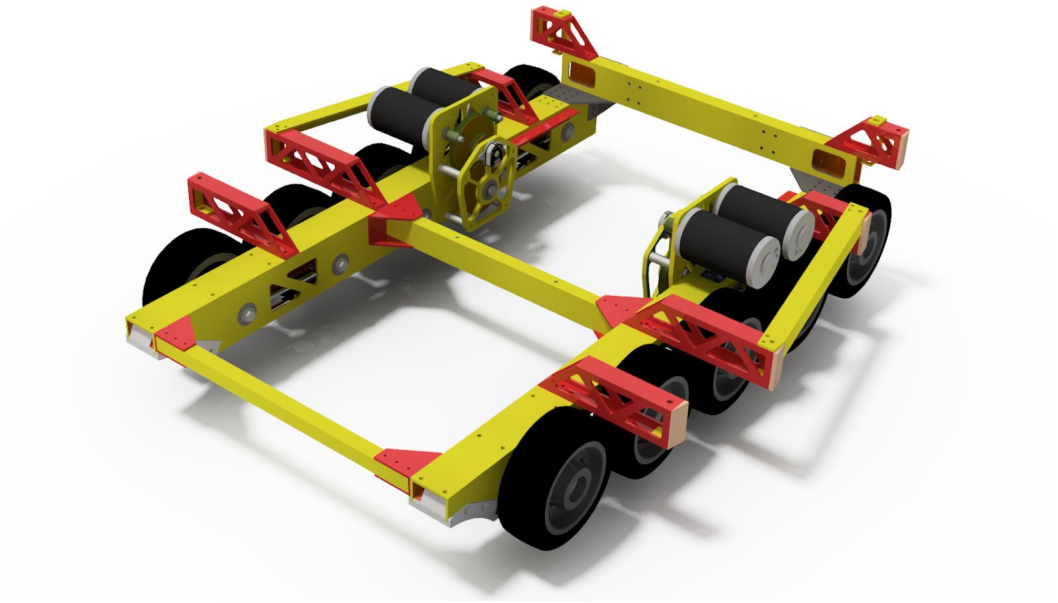
Team 846, The Funky Monkeys

Introducing our 2016 robot: Monkey Python

Drivetrain

Amrita Iyer (Senior), Ria Pradeep (Junior)



Design Requirements:

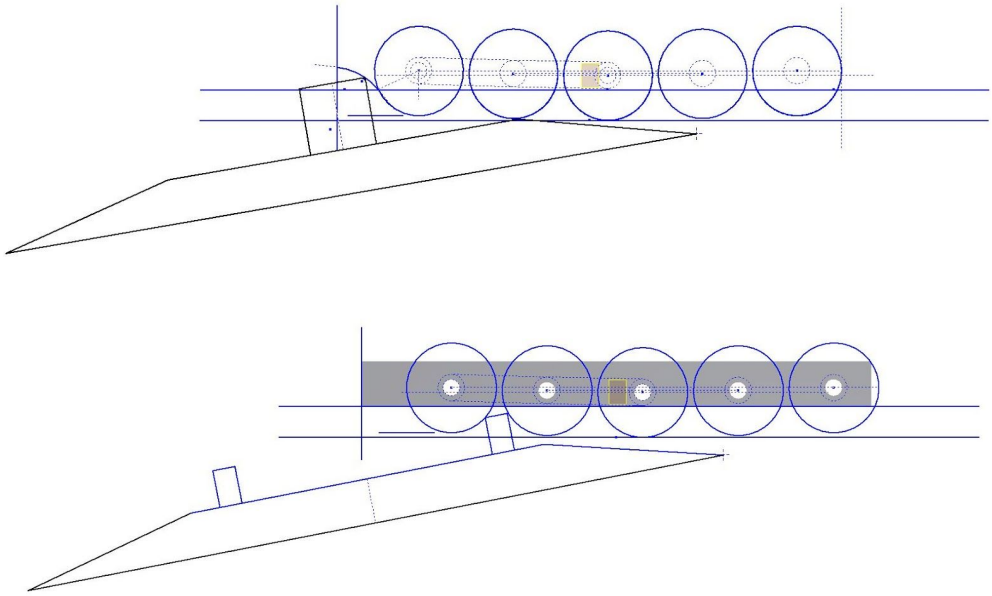
As soon as the game was released, we knew our drivetrain would need to do more than be able to move around the field. We compiled a list of design requirements for this subsystem.

1. *Defeat the four terrain defenses (rock wall, rough terrain, ramparts, moat)*
2. *Clearance and torque to go over rock wall*
3. *Not catch on any defenses*
4. *Maintain a constant line of contact with the ground*

Drivetrain

In order to compete in this year's game, it was essential that our drivetrain be able to to cross all of the defenses. While brainstorming, we came up with three possible wheel configurations that would meet our requirements: 6 8" wheels, 12 4" wheels, and 10 6" wheels.

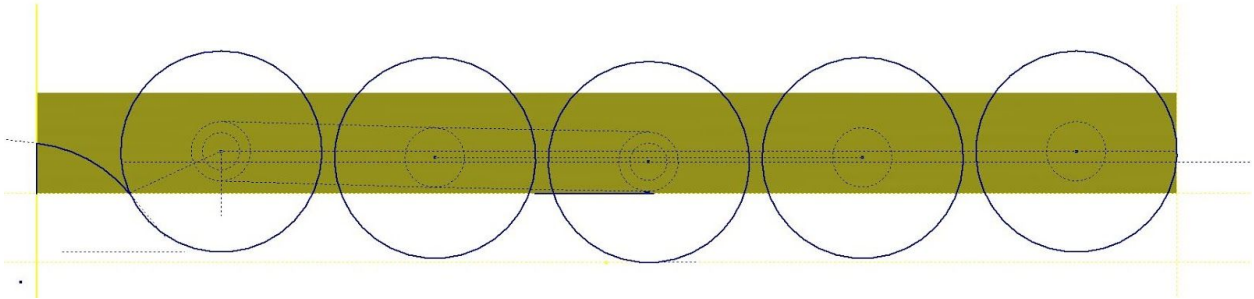
To test each of these configurations without having to prototype the layout, we created a 2D CAD sketch of each setup driving over the rock wall, rampart, and moat defenses.



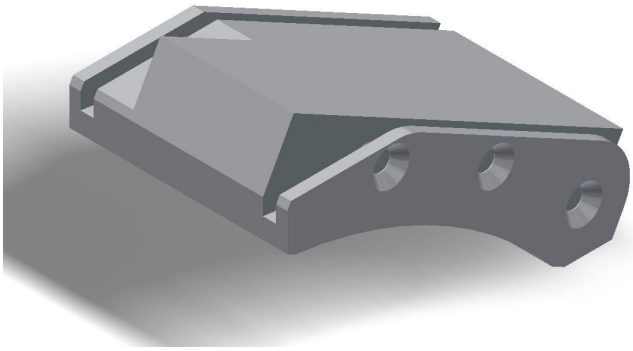
Rock Wall and Moat verification sketches

We quickly found that the ten six-inch wheel design worked best with the defenses, since this was the only configuration in which the wheels were spaced so that the robot would not straddle the two metal bars of the moat at the same time.

We soon realized that this ten six-inch wheel configuration would not easily make it over the rock wall since the highest point of the 4.5" tall bar fell above the midline of the 6" wheel. To facilitate the robot's climb, we wanted to manufacture some type of angle or curve out of the front of the frame members. We decided to manufacture a curve rather than a slanted line since the curve had the advantage of incrementing the force of the impact of the robot against the rock wall (unlike the straight angle which would absorb the full impact at one moment).



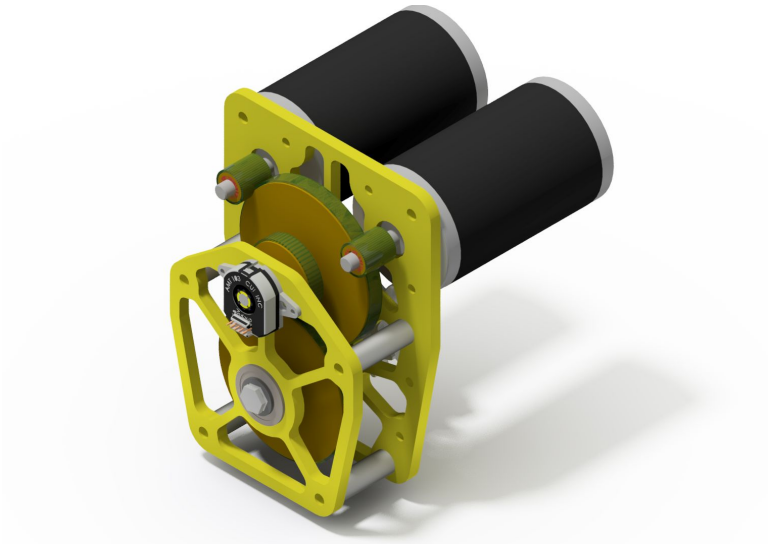
Sketch of wheel geometry and ramp



Delrin ramp insert

Finally, in order to not only save space inside the robot but also protect the drive chains from getting caught on defenses as we drove over them, we decided to place the chain assemblies inside the side frame member tubes. This was our first time experimenting with such placement, and our concerns included limited access to the chains themselves as well our limited ability to adjust the chain tensioning easily. After repeated prototyping with quantized spacing for the center to center sprocket distances, we were able to find the optimal wheel layout, and smoothly ran our first set of wheels!

Drive Gearbox



We knew that a primary requirement of the drivetrain was to go over the defenses, so we needed to make sure that our gearbox provided enough torque to do so. We first imagined our robot in a worst-case scenario: on flat ground attempting to mount a straight wall. From this theoretical position, we calculated the torque required to go up

that wall, and were able to verify that the gear reduction we selected provided at least that amount of torque.

