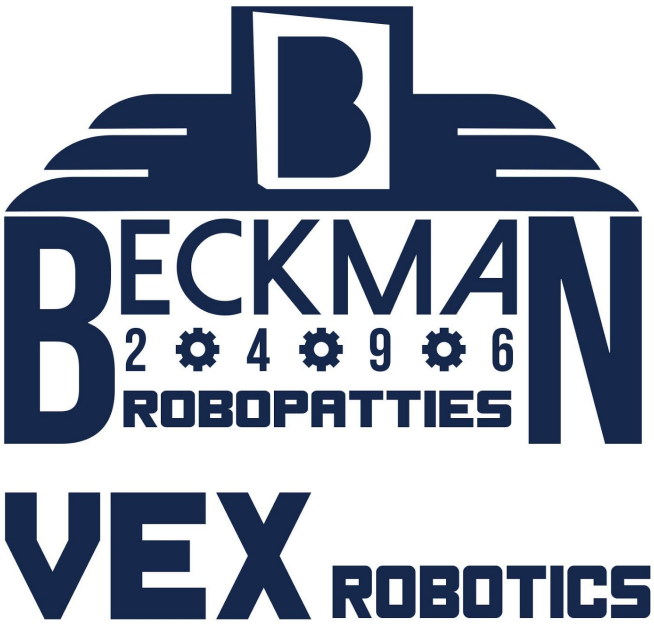


2496W Documentation

Arnold O. Beckman High School

Authors: Aayush S., Andres G., Archis S., Ashwin D., Conner W., Tyler O., Saachi G.

Table of Contents

Programming Notebook Included at End

Table of Contents

Programming Notebook Included at End

Table of Contents

Programming Notebook Included at End

Defining the Problem (Team, 6-10)

Safety (<S#>) and General (<G#>) rules

These are some of the rules that we found crucial to understand as a prerequisite for understanding how the VEX Robotics Competition and how this game in general works. The following rules are taken from the official game manual, the version being the one that was available during the specified date. Here are some of the rules that we believe to be relevant in defining the constraints for this game.

<G4> Robots begin the Match in the starting volume. At the beginning of a Match, each Robot must be smaller than a volume of 18″ (457.2 mm) long by 18″ (457.2 mm) wide by 18″ (457.2 mm) tall. Using Field Elements, such as the field perimeter wall, to maintain starting size is only acceptable if the Robot would still satisfy the constraints of <R5> and pass inspection without the Field Element. Robots in violation of this limit will be removed from the field prior to the start of the Match, at the Head Referee's discretion.

<SG1> Starting a Match. Prior to the start of each Match, the Robot must be placed such that it is:

- **a. Contacting its Home Zone.**
- **b. Not contacting the gray foam field tiles outside of the Alliance's Home Zone.**
- **c. Not contacting any Balls other than the Preload.**
- **d. Not contacting another Robot.**
- **e. Contacting exactly one (1) Preload.**
- **i. The Preload must be contacting exactly one (1) Robot.**
- **ii. The Preload must be fully within the field perimeter.**
- **iii. The Preload must not be breaking the vertical projection of the Goal, i.e. the Preload must not be inside or above the Goal.**

<SG5> Balls may not be de-scored from the top of Goals. Balls that are Scored may not be lifted by any means such that the Ball goes above the top edge of the Goal.
It is expected that while removing Balls from the bottom of the Goal, this may cause the top Ball to momentarily go above the top edge of the Goal. This would not be a violation of this rule and is considered to be normal game play. If the Match ends while a Robot is removing a Ball from the bottom of the Goal that contains three (3) Balls and the top Ball remains partially above the top edge of the Goal, that Ball will be considered Scored and no penalty to the Team will be assessed. Minor violations of this rule that do not affect the Match will result in a warning. Match Affecting offenses will result in a Disqualification. Teams that receive multiple warnings may also receive a Disqualification at the Head Referee's discretion.

<SG8> Possession is limited. Robots may not have greater-than-momentary Possession of more than three (3) Balls of its opposing Alliance's color at once. When two Robots from the same Alliance are working in tandem and blocking Balls, those Robots may not possess a total of more than six (6) Balls of its opposing Alliance's color at once. Robots that violate this rule must stop all Robot actions except for those actions that are attempting to remove the excess Ball.

<R10> A limited amount of custom plastic is allowed. Robots may use non-shattering plastic from the following list; polycarbonate (Lexan), acetal monopolymer (Delrin), acetal copolymer (Acetron GP), POM (acetal), ABS, PEEK, PET, HDPE, LDPE, Nylon (all grades), Polypropylene, FEP; as cut from a single 12" x 24" sheet up to 0.070" thick.
a. Shattering plastic, such as PMMA (also called Plexiglass, Acrylic, or Perspex), is prohibited.
b. Plastic may be mechanically altered by cutting, drilling, bending etc. It cannot be chemically treated,
melted, or cast. Heating polycarbonate to aid in bending is acceptable.

Relevant Definitions for Understanding the Game

Alliance Home Row – The three (3) Goals in each Alliance's Home Zone.

Autonomous Bonus - A point bonus of six (6) points awarded to the Alliance that has earned the most points at the end of the Autonomous Period.

Autonomous Line – The pair of white tape lines that run across the center of the field. Per <SG2>, Robots may not contact the foam field tiles on the opposite Alliance's side of the Autonomous Line during the Autonomous Period.

Ball – A hollow plastic spherical-shaped, dimpled object, with a diameter of 6.3" (160mm), that can be Scored in Goals.

Connected Row - A row where all three goals are owned by the same alliance.

Goal - One of nine (9) cylinders in which Robots can Score and remove Scored Balls. The Goals are all 18.41" (467.6mm) tall and have an inside diameter of 7.02" (178.3mm). The Goal consists of four (4) retaining rings and four (4) PVC pipes. The outer edge of the ring is considered to be

the outer edge of the Goal. The upper edge of the top ring is considered to be the upper edge of the Goal.

Home Zone – One of two (2) areas, one (1) for each Alliance, where Robots start the match and define the location of the Alliance Home Row. The Home Zones are defined by the inner edges of the field perimeter and the outer edge of the tape line that runs across the field adjacent to the Alliance Station, i.e. the tape line is part of the Home Zone. The Alliance Home Zone is closest

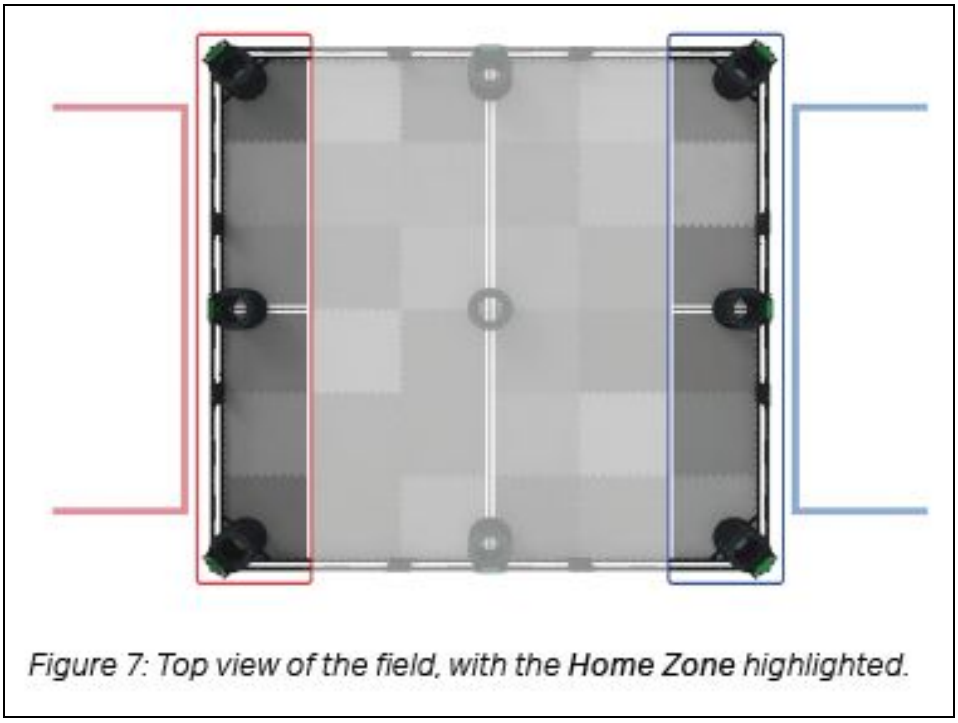


*Figure 7: Top view of the field, with the **Home Zone** highlighted.*

to their Alliance Station The Home Zone refers to the foam field tiles; it is not a 3-dimensional volume.

Owned - A Goal status. A Goal is considered Owned by an Alliance if its colored Ball is the vertically highest Scored Ball in that Goal.

Figure 8: This Goal is Owned by the blue Alliance, as the top-most Ball is completely within the upper edge of the Goal.

Figure 9: This Goal is owned by the Red Alliance, as the top-most Ball is not completely within the upper edge of the Goal. The top red Ball is Scored because it is below the upper edge of the Goal.

Figure 10: This Goal is owned by the Blue Alliance, as the top-most Ball is not completely within the upper edge of the Goal. The two bottom-most Balls would both be considered Scored, as they are partially within the outer edge of the Goal

Possession – A Robot is considered to be Possession a Ball if a Ball is in an unscored position and either of the following criteria is met:

• The Robot is carrying, holding, or controlling the movement of a Ball such that if the Robot changes direction, the Ball will move with the Robot. Pushing/plowing Balls is not considered Possession, however, using concave portions of your Robot to control the movement of Balls is considered Possession.

• The Robot is blocking the opposing Robot's access to Balls that are located between Goals along the field perimeter. Blocking access to Balls is considered Possessing those Balls only if the

opposing Robot is attempting to make contact with those Balls from close range and those Balls are at least partially within the width of the Goals between the Goals. Robots on the same Alliance working in tandem to block access to Balls would share the Possession of the Balls. See <SG8> for Possession

Row - Three (3) Goals that make up a straight line. There are a total of eight (8) Rows including two (2) Alliance Home Row.

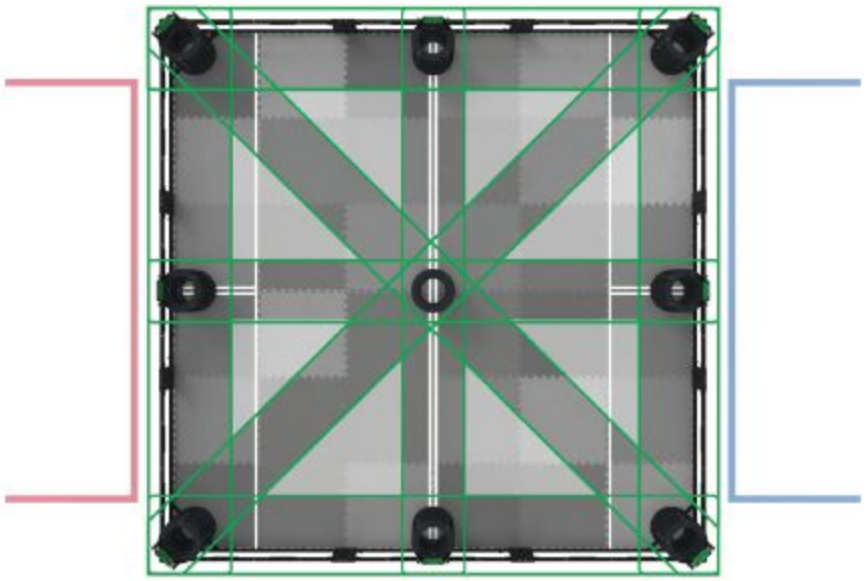


*Figure 11: Top view of the field, with the 8 possible Connected **Row** options highlighted*

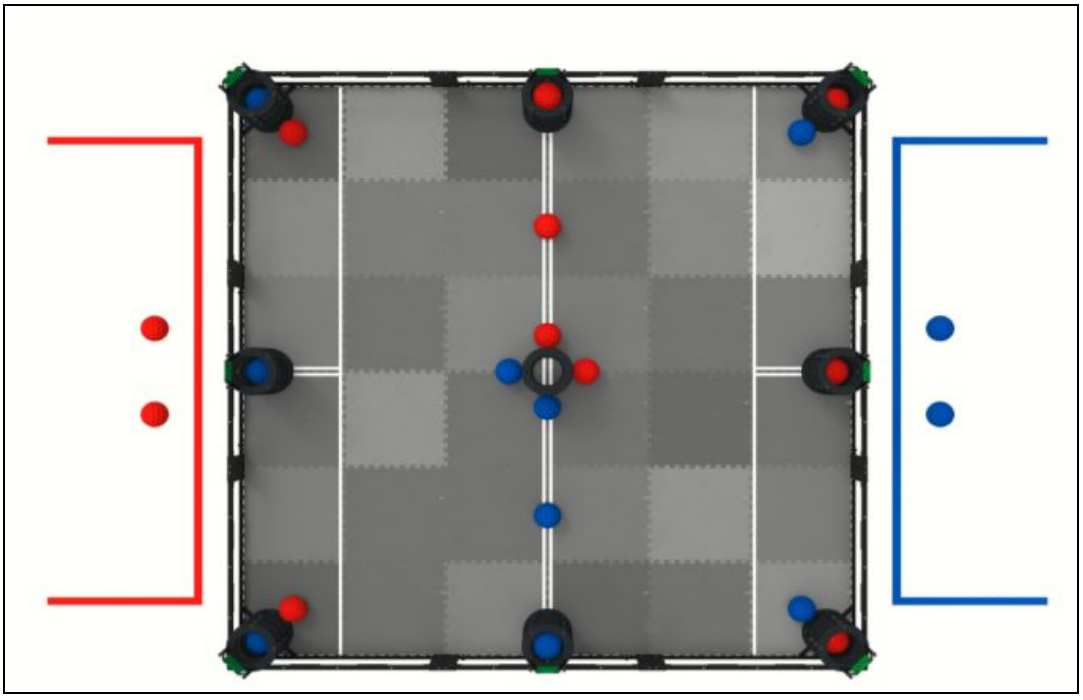
Scored - A Ball status. A Ball is considered Scored in a Goal if it is not touching a Robot of the same color as the Ball and meets all of the following criteria.

- The Ball is fully or partially within the outer edge of the Goal.

- The Ball is fully below the upper edge of the Goal.
- The Ball is not contacting the foam tiles outside of the Goal.

Overview of This Season's Challenge: Change Up

The VEX Game for the 2020-2021 is called "Change Up" in which the general object of the game is to engineer a robot that is able to compete in such a manner that it scores more than the opposing alliance. One addition for this game that was different from this game is that for a "home row" that an alliance is able to win during the autonomous period, one win point is awarded to that alliance. In other words, there is a direct contribution of the autonomous period to a robot's standing in the overall tournament, and this is different from previous years.



Field Description

- The field is 12 feet x 12 feet
- Balls
    - 16 red and 16 blue
    - 2 red balls as the red alliance preload
    - 2 blue balls as the blue alliance preload
    - Balls have a diameter of 6.3 inches (160 mm)
- Goals
    - 9 goals where either alliance can score
        - 18.41 inches (467.6mm) tall
        - 7.02 inches (178.3 mm) inside diameter
    - 4 goals start with red balls at the bottom
    - 4 goals start with blue balls at the bottom
    - 1 goal (center goal) starts with no balls in it
- Alliance Specific Zones
    - Home Zone
        - The areas inside the inner edges of the field and the outer edge of the tape line
        - One for the red alliance and one for the blue alliance
    - Autonomous Line
        - The pair of white tape lines that run across the center of the field
        - During autonomous, you are not allowed to cross the pair of white lines

The way that the robotics competition is structured is that there is a 15 seconds autonomous period in which the robots are able to score in their area of autonomous, which is from their alliance wall until the autonomous middle line (draw this in the log or label it).

- The team that has the greater score immediately after the autonomous phase will win 6 points that contribute to the final score of the match
- If the home row is scored, then 1 WP is awarded to the alliance

Afterwards, there is a 1:45 user control period in which the robots will continue to compete and try to end with a score greater than the opposing alliance. Getting a greater score will result in 2 WP -- these contribute to the final tournament rankings.

Constraints When Building the Robot

- There must only be one robot per team and the components/subsystems of the robot are limited to
    - A robot base that allows for the robot to move throughout the field
    - Power and Control system of the robot, which includes VEX Batteries, Brain System, and the appropriate electronics
    - Any additional subsystems that are meant to play the game and manipulate game elements
- Robots must be safe and must fit within a 18″x18″x18″ volume at the beginning of a match
- Robots are built from the VEX V5 or Cortex System (components can not be altered unless specified in the VEX Game Manual)
    - Parts that are exceptions to this rule are color filters for the VEX Light Sensor

    - Any non-aerosol based grease or lubricating compound, when used in extreme moderation on surfaces and locations that do NOT contact the playing field walls, foam field surface, Balls, or other Robots.
    - Anti-static in moderation
    - Hot Glue to secure cable connections
    - Unlimited amount of 1/8" braided, nylon
    - Additional Commercially available items that are used with the role intent of assisting in wire-bundling and or management
- Certain Plastics are allowed too if they fit from a 12"x24" sheet upto 0.07" thick
    - Shattering plastic is not allowed (PMMA)
    - ABS, PEEK, PET, HDPE, LDPE, Nylon, Lexan are just an example of a couple
- Non Functional decoration is allowed and tape is limited to labeling and assisting with electrical management (like labeling motors and taping wires)
- The preceding rules were those that are most relevant to our team: a more comprehensive/fuller list can be found in the Vex Official Game Manual

Objective for Today

- Analysis of what our team believes to be the greatest priority within this season's challenge
- Goals that we hope to achieve with our robot
- Decision Matrices and generating ideas through brainstorming
- Initial timeline &/or deadlines based on the goals we set

Team Specific Game Analysis (Team, 6-10)

General Analysis of the Game

This year's game is different from the previous season's game (Tower Takeover) since this game is more back-and-forth. What is meant by this is that robots have limited scoring elements and the score will fluctuate back and forth since what determines overall points is the number of scored balls and the number of owned rows of goals. Since this is a number that is greatly prone to change, we suspect that one evident attribute of a successful robot in this year's game would be scoring/cycle speed (and just general speed as well since that contributes to scoring speed). The reason for this is because if a robot is able to create more points than the opposing alliance's robot can create/prevent, ultimately, what will happen is that the robot that is superior in terms of "speed." Well, that prompted our team to ask the question of what exactly is speed and how can that be achieved? Speed ultimately comes down to two big factors: the design of the robot itself, and drive practice - how familiar our driver is with the robot. With this, our team generally understands how paramount the seemingly general notion of "speed" is to this game.

Another byproduct of such a back and forth game is that defense will be crucial, as defense, in a back-and-forth game with an open field, will essentially allow for a greater scoring deficit that could be created. Elaborating on this, defense creates a score deficit as it prevents the other team from scoring. Speaking in a broader context, in VRC, historically, defense has been an integral strategy for teams in back-and-forth games (specifically Turning Point and even In the Zone to some extent). With this, our robot should be either able to be defense prone and or play defense.

Now, there are two more things that our team considered that were more specific to this year's game. Firstly, being able to have an owned row within this game is huge: it results in a bonus of 6 points. With this, the ability to be able to take out balls from the goal and then put back those balls in a different order such that your alliance's color is on the top would be something that would be ideal to do within this game as it would allow for a team to manipulate the limited game elements in a way that is quick and efficient to give a team a small competitive edge. Let us denote this action as *shuffling.* The team personally believes this action is what will give teams the clear ability to greatly influence the result in their favor by owning rows.

Moreover, autonomous is extremely important within this game as being able to own the home row within the autonomous results in one win point. Additionally, being able to get the home row results in an additional connected row for the end of the match, and likely autonomous bonus (as we speculate that should be sufficient to win early season).

Based on the analysis our team had, we developed the following criteria of what we believe would best approach the design problem presented by this game.

Generated Criteria For the Robot

- Ball capacity as game elements are limited and since the game is a back and forth game; however, we do not believe this will be a large priority **(generated by Andres)**
- Drivetrain Speed; the reason for this is because the game is back and forth and we believe the ability to be "quick" will be extremely important **(generated by Conner)**
- Ability to "Shuffle" the Goals + Intaking: once again, the game is a back and forth game and specific for this game, owning "rows" are crucial and we believe shuffling/intaking will be important **(generated by Aayush, Archis, and Ashwin)**
    - Ability to collect & filter; we believe this will give a small competitive edge during shuffling **(generated by Sterling and Saachi)**
    - Ability To Descore (this falls partially falls under shuffling and intaking)
- Autonomous Capabilities **(all members of the team unanimously believe this will be something extremely important since)**
- Defensive Capabilities **(all members of the team unanimously believe this will be something greatly important)**

After developing this criterion, we combined similar criteria and then ranked them on what we deemed their level of importance was relative to each other.

Ranked (most to least important)

1. Autonomous Capabilities
2. Ability to “Shuffle”/Collect and Filter
3. Chassis Speed
4. Scoring Speed
5. Defensive Capabilities

Goals for the Robot (Based on the Robot’s Generated Criteria)

This season we want our robot to be able to outscore our opponent by scoring balls in goals and taking goal ownership to score six-point rows. Our robot should be able to effectively manipulate game components in order to quickly and consistently switch ownership of goals by de-scoring and scoring balls at the driver’s will. Additionally, the robot should be able to quickly maneuver around the playing field to arrive at various goalposts in an attempt to gain and sustain ownership of these goals. Our robot should also be designed to have a three or more ball capacity in order to fully fill any hollow goal. Finally, this robot should also be able to hold its own in autonomous control by taking ownership of at least two of our home row goals using autonomous programming. A competent autonomous is crucial as possession of the home row produces a win point for the alliance which completes it.

Brainstorming Robot Decisions & Evaluation Through Decision Matrices

Breakdown of Different Robot Types (Team, 6-17, 6-24)

Lift Robot

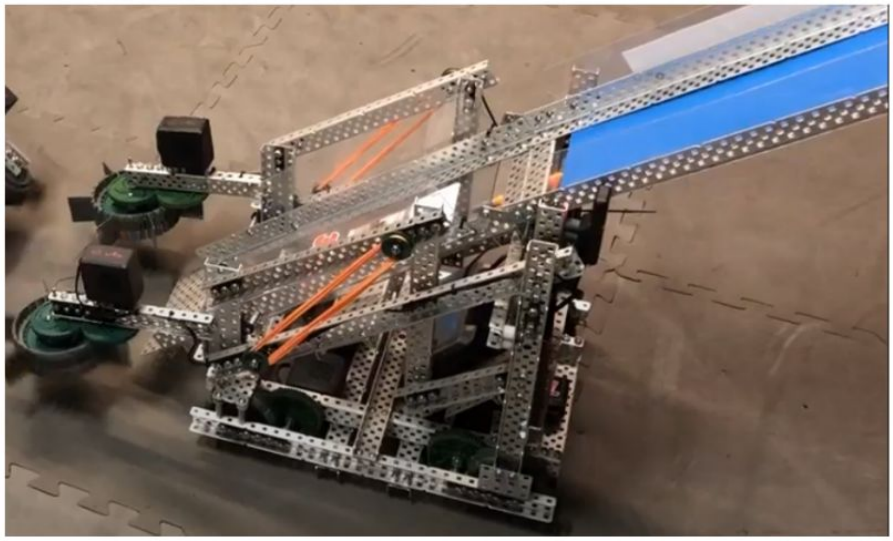
- Four or Six Bar Lift
- Has one Intake that can both score and descore
- Uses a tray to store balls
- 3+ ball capacity

Pros:

- Doesn't require a separate motor for descoring
- Can block robots from scoring by raising lift over the goal

Cons:

- Needs lift to be down in order to descore and intake and therefore cannot score and descore at the same time or score and intake more balls at the same time
- Can fall out of the field/tip on the field
- Slower scoring speed than shooter robot



Courtesy of team 3141S, which posted an early season reveal of their robot during the month of June.

This picture of the robot to the left was taken from their team.

Shooter Robot

- Descoring intakes at the front of the robot feed balls into the indexer which feeds balls into the shooter to score
- 3-4 ball capacity

Pros:

- Can descore and score at the same time
- Fastest scoring speed

Cons:

- Lower capacity than other designs
- Needs 2 motors for descoring intakes

This picture of the robot to the right was taken from a reveal of a VEXU team named "YNOT." They posted an early season reveal on YouTube similar to team 3141S.

Chain-Bar Robot

- Uses a position orienting lift
- Can use a tray to have a 3+ ball capacity (plus the intake motors are on the chassis)
- Can have room for a special hoarding area

Pros:

- Can have a higher ball capacity
- Can defend goals by lifting tray over them

Cons:

- Needs 2 motors for descoring intakes
- Can fall out of the field/tip on the field
- Slower scoring speed than the shooter robot

This robot goes by the name of "Dark Knight" and it is a VRC team. The team number is 46535K, and they are an example of a chain bar. This image of them scoring was snipped from their YouTube reveal.

Ramp Robot

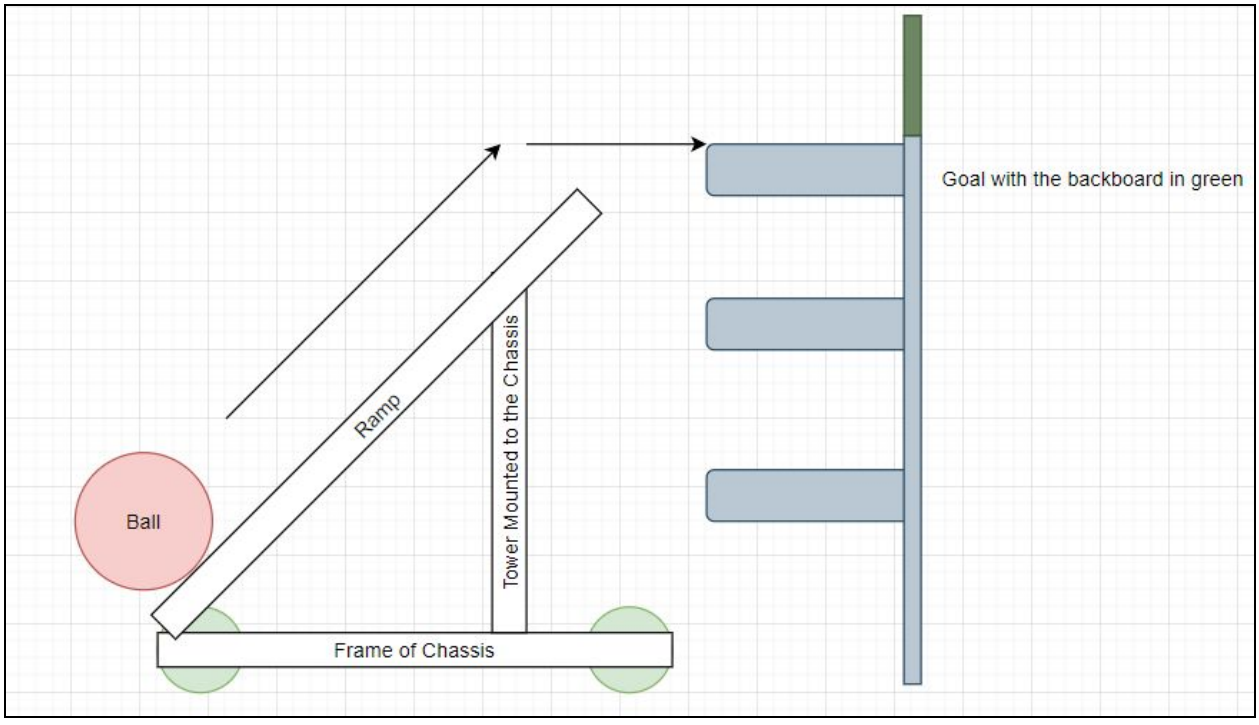
- Little moving parts
- Simple design
- Heavy

Pros:

- Strong defense because of weight
- Easy to build

Cons:

- Slow scoring speed
- Slow chassis speed because of weight



Generated by Conner, a team member of 2496W

Weighting the Criteria

The criteria that will be used to evaluate which robot type our team (2496W) wishes to design, build, and program for this season will be modeled off of the aspects of the game that we decided to be the most important (previous document).

Here it is once again:

1. Autonomous Capabilities
2. Ability to "Shuffle"/Collect and Filter
3. Chassis Speed
4. Scoring Speed
5. Defense Capabilities

For our decision matrices, we've decided that all criteria will have a max score of 10 so that it is easy to compare how a design stacks up between criteria. For example, if a design scores a 5 in autonomous capabilities and a 5 in chassis speed, it means the design has equally strong autonomous capabilities and chassis speed. However, each criterion has a different weight multiplier depending on how much it is valued as seen in the list above. This multiplier will be applied to the base score that it is initially given when the total is being added up. To summarize, the initial value from 0 to 10 will be visible in each individual criteria column, and the multiplied value will be added when it comes to tally up the total.

The following numbers assigned to each criterion within the design matrix are based on the relative importance of each criterion. Just to review, autonomous is game-breaking this season. Teams have the potential to win a win-point and gain such a massive advantage when starting out in the game; hence its highest ranking. Next comes the ability to collect (intake) and

shuffle. This is the second-highest score as being able to manipulate game elements are essentially the fundamentals of being able to score. With this, the second-highest weight. Next come speed and scoring speed and combined, they should have the arguably the most weight since speed encompasses all of the criteria we included (autonomous is easier with speed, shuffle comes with scoring speed, and defense depends on speed and torque).

Type of Robot Design Matrix

Key

Bolded Score - Best Score for That Criteria

Highlighted Row - Winning Design

Robot Type/ Criteria (out of 10 each)	**Auton Capabilities (7x)**	**Ability to Collect/Shuffle (5x)**	**Chassis Speed (4x)**	**Scoring Speed (4x)**	**Defense (3x)**	**Total (out of 230)**
Lift Robot	7	7	**9**	7	8	172
Shooter Robot	**9**	**9**	**9**	**9**	7	**201**
Chain Bar	7	7	8	7	8	168
Ramp Robot	5	3	7	4	5	109

Lift Robot Score Breakdown

- Autonomous Capabilities: a lift robot falls short in autonomous ability as it must constantly lift up and down in order to score into goals. This slows down the robot making it less effective during autonomous
- Ability to Collect/Shuffle: once again the robot is hampered by the fact that it must constantly move up and down to level the intake with the various entryways. To shuffle, the bot must go down and must later go up in order to score, making the shuffling speed slower overall.
- Chassis Speed: The chassis speed does not really depend on the type of robot in play as long as the difference in weights for the robot are marginal. In other words, since there are many ways to reduce weight on this type of robot (using 1/2 C-Channels, polycarbonate, etc), weight is not a significant factor and hence the score of a 9.
- Scoring Speed: The reason why this design gets the same score as the chain bar but a lower score than the shooter robot is because the lift robot, in order to reach the goal height of a height slightly above 18 inches, needs to lift in order to move the balls, the game elements, up.
- Defense: a lift robot has stronger defensive capabilities through its ability to cover goals preventing the opposition from scoring in these goals. Additionally, the lift is able to protrude outwards creating a wall of sorts. This effectively covers certain areas prohibiting opposing robots from entering those areas. While this may be true, while playing defensive, a lift robot has the possibility to tip

over as the robot is unbalanced when fully extending. As such this type of defensive play is less reliable.

