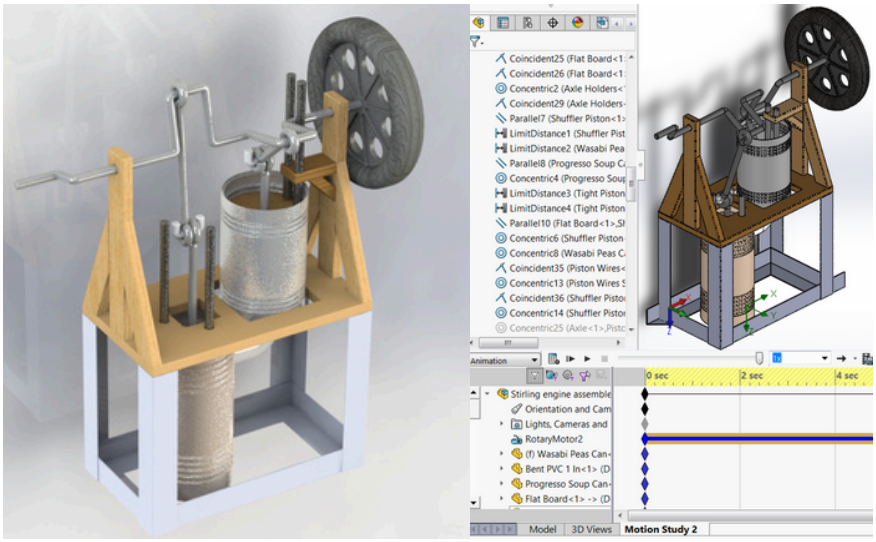
PERSONAL PROJECTS

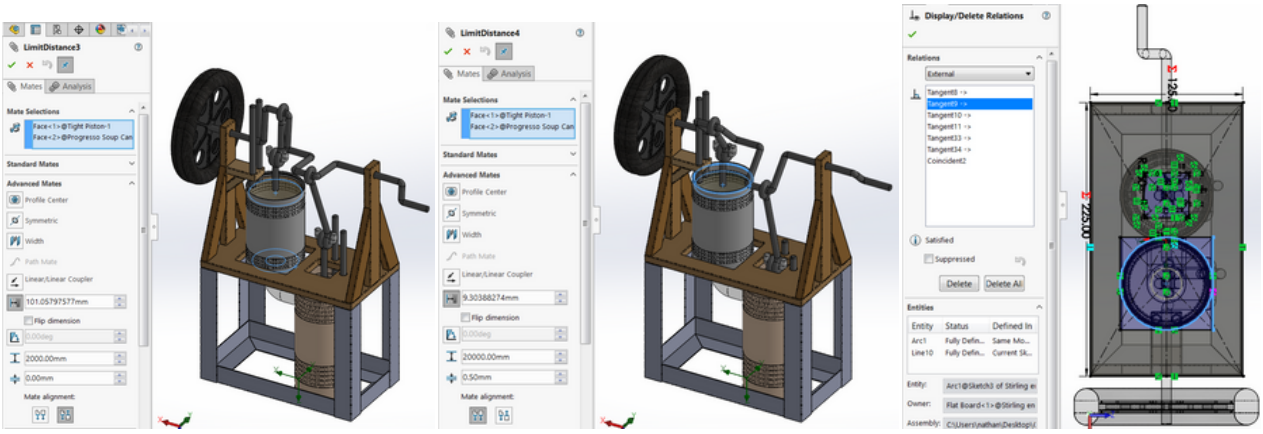
CSWP MECHANICAL DESIGN — DECEMBER 22, 2021

After two and a half years of teaching myself SolidWorks, from basic sketches and extrudes, to lofts, and then surface modeling, I decided to finally try the CSWP. Being my hobby and free time activity that took up most of my day during summer, or when I was done with homework, I effectively taught myself how to use the software, and made many projects. I passed the exam.

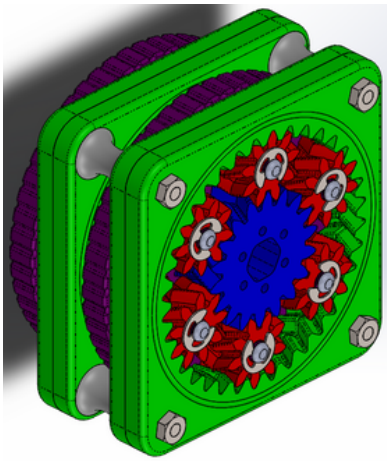
GAMMA STIRLING ENGINE — MARCH 6, 2020

This was a significant project for me, because I learned a lot. This was the assembly that I did with moving components. I learned to utilize the concentric mate to allow the pistons to move, and to use the limit distance mate to make sure that if I decided to change the length of the rods, that the pistons would stop at the bottoms of the cylinders, instead of moving through them. This was also the first assembly I learned to define sketches externally. I was able to define the wooden frame's sketches with objects in the assembled engine. I also learned to do basic animations, and make the engine continuously rotate.





VARIABLE PLANETARY GEARBOX — AUGUST & OCTOBER 2020

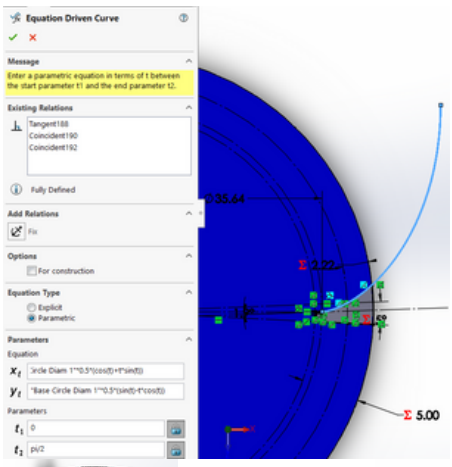


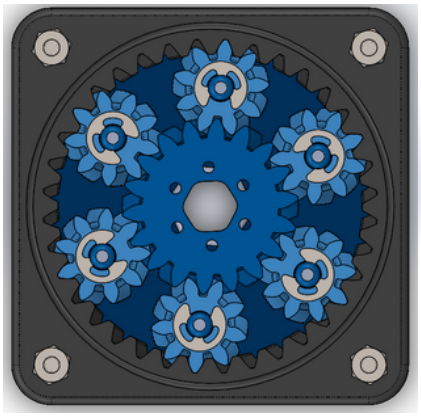
This project was a significant learning experience, and an important step in allowing me to make my transmission later on. Interested in building a generator, I needed a gearbox. However, I couldn't find the exact size that I needed online, so I decided to make my own gears. In making my custom gears, I learned what an involute was, an equation driven curve to achieve the tooth profile. Using variables, I was able to control things like pitch diameter, pressure angle, and the teeth each gear had.

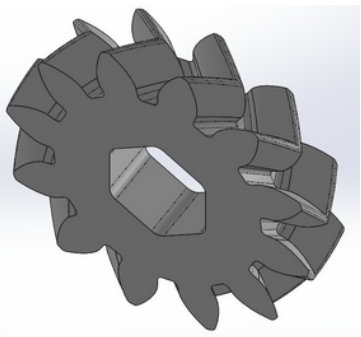
I then went a step up in October, and decided to make a planetary gearbox. The variables you could input were: The ring gear pitch diameter, the ring thickness, and the number of teeth of the sun and planet gears. Instead of customizing the sun's and planet's pitch diameter, they would be automatically calculated based on those four inputs. They would be made to automatically fit within the ring gear, and have the proper size ratio.

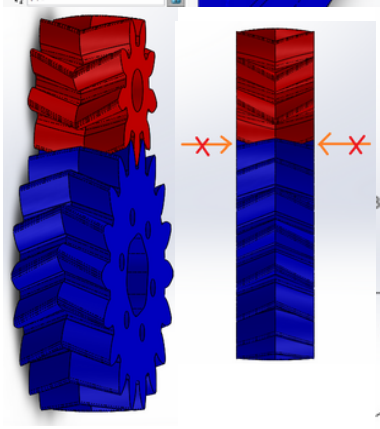
In the assembly, I learned how to control planetary gear ratios I learned that you needed two inputs. I typically held the ring gear (inputt #1), and spun the carrier (input #2), driving the sun gear, outputting higher speed. I learned How to get multiple ratios with one gearbox, including reverse (holding the carrier, and outputting either the sun or the carrier), or higher torque (holding the ring, and spinning the sun, outputting the carrier).

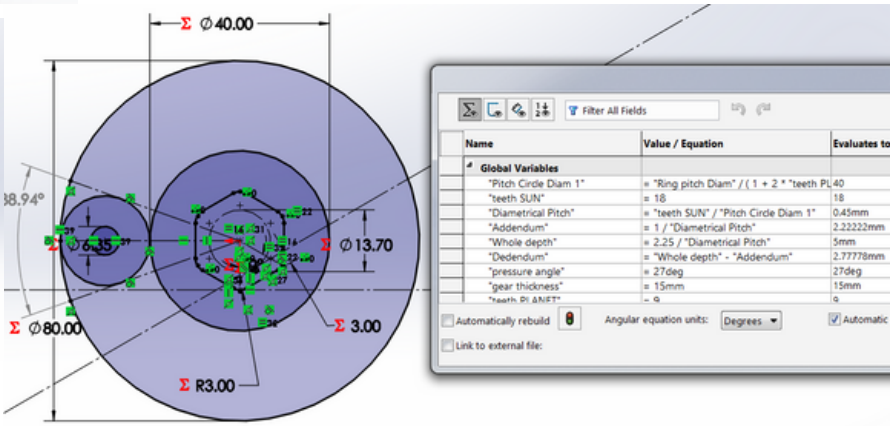
Learning that helical gears were quieter, and could take more abuse, turned my straight cut spur gears into helical ones with a sweep command. However, helical gears will push each other out due to their angled teeth producing axial force. Instead of using a thrust bearing, which is normally how this is solved, I instead put a helix counterclockwise, and turned my gears into herringbones.
Herringbone gears have all the same advantages as helical gears, however, the axial forces they produce are cancelled out, because the helixes go in opposite directions. This double helix also allows the gears, if they are put on an axle, to be locked axially, since the angled teeth will prevent the gear from sliding in and out. This is why the E clips on the planetary gears are only used to lock in the carrier to the box, and is why the sun gear is held in place despite there not being set scews, clips, or any other locking hardware.



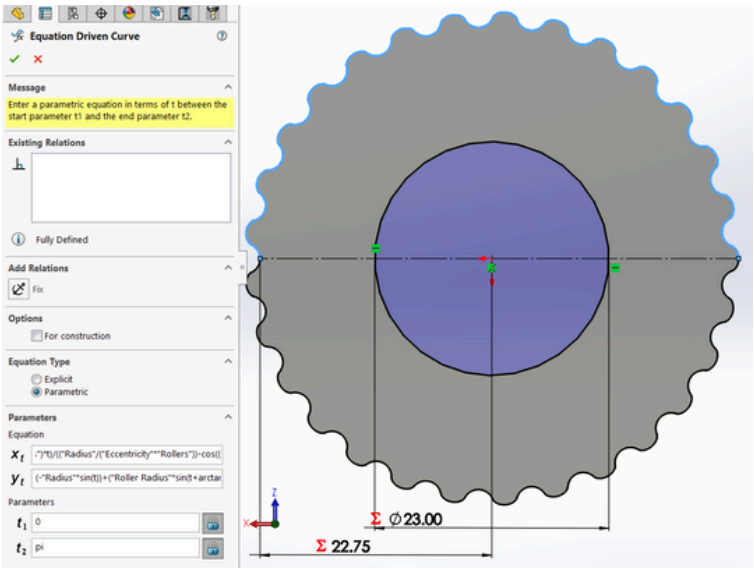








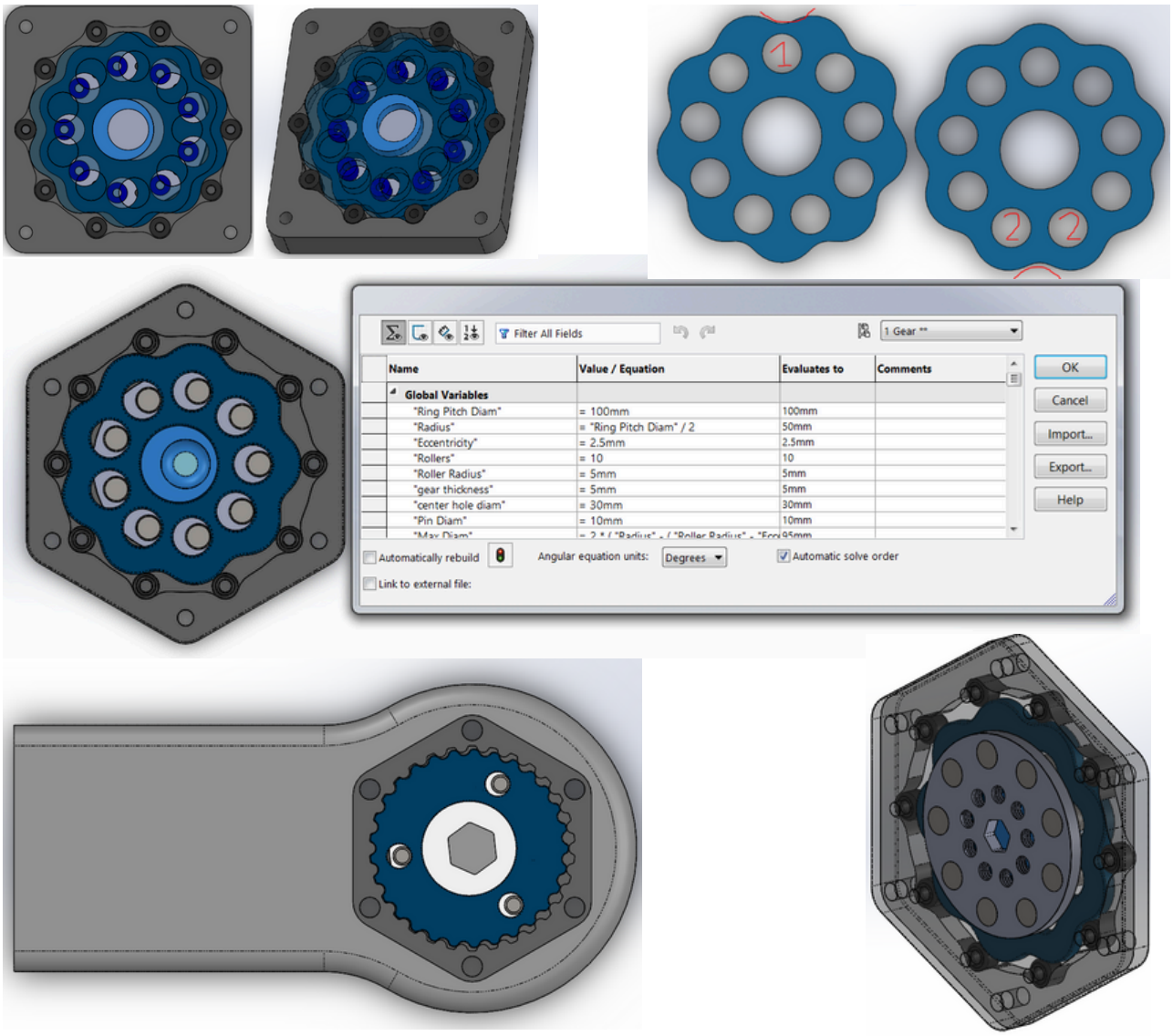
Wanting to explore more gearing, I tried my hand at making a cycloid gearbox. Here, I again used an equation driven curve to generate the gear profile, and controlled it's parameters with global variables.



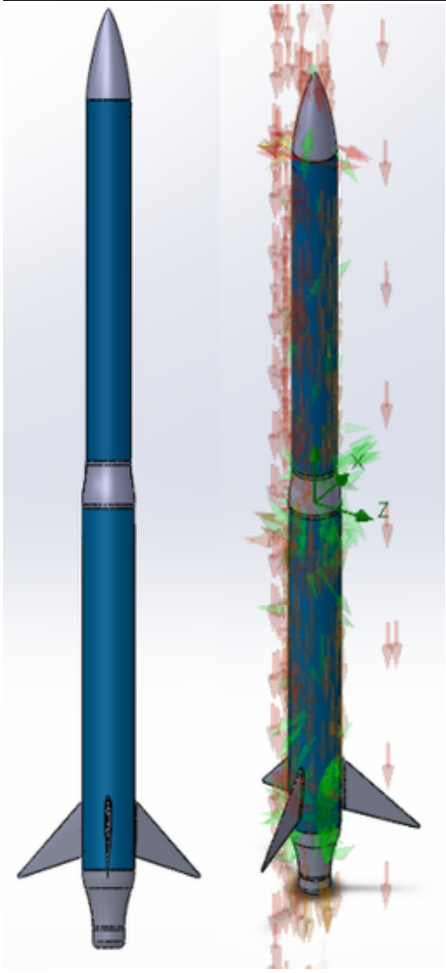
Like the planetary gearbox, the cycloid gearbox requires two inputs. Typically, the ring (whose teeth are called rollers) is held in place, and the eccentric shaft is spun. When the cycloid makes a full rotation around the ring, the output, a disk whose pins go through the cycloid, will rotate once. This offers a heavy reduction.

ANother way to drive it is to hold the disk with pins still, spin the eccentric shaft, and output the ring. This is what I have proposed for an arm.

One problem with the cycloid gear is that because it spins eccentrically, it wobbles. This is why I have ways of mitigating it. One way is to add a second disk on top, 180 degrees opposed to the first. However, this solution is not exact, as the mass isn't equally cut radially, as shown below. This is why my favored solution is to make an exact copy of the first cycloid, and place it 180 degrees, The cost of this is making the pins on the disk shorter.



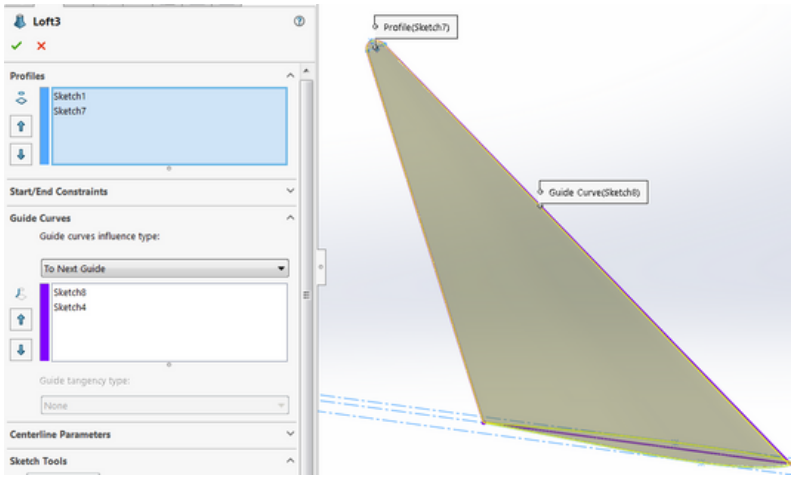
TARC ROCKET — NOVEMBER 10, 2019



This was the one of the first projects I did independently. My robotics team gave me SolidWorks in late June of 2019, and I had been making parts and assemblies for them. This was one of the first times I started using SolidWorks for myself, and I managed to design my own rocket for the TARC 2019-2020 season. Unfortunately, due to COVID, the rocket did not get to compete.

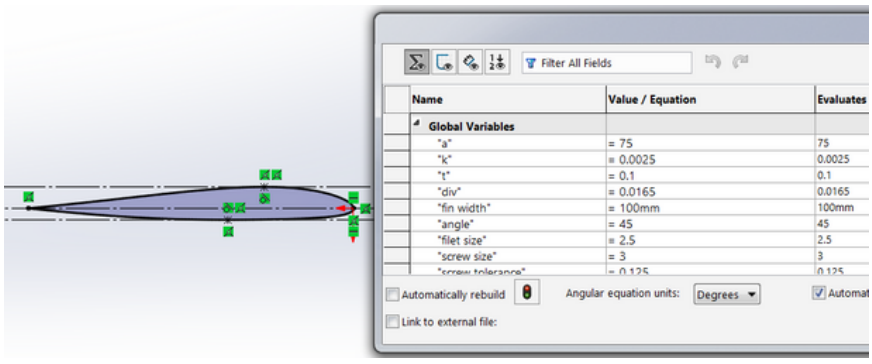
I designed all the components of the rocket myself; I decided to use a Haack nose cone, because I read that it minimized drag. To achieve that shape, I used an equation driven curve, with variables allowing me to change its length, base width, and other factors. Similarly, I also designed the fins. In previous competitions, including the 2018-2019 season in which we competed nationally, we had used tube fins. However, not wanting to rely on drag to control the height of the rocket (due to it being affected by wind and temperature, which are out of my control), I tied minimizing drag by making my fins an airfoil shape. I ended up with two fins: one asymmetrical airfoil to give the rocket spin, and one symmetrical airfoil, to make the rocket go straight. In the end, not wanting to risk anything, I used the symmetrical fins. I also completely redesigned the transition cone (the one in the middle that transitions the thinner top tube to the wider lower tube), and the tail cone. Previously, the team used sharp edges and cones with triangular cross sections for both. However, wanting to minimize drag as much as possible, I used style splines to accurately create a curvy shape for each.

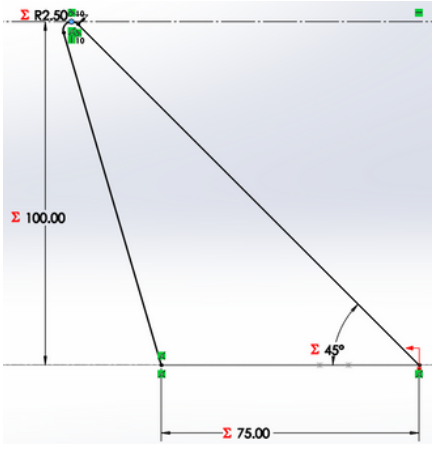
When I finally launched, the highest my rocket went was 1209 feet. Although the target was 856 feet, the performance of my rocket suggested that a weaker, and thus cheaper motor could be used to launch the rocket to the desired height.

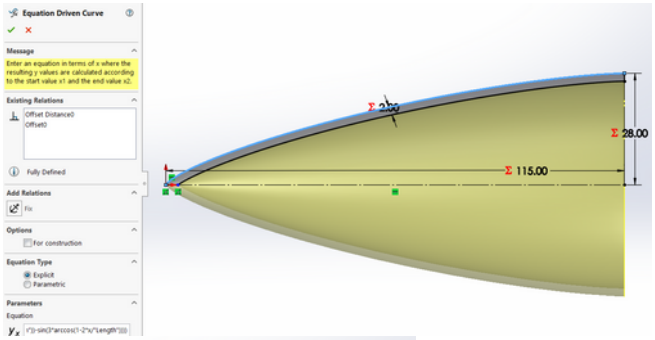


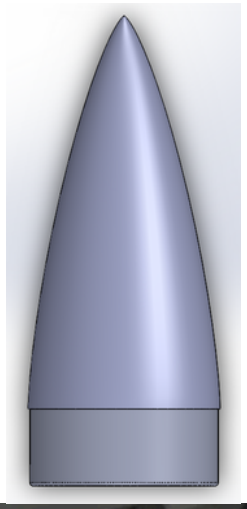
I learned how to use the Loft command with this project. Here, I needed to transition the airfoil shape to a single point at the very tip. However, instead of having the contours from the edges of the airfoil to the tip be straight lines, I wanted it to conform to an angled wedge shape with a rounded tip.

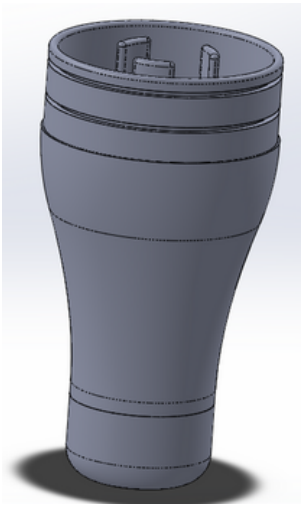
This is when I learned to use guide curves properly.