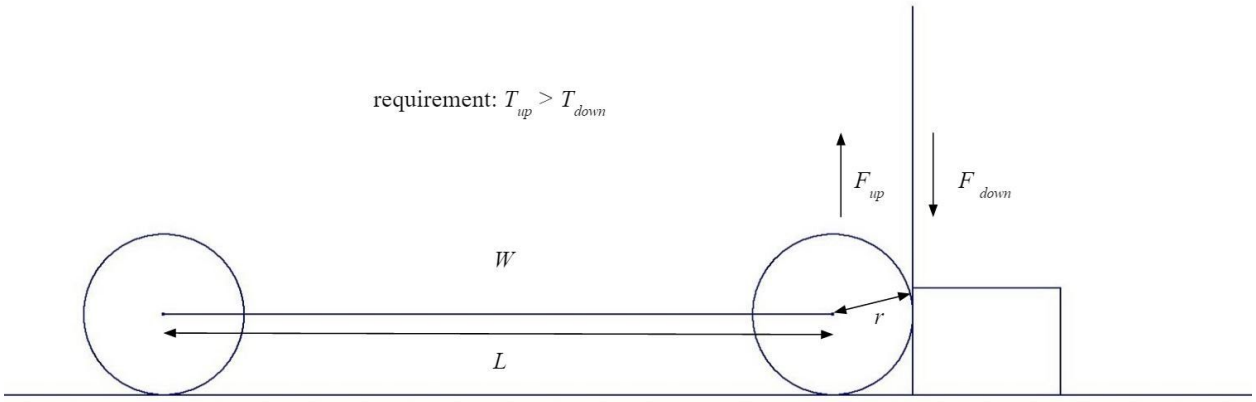
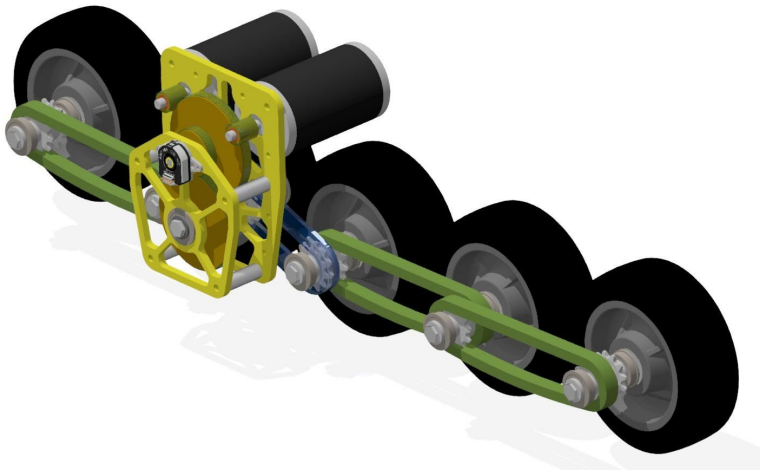


Diagram used to calculate required torque

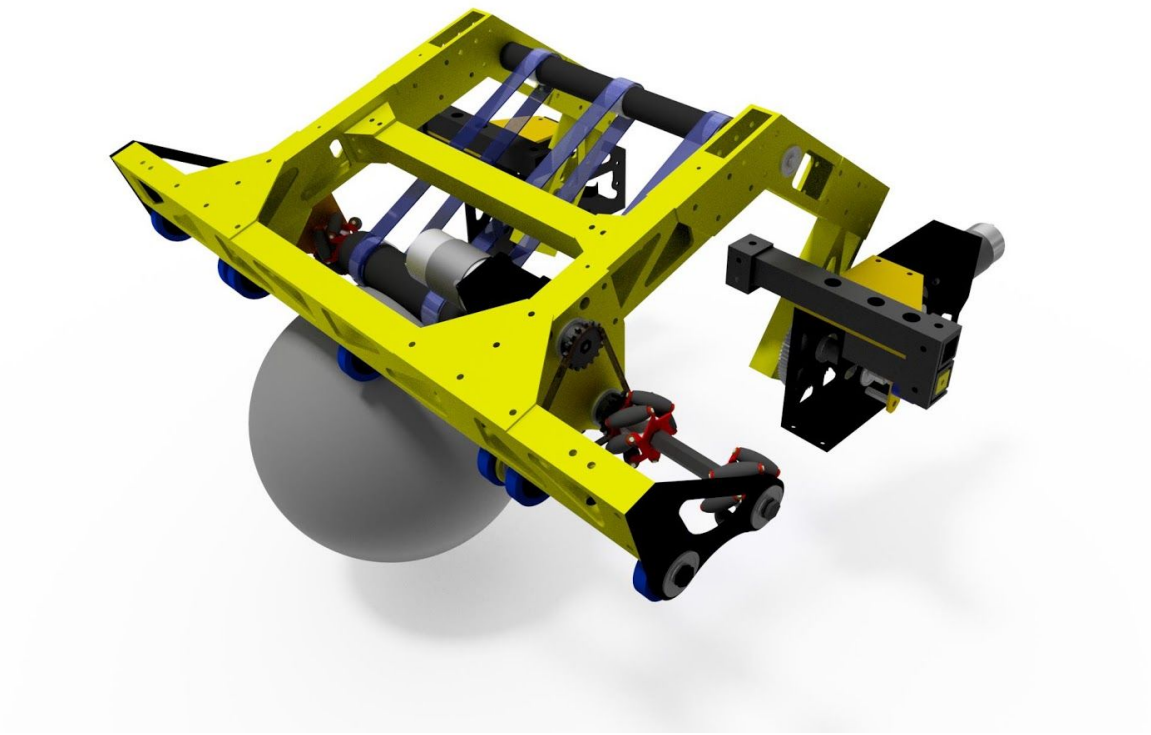
For the past two years, we have designed drive gearboxes with motors that hang over the wheels and thereby maximize space inside the robot. In order to continue with that feature this year, we lifted the output axis of the gearbox over the axis of the center wheels. Instead of having a gearbox that directly powered one wheel, we designed a gearbox that powers a single elevated shaft, which then drives the rest of the wheels with linked chains.



Model of wheels and gearbox with chains

Intake

Srinjoy Majumdar (Senior), Arthur Zhang (Sophomore), Andrew Ng (Freshman)



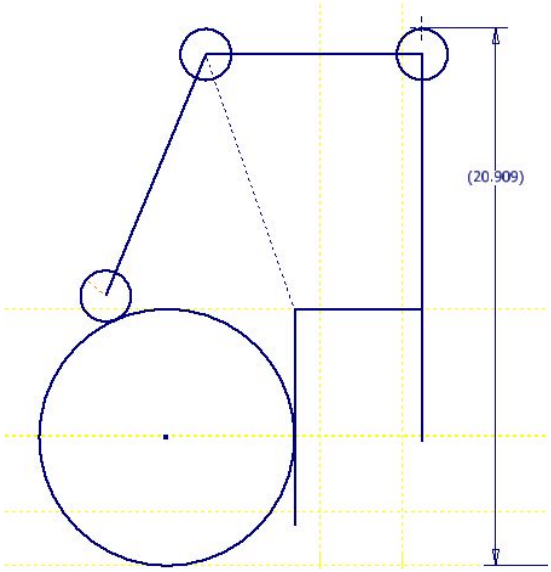
Design Goals

Our robot intake is a counterbalanced arm that can collect boulders from any direction and defeat the Portcullis and Cheval de Frise defenses.

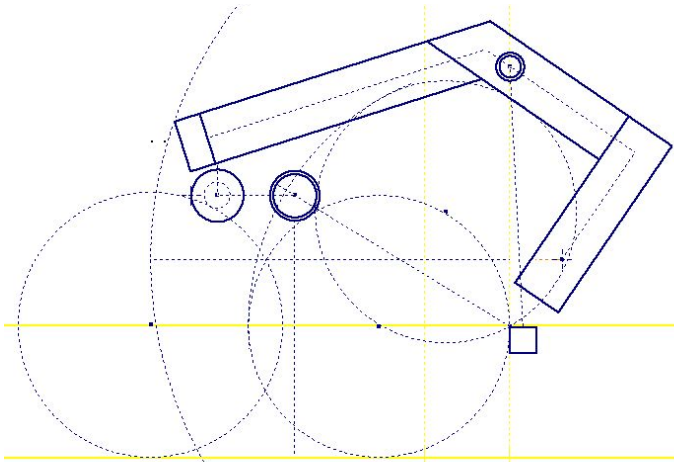
We set several requirements for the intake subsystem early in the season:

1. *Fit under 15 inches outside of robot for low bar*
2. *Gain control of ball quickly*
3. *Wide operating range (full width of robot)*
4. *Score Low Goal*
5. *Pass ball to shooter*
6. *Use minimal volume inside robot during operation*
7. *Function as spacer when shooting*
8. *Defeat Cheval de Frise and Portcullis*

We considered intaking the boulders both over and through the front bumpers. After analyzing the geometry for both systems, we realized that an over-the-bumper intake would not have a position extending under 15 inches outside the robot, thus not meeting our first requirement. However, the through bumper design required us to split our drivetrain frame, making it weaker. So we combined the two designs and decided to intake over our frame while still splitting the bumpers. As a result, we maintained a position under 15 inches and a rigid drive base.



Over Bumper Intake Design



Through Bumper Intake Design

Collecting

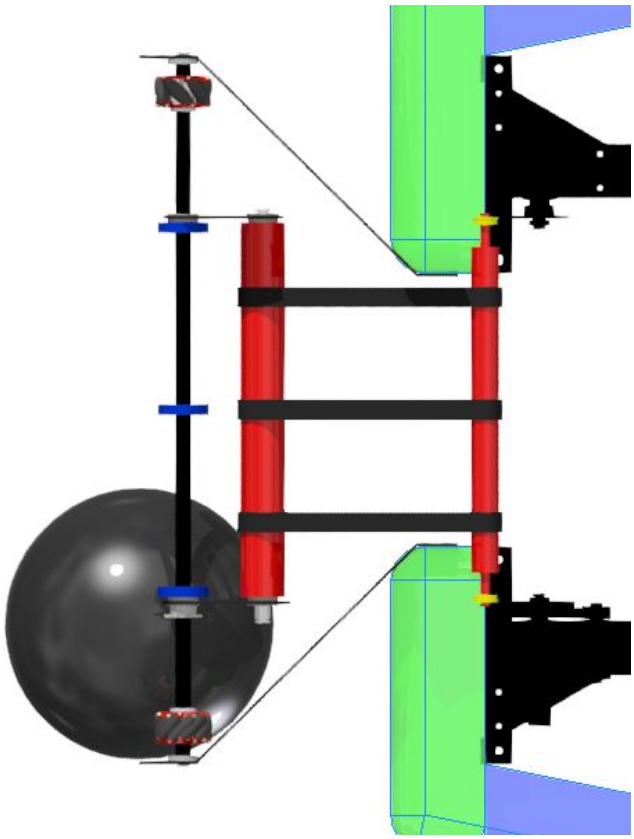
Why Mecanum Wheels?

After successfully using mecanums on our 2014 intake to funnel the balls into the center of the robot, we decided to once again use mecanums on our intake this year to funnel in the boulders. Initially, we mounted these wheels on our intake in combination with a polycarb funnel to pull in boulders over a wider range.

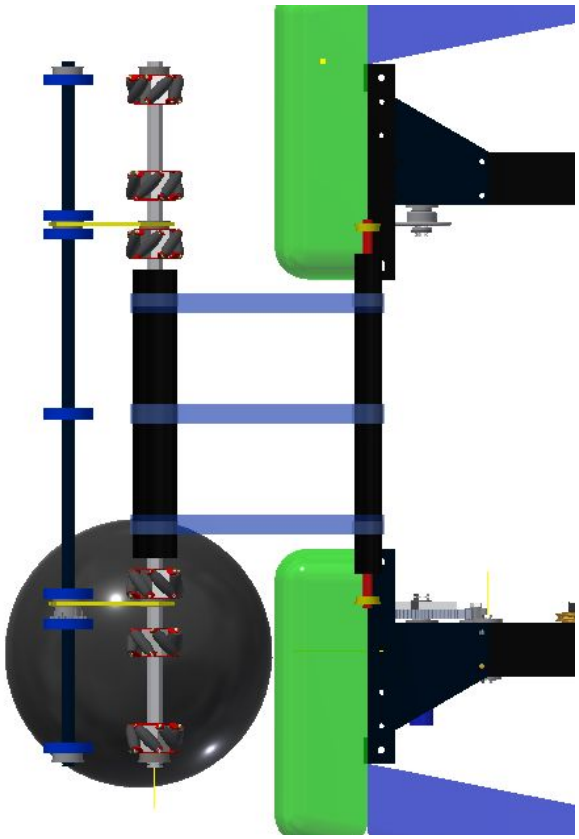
After running several tests on our intake, we found that the mecanum wheels were not effective in funneling the boulders if they were not positioned against a rigid surface. The intake ended up jamming the boulder on the sides of the polycarb funnel.