Shooter Robot Score Breakdown

- Autonomous Capabilities: because of it's shuffling and shooting capabilities, a shooter robot has high potential to score the home row in autonomous
- Ability to Collect/Shuffle: a shooter robot will have shuffling capabilities because of the way the shooter mechanism intakes the balls in the goalposts
- Chassis Speed: Since shooter robots do not have a lot of heavy subsystems, the chassis speed will be faster compared to other robots with more parts and subsystems.
- Scoring Speed: Shooter robots have fast scoring speeds because shuffling the balls in a goal is relatively fast as the robot encompasses the goalpost in a "circle"
- Defense: The shooter's defensive capabilities are directly connected to its fast cycle and shuffling speeds enabling the robot to quickly descore and shuffle goals. Outside of descoring, the robot has a lesser defense.

Chain Bar Robot Score Breakdown

- Autonomous Capabilities: Slower at scoring but able to score from farther away because of the length of the chain bar.

- Ability to Collect/Shuffle: Takes longer to shuffle because for each ball you need to move the chain bar and it takes up much more time compared to the other options.
- Chassis Speed: Slightly heavier robot due to the lift so it makes it slightly harder to move the chassis but not a very noticeable difference if made properly.
- Scoring Speed: Slower than the other options due to the fact that it is required to move the chain bar to score which sacrifices speed.
- Defense: The chain bar robot received a score of 8 in the defense category because the chain bar doesn't have as many defensive capabilities as the other types of robots

Ramp Robot Score Breakdown

- Autonomous Capabilities: A ramp will need a wind-up period and would be limited by an intake on the opposite side of the robot or wait for a trapdoor mechanism to move, losing crucial seconds in the 15 second autonomous period.
- Ability to Collect/Shuffle: A ramp would have a hard time shuffling out colors and collecting quickly because having two intakes and a trapdoor would take a long time to cycle out colors and intake balls because of the number of moving parts, the complexity of the design, and length of intake/uptake combo.
- Chassis Speed: Since the robot has a big metal ramp on it, it will cause the robot to lose speed for moving around, whereas the other robots are more compact and maneuverable.

- Scoring Speed: Will be slower than the others because the ramp will naturally have prolonged unloading speeds unlike the shooter robot and will have to increase in elevation to score, taking away more time.
- Defense: The ramp is the only good defensive part of the robot, but won't be able to hold on its own from the other aspects and won't be able to catch up with the other robots.

Explanation

Based on the decision matrix, the robot that won was the "shooter" robot as it got the highest quantitative rating based on the criteria that we assessed the robot on. Overall, when trying to pinpoint what feature of this robot is the reason for such a high score, our team collectively agreed that this type of robot is significantly more efficient than the other types of robot since this type of robot does not require moving a lift. Rather, the uptake mechanism will move up the ball and score it, and this, from our analysis will be significantly easier to work with in all aspects of the game: programming and driving. In addition, with this type of robot we do not need to worry about changing the center of gravities since there will be no lift. This will make driving and programming easier since there should not be the worry of needing to remedy the issues that may arise due to a non-center center of gravity. Moreover, where this design truly shines is within the autonomous phase and the ability to shuffle. The reason for this is because the build is simple and the center of gravity of such a robot will likely be in the middle of base. This means wheel slippage with encoders will be less of an issue, and hence a more consistent autonomous. As for the ability to shuffle, this stands out because descoring the ball, uptaking, and scoring are all one continuous motion, and with this, type is not wasted by

the driver having to re-align and etc (but with a lift type robot, the driver will first need to descore, then align to be in a position to score, and then score).

Planning the Specific Subsystems of the Robot

Based on the type of robot that was chosen, the specific subsystems that the robot will have is different. For instance, the lift robot will have some type of lift while the shooter robot will not. Here, the design that seemed to suit the needs of the team the best was the shooter robot, and based on the general design that has been circulating for this type of robot, the subsystems that are required for this type of robot are:

1) Chassis
2) Flywheel
3) Intake
4) Uptake

Now for what can be modified for each type of subsystem, here are even more specifics:

1. Chassis
    a. Chassis Shape
    b. Wheel Size
    c. Wheel Type
    d. RPM + Gearing
2. Flywheel
    a. RPM
3. Intake

   a. Types of Intakes

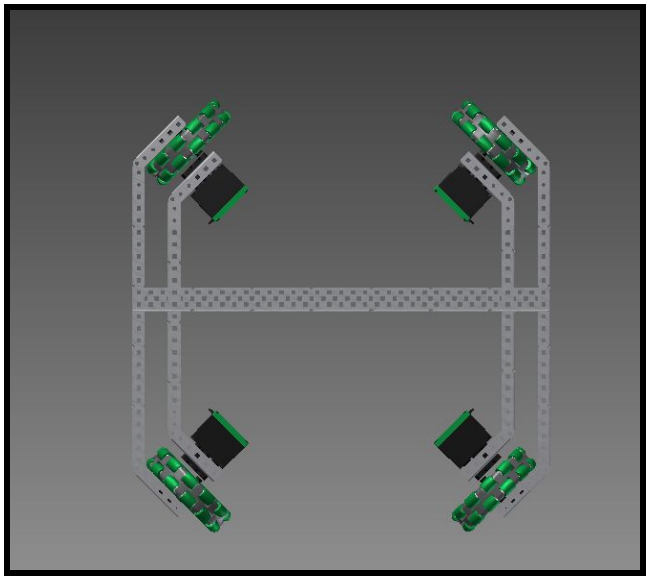
4. Uptake

   a. Different Types of Uptake Mechanisms

Chassis Shape Breakdowns

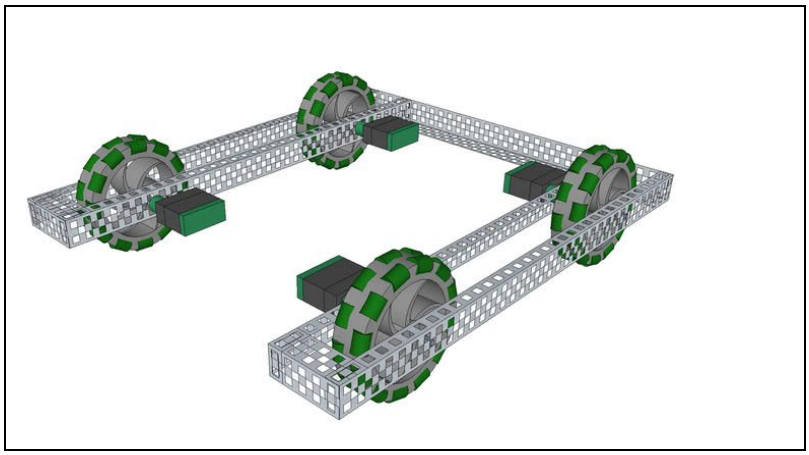
X chassis:

- Pros:
  - Can strafe
- Cons:
  - Slower general speed
  - More prone to defense since it can strafe and move in more directions
  - There is a bar in the middle, and that leaves us less room for the shooter subsystem



U Chassis

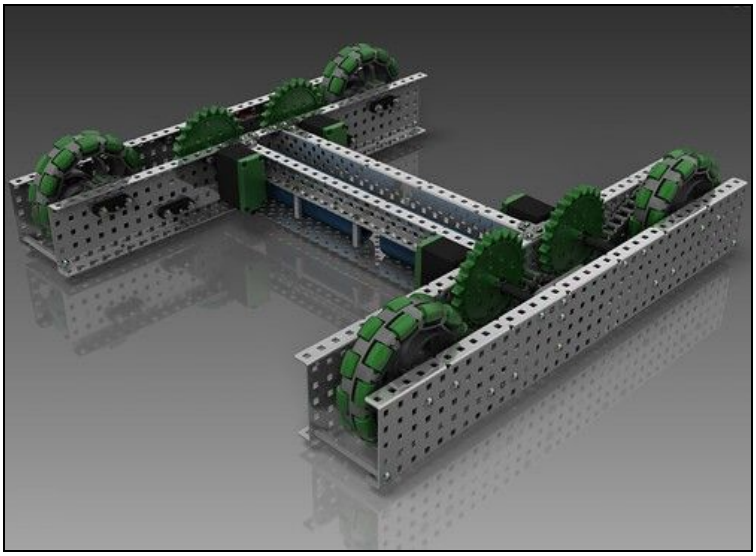
- Pros:
  - Simpler to build than the other design
  - Has room in the middle for the shooter subsystem (more building space and easier tolerance)
  - Higher speed than the X-drive
- Cons:
  - Can not strafe (this is not a major impediment)



H chassis

- Pros:
  - Sturdy and reliable
  - Able to operate in the front and the back of the robot
- Cons:
  - Bar can get in the way

- Can't strafe



Chassis Shape/Design Decision Matrix

Robot Type/ Criteria (out of 10 each)	Autonomous Capabilities (7x)	Ability to Collect/Shuffle (5x)	Chassis Speed (4x)	Scoring Speed (4x)	Defense (3x)	Total (out of 230)
X Chassis	10	10	7	9	6	202
U Chassis	9	10	10	9	9	207
H Chassis	9	7	10	9	8	198

Explanation

Based on the decision matrix, we decided to pick a hybrid of the U and H chassis. What we mean by this is that we will keep the signature U chassis design of having one bar in the back because the decision matrix shows this kind of design excels in all criteria. However, we also are planning to add a bar that is lower than the back bar in the middle of the chassis to provide more support and also a mounting area for our ball manipulator. Mounting the middle bar lower nullifies the main disadvantage of the H chassis design which was that the middle bar took up space and interfered with the space of our manipulator. For all of these reasons, we decide on a U and H chassis hybrid.

Wheel Size for Chassis Breakdown

4 inch (standard wheel size):

- Uses an omnidirectional or traction wheel
- Commonly found on VEX Robots
- Can be used with High Strength axles
- Pros
  - Speedier
  - High ground clearance
  - Common
- Cons
  - Higher center of gravity

3.25 inch (2496Y Tower Takeover)

- Uses an omnidirectional or traction wheel
- Uncommonly found on VEX Robots
- Can be used with High Strength Axles
- Pros
  - Lower center of gravity
  - Moderate speed
- Cons
  - Uncommon
  - Low ground clearance

2.75 inch

- Uses a traction or omnidirectional wheel
- Rarely found on the chassis of VEX robots
- Can use High Strength axles
- Pros
  - Lowest center of gravity possible
- Cons
  - Very low ground clearance
  - Lowest speed

Explanation

The 3.25 inch wheels were chosen because their lower speed is beneficial to new drivers because the lower speed would make driving easier. Another benefit of the 3.25 inch wheels is the lower center of gravity would make driving for our new driver easier by allowing the robot

to handle better, but not at the expense of lots of speed. The 4 inch and 2.75 inch wheel were not chosen because they did not achieve the balance above.

Wheel Type for Chassis Breakdown

All Omnidirectional Wheels (2496Y Tower Takeover):

- Pros:
  - Less turning scrub
  - Easier and more predictable turns helps with auton
- Cons:
  - Lower traction makes it easier to push
  - Rollers allow bot to be pushed sideways

Omnidirectional/Traction Wheels Split (2496J Turning Point):

- Pros:
  - Can turn better than a full traction wheel chassis
  - Helps against defense
- Cons:
  - Still turns worse than a fully omnidirectional wheel chassis
  - Turning speed is slower
  - More difficult to program autonomous with them due to the odd center of turning

All Mecanum Wheels (2496Y Turning Point):

- Pros:

        - No additional friction when turning
        - Strafing capabilities
    - Cons:
        - Bumpy movement
        - Heavy and in result slower

All Traction Wheels (2495X TP):

- Pros:
    - Higher traction makes the robot harder to push
- Cons:
    - More turning scrub
    - Different turning angle compared to omni/different turns
    - More difficult to program autonomous with them due to the odd center of turning

Chassis Wheel Type Decision Matrix

Robot Type/ Criteria (out of 10 each)	**Auton. Capabilities (7x)**	**Ability to Collect and Shuffle (5x)**	**Chassis Speed (4x)**	**Scoring Speed (4x)**	**Defense (3x)**	**Total (out of 230)**
Omni/Traction Split	7	7	8	7	9	171

All Omni	9	9	10	9	7	205
All Mecanum	9	9	7	8	9	195
All Traction	5	5	5	5	10	143

Explanation

Omni wheels were selected for the chassis due to the overall speed and maneuverability these wheels provided.While the rollers on the wheels do make the robot slightly easier to push around, the high movement and rotation speed compensate for this. Additionally, the simple and effective turning capabilities of these wheels allow for easier autonomous programming.

Wheel Gearing for Chassis Breakdown

200 rpm (Standard Cartridge 1:1)

- Pros:
  - More control
  - Direct Drive ratio allows for more space for other components
  - More torque for defense
- Cons:
  - Slow speed
  - Less scoring capabilities

300 rpm (Torque Cartridge 3:1 or 36:12) [2496Y Tower Takeover]

- Pros:

        - Small gear ratio allows for more space

333 rpm (Standard Cartridge 5:3 or 60:36)

- Pros:
    - Fast speed allows for significantly faster cycle time and scoring speed
- Cons:
    - Could be uncontrollable if using 4 inch wheels
    - Requires gearing

257 rpm (Turbo Cartridge 3:7 or 36:84)

- Pros:
    - Faster than most chassis
    - Needs less maintenance
- Cons:
    - Large ratio to build and therefore requires more space
    - Could be uncontrollable if using 4 inch wheels

360 rpm (Turbo Cartridge 3:5 or 36:60)

- Pros
    - Fastest possible gear ratio
    - More scoring capabilities due to speed
- Cons
    - Large gear ratio requires more space
    - Harder to control exact position on field - not good for auton

250 rpm (Standard Cartridge 3:2 or 36:24)

- Pros
  - Faster than most chassis
  - Compact gear ratio
- Cons
  - Uses low strength gears
  - Requires more maintenance

Wheel Chassis Gearing Decision Matrix

Gear Type (out of 10 each)	**Auton. Capabilities** (7x)	**Ability to Collect/Shuffle** (5x)	**Chassis Speed** (4x)	**Scoring Speed** (4x)	**Defense** (3x)	**Total**
200 rpm (Standard Cartridge 1:1)	7	N/A	8	8	8	137
300 rpm (Torque Cartridge 3:1)	9	N/A	9	9	9	162
333 rpm (Standard Cartridge 5:3)	**10**	N/A	**10**	**10**	**10**	**180**
257 rpm (Turbo cartridge 3:7)	8	N/A	8	8	8	144

360 rpm (Turbo Cartridge 3:5)	10	N/A	9	9	9	169
250 rpm (Standard Cartridge 3:2)	8	N/A	9	9	9	152

Explanation

The reason that 333 rpm is the gear setting that will be chosen is because when choosing such a gearing for the robot, we want to ensure that the chassis will be fast enough so that we have the competitive advantage of speed while not having a speed that is uncontrollable. From our experiences with each of the settings, we had to assess how driving and programming with each setting will be. Overall, from our experiences using lower 300 RPMs with smaller 3.25 wheels result in a linear speed that is slightly faster than direct driving 4 inch wheels that are 200 rpm. Moreover, using 60t to 36t gears with smaller wheels will allow us to stack the motors, resulting in more space for us in the middle for our ball manipulator. Many of the other gear ratios considered here would result in us not having enough space for our flywheel mechanism or would require a significantly more complex design to fit them for results that would be worse than that of the 333 rpm gear ratio on 3.25 inch wheels. (this will be similar to the mechanism used for the drivetrain on 5225 In the Zone Robot).

Flywheel RPM Breakdown

600 rpm (turbo cartridge 1:1) [YNOT Change Up]

- Pros:
  - Lower speed enables robot to score at a more controlled pace
  - Easiest to build as there is no added gearing involved
- Cons:
  - Must get closer to score

1800 rpm (turbo cartridge 3:1)

- Pros:
  - faster scoring rate
- Cons:
  - Speed takes away control
  - Added gearing
  - Unnecessary speed

9800 rpm (standard cartridge 49:1) [2495X Turning Point]

- Pros:
  - Fast speed enables wheel to shoot balls from a distance
- Cons:
  - Added gearing
  - Shooting is unnecessary
  - High speed takes away control

Flywheel RPM Decision Matrix

Robot Type/ Criteria (out of 10 each)	Autonomous Capabilities (7x)	Ability to Collect/Shuffle (5x)	Chassis Speed (4x)	Scoring Speed (4x)	Defense (3x)	Total (out of 160)
600 rpm	9	8	N/A	9	N/A	139
1800 rpm	7	7	N/A	9	N/A	120
9800 rpm	6	6	N/A	9	N/A	108

Explanation

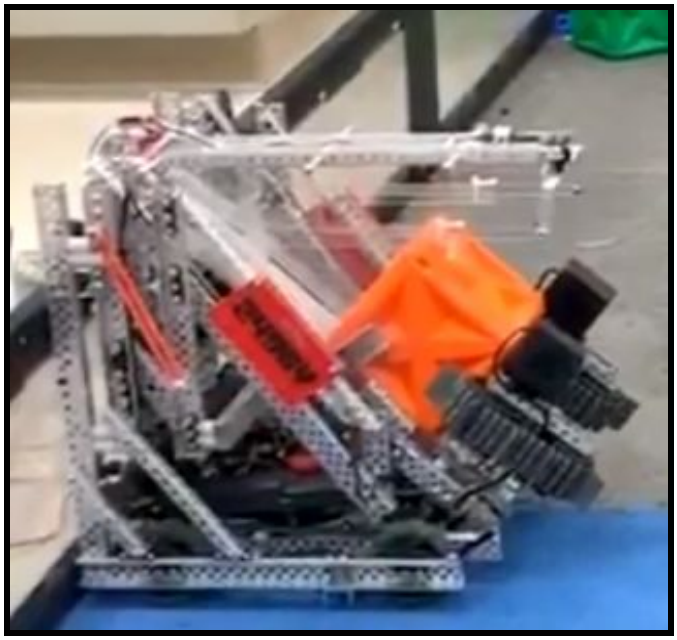
Based on what our team analyzed from early season reveals, all of the options other than 600rpm are significantly faster than the 600RPM, and based on the material of the ball (like how light they are) and how tall the goals are (we do not need to "shoot" the balls far); opting for a lower rpm would be the best. Moreover, most other robots such as the "YNOT" robot have 600 rpm flywheels, and their gear settings seem to be the most optimal.

Intake Breakdown

Single Stage Chain Intakes (2496Y Tower Takeover)

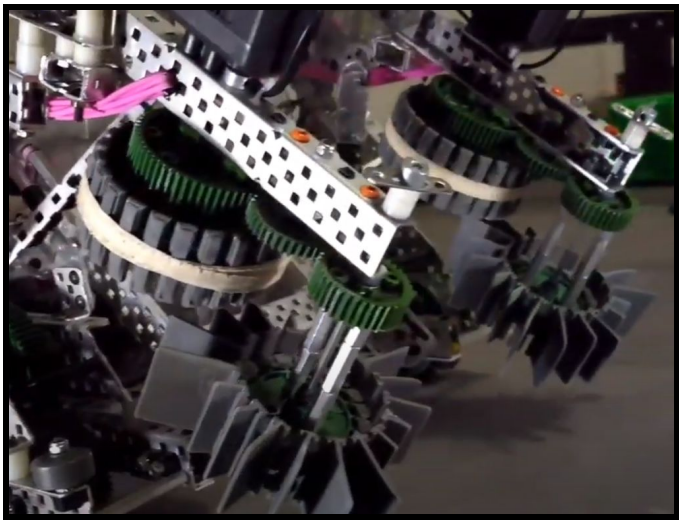
- Pros
  - Compact
  - Can grip extremely well

- Cons
    - Harder to maintain and build
    - Complex design



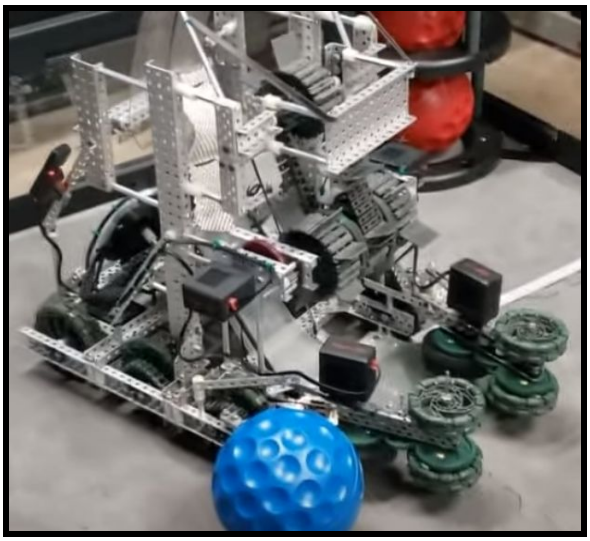
Two-Stage Chain Intakes (369A Tower Takeover)

- Pros
    - Can have two different speed intakes
    - Can grip extremely well
- Cons
    - Harder to maintain and build
    - Complex design



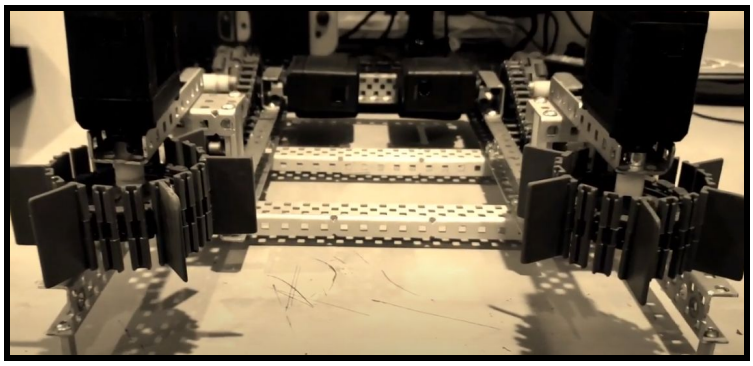
Small Omni Wheels (742A RI3D)

- Pros
  - Simple
  - Easy to build
  - Easy to maintain
- Cons
  - Space inefficient
  - Heavy



Circular Single Stage Intakes (21S Change Up)

- Pros
  - Weighs the least
  - Simple design
  - Easy to maintain and good repair quality
  - Smallest conveyer
- Cons:
  - May not have a good grip



Explanation

The circular single stage intakes were chosen because of their weight savings compared to the other designs. Additionally, the circular single stage intakes are simplistic and would take less time to build and maintain for similar performance.

Uptake Breakdown

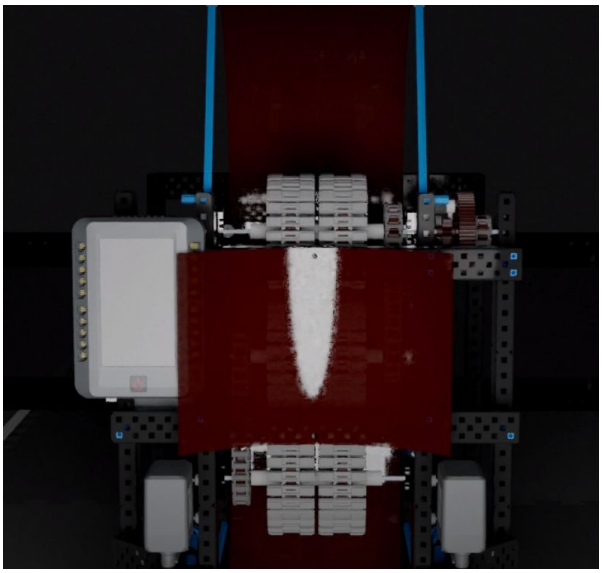
Generic Roller (YNOT Change Up)

- Pros
  - Customizable and extremely robust
  - Simplest to build while not compromising performance
  - Indexer possibilities to enhance driving
- Cons:

  - Need a custom gear for bottom

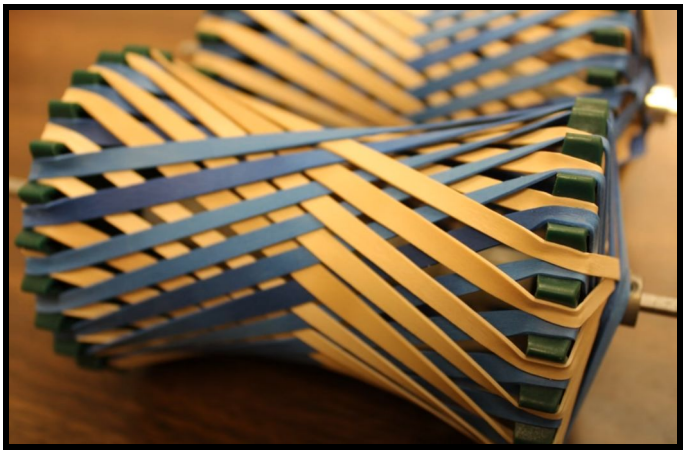
Tank Tread (5090X Change Up)

- Pros:
  - No significant pros over the other mechanisms
- Cons:
  - May be heavier over all and lead to the shaft that connects the mechanism to bend (more issues)
  - Harder to maintain



Braided Roller (1483 In The Zone)

- Pros:
  - Helps guide the balls towards the center
- Cons:
  - Difficult to braid the rollers
  - High maintenance
  - No other benefits aside from centering



Explanation

The generic roller was chosen because it is arguably the most simple design, and based on the other robots that we have seen using such a mechanism, there is no significant performance difference between this type of roller and other types of uptakes. Moreover, the compression that the braided intakes provide are not necessary. Additionally, one issue we see with the tank treads is that they will be significantly heavier than the rollers; this may lead to the shaft connecting the mechanism to bend. We want to avoid such complications and go with the most simple and robust design.

Recap of Credits

Robot Type Credits

Lift Robot (3141S: VRC Change Up)

Shooter Robot ("YNOT": VEXU Change Up)

Chain-Bar Robot (43565K: VRC Change Up)

Ramp Robot (Member of 2496W, Conner)

Breakdown Diagram Credits

Chassis Shapes (Generic Google Images)

Chassis Wheel Type (Referenced 2496Y in TT, 2496J in TP, 2496Y in TP, and 2495X in TP)

Chassis Wheel Size (referenced 2496Y from TT)

Chassis Gearing (referenced 2496Y from TT)

Flywheel RPM Breakdown (YNOT, 2495X)

Intake Breakdown (2496Y, 369A, 742A RI3D, 21S Change Up)

Uptake Breakdown (YNOT, 5090X, 1483 ITZ)

How We Structured Everything Due to COVID-19 (Ashwin, 9-7)

Effects

Due to the pandemic that is currently occurring, for us, this means that the entire team will be needing to make modifications to our meetings and practice times to ensure safety for all; afterall, we are committed above all else to the safety of our members and those around us during this time.

Overall, these are the things that will be happening due to COVID-19.

1) Limited Practice Time
    a) Typically, in the previous years of the program, an average of 10 unique hours per week would be spent. Moreover, the meeting times would be significantly more flexible since we are lucky enough to have a coach who is dedicated to setting meeting times that are accommodating and convenient for the students. This year, our schedule is like this:
        i) Max capacity of 6 people per practice
        ii) Monday and Wednesday 2-5 pm (6 hours in total)
        iii) Must all wear masks
        iv) No sharing tools
        v) Must be situated at individual tables
2) Limited Number of Competitions: As of now, we are unsure about which events will be open in California and any speculation that we do as a team will likely be inaccurate.

Keeping this in mind, our goal is to get a functional robot and try to get as much practice as possible.

3) Non-Traditional Competitions
    a) Rather than there being in person regular events, there may be virtual events, most notably skills competitions and the virtual competitions that RECF recently announced.
4) Conflicting Schedules
    a) Some members may not be able to go to all practice because of the risks associated with going to practice. We as a team completely understand this, so we will be finding ways to maximize the people that we have and structure the team so that members at home as well will be able to contribute.

Adapting to these Circumstances

To begin with, what we realize is that time, resources, and potential for testing/troubleshooting is obviously important but it is limited. With this, what we realize is that we must plan much more extensively than in previous years and be able to deal with any challenges that we encounter without any trouble. In other words, we need to be especially discerning and keen. Here is our plan:

1) First, we want to maximize our team's strength by breaking members up into distinct "Teams"
    a) CAD: Ashwin, Archis, Conner. These members are responsible for creating models that the build team will reference when constructing the robot in person.

This is how we ensure that we do not waste any time experimenting with things like spacing or sizing. These members have the strongest design sense.

b) BUILD: Andres, Archis, Conner, Sterling, Saachi. They are the ones most willing and available to come to practice. With this, we decided that they will be the ones building.

c) PROGRAMMING: Ashwin, Andres, Tyler Aayush. These are the people behind the software development of the robot.

d) INNOVATE: Ashwin, Aayush, Archis, Andres. Responsible for future plans and movement of the robot.

2) To manage progres, we created a Discord server with an announcement system. Moreover, we used a project board that goes by the name of "Notion."

3) Lastly, we invested in remote programming through a VSCODE extension with the name of "Live Share" which essentially allows everybody on the software development team to be able to edit the code just like a Google Doc.

