototype was built to fully test the configuration and add it to the programmers test chassis for further testing.

The Shooter

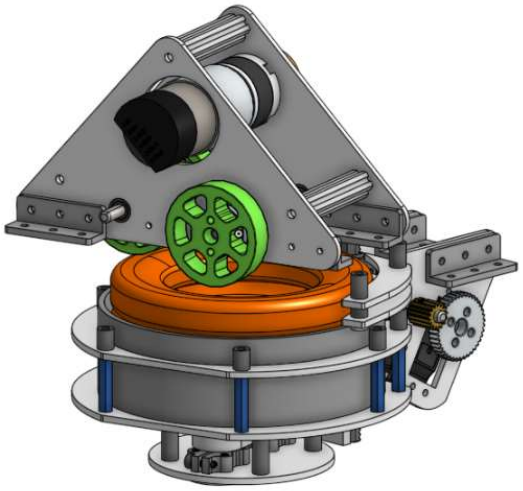
The shooter on our robot was designed to be as fast, precise and compact as possible. This shooter consists of a flywheel that is powered by two 3.7 motors with a 3:1 increase and an arc that is part of the main frame of the shooter. The hopper attached to the shooter pushes the rings just enough to contact the flywheel which causes the ring to rapidly accelerate around the arc turning 90 degrees and exiting the shooter at a high velocity. This shooter was designed to pivot up and down with a range of about ten degrees. This pivoting action is controlled by two servos on either side of the shooter. This allows us to have more variability in adjusting our aim.

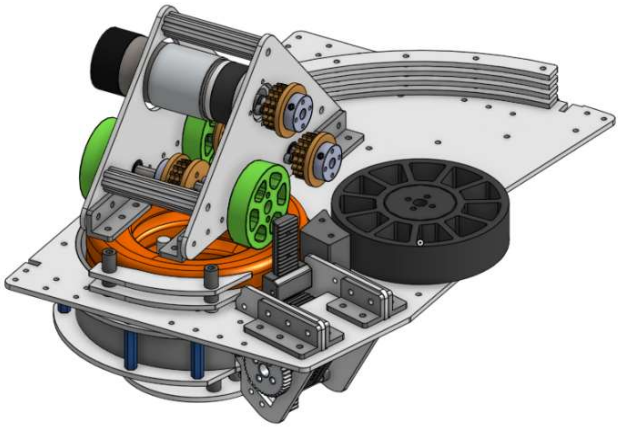
Previous shooter prototypes:

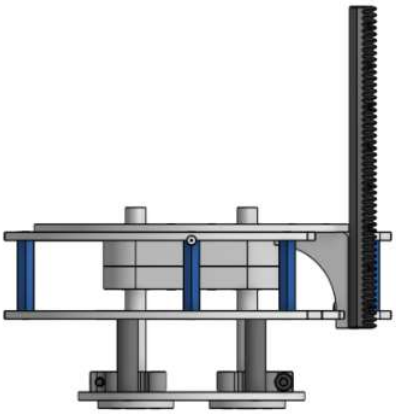
We started off with two main shooter prototypes, a straight shooter and a curved shooter. Each of these was designed with the supporting use of any motor, nine different gearing ratios and five different compression measurements. Through some very extensive testing, we determined the best motor to use, the best gear ratio and the best compression. All of this information was used to design the final shooter. We also tested a dual motor gearbox, that we designed ourselves, and an inertia wheel. These two additions allow the fly wheel to spin up quicker and not slow down as much between each shot, allowing us to shoot at a faster rate.

The Hopper





The hopper of our robot has been designed to be an efficient, quick, and reliable storage and dispensing mechanism. Our hopper can allow rings to pass through smoothly, but can also retract its storage plate to hold rings when we decide to collect more than one. To dispense the rings, the plate moves upwards, pressing the rings against powered wheels overhead. This causes the ring to get pushed directly into the shooter. This sequence can be very fast, allowing our robot to score quickly.