To solve this problem, we moved our mecanum wheels from the outer shaft to the inner shaft so that the boulders would be in contact with the bumpers and the mecanum

wheels when collecting from the side. With this new design, we maintained both the extended reach and the ability to collect from the corners of the robot.

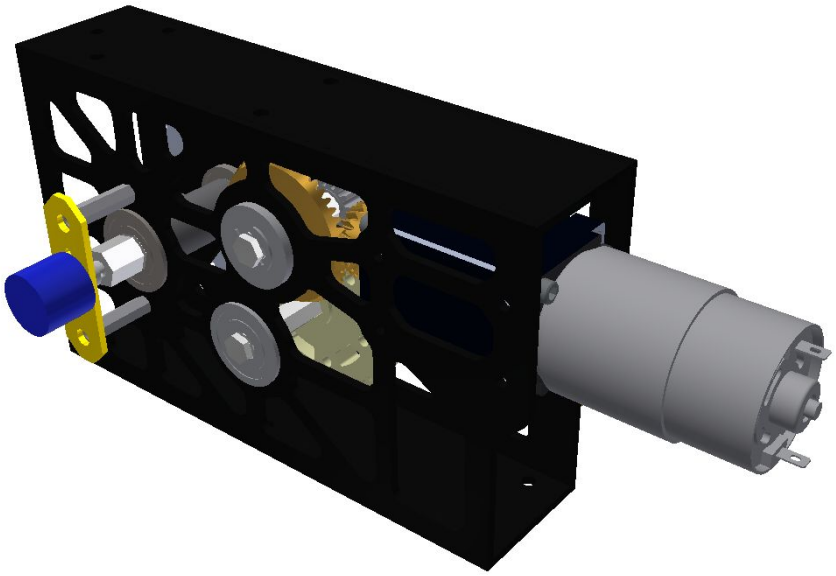


Old Design with Mecanum on outside with Funnel



Improved Design with Mecanums on Inner Shaft

Intake Arm Gearbox



To defeat the defenses, we needed to control the intake's speed and position at multiple points, which would have been hard to achieve with a pneumatic cylinder. As a result, we decided to design our own gearbox to move the intake arm. Initially, we tried a design with the motor mounted inline with the intake arm's axis of rotation. However, we found that with this configuration the motor did not fit inside our frame perimeter. We therefore decided to use bevel gears to mount the motor perpendicular to the arm's axis of rotation which would keep it from interfering with the bumpers.

We stripped the bevel gears on the intake arm gearbox early in our testing when the software team ran the intake past its soft limit. Before replacing the gears, we decided to analyze the strength of the gears to ensure that they would not fail under normal operation. We were surprised to discover that the brass material on the bevel gears was actually weaker than the surface treated 7075 Aluminum spur gears. Additionally, the bevel gears we were using had a bigger diametral pitch and smaller face width, making them even weaker.

$$W = \frac{SFY}{P}(.75)$$

W = Tooth Load Along Pitch Line (lbs)
S = Safe Material Stress (from "Gear Theory")
F = Face Width
Y = Tooth Form Factor (from "Gear Theory")
P = Diametral Pitch
V = Pitch Line Velocity (fpm)

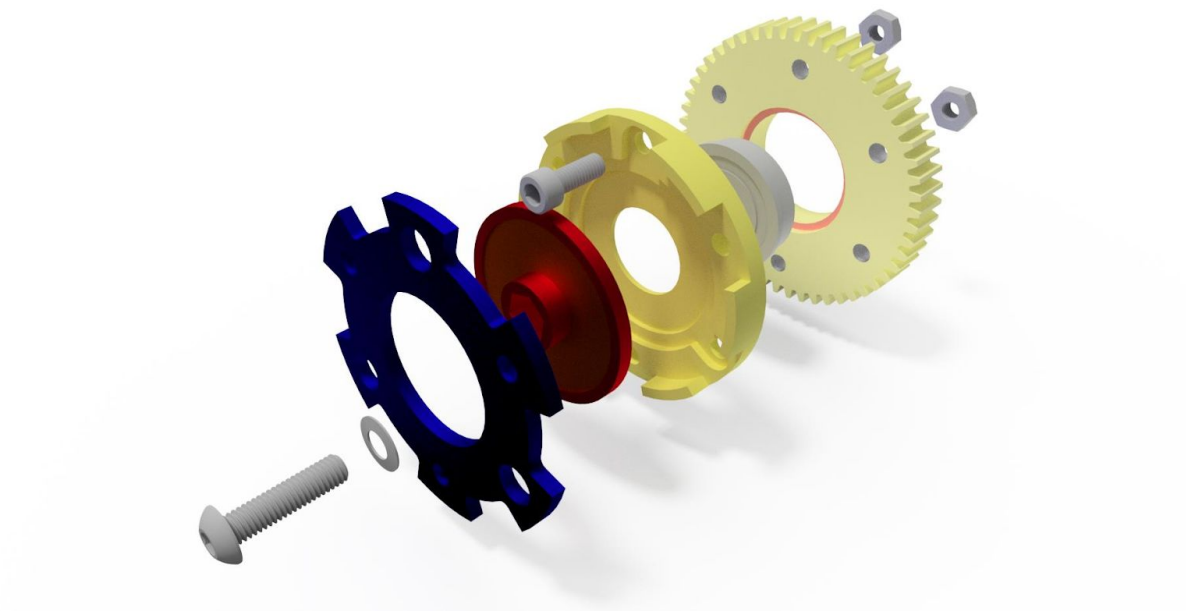
Source: Boston Gear

	Aluminum Spur Gear	***Brass Bevel Gear***
Yield Strength (PSI)	*73000*	*29000*
Face Width (in)	*.375*	*.3125*
Diametral Pitch (T/in)	*20*	*24*
Max Tooth Force (lb)	***164.25***	***34.74***

As the bevel gears were almost 5 times weaker than the spur gears, we decided to add an additional spur gear stage in the gearbox before the bevel gears so that the loads on the bevel gear would lessened.

The intake arm is also neutrally balanced at all points of its travel using two surgical tubing springs. These springs allow the intake to stay at any position without any motor power, reducing the load on the gearbox.

Friction Clutch



To prevent the gears from breaking at all, we calculated the max tooth force from the gearbox to design a clutch that would slip when the torque on the gearbox got close to breaking the teeth.

$$A = \pi(r_2^2 - r_1^2) \qquad P = F/A \qquad dF = P\,2\pi r\,dr \qquad dT = r \times dF$$

Using these equations, we were able to derive the following equation:

$$T = 2 \times \int_{r_1}^{r_2} P\,2\pi r^2\,dr$$

We had to make sure to multiply the frictional force by 2 in order to account for the fact that there is contact on both sides of the friction plate. Then we simplified the equation.

$$T = \frac{4}{3} F \frac{r_2^3 - r_1^3}{r_2^2 - r_1^2}$$

From that, we realized that having friction in an area of contact at the center of the friction plate would be inefficient, so we made the area of contact of the frictional force as close to the edge of the friction plate as possible.

We calculated the amount of pressure we would need to sandwich the friction plate in order for the friction plate to slip at a torque that is less than the torque at which the spur gear would break.

$$F_f = \mu N$$

The frictional force of the clutch is the product of the friction coefficient of steel on steel and the normal force, we changed the normal force until we got the frictional force desired.

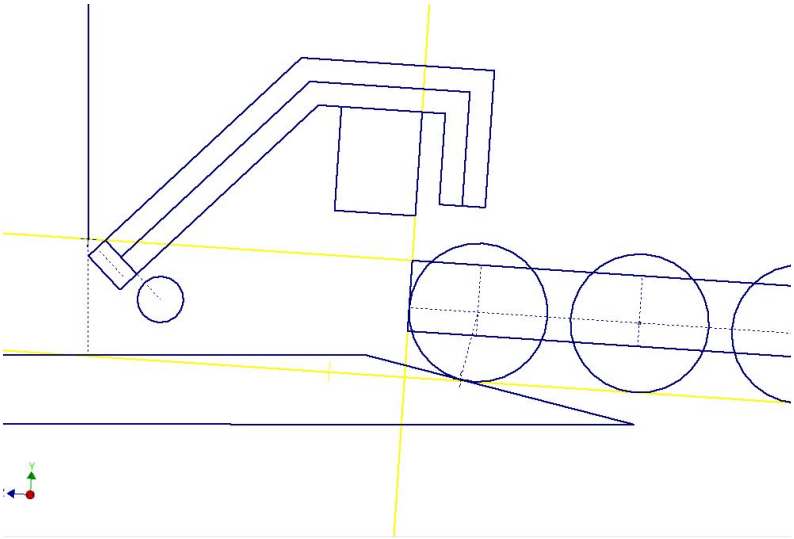
In order to achieve that force, we decided to use disc springs underneath screws that go through the entire clutch because of how compact they are. With screws for tensioning, we were able to control the pinch force by tightening and loosening the screws, thus allowing us to adjust the frictional force.

Defenses

Portcullis and Cheval de Frise

We designed the intake to defeat both the Portcullis and Cheval de Frise defenses. Since the opening under the portcullis door is only 4 in, we designed the arm in a wedge shape so we could open the portcullis door easily. As we move through the defense the arm lifts up and shifts the portcullis door from on top of the intake to the top of the shooter smoothly. The arm is also used to push down the planks on the Cheval de

Frise. We approach the defense with our intake up and rotate it down once we're above the planks and drive forward.



Intake under Portcullis

Shooter

Owen Li (Senior), Jing-Chen Peng (Sophomore), James Jiao (Freshman)

The shooter is a flywheel mounted on a counterbalanced arm that is capable of shooting boulders into the high and low goals from the batter.

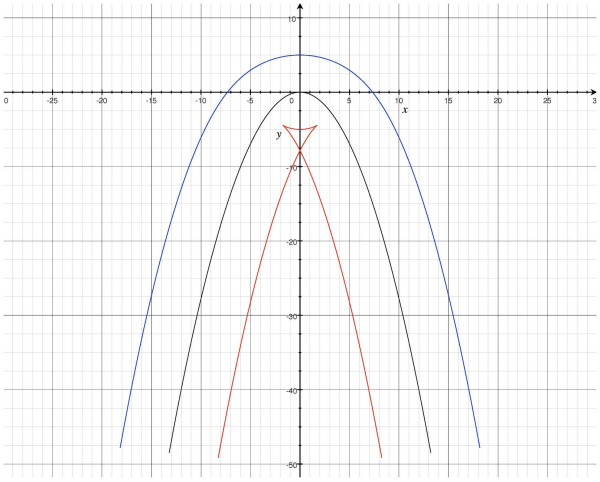
During our initial strategy discussions for FIRST Stronghold, we quickly identified the ability to shoot as vital to ensuring both a win and an extra ranking point. Nevertheless, shooting would be ineffective unless we could guarantee an accurate and consistent shot.