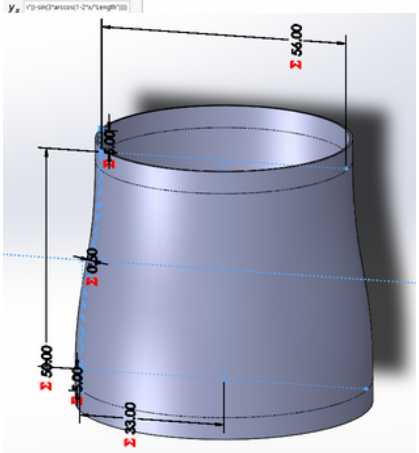


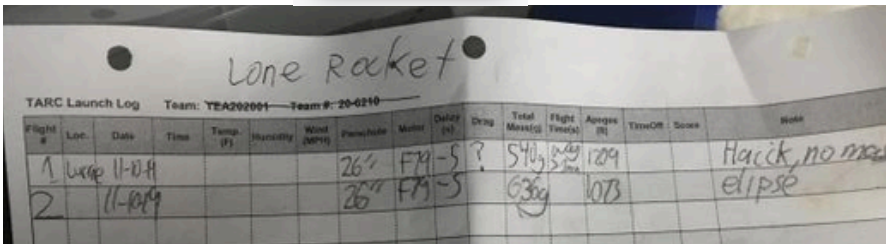


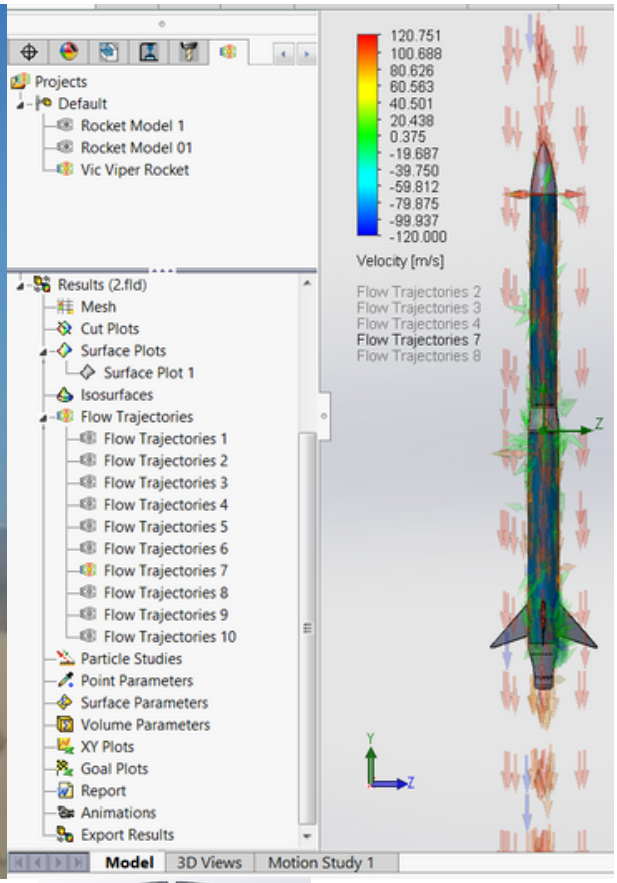






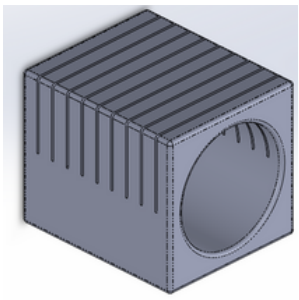


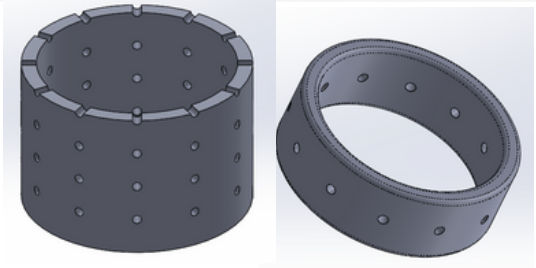




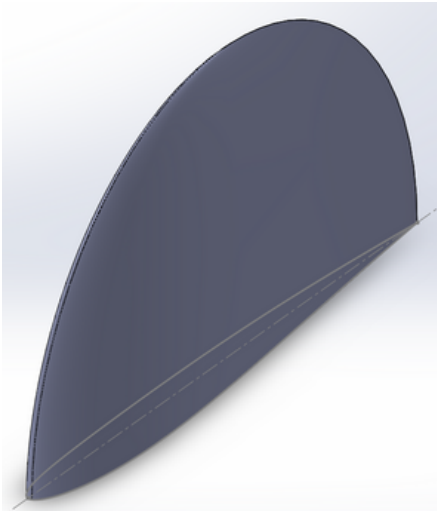
In this project, I also experimented with the flow simulation add in. Although CFD's may be inaccurate, I wanted to get an idea of the drag before actually launching to improve my parts. I set a high velocity downward flow against the nose of the rocket, and let the simulation run and generate graphics. I predicted that as long as the color gradient of the arrows doesn't change too much, then the drag will have been minimized. This is because the redder the arrows, the faster the velocity of the air. As long as the air didn't slow down too much after hitting the rocket, then it would have indicated that I had successfully minimized drag.

I had also designed tools to make the manufacturing process easier. Because I lack precision when drilling on the tubes, I created a ring with holes to guide the drill properly.



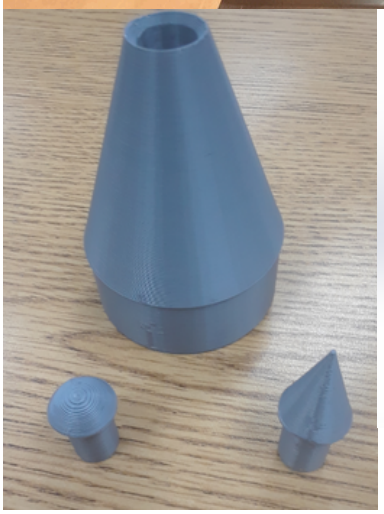


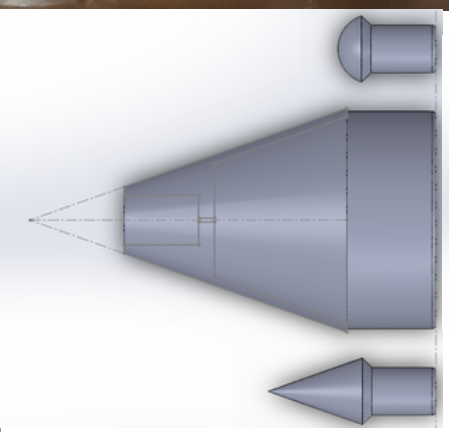
Prior to settling on the final design of my rocket, I brainstormed many ideas for it, and used SolidWorks, which I had just recently acquired, to visualize and make them. In my view, we had barely made it to DC in the 2018-2019 season, so I came up with many proposals to make the rocket flight more predictable. I settled on reducing drag, since unlike our control over the mass of the rocket (we could add or remove weights to it), drag was dependent on the weather.

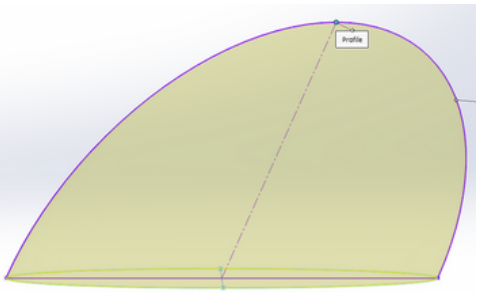


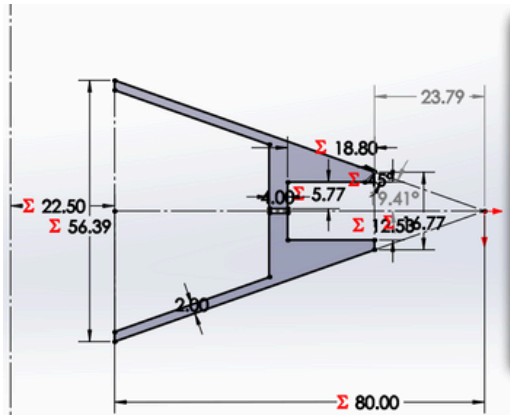
I created several nose cones: a conical nose cone, which could have its tip swapped out to be rounded, or pointed, an elliptical nose cone, and the Haack nose cone, which I finally settled on, since it reduced drag the most. As seen below, I used several variables for the conical design. This was the first project in which I extensively used variables to control the shape of my object, and where I learned to fully define everything without using the fix relation.

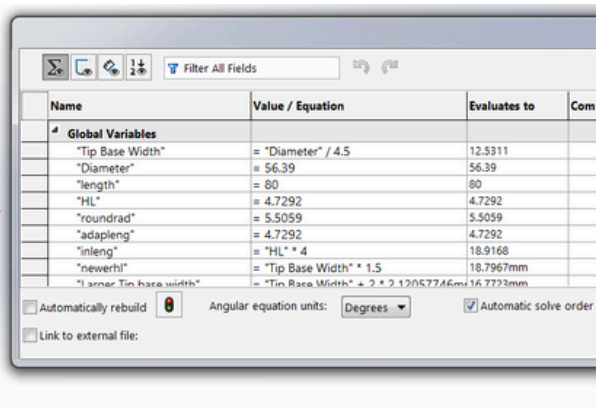
I also read that elliptical fins were suited for model rocketry, and wanting to avoid just extruding an ellipse, which would have created a flat face for the wind to drag against, I opted for having an elliptical and airfoil shaped base in the form of an elliptical fin. However, the boss and cut extrude operations that I normally went with did not give me the proper desired shape. This was when I first learned to use the loft command. This would make me dive deeper into SolidWorks, as after that, I was excited about all the curvy shapes I could make.

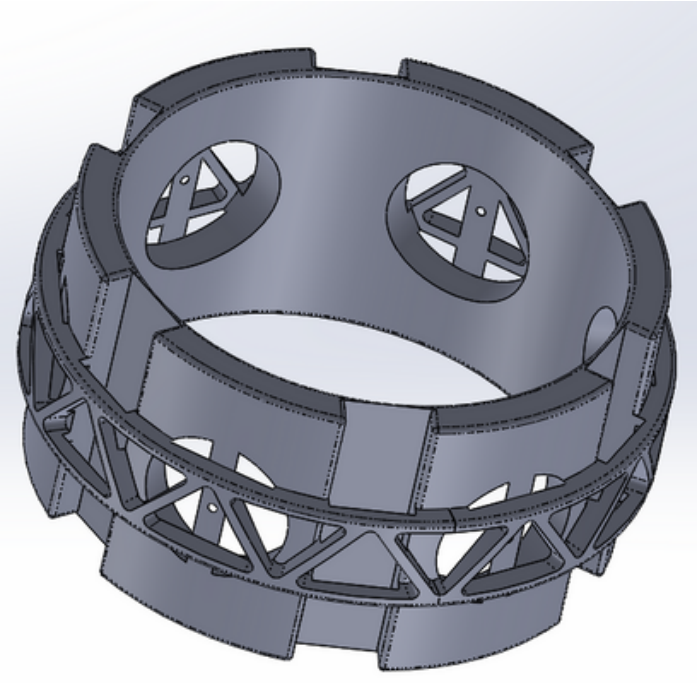






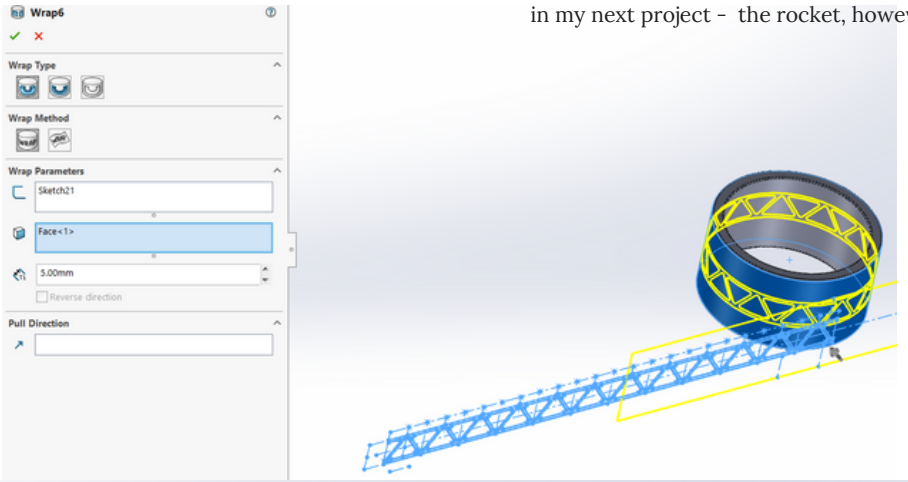


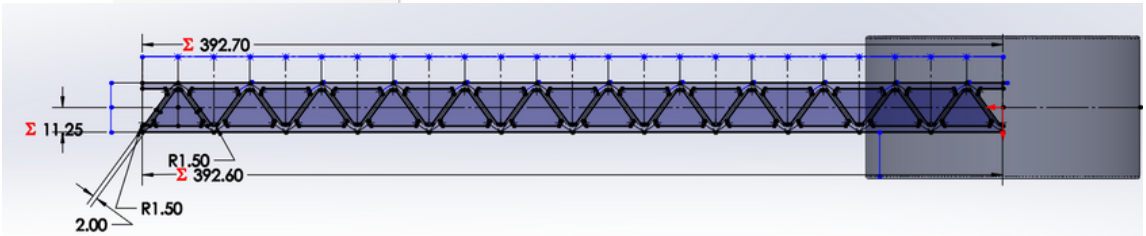


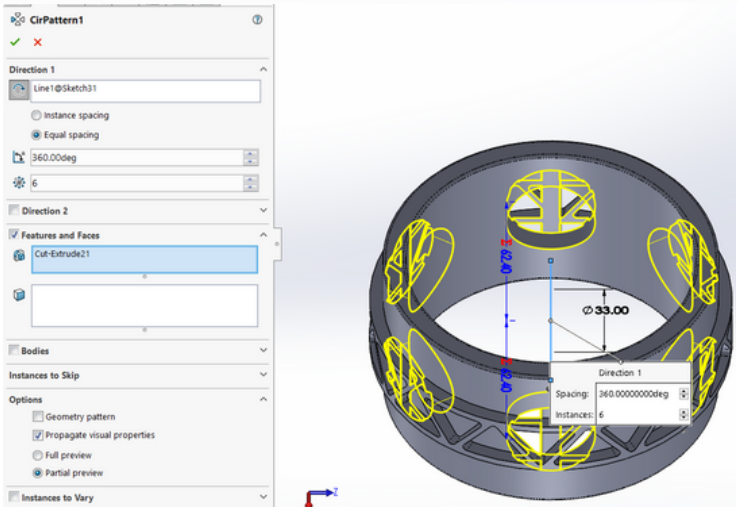


At this time, I was given SolidWorks by my robotics team recently on June 29, 2019 to help with their 2019-2020 FTC season. This was truly the first time I used SolidWorks for myself, instead of the team (I started working on the rocket shortly after this project). Being interested in generators, I wanted to design a magnetic stator. I had an alternator from an old car, and I tried using its rotor as the part that would generate electricity, since it had more coil windings that the stator it already had. Although this idea failed, it was a good learning experience, and it was the project that made me dive deep into SolidWorks.

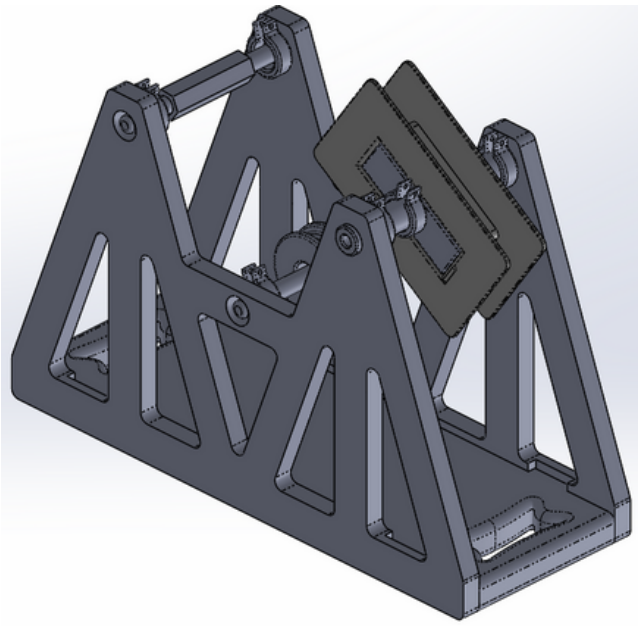
Unusual for a first project, I learned to use the wrap and circular pattern commands. Creating the "truss" pattern around the drum was harder than I thought it would be, and took me several hours to figure out, just barely starting with the software. In the end, I discovered how to properly wrap designs, and to linear pattern shapes in sketches so that I wouldn't have to repeatedly draw them. I also learned that instead of manually creating the cut extrude for each hole, I could pattern them. Towards the end, I discovered that SolidWorks had variables, and equations. This was a huge discovery, because it meant that I could easily change the size and dimensions of my part remotely, without having to redraw everything. I would only start to use them extensively in my next project - the rocket, however.







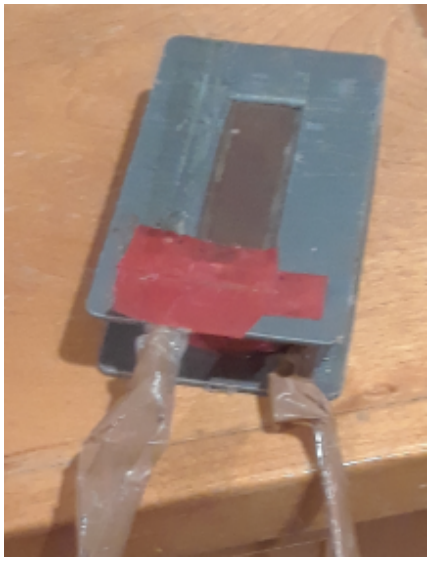
Learning from my previous failed attempt at a generator with using the rotor of the alternator, I decided that I needed to make my own coils. I learned that Lundell claw pole rotors' magnetic teeth create the magnetic poles perpendicular to the rotor, and that's why trying to use the rotor for its coils failed.



Wrapping my own coils by hand turned out to be messy and inefficient, and I lost count easily. I decided to make a jig that would make the coil winding process easier. It started toff as a simple bridge like object, with one of its tips holding the bought spool, and the other tip with the coil I wanted to wrap the wire onto. The middle would be used to tension the wire to give me a clean winding.

Inspired by videos on YouTube, I also decided to add a counting device last minute - the calculator. The way it worked was that a magnet on the square spool would close a reed switch when it passed by. This would complete the circuit with the "equals sign" button, meaning that if you pressed "+" "1", then it would add 1 to the sum every time the spool made one full rotation. I debounced the switch with a small capacitor This allowed me to keep track of the windings.

The sheer size of the project meant it would have been expensive to print. This is why I opted to CNC it with my robotics team's newly acquired machine.

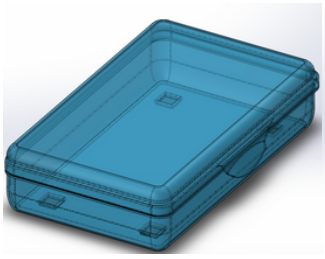


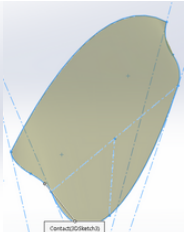
Sterlite Crates - November 2020 & June 2021

Wanting to find a quick, ready built chassis for a gearbox, I settled on a Sterlite crate that you could find at Wal-Mart. I tried sizing and spacing everything properly with calipers and measuring tape to the best of my ability to create the most realistic model. The more realistic they were, the better I could design parts around them. I used linear patterns to get the array of holes for both crates. Later on, in 2021, I would begin to create my transmission out of the mini crate. The parts fit well for the most part, meaning I CADed the crates quite accurately.

Sterlite Pencil Case - November 2020

Being resourceful, I wanted to find a quick housing for a project. I settled on a ready made, fully enclose box, that I could easily open - a pencil case, instead of printing a crate, or building my own (I lack precision, since I have no bench vise, I have to hold everything by hand and feet). It was a mostly simple shape, however, the interesting part was the lip. I had to create a curvy surface in 3 dimensions that was accurate to the lip of the actual crate. This is when I first learned to use surface fill.



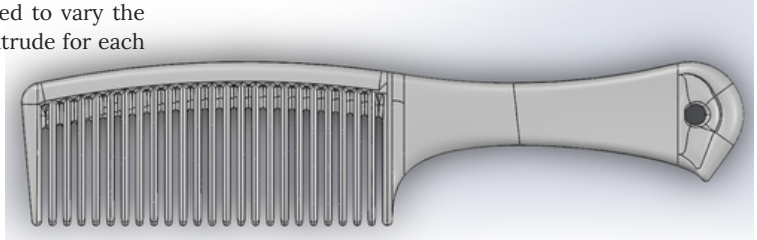


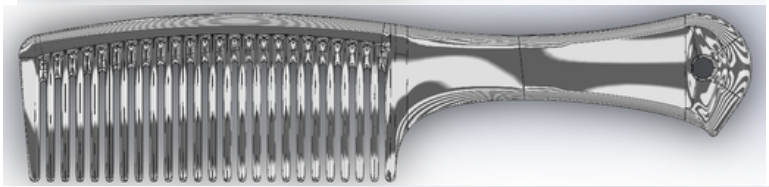
Sleek Handle- September 2020

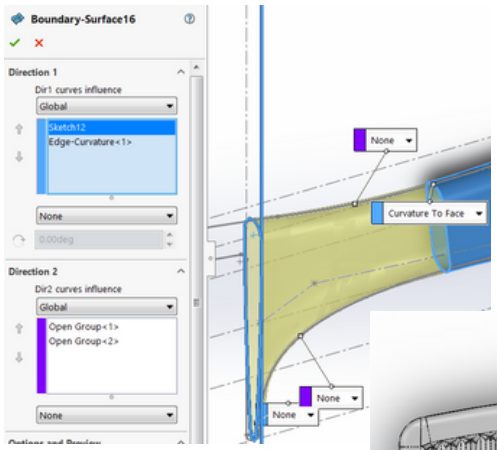
I had just recently started my first semester at CSUF in August 2020, and all classes were online because of the pandemic. Because it ws so early on in the year, the work load was very light, and so out of boredom, I decided to CAD a curvy handle. It was done with simple lofts and mirrors - nothing compared to the surface modeling I would do later. Take note of the shape of the base, and how the "concave" parts smoothly transition to the flat rectangular top.

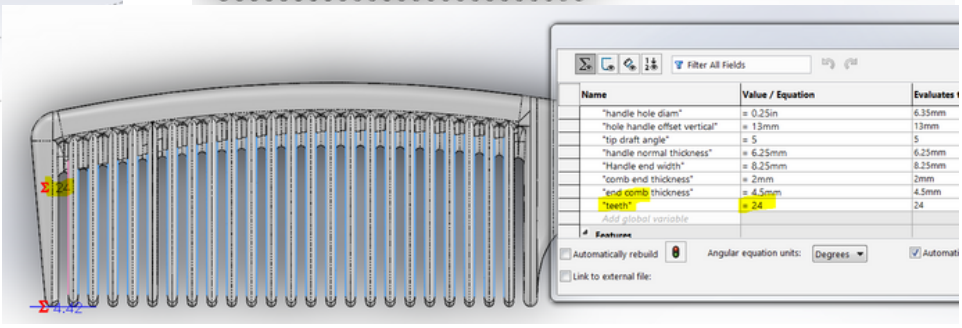
Plastic Comb - June 2021

I had just finished my freshman year on a high note, and wanting to celebrate, I CADed a comb that I had. I was excited for this because of all of its curvy surfaces. I made sure that all surfaces smoothly transitioned into each other, and that there were no sharp edges. I did this using boundary and filled surfaces, this time using the "continuous curvature" option, and I checked my work with the zebra stripes. This project was quite difficult because I had no straight edges to work with, aside from the cross section (if you sliced it down the middle), meaning I had to rely on sketches and splitting boundary surfaces. What made it more difficult was that I wanted to vary the number of teeth. I did this by linear patterning a cut extrude for each tooth, and then filleting each.





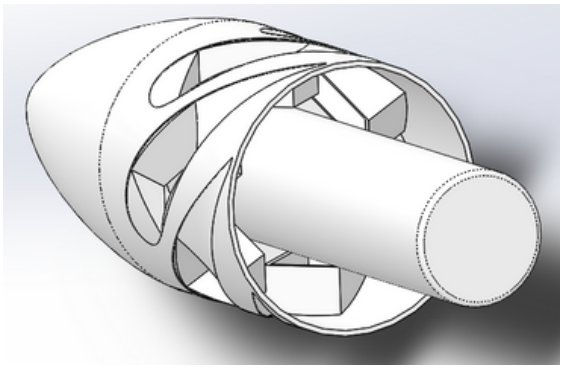


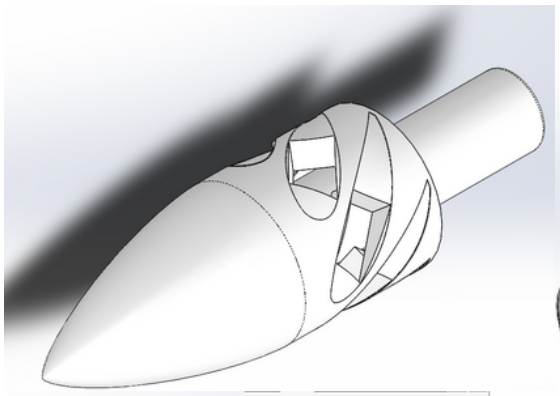


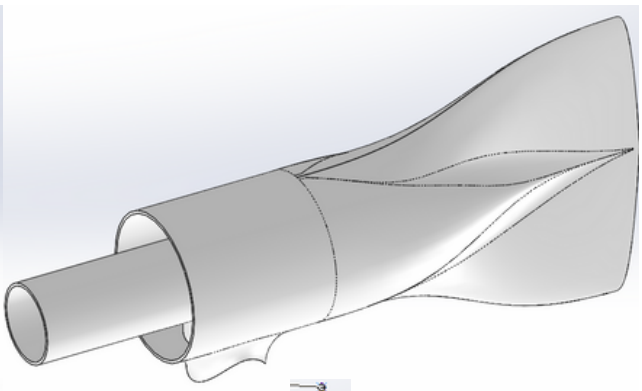
This was more of an art project than anything useful. I started CADing it right after school had gone online, because I thought that I needed more practice making curvy shapes. In April 2020, I had only been CADing for about 10 months, and so my only exposure to CADing anything sophisticated and curvy was with lofts, and so this project had no surface modeling. I also did not know about turning tangent edges phantom, so I had to eyeball if surfaces transitioned smoothly into each other. I was wrong in assuming that using smooth guide curves with lofts will ensure a smooth transition. Had I known about using filed and boundary surfaces, I would have used them, since they are more appropriate for a project like this.