Chassis Build and CAD Process (Ashwin, Conner: 9-7, 9-9, 9-14, 9-16)

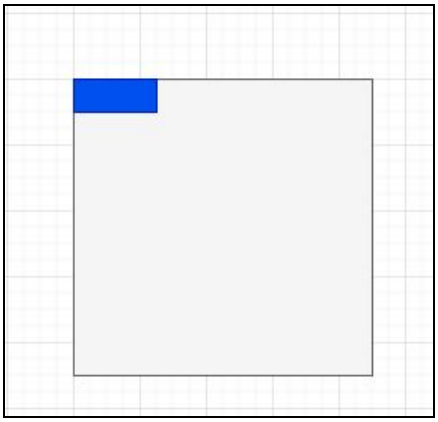
Objectives

- Complete a functional build of the chassis as soon as possible so that we are able to test and proceed with the rest of the build
    - Plan what the chassis will look like through CAD
    - Build the sides separately (left and right).
    - Connect the two sides together and work on bracing
    - Test the chassis immediately to ensure everything is working as intended

Dimensions of the Chassis (Length)

As per the discussion we had with our team previously, we knew exactly the gear ratio and the wheel size that we were going to be using. With this, the only factors that we needed to consider were the dimensions of the chassis, where the motors are situated, and the spacing.

Overall, looking at the designs of other teams such as JTMS, 99999V, 5090X, YNOT we knew that our intake at the front would protrude ~5 inches. To avoid any flipping-out mechanisms, we envisioned our chassis using a 25 hole for the length so that we would be able to avoid flip-outs and thereby increase the consistency of the robot.



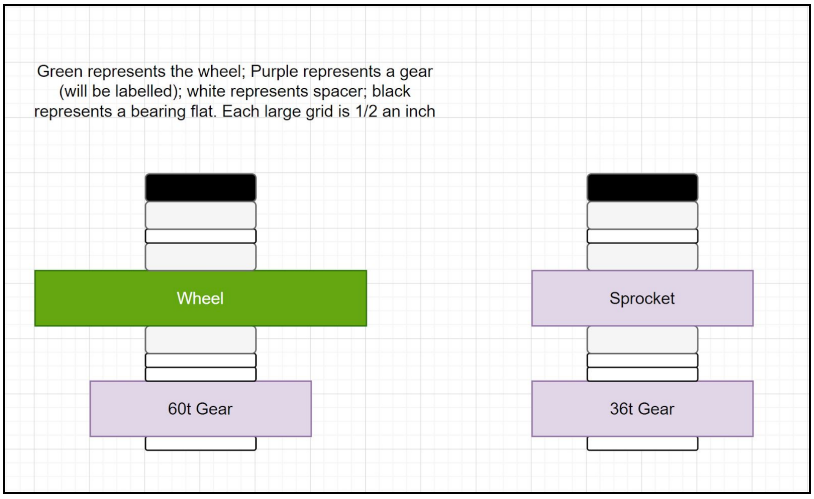
The diagram on the left is to give context of scale. The grey box is 18 inches while the blue represents the space that we predict one intake would consume of the 18 inches. This prediction is based on the research that we did by viewing the open source designs of other teams.

Note: one minor tick within the diagram is equivalent to 1"

As for the width, actually determine that according to the size of the ball. In other words, first what we did was plan the spacing that each side of the chassis would take up and then mathematically reason would be optimal.

Spacing of the Chassis

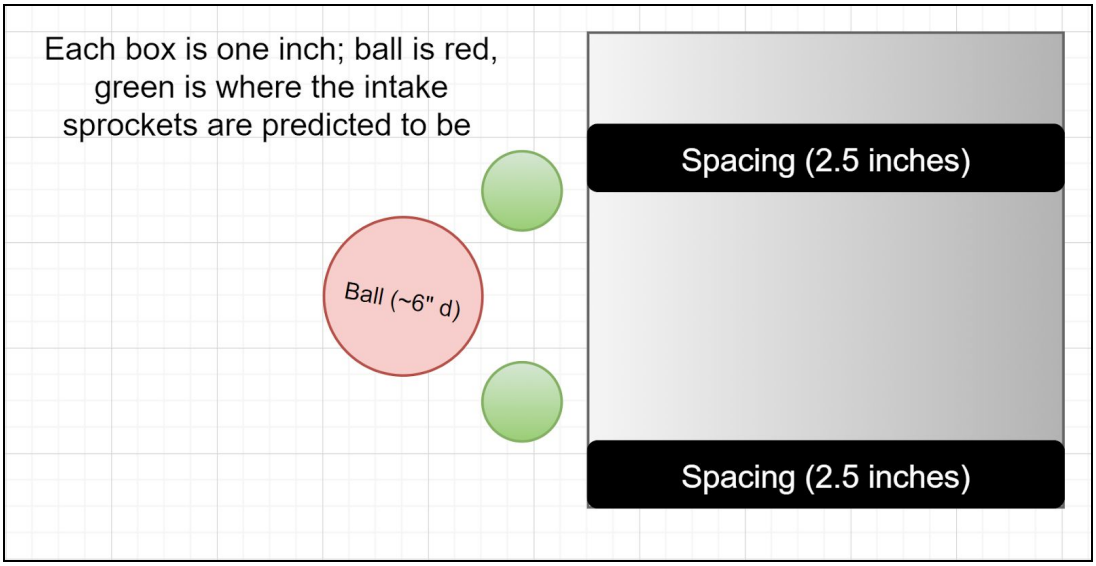
Whenever building a chassis, the convention is that typically the least amount of spacers should be used and that the spacing should be multiples of 0.5. This way the holes on the C-Channel would line up when bracing the chassis horizontally. To determine the spacing of the chassis, we used an online diagram tool (Diagram.net) to determine the thinnest that the two sides should be and then used CAD to complete a more thorough prototype of the model. Here are the results of our trial and error:



Overall, it seems that the smallest spacing we can employ to fit all of the gears and the necessary sprockets while giving enough tolerance would be 2.5 inches. Once again, the scale of the diagram is that one major grid box is equal to 0.5 inches. Knowing this information, we can now determine the width of the chassis to ensure maximum contact with the ball.

Dimensions of the Chassis (Width)

Once again, we used Diagram.net to visualize the dimensions by making a mock chassis, ball, and intakes.

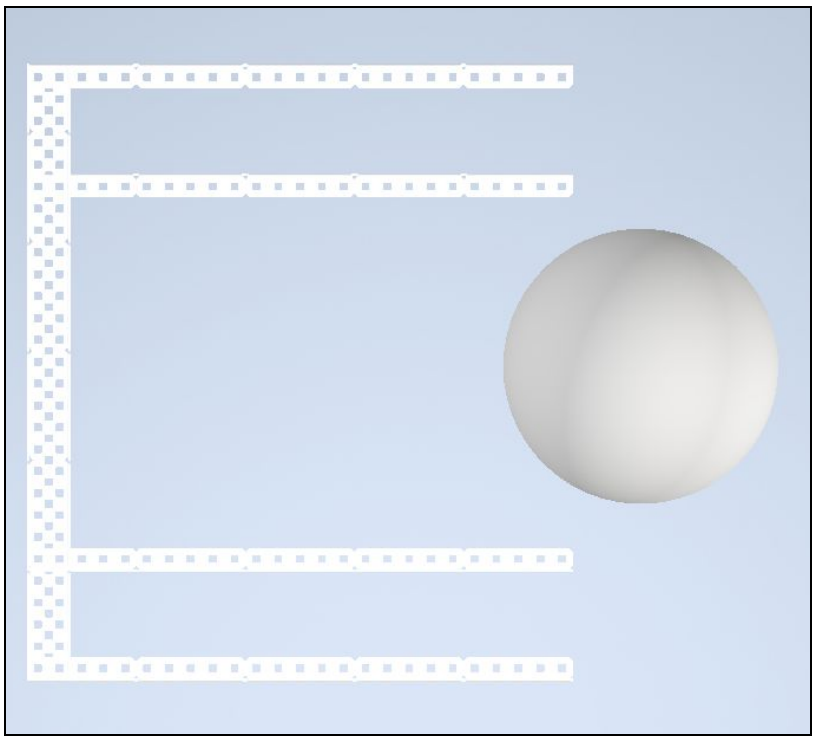


As seen, it looks like there is an excessive of ~4 inches as the grey gradient box is 18 inches. With this, we will be using a 28 hole bar to brace the chassis, making the width of our chassis now 14”. Overall, the dimensions are 12.5” x 14.”

CAD Process

Based on the information just presented, we first began to CAD with a skeleton of our chassis. In other words, it just outlines the dimensions and the pieces of metal that will be going on the chassis. On, we will be building off of this model and add the spacing that we were able to reason through the diagram that was made previously.

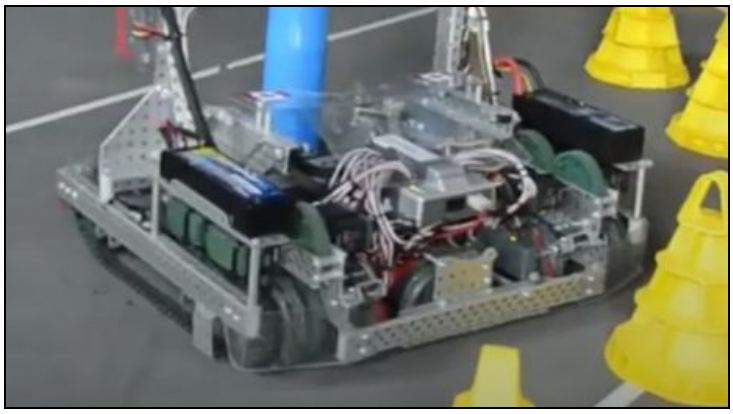
Finally, we can make a ball by having an extrusion revolve around a certain point and get a sense of how the dimensions are looking within a 3D space.



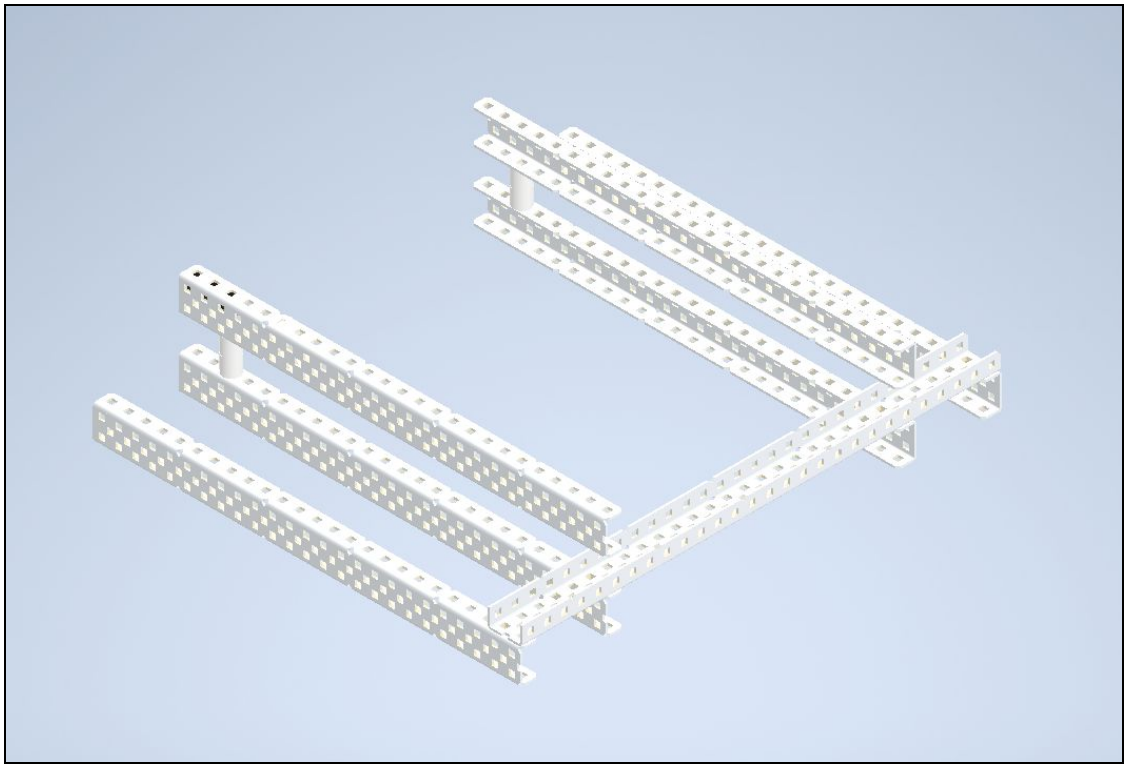
Blue Plane is 18 inches

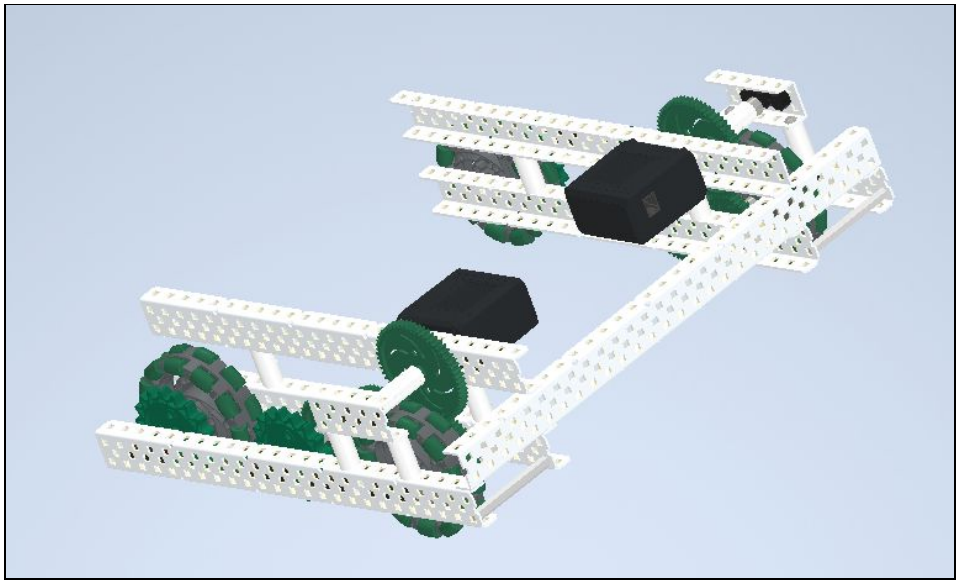
Based on this, the tolerance seems to be as intended and we are within the size limit undoubtedly. With this, when going to practice, we will be building the chassis by following the CAD.

The last thing was that we needed to plan within the CAD is where the motors would go. Since the ball would see contact with the intake and go through the front of the chassis, we know we would need to have the motors as far back as possible. With this, we actually decided to elevate the 60t gears on the chassis so that the motors would fit better. Here is the team that we got the inspiration for this idea from (5225A):



As seen here, in order to have more space for other mechanisms in the center of the chassis, the motors are elevated. However, rather than putting it outside and over the wheels, we will be putting our wheels inside. Then, we will be using chains to make the front wheels drive forward, and this is necessary because, like in the image to the left, all of the gears are towards the back.

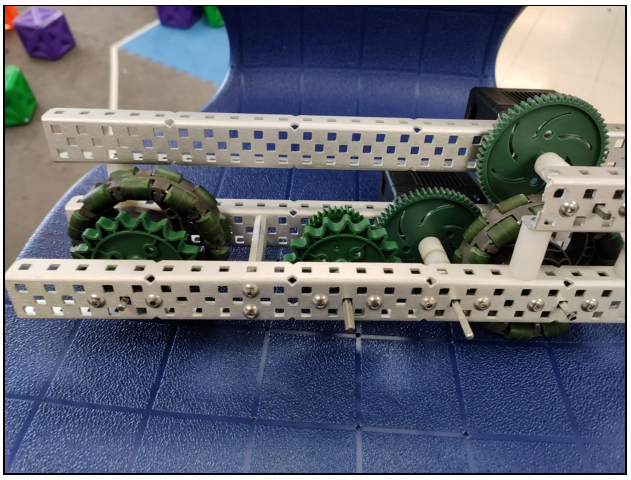


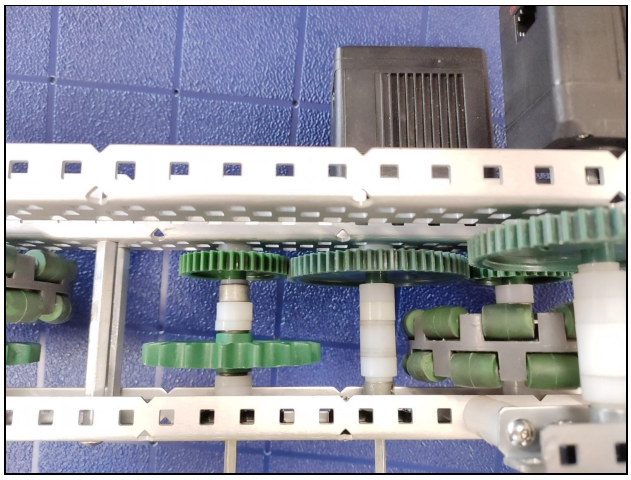


Building

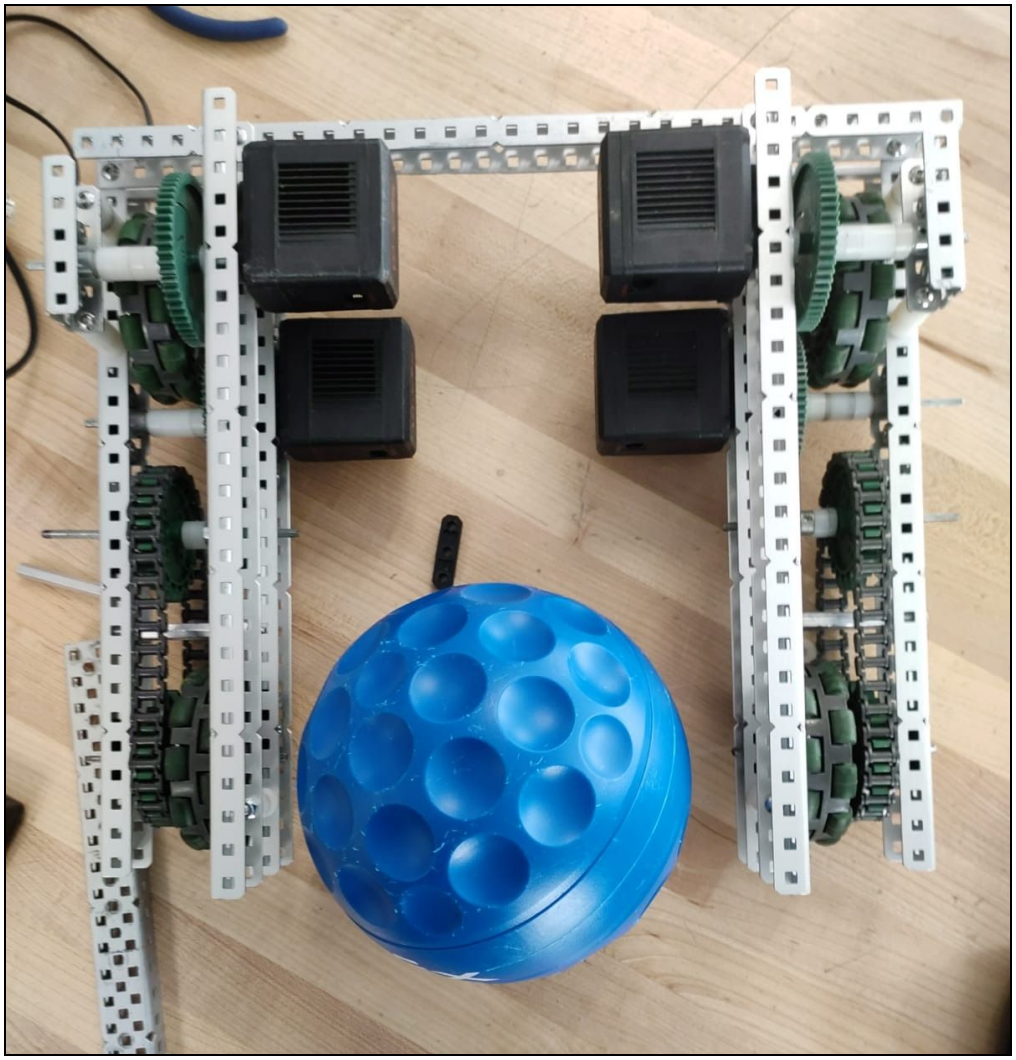
Now, we are able to build everything since there is a comprehensive model at our disposal. However, there is one more thing that needs to be addressed before building the robot. Since manufacturing tolerances can make a significant difference in how a mechanism operates, to ensure that the spacing can freely spin, which indicates low amounts of friction, we reduced all of the spacing by 1/16. We did this by changing any 1/8 to a 1/16.

The process that we followed when building was actually fairly different from the CAD process that we employed. Rather than first building a skeleton of the chassis, we built the sides -- right and left -- individually. Saachi was responsible for building the right side of the chassis while Aayush assisted by mirroring her and building the left side of the chassis to ensure that both parts could be done almost simultaneously. Here are the reference images that we took while in the process of building:





Then, we connected the sides together, forming a structure like this:



Quality Assurance

To make sure that everything was put together correctly, we first checked the friction of the spacing. In other words, everything needed to be uniform. We also spinned the gears and the wheels manually to check if we felt any resistance. If we did feel any resistance, this would mean that the spacing is too great on one side. The solution to this issue for us was simply reducing the spacing by 1/32. Fortunately, this was only an issue for us on one of the sides (the right, elevated gear). By augmenting the modification of reducing 1/16 to reducing (1/16 + 1/32), the gears were now spinning freely.

Based on the preliminary checkpoints, it seems that we should be proceeding with testing the chassis through the use of software. In particular, we want to determine whether the speed of the chassis is appropriate and if the chassis moves straight as intended.

Chassis Testing Process (Aayush: 9-21)

Testing Objectives

Overall, we wish to be able test the following regarding our chassis. First, we want to ensure that it drives straight. Second, we want to make sure that it is as fast as we intend. Lastly, we want to perform "browning" tests, which is essentially a strain test.

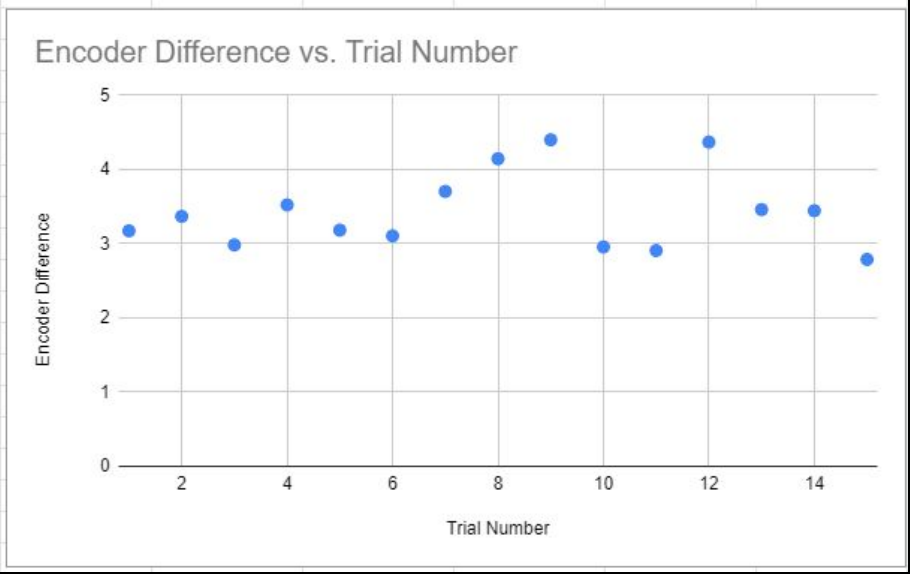
Drive Straight Test

The setup for this is that we want to see the difference in the encoder count of the right side and the left side after both sides of the chassis are going at the same power for any given time. In other words, we will be writing a program that will be moving the chassis forward for X milliseconds; then, we will be storing the encoder difference and print that out to a seperate file. Here is a snippet of the code:

```
#include <iostream>
#include <fstream>

//Using the STL built in logger class.
//Param 1: File Name
//Param 2: lets the class know we are printing out to a file

std::ofstream test_datastream_logger("Test_Data.txt", std::ofstream::out);

//Param 1: "int speed": speed that the motor will go at
//Param 2: "mS": time we will be moving the chassis for

void drive_straight_test(int speed, int mS){
        int time = 0;

        while((time * 25) < mS){

        //function that moves chassis. First parameter is right power and second is left.
        chass_move(speed, speed);
        pros::delay(25); //delay to prevent hogging resources

        //difference between right and left position
        float delta_encoder_count = (float) (get_right_position() - get_left_position());
        test_datastream_logger << "Encoder Difference: " << delta_encoder_count;

        ++time;
        }
        chass_move(0, 0);
        test_datastream_logger.close();

}
```

Here are the results of the test. To ensure that we get a sample of all different types of movements, the mS and speed parameters were changed as well for different trials.

Trial Number	Encoder Difference
1	3.172738698
2	3.367712583
3	2.983498403
4	3.52238874
5	3.182837212
6	3.105058456
7	3.704413765
8	4.144276674
9	4.399640927
10	2.955131243
11	2.907296193
12	4.368086562
13	3.459131565
14	3.443741102
15	2.787690366



As we can see, the encoder difference, or the difference in position between the two sides, always remains less than 5 encoder count. For our chassis, one wheel rotation is equal to 360 * (3/5) = 216 encoder units, this is a fluctuation of 0.07 inches (math was derived using the equation S = r * theta). This indicates that our chassis goes very straight, and it was a number far better than our expectations.

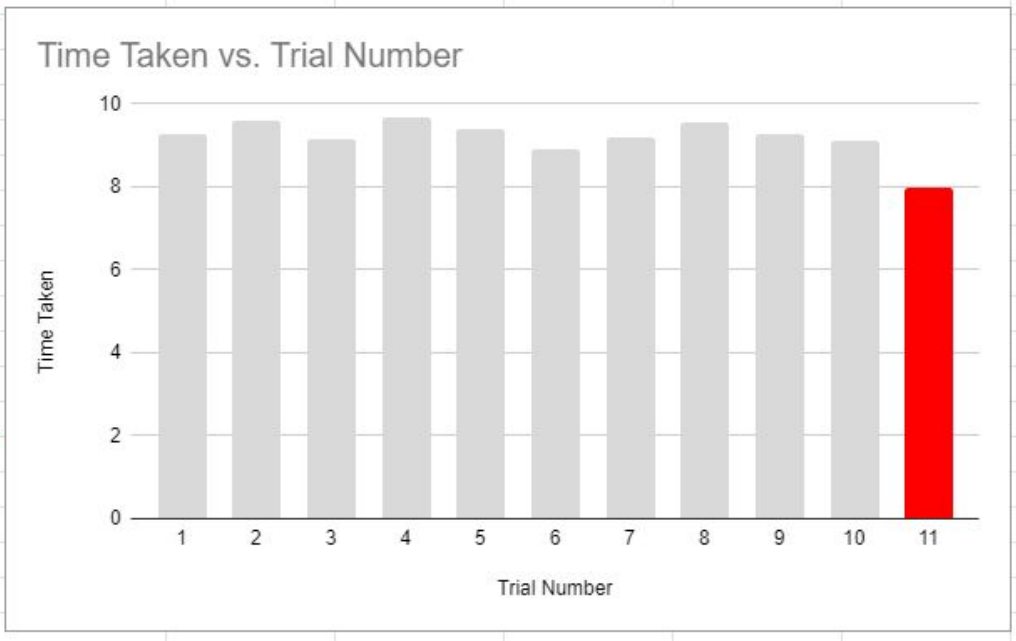
Speed Test

In order to test our speed, we wanted to compare the time it takes to actually move from end to end of the field with the calculated time. To determine the time taken to move across the field, we mounted temporary limit switches that will return true when the robot hits the end of the field. The time when the value is true will be printed to the terminal. This was our code.

```
#include <iostream>
#include <fstream>
#include "main.h"

//creating limit switch object
pros::ADIDigitalIn limit_switch('B') //wired port B

int main(){
        int time = 0;
        while(1){
                //if the limit switch is pressed, break loop
                if(limit_switch.get_value()){
                break;
                } //else move the chassis at full speed (127)
        chass_move(127, 127);

        pros::delay(10); //delay of 10 ms
        ++time; //increment time
        }
        chass_move(0, 0); //stop the chassis
        std::cout << "Time Taken: " << (10 * time); //print to terminal
}
```

For reference, the theoretical value was calculated through this math: we simply used distance = velocity * time and solved for time since we knew distance (144 inches). As for velocity, we took the RPM of the motors, multiplied by 60 to get RPS, and then multiplied by the radius of the wheel to get linear speed. The units were inches per second.

Trial Numbe	Time Taken
1	9.263338608
2	9.604221002
3	9.149738806
4	9.664270256
5	9.388626765
6	8.911646819
7	9.191728515
8	9.553962687
9	9.273623844
10	9.109791875
11	7.98336



Since the theoretical time (shown in red) is vastly different than our current time, we in fact graphed the velocity of the motor by printing out the velocity to a file and then using Excel to generate a graph. It appears that the reason the theoretical number is vastly different is because of the fact that we did not calculate for acceleration as VEX does not publish clear data regarding this. Here was our graph average for 5 attempts.

"Browning Test"

This is to measure whether the chassis could go with strain under long periods of time. To test this, we made a program that spun the chassis at full power. Moreover, we added weight by adding ~10 25 hole C-Channels. If the robot lasted for five minutes, we knew that the robot would be fine within a real match since they are sub 2 minutes. Overall, the robot did last more than five minutes and had no noticeable changes in speed/performance.