Analyzing the Shot

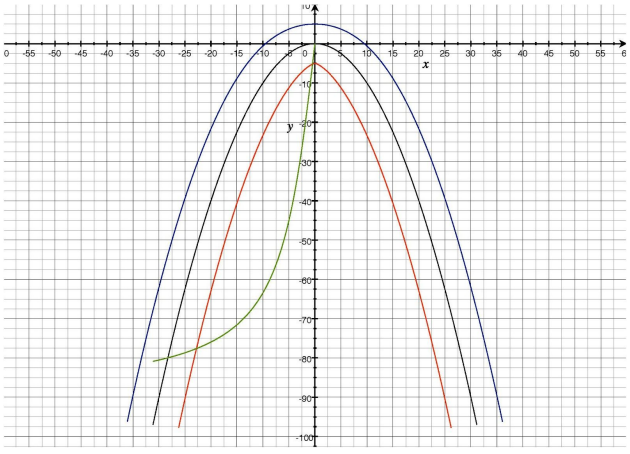
After rigorous initial brainstorming sessions, we decided to design a 1-roller flywheel for the following reasons:

- *This configuration allowed us to fold everything in a compact space.*
- *Backspin imparted on the ball, when hitting the top of the goal, would spin the ball into the goal.*
- *Backspin carries the ball forward and adds lift due to the Magnus Effect, even though those effects might be negligible.*
- *Momentum transfer from the flywheel to the ball eliminates the need for heavy motors and a large squeeze on the ball. (We used RS550's to drive our flywheel).*
- *Momentum transfer is much more reliable than attempting to consistently spin up to a certain speed.*

Now that we had decided on the flywheel, we had a lot of other things to put into consideration, the first of which was the ball's trajectory. We first calculated how much energy would be needed to just barely reach the goal. We quickly realized that there was a problem: some edge of the ball would undoubtedly hit the wall first, even if the center of the ball was on the trajectory to clear it. To completely understand this phenomenon, we drew out the entire ball projection as it passes through the goal:



(The "delta" in the red curve above is where bottom of the ball hits the wall, even when the center of the ball should be on a trajectory through the goal.) We then derived an equation for the shot such that, with that x-velocity, the three points would coalesce, which solved our main problem.



x-axis: distance from goal
y-axis: shot height from under goal
y-axis (for green): shot angle (negative to fit in the same quadrant)

With this information, we determined that any shot from behind the black curve was an optimum shot.

Shooter Arms and Crank Gearbox

The final model of the shooter arms and crank gearbox, with counterbalancing.

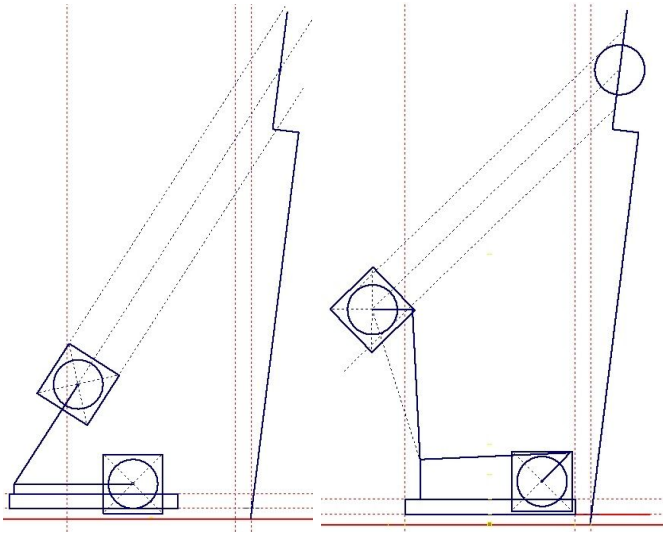
We designed the arms and crank gearbox to raise the main shooter assembly into our predetermined shooting position. This was accomplished by using a four-bar linkage to hold the shooter head steady while raising the entire apparatus with a crank.

The shooter arms and gearbox were created with the following requirements in mind:

1. *Fold into the robot to clear the low bar.*
2. *Provide the shooter head with an optimal shooting angle.*
3. *Minimize backlash to provide a reliable shot.*

Iteration

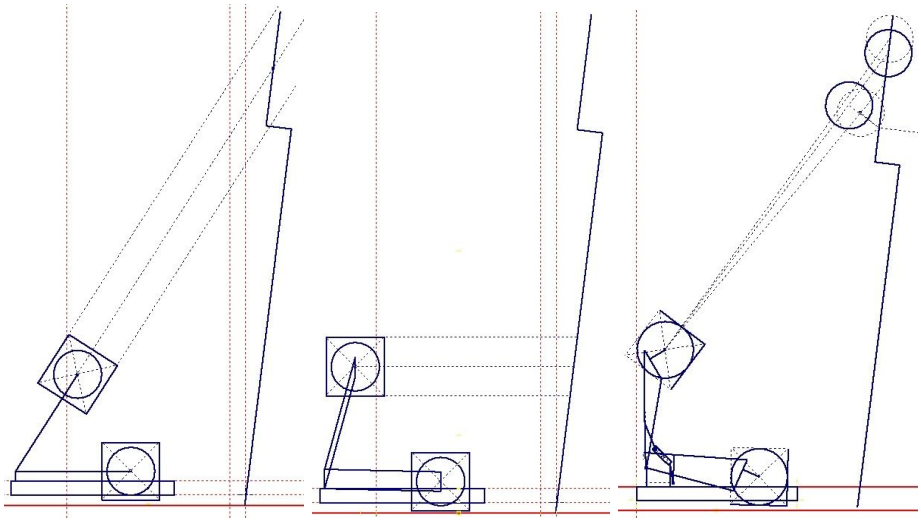
To ensure that we had the most favorable firing angle possible, we tested many different types of lever linkages to determine the ideal system. Initially, we went after a simple one-arm rising mechanism.



Comparison between the geometry of a simple, one-armed shooter and a two-armed shooter.

Unfortunately, we discovered that only having one arm was incapable of supplying the angle that we desired; as such, we turned to using chains and sprockets to rotate the shooter head. Unfortunately, that design stuck out the back of our robot, and introduced an amount of mechanical backlash that led us to look at other options.

We arrived at the idea of a four-bar linkage by adapting several key portions of the chain and sprockets design. We realized that by creating a difference in the lengths of the two sides of a four-bar linkage, we could create a linkage that rotated the shooter head as the arms lifted up, achieving the optimal shooting angle.

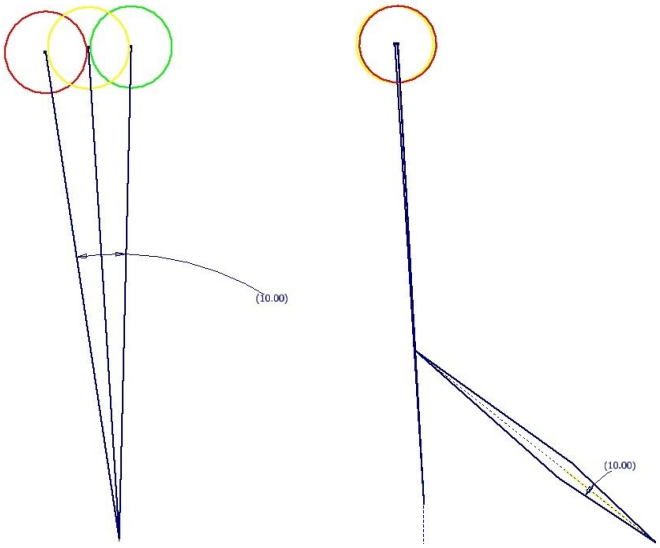


Comparison between the possible shooting angles with a one-bar shooter, a four-bar parallel shooter and the four-bar unparallel design.

Crank Design

Our shooter is raised and lowered by an assembly of cranks connected to a motor. We chose this style over other possible solutions, such as using pneumatic cylinders, for three main reasons: it could hold its position at the end of travel; it minimized backlash while we were shooting; and it allowed us to set multiple positions for our shooter.

We initially wanted to drive the shooter arms at their pivot point in order to avoid an over-complicated design. However, we recognized that this option would force our motor to hold the shooter arms up, which would waste power. We fixed this by placing the three pivot points of the crank arm collinear to each other when the shooter was at its highest point, causing any force exerted by the weight of the shooter on the cranks to be passed directly to the supporting axle, which would lock the assembly in place. It also reduces backlash from the motor.



Comparison between backlash caused by two different methods for lifting the shooter. On the left is a design driven from its pivot point; on the right is a design driven by a crank. The yellow circle shows the desired position, and the green and red circles show the positions five degrees away from the desired position.

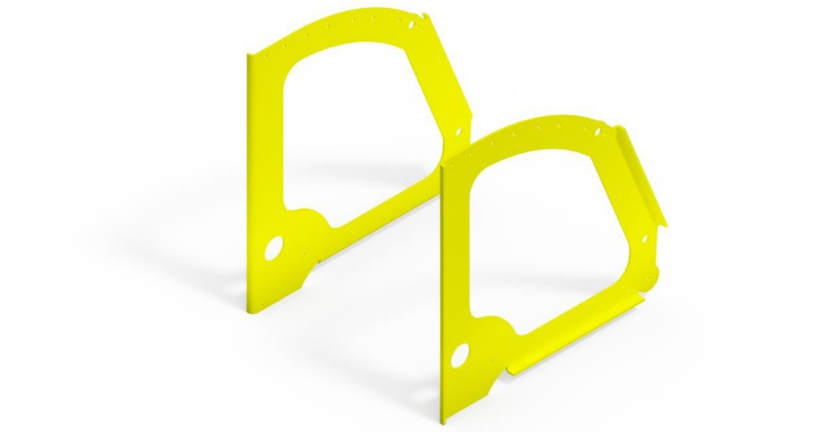
Shooter Head

The shooter head is composed of two side plates, a polycarbonate sheet hood, a flywheel, and a trigger roller.

The shooter head, which is mounted on the end of the shooter arms, allows our robot to quickly acquire and launch the boulder. It is able to accept the boulder from the collector, purge back into the collector, and shoot into the high goal. After the boulder is collected and enters the shooter, it is secured by a trigger roller and several cross-bars, beyond the reach of the flywheel. When we want to shoot the boulder, the robot drives up to the tower, raises its arms, and activates the flywheel in the shooter head in preparation of launching the boulder into the high goal. The trigger roller nudges the boulder in between the flywheel and the polycarbonate hood once the flywheel reaches a sufficient speed, propelling the boulder into the tower goal.

Side Plates

While we were planning the different aspects of our robot, we recognized that our first priority in designing the shooter head was modeling the basic layout of the side plates. The biggest challenge was fitting all these features in a self-imposed 11”x12”x12” to fit between the collector.

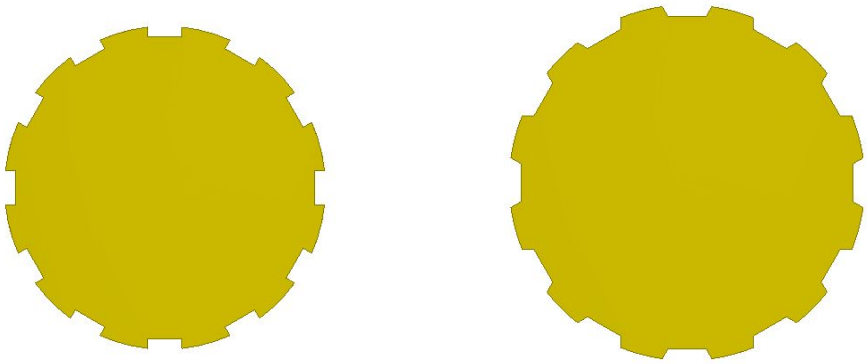


The side plates sandwich the flywheel, the trigger roller, and the polycarbonate hood.