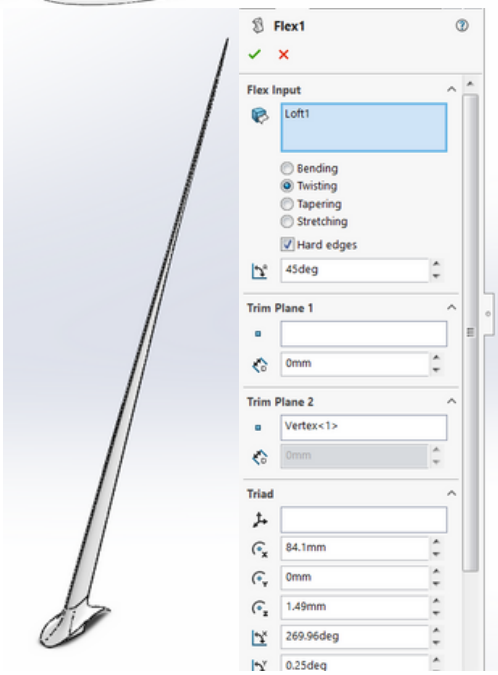
Copying from my rocketry fins, I used an asymmetrical airfoil for the base of the blade, and lofted it into a single point in the form of a rounded edge. I also used a Haack shaped nose for the front of the turbine.

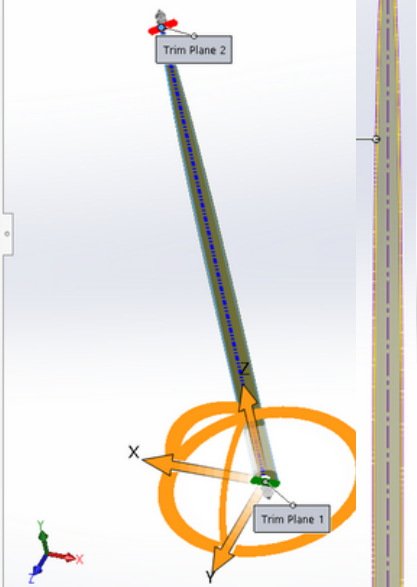
This was also the first time I used the flex feature in SolidWorks to smoothly cant the blade 45 degrees over distance.

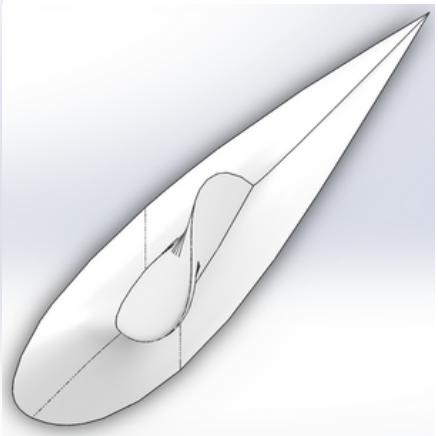




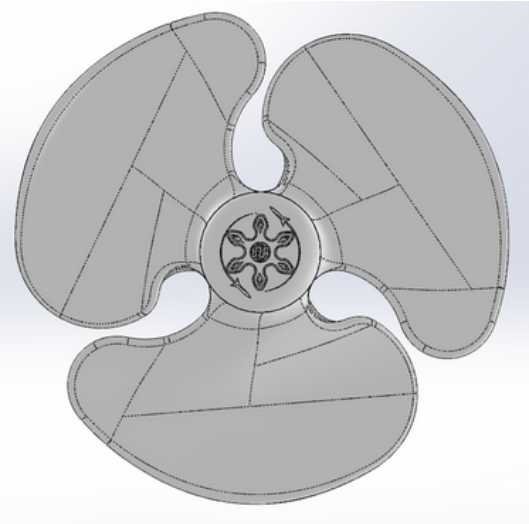








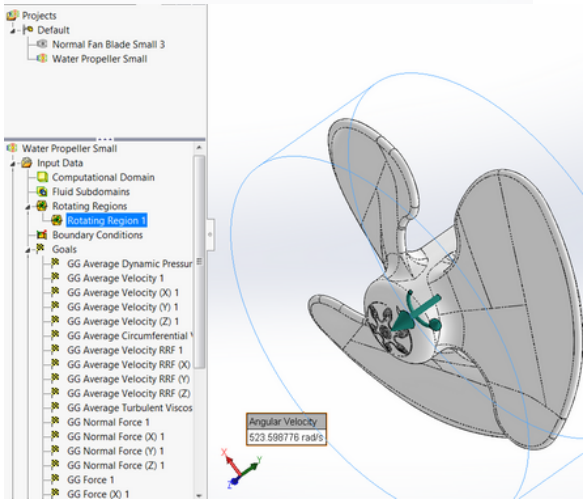
FAN BLADES — MARCH & APRIL 2021

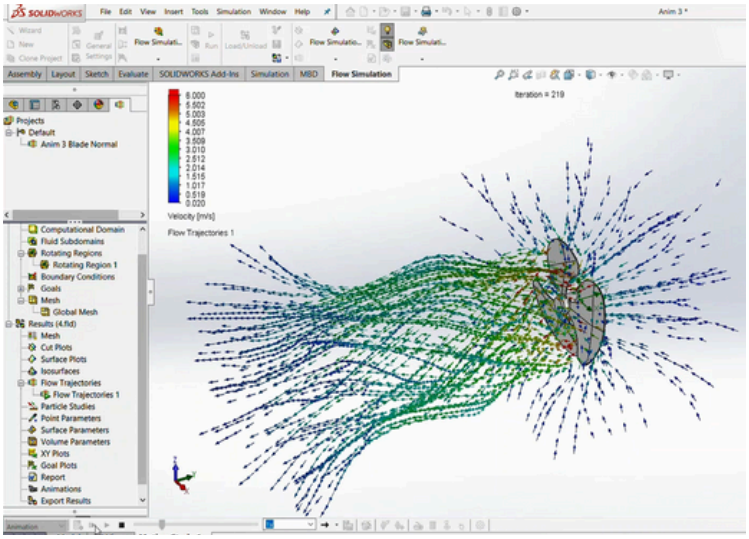


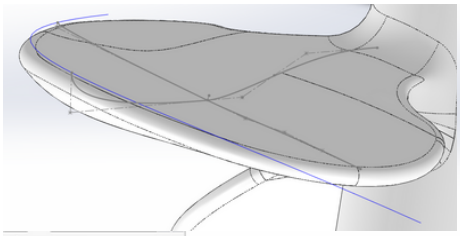
The successor to the wind turbine from a year before, this fan blade is actually smooth all around, and has no sharp edges. This was the first project where I began using boundary surfaces, and where I learned about what the "Contact C0", "Tangent C1", and "Curvature C2" options meant when using surface fill or boundary, and since then, it has helped me immensely. For the first time ever, I used phantom tangent edges and zebra stripes to verify the smoothness, and I was properly able to thicken the fan blade with a variable.

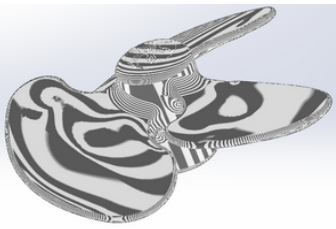
I also properly used rotating regions in this project, simulating the airflow if the fan spun, generating beautiful graphics. I personally find the airflow paths beautiful.

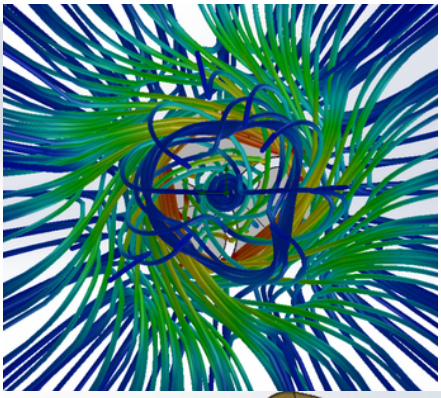
In this project, I also explored the plastic injection molding from SolidWorks Plastics out of curiosity, and I was able to simulate how plastic would flow had I injection molded the fan.

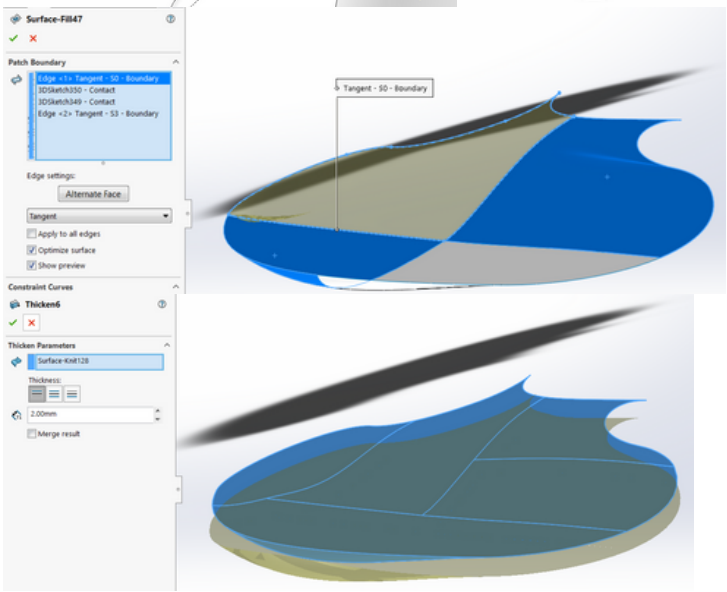


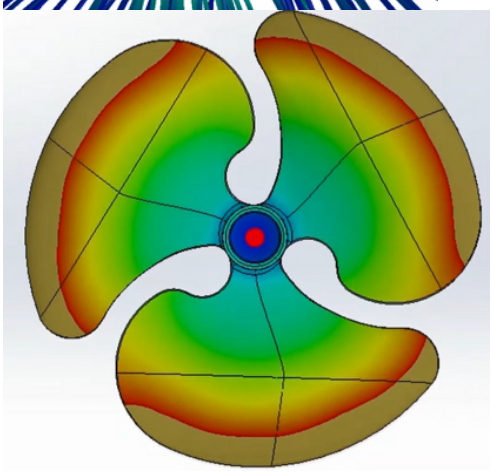






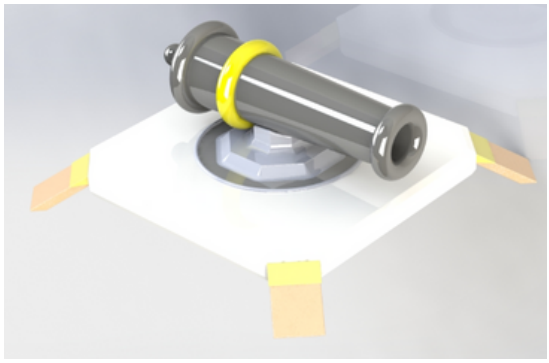




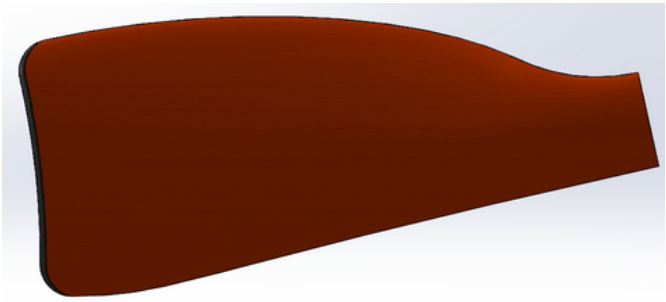


CLASH OF CLANS LEVEL 7 CANNON — OCTOBER 6, 2019

By this point, I had SolidWorks for about two months. I had CADed mostly realistic objects, and practical things for my robotics team and for myself. I decided to have a little bit of fun by CADing a cannon from one of the games I played - Clash Of Clans. This project wasn't as steep of a learning curve, as I had already learned to use extrudes, revolves, and lofts. I even learned to use reference geometry prior, allowing me to extrude the legs of the cannon at off angles. However, this was the first project in which I changed materials and appearances, and where I learned to use the Photo360 option to get a nice result.



MOSIN NAGANT BUTTSTOCK — MARCH 9, 2021

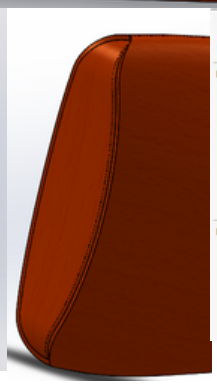


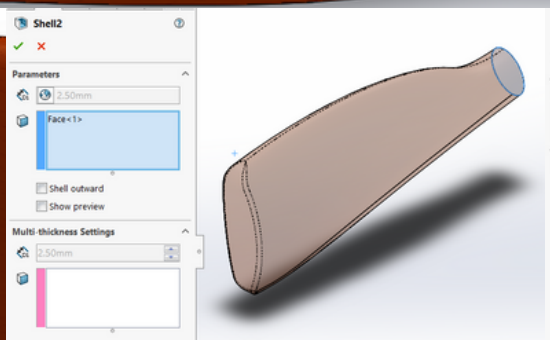
I wanted to CAD curvy objects again after designing many blocky projects. I was getting tired of the extrudes and linear patterns, and I wanted to get back into lofting and making some very aesthetically pleasing things.

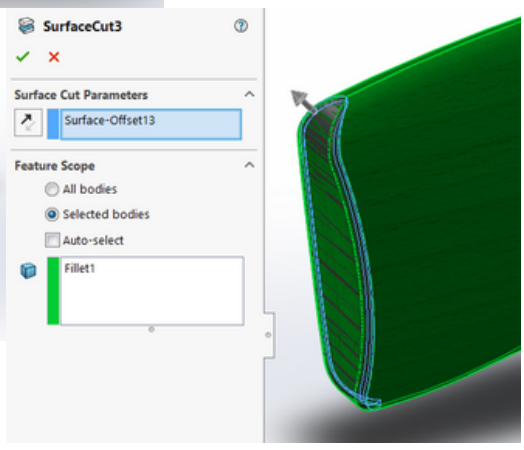
I settled on a Mosin-Nagant buttstock, since compared to other rifles of its time, had the curviest and most complex-shaped buttstock. I thought this would be a nice challenge, and a good way to get back into curvy objects. I wanted to achieve the curves with lofts only- avoiding any fillets. This would prove to be time consuming.

At this point, I had not done any extensive surface modeling, and so I still relied on using lofts. What made this difficult was lofting the butt plate in the rear to the elliptical front, since the butt plate was a 3D sketch. I ended up using four guide curves, and in the end, and even without knowledge of using phantom tangent edges, zebra stripes, and boundary and filled surface C1 and C2 conditions, the project ended up with no sharp edges (except for the abrupt angled "cut" at the front with the ellipse.)

Creating the butt plate proved to be even more difficult. I easily created its shape using surface offsets and thickening it, but the cutaway for the butt plate ended up being the real challenge I ended up shelling the entire object, leaving only the faces that would have made the butt plate. I then ended up using surface offsets and then a surface cut to cut away a space for the butt plate to fit snugly on the end, since cut thicken wouldn't work.

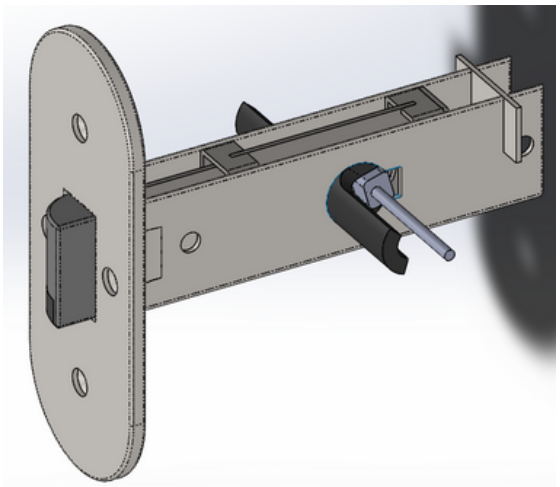






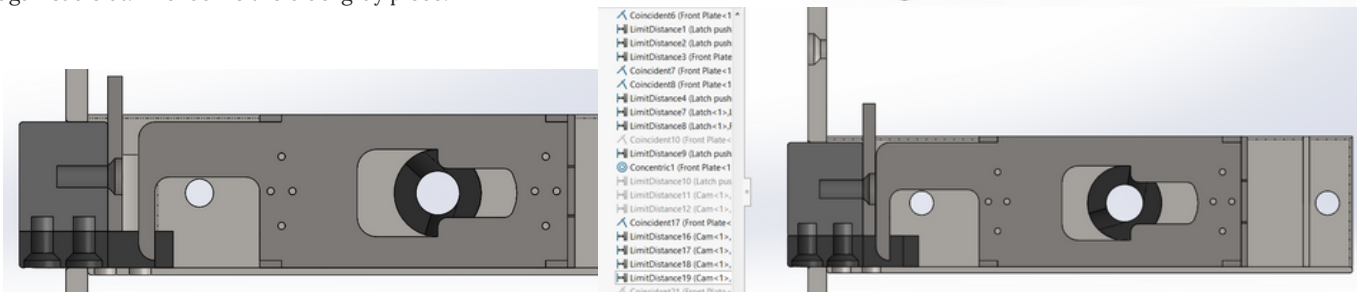
DOOR LATCH — JUNE10, 2020

When my senior year of high school unceremoniously ended during the pandemic, I decided to make my own door latch mechanism out of boredom, and because I recently had to fix my own door. I noticed that my door had plastic torsion springs, which made trying to fix it difficult. Instead, I managed to get the same spring action by using a pen spring inside the latch instead. It worked, and deciding that my idea of using a compression spring was simple than using torsion springs to get the back and forth motion of the latch, I designed my own in SolidWorks.



The latch works by having a compression spring in between the back wall and the dark gray piece, forcing it out. When the black piece is cammed (by the knob), it will force the dark gray piece back, compressing the spring, and pulling the latch back. I achieved the camming action with limit distance, as limit angle wasn't working properly in both directions.

The lock works by simply preventing the camming action by jamming it against a slot. The lock is the blue-gray piece.



VARIOUS HARDWARE COMPONENTS — JUNE-DECEMBER 2020

In manufacturing, sometimes it is easier, or stronger to use off the shelf parts instead of making and printing your own. This is what I did when I needed hinges, or if I needed to improvise a cheap part for the job.

The only problem with CADing these parts is that sometimes, there is no model available online for them. This is especially true for off the shelf components from hardware stores such as home depot. It is important to have an accurate model of any part, even if its cheap and off the shelf, so that it can be planned and CADed around.

This is what I did with things like torsion springs, sheet metal parts, and hinges. I measured each part with a caliper to the best of my ability to get accurate dimensions, and tried placing the holes as accurately as possible by measuring the distance of its edge to the edge of the sheet.

For the sheet metal parts, especially the curvy ones that I bent, I didn't use the sheet metal add in, as I had to worry about minimum curvature radius. I instead just used lofts to achieve those shapes.

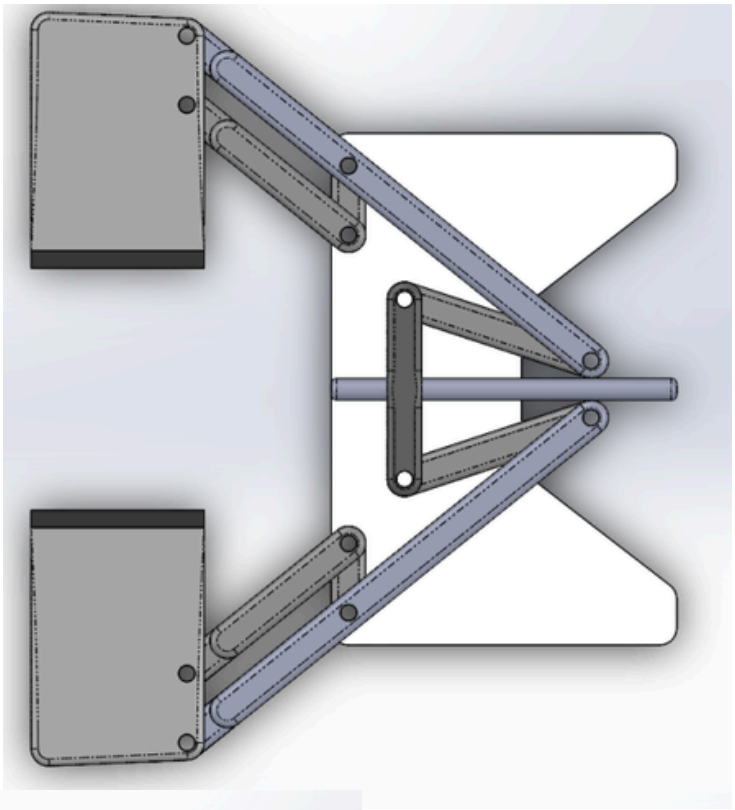
In the hinge assembly, I was finally able to get the limit angle mate to work, and the hinge would stop opening or closing if rotated in or out enough.

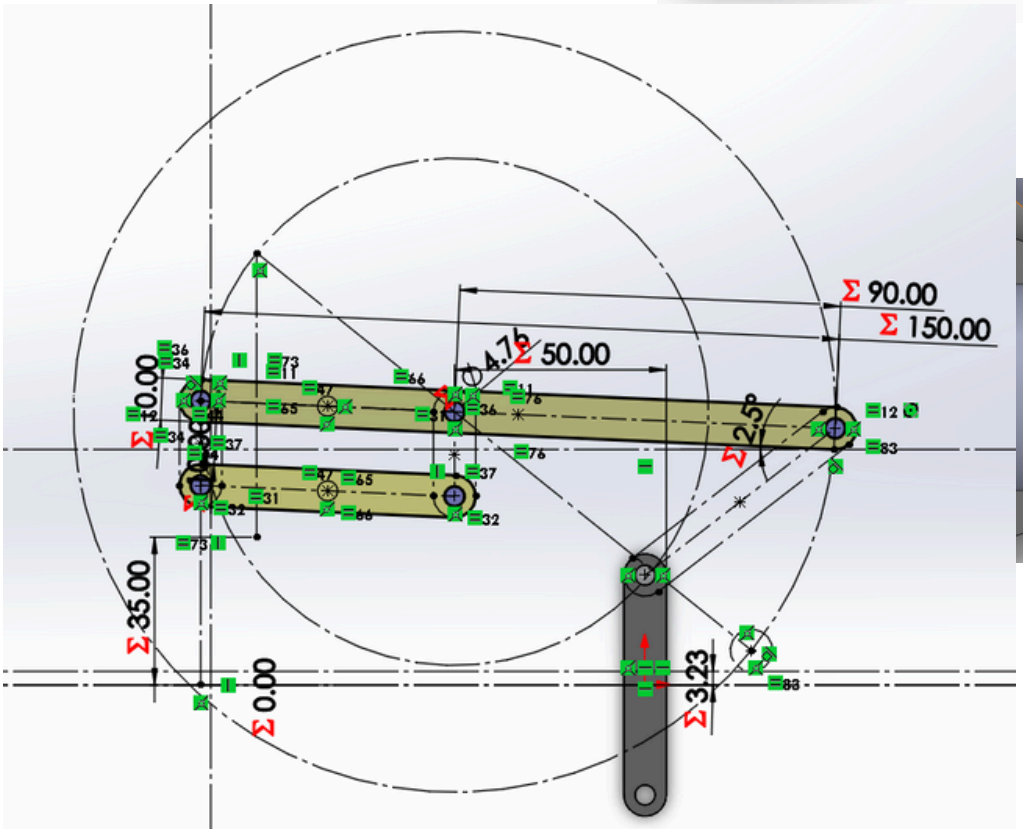
In fall 2021, I joined the Titan Rover team at my school, CSUF. On the rover was a mechanical gripper that needed to be redesigned. Although I had not taken a class on 4 bar mechanisms, and did not have a good intuition of them, I decided to try anyway.