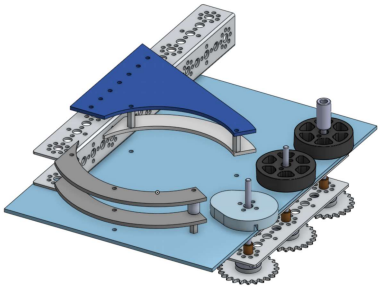
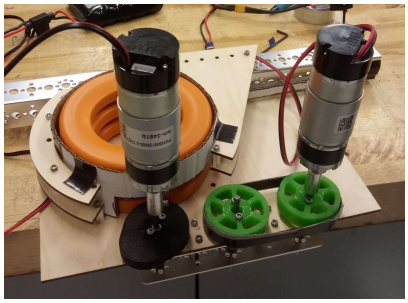
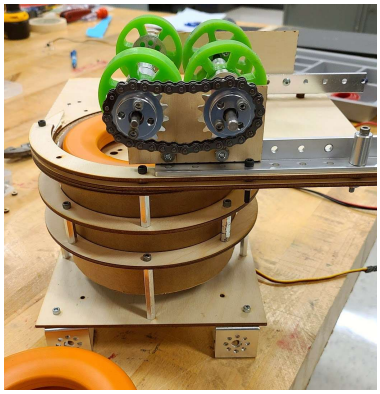
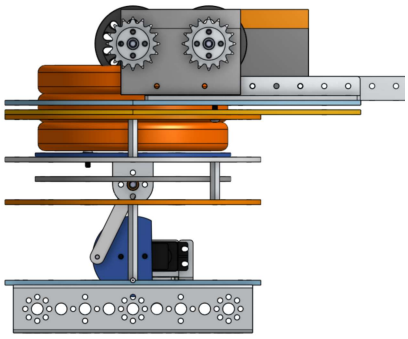
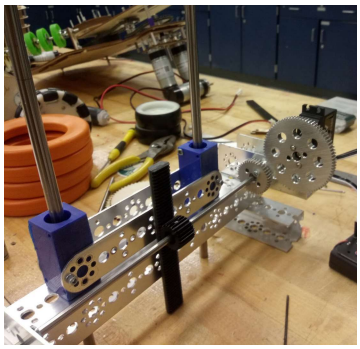


Our design utilizes a rack and pinion system to move the base plate up/down. This is powered by a servo, so that we can hold accurate positions. This moving plate is constrained to a single-degree of movement by two linear rods. These are set as guides for bearings to roll on.

This complex design was a result of several tests. We first started with a more common approach - allow the discs to settle down in a cylindrical container. However, we realised that an upwards-moving hopper was more efficient, and did not rely on gravity for the discs to settle down.

We then started with our second prototype, which worked by allowing our base plate to ride along a cardboard tube. This design was powered by a servo in a piston-like setup. While it worked, we improved on the design in two main ways; the rigidity (linear rods) and the actuation (rack and pinion). This allowed for a smaller hopper that was more robust.

Previous hopper prototypes:

Reading top-left to right, bottom left to right:

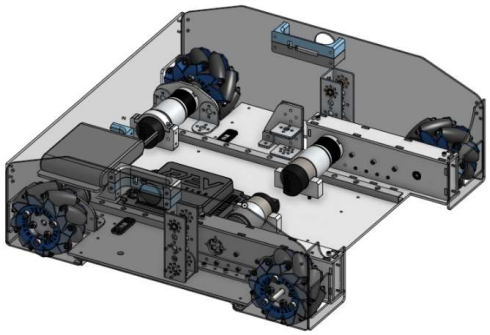
-The first image was created with a pegboard, some screws, and cardboard. It was based off a common hopper - load the discs from the top, dispense from the bottom. However, we used motors to power the wheels - faster than servos.

-The second and third image were of the second prototype. This built off of the first iteration, and used a “cam” to pusch the rings out. The cam design allowed the rings to settle properly, but contact the wheel when dispensing. This design was fast, but overall inefficient.

-The fourth and fifth images display the third hopper we created. This design collected rings similar to the previous designs, but dispensed the rings by moving upwards. This design was too tall and not robust enough, but was an important proving ground.

-The last picture shows a test before our final hopper design. This held a platform on linear rods, and used a rack and pinion to move the platform up/down. This design was miniaturized for the final design.

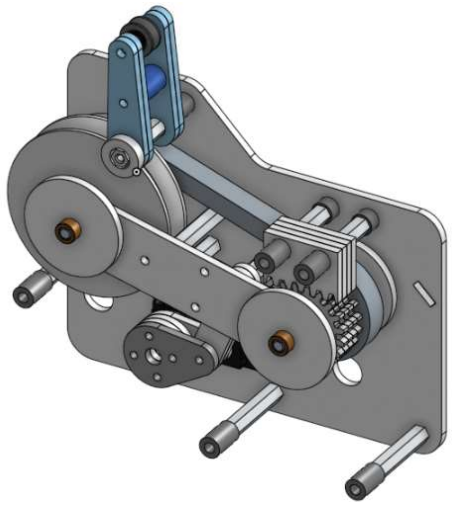
Final Drive Base



The final design for our drive base is based on a mecanum wheel drivetrain. The motors for the front wheels are set back and use chain to power the front wheels, this allows more room for the collector to fit in place. The rear wheels are directly connected to the motors which are attached to plates anchored to the baseplate. Bumpers were added to help mount the cameras and hold everything together.