Designing the Flywheel

By deciding to use a flywheel to create a momentum transfer from the roller to the ball instead of using the sheer speed of a motor to shoot the ball, we were able to use a smaller motor, which greatly reduced the weight that we were holding at the tip of our arm. After settling on an aluminum flywheel, we had to come up with a way to grip the ball.

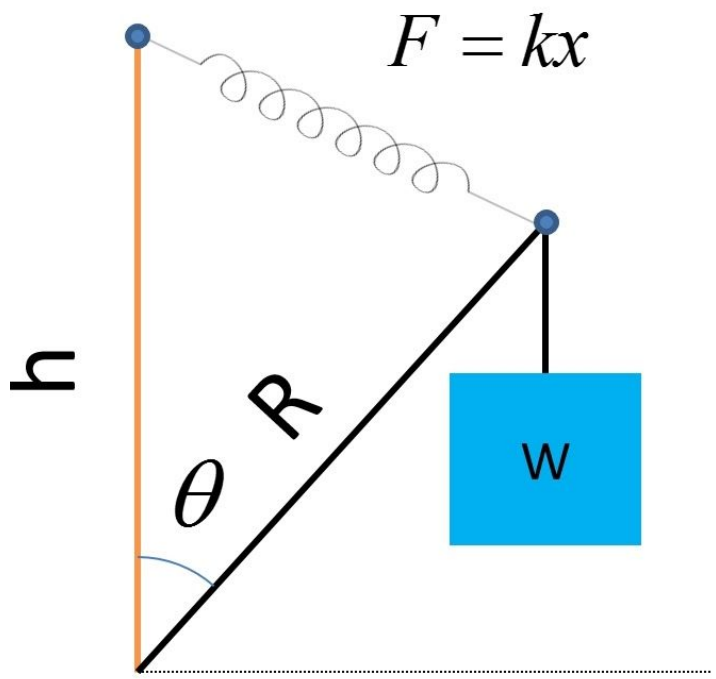
After researching methods to increase friction between the ball and flywheel, we decided to cut grooves in the flywheel and stretch surgical tubing over it. We then made some sketches in Inventor, and found something troublesome: with just the straight cut, the angles on the slots were angled too sharply, and would tear up the balls. To fix this, at each point, we milled three times: once for the main slot and twice offset from the center for the slots adjacent.



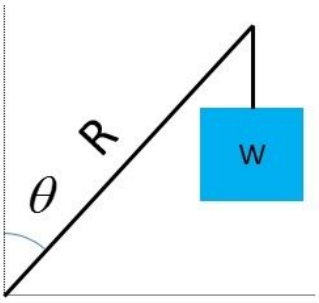
Before vs. After Chamfering sides to reduce undercut.

Counterbalancing

When designing the shooter and collector, we decided to incorporate counterbalancing into these subsystems to reduce load on the gearboxes driving these subsystems and make software control easier.



The problem: Counterbalancing a weight on an arm.
As shown to the right, an ideal spring attached to the end of an arm and a mount point above the arm's pivot point will perfectly counterbalance a weight on the end of the arm as long as the weight is equal to the spring constant of the spring multiplied by the distance between the arm's pivot point and the spring's mount point.



$$T_{weight} = \vec{R} \times \vec{w} = |R||w| \sin\theta$$

$$\vec{x} = \vec{h} - \vec{R}$$

$$T_{spring} = \vec{R} \times \vec{F}$$

$$= \vec{R} \times k(\vec{h} - \vec{R})$$

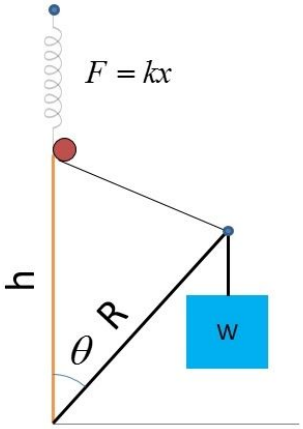
$$= k(\vec{R} \times \vec{h}) - k(\vec{R} \times \vec{R})$$

$$= k(\vec{R} \times \vec{h}) - 0$$

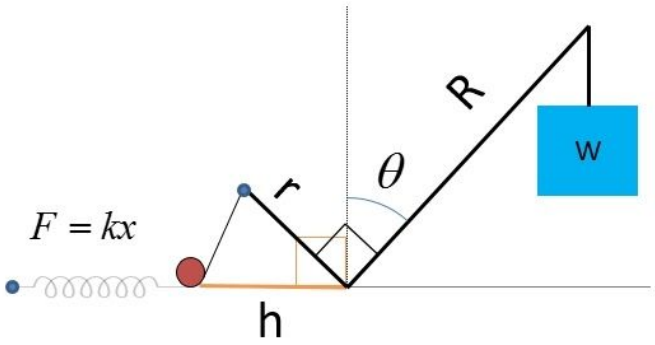
$$T_{spring} = k|R||h| \sin\theta$$

$$|R||w| \sin\theta = k|R||h| \sin\theta$$

$$w = kh$$

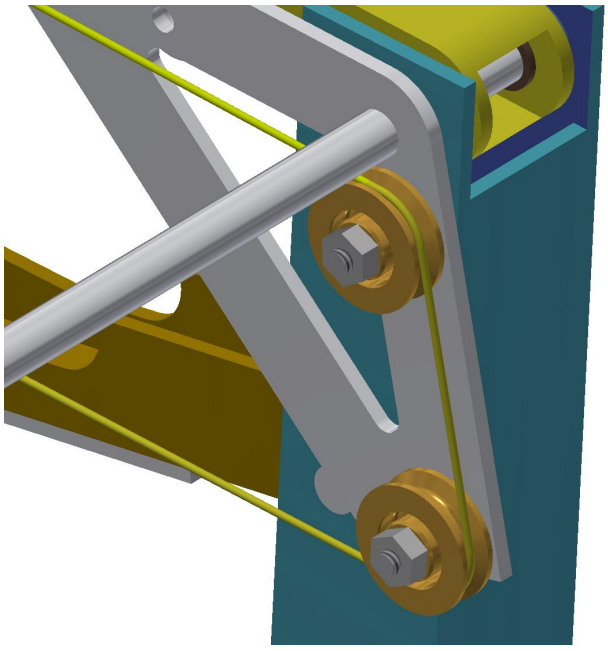


Left: Moving the spring away from its original position and connecting it to a string allows us to simulate an ideal, zero-length spring.



Right: By adding a second, shorter arm to the original arm, we can reposition the spring to a more convenient location.

For our springs, we chose to use latex tubing for two reasons: It allows us to easily adjust the length and spring constant of our springs, and it is capable of stretching a much longer distance than most metal springs of similar length without breaking.



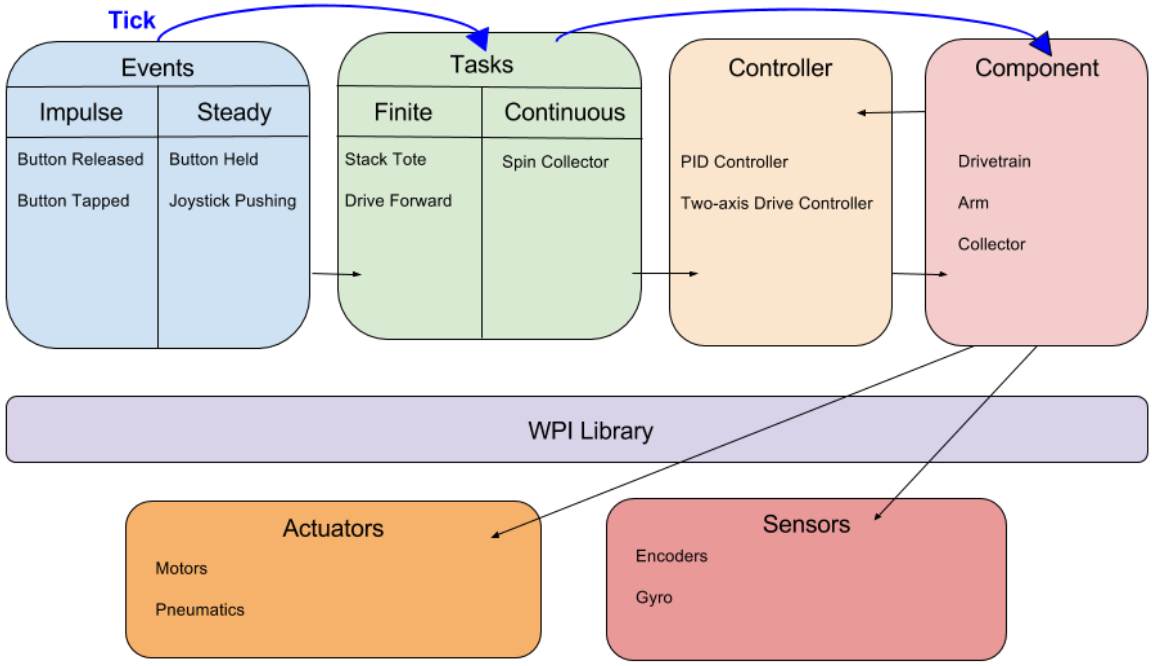
Left: Measuring the spring constant of 3/4" latex rubber tubing with 43 lbs of weights. Andrew Ng (Freshman).
Right: Pulleys are used to both change the direction of the string and also allow us to simulate a zero-length spring. (Spring is to top left corner)

Software

Shadaj Laddad (Sophomore), Philip Axelrod (Sophomore), Sanghak Chun (Sophomore)

Potassium Architecture

This year, we developed a novel architecture for our robot software that focuses on having clear connections between different levels of our code, preventing conflicts between the responsibilities of different areas, and building modular code that avoids unnecessary communication between unrelated parts of our code. Our architecture is implemented with functional-programming principles and focuses on immutability, both of which are major trends in the software industry that have been adopted by large companies such as Google and Facebook. In addition, our architecture follows the "functional-reactive programming" style, hence the name Potassium (a very reactive chemical element).