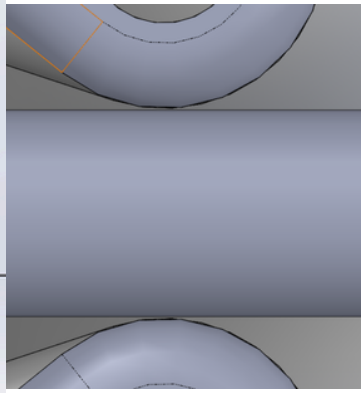
Instead of wanting to guess or randomly input the length of each bar of the mechanism, and then finding out its motion experimentally in the assembly, I decided to CAD all the parts at once, saving each as a configuration.

In the sketch, I had defined the length of each arm by its shape in resting form, and other parameters, such as its minimum width when the gripper is closed all the way. I also ensured that the bars would not hit the actuator rod when fully closed, using a variable to input a clearance. I also allowed the location of the joints to change, giving different mechanical advantages.

In the end, I built a parallel 4 bar gripper mechanism in 5 hours. Sadly, they decided not to change anything by the time I finished, because they were on a time crunch, and could not afford the time to redesign anything.



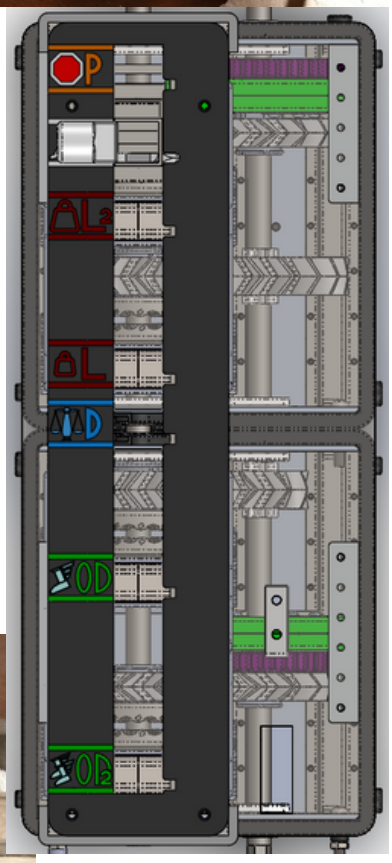


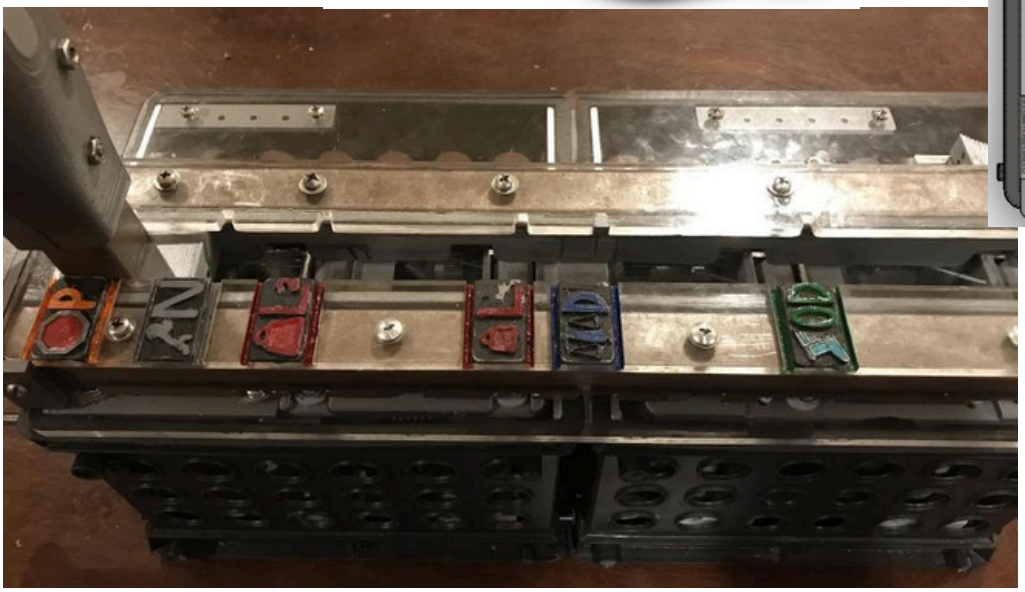


GEAR SHIFT TRANSMISSION — JUNE 2021 - MARCH 2023

From June 22 2021 to March 26 2023, I worked on a gearshift. This project really was the culmination of all my knowledge and experience, using everything I learned from the beginning of my journey. I extensively used variables and patterns - something I learned at the start of my time with SolidWorks. I defined almost all parts of the transmission within the main assembly, meaning if I change one component, all other components will conform to that change - something I learned to do with the Stirling engine project. I used the mini Sterlite crate from my Plastic Products projects, because the holes there provided precise places for the axles to go through in a nice, boxy frame. I extensively used gears to achieve different ratios - something I learned almost 6 months prior. I did extensive surface modeling to make the ergonomic gearshift knob and button - ensuring that there were no abrupt edges, and that all surfaces had a smooth C1 or C2 connection - something I learned in my long lofting and surface modeling journey. The clutches, which are the core of this project, included surface modeling, variables, and patterning to achieve the ideal geometric shape.

Below, I will go over each part and its workings in detail, and how I made them in SolidWorks.

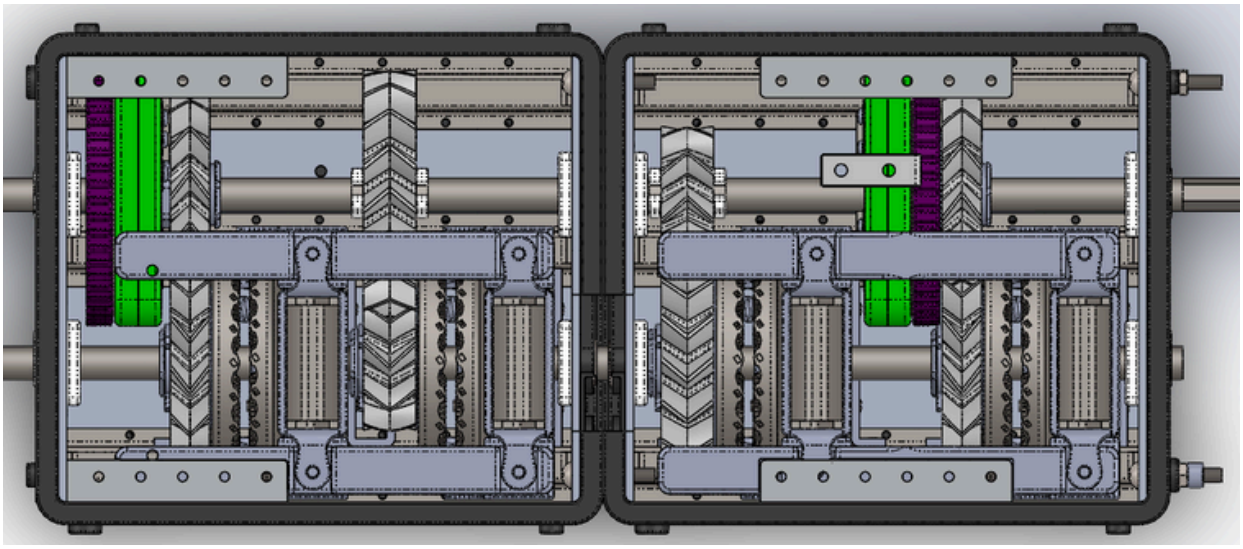




GEAR SHIFT TRANSMISSION - HOW IT WORKS

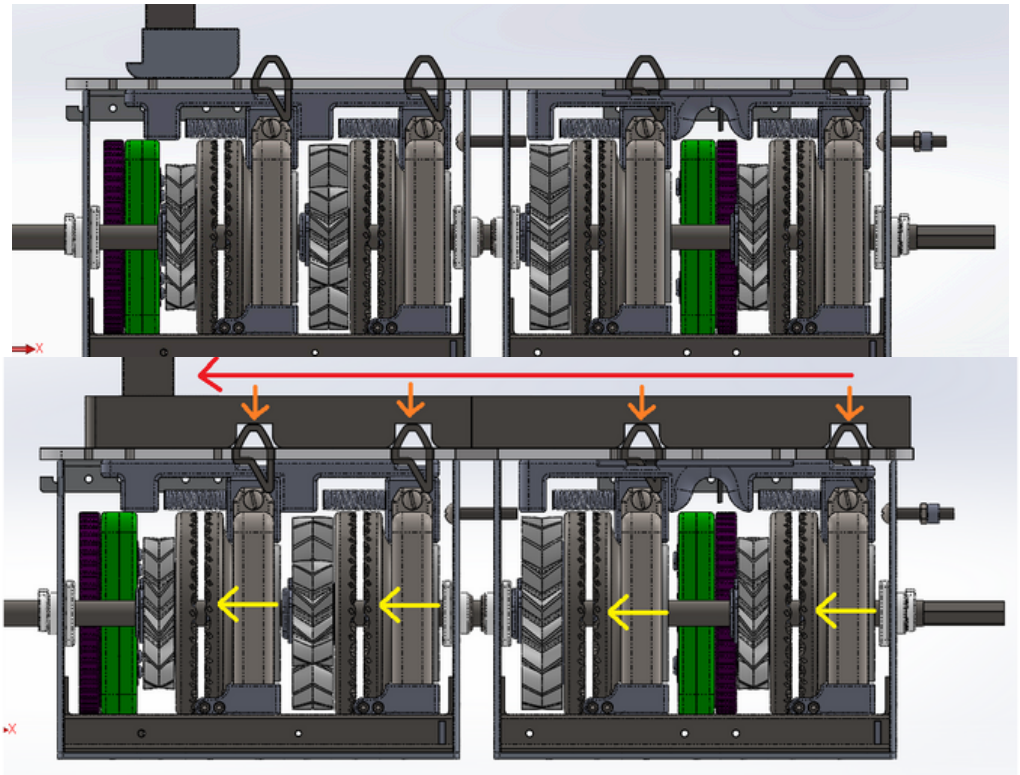
The transmission has two axles. One is an input axle (bottommost), and the other is the output (topmost). The output shaft has gears rigidly attached to it, meaning if the output shaft spins, then all the gears on it will spin as well. The input shaft has gears mounted onto it with needle roller bearings. This means that while the gears are on it, they are not rigidly attached, and so they can slip around the input axle.

On the input shaft are the clutches. Each clutch has two parts: One rigidly attached to the gears (recall that these gears can slip around the axle), and the other face that is locked with the axle. These clutch faces can slide axially, but rotate along with the axle (Later, I will show that there is a delay in rotation because of the shock absorbing springs). When moved forward, the clutches engage, locking each other. Because one clutch was locked with the rotation of the axle, and now both clutches are locked with each other, both clutches are locked with the axle, and spin as one unit. Because the clutch face connected to the gear is locked in with the axle, the gear no longer slips around the input axle, and spins along with it. Now that the engaged gear on the input shaft spins, it spins the gear on the output axle, and a ratio is achieved.

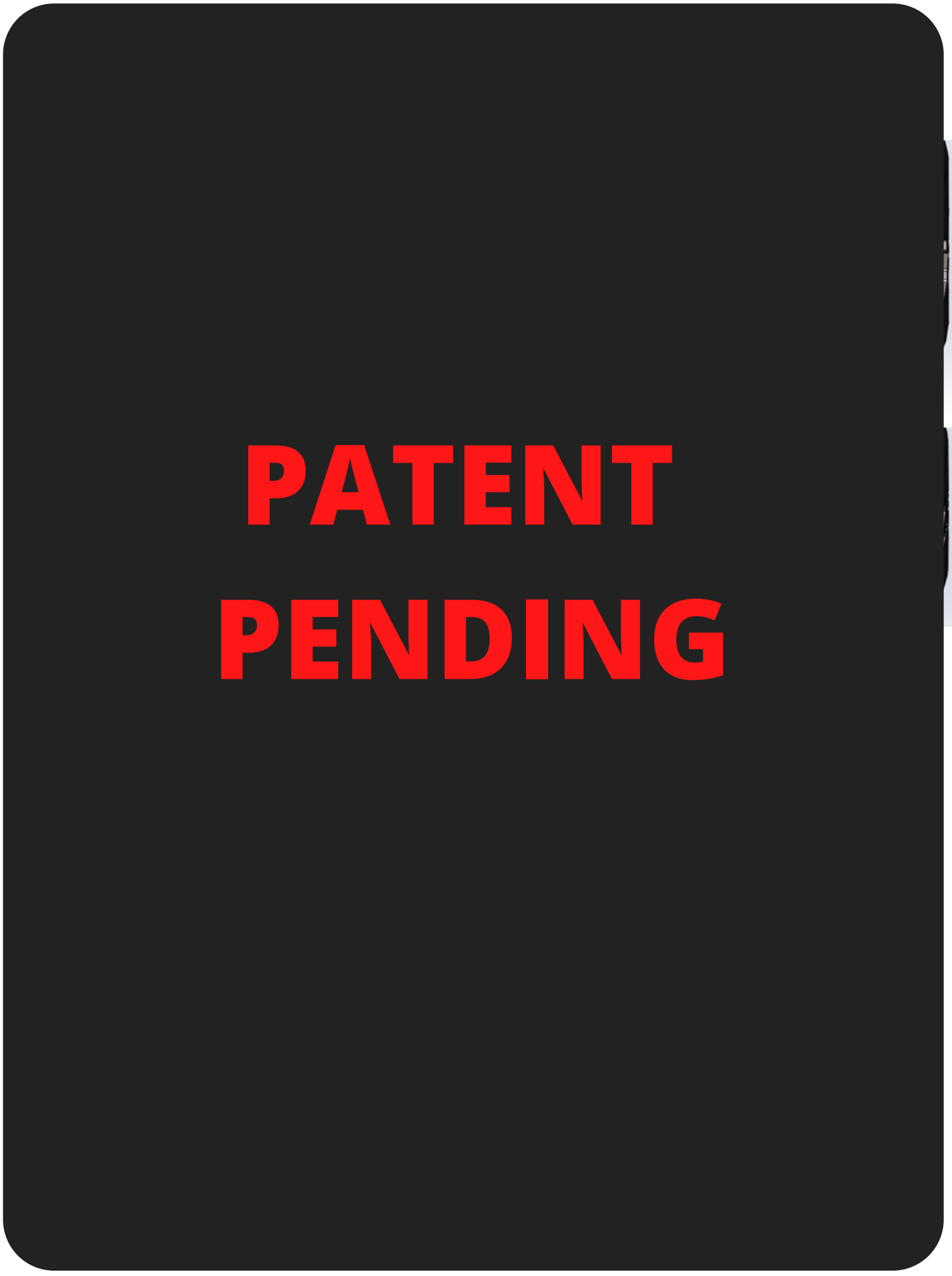


Each clutch is only allowed to move forward to lock in. This is achieved by using wedges, with angled bottoms. The face of the clutch that is allowed to move is housed in "carriers", which have an angled top. When the wedge moves down, it will force the carrier to move forward, engaging the clutch. The top of each wedge is shaped like a door latch. This is to allow the gear shift stick to slide smoothly over it. When the stick is slid over the wedge, it will force the wedge to move downwards. As long as the stick slides over the clutch, whether coming from the left or from the right, it will depress the wedge. Because all that is required for the clutch to engage is for the wedge to move down, it doesn't matter which direction you shift the stick.

Parking works by engaging all clutches at once. Because an axle can't spin at more than one speed at a time without shearing or deforming, by forcing all clutches to engage the gears, the gears will try to do this exact thing. This causes the entire gearbox to lock.



PATENT PENDING

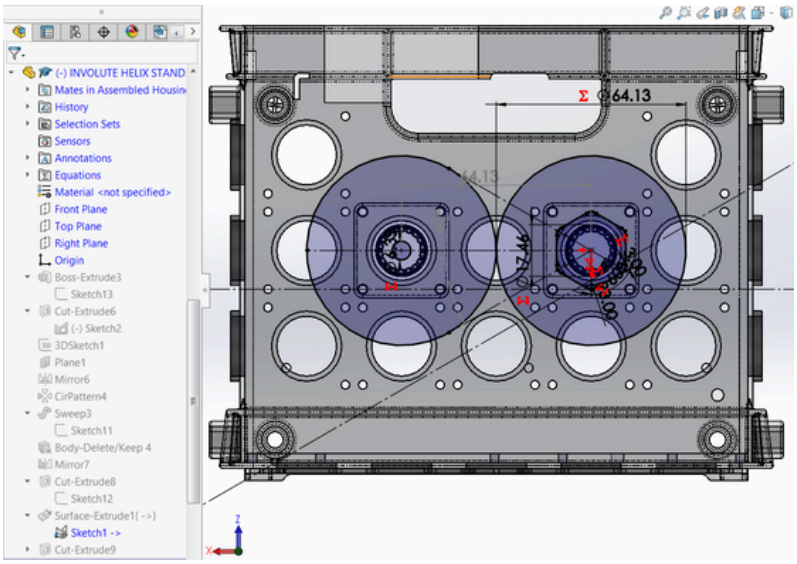

PATENT
PENDING

GEAR SHIFT TRANSMISSION - THE GEARS

All gears are defined by five things: Pitch Diameter - the imaginary circles that roll on each other, teeth number, pressure angle - the angle at which teeth collide, and addendum and dedendum - how far the tooth extends past the pitch circle. (In most cases, Pitch Diameter and teeth number are grouped into Module). If those three inputs are defined, everything else gets defined: The base circle diameter, the tooth width and shape, and spacing.

For this project, because the distance between the holes of the crate weren't nice numbers, I decided to define the pitch diameters within the assembly itself. This way, the pitch circles of each gear would be centered on the axle hole, and their pitch circles would be tangent. I then fully defined the pitch circles by setting their diameters to be the distance between the centers multiplied by the ratio of teeth.

In the picture to the left, the ratio between the gears is 1:1. I used dimension between the distances (which was driven), and multiplied it by 12/12, because each gear had 12 teeth. For the other gears, I multiplied the distance by 18/9.



The ratios are as follows: (input : output)

1:3 , 1:2 , 2:1, 3:1

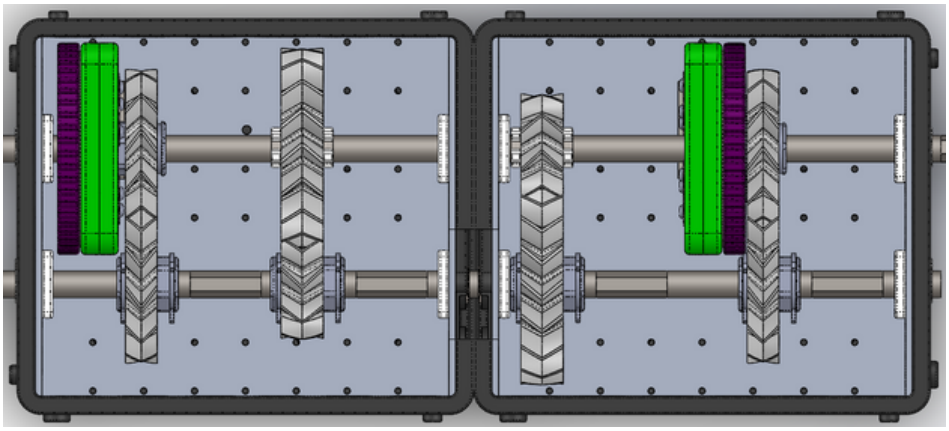
Sadly, I could not fit a direct drive with the space I had. (although I could fit parking).

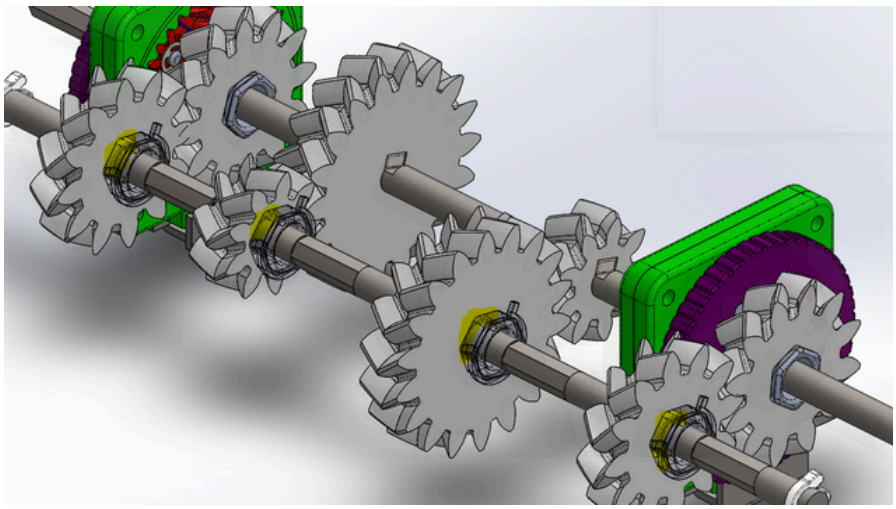
The 1:2 and 2:1 gear ratios are straight forward. The ratios are achieved with a simple gear pair. However, for any higher ratio, the small gear becomes so small, that it wither won't fit around the axle, or the axle bores too big a hole through it. This is why for the 1:3 and 3:1 ratios, I used planetary gears.

The 1:3 and 3:1 are achieved using planetary gearboxes, with the ring held.

In the 1:3 gearbox on the far left, the sun gear is the input gear, which then outputs the purple carrier. The carrier is then attached to the output axle. The sun gear is driven by the 1:1 ratio. Multiplying the ratios out - 1/1 * 1/3, the output is a 1:3 reduction.

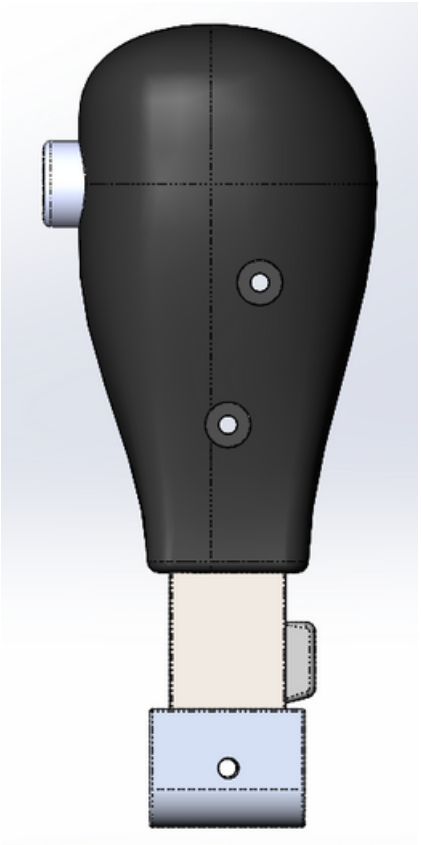
Similarly, the 3:1 gearbox is driven by a 1:1 gear ratio. The gray gears are then attached to the carrier, which then outputs the sun gear. The sun gear is connected to the output. Multiplying the ratios: 1/1 * 3/1, the output is a 3:1 overdrive.





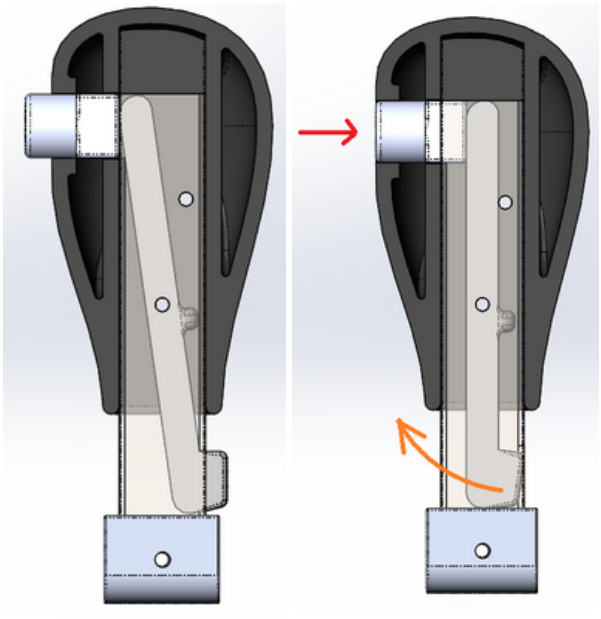
GEAR SHIFT TRANSMISSION - GEAR SHIFT MECHANICS

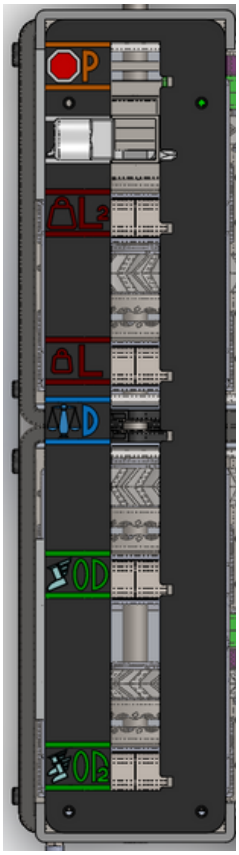
The gear shift knob is a very simple part. Because the gears are engaged by depressing the wedges, all that is required is that something presses them down. This is what the gear shift stick does. The bottom piece is called the "depresser". It has large arcs on its front and back to allow it to slide smoothly over the wedges, and has a large flat bottom to ensure that the wedge stays depressed. All that is required to shift to the correct gear is to ensure that the depresser slides over and stays over the correct wedge. This is what the latch is for.