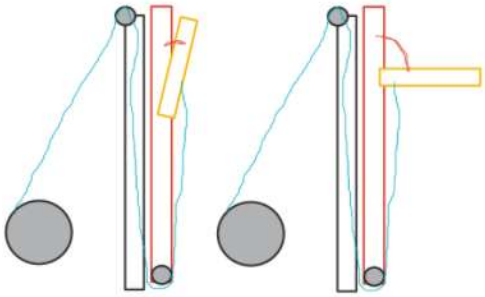
Wobble Goal Mechanism

Early in the season, we decided to make our wobble goal mechanism a lift. A lift requires less space on the robot horizontally, and can easily lift the wobble goal. These are usually viewed as a simple design, however, our lift has some unique features:



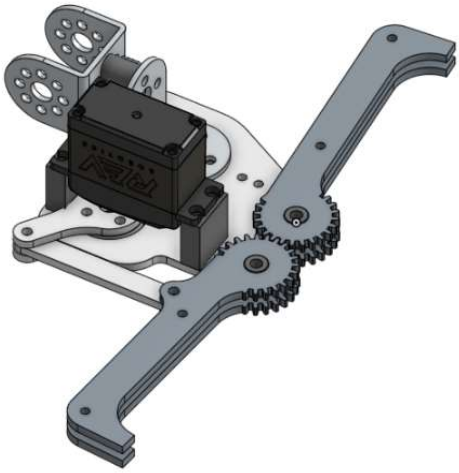
The PTO

Because we decided to use two motors for the shooter we realized we wouldn't have a motor to run the lift (for the wobble goal). To solve this issue we designed and incorporated what we call a PTO. This mechanism effectively transfers the power from the collector to a pulley that can lift the linear slide. The PTO utilizes a servo to move a gear up and down either engaging the gear to a gear on the collector or engaging the gear to a hard stop.



Stringing of the Lift

Our wobble goal mechanism has an arm that pivots outwards, and a lift that moves upwards. To avoid adding an extra servo, we thought up a way that will automatically extend/retract the arm with the same string that extends/retract the lift. To do this, we anchor our string on the pivoting arm. We then bring the string down to the bottom of the moving stage, wind back up to the fixed stage, and then route the string to our spool. When the string is pulled, it will force the arm to be pulled down, and will then move the lift. To retract the arm, we used surgical tubing to have the arm pivot back naturally.

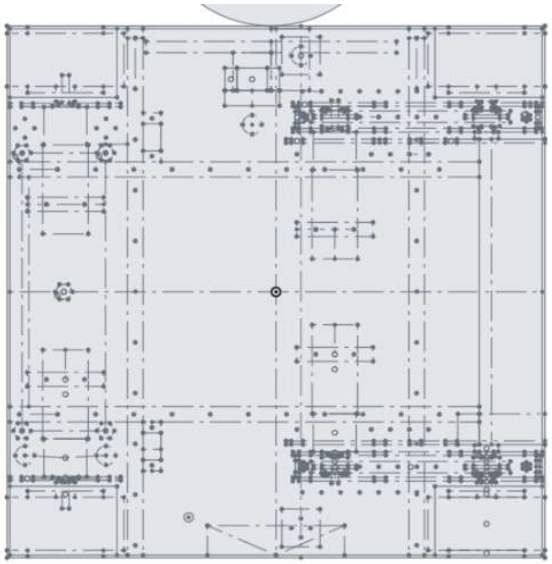


Dual Pivoting Fingers

In order to maximize the effectiveness of our claw, we wanted both of the fingers to move. Again, we did not want to add additional servos. So, we took inspiration from a 2019 FRC robot and used two "fingers" with a gear mesh. This forces both fingers to move with the rotation of just one, and if properly made, the two will meet in the middle. We also did not want the servo to take any moment loads. So, we allowed our two fingers to pivot on separate bushings. We then used a 4-bar linkage with a servo to open/close the claw.

Starting the Final Design

Design of the final robot was done primarily through detailed side-view and top-view sketches. These sketches were later extruded as plates and laser-cut.