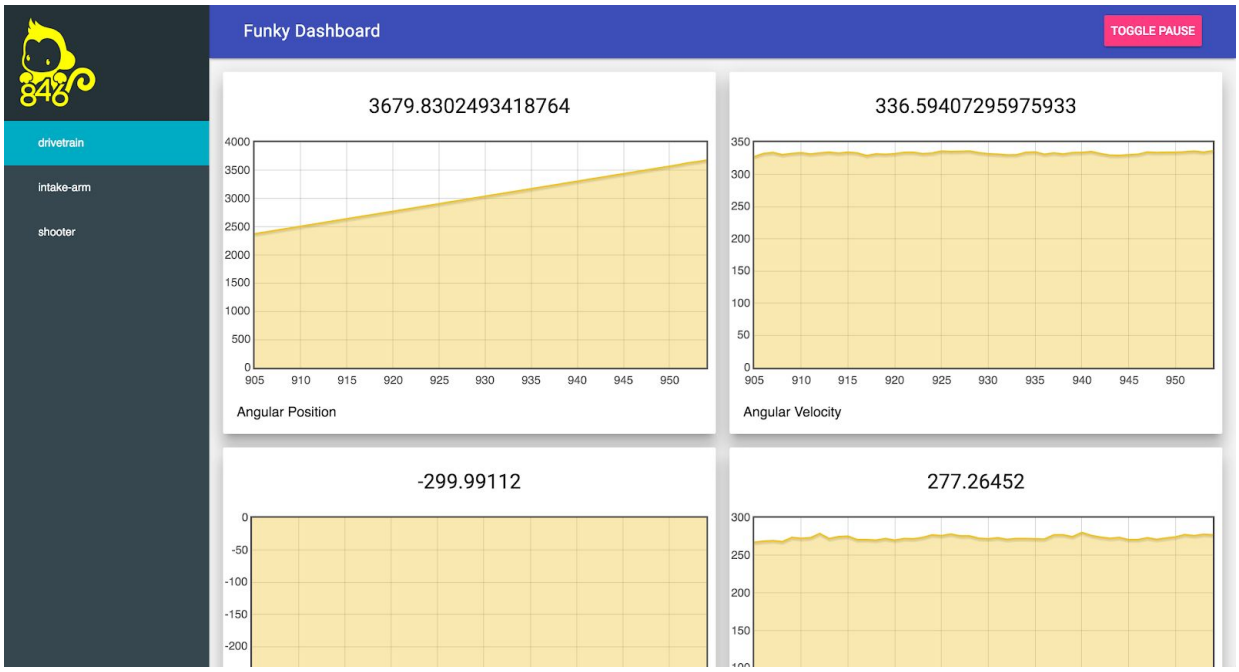


Overview of Potassium

Potassium helps with software development by having distinct sections of robot control. At the base level are components, who handle the transfer of control commands such as motor speeds to control interfaces like speed controllers. To get these control commands, components request data from controllers, which contain the actual logic for things such as position control.

Each component is always attached to one controller, so the next level in Potassium is tasks, who make the changes to controller attachments. Tasks have the job of coordinating controller assignments to different components in order to execute an action. Through a simple DSL (domain specific language), tasks can be composed using words such as “and”, “then”, and “meanwhile”. Finally, to trigger tasks are events, which can be assigned to both impulse events such as a button being released and steady events such as a button being held. By connecting events to tasks, all of the driver control mappings can be easily set up in just a few lines of code!

Funky Dashboard



As our robots became more complex and performed more intricate maneuvers, the need for effective data-logging increased as well. With the additional computing power available on the roboRIO, we were able to design an innovative interface to display and log various types of live data from the robot.

Funky Dashboard uses the latest HTML5 technologies to offer a smooth and easy-to-use interface. It streams data from the robot through WebSockets, which offer an efficient and cross-platform method to communicate between the browser and the robot. WebSockets are also bidirectional, which opens up the ability for robot operations such as automated calibration routines to be controlled through the browser. To implement our own web server, we used Akka-HTTP, which is a web server built on top of the Akka actor library for the JVM. This was chosen due to its simplicity of use and for its ability to stream data. To implement the client, we chose to use Scala.js, which

allows us to write our entire web code in the Scala programming language and have the code compiled to JavaScript for execution. As Scala was originally designed for the JVM, this allowed for using almost identical models on both the server and the client.

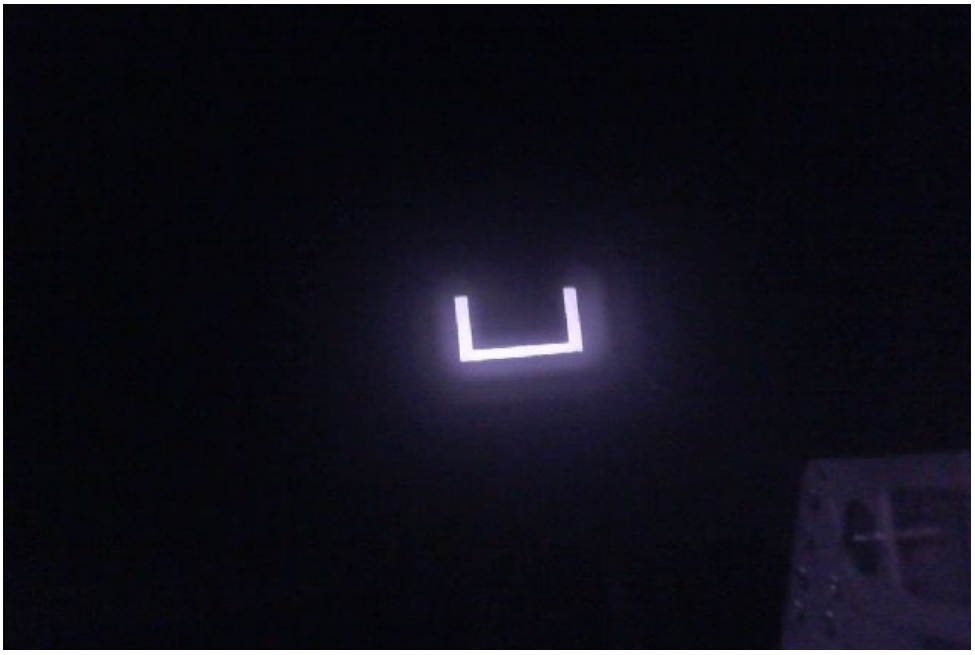
Funky Dashboard supports all the types of data you may want to display. To display charts, Funky Dashboard leverages Flot.JS, an open-source charting library. Creating your own data type is as easy as providing a serialization method on the server and a renderer on the client. Datasets are inserted by providing a function that outputs data to be displayed; this function will be called by the dashboard whenever it publishes new data.

The dashboard uses Google's Material Design spec for its user interface. Each dataset is displayed as a "card", one of the core elements of Material Design. Each card is managed independent of other cards, so cards can be frozen to ignore incoming data. The sidebar allows for switching between different data categories. By doing this, data can be sorted out by components, which allows for efficiency when working on specific parts of the robot. To make viewing data even more convenient, Funky Dashboard uses responsive design so data can be analyzed with phones, tablets, laptops, and even TVs!

Computer Vision Auto-Aim

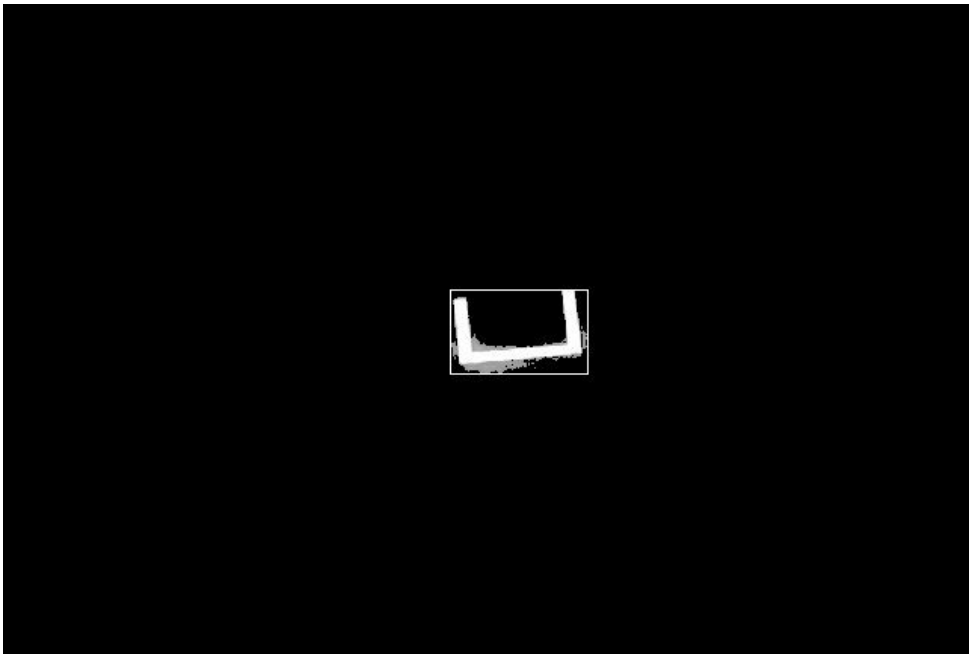
Especially during autonomous, it can be difficult for the robot to always drive to the perfect position for a shot into one of the goals. To give the robot the ability to "see" the goals, we implemented a goal tracking system with computer vision algorithms.

To make it easy set up hardware for collecting targeting images, we chose to use green LED rings as our retroreflective tape light source. By using light in the visible spectrum, we eliminate the need to modify the camera we use for collecting images and open up the possibility of using the camera for other purposes such as streaming to the drivers for a better idea of where the robot is on the field (something especially important with driver view obstruction from obstacles). To make sure that our robot does not mistakenly target bright sources of light other than the reflective tape, we use HSV (hue-saturation-value) filtering to filter out all non-green light, and have manual exposure controls set on the camera so that recalibration at different events with different lighting conditions is not needed.



The value part of HSV-converted images, where the goal tape shows up brightly

Once the algorithm receives a new camera frame, it filters out areas of low reflection and then looks for bounding rectangles around the contours of the image. By applying some extra logic on the expected height-to-width ratio of the boxes, we are able to successfully detect goals. The whole algorithm runs on the Intel NUC coprocessor, which sends UDP packets with detected goals to the roboRIO.



Identified target from the previous image

On the roboRIO, transformation from raw image coordinates to angular errors takes place. With a little algebra and trigonometry, our software is able to calculate both the distance to the tower as well as the number of degrees the robot should turn to face to goal. During an auto-aim routine the angular error is used to create a new rotational target for the robot, which the software rotates the robot to using the IMU.

Scriptable Automation Routines

During rapid iteration cycles of autonomous routine development, we found that deploying to the robot took a substantial amount of time such that it quickly became tedious testing different versions of autonomous routines. We can now develop routines in an agile fashion by switching autonomous routine specification from robot code-based models over to a JavaScript-based model.

Our autonomous routines can now be written by using JavaScript to chain together different tasks that are coded in the main robot code base. On the robot side, we use Rhino, a Java based JavaScript interpreter written by Mozilla that allows us to easily expose Java APIs to the JavaScript routines so that we don't have to implement additional functionality in Potassium to handle JavaScript routine writing.

For example:

```
function constructRoutine() {
  var moveForwardForCollect = new MoveForward(10.0);
  var collectBoulder = new Collect();
  var turnRight = new TurnDegrees(90.0);

  return moveForward.then(collectBoulder).then(turnRight);
}

return constructRoutine();
```

Absolute Rotation Measurement with IMU sensor

Maintaining and turning to a specific direction is an essential part of this year's game. We must maintain a forward direction to align to defences and goals, cross defences, and assisted direction driving. To do this, we mounted, communicated with, and implemented negative feedback control with an Inertial Measurement Unit, also known as an IMU. The IMU reports data about the angular velocity about the x, y, and z axis

that we use to maintain and turn to a specified direction.

Establishing a connection to the IMU was only the first step. The IMU data reported by itself is extremely unreliable. Even when the IMU does not move, it reports data suggesting that it was moving. This causes a very large drift in our estimated rotation. To solve this issue, we calibrate our gyro by measuring data the IMU reports while it is stationary. We take the average of this data to get the drift, which represents an excess value added to every real value that the IMU reports. When we sample data from the IMU, we subtract the drift from the reported data, since we know that the drift is added to the actual value.