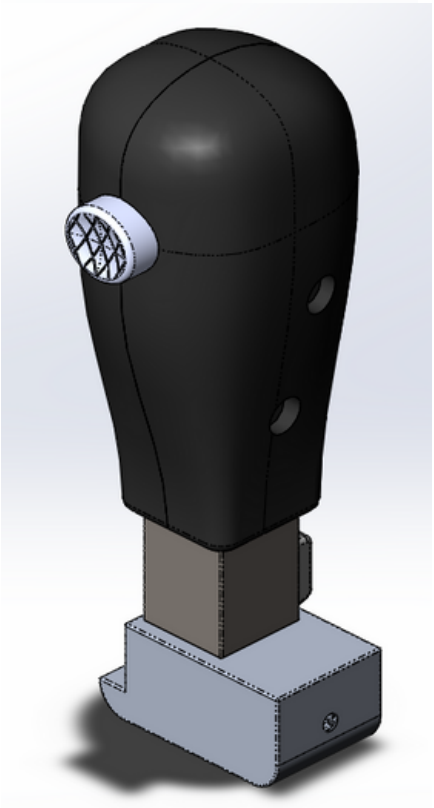


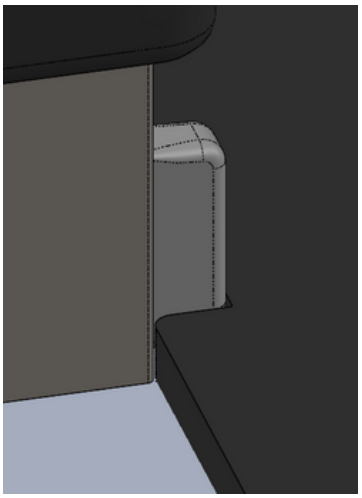
In the gear shift plate, there are notches on the right hand side of the large rectangular slot. These notches are positioned to their corresponding wedge. The gearshift stick is held in place over the correct wedge because of the latch and notch. This is all the gear shift stick does - slide and latch.

Because of this simplicity, the gear shift stick only has two moving parts: The latch, and the button. The latch is actually a larger piece, and is a lever. This lever is hinged on a screw, and tensioned by a torsion spring. When not pressed, the torsion sping will naturally rotate the lever so that the latch is out, thereby locking the stick into the notch. When the button is pressed, it pushes and rotates the lever so that the latch comes inward, allowing the stick to be moved again. The button doesn't need its own spring, because when not pressed, the lever, pushed by the torsion spring ensures that the button will be pushed out.







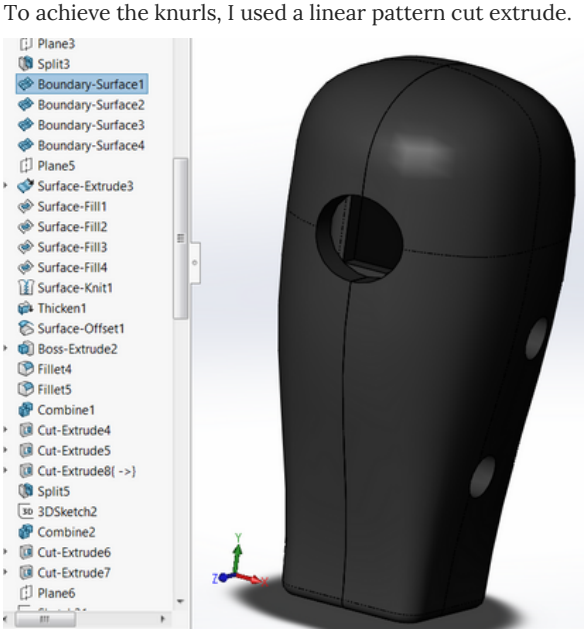


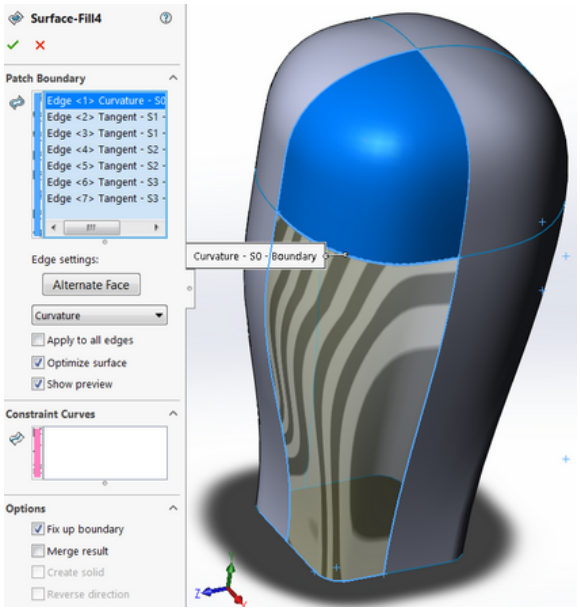
GEAR SHIFT TRANSMISSION - GEAR SHIFT KNOB AND BUTTON

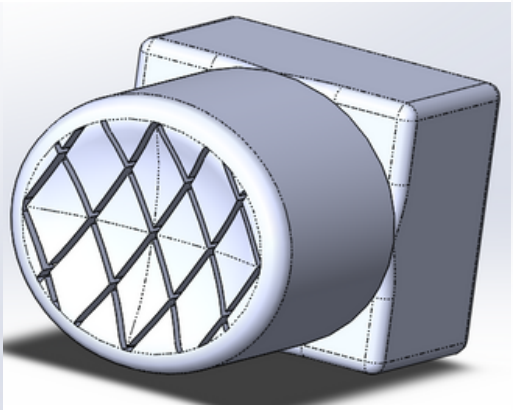
To achieve an ergonomic design for the gear shift knob, I used extensive surface modeling. I used many boundary surfaces and filled surfaces to achieve this, and I made sure that each surface was a C1 or C2 curve, and that there were no sharp edges on the model.

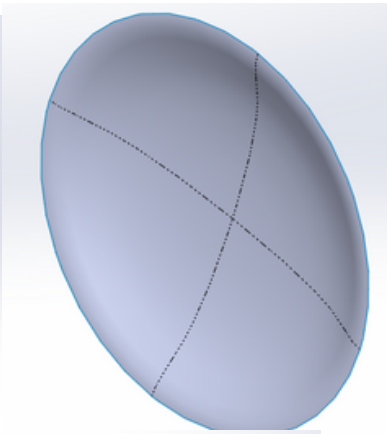
On the button, used another boundary feature to get the dimple. I wanted to make sure that the button was comfortable to press, and so I went with a surface that gently curved in towards the center.

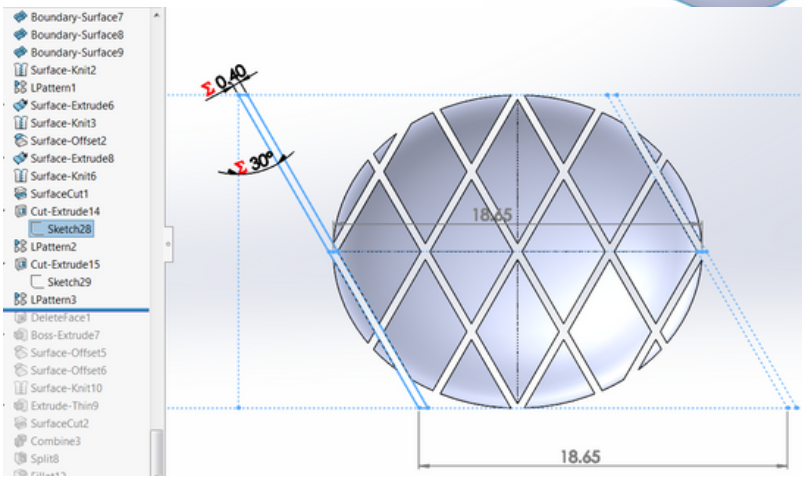
To achieve the knurls, I used a linear pattern cut extrude.



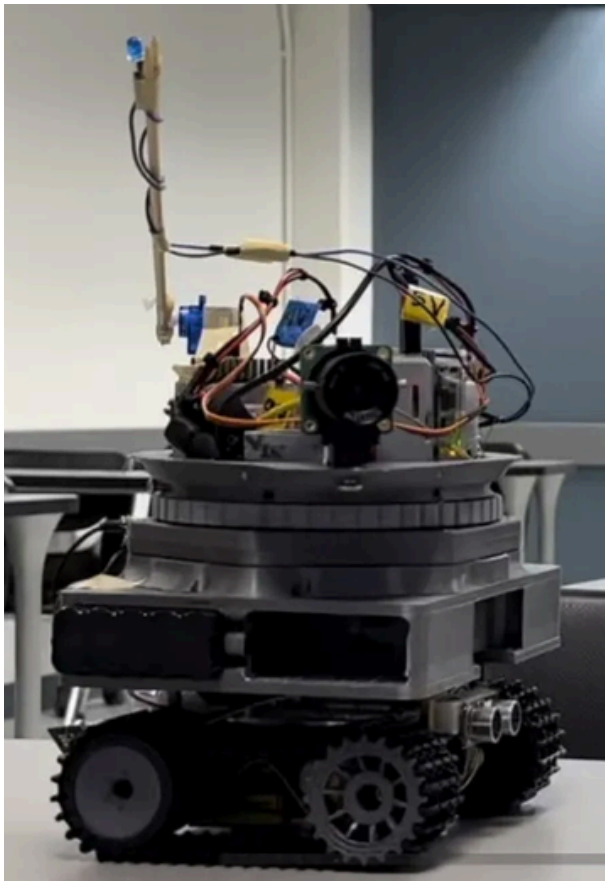








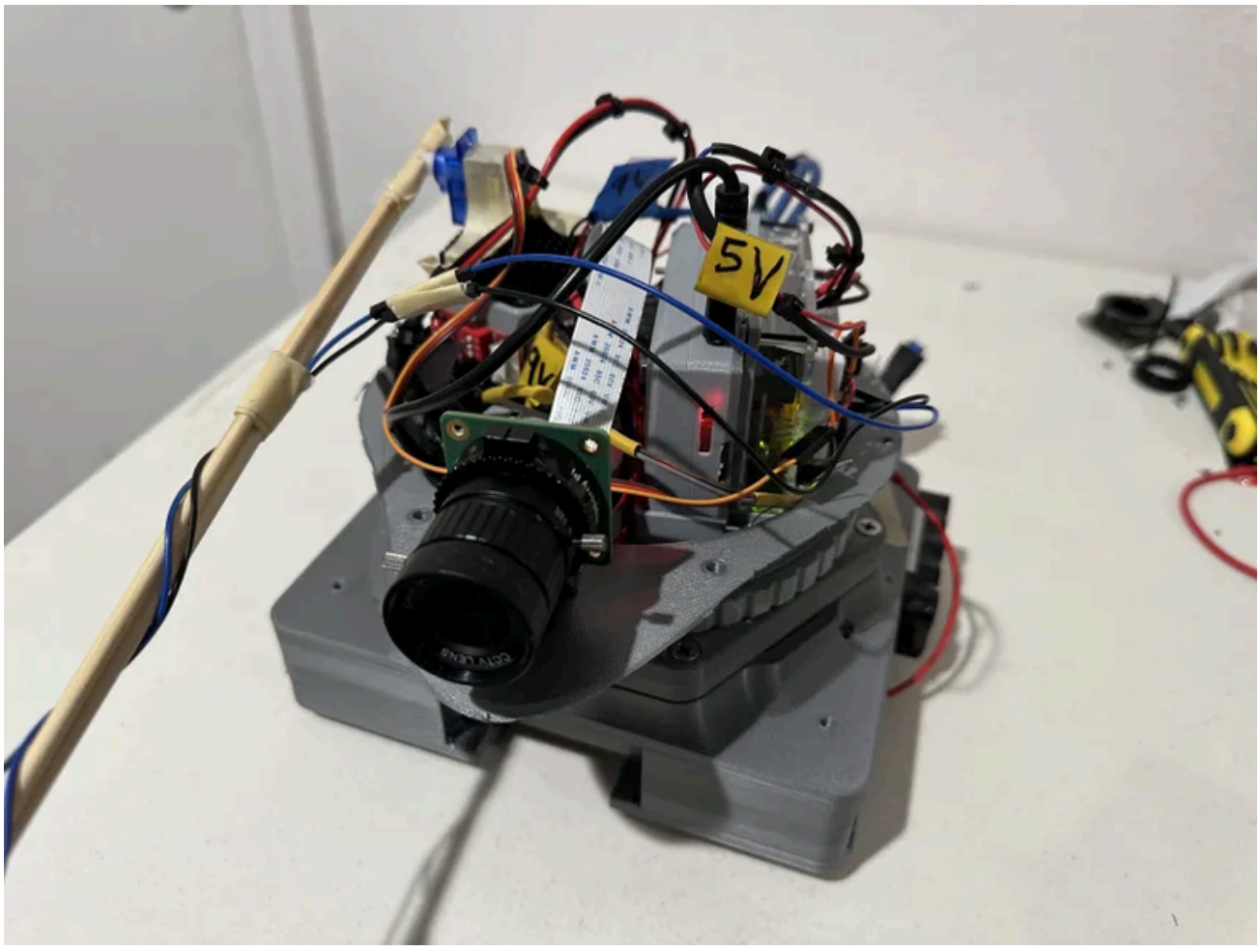
AUTO-AIMING TURRET — JANUARY - DECEMBER 2023



This was my first time experimenting with a Raspberry Pi, serial communication, Haarcascades and later LBPcascades. It was my first real experience in combining my programming experience with my hardware experience, and interfacing the two. While the CAD and mechanical innovations were easy, the low level computer science concepts offered a much steeper learning curve.

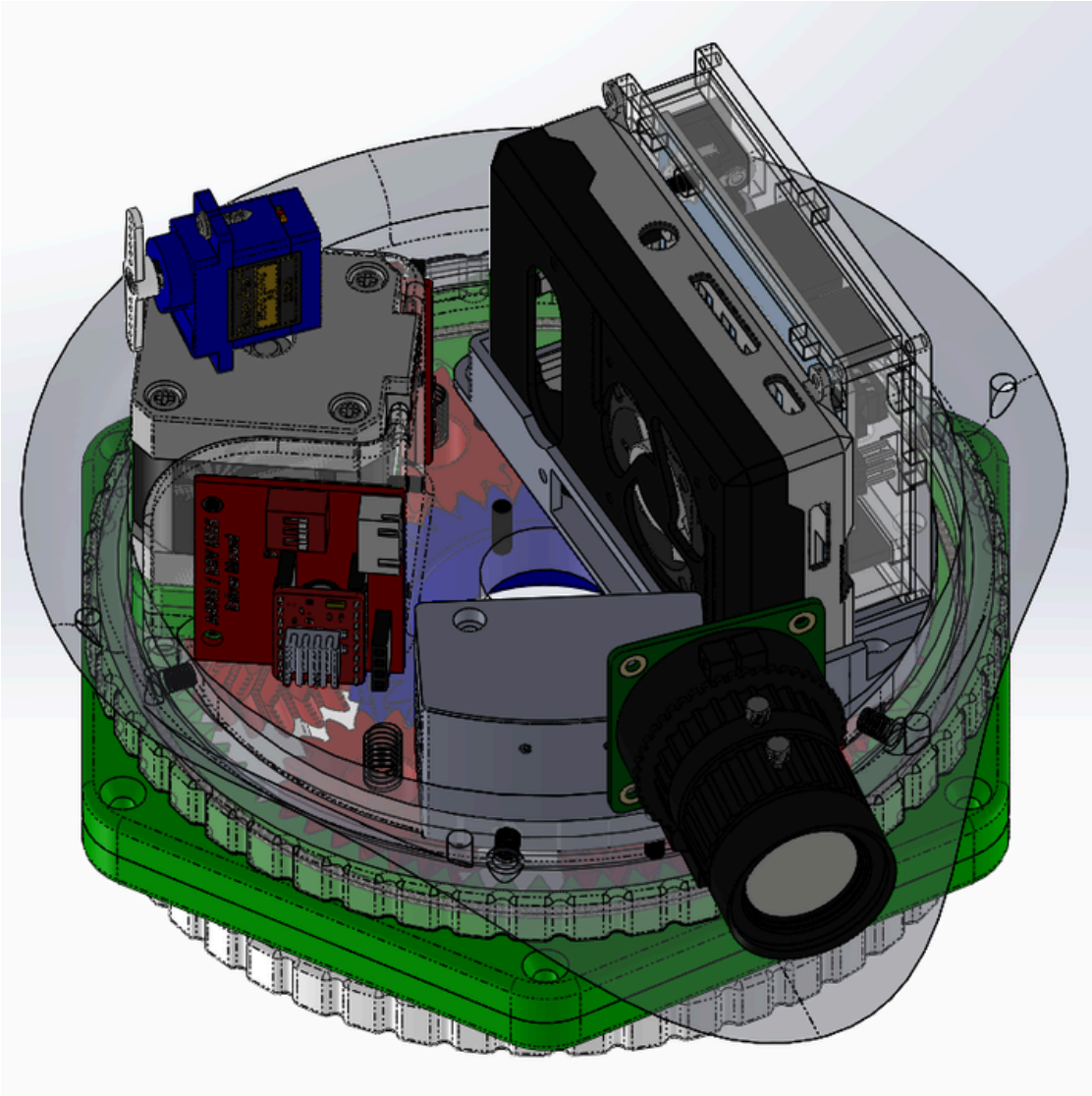
The Auto-Aiming turret uses a Pi-Cam, to gain visuals, a Raspberry-Pi, to do image processing and coordinate fining, and an Arduino, which receives instructions from the Pi, and then commands a stepper motor, moving the turret. The turret will aim for the nearest face, and will try to center its view with the target. The farther the face is from the “centerline” of the camera view, the faster the turret moves towards it.

This project took nearly an entire year, and most of the time was really spent learning concepts such as serial communication, different communication protocols, threading, and image buffering. From about January to March, I was still experimenting with serial communication. Then, I started getting hardware and 3D prints done from about June to August. October to December was spent assembling and then finally getting both code and mechanicals to run together.

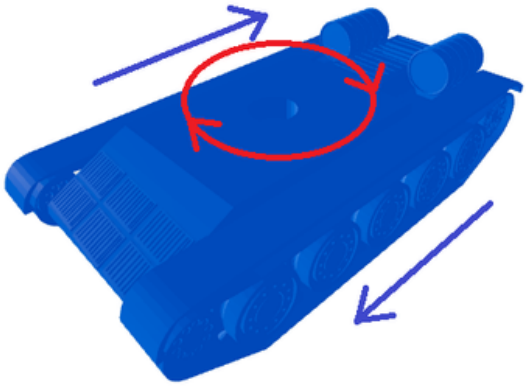


Although I was experienced in Solidworks already, the shear number of parts and redesigns made the CAD take a total of about a month's work. (I had to spread my focus to other things during June - August) The turret has three general subassemblies:

- Movement
  - Turret
    - Ring gear and Planet Gear Base
      - What allows the base to move, and has a 1:3 ratio
    - Turret Base (Planet Carrier in Planetary Gear terminology)
      - Where everything is mounted onto
      - There is a space dedicated to mounting the Arduino and Pi together
      - There is a dedicated "box" for the stepper motor to fit into. Its walls allow an A4988 stepper motor driver to be mounted onto it, ensuring close connections between the stepper motor, and its driver
      - There is a dedicated area onto which to mount the power electronics
  - Motors
    - Stepper Motor, which actuates the planet gear that drives the base
      - The stepper motor is geared down with a 1:4 Planetary Gearbox
- Computers and Microcontrollers
  - Pi-Cam and Raspberry-Pi to get and process images, and then return coordinates of faces onto each frame that the camera got. This information is then sent over to the Arduino
  - Arduino then sends commands to the stepper motor drivers
  - These are housed together, ensuring easy wiring between both components of the "brain" of the turret
- Power Electronics and Motor Drivers
  - Motor Drivers, as mentioned above, are mounted onto the wall of the stepper motor "box"
  - Power electronics are all housed together in a small box (the box has a round wall, and is square everywhere else)
  - If a battery cannot be mounted onto the turret base, there are screw holes that allow a slip ring to be mounted in the center. The slip ring only needs two wires: power, and ground, as all the other signal and electrical connections are on top of the base, and move along with it.



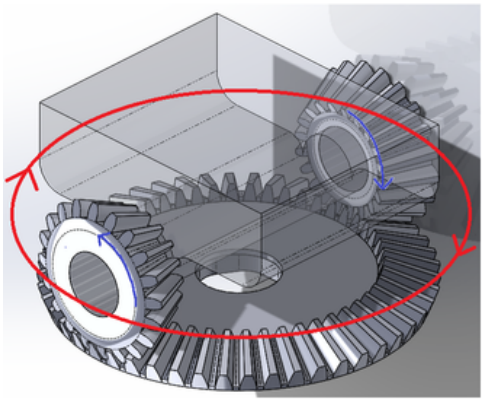
What makes the turret especially unique is that it was designed to never have wires tangle as the turret spins, and can even move past 360 degrees. This is achieved by, as mentioned before, having everything mounted onto the turret base.

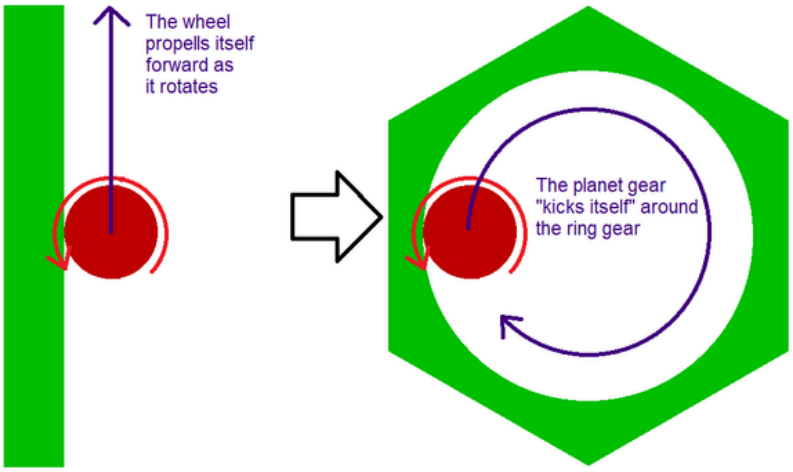


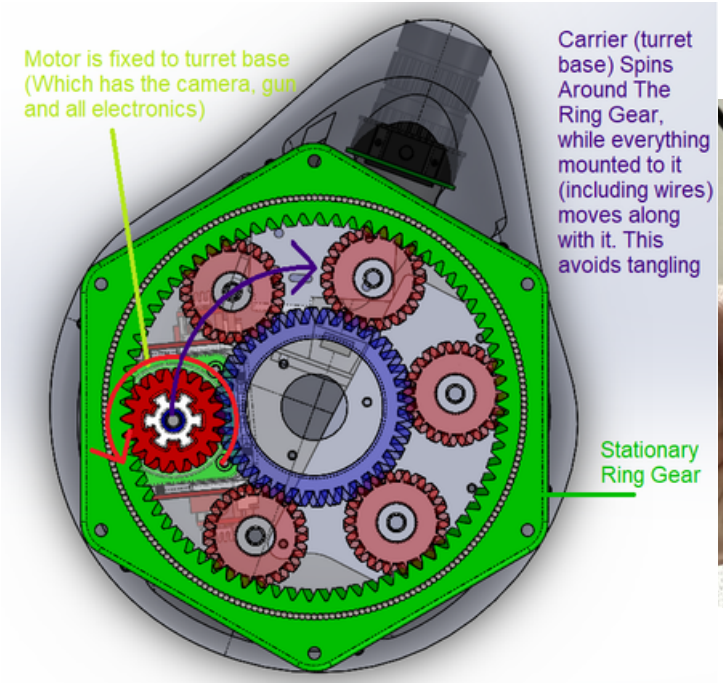
The original concept of this came from imagining a tank trying to turn - in order to turn in place, both tracks must move in opposite directions. I then imagined that instead of wheels and tracks, what if the tank was driven by gears? So I then imagined the ground being a giant bevel gear, and the tracks also as bevel gears. I then realized this could be flattened to a planetary gearset, and then I later realized that because the gears already constrained motion, I only needed to drive one gear.