Intake Build Process (Archis, Conner: 9-23, 9-28, 9-30)

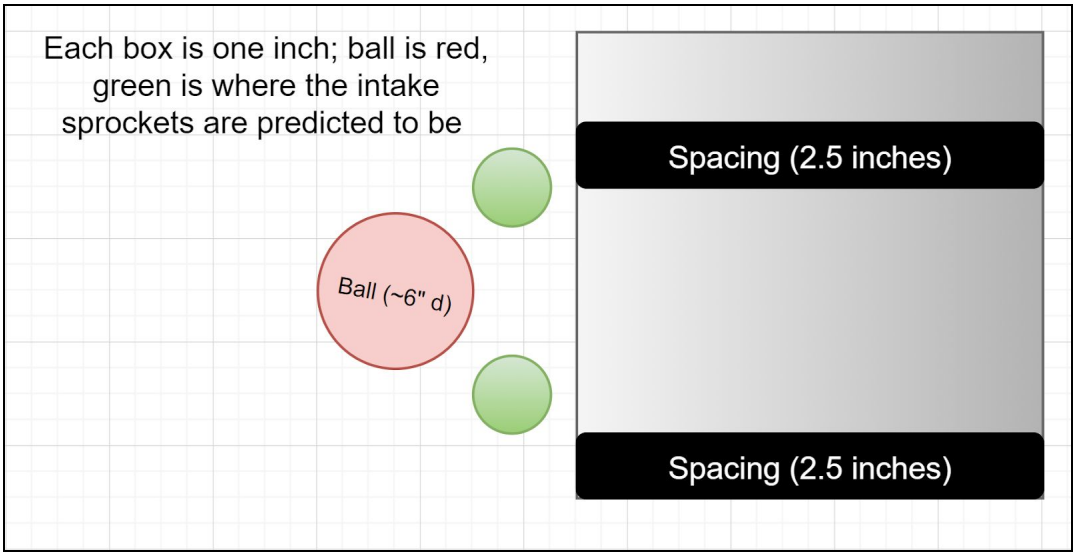
Objectives

- Complete a functional build of the intake as the rest of the build depends on the intake
    - First, we will be using CAD to make a prototype model of the intakes
    - Build the sides separately (left and right).
    - Connect the intakes to the chassis

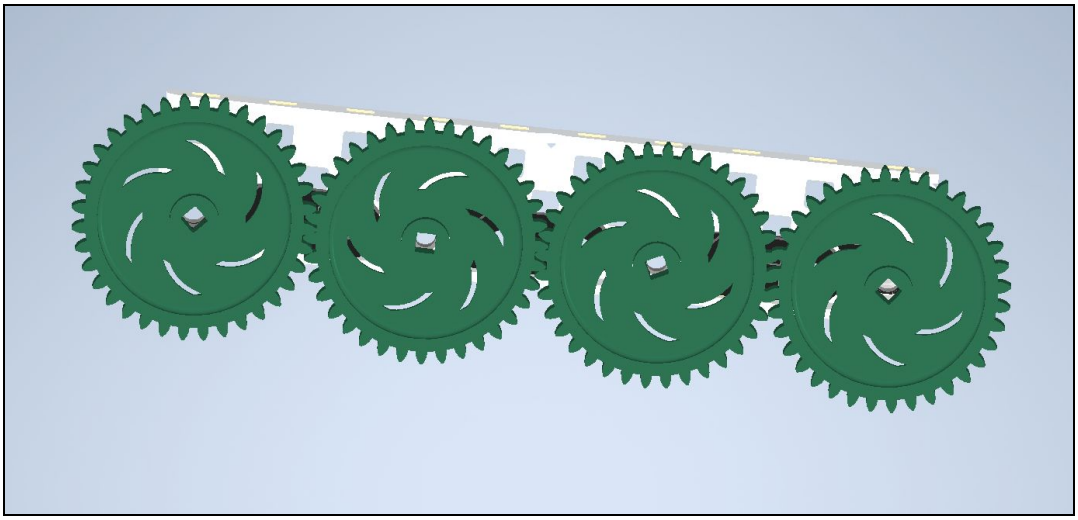
Dimensions of the Intakes

Overall, we wanted to figure out the lengths of the intakes using the dimensions of the chassis. Since it was planned that we generally wanted to avoid any sort of flip-outs within our robot, we will be maximizing the space that we have left for the intakes. As of now, the chassis is 12.5 inches, and since the max limit is 18 inches, we want the intakes to be a little below 5.5 inches. With this, hope to make the intakes approximately five inches in length.

Now in order to figure out the pieces of metal used with the intake build, we will be first beginning with the size of the sprocket that we will be using. According to the diagram used for the chassis, the sprockets in order to fit the ball correctly, we will be using the second largest sprocket size. Another reason for this is that we will be attaching flaps onto the sprockets to ensure that we are able to grip the balls better. The diameter of the sprocket is ~3.16 inches according to the VEX website. The diagram that we are referring to within this log is below:



Now based on this, the hole of the sprocket is located in the center of the sprocket. However, the most important criteria of the intake is that in order to stick through with the design within the decision matrix, the bar length must be so that it permits 36t gears to be back to back to back. Here is a visual:

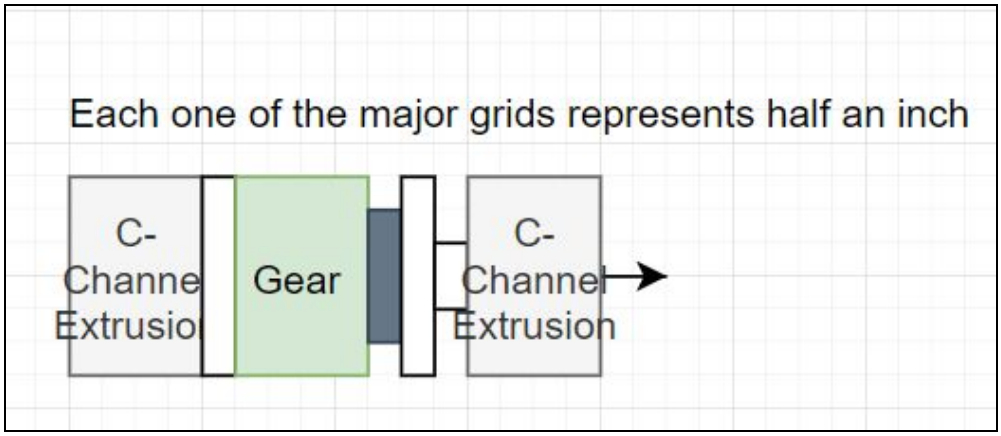


Based on the CAD model seen above, since we want to maximize the space that we have and this is the maximum number of 36t gears that we can put back to back before crossing the

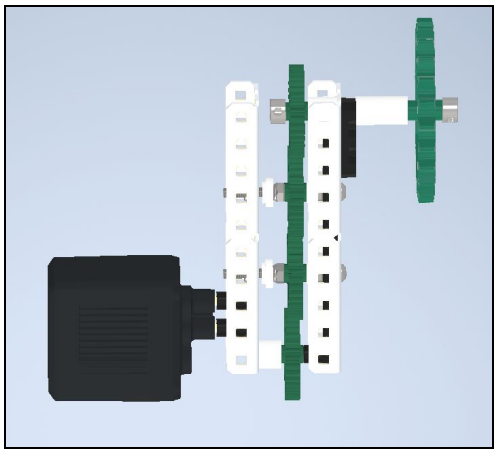
size limit, we will be needing to use the bar length within the image above. To be more specific the bar length for the intakes will be 10 holes.

CAD Process

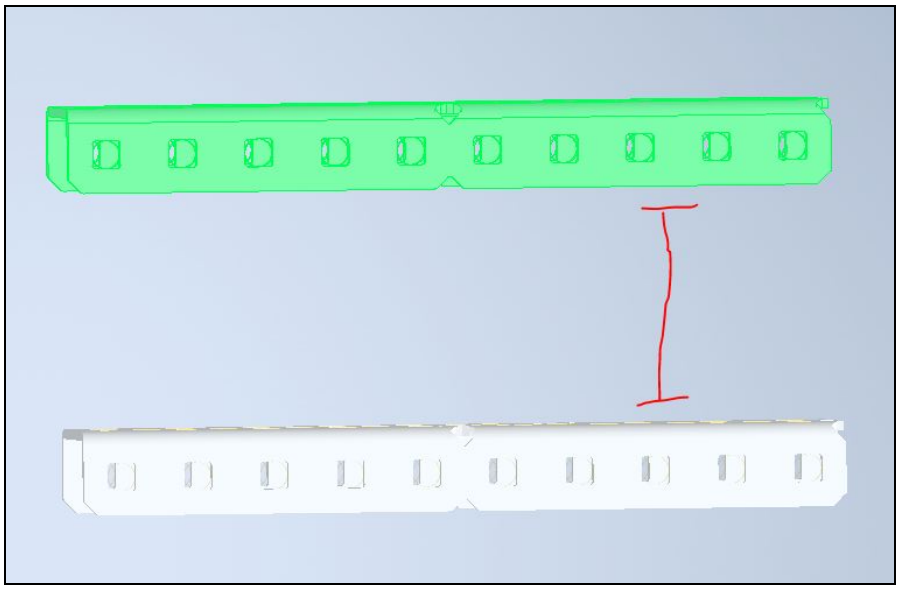
Based on this knowledge, we should be able to get a working CAD model. What we did was first put a C-Channel with the appropriate hole count. Then, our objective at this point was determining the least thin possible spacing. In order to figure out the spacing, CAD was used as a usual visual tool that gave us a representation. Moreover, we knew the diameter of the high strength 36t, the bearing flat, and the extrusion within the C-Channel. With this, we were able to form a preliminary spacing design for the intake. Here is what it is.



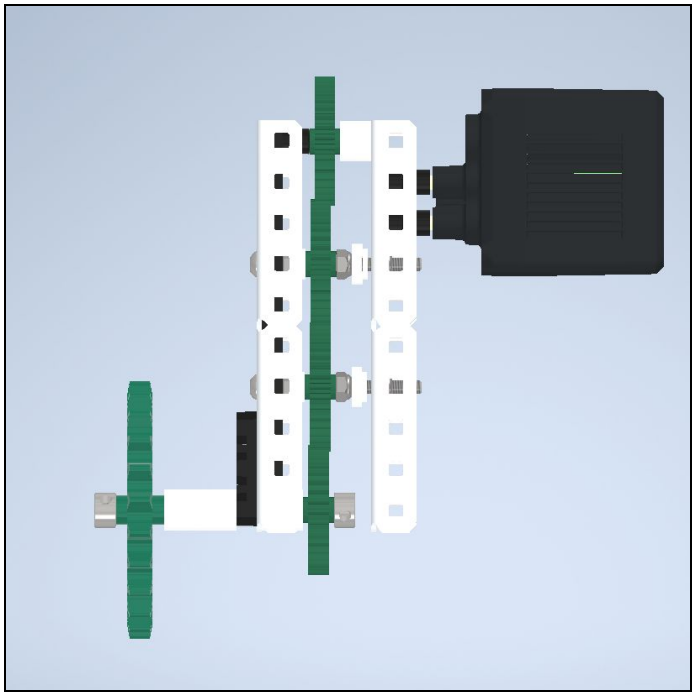
This is the thinnest possible spacing while allowing enough tolerance for the gears. Another thing to consider was the use of screw joints within the intake. Since at the ends there will need to be an axle since a sprocket and motor exist at the ends, we will be using the middle gears on a screw joint. The advantages of using the screw joint is increased stability and strength compared to regular shafts. Moreover, another



thing we had to keep in mind when modeling through CAD was that there needed to be bearing flats. With this, we decided that the side with the C-Channel extrusion will have the bearing flat and another 1/4 spacer in order to fill out that extrusion. The other side's face will be contacting the nylock. Here is a representation of the C-Channels (the green C-Channel is the one will be hosting the bearing flat; red is where the spacing we described above will go):

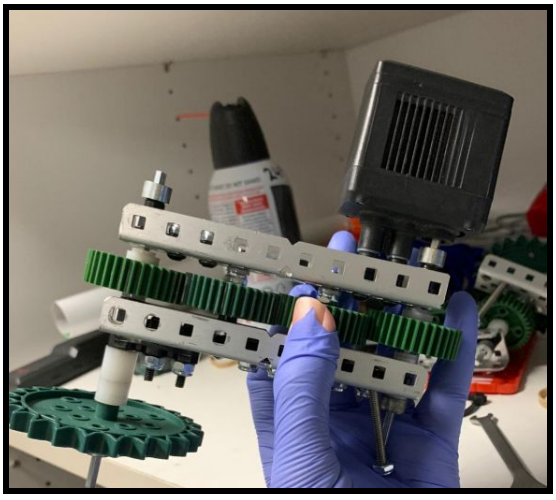


Based on this, we were able to form a functional CAD model of the intake. Unfortunately, the CAD needs to be using low strength gears since insert constraints are unable to operate correctly on the high strength gears. Then at the very end of the screw joint, there is a nylock.



Build Process

Since we had a CAD that was completely ready for the team to follow, all that happened for the build process was that we finished the physical model by following the CAD verbatim. Here was the result:



Custom Uptake Sprocket Build Log (Conner, Aayush: 10-5, 10-7, 10-12, 10-14)

Objectives

- Create and complete a sprocket that uptakes balls with speed and feeds them into the flywheel.
    - CAD the sprocket
    - 3D print the sprocket
    - Test the sprocket
    - Modify the sprocket
    - Final product

Sprocket Design

The sprocket was designed with enabling two points of contact on the ball, while the ball was in between the intakes and the tower, and creating a faster uptake. The two contact points were a must for this design because if balls were to only get one contact point, the ball could get stuck in a dead zone where it could not be properly intaked. This posed an issue because if the ball got stuck in a dead zone the intake speed would slow down and the ball could get irreparably stuck, ending the skills run early. The faster uptake was a must for this design because with the time constraint of skills, the functions of the robot needed to be quicker. This made the uptake a point of scrutiny because it would be simple to make quicker, especially with the laws of energy and angular speed.

This brought us to the decision of a custom uptake sprocket because the increased radius would increase linear speed, increasing the kinetic energy of the system, and enable the two contact points easier. The increased kinetic energy of the system made the ball go faster because

there was more energy that could be transferred, speeding the ball up. The increased radius made achieving the two contact points easier because the uptake would poke out of the robot more, making the space between the intakes and uptake smaller.

The sprocket is backed by a 30 tooth VEX high strength sprocket in order to keep the polycarbonate sprocket from bending because VEX Robotics Competition limits the thickness of polycarbonate that can be used on robots. Since the limit is 0.0625 inches, it is too thin to stand on its own, meaning it needs the support. The 30 tooth sprocket was used because it would provide the most area to support the sprocket.

Sprocket Dimensions

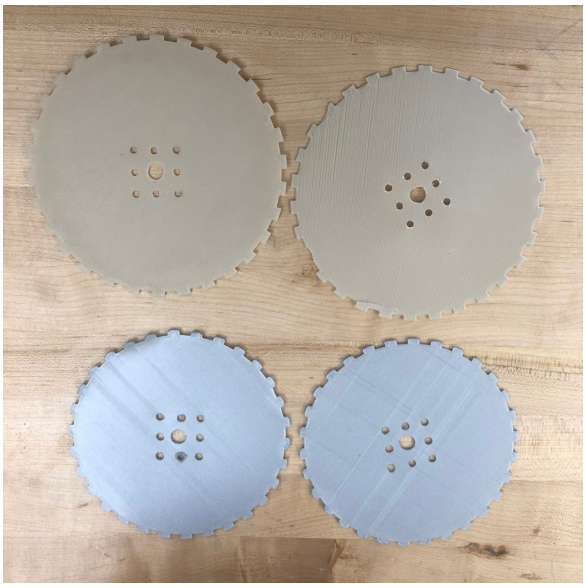
The dimensions of the first sprocket are as follows

- Outer diameter: 6 inches
- Inner diameter: 5.75 inches
- Holes: nine 0.17 inch holes equidistant in a 1 inch box
- Thickness: 0.06 inches
- Teeth: 30 equidistant rectangular teeth

The dimensions of the second sprocket are as follows

- Outer diameter: 5.5 inches
- Inner diameter: 5.25 inches
- Holes: eight 0.17 inch holes with one 0.38 holes equidistant in a 1 inch box
- Thickness: 0.06 inches
- Teeth: 30 equidistant rectangular teeth

30 teeth were used on the sprocket because then the teeth on the sprocket backing the custom sprocket would line up with the teeth on the sprocket. The thickness is 0.06 because of the VEX Robotics Competitions rules on polycarbonate, which limit the thickness to 0.0625 inches. The inner and outer diameters on both of the sprockets are 0.25 inches in difference because the thickness of a legal rubberband is roughly 0.125 inches, meaning that when the rubber bands were put on the uptake, the sprocket would have smooth edges, preventing it from catching on other things.

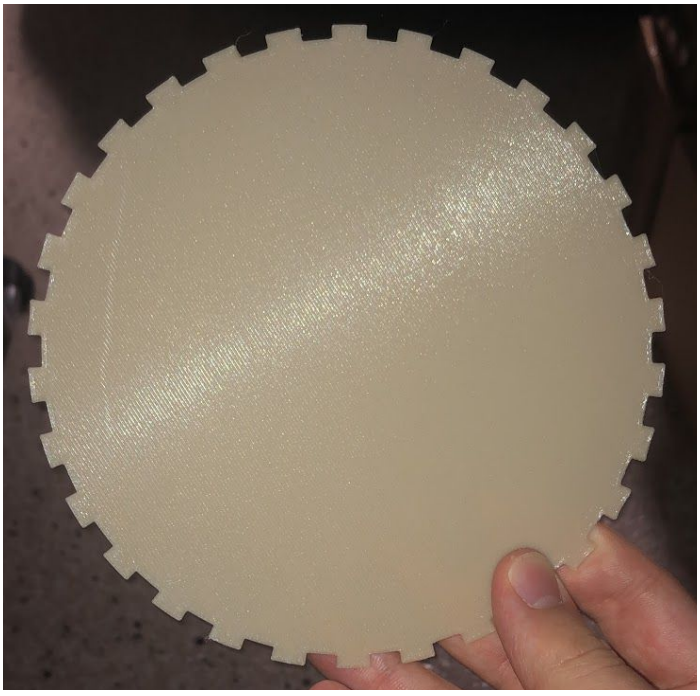


CAD

The CAD was accomplished using Autodesk Inventor. A circle of the desired outer diameter was drawn, then a circle of the desired inner diameter was drawn with the same center point. Next, one line was drawn at the start or end of one of the 30 teeth between the two created circles. This line was then used for the pattern tool and any of the circles used for the axis. Now 60 equidistant lines would be created and every other section made by the lines and circles would need to be cut in order to form 30 teeth. Then, a one inch box would be drawn around the center point of the circles. Then a circle of 0.17 diameter would be drawn on one corner of the box and the pattern tool would be used to get the 9 holes. For the second sprocket, the center hole was deleted and a new hole of diameter 0.38 was drawn. Finally, the desired section of the sprocket would be extruded 0.0625 inches using the extrude tool.

3D Printing the Sprocket

Using the CAD mentioned above, an STL file was created. One issue we ran into when creating the STL file is that Autodesk Inventor exports STL files in centimeters, unless changed in settings to millimeters, and Cura accepts STL files in millimeters. If this happens, either manually use Cura to make the sprocket 10 times bigger or change your inverter setting to export in millimeters. Using the STL file exported in the correct units, import the file into Cura, a slicing tool, and select the right 3D printer to use, then slice the file into a G Code file and export it to your 3D printer. Finally, using the G Code file, print the sprocket on your 3D printer.



Sprocket Testing

Using the 3D printed sprocket, we tested the custom uptake sprocket and found some issues. These issues were the sprocket being too big to take the balls, not getting close enough to the goal to effectively descored balls, and the center hole being too small to fit over the sprocket. In order to fix these problems that arose during testing, we decided to make a new version of the sprocket.

Modifying the Sprocket

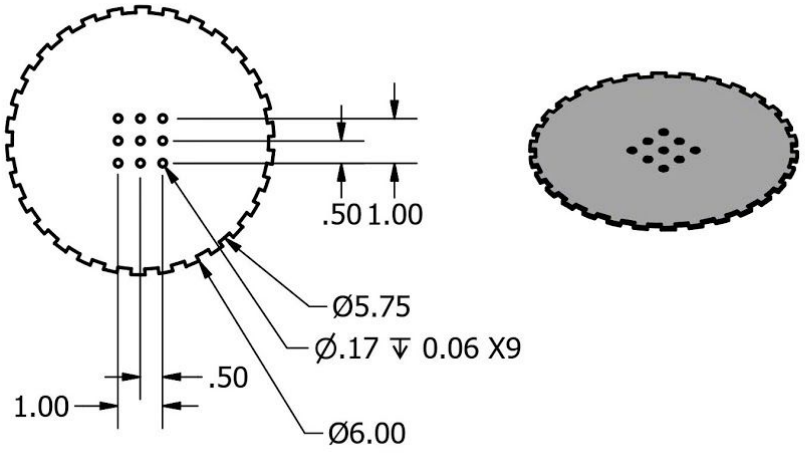
In order to fix the sprocket being too big and not getting close enough to the goal, we shrunk the sprocket down by 0.5 inches and changed how the sprockets were placed to support

the custom sprocket. Originally, the support sprockets were closer to the towers, but now were closer to each other because then the custom sprocket contacts the ball and goal at a point of the ball and goal where the chords are smaller. The problem of the sprocket not fitting properly over the support sprocket was fixed by changing the center hole to the correct diameter of 0.38 inches.

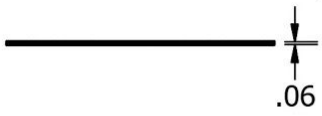
Final Manufacturing

The final manufacturing was done on a CnC Router with a six inch by six inch square 0.0625 inch thick piece of polycarbonate for the stock material. The CAD file was prepared for the router using the Autodesk HSM plug in for Autodesk Inventor. The CAD file was opened up in Inventor and using the CAM tab, we went through the tool and stock material set up for the router. Then, we used the 2D pocket and 2D contour tools to define which parts the router needed to cut, primarily the outer edge and the holes. Next, we exported the file to the router and set the router up to cut the sprocket. Finally, the router cut out the sprocket and we mounted it on the robot. Upon testing the new sprocket it worked as intended and the material change from PLA to polycarbonate had a negligible effect.

Sprocket Version 1 Design Log



The sprocket is designed to be a rubberband holder supported by a normal 30 tooth sprocket. To accomplish this there are 30 teeth on this sprocket and the teeth are at the depth of the thickness of a rubberband. The diameter of the sprocket is 6 inches in order to leave as little space between the flywheel roller and the uptake. The reason 2 rollers could not be used for this task is because then the rollers would have to be geared or chained together, which would be heavier and have more moving parts. The holes are spaced in order to line up with the holes on a 30 tooth sprocket.



Custom Uptake Sprocket Ver. 1
SCALE 1 / 4

Tower Build and CAD Process (Archis, Saachi, 10-14, 10-19, 10-21)

Objectives

- Create and complete a tower capable of uptaking and storing game objects effectively in order to maximize functionality of the robot
    - Plan and CAD the tower
    - Build the towers
    - Brace the towers to maximize strength and stability
    - Test the towers

Tower Design

Our primary topics of consideration when designing the tower were its height/width, its placement on the chassis, uptake and shooter placement, motor placement, and finally bracing. Stability and efficacy were the key factors in designing the tower. However we focused on bracing to add stability after actually mounting the tower,

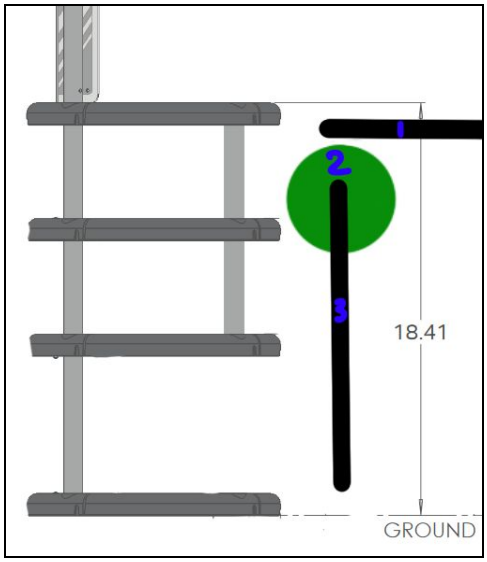
Tower Height and Width

When determining tower height, we needed to consider 2 factors: the height of the goal and the 18 inch height limit.

1.Hood - top: about 17.5"

2.Sprocket - top about 16.5"

3.Tower bar - top about 15.5"

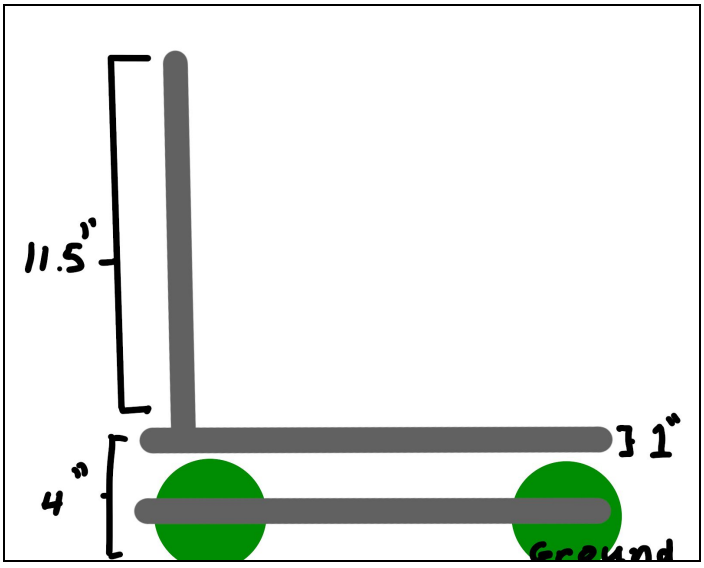


As seen in the image on the left, the top height of the goal is 18.41 inches. As such, in order to effectively reach the goal to score, we needed to make a tower which was tall enough to bring the top sprocket of our outtake as close to the 18 inch limit as possible.

The closer we were to the top of the goal, the easier it made it for our outtake to score. However, we needed to consider room for the hood and as such, we had to undershoot by around an inch (the sprocket had to reach around 17 inches). In terms of the tower bar itself, we did not want it to fully cover the entire sprocket as we wanted the ball to hit the hood straight from the sprocket without interference from the tower bar.

With this in consideration we started determining the bar length. Upon creating the chassis, we had already placed a frame upon which we could mount the tower. Additionally we established that the tower bar needed to be sturdily mounted. In mounting to the frame, we would be able to attach each tower bar to the four interior holes of the top frame c-channel.

The frame was around 4" off the ground. In return, we needed a bar which extended



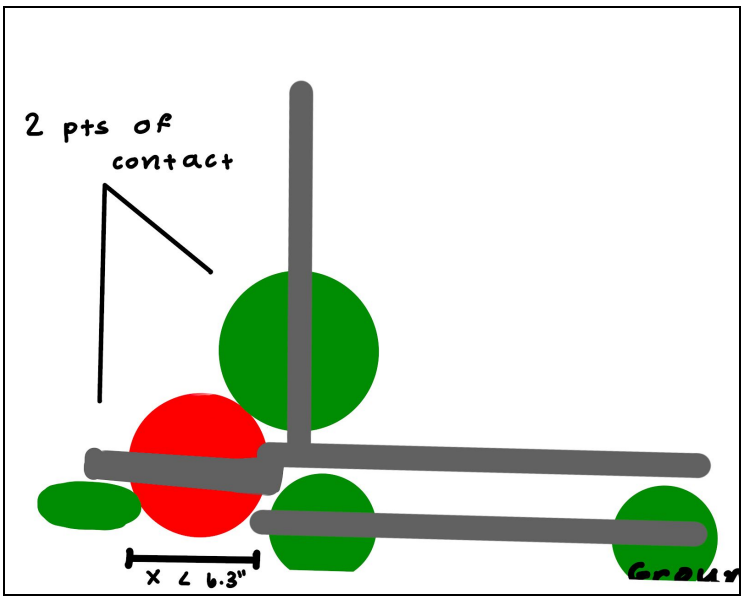
11.5 inches above the ground. Taking into account the c-channel width of 1" we needed to use a 12.5" long bar in order to attach to the four interior holes (1" wide overall). As such, we used a 25 hole bar for the tower.

As for the width, we had to ensure enough width for the uptake and outtake. However we merely had to mirror the inner bars of the chassis as they were the ideal distance to incorporate the ball and were wide enough to incorporate the width of the indexer.

Tower Placement

Next we looked at tower placement. When considering tower placement we had to consider the diameter of the ball and cohesion with the intake. The diameter of the ball is 6.3 inches. In return, there needed to be at least 6" between the back of the uptake sprocket and the back of the chassis to hold a ball. Consequently, an 8.75" distance between the tower and the back of the chassis was the bare minimum required (considering our 2.75 inch custom sprocket radius).

Additionally, we had to consider cohesion between the intake and uptake. In order for the intake to smoothly lead into the uptake, the ball needs to immediately contact the uptake sprocket upon dragging in the ball. The ball immediately needs two points of contact. To do this the spacing between the intake and uptake has to be less than 6.3." Ideally, the closer the distance the better. A visual of that is seen below:

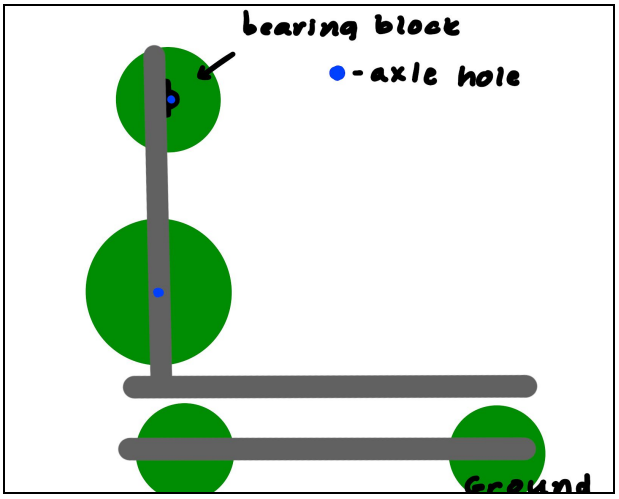


As such we decided to place the tower on 6th and 7th holes from the front of the chassis. This provided us with 9 inches of distance between the tower and back of the chassis. We overshot by an inch to free up room and provide flexibility while building. Furthermore, as our intake sticks out around 5.5 inches, the distance between the intake and outtake is only about 3.5 inches. This makes it so the two points of contact are immediately established upon intaking the ball.

Determining Uptake + Shooter Placement

The shooter was meant to be placed at the top of the tower, protruding above the bar. As for the uptake, we had planned it to be significantly below the uptake considering its sizable 5.5" diameter. Additionally as its diameter was wider it needed to be placed on a slightly more forward hole in comparison to the outtake.