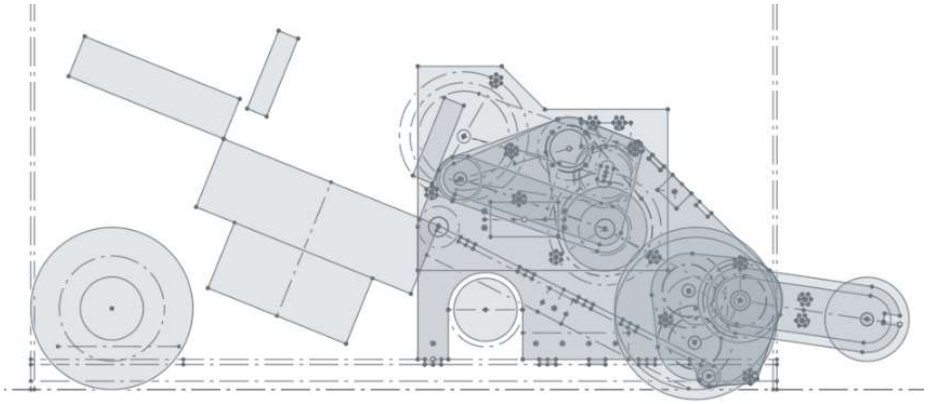


Top Sketch, Chassis

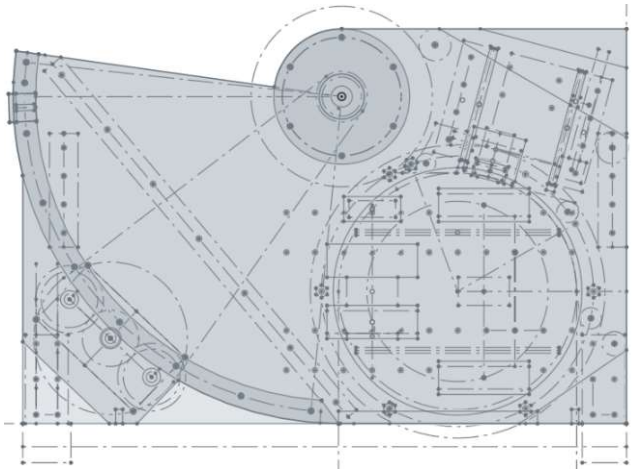
For the top sketch, our main focus was the chassis. This was because our chassis was going to be made with one large plate acting as the main structure of the robot. This approach allowed us to mount structures almost anywhere it needed to be. Through the use of reinforcement, this plate created a strong foundation to the robot.

Side Sketch, Collector

This sketch included everything to do with the collector; the spacing of the wheels, the placement, the structure, and how the collector fed into the shooter.



This image consists of the several sketches we created on top of each other. This allowed us to reference common parts, but still keep our sketches (somewhat) neat. In the end, these sketches contained a massive amount of information about our robot, so they were complex.

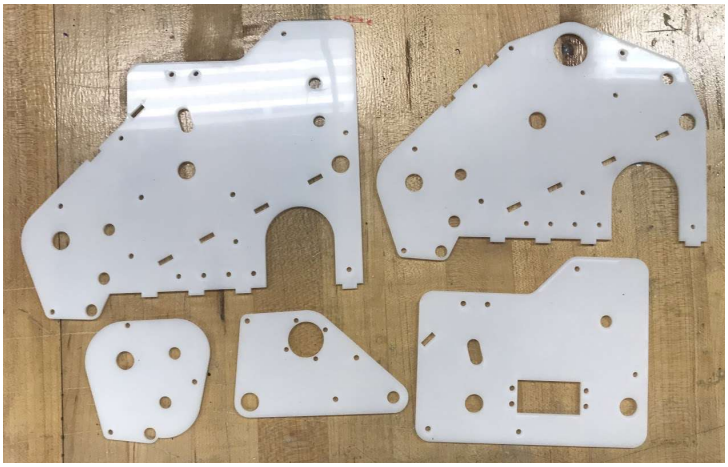


Shooter Top Sketch

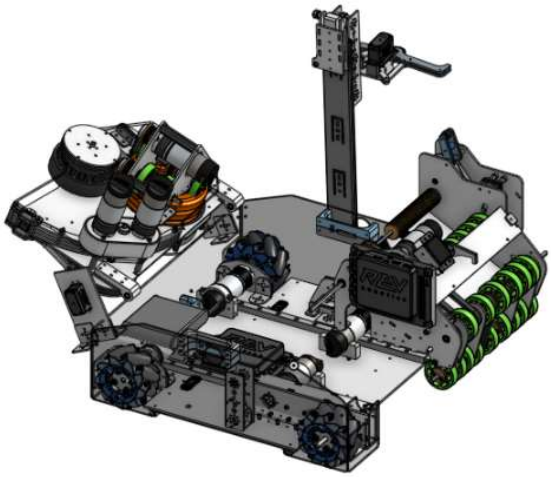
This sketch was created with one major constraint: size. With information from the above sketches, we were able to create a rectangle in which the shooter was limited to. We then added all the componente, from the shooting wheel, to the hopper. We included the top view parts of the hopper in this sketch to better plan out its placement on this sub-system.