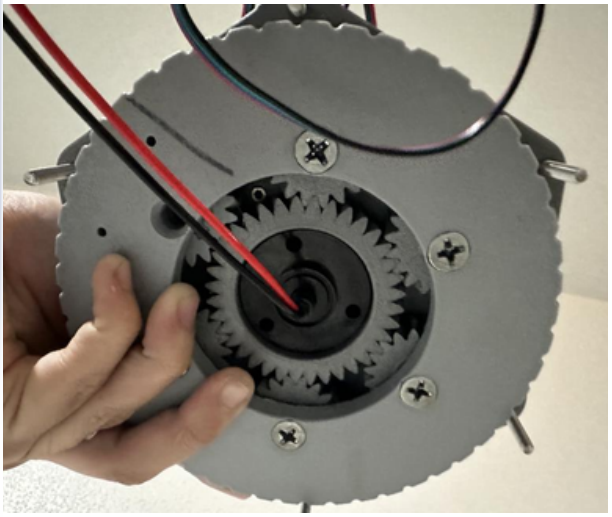
In the end, the turret moves by having its “wheel” (planet gear) kick itself against the “ground” (ring gear) to spin around. And because everything is wired on top of the “chassis” (turret base), the wires move along with it, meaning no relative motion between the ends of the wires (both ends of the wires are on the “chassis”), meaning no tangling.

This gearing also creates a reduction in speed, and an increase in torque. The final ratio is 1:3, in which the planet gear must rotate 3 times in order to make a full revolution around the ring.

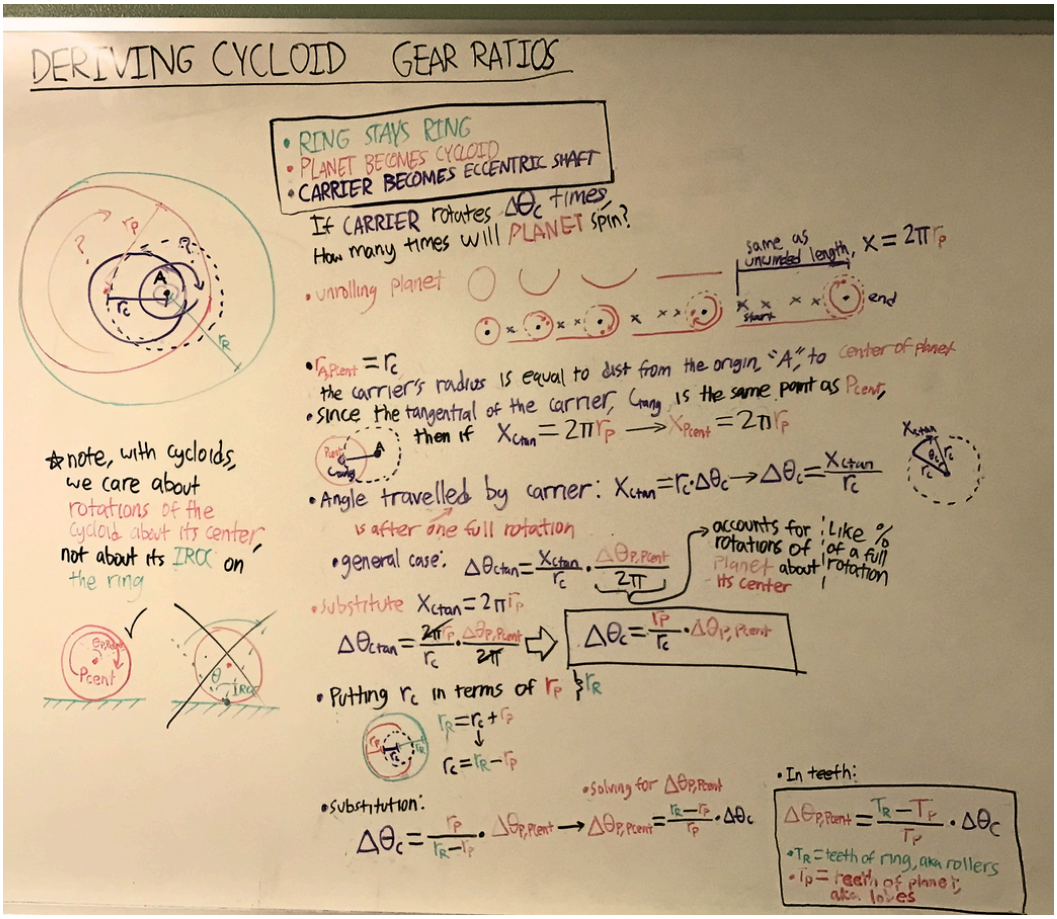








AUTO-AIMING TURRET — RING GEAR BASE CALCULATIONS



Keeping in mind that the Carrier is the turret base:

- $T_R = 72$
- $T_P = 18$
- So then...
- $R_{atio} = \frac{T_R - T_P}{T_P}$

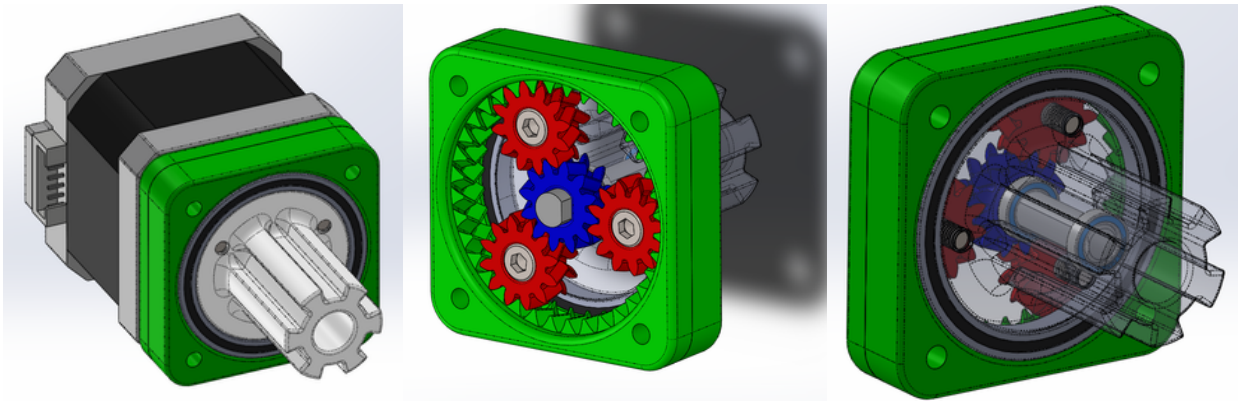
$$R_{atio} = \frac{72 - 18}{18} = 3$$

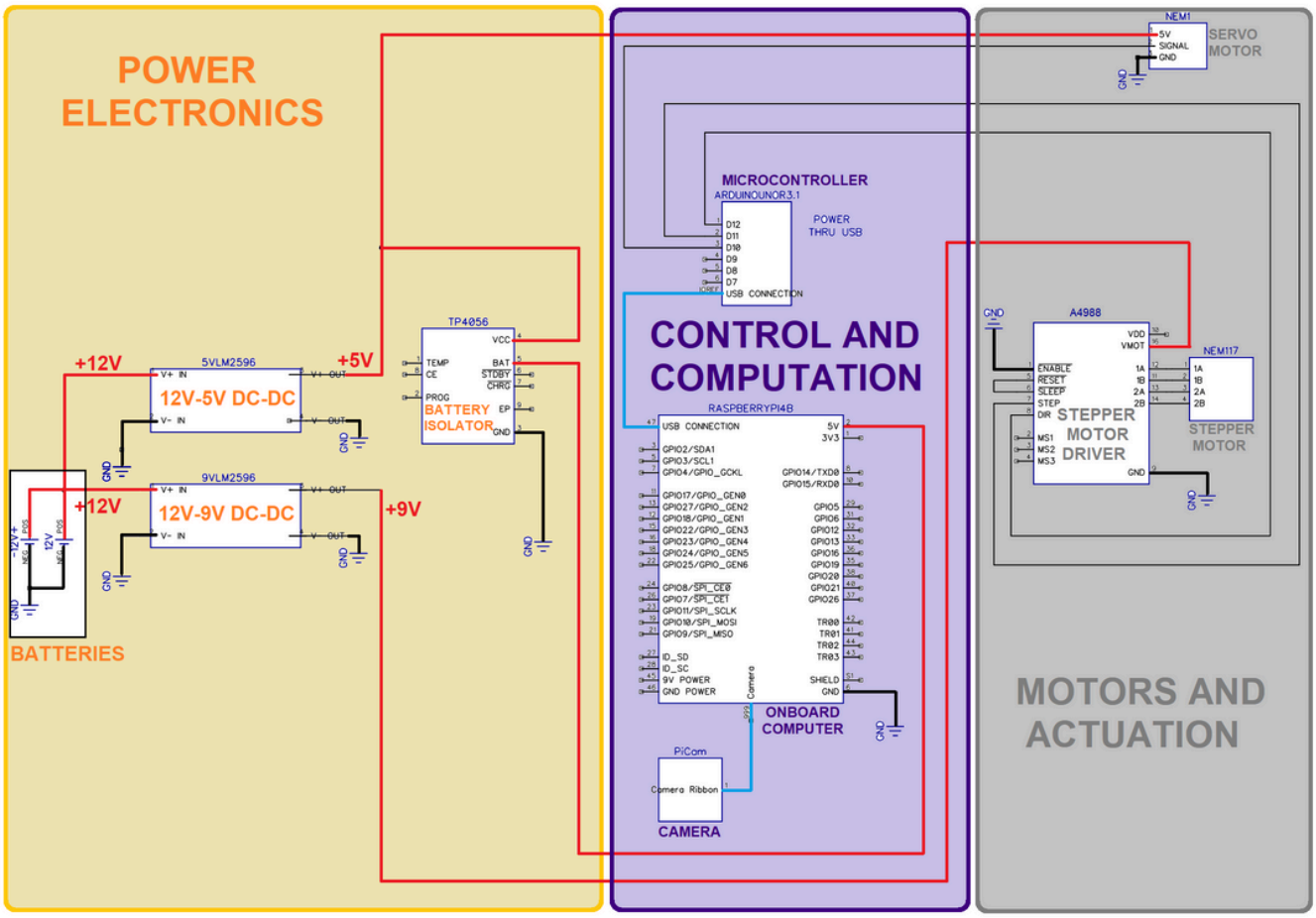
AUTO-AIMING TURRET — STEPPER MOTOR AND GEARBOX

Because I was testing this first without the turret, I needed to ensure that the motor I chose would move the a desired test angle, and I needed to be able to verify the angle it hit. DC motors need a proper PID feedback for this, so to simplify things, I went with a stepper motor, because assuming they don't skip steps, they always move to their desired location without any feedback needed.

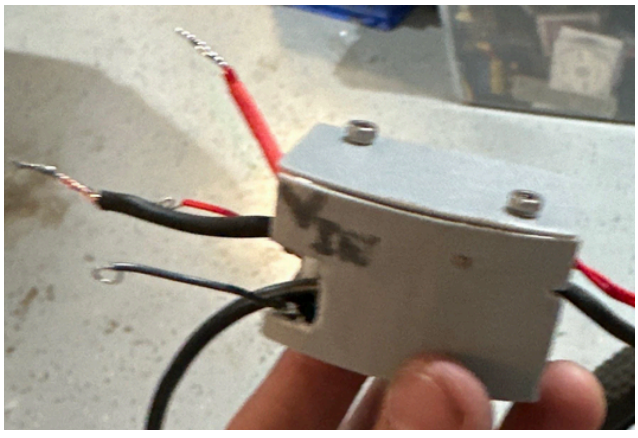
I first tested it by sending it manually typed coordinates through the Arduino serial. Then, I connected the camera and Pi to send coordinates to the Arduino. Before having the system where it would try to center itself with the target, initial testing had it so that the motor would point towards your face as you moved it in front of the camera, and stop there. Its coordinate (this will be elaborated upon later) would not be centered

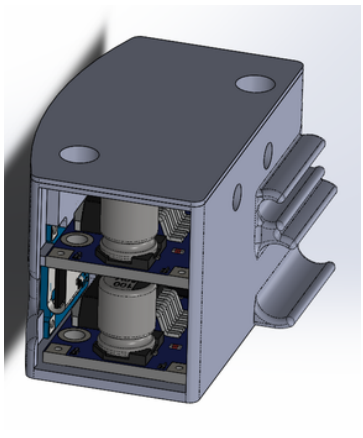
The problems arose when I assembled the motor and turret together. Despite the inherent reduction of the ring base, I found that it was still too difficult for the motor to actually move the base with everything else on top of it weighing it down. So I then decided to gear down the stepper motor with a planetary gearbox. The output shaft (connected to the planet carrier) was centered against both the stepper motor shaft (two blue bearings), and against the ring gear (black bearing), ensuring straight alignment and operation. The gearbox was a 1:4 reduction, meaning that coupled with the ring base's inherent reduction of 1:3, the final motor reduction was 1:12 - the motor would have to spin 12 full revolutions to rotate once around the ring. While this made the turret much slower to move and chase faces, it actually helped, since the pi cam would not have a fast enough framerate, making its visuals blurry when it was panning at high speeds.



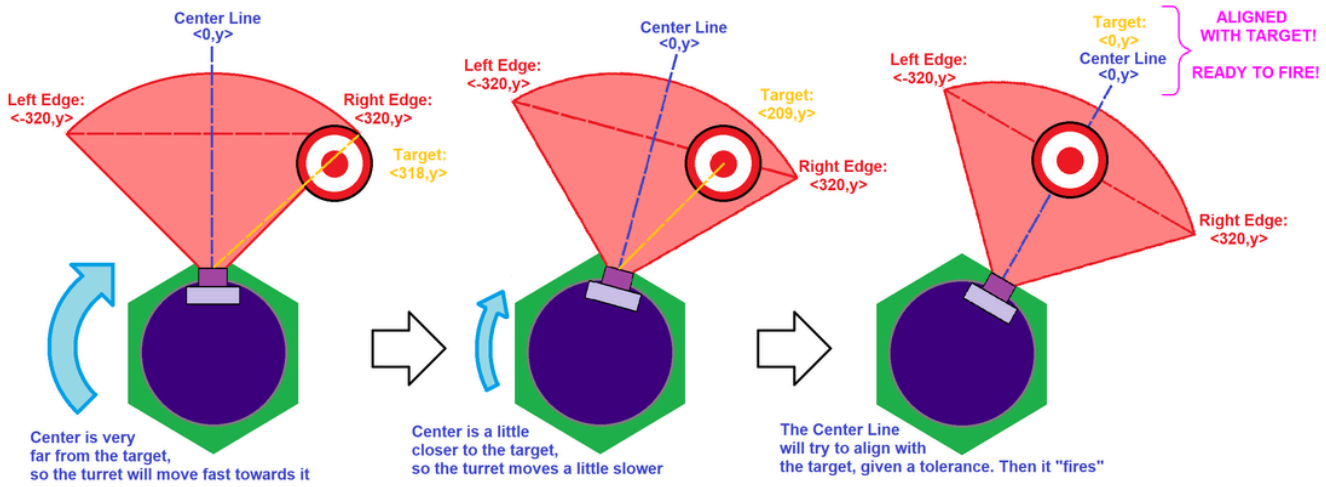


- List of electronics Used:
- Power Electronics
  - LM 2596 DC-DC Stepdown DC-DC Converter (x2)
  - TP4056 Battery Isolator (x1)
  - Slip Ring (x1) - Although the batteries are not on the turret, they only require power and ground wire through the slip ring
  - 12V NiMH 2000mAh Batteries (x2)
- Control and Calculation
  - Arduino Uno R3 (x1)
  - Raspberry Pi 4B,8G (x1)
  - Raspberry Pi HQ Camera (x1)
  - Arducam Lens (x1)
- Motors and Actuation
  - NEMA 17 42Ncm Stepper Motor (x1) - Controlled the horizontal rotation
  - A4988 Stepper Motor Driver (x1)
  - A4988 Shield (x1) - Controlled the vertical rotation
  - SG90 Servo Motor (x1)
- Etc
  - LED (x1) - The LED and buzzer would activate when the turret was properly aligned with a face, simulating “firing”
  - Buzzer (x1)

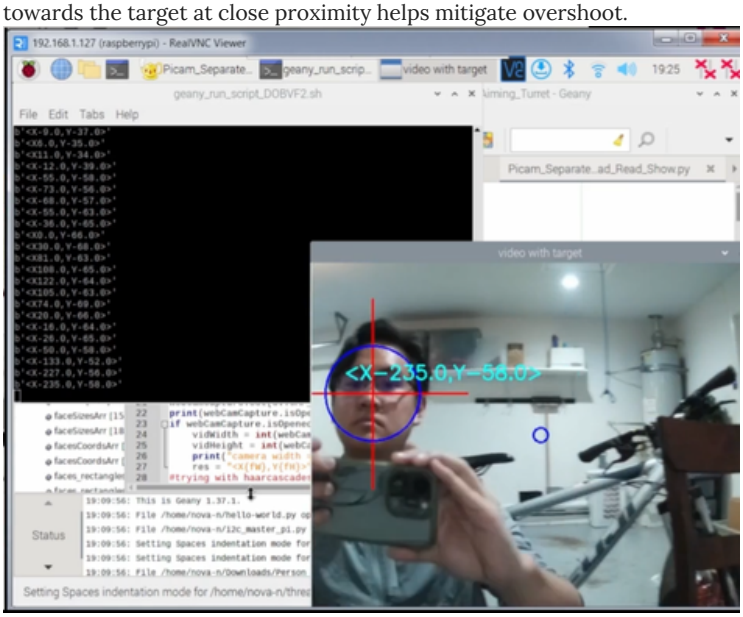




A CAD model and the actual housing for the power electronics

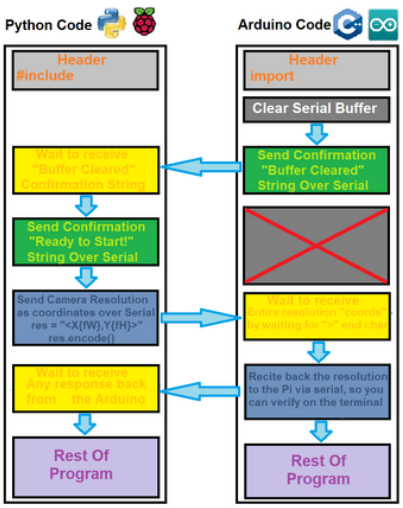


The aiming algorithm is explained in the diagram above. (It is only concerned with the horizontal, aka, the turret base turning, not aiming up, since vertical aim is done by a separate servo, and does not move the camera). As shown, the turret aims with something to proportional control. When a face, or target is within the camera's field of view, the camera will want to align itself with it. This is because the "gun" is fixed to the turret base, as is the camera, meaning if the camera is aligned with the target, then so is the gun, making it ready to fire. The farther the "Center Line" is away from the target, the faster the turret swings towards it. It is beneficial for the turret to be slow when close to the target for two reasons: Firstly, because the Raspberry Pi isn't very powerful, when the turret pans quickly, so does the camera. This causes a blurry image, and it cannot process frames that quickly, leading to a very skippy video that it reads. This means it can skip faces or skip over the target all together. Secondly, having the turret actuate slowly towards the target at close proximity helps mitigate overshoot.



The camera has a resolution of 640x480p. When the camera is on, these pixels can be turned into coordinates (note, depth is not taken into account here, but that will be addressed later). This means that the horizontal is 640 pixels wide, and 480 pixels tall. Setting the center coordinate to <0,0> (the blue circle), that means the left and right edges have coordinates <-320,y> and <320,y> respectively. The Pi then finds faces in each frame of the live video (will be expanded upon later), and translates those into coordinates. These are seen on the black terminal output window, and are constantly sent to the Arduino. As the turret moves closer to the face in question, the face's X coordinate gets closer and closer to the center line, meaning the X coordinate gets closer to <0,y>. When this occurs, and the servo motor points the "gun" (The wooden stick with lights), it "fires".

AUTO-AIMING TURRET — START AND SYNCHRONIZATION



The arduino needs to know when it is being sent coordinates by the raspberry pi, and the raspberry pi needs to know when it can even start sending coordinates over. This is what the synchronization on startup is for.

The Arduino will always start faster than the Pi's Python code, since the Arduino immediately starts once the power is turned on. It needs to wait for the Pi to send over the camera resolution, so that it can calculate how much to move the motor, mapping the coordinate's position on the Pi's screen, to motor movements from the Arduino.

Once it has received the resolution, the Arduino can start receiving coordinates, and will always be waiting for them in the serial buffer.

I used the "Serial Communication Basics" tutorial by Robin2 on the Arduino Forums to learn how to transmit data via the serial bus to and from thar Arduino. The serial buffer has to constantly anticipate for incoming characters, and as soon as one is read, it gets tossed out of the buffer. This is why these characters must be stored.

To know when a message starts, it picks up on the "<" char. This means that any letters and numbers after it can be parsed, and are for sure part of the same coordinate, and not some left over number from the previous sent coordinate.

To know when a message ends, it picks up on the ">" char, meaning that any other numbers or characters that come after it are irrelevant to that coordinate. The string "<X--,Y-->" then gets stored as a variable, and then is further processed.

The ParseInt() function is known as a blocking function, meaning that any code after the function call will not run until the entire string has been parsed. Parsing the string "<X--,Y-->" using the will cause the motor to run jagged, because the code is constantly starting and stopping, because it has to get through several characters before reading the numbers.

```
void recvStartEndMarkers(){
  //static is a variable specific to a function. When it is called repeatedly, it remembers its value between calls,
  //despite being defined initially in the function. This happens until the program ends. The variable isn't reinitialized.
  //this is similar to defining a variable globally so that it doesn't reset every time you call the function
  static bool recvInProgress = false;
  //static byte ndx = 0; //byte is an unsigned number from 0 to 255, so takes one byte of memory
  static int ndx = 0;//C++ doesn't recognize byte, so just using int instead. int is signed, so takes 2 bytes.
  char startMarker = '<';
  char endMarker = '>';
  static char rc;
  //is static bool recvInProgress, since entire coordinate is NOT sent all at once over serial

  while(Serial.available() >0){
    rc = Serial.read(); //picks up one at a time
    if(recvInProgress == true){
      if(rc != endMarker){ //while the chars aren't '>', keep going.
        //Serial.println(rc);
        recievedChars[ndx] = rc; // ndx increments with the loop, and is the char position of the new recieved char
        //Serial.println(recievedChars);
        ndx++;
        if(ndx >= numChars){ //if ndx is position 32 or greater, ndx becomes position 31
          ndx = numChars - 1;
          //How many characters can be received?
          //In the examples I have assumed that you will not need to receive more than 32 bytes. That can easily be altered b
          //Note that the 64 byte size of the Arduino serial input buffer does not limit the number of characters that you ca
        }
      }else{
        //Serial.println(recievedChars);
        recvInProgress = false;
        recievedChars[ndx] = '\0'; //terminates the string, since will increment one more times than the # of chars
        strcpy(tempChars,recievedChars);//copies receivedChars onto tempChars
        ndx = 0;
        coordsReady = true;
      }
    }else if(rc == startMarker){
      recvInProgress = true; //if rc finds '<', then starts the if statement above
    }
  }
  //Serial.println(recievedChars);
  return;
}
```

The coordinates are "Parsed" by using token delimiters. When a coordinate is read through by the above function, what is saved is "X--,Y--". The start and end markers are removed. The only thing left to remove are the commas, and letters. This is what strtok() does. What remains is "X--Y--". The numbers after the letters must now be saved to their respective coordinates.