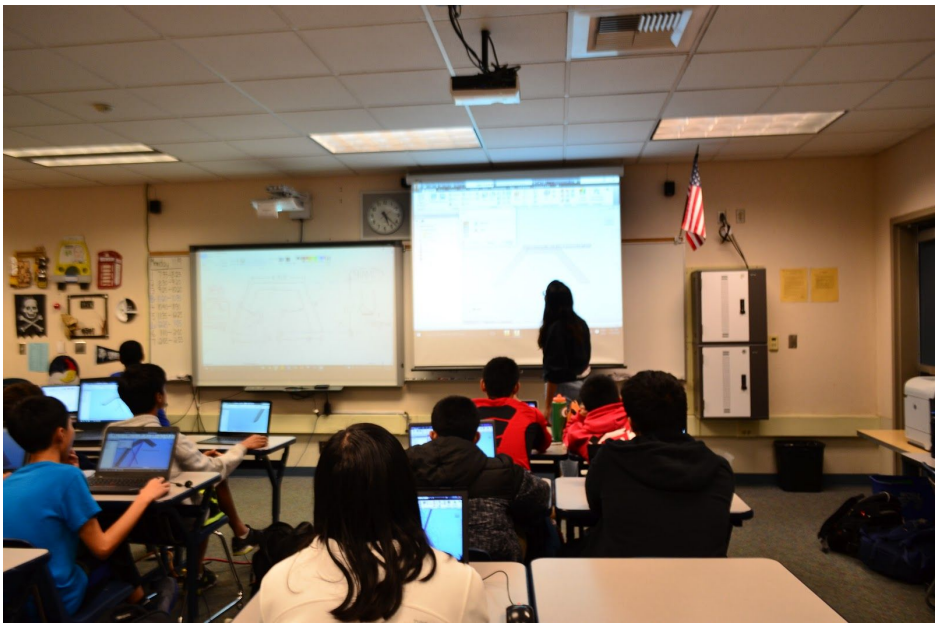
The digital IMU that we use can only report angular speed. To derive our position from velocity, we experimented with different techniques for integrating velocity to derive our angular position. First, we implemented a Riemann sum, where we assumed that the velocity reported by the IMU stays constant from the data sampling. However, this is a poor approximation, as there is always velocity change that occurs in between sampling updates. A better method of integration we implement in our code is trapezoidal integration. In trapezoidal integration, we take the average of the previous and current velocity, which makes a more accurate approximation that accounts for the change of velocity that occurs in between data sampling. We multiply the averaged velocity by the time passed to find the change in angular position from the last sampling. Our code abstracts the process of calibration and integration into a separate class. This is a valuable asset to our software as it make the process of switching to a more accurate IMU much prone to error and with less hassle, which we in fact did.

CAD in Robot Design

Amrita Iyer (Senior)

CAD Training for Members

A primary tenet of Lynbrook Robotics is to teach CAD to new members to increase the number of people involved in the robot design process during build season. Over the past six years, we have trained over 50 members each year in using the Autodesk Inventor CAD software. Our training includes both classroom-like instruction through a series of lectures and presentations as well as personalized teaching before and during build season. Students are able to gain comprehensive knowledge of Autodesk Inventor through individual designs that the students work on and smaller in-class design competitions. The students then use their new CAD skills to contribute during build season. In fact, this year we were even able to get four of our rookie members deeply involved in the complex design work.



2016 CAD Workshops

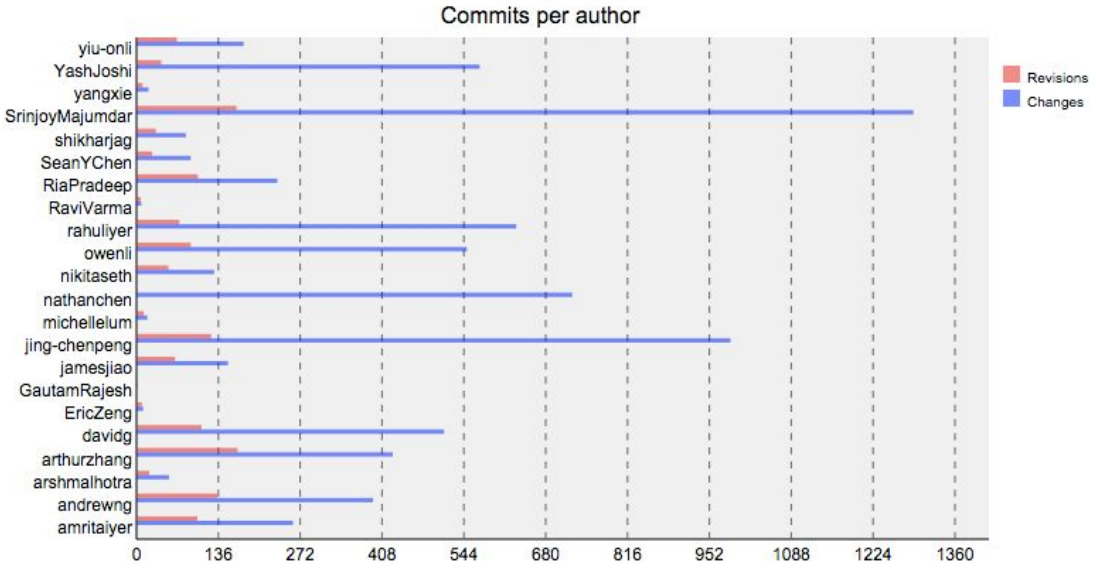
Division of Responsibilities and Workload

Lynbrook Robotics uses a "top-down" approach to split up the design workload amongst multiple members. We analyze the necessary functionalities of our robot and then split up the design responsibilities amongst different subsystem leads, who collaborate to put together their CAD assemblies into one final robot model. Through a use of

sub-assemblies and modular robot components, we are able to create a robot model that can be easily modified when necessary. Our robot's three primary subsystems this year included our drivetrain, our shooter, and our collector. All three of these subsystems were student-led and student-designed.

File Management and Collaboration through Subversion

Over the years, Lynbrook Robotics has come to use the Subversion version control software. We have re-purposed this software, which is used for managing and sharing software code, to facilitate our CAD modeling work. Subversion has become crucial in managing our multiple robot CAD files, including our machinist drawings for fabricating parts. Subversion has also allowed us to involve almost a third of our members in design. These members have been able to get very active during build season in creating CAD parts for our robot, creating drawings for machining, etc. We currently have 1407 revisions to our Subversion repository, which we host on our private server.



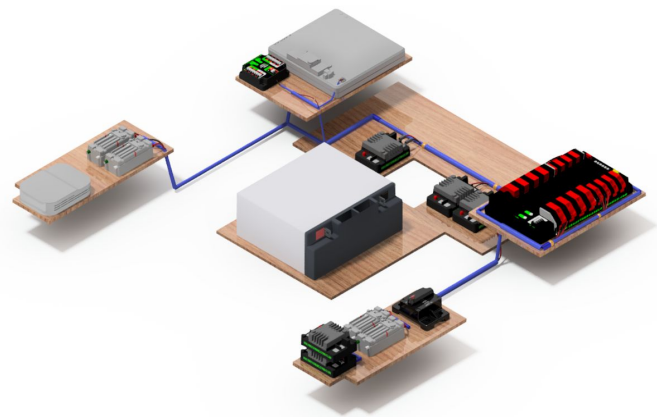
Statistics for repository commits per author.

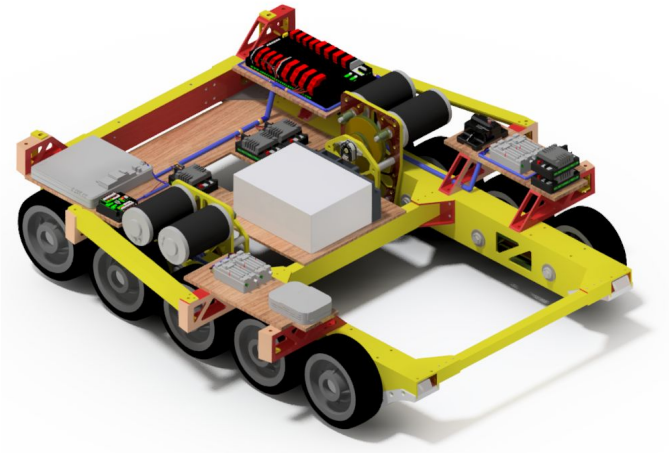
Electrical

Eric Zeng (Senior), Shikhar Jagadeesh (Junior), Nathan Chen (Junior)

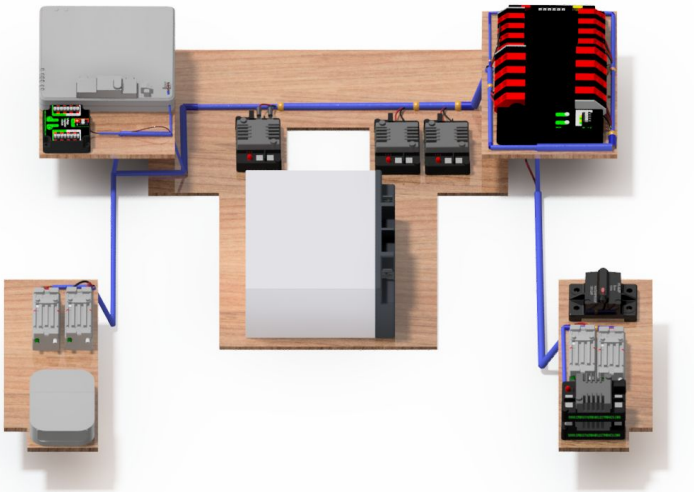
Wiring the Robot Using CAD

After two years of successfully using Autodesk Inventor's Cable and Harness environment, we decided to implement the tool in our electrical design once again this year. We used the Cable and Harness environment to model the wiring of almost all of the electrical components. As the robot subsystems continued to iterate, we updated the wiring to accommodate for those updates.





Autodesk Inventor allows us to route and measure all wires in the model. Since we had the lengths of all the wires, we were able to cut all of them to length before the robot was even assembled. The electronics for our robot were finished before our robot was even assembled.