Manufacturing

Our team used several tools in the process of manufacturing, however, we relied heavily on our school's laser-cutter and 3d printers. These tools were always in the forefront of our design; if there was a way to reduce supports when printing, we did it.



We also took time this year to select better materials. Our primary material is delrin, which is a branded acetal. This plastic does not shatter easily, is easy to laser cut, and is dimensionally stable. It is also easy to thread, which we did frequently. In addition to delrin, our team slo used wood and acrylic where required.



Overall Robot Concepts

Modular - Our robot has three main modules; the collector, shooter+hopper, and claw. Each is designed to be easily mounted/unmounted from our chassis. This allowed us to build the robot in separate pieces, and then assemble everything together. In addition, this will also make repairs much faster during competitions - we now have easy access to some of the more hidden parts of the robot.

Refined- While we did not get much time to test our final robot, we are still confident because of the massive prototyping effort we went through. Every variable of the shooter was tested and perfected, every aspect of the collector has been tested to be the best.

Designed with CAD, for laser-cutting - We took our available tools and used them to the fullest. In addition to using nuts and bolts, we also use a tab-and-slot system where beneficial. Because we were designing the entire robot in CAD, we were also able to optimize specific parts to be smaller and closer together, all while allowing us to be confident in the assembly process.

Outreach

With COVID-19, spreading awareness of our robotics team has been much more challenging. However, we have been able to still reach out to engage our community in robotics. On February 10th, we took part in our school's electives fair where we talked to students about our team and the benefits that robotics have had on all of us. We have had several students take interest in the team, and we plan to meet with them after the competition season. We also created a website to spread awareness about our team.

Team 15317 - Error Code 404

Programming:

Overarching Goals:

Our programming team aims to work in harmony with our building team. We aimed to create code that was intuitive, efficient, and, most importantly, easy for us to understand and for drivers to use.

Programming Goals:

Autonomous - Utilize full functionality of the robot in an efficient, reliable program. Create a program that uses sensor inputs and an intuitive, flexible design. Control the robot's driving, lift, claw, shooter, and hopper.

Driver Control Period - Create a program that aids drivers with auto-aiming and position features. Create controls that are comfortable to use and easy to keep track of.

Meetings:

We relied heavily on online meetings, and we chose to use software to aid us with the process of remotely developing our code. We used Android Studio for coding, and we used GitHub to upload and update our code. This allows us to easily view our recent changes, revert changes, and keep track of developments in our code over time. It also allowed us to access each other's edits and improve our code as a team.

📖 **To view our full code for this season, please visit our team's GitHub Repository.**

Code Highlights

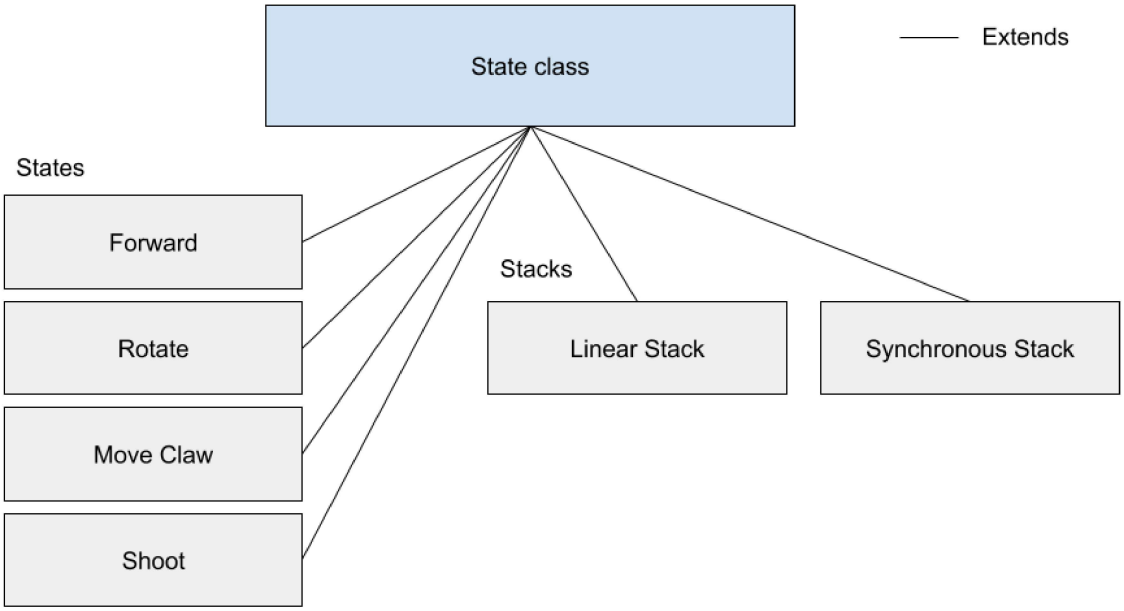
Within this section, we describe the structures and features we have implemented to create a powerful robot capable of as much autonomy as possible. We have split these highlights into two sections. Section I goes over the state engine we use to rapidly prototype autonomous scripts. Section II goes over our utilization of Vuforia and the Field Navigation Images to accurately shoot rings from anywhere on the field.