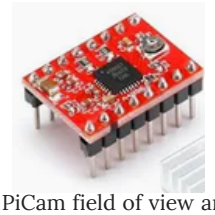
It would be cumbersome to have a variable for each coordinate. X for x, Y for y, Z for z, and so on. What if I had several more coordinates than X and Y? This is why I stored my coordinates in an array. [X,Y]. As seen, the first slot in the array goes the X coordinate. The Y coordinate goes into the second.

```
void coordAssigner(char fromSerial[],float (&coordsGiven)[sizeof(coordAxes)],const char *delim,const char coordAxesGiven[]){
  //don't need to redefine default argument in function definition, just need to do it once
  //need to specify size of coords array, or will get error "reference to array of unkown bound int"
  char *token;
  token = strtok(fromSerial, delim); //breaks out the tokens X100 Y200 and Z400 out
    //but if you check token, it only displays 1 at a time. Must call strtok again, but with NULL to get the next token
  int iteration = 0;
  while(token != NULL){
    if(token[0] == coordAxesGiven[iteration]){ //compares the first char from the token to the char of the coordAxes, so compares char to char
      token ++; //just moves it past the letter X, Y or Z to the number
      coordsGiven[iteration] = atoi(token);
    }
    iteration ++;
    token = strtok(NULL,delimiter); //gets to the next token
  }
  for(int j = 0; j<sizeof(coordAxes);j++){
    //Serial.print(coords[j]);
    //Serial.print(",");
  }
  //Serial.println();
  return;
}
```

```
desiredMoveHoriz = -1.00 * round( coords[0]/cameraResolution[0] * actualSteps);
```

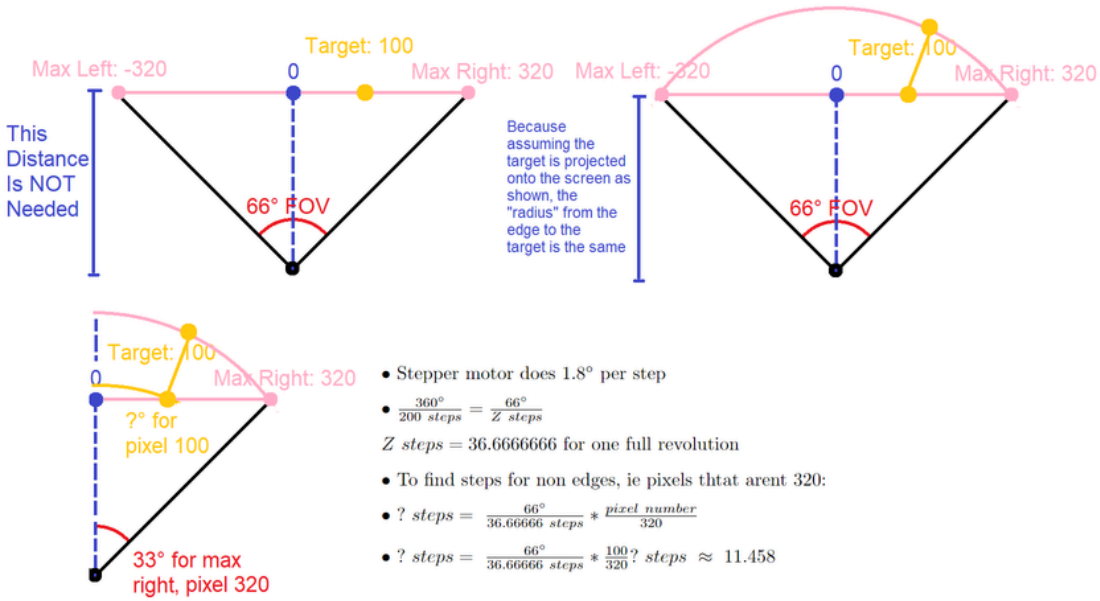
AUTO-AIMING TURRET — ARDUINO MOTOR CONTROL

The stepper motors on the Arduino were controlled by an A4988 stepper motor driver, and the library that came with it.



```
#include <AccelStepper.h>
```

I used a NEMA 17 stepper motor, which did 200 steps to do one complete revolution. The PiCam field of view angle was 66°, and the camera resolution was 640p horizontally. I needed these to solve for how many steps it would need to take to track a face. Drawn below is the mathematical and geometric explanation.



This simple math was easy to test. Before attaching the camera to the turret, I simply attached a piece of tape to the stepper motor shaft. When the Pi would send coordinates, the Arduino would move the motor, and I could see the shaft position easily. The tape would easily point in my direction, without overshoot.

AUTO-AIMING TURRET — FACE TRACKING AND CAMERA CODE

```
cascade_front_faces = cv.CascadeClassifier("/home/nova-n/git/Auto_Aiming_Turret/LBPcascades/lbpcascade_frontalface.xml")
cascade_side_faces = cv.CascadeClassifier("/home/nova-n/git/Auto_Aiming_Turret/LBPcascades/lbpcascade_profileface.xml")
```

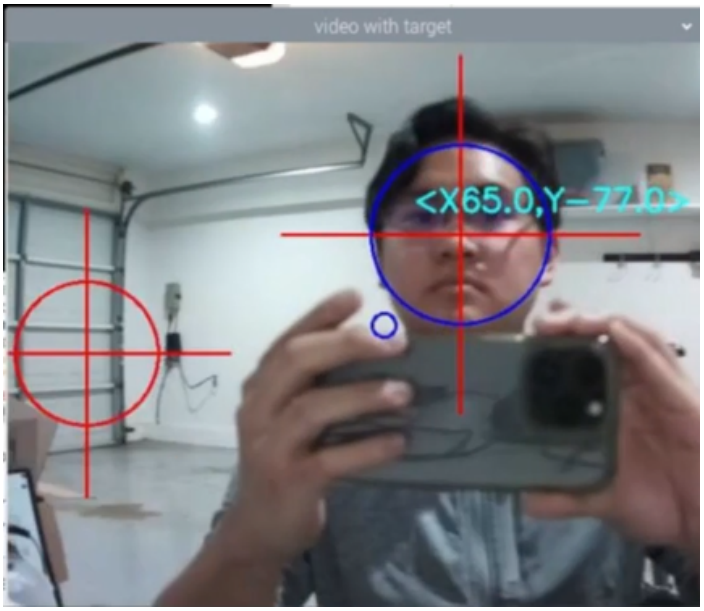
Although I first used Haarcascades to detect faces, I found it was too slow for the responsiveness I needed. This is why I switeched to LBPcascades. Although it was less accurate, the turret would at least not lose the face if it moved too quickly out of view, because there was less lag to process and move with LBP.

Each frame of the camera then went through processing, so that faces could be identified, and marked on screen.

```
faceSizesArr = [] #reset face sizes array
facesCoordsArr = [] #reset the coordinates array
for (x,y,w,h) in faces_rectanglesVid:
    #cv.rectangle(frame,(x,y),(x+w,y+h), (0,0,255),2)
    #cv2.circle(image, center_coordinates, radius, color, thickness)
    cv.circle(frame,(x+w//2,y+h//2),(w+h)//4,(0,0,255),2)
    cv.line(frame,(x+w//2,y-h//2),(x+w//2,y+3*h//2),(0,0,255),2)
    cv.line(frame,(x-w//2,y+h//2),(x+3*w//2,y+h//2),(0,0,255),2)
    faceSizesArr.append((w+h)//4)
    facesCoordsArr.append([x+w//2,y+h//2])

#If no faces detected, will only send <X0.00,Y.00> ONCE before not sending anymore
if len(faceSizesArr) == 0 and wasEmpty == False:
    coords = "<X{fX},Y{fY}>STOP-NOTHING-HERE".format(fX = "NULL", fY = "NULL")
    arduino.write(coords.encode())
    print(coords.encode())
#sending the largest target, aka largest face, aka, closest face to the arduino
if len(faceSizesArr) != 0:
    wasEmpty = False
    largestFaceIndex = faceSizesArr.index( max(faceSizesArr) )
    cv.circle(frame,(facesCoordsArr[largestFaceIndex][0], \
    facesCoordsArr[largestFaceIndex][1]), max(faceSizesArr),(255,0,0),2)
    #Note, I want the center of the screen to be (0,0), but it defines the top left as 0,0, so I must modify the code
    coords = "<X{fX},Y{fY}>".format(fX = facesCoordsArr[largestFaceIndex][0] - centerCoord[0] \
    , fY = facesCoordsArr[largestFaceIndex][1] - centerCoord[1])
    arduino.write(coords.encode())
    print(coords.encode())
    cv.putText(frame , coords,(facesCoordsArr[largestFaceIndex][0]-40,facesCoordsArr[largestFaceIndex][1]-20), \
    cv.FONT_HERSHEY_SIMPLEX,0.9,(255,2550),2)
else:
    wasEmpty = True; #notice executes AFTER sending <X0.00,Y0.00>
#cv.circle(frame,(int(2*centerCoord[0]),int(2*centerCoord[1])),1,(255,0,0),2)
cv.circle(frame,(320,240),10,(255,0,0),2)
```

LBP had more false positives than Haarcascades. But even if the face detection software detecteed multiple faces, the turret would not get confused. The Pi would only send the closest target's coordinates to the Arduino. As a face/target gets closer to the camera, it will appear larger than faces further away. This data is taken into account, via the radius of the circle that is marked on each face.

This was also when I first learned to use threads, multi-processing, and what I/O bound operations were. I made classes, that created objects which had methods to read and show video. The reading and showing are I/O bound operations, so they were run in separate threads. There was no need to lock threads, because the reader just gets from the buffer, and outputs a frame. The shower takes the frame from the reader. It was best to do these in objects, because while functions can return the frames of the webcam, when starting a thread, there is no way to get the output of a function. The object can hold these frames instead, and these will be accessed



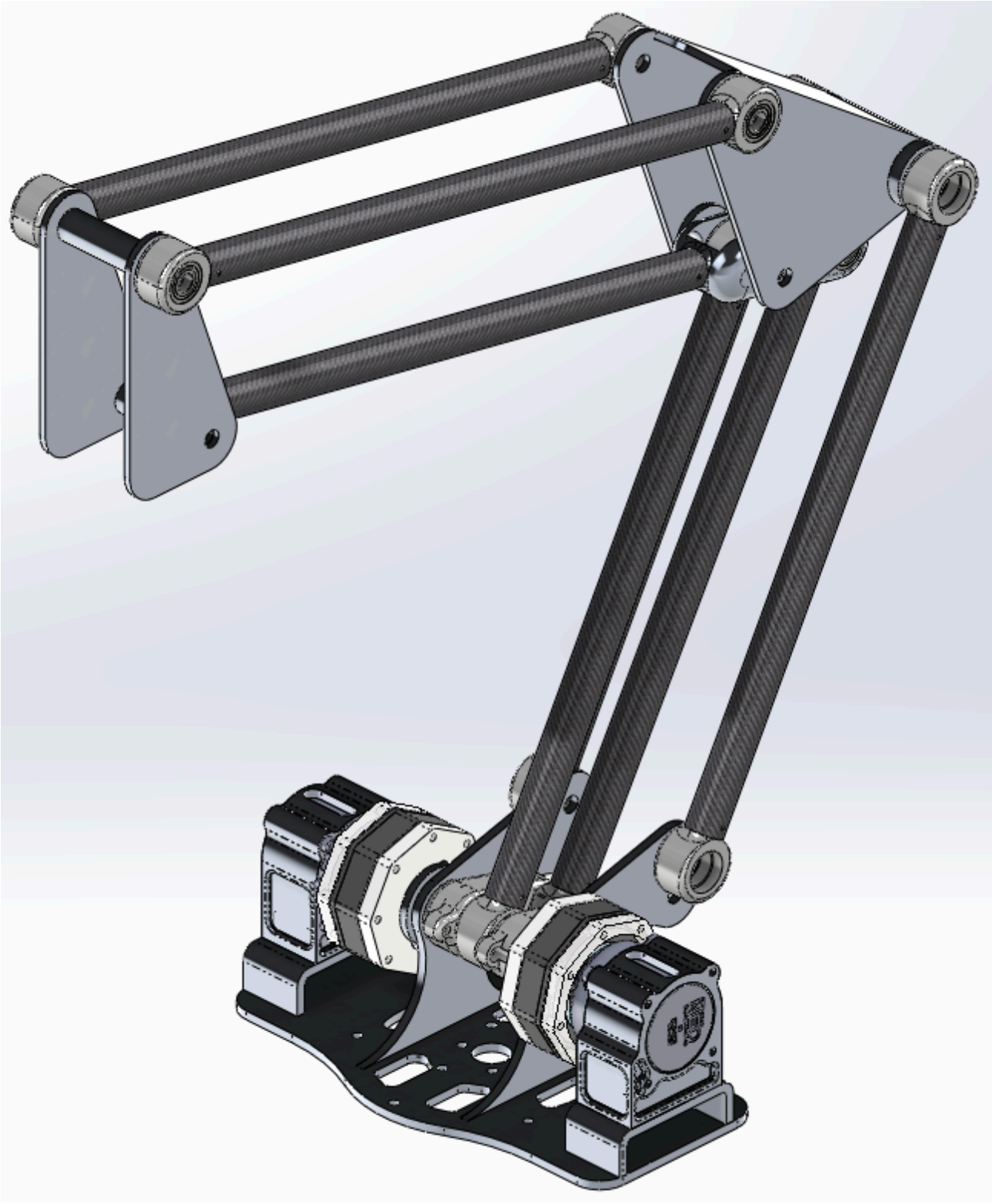

```
frameFinishedReading = threading.Event()
frameQueue = queue.Queue()
gotFrameTrueFalseQueue = queue.Queue()
#communication between threads, and also, a way for the main thread
#(the main program) to know if vidReader is finished reading, and has outputted a frame
class videoReader:
    def __init__(self,cam,lock): #initial state of an object's attributes
        #capture.read() returns two outputs: a boolean that says if a frame is succesfully returned, and each frame
        self.camera = cam
        (self.gotFrame , self.frame) = self.camera.read()
        self.stop = False
        self.timeAtBeginLoop = None
        self.timeAtEndLoop = None
    #can't use self.camera as a default value for the function
    def getAndReadFrame(self,camToRead = None): #the function that needs to be continuously called
        #the self.frame is just the initial value, and __init__ is not a
        print("called getAndReadFrame")
        camToRead = self.camera
        global latencyTime #must declare global in the actual function that accesses it
        while self.stop == False:
            self.timeAtBeginLoop = time.time()
            if self.gotFrame == False or (cv.waitKey(1) & 0xFF == ord("d")):
                self.stop == True
                gotFrameTrueFalseQueue.put(False)
                break
                return
            else:
                #lock.acquire()
                (self.gotFrame , self.frame) = camToRead.read()
                #lock.release()
                #WAITING FOR EVENT CAUSES DELAY AGAIN
                #frameFinishedReading.set() #Event Set
                #print("event set")
                #frameFinishedReading.clear()
            self.timeAtEndLoop = time.time()
            latencyTime = self.timeAtEndLoop - self.timeAtBeginLoop
        return

#Because the video displayer takes form a global queue, its best to leave this as a function, instead of an object
def videoDisplayer(displayerKeepGoing = True):
    while displayerKeepGoing:
        displayerKeepGoing = gotFrameTrueFalseQueue.get()
        if displayerKeepGoing == False:
            break
            return
        elif frameQueue.qsize() > 0:
            cv.imshow("video with target",frameQueue.get())
        else:
            pass
    if displayerKeepGoing == False:
            return
    return
```

ROVER PANTOGRAPH ARM — SEPTEMBER 2023 - MARCH 2024

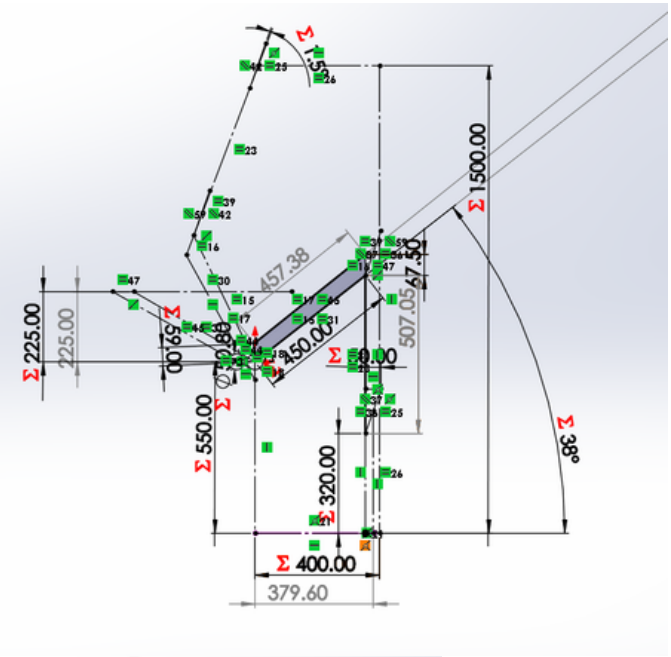
I was involved with Titan Rover for my senior design project, and I was in charge, and worked completely alone on the robotic arm. The design took several iterations, and had strict requirements. Although I didn't have time to program the arm due to being stuck in the machine shop, and dealing with my other academic classes, I still believe I outputted a good product, and the analysis that went into it is invaluable, and applicable to other projects.

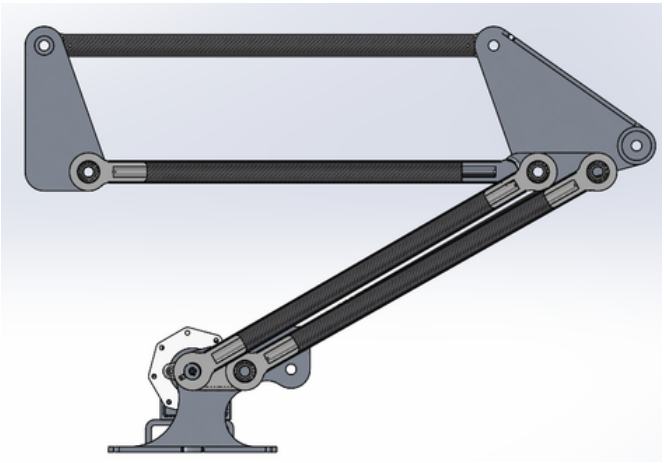
This project primarily focused on 4-bar linkages, and their mechanical advantage. But it also tested my spatial ability, when I tried to balance the arm's range, and robustness, while also trying to get the arm to not run into itself. It also tested my CAD skills, because I ad to design things as if they could be manufactured, and not just 3D printed.

The next few pages will go over in much more detail, all the intricacies of the arm, and the analysis that went into it.

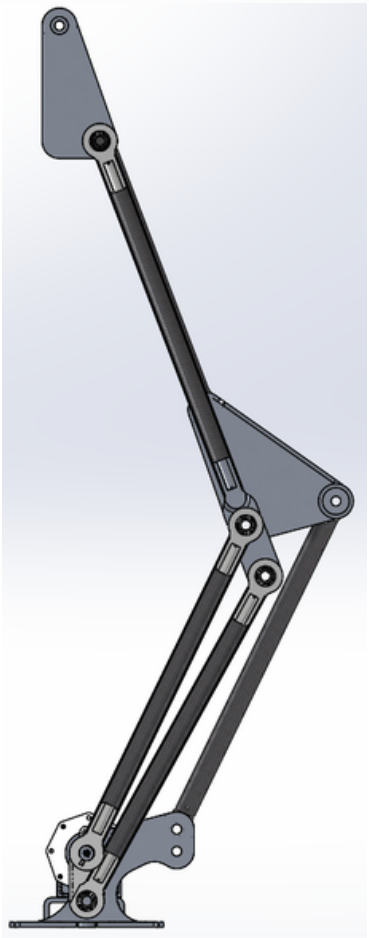


The arm had 3 main requirements: It had to be able to easily reach the ground, It had to easily be able to reach 1.5m up from the ground, and it had to fold up when not in use (especially when the rover itself was moving). Those geometric constraints are shown in the sketch below. All subsequent parts and assemblies were defined around that sketch.

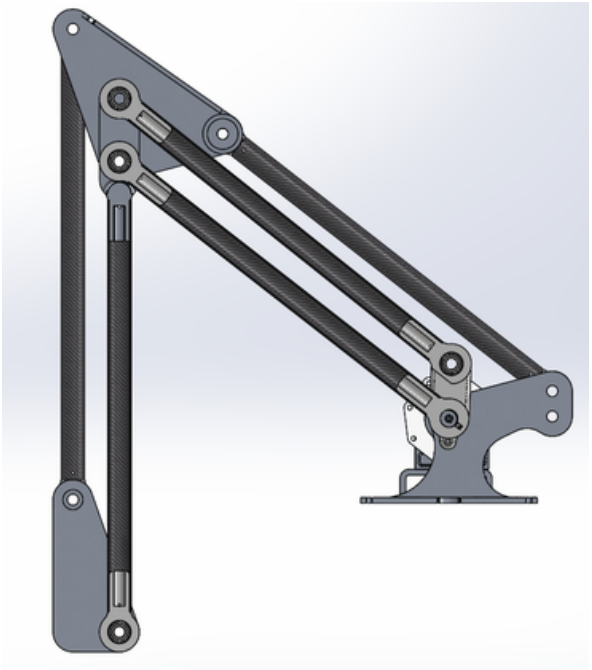




The arm folded up. This moves the center of gravity lower, and closer to the middle of the rover, which would help against tipping over



The arm reaching all the way up. The wrist would have covered the horizontal distance.



The arm reaching down. The wrist would have covered the rest of the distance to the ground. As will be shown later, the wrist's orientation won't be affected by the orientation of the other linkages.

Notice that tall these arms are in a cut plane view. Take a while to appreciate that the linkages do not run into each other when in these positions. They also never run into each other transitioning between positions.

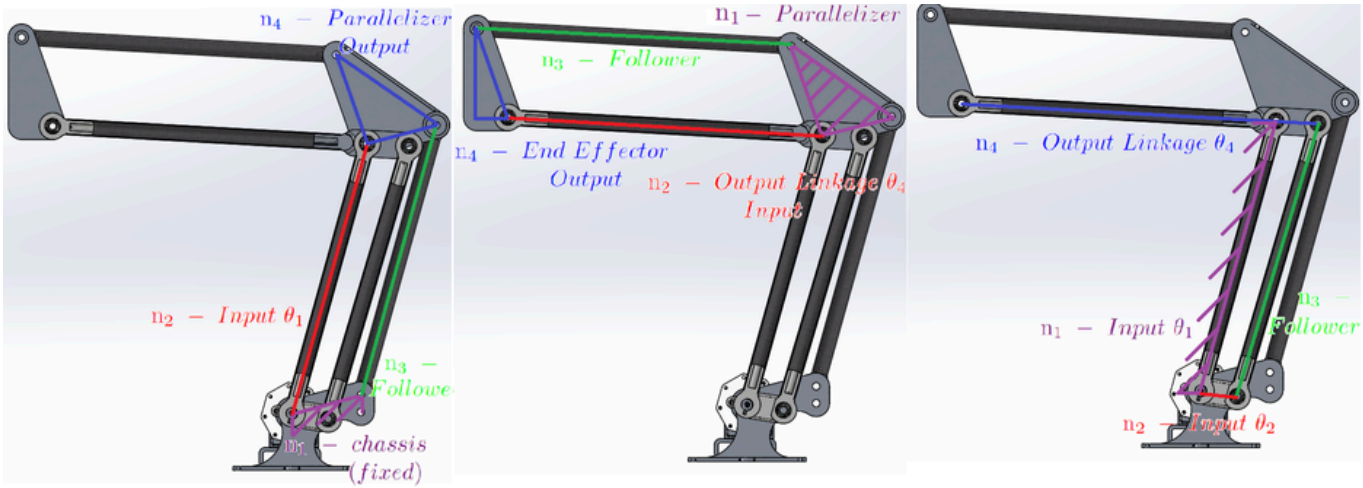
The arm is made of three different 4 bar mechanisms. Shown in the first and second pictures are the parallel arm mechanisms. As shown in the previous page, no matter how the linkages are moved or oriented, the output triangle (at the very tip) always stays parallel to the ground. This is useful, because the wrist was supposed to be mounted onto it. This meant that the wrist's orientation would not be influenced in any way by the position and orientation of the other linkages.

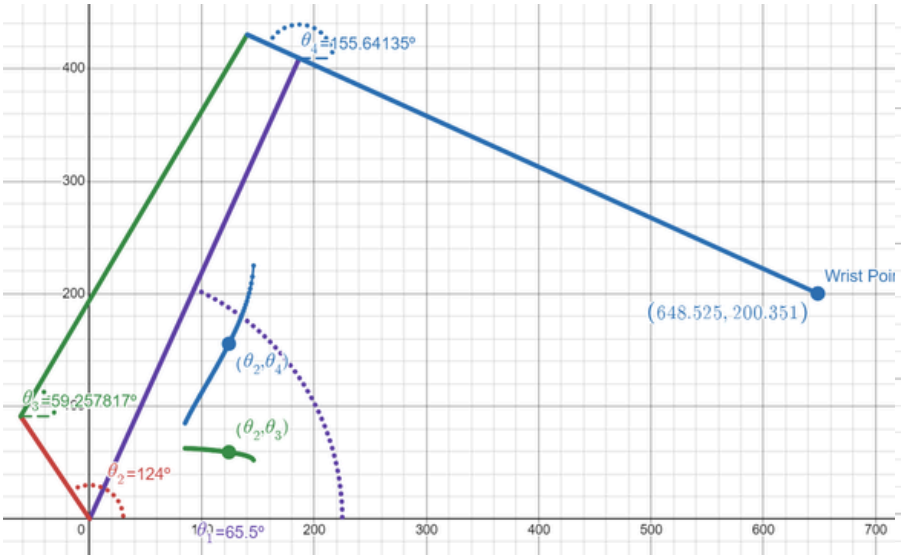
The first picture has a motor that powers theta1. This motor controls the "shoulder" of the arm

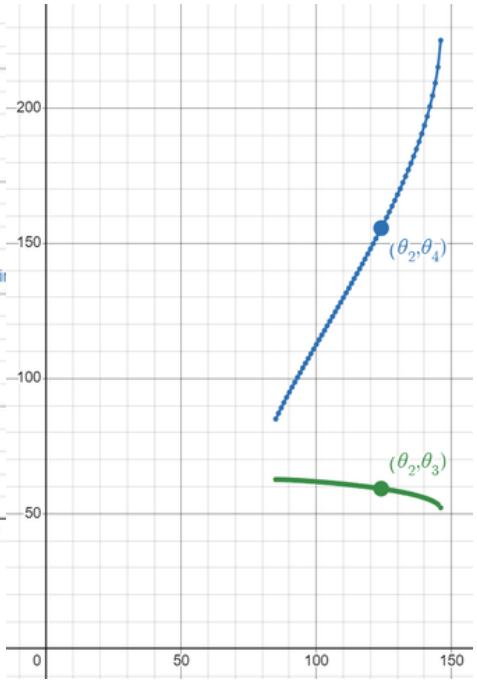
The third picture shows how the "elbow" of the arm is actuated. Instead of having a motor mounted onto the elbow itself, which would cause a heavy mass to hang and swing at a long distance, the vibrations of the motor would have severely amplified the moment put on the arm. This is why the elbow's motor is also placed at the shoulder, at the exact level of the first motor. The elbow motor drives theta2, and through a 4 bar linkage, actuates and changes the angle of theta4.