Considering this we determined the placement of both the uptake and shooter as seen below:



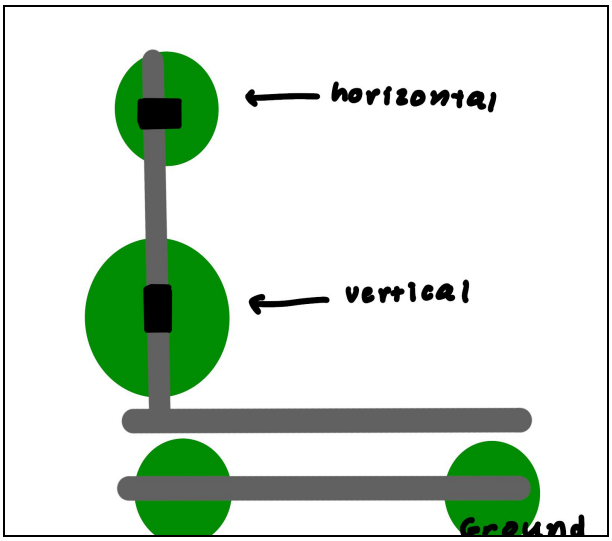
To offset the shooter, we decided to add a bearing block to the tower. This would shift the axle hole inward.

The uptake has a 5.5" diameter/ a 2.75" radius. Considering the 6.3" diameter of the ball, the bottom of the uptake sprocket had to be at least 6.3" above the ground.Therefore, We placed it on the 12th middle hole of the tower c-channel. In doing so, this would place the sprocket 9.25" (4" from the ground to top of the chassis +5.25"

protrusion from top of chassis) from the ground, making the bottom of the sprocket 6.5 inches from the ground.

For the shooter we had to place it in the topmost location where it would not cause the robot to exceed the 18" height limit. With this into consideration, we centered the axle hole on the second to last hole of the c-channel.

Motor Placement



Finally, we determined motor placement. As the power provided by one motor was sufficient, we only needed to place motors on one side of the tower and as such were only concerned with motor placements on that singular side. Considering the location of the uptake and outtake we decided to place the motors as seen to the left.

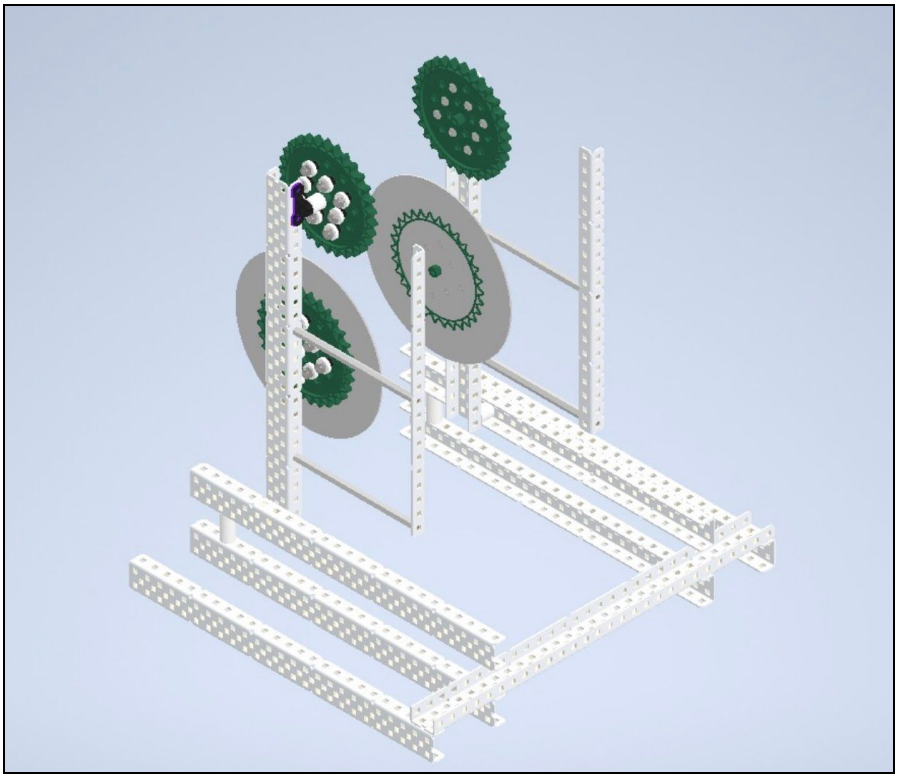
CAD

After finalizing these factors, we started to CAD the robot on Inventor.Objects such as the ball, chassis and sprockets were already made, making modeling the tower rather simple. We merely had to import the specified c-channels, motors, bearing blocks, and screw, and attach them to the robot as per the determined design.

However, in our CAD, we added an additional component. When designing, we were planning on creating a plexiglass ramp. As such, to hold the ramp, we needed to create an extension past the tower. These extensions would essentially be a support to which we could screw on the ramp.

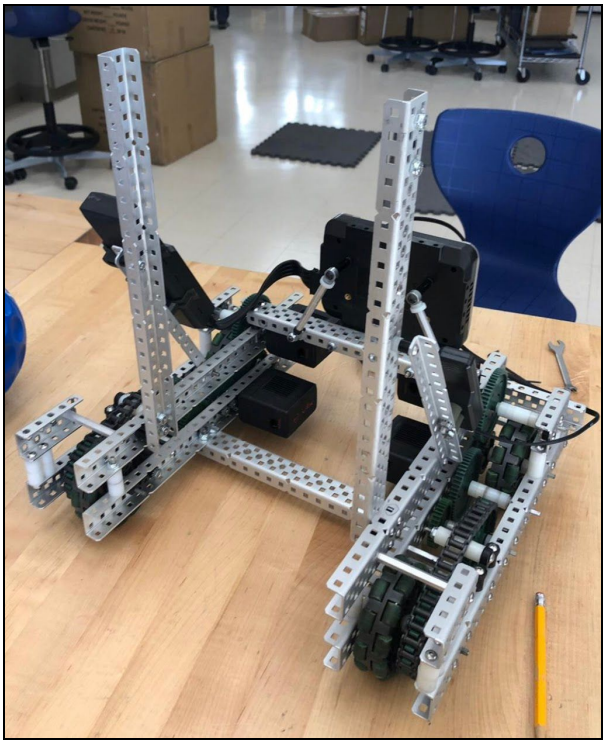
For the extension, initially we wished to use c-channels. However, we decided on 1*1 L channels. These are lighter than c-channels while providing the same function as them. Now to connect the 1*1 Ls, we sought to use c-channels as the most stable connectors. However, when using c-channels, it was impossible to mesh with the tower. In return, we decided on using standoffs.

Applying all of this, we developed an initial CAD of the tower:



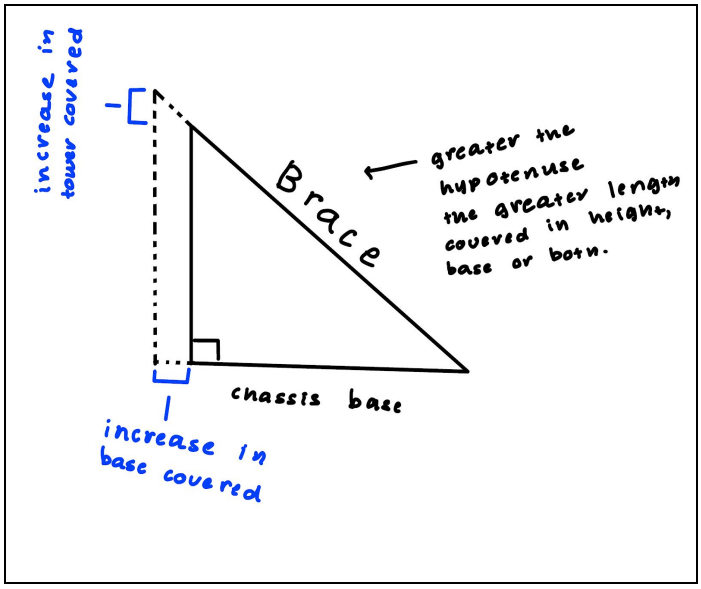
Building and Stability Insurance

The CAD we developed essentially acted as a step by step guide while building. Following the CAD, we mounted the towers and motors onto the chassis. With a temporary brace:

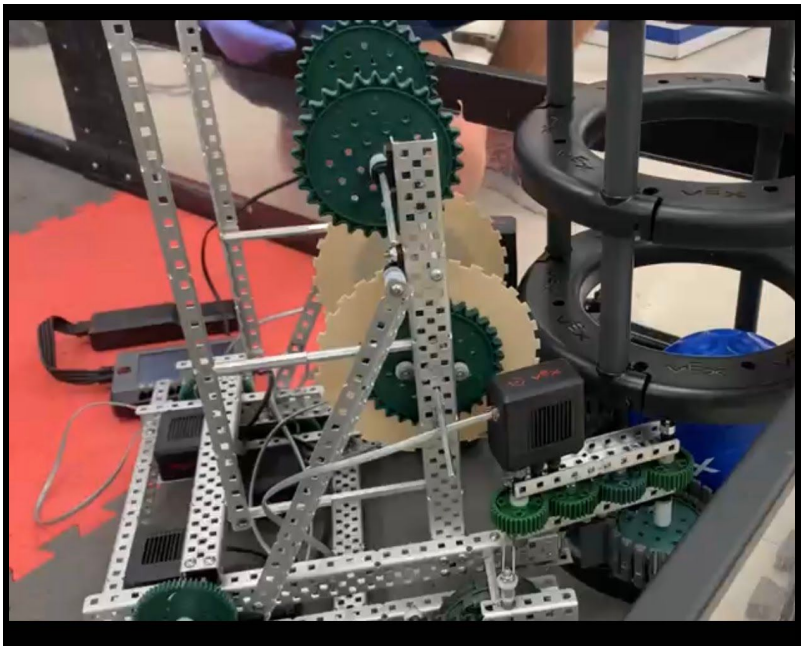


Our ideal bracing method was triangle bracing. Triangle bracing creates a three point brace for the towers. With three points of contact, it ensures that towers will not shift forward and backwards. At first we merely used a 15 hole brace however this soon proved insufficient. The shorter length of the brace failed to completely prevent tower shifting.

We soon realized that we needed to ensure that the brace covered the maximum distance possible. The more distance the bar covered, the more support the tower will have both vertically and horizontally. A visual representation of that is seen below:

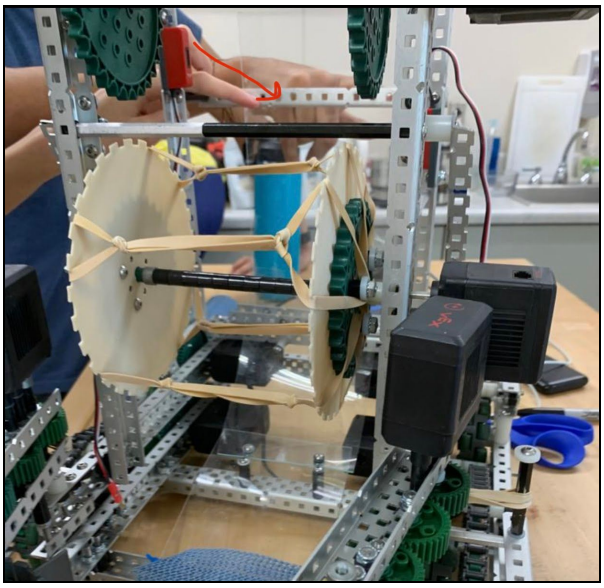


As such, we replaced the 15 hole brace with a 20 hole one enabling us to create an 8x15x17 right triangle. Previously we had a 5x12x13 triangle brace however the newer one provided us with 3 more holes covered horizontally and 3 more holes covered vertically. We attached the bracing onto a bearing block shifting outward to maintain the structure of the right triangle. Additionally we included a ¼" spacer between the block and brace to prevent an inward tilting of the brace.



With this, forward/backward shifting completely stopped.

Now, while testing, a new issue arose. While the towers were not shifting, they were caving inward. This issue was caused by a lack of support between the towers. The axle connection between the towers created stress inward yet there was nothing counteracting it/stopping it.



As such (as seen on the left), we implemented standoffs connecting the two towers, This provided interior stability while preventing inward/outward tilting of the towers.

Polycarb Mounting (Tyler, Conner, 10-26, 10-28)

Objectives:

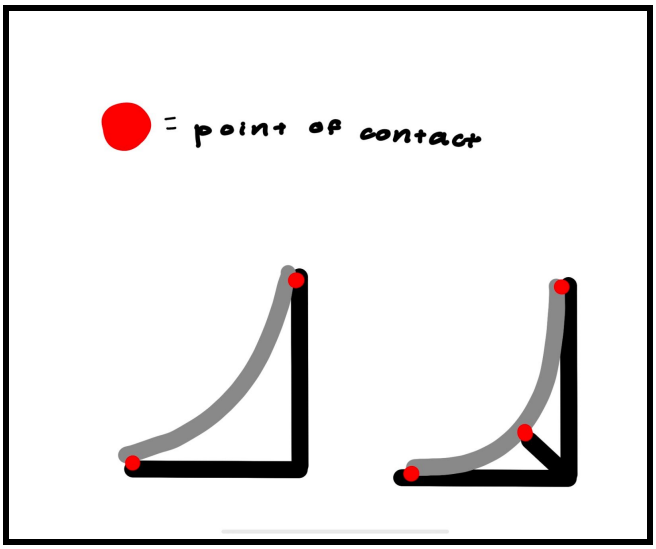
- create a polycarbonate ramp capable of facilitating the movement of the ball through up the tower
    - Plan the placement of the polycarbonate
    - Cut the polycarb
    - Install the polycarb
    - Test the polycarb ramp

Why Polycarb?

When designing our uptake, we needed a ramp in the back of the robot to facilitate the movement of the ball. A ramp would relieve the force required to move the ball upward and as such, would make a more effective and efficient ball-cycling system. A key aspect of such a ramp is a curved body. A smoother, curved ramp makes it so that there is less friction and jitter when the ball moves. With metal parts, it is harder to make such a ramp. Therefore, we used polycarbonate. Polycarbonate is flexible; this allows us to manipulate it to our desire.

Mount design

When designing the polycarb, we initially planned to achieve the curve by using a heat gun and bending it physically. However, we soon realized that the curve could be achieved mechanically. To bend the polycarb in such a way, we needed to attach it to multiple points.

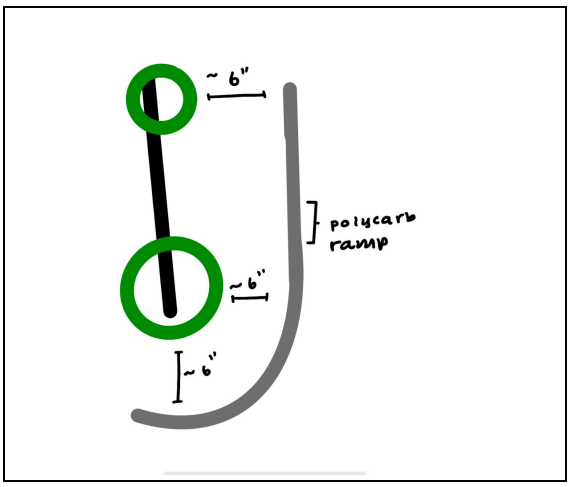


Here we see that the curve can be achieved with only 2 points. However, each additional point grants us greater control over the curve of the polycarbonate. As such, we wanted at least 3 points of contact between the ramp and the mount.

Thus, we essentially needed to create at least 3 polycarb mount points.

Specifics

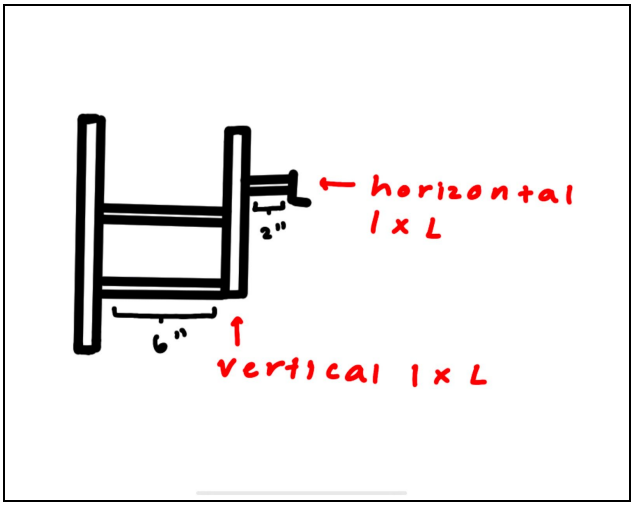
The primary criteria we had in designing the mount and choosing its placement was that it had to be a far enough distance from the uptake so that the ball could fit at all points of the system. A visual of this is seen below:



The ball has a diameter of 6.3" therefore the distance between the ramp and the back of the indexer and outtake must no more than 6.3" We undershoot to create compression and to increase contact between the systems and the ball.

To achieve this, for the back, we needed to create two structures around 8" away from the tower considering the 2.75" diameter of the radius. These would act as the first two points of attachment. (We went for two attachments in the back as we wished for the back polycarb to be completely straight).

When creating such a structure, we considered the weight of the structure as we did not want to add any unnecessary mass to our robot. As such, instead of merely placing two channels this distance away, we created a thin protrusion using standoffs and 1x1 L channels.



To the left is a draft of the protrusion. The tower is connected to a 1x1 L channel 6 inches away. The attachment is created with a 4" and 2" standoff. The 1x1 L channel is further connected to a horizontal L channel by a 2" standoff. The horizontal L - channel acts as the back brace for the polycarbonate ramp

For the bottom connections, we decided to add another horizontal 1 x 1 L directly on top of the chassis. On this L channel, 2" standoffs were optimal for producing a curve that established a 5.5" space away from the back of the indexer sprocket. Then, we decided upon fastening the front of the ramp with standoffs reaching the bottom of the central bar of the chassis.

Dimensions of The Polycarb

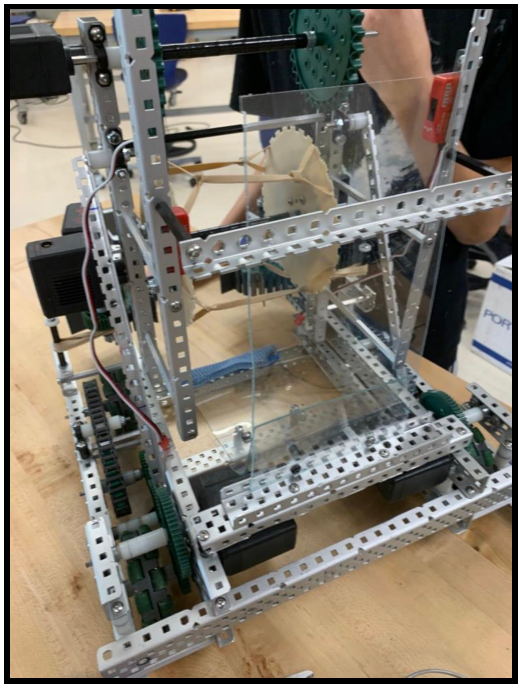
Prior to building the ramp, we needed to establish the dimensions of the polycarb/ The polycarb needed to cover from the top of the chassis all the way to the top of our robot. The tower itself reaches a height of 17.6 inches, however, mounted on the chassis, it only takes up around 12.5 inches. As for the chassis, from the front of the chassis all the way to the predicted location of the back mount, takes up around 10" THerefore, the polycarb needed to be 20-22" in length to

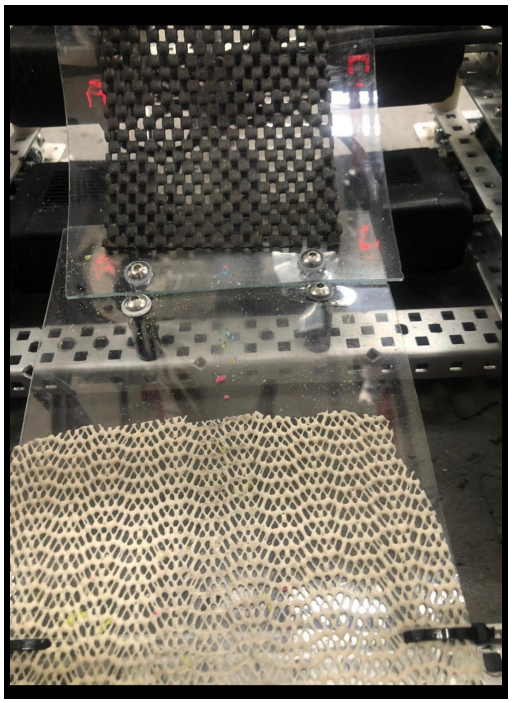
account for the entirety of the passage. The width of the polycarb had to match the width of the towers. The towers were around 10 inches apart; as such, we wanted to undershoot to a length of about 8." With this in mind, we began building.

The Build Process

We first cut out the polycarb and drilled holes in accordance with where the polycarb would be mounted. To do so, we first laid in the desired location and pinpointed the locations of the specific holes on the piece. Then, we drilled the holes.

Our first polycarb drilling process was unsuccessful. The holes did not align properly, therefore, we needed to redo them. However, we soon ran into the issue of the lack of long enough polycarb. We only had three pieces of polycarb left, each only around 7" in length. Therefore, it was impossible to proceed with our initial plan of using an entire piece. Instead, we decided to use the three pieces to artificially create the same type of ramp as a single piece would provide. We used one 7" polycarb piece for the back and one for the front/bottom, and one for the curve as seen below:



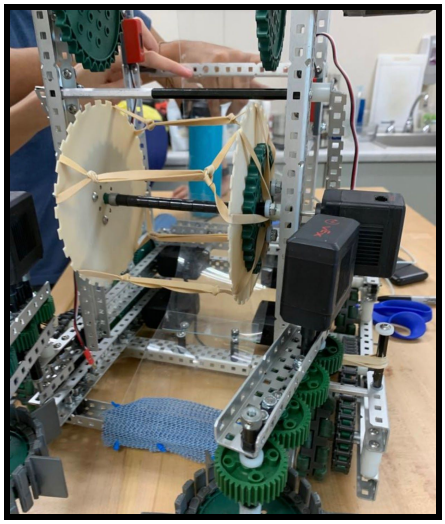


The first piece (see image 2): We mounted the first piece using 2″ standoffs at a singular connection point. These standoffs were mounted on top of our central chassis bar spaced 4 holes apart. To relieve stress on the material, we included washers slightly spacing the screw from the piece.

The second piece (see images 1 and 2): The second piece/ the curve piece was fastened to two points. The first attachment was on the same chassis bar as the first piece. Here, we similarly used 2″ standoffs spaced with washers around 4 holes apart. For the second attachment, we fastened a 18 hole wide c-channel on top of the chassis. We fastened the top of the piece on this new bar on the 7th and 11th holes. Here we used slightly 1/16″ washers to fill the gap between the standoff and polycarbonate, and to relieve

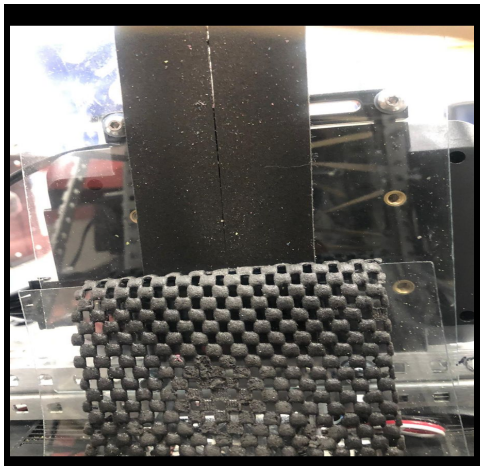
To mount the third piece, we first started with the structure mentioned earlier. We connected 25 hole L channels to the tower using a 2 inch and 4-inch standoff. We did this at two points, specifically holes 1 and 8. Next, we added the secondary extension with the 2 inch standoffs. To the standoffs, we horizontally attached a 15 hole L channel. Then, we attached the plexi glass to this L-channel on the 4th and 11th holes, using 1/32" washers in between. For the second connection, we used the 18 hole wide bar we had added and added another 15 holes bar on top. Then, we attached the bottom of the polycarb using screws and 1/32" washers at the 6th and 11th holes. The overall initial build:

We added mesh to the front of the ramp by zip tying it to the bottom c-channel. This added traction to the front of the ramp preventing the ball from initially slipping down the ramp. We later replaced this mesh with a longer piece of mesh.



Testing: Issues and Solutions

Upon testing the newly created ramp, our primary issues were a lack of compression between the ball and the indexer and a lack of traction on the curve of the ramp. A look at our solutions:



Explanation:

A lack of compression is generally caused by the distance between the indexer and the ball being too wide. We experienced the lack of compression towards the back of the ramp. To solve this, we simply added a piece of foam. The foam strip has adhesive on its back so we merely had to stick it on. As for the lack of traction, we decided to fasten mesh across the curve of the ramp. We already had enough traction in the front of the ramp due to the razor mesh and enough traction in the back with the newly added foam strip. Applying similar logic to the curve of the ramp, we zip tied a flap of black mesh. However, we did not completely fasten the frontal part of the mesh as it granted us greater flexibility and adjustability for the mesh.

A look at our finalized ramp build in action:

The ramp acts to guide the ball while providing a back brace to keep the call in contact with the indexer and shooter.

Uptake and Shooter Build Process (Tyler, Ashwin, 11-2, 11-4, 11- 9, 11-11)

Introduction and Objectives

With the competition our robot wanted to mainly focus on speed and efficiency over anything else. Because of this our robot instead of being complex prioritized speed while holding a compact form. We wanted to focus on getting a simple indexer and shooter like sprocket system in our robot. With the recently added rework of VEX Change Up we also wanted quick shots into goals due to the lack of interference from other robots. As a result avoiding multi-step processes and non-optimal movements like a lift or chain bar to serve as an outtake for our robot would be vital.

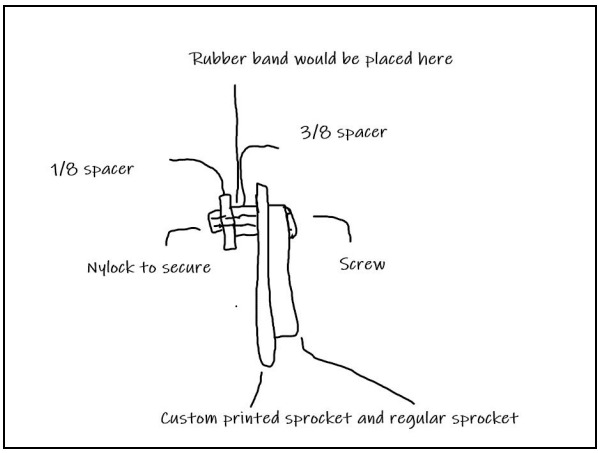
As such, our objectives:

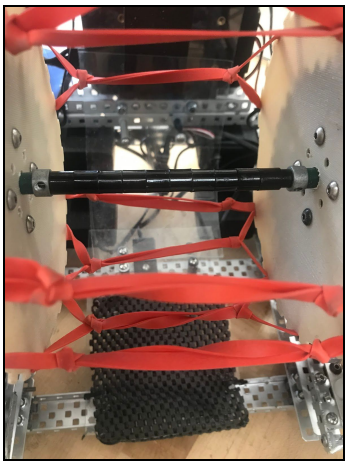
- Compact and simple design for indexing and outtaking
- Ensure that we are able to maintain consistency
- Also prioritize speed and cycle shots easily so as to not spend too much time at a goal.

Building the Indexer and Outtake

For our indexer we attached it to the main tower of our chassis. We used a custom printed sprocket made of polycarbonate with regular sprockets on it and used banding for tighter compression to help push up the ball up the robot. The regular sprockets were 30-toothed and had screw-spacer-nylock builds in order to securely mount the rubber bands as an “anchor” to latch onto.

One advantage of the custom printed sprockets is that we were able to drill holes in the poly carbonate in order to mount spacers and nylocks on the sides of it in order to mount on the rubber bands. Both indexer and outtake were mounted on our box-like chassis on a robot with one motor at 200 RPM on the side of it. On top of it we also put in a regular sprocket system for our outtake to provide the flow-like movement combined with our mounted polycarbonate and hood. We also added rubber bands interconnecting the sprockets together, which also served as the main grip for the ball to rub against. For both our indexer and outtake, an axle would go through the sprockets, 11 black ⅜ spacers and two inner shaft collars straight to the motor. The spacers spaced them out to the appropriate position and lock them in place, however we realized later that the spacers would not be completely necessary as the shaft collars would be able hold their own.



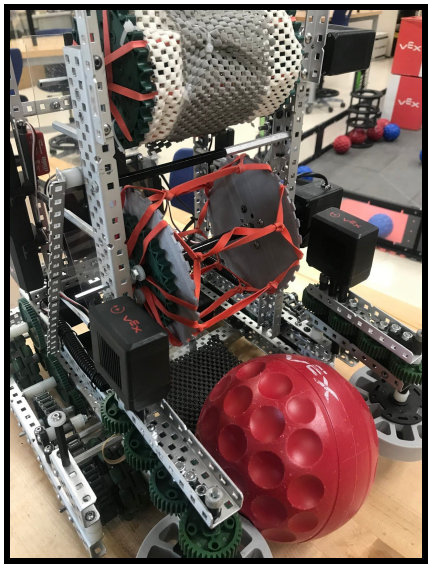


Rubberbanding and the Custom Sprocket

At first, we used competition-illegal sprockets to serve as a prototype, but later swapped with a polycarbonate after realizing that the prototype worked perfectly for us. The rubber banding was in a sideways Y shape on our indexer. However, this would later be changed as the speed of the index would be much slower than we thought. In order to cover more surface

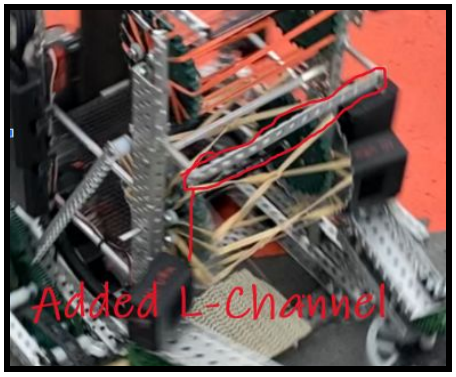
area, we switched to interlaced X-shaped rubber bands to reliably index the balls at a good speed.

Adding Mesh



After certain malfunctions with our hood system and outtaking not reaching the appropriate power required to consistently score a goal, we thought of adding a double-layered mesh to the outtake in order to create more friction and grip, only to realize that the hood was the major cause of the problem, and that the mesh would only slow the ball down more.