I. The State Engine

Over the last two seasons, our team's efforts put into coding have increased exponentially. We wanted to create code with longevity; something that our team could build on in future seasons. So, we have implemented large features which aim to make programming easier in the future.

Our autonomous program is made up of a series of states, each with a specific robot function. For example, we have states to move the robot forward and to open the robot's claw. We also have states for more complex routines, such as sensing the number of rings in the starting position.

Instead of moving a mechanism on the robot, waiting, stopping the mechanism, waiting, and so forth, we make a list of actions for the robot to do. This also decreases the risk of the robot becoming stuck in a “while” loop in our code. We created a data structure called a "stack," which allows states to be chained together. A stack is also extended from the "state" class, meaning that stacks themselves can be chained together as well.



States such as the example ones depicted on the left side of the graphic are extended from a main State class. Stack classes are extended from the State class as well, meaning that stacks can be used as states within other stacks.

Making States

Because of the modular design of our state engine, adding new functions to the robot is as easy as creating new states and adding them to a stack. Each state consists of the initialization of a component (1), a statement where the component begins performing a function (2), and a conditional to check when that component’s task is completed (3).

1. Initialize the state.

```
public void init(RobotHardware r) {
  robotHardware = r;
  d = robotHardware.d;
}
```

2. Begin running the state.

```
public void start() {
  d.resetAllEncoders();
  d.runToPosition((float) goal, 0.5);
}
```

3. Stop running when a certain condition is met.

```
public void loop() {
  if(!d.isBusy()) {
    running = false;
    d.setPower(0,0,0,0);
  }
}
```

Our state engine allows us to edit and build our autonomous program in an intuitive and easy way. It also proactively defends against common code errors by avoiding “while” loops and gives us an easy layout of what actions we want our robot to complete during the autonomous period.

Stacks

As mentioned previously, we created a data structure called a "stack," which allows states to be chained together. We have implemented two kinds of stacks: a linear stack and a synchronous stack.

Linear stacks

A linear stack manages the program by initializing each state and moving to the next state when the current one is finished.

This stack is used primarily for autonomous scripting. Creating linear stacks allows us to rapidly create and test autonomous scripts by only changing the order of states in the stack.

```
public LinearStack zerostates = new LinearStack(new OurState[] {
        new ForwardUntil(-6), //backward 6 inches
        new LiftUntilPos("horizontal"), //move lift to set position
        new MoveClaw("close"), //close claw
        new LiftUntilPos("above ground"), //move lift to set position
        new ForwardUntil(-24), //backward 24 inches
});
```

Above: An example linear stack where we grab the wobble goal and move to a different position while holding it.

Synchronous stacks

In a synchronous stack, all of the states in the states list run at once. The stack keeps track of which ones have finished and only moves on when every state is done running.

The synchronous stack has a unique property; when a state inside the stack has finished, it does not only stop running, it is also removed from the stack. This means that we can add new states to the sync. stack after initialization, making the sync. stack perfect for the Driver-Controlled Period.

Using Auto Scripts During Driver Control

```
//this synchronous stack will turn 90 degrees and close the claw at the same time.
OurState[] syncStatesList =  { //here we create a list of states
    new TurnUntilAngle(90),
    new MoveClaw("close")
```

```
};
//here we create the sync stack using the list of states
public SynchronousStack syncstack = new SynchronousStack(syncStatesList);
```

Above: An example of a synchronous stack containing two states. The robot will move to the next step after this once both turning and claw movement have finished.

The unique properties of a synchronous stack allow us to easily implement pre-scripted movements during the Driver Controlled Period. With the push of a button, the drivers are able to start a sync. stack that controls the robot autonomously. We currently use this to precisely move the robot to accurately hit the power shot targets.

```
OurState[] syncStatesList =  {
    new AlwaysRunning(),
};
SynchronousStack states = new SynchronousStack(syncStatesList);
```

Above: Initializing the synchronous stack used to perform autonomous scripts during the Driver-Controlled Period.

```
OurState[] s = { //initializing a table of states. this will be passed into the sync. stack.
    new LinearStack(new OurState[]{ //we want a specific order of operations, so a linear stack is used.
        new TurnUntilAngle(heading),
        new StrafeUntil(-y),
        new ForwardUntil(-x),
    }),
};
states.addState(s); //addState is a method of Sync. Stack which adds the table of states to the stack.
```

Above: When the B button is pressed, as long as there is a Navigation Image within sight, an x, y, and heading value is calculated. Those values are how the robot needs to move in order to be lined up to shoot the power shots. This code segment is where those values are passed into a Linear Stack. This Linear Stack is then passed into the Driver Control Period's Synchronous stack.
This code segment shows how states can be queued up during the Driver Control Period, as well as how stacks can function within other stacks.