In addition to the range, compactness, and other geometric requirements, I had to consider the mechanical advantage that my 4-bar linkage would produce. While I could theoretically reach anywhere with the longest possible arms, I would also be sacrificing a lot of mechanical advantage, possibly negating any of the precious torque the motors powering it could provide. The nature of it being an arm meant that I was already carrying heavy loads out at a long distance from a pivot, already heavily fighting the torque of the motor.

This is why I created a graphing calculator to analyze the 4 bar kinematics (of the third picture) https://www.desmos.com/calculator/s3wx35sbtg

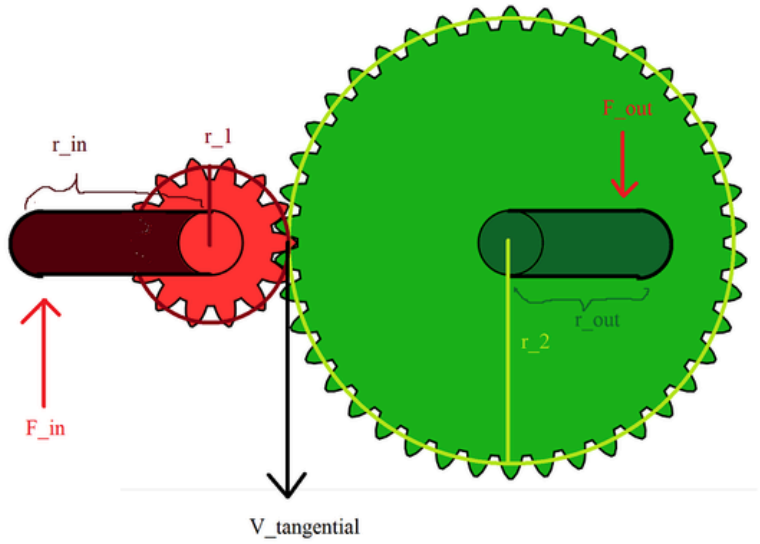






The arm had inputs of torque, but the load it dealt with was a force. Although mechanical advantage is typically used to describe gears and 4-bar mechanisms, they are usually ratios of input force to output force, or input torque to output torque. There is not measurement or term for input torque to output force.

This is why I created my own Mechanical Advantage_forceToTorque number. But to understand it, I want to show how it is derived by first showing how mechanical advantage is derived, using the example of a simple gear pair.



Starting off with the definition of Mechanical Advantage (MA):

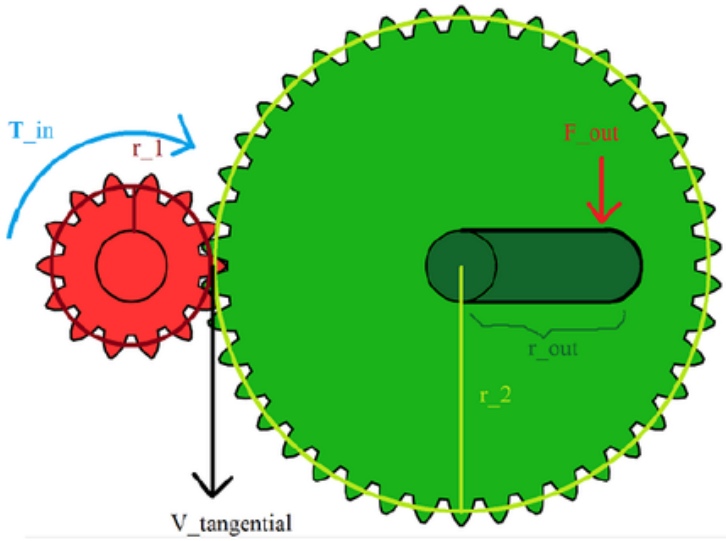
- $MA = \frac{F_{out}}{F_{in}}$
- $MA = \frac{F_{out} \times r_{out}}{F_{in} \times r_{in}} \left(\frac{r_{in}}{r_{out}}\right) = \frac{T_{out}}{T_{in}} \left(\frac{r_{in}}{r_{out}}\right)$

Assuming no losses of power, the power in should equal the power out. Then substitute into MA

- $P_{in} = P_{out} \Rightarrow T_{in}\omega_{in} = -T_{out}\omega_{out} \Rightarrow \frac{T_{out}}{T_{in}} = -\frac{\omega_{in}}{\omega_{out}}$
- $MA = -\frac{\omega_{in}}{\omega_{out}} \left(\frac{r_{in}}{r_{out}}\right)$

Knowing that tangential velocity is equal to angular velocity cross radius (assuming a constant radius, which spur gears don't change radius), and knowing that tangential velocity at the pitch radii of both gears must be the same...

- $V_{tangential} = r_1\omega_{in}, V_{tangential} = -r_2\omega_{out} \Rightarrow -\frac{\omega_{in}}{\omega_{out}} = \frac{r_2}{r_1}$
- $MA = \left(\frac{r_2}{r_1}\right)\left(\frac{r_{in}}{r_{out}}\right)$



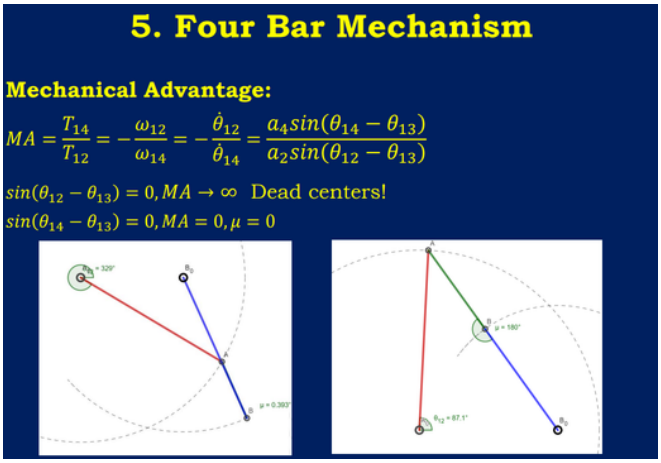
Now, instead of having an input force, we can describe T_in as F_in x r_in, and have an input torque. In order to keep the mathematical ratio of F_out to T_in, if we turn F_out into T_out by multiplying it by r_out, we must also divide by r_out.

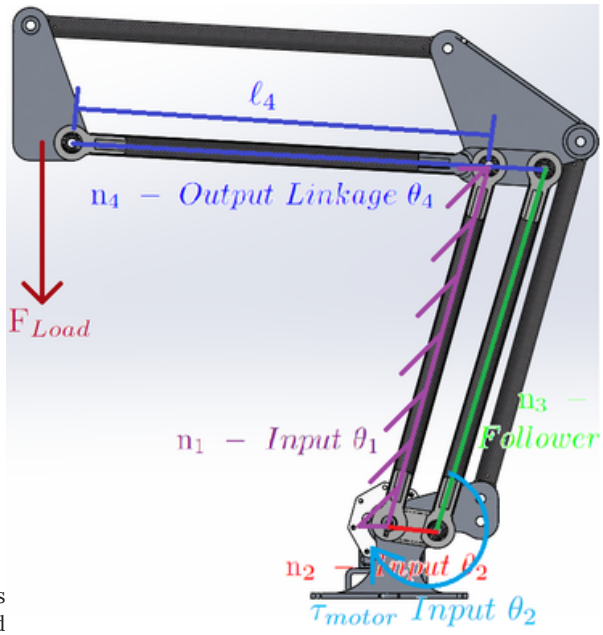
Eventually, the ratio of input torque to output force can be described purely with the geometry of the system: the radii of the gears, compared with the radius of the output lever.

$$MA_{force-to-torque} = \frac{F_{out}}{T_{in}} = \frac{F_{out} \times r_{out}}{T_{in}} \left(\frac{1}{r_{out}}\right) = \frac{T_{out}}{T_{in}} \left(\frac{1}{r_{out}}\right) = -\frac{\omega_{in}}{\omega_{out}} \left(\frac{1}{r_{out}}\right) = \left(\frac{r_2}{r_1}\right)\left(\frac{1}{r_{out}}\right)$$

The concept can now be expanded to the more complex 4-bar mechanism. The input torque, T_motor, which controls theta_2 comes from the elbow motor. The output force is F_Load, which will come from whatever the arm is carrying. The MA_torqueToForce will be the ratio of T_motor to F_Load.

Also shown below is a velocity analysis of any 4-bar linkage. Again, it is defined by the system's geometry. This time however, it is also defined by the system's position itself.





$$-\frac{\omega_{1,2}}{\omega_{1,4}} = \frac{r_4 sin(\theta_{1,4} - \theta_{1,3})}{r_2 sin(\theta_{1,2} - \theta_{1,3})}$$

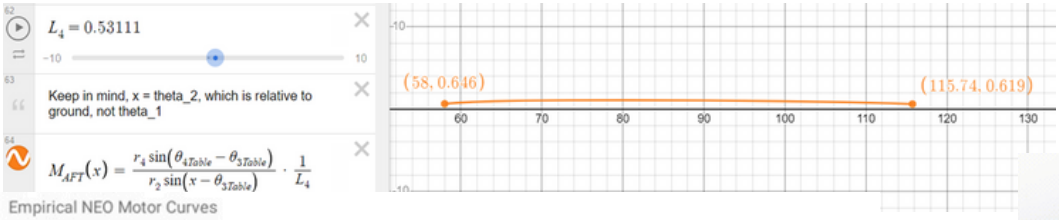
Shown below is the calculation for torque to force. The same logic applies as last time. In the end, the mechanical advantage depends on the actual speed of the linkages themselves. Plugging in the velocity analysis from above, MA is clearly and easily defined.

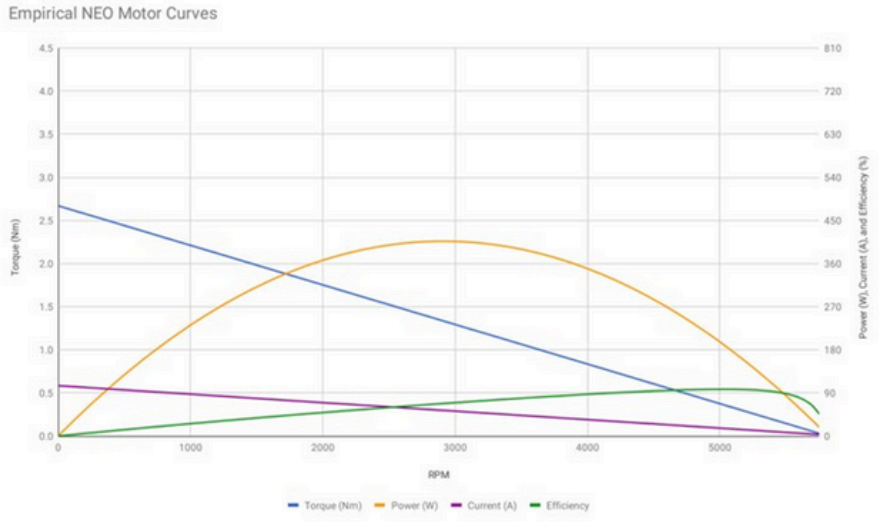
It can be seen that the MA changes as the position of the arm changes. The graphing calculator also keeps track of that,

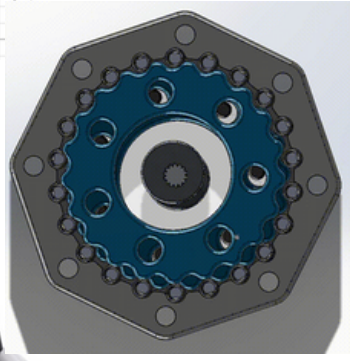
$$MA_{force-to-torque} = \frac{F_{load}}{T_{in}} = \frac{F_{load} \times \ell_4}{T_{in}}\left(\frac{1}{\ell_4}\right) = \frac{T_{out}}{T_{in}}\left(\frac{1}{\ell_4}\right) = -\frac{\omega_{in}}{\omega_{out}}\left(\frac{1}{\ell_4}\right) = -\frac{\omega_{1,2}}{\omega_{1,4}}\left(\frac{1}{\ell_4}\right)$$

$$MA_{force-to-torque} = \frac{r_4 sin(\theta_{1,4} - \theta_{1,3})}{r_2 sin(\theta_{1,2} - \theta_{1,3})}\left(\frac{1}{\ell_4}\right)$$

This became useful when sizing the gearbox around the motors.

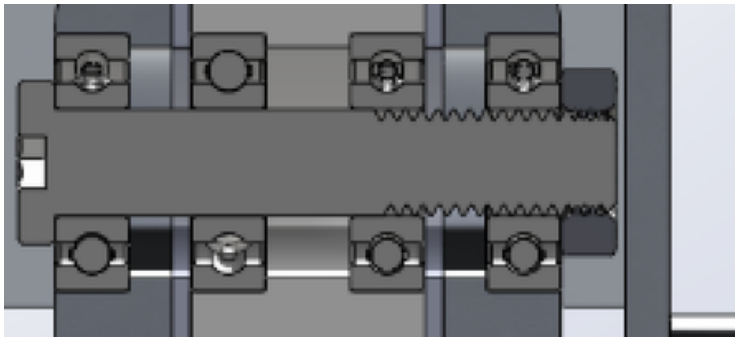






ROVER PANTOGRAPH ARM — SCREWS AS AXLES AND BEARING CREATIVITY

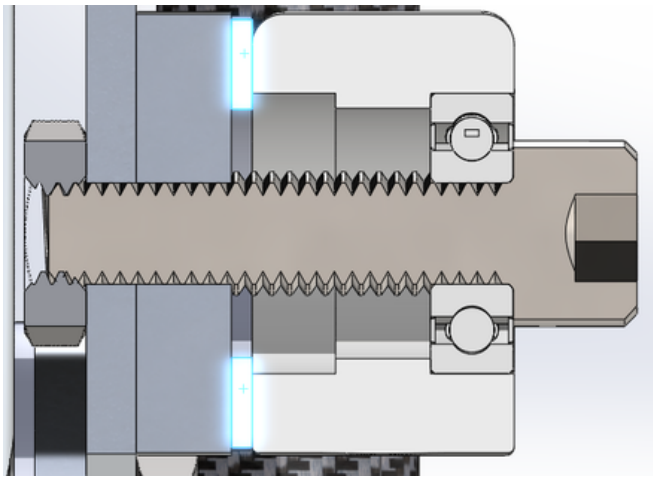
I thought it would be efficient to use heavy M12 screws as both a way to secure parts together, as well as an axle. Normally, tightening something would make it resistant to movement, and would add heavy friction. However, I did some creative things to get around this.

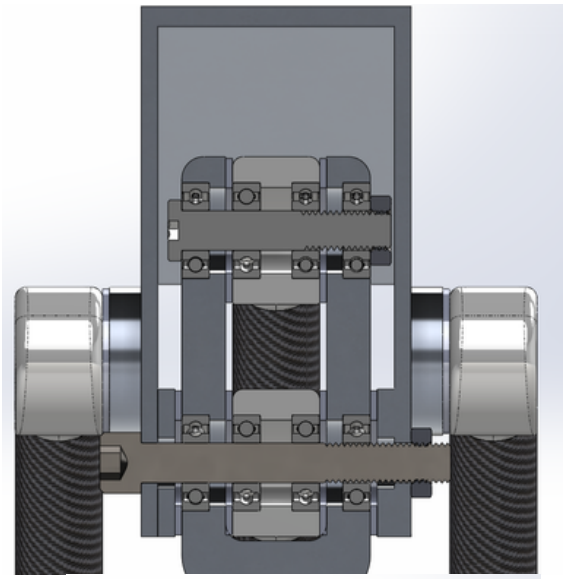


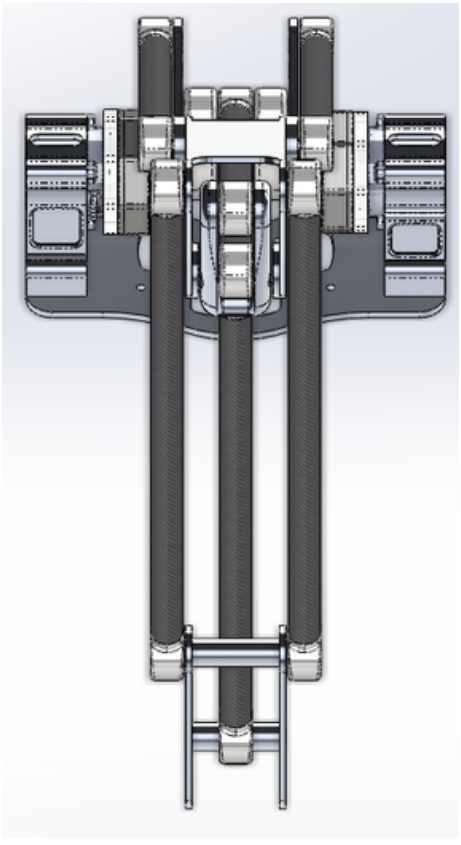
As can be seen to the left, the head of the screw, and the nut only touch the inner race of the bearing, while the outer race of the bearing is firmly embedded into the linkage. The bearings are prevented from sliding inwardly die to the "shelves", and are kept in place from sliding outwardly via the screw and nut. This therefore also keeps the linkages in place.

Think of it this way: The screw and nut are so tight, that they prevent the inner race from moving. However, the outer race is still free to move. And because it is embedded into the piece, the piece is also able to freely move around the bearing.

In cases where linkages and joints must "rub" on each other, I was still able to tighten them together axially, while allowing them to rotate freely and easily. Highlighted in blue are thin thrust needle bearings. Even if the two faces that rub on the bearing move at different speeds, there will still be smooth motion, as the needles will roll in between them with (negligible) slipping. (Negligible because they are cylindrical bearings, instead of cone shaped, meaning the radius of the rolling circle doesn't match the radius it is rolling around the screw).



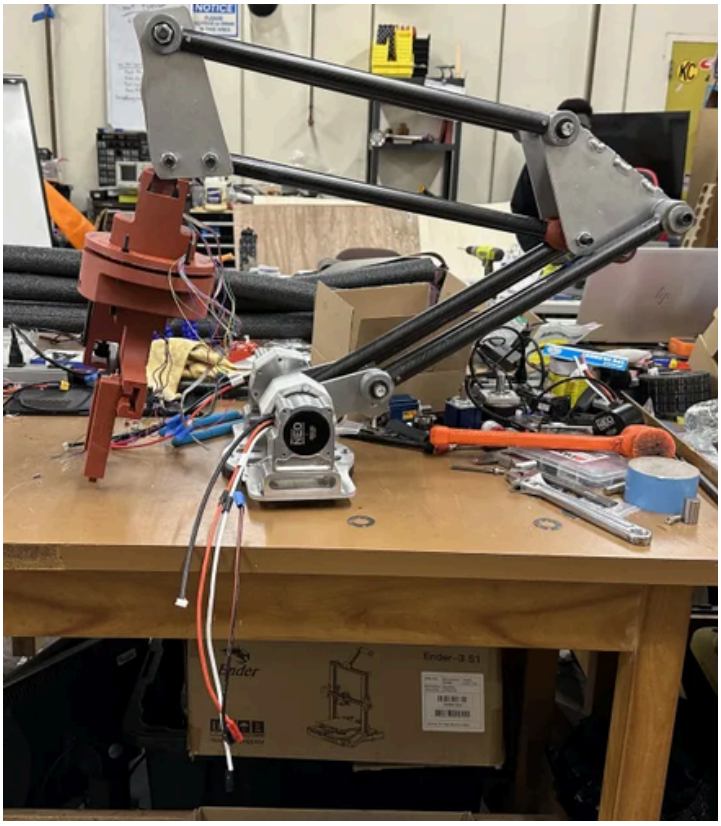




Shown to the left are screenshots of the arm's linkages not running into each other, and having ample freedom and space to move, while still being compact and organized.

ROVER PANTOGRAPH ARM — IN REAL LIFE

Below are pictures of the actual made product, as well as its individual components.

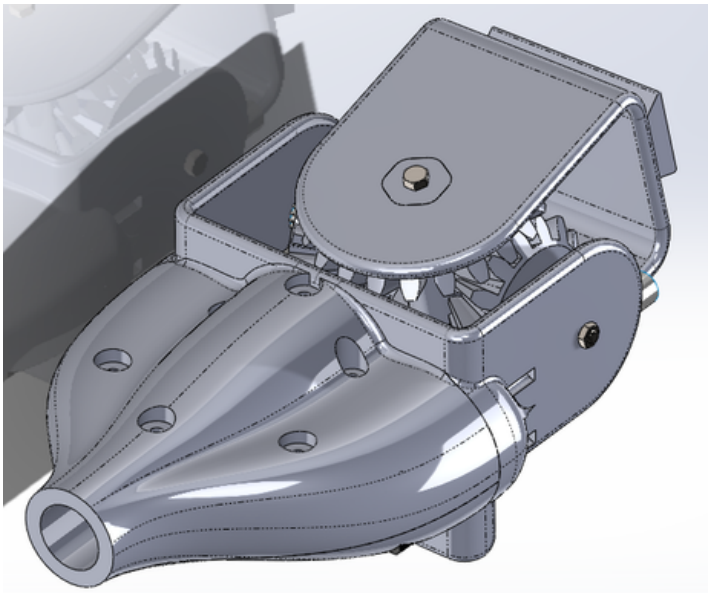


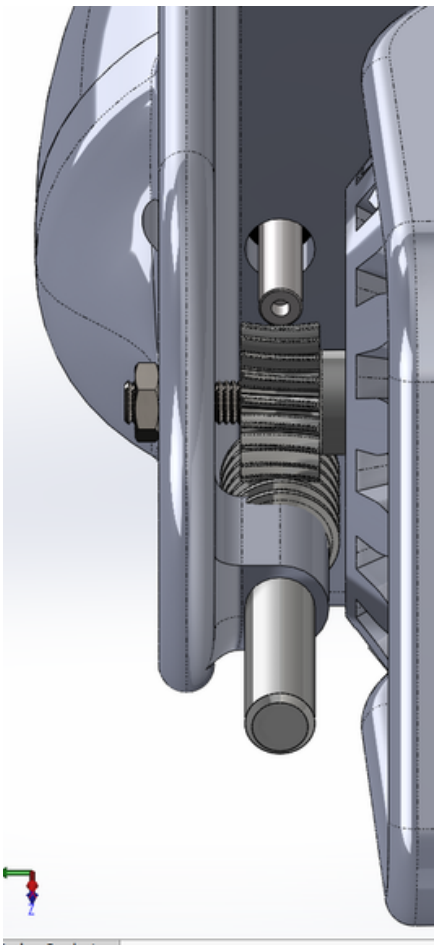
ROVER PANTOGRAPH ARM — DIFFERENTIAL BEVEL JOINT WRIST

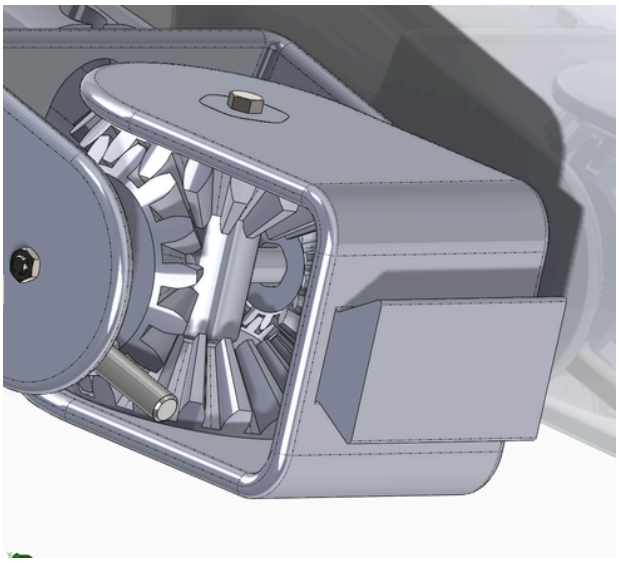
Shown below was an idea for the wrist. This is known as a differential bevel joint. It works like a car differential, but in reverse. Two side bevel gears are spun, and they output the speed and orientation of the dovetailed prong. If they move in opposite directions at equal magnitude speeds, the output (the dovetailed piece) will yaw. If they move both in the same direction at the same speed, the output will pitch. This is a nice compact way to get two axes of rotation centered at the same point. Two motors are still used, but this configuration is much more compact.

They were supposed to be driven by worm screws, to prevent back driving (so the load wouldn't fall), and a high reduction (so the wrist could lift load).

Unfortunately, there just wasn't enough time to implement this wrist.







Shown below are screenshots of me learning to use fusion 360 to mill solid stock of aluminum in HAAS CNC machines. I learned about speeds and feeds, and creative ways to set up workpieces.