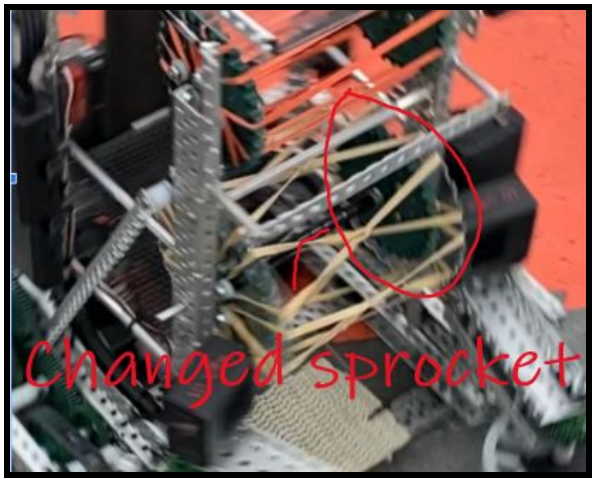
Also our indexer had contact with the goal it was acting upon, therefore causing potential problems transferring the ball to the outtake, so an L channel with standoffs extending it outward towards where the goal would normally make contact were added to prevent damages to the sprocket.



Problems with the Custom Sprocket and Changes

After our skills competition, we also foreseen a problem that the custom sprocket was grinding against the ball too much during the intake, giving it too much compression. This was fixed so that the green sprocket and custom sprocket were swapped in position as the green

sprocket was smaller and gave more room. We also felt that the speed of the motor was not fast enough and switched to 600 RPM. This was reverted back after the speed caused too many problems with the consistency of the shots and was not able to handle more than one ball at a time. Another issue that arose was the lack of power when the ball was out taking, so we added side mounts like our indexer and added twice the amount of rubber bands to provide more compression.



Mechanical Stop Build Process (Tyler, Andres, 11-16)

Objectives

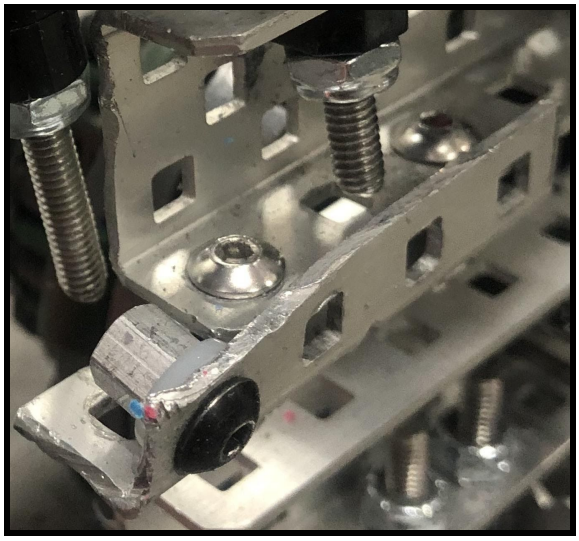
- Complete a functional build of the mechanical stop for the intake
    - We believe that optimization will come after getting a functional mechanical stop, so we definitely will have that for another meeting in the future

Overview

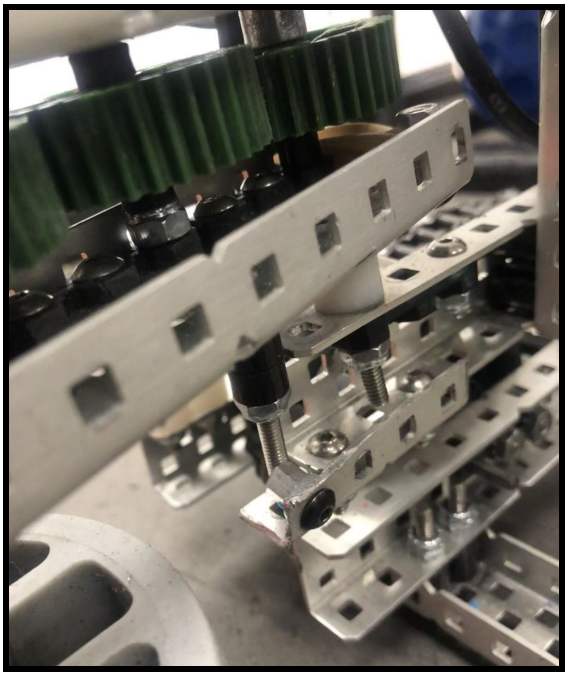
Since we are done building the intakes for the robot, we need to add a mechanical stop to them so that they do not end up going too inward when they are in resting position. A mechanical stop will also prevent the intakes from getting damaged, because of the rigorous use of them on the field. Moreover, we knew there was a certain point we did not want the intake to go past. To build our mechanical stop, the most efficient way was to extend a screw from the intakes and have it hit something so that it stops on its own. After examining our intakes, we noticed that there were already 2 screws on either side that were extended, but we just needed a mechanism for the screw to hit so that it stops.

Build Process

To start the build process of the mechanical intake, we added a 1XL to the bottom parts of the inner bars of the chassis. We realized that there was a certain point we didn't want the intakes to go past. By adding the 1XL's, as shown on the right, this would be



the point the intakes could not go past. After adding the 1XL's, we realised that we wanted the intakes to stay further out, so we added a 1/16th spacer and a 0.25 inch standoff, which increased the mechanical stop by 5/16ths of an inch. After doing this, we saw that the intakes would hit the standoff and not go past that point in any case. On the left, there is a picture of the completed mechanical stop hitting the standoff. No matter how far out the intakes go, the mechanical stop doesn't let it go past a certain point, making them always stay in the same position when they are at rest. This leads to a number of benefits for the robot in the long term, and when we are on the field.



Results

There are a number of benefits from the mechanical stop. By having it here, we are able to intake balls much more efficiently because the intake is always in the perfect position to pick up the ball. It doesn't have to readjust itself so that it's compatible with the ball, which makes our robot faster overall. In addition, by having the mechanical stop, it prevents the intake from bending inwards too much, which could cause it to break. By always stopping the intake from going in too much, it reduces the chances of it breaking, therefore, being better in the long run.

Mechanical Stop Testing Log (Ashwin, 11-18)

Observations of Intaking Being Far Apart

Upon finishing the new intake design, we mounted the new intakes where the old intakes were mounted. Then we placed the robot on the field in order to test the new intakes. During the test it was revealed that the intakes were too far apart. Some solutions to this problem were moving the intake mounts closer together, moving the intakes higher, and/or making the mechanical stop shorter. The first two ideas were taken out of the idea pool because they would have taken the more time, while being just as effective in achieving the desired effect.

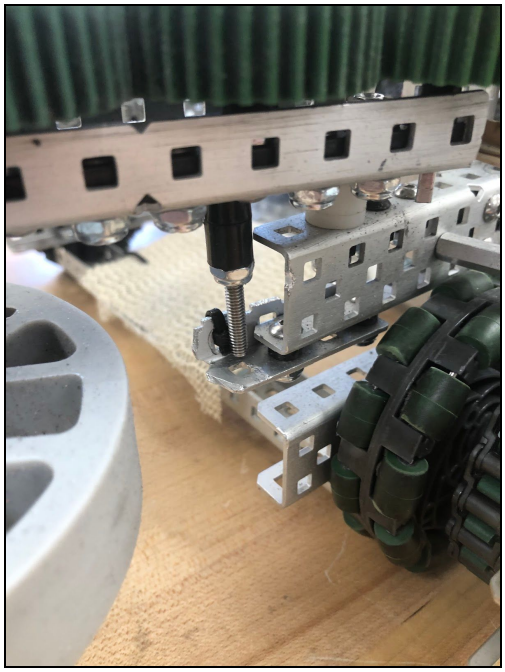
Trial and Error Range Testing

Once the approach was chosen the method to determine the optimal size was needed. This resulted in ideas like shortening the mechanical stop by increments and changing the size based on previous test results were thought up. The mechanical stop size determining method was decided to be shortening the mechanical stop by increments because we thought it would take the least amount of time. In other words, the process that we used was simple trial and error since it was the most effective solution within this situation.

We started with the old mechanical stop, then shortened the stop by one size each time. These sizes from longest to shortest were old mechanical stop, 0.5in standoff, 0.25in standoff, keps nut, and zip tie.

	Old Mechanical Stop	0.5in standoff	0.25in standoff	Keps nut	Zip tie
Success?	No	No	No	No	Yes

After the test was completed the zip tie was chosen because it successfully intaked the balls with the greatest speed. Overall, we did not want to spend too much time determining the intake range since the intakes were designed to fold, meaning that we already had a significant amount of tolerance. This is one of the reasons why the testing procedure was less comprehensive for this aspect of the robot: for time efficiency's sake.



To the left is an image of the mechanical stop.

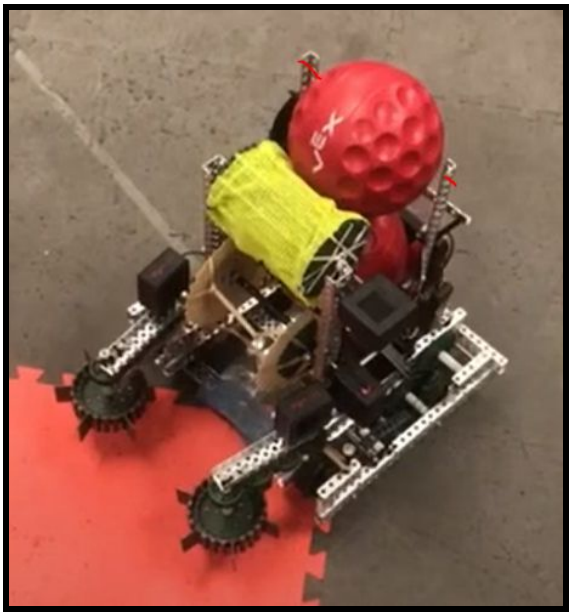
Hood Build Process (Team, 11-30, 12-2, 12-7, 12-9)

Objectives

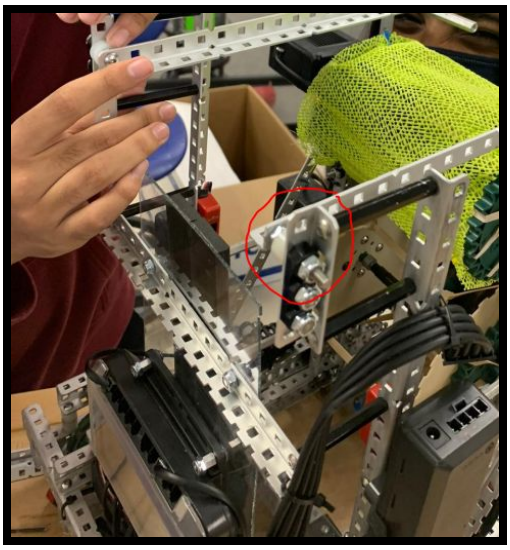
- Complete the polycarbonate cut outs for the hood
    - Make a mechanical stop for the hood
        - Adjusting the range of the mechanical stop to figure out the optimal position of the hood
    - Test the hood to ensure that it is working properly

Inspirations

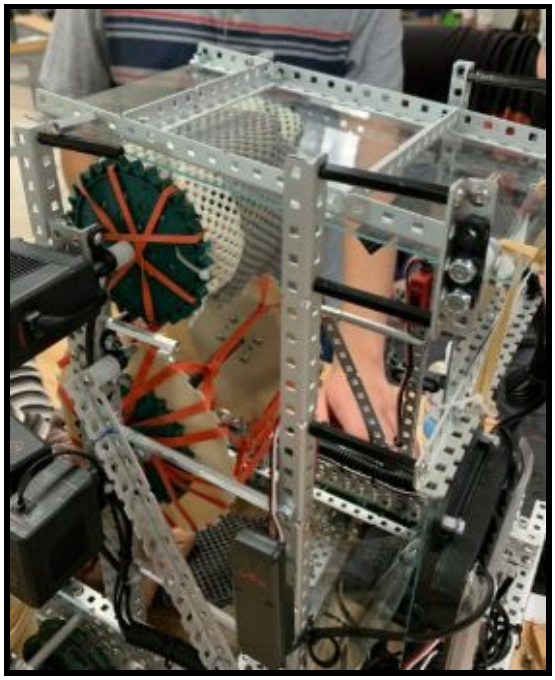
Since we wanted to avoid deploys all together, what we did was create a "folding hood," which was inspired by a New Zealand team named Wingus Dingus. This is the development of the robot at the current stage, and as of now, everything is built except for the hood. The plan is that at the points indicated by the red, we want to create a folding piece of polycarbonate. We want to put the polycarbonate on a pillow bearing and then mount it to the L-Channel using a screw joint. Following this idea, this was the prototype that was formed.



There were bearing flats connected to the L-Channel that is supporting the ramp, and then to these bearing flats were screw joints.



Now, based on the right image, we completed the screw joints. Then all we did to complete the hood was put a piece of polycarbonate across the L-Channels. We had the length of the polycarb just so it barely lays over the edge of the sprocket (9.5 inches). Here is the final result.



Problems With the Shooter Arc (Archis, Saachi, Andres, 12-16)

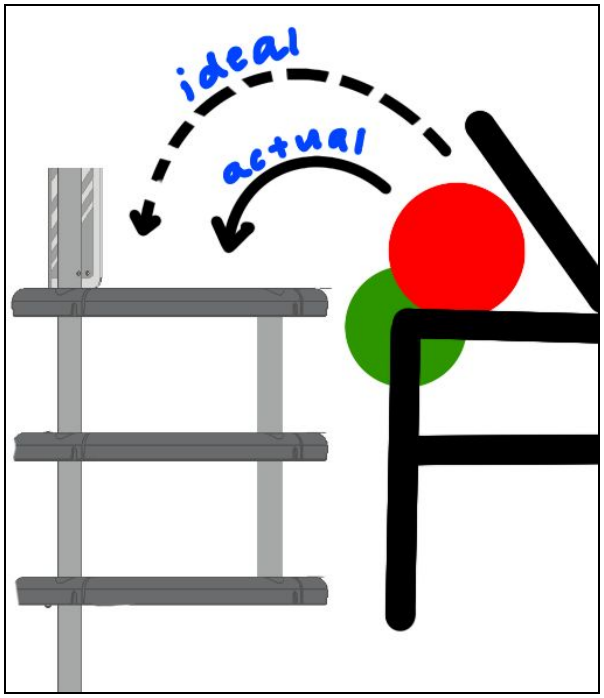
Objectives:

- determine the problem with the arc and troubleshoot
- Develop a solution to the overarching problem

The Problem

While testing our robot we noticed an issue with our ball out taking/shooting. When we shot the ball, the arc was less than ideal. The arc was far less than what we envisioned.

A visual representation:



This created inconsistency when scoring the ball. At times the ball would not go in the goal and would fall short. This required us to shift our driving methods and play style during skill runs,

Causes

Through initial troubleshooting we determined that the overall loss of speed was likely caused by these factors:

- Low battery power
- lack of compression and tension in rubber bands
- Lack of counterweight

Troubleshooting

When troubleshooting we followed this process:

- Video the shot
- Slow down the video and watch
- Discuss potential issues
    - Address potential issue and provide solution
    - Address next potential issue and provide solution
    - So on so forth

Battery

We started off by replacing the battery. Previously while testing, a low battery power caused our motors to slow down and led to an overall shorting of power. When we replaced the battery, the arc slightly improved. However, the difference was negligible and the inconsistencies came back leading us to believe that the battery was not the primary problem.

Compression

When watching the video, we noticed that the overall contact between the shooter and the ball was faulty. The contact between the ball and shooter was minimum and there was no compression whatsoever. Compression is necessary in establishing an ideal shot. The compression between the shooter and ball leads to greater friction and contact in return leading to a greater transfer of energy from the shooter to the ball.

To establish greater compression we wrapped mesh around the shooter. We experimented with different forms of mesh to determine which provided the ideal amount of compression.We first used razor mesh, the thinnest version. However, the razor mesh did not provide enough compression leading us to using regular mesh. Regular mesh provided too little compression once again and as such we double layered it.

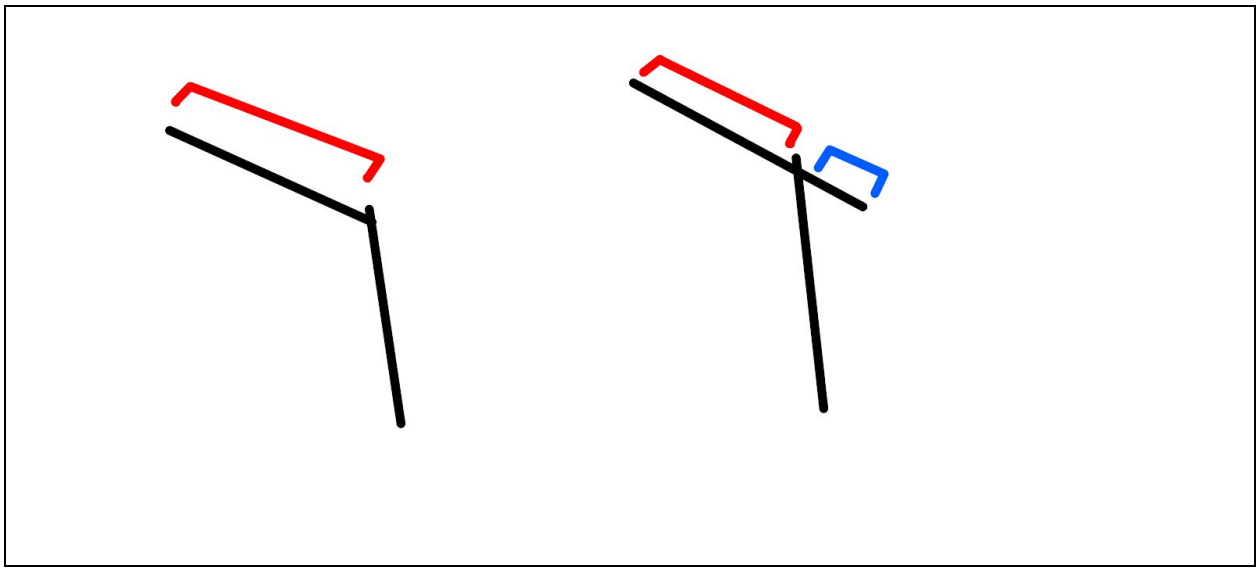
When we tested again, the compression seemed somewhat too much. The level of compression was causing the rubber bands to cave inward. However this also led us to switching out the rubber bands. As they were caving inward, it became evident that the bands had lost elasticity and therefore we replaced them with new ones. After replacing the rubber bands with fresh ones, this time, we simply used a single layer of regular mesh. The added compression from the fresh bands along with the single layer of mesh proved to be ideal.

Counterweight

With the new compression, the arc significantly improved however it was still falling somewhat short. Our robot required a degree of momentum to consistently score the balls.

We soon noticed that the ball was struggling to push up the hood during the shot. The hood acts as a weight and guide to the shooter. The weight from the hood helps provide added force to push the ball out. However, past a certain weight, the weight of the hood hampers the momentum of the ball. As such, we first tried to lower the weight of the hood. We removed the connecting bars of the hood. However the weight reduction proved to be insufficient.

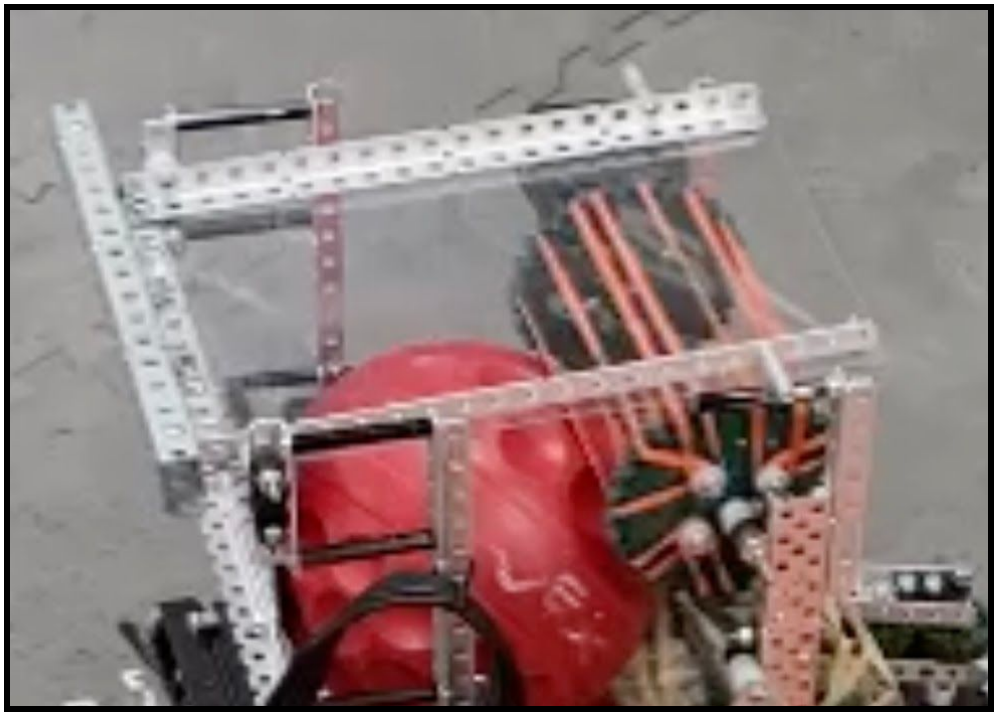
In response we moved forward the pivot point of the hood by one hole. Moving the pivot point shifts the amount of the hood which the ball must force up. Additionally, the part past the pivot point relieves the force required to push up the hood, acting as a sort of counter weight (think seesaw). A diagram of that is seen below:



Red = Part of the hood which weighs down against the ball

Blue = Part of the hood which eases the force required from the ball to produce sufficient torque to move the "lever"

The arc improved significantly yet once again was just slightly too short. In response, we added counterweights to the back of the hood. We attached two c-channels as well as two dangling screws + kep nuts. These were attached to the segment past the pivot point relieving the weight of the hood directly on the ball. A look at the improved hood:



With this our arc improved substantially. The shot became more consistent, easing our shooting restrictions.

Maintenance

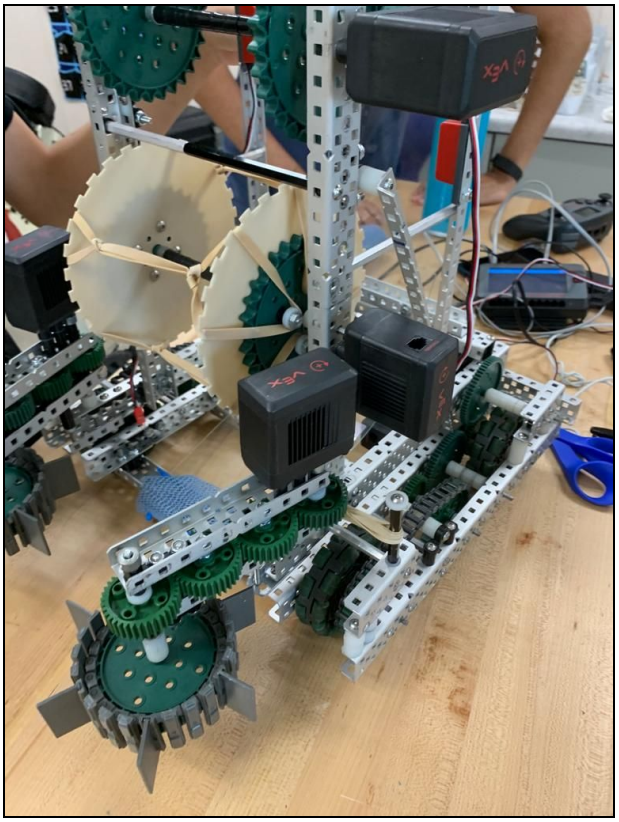
To maintain the consistency with our shot, we regularly replace the rubber bands. Additionally the mesh wears down from time to time and as such, we replace that as well. Furthermore, prior to driving, we use a fully charged battery to ensure no battery related errors.

Problems with the Sprocket Intake (Archis, Aayush, Conner, 1-2)

Objectives

- Create an intake that gets better contact on the balls and fits within the size constraints.
    - Define the problem
    - Brainstorm new solutions
    - Choose the best solution
    - Mount the solution
    - Test the solution

The Problem



The problems we ran into with the sprockets with tank tread and flaps was sizing issues and intake speed. The problems with size originated from the fact that our slightly expanded robot was now too big with the intakes as is. The second issue was the intakes were not as fast as we hoped them to be.

Ideas to fix the Problem

In order to fix the issue we came up with for potential solutions: omni wheels, the least stiff flex wheels, the stiff flex wheels, and the extra stiff flex wheels. The advantages to the omni wheels were availability and ease of use. The disadvantages are the stiffness and longevity. The flex wheels had the same advantages between with better compression (better compression as the stiffness went down) and

flexibility. The disadvantages of the three types of flex wheels are the difficulty to use and the limited size options.

Choosing the Flex Wheels

The least stiff flex wheels were chosen because they offered the most compression of the options, which would increase how fast we could intake, index, and shoot out a ball from our robot. This was significant because it gave us more flexibility in skills because the robot function took less time. The sizing issue of the flex wheels was a non issue because the size of flex wheel we needed was conveniently available in the robotics room. The difficulty of use of flex wheels were fixed by trial and error, where we just learned how to use them over time.

Mounting the Flex Wheels

First, the contact height of the old tank tread sprocket design was measured from the bottom of the intake c channel. Next, the robot was tilted 90 degrees, so that the intakes would be resting such that the bottom shaft collar of the final shaft of the intakes is accessible. Then, the old sprockets were removed along with all of the spaces on the final shaft of the intake. The flex wheels were then measured for thickness with calipers. Finally, we took the height measured earlier and subtracted the thickness of the flex wheels, put that

distance of spacers in and then the flex wheel with a shaft collar at the way bottom.

Testing the Flex Wheels

After mounting the flex wheels, the new intake design was tested for speed, contact, compression, and durability. As for speed, the intakes were faster than before. For contact, the ball was contacted at more points at the beginning of the intaking process and the same amount of contact for the rest of the intaking process. For compression, the compression was greater than before. For durability, the intakes were as durable as before. These all combined to make the intake better for the game because it enabled us to save more time in robot functions, more specifically intaking.

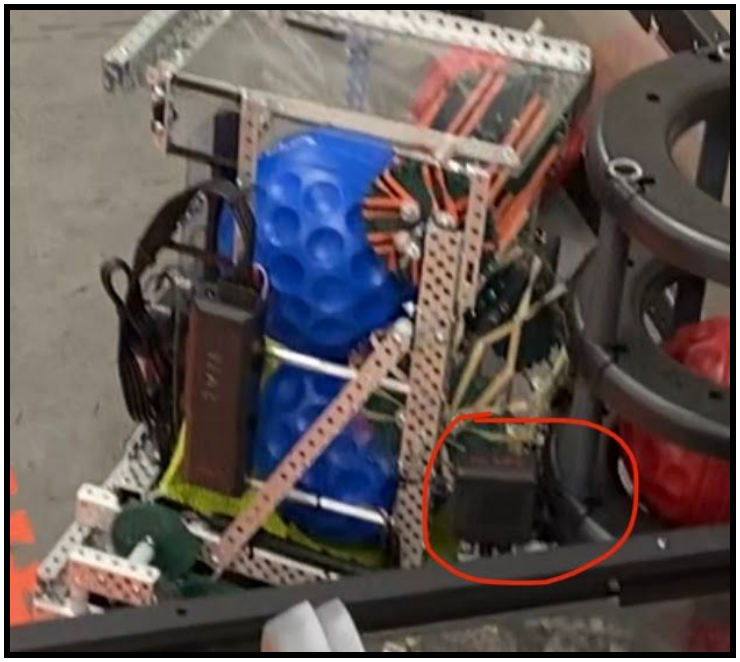
Problem with the Intake Motor Protruding (Tyler, Ashwin, Saachi, 1-11)

Background and Context

After careful analysis of the robot's performance during the first skills competition, we had deduced that our robot's intakes were more-so subpar than we had expected.

When reviewing footage of the competition, we realized that we felt we could've had more potential to achieve higher performance if our intakes were slightly more adjusted, as whenever the robot would go for a goal, the intakes would be the first contact to the goal. One of the main issues we saw with this design is the scrappy due to its head-on-collisions with the goal, and damage to the robot intakes was more than frequent.

As seen, after changing the custom sprocket to be smaller, it is now the motor that is the limiting factor for descoring. With this, we wanted to tackle this issue immediately.



Not only was this severely affecting our robot, but furthermore the motors prevented us from extending our intakes further into the goal the intended length. This in turn would lead to problems with the intaking and descoring from certain goals, as the flex wheels did not have enough grip onto the ball because of the lack of center contact and exerted force.

Intro and Objectives

Objectives:

- Move motors to a different position that will not interfere with anything important.
- Ensure that the motors will still be able to control the intakes while also not causing contact issues with the goal.
- Be able to maintain consistency with the intakes and descoring.

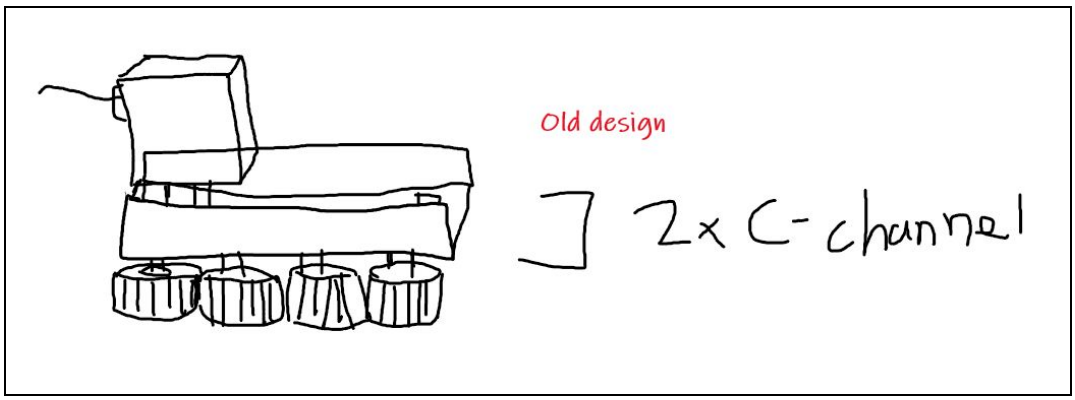
Resolution

We decided that the motors would most likely not move from the original place on the intake, as it would be too difficult to cram a motor onto the already limited space on our robot due to this being an unforeseeable event. Instead, we chose to increase the width of the c-channel to give the motor some room to sit on and rotate the motor 90 degrees, so that it sticks forwards and goes to the side.