II. Vuforia & Automation During Driver Control

We incorporated Vuforia into our program for the first time this year, and we have found it to be a major tool for tracking our robot's position on the field during TeleOp. This helps us add other intuitive features to our TeleOp programming.

One way we use Vuforia in our TeleOp program is our auto-aim feature. With a press of a button, the robot can (1) accurately turn to face the goal and (2) change the angle and power of its shooter to aim for the highest goal - all from anywhere on the field.

To make the robot turn to face the goal, we first use the heading of our robot returned by Vuforia to make the robot turn to face the wall.

However, to make the robot turn to face the actual goal, we take our use of Vuforia a step further.

Using its displacement from the center of the robot and preset locations of each wall target, Vuforia can calculate the x-position and y-position of the robot based on its perception of each wall target. We also enter the location of the base of the goal into the code.

```
heading = -90 - cam2.getHeading(); //COLLECTOR-SIDE
x = 72 - cam2.getPositionX(); //(72, 36) is the Vuforia coordinate of the blue goal in inches
y = 36 - cam2.getPositionY();
theta = (float) (Math.atan2(y, x) * (180/Math.PI));

if (theta >= 0) {
  float rotationAngle = (heading + (theta*1.2f));
  d.rotateToAngle(rotationAngle, -0.5); //counter-clockwise rotation
} else if (theta < 0) {
  float rotationAngle = (heading + theta);
  d.rotateToAngle(360-rotationAngle - 2, 0.5); // clockwise rotation
}
```

Using all of this data, both from our prepared measurements and Vuforia, we can locate both the robot and the goal as points on the field. This means that, using our code, we can find the angle we need to turn to in order to face the goal and rotate our robot to that angle in the most efficient way possible

Using Vuforia to calculate the necessary incline of our shooter is similar. Our robot features a shooter whose angle can pivot upwards and downwards using servos. This allows us to shoot from a wider range of locations on the field. Vuforia helps us find the distance from the robot to the base of the goal. This is necessary in order to calculate how high the shooter needs to be angled; in our case, the higher the angle of the shooter, the farther the rings travel.

Using the displacement of the camera from the center of the robot, we can again find the x-position and y-position of our robot on the field using Vuforia outputs. We then use these to calculate the distance from the robot to the base of the goal. Since we already know the location of the goal on the x-y coordinate plane (the field), it is basically the Pythagorean Theorem.

```
if(cam2.isTargetVisible()) {
                    double x = 72 - cam2.getPositionX() + 1.125;
                    double y = 36 - cam2.getPositionY();
                    distToGoal = Math.sqrt((x * x) + (y * y));
```

Using the displacement of the camera from the center of the robot, we can again find the x-position and y-position of our robot on the field using Vuforia outputs. We then use these to calculate the distance from the robot to the base of the goal. Since we already know the location of the goal on the x-y coordinate plane (the field), it is basically the Pythagorean Theorem.

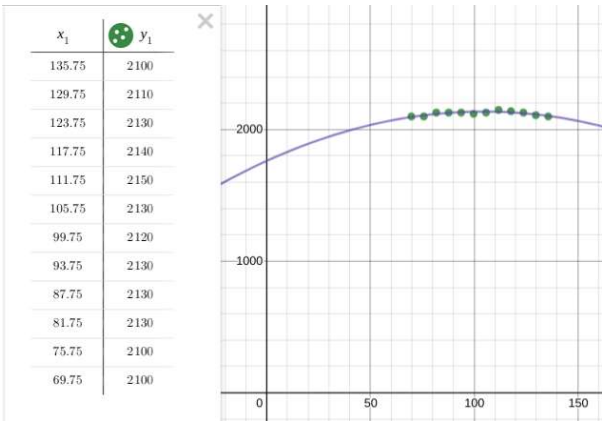
Once we had the distance of our robot to the goal, we relied on data we gathered ourselves. We tested which shooter angle made rings reach the goal at a given distance from the goal while ensuring that the rings did not have enough power or height to travel farther than 16 feet. By testing several points, we could interpolate values in between these points with a satisfying degree of accuracy.