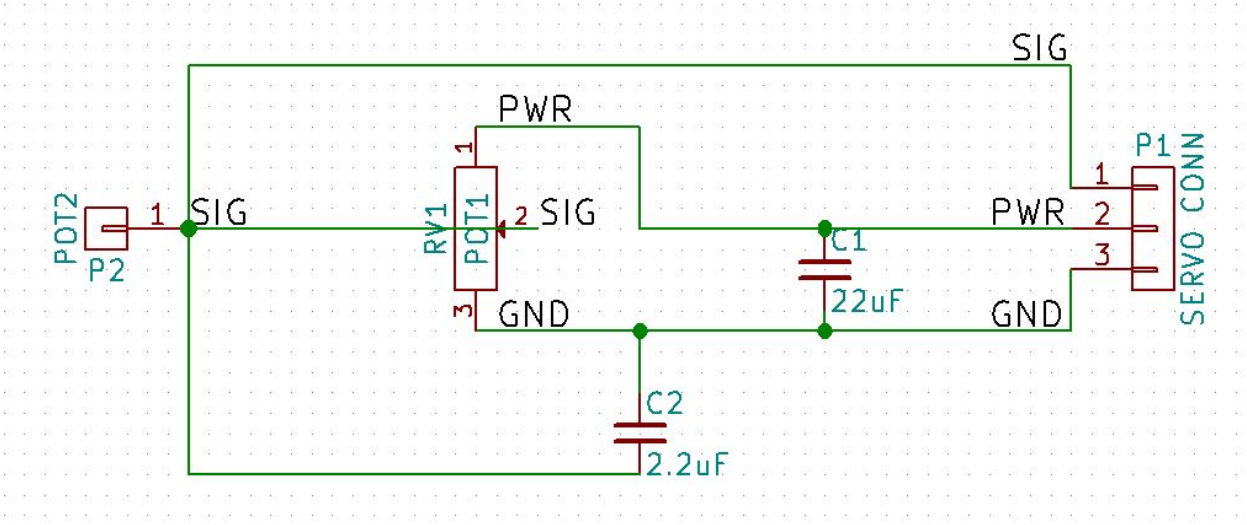
Printed Circuit Boards

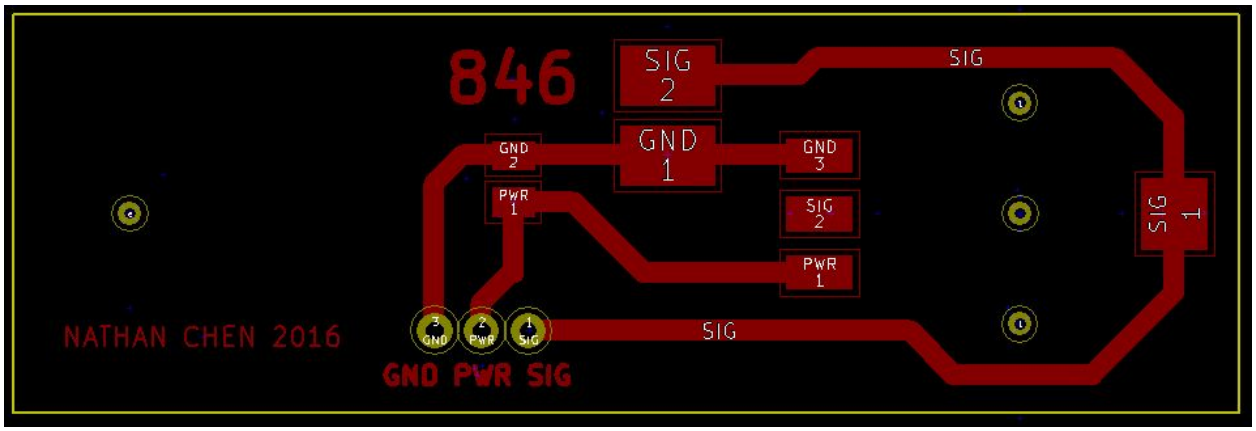
Nathan Chen (Junior)

This year, the Funky Monkeys implemented printed circuit boards (PCBs) on our robot. However, instead of making our boards by sending our designs to a professional manufacturer, we made our PCBs in-house. The result was a fun, educational do-it-yourself experience.

Step 1: Design

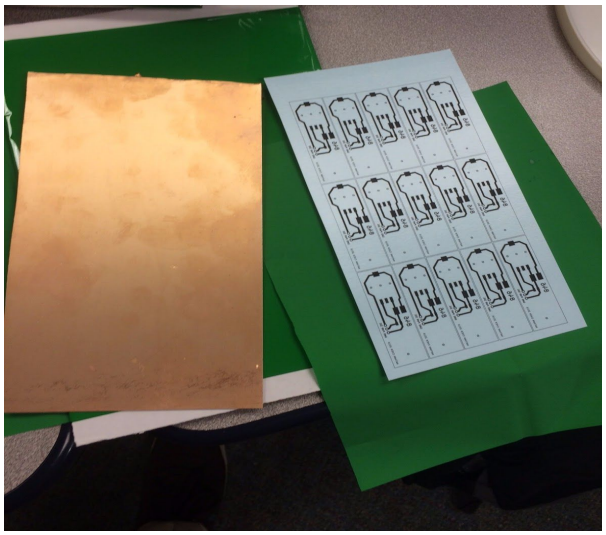
Our PCBs were designed with KiCAD, a computer program that designs circuits and circuit board layouts.

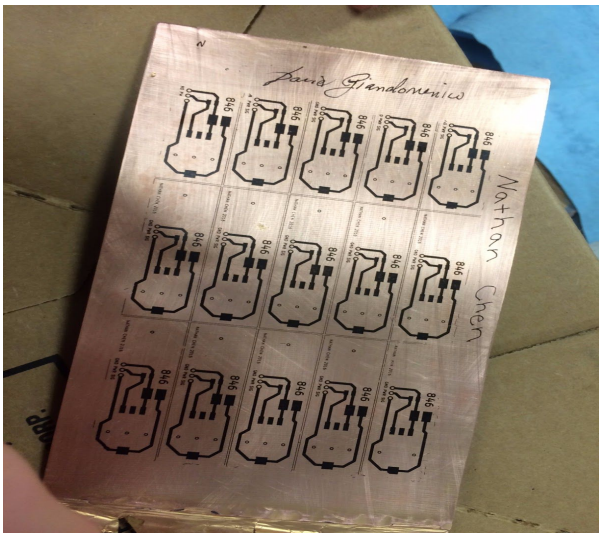




Step 2: Printing

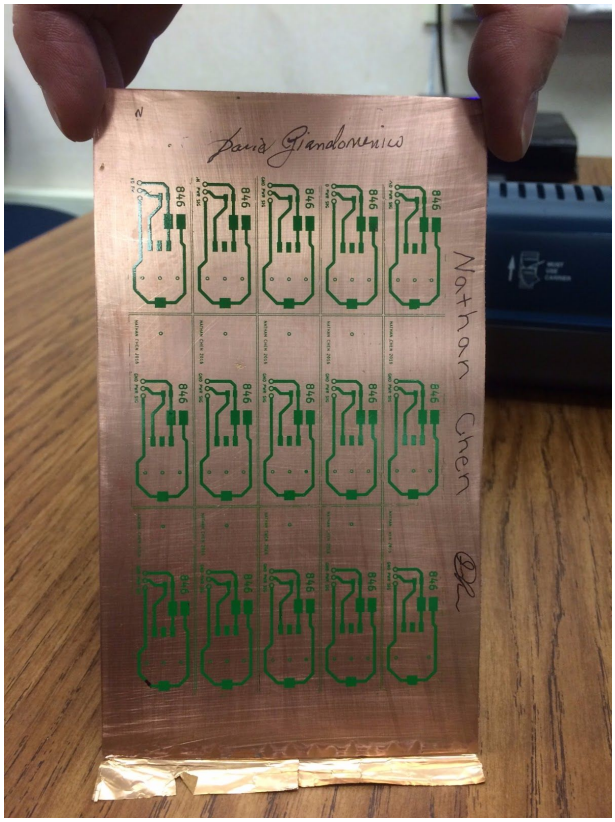
We then printed the board layout onto special transfer paper that could transfer ink onto copper.

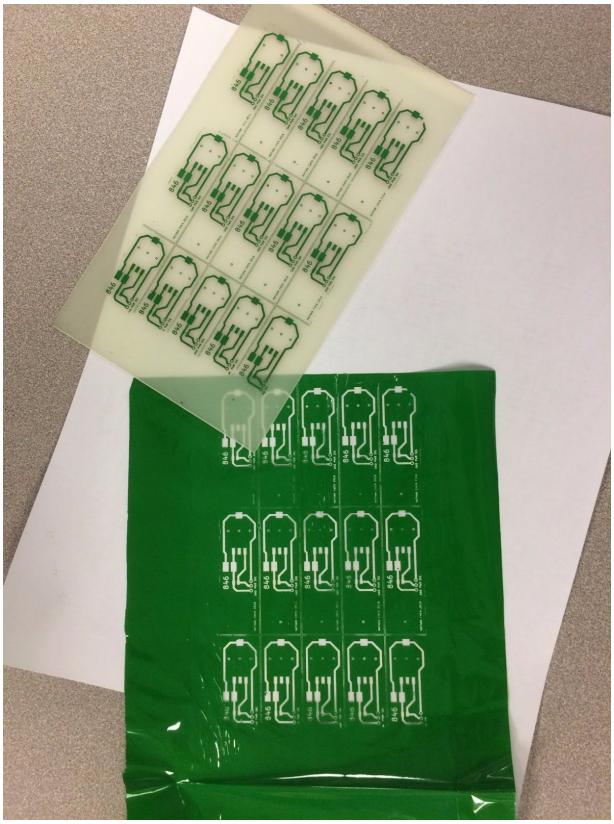




Step 3: Etching

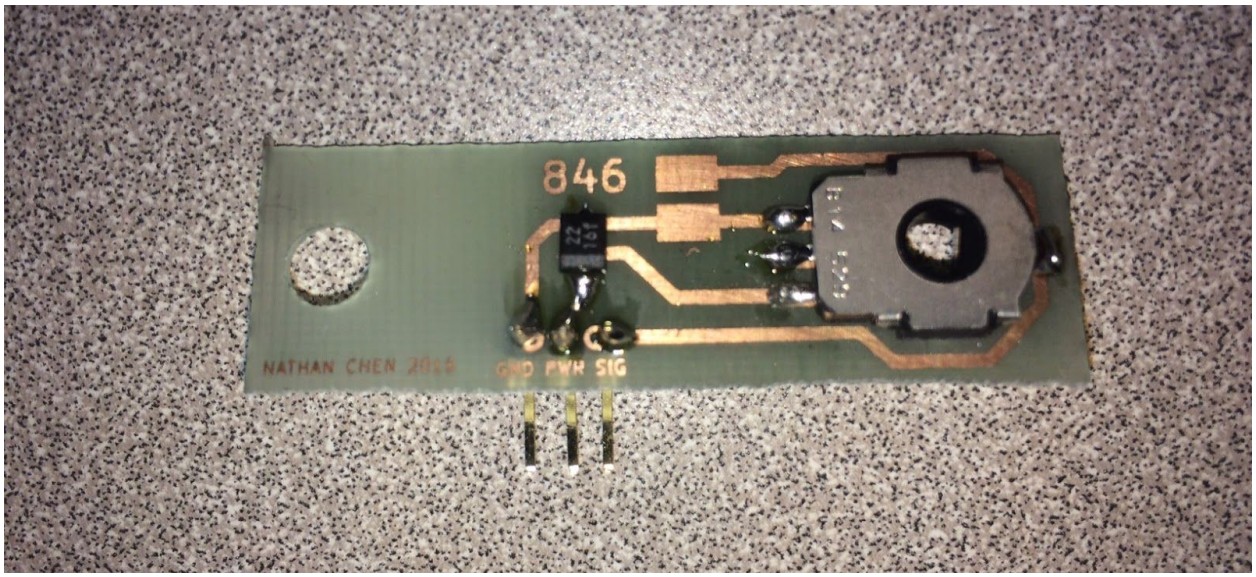
After the copper traces were marked by the transfer paper, we sealed them in with a green laminating foil. Then, we used ferric chloride to dissolve the rest of the copper, leaving only the copper paths we wanted.





Step 4: Drilling and Soldering

Finally, we drilled necessary holes and soldered on parts, completing the circuit board!



Conclusion

Monkey Python's modular design allows electrical, software, and hardware student subsystem leaders and their teams to work on various aspects of robot design at the same time. Our Subversion and Git repositories allows all the parts of our robot design to be easily combined.

Throughout the season, our robot has been able to do well in competition and impress many people with the design, winning us the Judge's Award at the Arizona Regional and the Quality Award at the Silicon Valley Regional. We are proud of all of our accomplishments this season and look forward to improving it for the offseason competitions this fall.