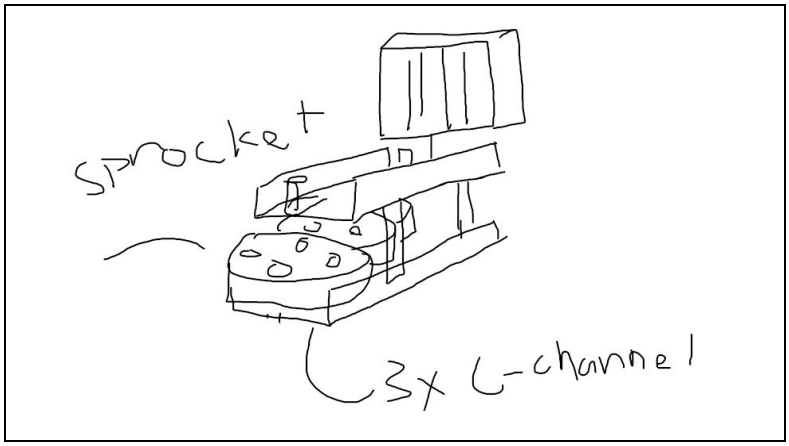
In the process of this, we also decided to do two things at once and nylock the motors to the second L-Channel we added vertically with the sprockets. Before, the intake would become worn out and unstable due to multiple collisions with the intake's flex wheels, causing stress onto the intake axle and motors, and therefore they would fall off. Since we added a standoff connecting the two L-Channels together it also gave us an extra option when it came mounting the rubber bands to the mechanical stop.

Moving the Motors

Before:



After:



We tested this modification by essentially going on the field and having our driver first tell us if it was qualitatively working. In other words, we had Andres practice descoring, and what he communicated was that previously, what he did in order to descore was shuffle around the goal for a little bit. Descoring in the past required a little more movements from his end. However, right now, he described that as long as he loosely lined up with the goal, the intakes would be able to descore without any effort except for one button click. Since it was clearly visible that the intakes were now working better and as intended, we left the solution as it is.

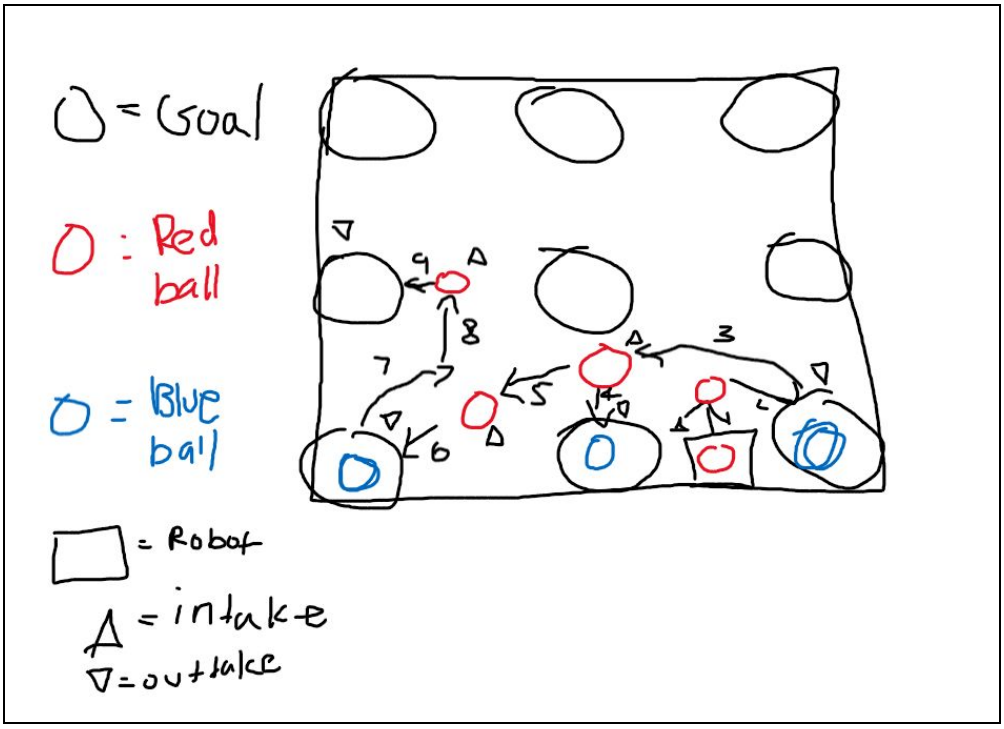
Skills Competition Reflection #1 (Team, 1-9)

Autonomous Period Reflection

During the competition, we had intended the bottom 3 goals to be effectively scored in with one ball, then the left middle goal would be scored in with a ball. Because of the lack of performance and programming in our robot, no descoring would be planned until driver control.

Another problem that severely slowed us down was the movement before intaking a ball. In the event of a robot intake, the robot would travel at a constant rate before reaching the ball, reach a halt, then at a much slower speed intake the ball. The result was a delayed and underwhelming performance of the robot. However, this would be very hard to deal with, as it was the surefire way to guarantee a ball pickup. If the robot was traveling at a faster rate than what we had intended, the robot's intakes would most likely have been misoriented rather than slowly brought into our robot.

In a normal event of all the objects being perfectly placed, our programming route would lead to all 4 goals being scored in as described in the picture below.

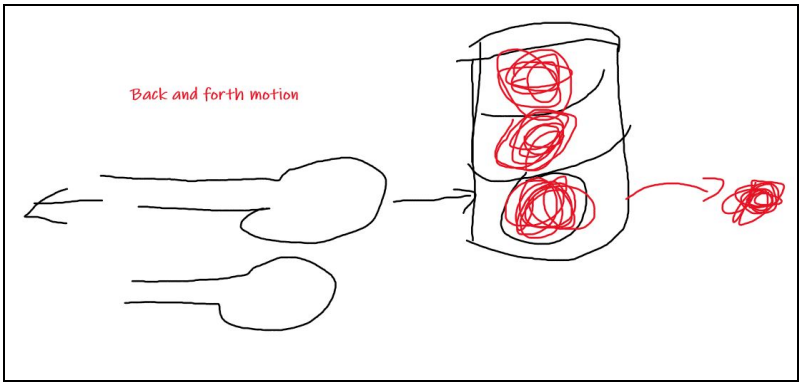


However, in an unnatural event during the skills run, our robot lineup was slightly off-par, resulting in a lucky descore of a blue ball in a corner goal and replacing it with a red ball as well. Other than that, our robot worked perfectly as intended and was able to use the goals to re-shift itself back into the proper position and lineup, adding extra points to our score.

Driver Control

Outtaking Issues

During our driver control period, we noticed that mostly everything went well



including our scoring, movement, and intaking all were better than enough. The results of the robot during the competition satisfied us, however one of the major flaws of

the robot was the descoring feature, which we could have saved a couple more points if it was resolved. While we spent very little time on scoring a red ball into a goal, most of the time at the goal was due to the lack of control and maneuvering of the descoring. In detail, in order for our robot to start descoring the center-goal, it would have to repeatedly drive back in forth, slamming the intakes of the robot as an outward force to push out the ball out the other side. This was not only very inefficient, but inconsistent and took a very long time to do so, as our robot was not able to effectively latch onto the center goal and intake balls from there. Reviewing the footage on the competition, from start to finish it took a whole ~5 seconds for the robot to outtake 3 balls from the center goal and leave it.

Moreover, during the scoring of corner goals, two balls would be removed from the goal in order to completely clear it. Multiple times during the driver control period our robot was intake both balls in the corner goal on the first try, and usually took at least two tries to get the second ball. Not only this, but our descoring was also very ineffective in corner goals, as the robot would have to turn around 180 degrees to release the balls, then turn all the way around just to score the red balls in.

Moving Forward

As noted before, in order to fix the issue, the Skills Analysis focuses on prioritizing these 3 things:

1) Improving the intake's reach

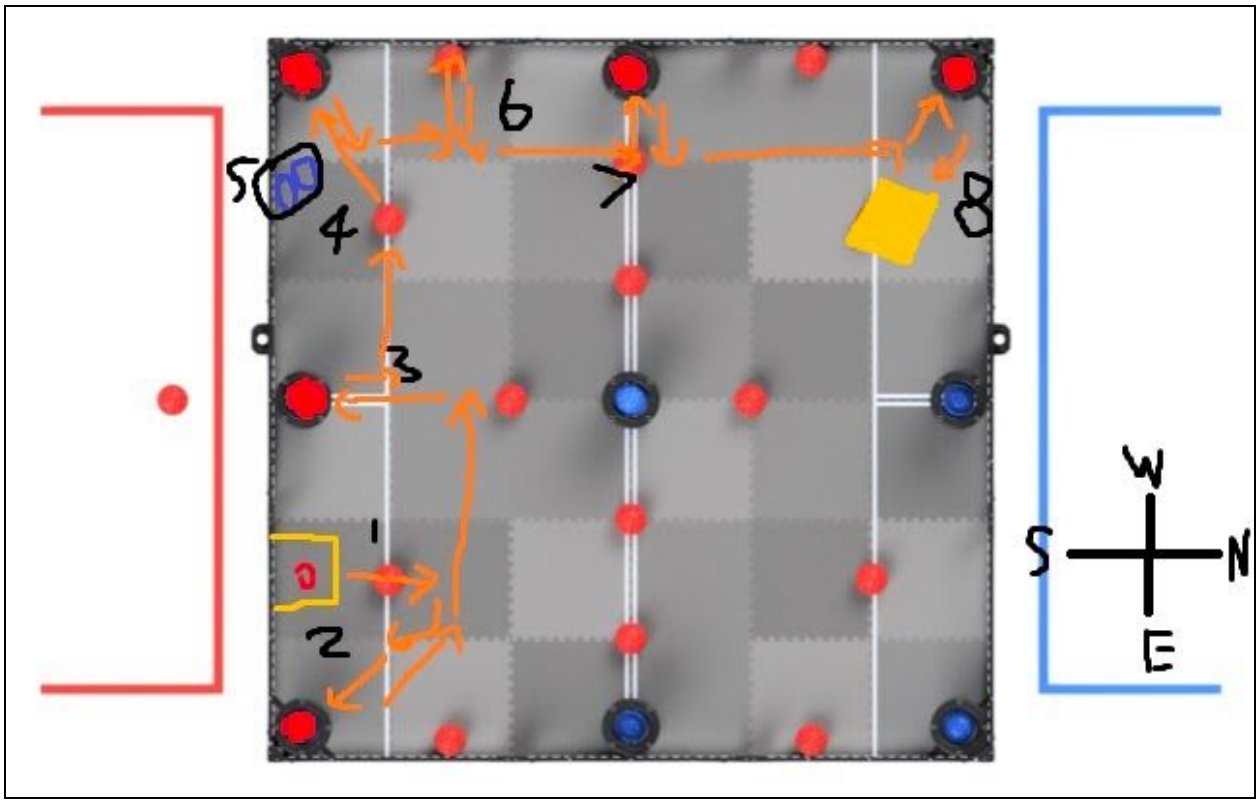
2) Making the balls go up our ramp smoother

3) Making the balls go up our ramp faster.

Personally, our team believes that we should immensely elevate our driver skills potential if we can change numbers like 4.5 seconds to 2 seconds and 3 seconds to ~1 second."

Skills Competition Reflection (Team, 2-20)

First, let's go over the most optimal run possible with our current programming skills:



1. We start in the south-east section of the field near the red corner goal with our preload loading into our robot. The robot then goes forward to the ball in front of our starting position.
2. Afterwards, it turns roughly 135 degrees towards the corner goal and drives forward into the goal. The robot then shoots one of our red balls into the goal and indexes another using our autoshoot-index function (for more information on this and all the PID movement functions we use, please refer to the programming notebook).

3. At this point, the robot still has one red ball and does 2 turns and 2 straight movements to reach the edge goal to the left of the corner goal. It then scores in that goal and positions itself for the next ball in front of the corner goal to complete the row.
4. However, before scoring in this corner goal, the intakes descore the 2 blue balls in the goal to add 2 additional points to our score.
5. The robot then scores the red ball and turns back 90 degrees towards the goals it has already scored and spins the shooter, indexer, and intakes out to remove the 2 blue balls in the robot.
6. It then turns to face north (north being the starting orientation of the robot) and drives forward briefly. Following this, it turns to face west to intake the ball along the side wall of the field. This serves to do 2 purposes, first, resetting the robot orientation to true west, second, intaking another ball.
7. The robot then drives towards the left edge and on the way picks up an additional red ball before scoring one of them in the edge goal.
8. Afterwards, the robot drives directly towards the top north-west corner goal and descores the 2 blue balls, scores the red ball, and drives away.

This totals to 69 points for our programming skills that ran perfectly, first try, during the competition.

Driver Skills

Our driver skills run was less planned out than our programming skills path. However, it worked out very well for us with us getting our highest score ever of 122. We did this by scoring at least one red ball in every goal, descoring every blue ball, and scoring 2 additional

red balls around the field. The video to the driver skills run is below:

https://drive.google.com/file/d/1upCidH6YBggxXnBfc5FbDRGPTkFMIzDW/view?usp=sharing

Moving Forward

With our next skills tournament being the Southern California State Championship, we want our driver and programming skills to be at the best it can be by then. For improvements on driver skills, the main improvement we could make would be enlarging the mechanical stop spacing to give us more tolerance for intaking balls. Right now in both driver and programming skills we have to slow down before intaking every ball or else we risk the ball bouncing off our intakes. As for specifically programming improvements, our PID constants could be tuned slightly better, and our slew rate could be larger to increase the speed of our robot. With more speed in autonomous, we could potentially take ownership of more goals, drastically increasing our score.

2496W Programming Documentation

Arnold O. Beckman High School

Authors: Aayush S., Andres G., Ashwin D. Tyler O.

Contents

Introducing the Programming Notebook

Why a new notebook?

If we were to try answering this question in one sentence, this would be the answer: our team consists of four programmers. We believe that it is only appropriate to be able to showcase the entire process of software, especially since half of our team -- 2496W -- will be working on it. Moreover, software is an aspect of the VEX Robotics Competition that is often overlooked, but throughout all our of experiences, we notice that the teams that are most successful are the ones who are able to make the best use out of software (e.g. 5525A two time world champions, 6627A TP Google winner and two time world division champion, 365X reigning skills champion).

How does our team operate?

We code using PROS CLI 3.14, an open source platform that was developed by Purdue University. Our preferred text-editor is Microsoft's VSCode, and the reason for this is that through VSCode we are easily able to sync the code with GitHub as the editor has built-in git integration. Moreover, another benefit of using VSCode would be the ability to "live-share." Rather than each person having their code independently on their laptop, instead, VSCode allows us to work on the same project at the same time, almost like a Google Document. Through this collaborative tool, we are all able to view the code together and make changes together, meaning that everybody is on the same page during the development of the code. Overall, we structured the development process in such a way that maximizes how much we collaborate.

Introducing the Members

Ashwin Dara, 12th

- 6th year competing in VRC, and has been a lead programmer for 4 years
- Will overlook all aspects of autonomous, operator control, and the custom libraries that we will be writing
- Hopes to learn more about control theory, data structures and algorithms, and the higher levels of mathematics behind programming

Andres Garcia, 10th

- 4th year competing in VRC
- Has been a lead programmer since freshman year
- Got introduced to coding through robotics in 7th grade and now a significant portion of his free time is dedicated to programming. He develops applications in Swift on the side
- He is responsible for writing and planning the driver control and autonomous code for the robot

Tyler Ogawa, 10th

- First year in VRC, but has some experience with writing scripts for other applications
- Wants to be able to receive first hand experience with APIs and programming with sensors
- He is responsible for the logic and code behind the uptake and flywheel of the robot

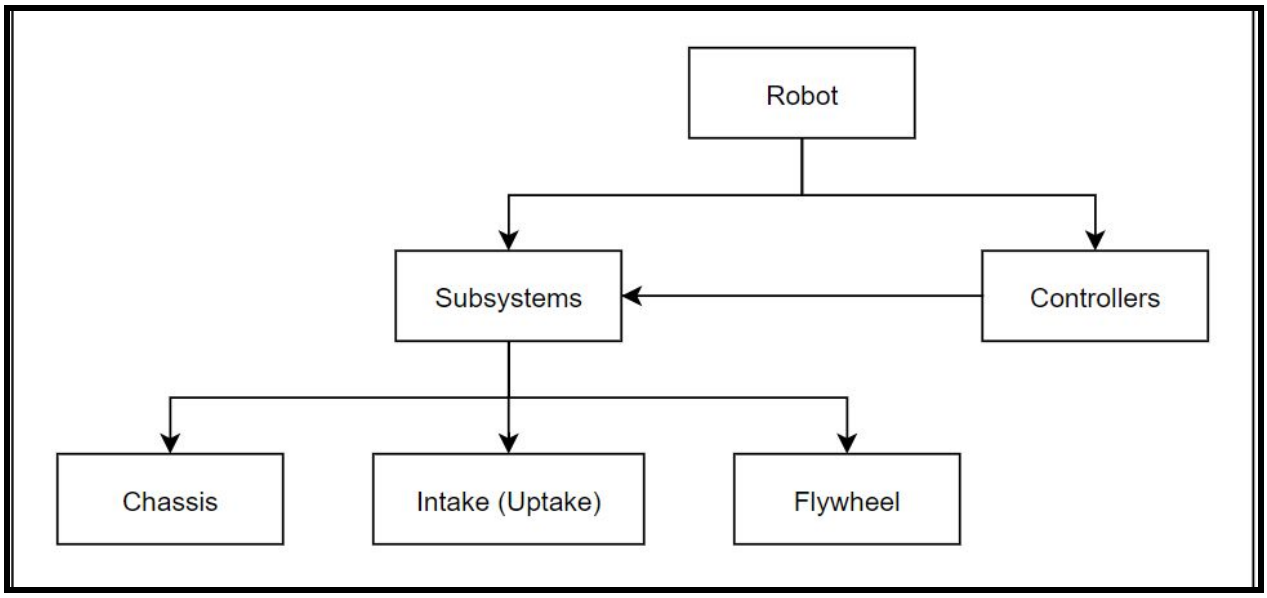
Aayush Seghal, 9th

- 3rd year competing in VRC, and has been a programming in 8th grade for Orchard Hills Middle School (8838D)
- Wants to continue learning as much as possible and continue growing as a coder, especially now as a high schooler
- Responsible for writing the operator control for the chassis and code for the front intakes

Introducing the Project Structure

Hierarchy of the Code

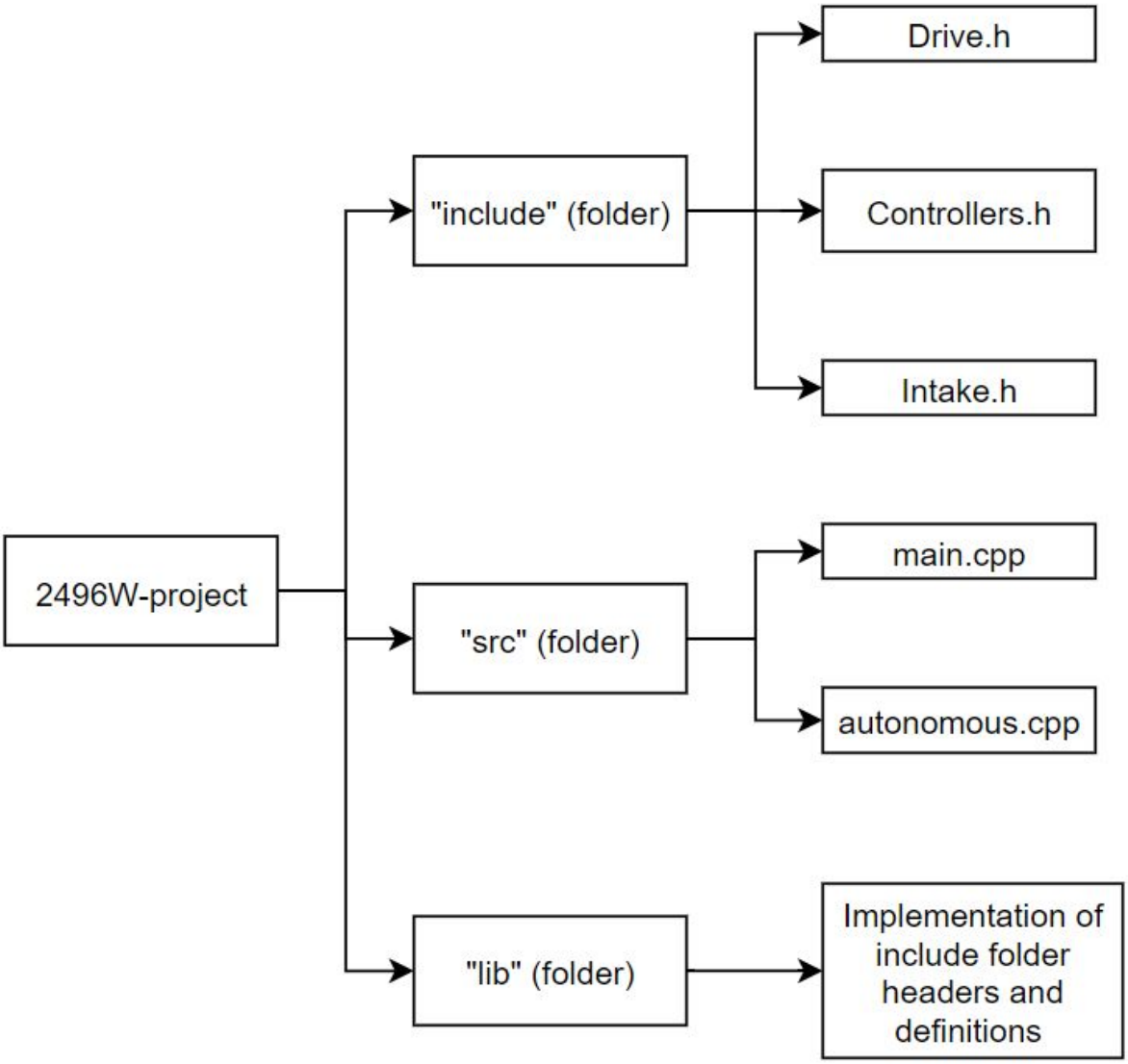
This season, we will be following the OOP (Object Oriented Programming) model. The reason for this is that it is quite convenient to have a parent subsystem class (which will have some methods that will be needed or useful for all of the other specific subsystem classes) and then have each subsystem of the robot be a more specific class. Here is a simplified example of the abstraction structure we are planning to use this season.



Overall, in the robot, there are subsystems. Each of these 3 subsystems will have the movement controllers (PID, Motion Profile, and other methods) inherited from the classes that we will code for the controller. By doing this, we are able to ensure that each subsystem has the appropriate data and methods for accurate movements. Moreover, this is ideal for having code that is concise since each subsystem's object will have their own controller configuration.

Project Structure of the Code

To accomplish the hierarchy we set out for, here is what the project structure will look like (folders and the directories).



Include Folder: within this folder will be the header files for all of the subsystem classes we will be using. For example, this would include "Drive.h", "Intake.h", and all of the controller's header files.

Src Folder: this will contain "main.cpp" and "autonomous.cpp." We want to avoid clutter in these two files at all costs and be as organized as possible. All that will happen in these two files is the updating of the *void opcontrol* and *void autonomous* with the scripts that we want to have in them.

Lib Folder: this folder will have the implementation of all of the header's in the "include" folder (more on implementation will be later).

2496W User Control Breakdown (Andres)

A strong robot, skilled driving, and consistent autonomous are the foundations of a top VEX competitor. The common link between the three of these is using automation and programming to your advantage. While building the robot, we constantly made decisions on what would best help our programming capabilities down the line, such as making space for line sensors on the hood guards so we could use "auto-shoot" in driver control (more on this later). In this log, I'll give an overview of our user control code, how it was coded, and the advantages we've gained.

Chassis Subsystem

The chassis is controlled by fairly simple reverse-arcade (forward and backwards movement on the left stick, turning on the right stick) code. As long as the absolute value the stick returns (from -127 to 127) is above 2, that much power is sent to the chassis motors. The left stick only takes the y-axis value and plugs that directly into both sides of the chassis. The right stick only takes the x-axis value and plugs that value into the left side of the chassis, and the same value multiplied by -1 to the right side of the chassis. If both sticks are being moved, the code adds the two values before sending the corresponding amount of power to each chassis side. The code is displayed below:

```
18    void opcontrol(){
19
20        int fPwr, tPwr; //forwardpower and turnpower
21        int rYaxis, lXaxis; //controller axis
22        rYaxis = controller.get_analog(ANALOG_RIGHT_Y);
23        lXaxis = controller.get_analog(ANALOG_LEFT_X);
24        fPwr = (abs(rYaxis) > 2) ? rYaxis : 0;
25        tPwr = (abs(lXaxis) > 2) ? lXaxis : 0;
26        chas_move(fPwr + tPwr, fPwr - tPwr);
27
28    }
29 }
```

Intake Control

The intakes, on their own, would have fairly simple code but it is wrapped in a lot of conditionals to prevent it from clashing with our autoshoot code. Basically, when button L2 (the far left bumper) is pressed, the shooter, indexer, and both intakes spin to push the balls out the bottom of the robot (think: reverse-intaking). This is mainly used to remove many of the opposing teams' balls from our robot at a time. Next, when button L1 is pressed, the intake and the indexer spin to intake the balls into our robot, and the shooter spins the opposite way to keep the balls in our robot. The main functionality of this button is to intake balls into our robot without shooting them. Additionally, this has the added benefit of indexing the balls into our robot in the same position every time which helps the autoshoot function properly. The simplified version of the intaking code is pictured below:

```
174         //Indexing and Outaking
175         if(controller.get_digital(DIGITAL_L1)){
176             intake_move(127);
177             indexer_move(127);
178             outake_move(-127);
179         }
180         else if(controller.get_digital(DIGITAL_L2)){
181             intake_move(-127);
182             indexer_move(-127);
183             outake_move(-127);
184         }
185         else{
186             intake_move(-127);
187             indexer_move(-127);
188             outake_move(-127);
189         }
```

Autoshoot

Autoshoot, my pride and joy. Autoshoot comes in two forms on this robot, the first is called "Autoshoot-No-Index" (abbreviated ASNI) and the second is called "Autoshoot-Index" (abbreviated ASI). ASNI occurs when the driver presses button R1 and results in the robot shooting a singular already indexed ball by running the indexer and shooter together briefly, and then the shooter alone. ASI works the same way except it also indexes another ball after shooting the previous one in the ready-to-shoot index position. ASNI is great for when I only have one ball and want to quickly score it in a goal. ASI is only meant to be used when you have another ball in your robot or you can easily intake another one to shoot two in rapid succession and reduce driver strain.

Let's dive into how it works! Since ASI is essentially ASNI with more steps, we'll use ASI as our main example. To trigger ASI, the user first presses R2 which activates a boolean

called aSI (to prevent confusion between the boolean and the abbreviation I will refer to the boolean as b_aSI). When b_aSI is true, ASI continually runs. This system is displayed below:

```
160             if (controller.get_digital_new_press(DIGITAL_R2)) {//rising edge press detection
161                 aSI = true; //Boolean which activates the continuous loop
162             }
163
164             else if(aSI){
165                 autoShootIndex(); //function that houses the ASI macro
166             }
```

Now let's break down what actually happens in the ASI function. The ASI function is divided into three main steps: shooting stage, indexing stage, resetting stage. I will break these down below:

1. **Shooting** - When the ASI function begins, an integer called stage is set to 1. This integer is how the code remembers what step it was on when it last iterated through the user control while loop. Stage 1 begins with the shooter and the indexer spinning to shoot out a ball, with the help of the hood, until the line sensor that is positioned at the ready-to-shoot index position no longer detects a ball. Once that happens, the code starts to close step one and activates a task (a.k.a. thread in some programming languages). If you think of the robot running your code as someone reading a book, we normally only read one book at a time. However, with multitasking/threading you can make the robot do the equivalent of someone reading two books at the same time. In other words, the robot will run the main code alongside some other code. This is very important for our shooting step because the shooting step uses the delay function which tells the compiler (it may be easier to think of the compiler as the robot) to temporarily stop reading the code. If we used the delay in the main task instead of the one we make for the end of step 1, the robot would freeze and the driver would lose control of the robot during the delay time. With that said, the function that is being run in the side thread essentially

changes the stage integer to 2, waits a bit, then tells the indexer to stop, then waits a little longer, then stops the shooter, and then activates a boolean called passedClose which tells the code it has finished the contents of the thread and the ASI function can move to step 2. Below is the stage 1 code followed by the function that is run in the side thread:

```
91  void autoShootIndex(){
92      if (stage == 1){//Shooting stage
93          if (index_bottom.get_value() < BALL_CONSTANT) {//Shoots First Ball
94              indexer.move(INDEXER_SPEED);
95              outake.move(OUTAKE_SPEED);
96          }
97          else
98          {
99              Task closingI (closeI);
100
101         }
102     }
```

```
81  void closeI(void* ignore){
82      stage = 2;
83      delay(20);
84      indexer.move(0);
85      delay(300);
86      outake.move(0);
87      passedClose = true;
88  }
```

2. **Indexing** - The indexing step (stage 2) works by having the indexer spin up towards the shooter and the shooter spin down towards the indexer continuously (similar to how it does when the L1 button is pressed, except without intake movement). This continues until the line sensor reads another ball in the ready-to-shoot position. Once that

condition is met, the stage integer is set to 3 to start closing the whole function. Below is the stage 2 code:

```
102     else if (stage == 2 && passedClose){//Indexing Stage
103         if(index_bottom.get_value() > BALL_CONSTANT){ //Indexes Next Ball
104             indexer.move(INDEXER_SPEED);
105             outake.move(-OUTAKE_SPEED);
106         }
107         else{
108             stage = 3;
109         }
110     }
```

3. **Resetting** - The resetting step stops the indexer and the shooter, resets booleans b_aSI and passedClose to false, and resets the stage integer to 1. This closes out the function and prepares it for the next run. Here is the resetting code:

```
111     else if(stage == 3){
112             indexer.move(0);
113             outake.move(0);
114             stage = 1;
115             passedClose = false;
116             aSI = false;
117     }
118 }
```

Autoshoot to Manual Fail-safe

The last main feature of the user control code is the ability to switch from autoshoot to manual shooting with just the press of the A-Button. When the A-Button is pressed, the manual boolean switches from false to true or true to false depending on its current condition. When the manual boolean is on, the code switches to manual shooting (running the indexer and the

shooter together for as long as the R1 button is being held). When the manual boolean is off, it uses the autoshoot code presented above.