Distance x Velocity	
Distance from pivot (bumper + 9.75")	**Graphed velocity so rings <= 16 feet**
135.75"	2100
129.75"	2110
123.75"	2130
117.75"	2140
111.75"	2150
105.75"	2130
99.75"	2120
93.75"	2130
87.75"	2130
81.75"	2130
75.75"	2100
69.75"	2100

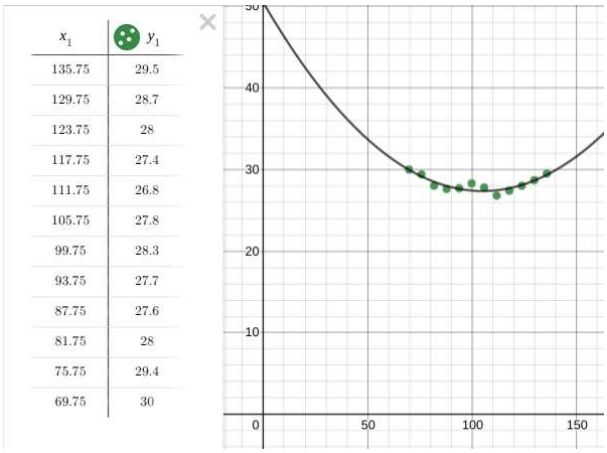
Distance x Angle	
Distance from Goal	**Graphed Angle so rings <= 16 feet**
135.75"	29.5
129.75"	28.7
123.75"	28
117.75"	27.4
111.75"	26.8
105.75"	27.8
99.75"	28.3
93.75"	27.7
87.75"	27.6
81.75"	28
75.75"	29.4
69.75"	30

In order to use our points to find a general rule for calculating our shooter angle for any given distance, we graphed our points using an online graphic calculator. We used separate functions for the "Distance x Shooter Angle" and "Distance x Shooter Power" graphs.

Distance x Power Graph

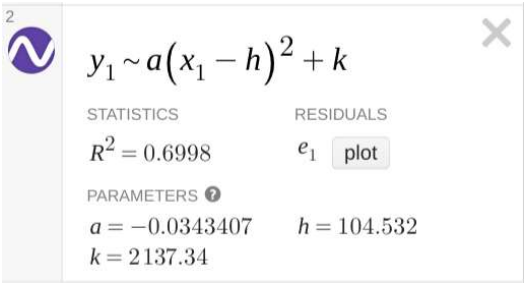


Distance x Angle Graph

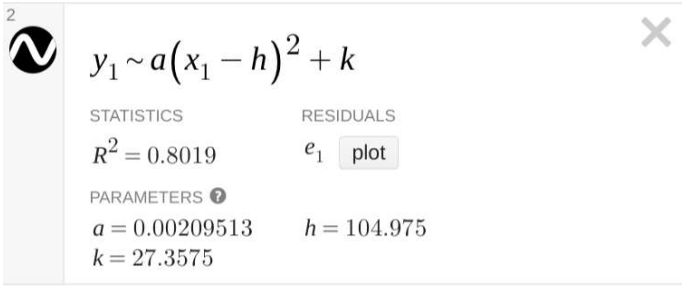


This allowed us to find lines of best fit for both functions, which allowed us to interpolate and extrapolate values in between and around our data set. We chose to find quadratic lines of best fit because they fit the subtle curve of our points.

Distance x Power Equation



Distance x Power Equation



Once we found the coefficients and other added values of these lines of best fit, we could incorporate it into our code. This allows us to find the correct angle and power for our shooter for any given distance between our robot and the goal, ensuring accuracy and a safe ring trajectory.

```
shooterAngle = (0.00209513)*(distToGoal-104.975)*(distToGoal-104.975) + 27.3575; //quadratic line of best fit equation

shooterSpeed = (-0.0343407)*(distToGoal-104.532)*(distToGoal-104.532) + 2137.34; //quadratic line of best fit equation
```

The final way we use Vuforia in our code is our auto-positing function for shooting the power shot during the end game. Using the same principles above, our camera finds our robot's coordinates and heading. Using the heading, it turns to face the field wall with the power shots.

Then, we use the coordinates of the robot to move to a specific spot on the field based on the Vuforia coordinate plane in inches: (-39, 17).

```
x = -39 - cam2.getPositionX();
y = 17 - cam2.getPositionY();

shooterAngle = 30;
shooterSpeed = 1865;

state = "rotate";
OurState[] s = {
  new LinearStack(new OurState[]{
    new TurnUntilAngle(-heading),
    new StrafeUntil(-y),
    new ForwardUntil(-x),
}),
```

Finally, we set the shooter angle and speed such that rings will successfully hit the power shot from this distance. This function can be used accurately from anywhere on the field where our camera sees Vuforia target. It is also one of the major ways we use our synchronous state engine in our Teleop program.