Takeaways

In summary, our team put a major focus on optimizing our driver-control by using clever programming and automation. With these features we hope to reduce driver strain to make it easier to take advantage of the uncommonly fast chassis speed we have. With a chassis at this speed, if the driver is not able to handle it properly, it becomes more of a detriment than an advantage. Therefore, we think these programming techniques will increase the overall speed of all our robots actions while also letting us harness our main leg-up on other teams.

Full User Control Code

The full user control code will be the following pages.

```cpp
#include "main.h"
#include "robot.h"

//left motors are reversed
pros::Controller controller(CONTROLLER_MASTER);
pros::Motor rDriveT(DRIVE_RT, MOTOR_GEARSET_18, false, MOTOR_ENCODER_DEGREES);
pros::Motor rDriveB(DRIVE_RB, MOTOR_GEARSET_18, false, MOTOR_ENCODER_DEGREES);
pros::Motor lDriveT(DRIVE_LT, MOTOR_GEARSET_18, true, MOTOR_ENCODER_DEGREES);
pros::Motor lDriveB(DRIVE_LB, MOTOR_GEARSET_18, true, MOTOR_ENCODER_DEGREES);
pros::Motor lIntake(INTAKE_LEFT, MOTOR_GEARSET_18, true, MOTOR_ENCODER_DEGREES);
pros::Motor rIntake(INTAKE_RIGHT, MOTOR_GEARSET_18, false, MOTOR_ENCODER_DEGREES);
pros::Motor indexer(INDEXER, MOTOR_GEARSET_18, true, MOTOR_ENCODER_DEGREES);
pros::Motor outake(OUTAKE, MOTOR_GEARSET_6, true, MOTOR_ENCODER_DEGREES);
pros::Imu imu(IMU_PORT);
pros::ADIAnalogIn index_bottom(INDEX_BOTTOM_LINE_PORT);
pros::ADIAnalogIn index_top(INDEX_TOP_LINE_PORT);

void right_move(float speed){
    rDriveT.move(speed);
    rDriveB.move(speed);
}

void left_move(float speed){
    lDriveT.move(speed);
    lDriveB.move(speed);
}

void chas_move(float lspeed, float rspeed){
    lDriveT.move(lspeed);
    lDriveB.move(lspeed);
    rDriveT.move(rspeed);
    rDriveB.move(rspeed);
}

void reset_encoders(){
    lDriveT.tare_position();
    lDriveB.tare_position();
    rDriveT.tare_position();
    rDriveB.tare_position();
}
```

```
#include "main.h"
#include "robot.h"
#include "intake.h"

#define BALL_CONSTANT 2700
#define INTAKE_SPEED 127
#define INDEXER_SPEED 127
#define OUTAKE_SPEED 127

void intake_move(float intake_speed){
    lIntake.move(intake_speed);
    rIntake.move(intake_speed);
}

void indexer_move(float index_speed){
    indexer.move(index_speed);
}

void outake_move(float outake_speed){

    outake.move(outake_speed);
}

void index(){
    intake_move(127);
    indexer_move(127);
    outake_move(-127);
}

void sis(){
    intake_move(0);
    indexer_move(0);
    outake_move(0);
}

void autonASI(){
    while (index_bottom.get_value() < BALL_CONSTANT) {//Shoots First Ball
        indexer.move(INDEXER_SPEED);
        outake.move(OUTAKE_SPEED);
    }
    delay(20);
    indexer.move(0);
    delay(300);
    outake.move(0);

    while(index_bottom.get_value() > BALL_CONSTANT){ //Indexes Next Balll
        indexer.move(INDEXER_SPEED);
        outake.move(-OUTAKE_SPEED);
    }
    indexer.move(0);
    outake.move(0);
}

void autonASNI(){
    while (index_bottom.get_value() < BALL_CONSTANT) {//Shoots First Ball
        indexer.move(INDEXER_SPEED);
        outake.move(OUTAKE_SPEED);
    }
    delay(20);
    indexer.move(0);
    delay(300);
    outake.move(0);
}


bool aSN = false; //autoShootNoIndex
bool aSI = false; //autoShootIndex
bool passedClose = false;
bool manual = false;
int stage = 1;

void closeNI(void* ignore){
    delay(20);
    indexer.move(0);
    delay(300);
    outake.move(0);
    aSN = false;
    stage = 1;
}
```

```
void closeI(void* ignore){
    stage = 2;
    delay(20);
    indexer.move(0);
    delay(300);
    outake.move(0);
    passedClose = true;
}

void autoShootIndex(){
    if (stage == 1){//Shooting stage
        if (index_bottom.get_value() < BALL_CONSTANT) {//Shoots First Ball
            indexer.move(INDEXER_SPEED);
            outake.move(OUTAKE_SPEED);
        }
        else
        {
            Task closingI (closeI);

        }
    }
    else if (stage == 2 && passedClose){//Indexing Stage
        if(index_bottom.get_value() > BALL_CONSTANT){ //Indexes Next Balll
            indexer.move(INDEXER_SPEED);
            outake.move(-OUTAKE_SPEED);
        }
        else{
            stage = 3;
        }
    }
    else if(stage == 3){
            indexer.move(0);
            outake.move(0);
            stage = 1;
            passedClose = false;
            aSI = false;
    }
}

void autoShootNoIndex(){
    passedClose = false;
    if(stage == 1){
        if (index_bottom.get_value() < BALL_CONSTANT){//Shoots First Ball
            indexer.move(INDEXER_SPEED);
            outake.move(OUTAKE_SPEED);
        }
        else{
            stage = 2;
        }
    }
    else if(stage == 2){
        Task closingNI(closeNI);
    }
}

namespace intake{
    void opcontrol(){

        //If in autoShoot mode
        if(!manual){
            //autoShoot
            if (controller.get_digital_new_press(DIGITAL_R1)) {
                aSN = true;
            }
            if (controller.get_digital_new_press(DIGITAL_R2)) {
                aSI = true;
            }

            if(aSN){
                autoShootNoIndex();
            }
            else if(aSI){
                autoShootIndex();
            }


            //Indexing and Outaking
            if(controller.get_digital(DIGITAL_L1)){
```

```
                intake_move(127);
                indexer_move(127);
                outake_move(-127);
            }
            else if(controller.get_digital(DIGITAL_L2)){
                intake_move(-127);
                indexer_move(-127);
                outake_move(-127);
            }
            else if(!aSN && !aSI){ //autoShoot isn't running and no buttons are being pressed, stop the ball manipulating
subsytems
                intake_move(0);
                indexer_move(0);
                outake_move(0);
            }

            //Switching to Manual Shooting
            if(controller.get_digital_new_press(DIGITAL_A)){
                aSI = false;
                aSN = false;
                stage = 1;
                manual = true;
            }
        }
        else{
            if(controller.get_digital(DIGITAL_L1)){
                intake_move(127);
                indexer_move(127);
                outake_move(-127);
            }
            else if(controller.get_digital(DIGITAL_L2)){
                intake_move(-127);
                indexer_move(-127);
                outake_move(-127);
            }
            else if(controller.get_digital(DIGITAL_R1)){
                indexer_move(127);
                outake_move(127);
            }
            else if(controller.get_digital(DIGITAL_LEFT)){
                indexer_move(127);
            }
            else if(controller.get_digital(DIGITAL_UP)){
                outake_move(127);
            }
            else{ //if no buttons are pressing then make ALL the motors of the intake stop using motor.stop()
                intake_move(0);
                indexer_move(0);
                outake_move(0);
            }

            //Switching back to autoShoot
            if(controller.get_digital_new_press(DIGITAL_A)){
                aSI = false;
                aSN = false;
                stage = 1;
                manual = false;
            }
        }


    }
}
```

```cpp
#include "main.h"
#include "drive.h"
// author @Aayush
namespace drive{
    /*Tank Control
    void opcontrol(){
        int rPwr, lPwr;
        int rYaxis, lYaxis;
        rYaxis = controller.get_analog(ANALOG_RIGHT_Y);
        lYaxis = controller.get_analog(ANALOG_LEFT_Y);
        rPwr = (abs(rYaxis) > 2) ? rYaxis : 0;
        lPwr = (abs(lYaxis) > 2) ? lYaxis : 0;
        right_move(rPwr);
        left_move(lPwr);
    }
    */

   void opcontrol(){
       //Reverse Arcade
        int fPwr, tPwr; //forwardpower and turnpower
        int rYaxis, lXaxis; //controller axis
        rYaxis = controller.get_analog(ANALOG_RIGHT_Y);
        lXaxis = controller.get_analog(ANALOG_LEFT_X);
        fPwr = (abs(rYaxis) > 2) ? rYaxis : 0;
        tPwr = (abs(lXaxis) > 2) ? lXaxis : 0;
        chas_move(fPwr + tPwr, fPwr - tPwr);
    }
}
```

2496W Autonomous Breakdown

Control Engineering Methods Used

From our member's past experiences, One of the most significant challenges of the autonomous portion within VRC is ensuring that the program is accurate. In other words, being able to implement movement functions such that the robot is in the position and has the heading we desire requires significant corrections as there are typically disturbing forces that will throw off the robot. For example, the chassis may not always move straight due to friction or unevenness in power given to the motors. Sometimes, the wheel encoder count may not be accurate due to wheel slippage, a phenomenon when high acceleration causes the wheel to spin without any real translational motion. To prevent such inaccuracies, something crucial to our team this season was the research of control engineering techniques.

PID Controller

One of the most common closed loop controllers -- something which will allow the robot to move to a certain setpoint -- is the PID controller. Moreover, this is a controller widely used within industry as devices such as thermostats, drones, washing machines, etc use this algorithm.

Here is the mathematical representation of the algorithm:

$$u(t) = K_p e(t) + K_i \int e(t)dt + K_p \frac{de}{dt}$$

To elaborate on this, u(t) is equivalent to the power that the motor will be receiving. Kp, Ki, and Kd are constants that we will be needing to tune. "e(t)" is the function of error with

respect to time. Error is known as the distance between where we hope to move the robot to and where the robot is currently right now (we figure this out using the encoders). According to the mathematical equation, the further away the robot is from the setpoint, the larger the power assigned to the motor will be. What this means for the robot is that it will accelerate when it is approaching the desired target, resulting in a more accurate and smoother motion compared to just assigning the motor to an arbitrary power and then stopping. Here is the code representation of the PID controller (based on the mathematical equation):

```
#define LINEAR_KP 0.0f //is a constant that we will need to tune
#define LINEAR_KI 0.0f //is a constant that we will be needing to tune
#define LINEAR_KD 0.0f //is a constant that we will be needing to tune

int target = 0; //currently, the target was declared as "0"

float pid_power_calculator(int sensor_value){
    int error = target - sensor_value; //the error is equal to the difference between where we are and where we hope to be
    float p_power = error * Kp; //the power from the Proportional component is equal to the Kp times the error

    int integral += error; //the integral can be approximated to the sum of the error at small time intervals (dt)
    float i_power = Ki * integral; //the I-component power is equal to the integral times the Ki

    int delta_error = (target - sensor_value) - error; //the change in error can be approximated as the derivative
    float d_power = Kd * delta_error;

    return (p_power + i_power + d_power);

}
```

However, there are two big limitations that we noticed with this controller.

1) The power from the "I Component" will be extremely large due to the fact that the [near] continuous sum of the error will be a very large number.
2) The initial acceleration will be very high if the power begins at 0 since the error will be very high.

In order to improve these aspects, what we did was the following:

1) Limit the integral to only kick in when error is small enough, which is whenever the "P component" is not powerful enough.

2) We also put a maximum cap on the power coming from the integral component.

3) Limit the acceleration by making power addition increments.

Here is a prototype snippet of the code for the modified PID controller:

```
#define LINEAR_KP 0.0f //is a constant that we will need to tune
#define LINEAR_KI 0.0f //is a constant that we will be needing to tune
#define LINEAR_KD 0.0f //is a constant that we will be needing to tune
#define LINEAR_STEP 1.5f //this is the power step size
using namespace pros;

//making variables global since need to store
int target = 0; //currently, the target was declared as "0"
int integral;
float pid_power;

float pid_power_calculator(int sensor_value){
    int error = target - sensor_value; //the error = the difference between where we are and where we hope to be
    float p_power = error * Kp; //the power from the Proportional component is equal to the Kp times the error

    if(abs(error) < ERROR_BOUND){ //only if the error is within a certain bound does the integral accumulate
        integral += error;
    } else integral = 0;
    float i_power = Ki * integral; //the I-component power is equal to the integral times the Ki

    int delta_error = (target - sensor_value) - error; //the change in error can be approximated as the derivative
    float d_power = Kd * delta_error;
    pid_power = (p_power + i_power + d_power);
    delay(50);
    float delta_power = pid_power_calculator(sensor_value) - pid_power;

    if(abs(delta_power) > LINEAR_STEP){
        pid_power += (sgn(pid_power) * LINEAR_STEP);
    }
    return pid_power;
}
```

The behavior of this controller will mimic that of a trapezoid/triangle motion profile. Additionally, we plan on using the PID controller for both the straight movements and the turning movements. Overall, the PID controller with slew is the main component of the autonomous as it is able to provide significant accuracy while being able to limit acceleration and provide us the smoothness that a motion profile would have; the only difference is that our movements will be less computationally intensive.

Rewriting the Final PID Class

```
#ifndef PID_H_
#define PID_H_
#include "main.h"
class PID{ //Author @Andres Garcia
public:
    float m_kp, m_ki, m_kd;
    int error, prev_error, integral, derivative;
    float power, prev_power;
    bool t_slew_on = true;

    PID(float kp, float ki, float kd){ //constructor will set the values of the constants
        error = prev_error = integral = derivative = 0;
        power = prev_power = 0;
        m_kp = kp;
        m_ki = ki;
        m_kd = kd;
    }

    float calc (int target, float input, int integralKI, int maxI, int slew, bool m_slew_on){
        prev_power = power;
        prev_error = error;
        error = target - input;

        std::abs(error) < integralKI? integral += error : integral = 0;
        integral >= 0? integral = std::min(integral, maxI) : integral = std::max(integral, -maxI);

        derivative = error - prev_error;

        power = m_kp*error + m_ki*integral + m_kd*derivative;

        if(t_slew_on && m_slew_on){
            if (std::abs(power) <= std::abs(prev_power) + std::abs(slew)){
                t_slew_on = false;
            }
            else{
                power = prev_power + slew;
            }
        }
        return power;
    }
};
#endif
```

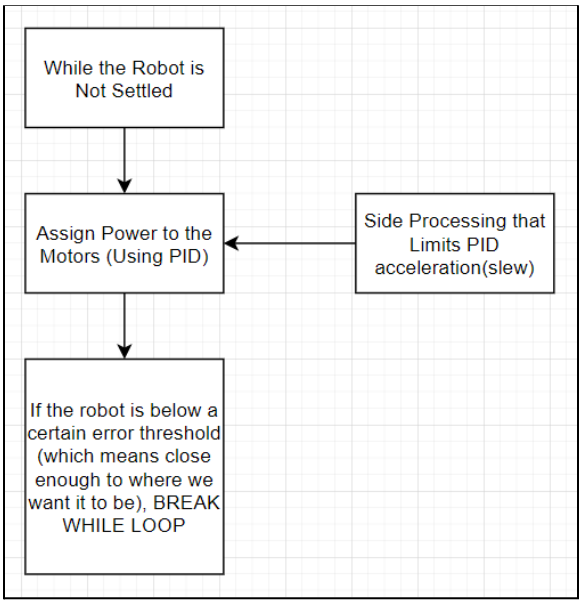
This was the class that we used within the final code (the interface and definition are combined for readability). Compared to the previous "prototype" draft code that was written, here were the following changes between the two codes:

1) PID is a Class. The reason for this is that we want to be able to make multiple instances of the single class. This way we avoid having to rewrite any code/calculators. Also, this opens the possibility of inheriting the class.
2) Std::min is employed to set the caps rather than using if-statements.
3) All of the other constants except for KP, KI, and KD are parameters for the power calculator. This was to create flexibility and make it easier to modify.
4) There is no delay(50). It will default delay to 10 ms.

The code snippet above was written by Andres Garcia.

Movement Functions Utilizing the Closed Loop Controller

Since we will be using both a turn and a straight move function based on the PID controller, we needed to be writing methods for that in autonomous mode. The logic that we wanted to follow is that the power from PID will continuously be calculated and assigned to the motors until the PID is able to settle. Here is a simple map of the logic that we wish to employ.



Linear Movement Function

Following the logic that was described earlier, here is how the movement function's definition was written:

```
int max_voltage = 0;
int max_turn_voltage = 0;
void move(int target, bool ask_slew, int slew_rate){
    //creating PID class instance using linear tuned constants
    PID straight(STRAIGHT_KP, STRAIGHT_KI, STRAIGHT_KD);
    //voltage is analagous to power, encoder_average is the average of both the sides, and heading is the difference in encoder count
    float voltage, encoder_average, heading;
    //this is the settle count
    int count = 0;
    //reseting the position in the beginning so the movements are relative
    reset_encoders();
    while(true){
        //encoder average is equal to the current translational position of the robot
        encoder_average = (rDriveT.get_position() + lDriveT.get_position()) / 2;
        //calculating the PID power
        voltage = straight.calc(target, encoder_average, INTEGRAL_KICK_IN, MAX_INTEGRAL, (sgn(target) * slew_rate), ask_slew);
        //calculating the heading correction power
        heading = (rDriveT.get_position() - lDriveT.get_position()) * LINEAR_STRAIGHTENING_CONSTANT;
        //putting a cap on the power of the voltage
        voltage = std::min(abs(voltage), (float)max_voltage) * sgn(voltage);

        chas_move(voltage - heading, voltage + heading); //moving the chassis at the speed of voltage with the correction power

        //conditions for breaking the while loop
        if (abs(target - encoder_average) <= 3) count++;
        if (count >= COUNT_CONST) break;
        delay(10); //delay to prevent hogging of resources
    }
    chas_move(0,0); //stopping the chassis
}
```

As we can see, we first make an instance of the PID class and set the constants in the constructor to be equal to the constants for straight movements. We need to make the PID class since we will be using the "calc" function. Afterwards, as long as the error is not less than 3 or passes a certain time range, then the while loop will continuously assign power to the motors. Moreover, we see a variable with the name of "heading_power." This is a variable that makes up the entirety of the straightening code to ensure that the robot always drives straight (we will go into more detail in a future section). Then we see the power being assigned through "cha_move." Finally, we have the conditions for breaking the while loop. Once the while loop is broken, we stop the chassis motors, as signified by the "chas_move(0, 0)."

Turn Function

We used the same logic from the previous linear move function for the turn function as well. However, there still are some differences. Rather than using the encoders to determine the position of the robot, we use the IMU, which is a gravity based sensor. This means that even if the encoders encounter wheel slippage, the IMU will always tell us the true length of our robot.

One important thing to note about the IMU sensor is that it requires calibration and is very sensitive to noise. To ensure that the sensor's values will not drift over time, what we did before writing this movement function was actually test if the IMU was able to give us consistent readings over a long period of time. Essentially, what we did was move the robot's orientation and simply print the value of the IMU to the terminal to ensure if it was stable. Here was the code used:

```
1   void debug_imu(){
2       std::cout << "IMU Sensor Value" << imu.get_rotation() << std:endl;
3   }
```

What the results concluded was that the IMU was still giving accurate readings even after 2 minutes and 30 seconds. With this, we were able to conclude that using the IMU sensor would be fine for the programming skills run, since it should not drift. Here is the code used for the turn function.

```
void turn(int target, bool ask_slew, int slew_rate){
    PID rotate(TURN_KP, TURN_KI, TURN_KD); //creating PID instance for calculate function
    float voltage, position, imu_start; //declaring variables
    int count = 0; //settle count

    imu_start = imu.get_rotation(); //beginning position

    while(true){
        position = imu.get_rotation() - imu_start; //the position is equal to the IMU right now minus where it was
        //this is relative movement for the turns

        //calculate function to give power to the motors
        voltage = rotate.calc(target, position, INTEGRAL_KICK_IN, MAX_INTEGRAL, (sgn(target) * slew_rate), ask_slew);

        chas_move(voltage, -voltage); //moving the chassis

        //break conditions (time and settle)
        if (abs(target - position) <= 1.5) count++;
        if (count >= COUNT_CONST) break;

        delay(10); //delay to prevent resources from being hogged
    }
    chas_move(0,0); //stopping the chassis once the loop breaks
}
```

Just like in the straight function, we first create the instance of the PID class so that we are able to use the calculate function. Then we go into the while loop immediately. We are constantly calculating the power to be assigned to be the motors and then we finally assign the power to the motors in "chas_move(voltage, -voltage)." The reason that one of them is negative is because we need the chassis to turn, meaning the right and left side need to be going in opposite directions. Then we have the same break conditions as before and a delay. Lastly, if the while loop exits, we make sure to stop the chassis completely "chas_move(0, 0)."

Timed Move Function

One other thing we implemented alongside using the closed loop controller was time functions. Although using timed functions in order to move the robot by VEX convention is considered inaccurate, what we realized is that driving into the goals to score will automatically re-align us. In other words, we do not need to have a negative acceleration going into the goal. Therefore, having a timed move function would save us a significant amount of time. However,

what we realized is that it would be highly beneficial to still include some components of the previous movement functions, such as the chassis straightening code. Here is what we coded:

```
void time_move(int power, int ms){
    int time_count = 0; //beginning the time at zero

    while((time_count * 25) < ms){
        //the difference between the right and left side is the encoder difference
        float encoder_difference = rDriveT.get_position() - lDriveT.get_position();
        //the correctional power (heading power) is equal to the encoder_difference times a constant
        float heading_power =  encoder_difference * LINEAR_STRAIGHTENING_CONSTANT;
        chas_move(power + heading_power, power - heading_power); //moving the chassis
        pros::delay(25);  //delay to prevent hogging of resources
    }
    chas_move(0, 0); //stopping the chassis when the while loop breaks after certain time
}
```

Auto-Shoot Functions

To make the process of scoring easier, what we did was actually use the autoshoot functions from driver control (documented in the other log), and all we did was use the same code but get rid of the initial condition where autoshoot was triggered by the press of a button. With this, we have two new functions for scoring the robot:

1) *void autonASI()* -- this was taken from the autoshoot index function within usercontrol
2) *void autonANSI()* -- this was taken from the autoshoot no index function

Additionally, we realized that we would be intaking a lot within the portion of autonomous and as of now, there is no specified function to help with intaking/indexing certain balls apart from autonASI(). With this, we also created an indexing function that will be able to intake balls from the field. The reason that it is a separate function for both starting the intake and stopping the intake is because we will be reusing the function a lot and we want to minimize the lines of code. Here is the code:

```
25  void index(){
26      /*
27      This functions moves the intake and indexer motors such that a ball would
28      be going up the robot. However, the shooter motor is reversed to prevent
29      any ballls from exiting the robot.
30      */
31      intake_move(127);
32      indexer_move(127);
33      outake_move(-127);
34  }
35
36  void sis(){
37      /*
38      This functions will stop all movement in the intake.
39      */
40      intake_move(0);
41      indexer_move(0);
42      outake_move(0);
43  }
```

Chassis Straightening Logic

The last thing that we wish to highlight as a team is the algorithm that actually allows for us to move perfectly straight no matter the field setting or even if another rams into our robot. We are able to follow the heading that we desire and the chassis will not be prone to being thrown off the path.

Although it was explained earlier, let's take a closer look into the algorithm.

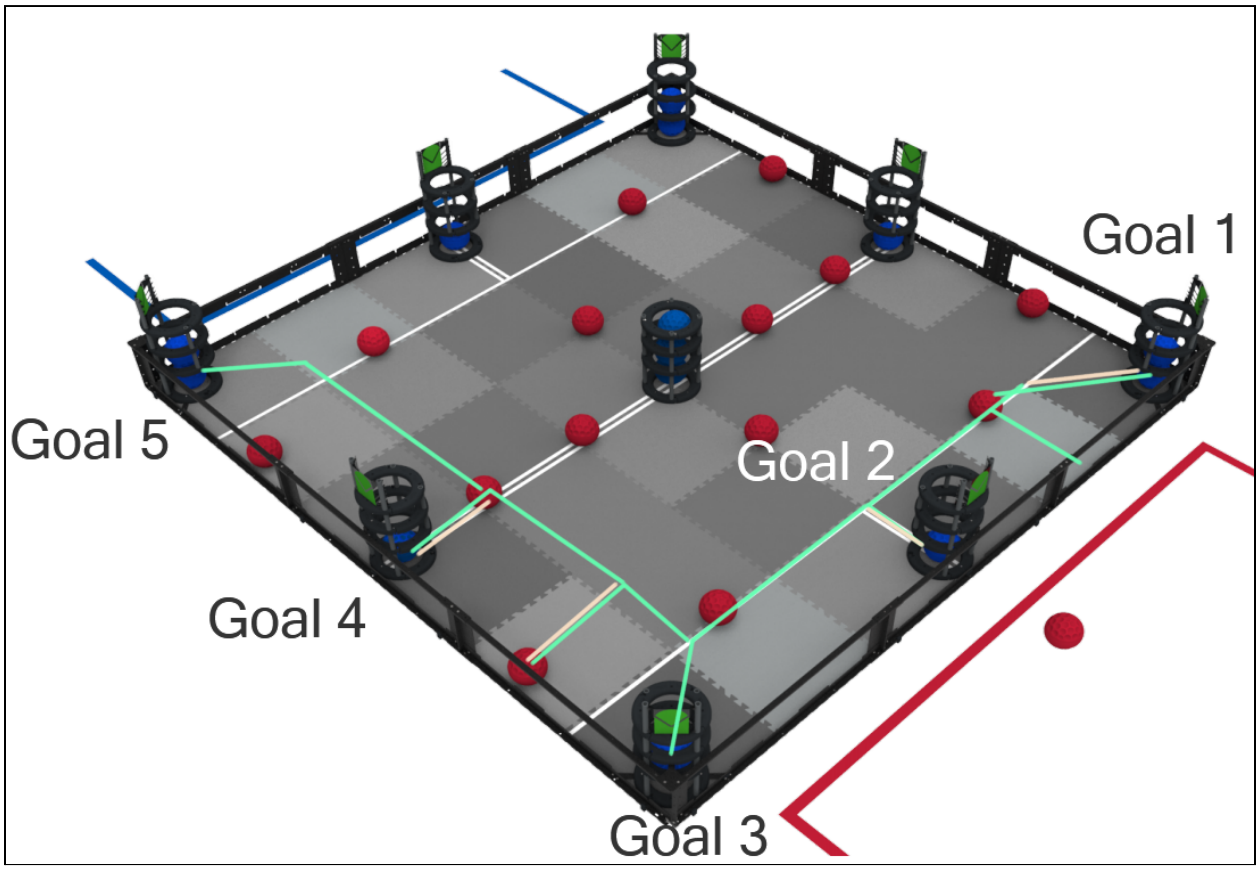
```
float encoderDifference = lDriveT.get_position() - rDriveT.get_position();
float headingPower = encoderDifference * 0.2;
chas_move(voltage - headingPower, voltage + headingPower);
```

Although it is only two lines of code, the two lines are extremely powerful. Let's say for example, the right side of the chassis has a tendency to drive forward more than the left. This means that the right encoder will have a higher value than the left encoder, meaning that the encoder difference will be a negative number. Since "voltage - headingPower" is the power of the left side, it actually receives a small boost in power. On the other hand, the right side

receives a reduction in power. Overall, we dynamically modify the power of both of the sides depending on the robot's position.

Autonomous Skills Path

Using all of the functions that were previously described we should be able to be any type of autonomous. For the most recent skills event, this was the autonomous route that we decided to take:



The route is that we begin to the right tile of the edge and then move forward to intake our first ball. Then, we turn towards goal 1, move forward, and score it. Then we move back, and turn to face towards the left side of the field. We then go forward and score within goal 2. Then we move backwards and turn once again to face the left side of the field. Then we move

forward, intake the ball, and score in goal 3. Then we back away, and turn to face the forward. Then we move forward, actually turn to the left 90 degrees, and pick up the ball at the edge of the field. Afterwards, we back away and face forward. Then we move to goal 4. After this goal, we back away from the goal, and rush to goal 5.

Overall, we realized that as a robot, it would be significantly easier for us to score the goals in the corner and the edge of the field rather than the center goal. Since we have been also very crunched on time throughout this entire season, we figured that keeping this as the programming skills route would be highly beneficial and easy to code. Moreover, this skills route can be easily improved upon, meaning that if we wanted, we would be able to expand the skills run by adding more goals in the same clockwise direction. For the latest competition (which is the state championship for us), we hope to be able to get all of the corner and edge goals. This should lead to about 80 point programming skills.

Full Autonomous Code

The full autonomous code/scripts will be on the following pages. Only the source files are shown, not the header files, to avoid redundancy.

```cpp
#include "main.h"
/*
* File Name: "PID.cpp"
*   The appropriate header file for this source file is not included within the documentation
*   as it would be redunant. With this, only the code definitions are shown.
*/
class PID{
public:
    /*
    Variables:
        "float m_kp, m_ki, m_kd": These are the three constants within the PID equation in the beginning of the
documentation.
            m_kp is equal to KP
            m_ki is equal to KI
            m_kd is equal to KD

        "int error": difference between where we were and were we want to be (target - position)
        "int prev_error": the error of the PID right before it updates
        "int integral": continous sum of the error once the error is small enough...meant to provide a boost of power to
overcome steady state error
        "int derivative": change in the error over time (error - prev_error) = e_{n} - e_{n-1 = delta e
        "bool t_slew_on": toggle for whether or not the slew is on

    */
    float m_kp, m_ki, m_kd, power, prev_power;
    int error, prev_error, integral, derivative;
    bool t_slew_on;

    PID(float kp, float ki, float kd){ //constructor init variables
        error = prev_error = integral = derivative = 0;
        t_slew_on = true;
        m_kp = kp;
        m_ki = ki;
        m_kd = kd;
    }

    /*
        Function: "float calc": returns the pid power depending on the inputs provided
            Param "int target": this is the setpoint that we want the robot to be at
            Param "float input": this float value is sensor/position that we will input into the PID
            Param "int integralKI": this the error bound in which the integral starts to acculumate
            Param "int maxI": this value is the maximum that the integral value can be
            Param "int slew": this value is the maximum acceleration of the robot
            Param "bool m_slew_on": this is a toggle for whether or not the PID is on
    */
    float calc(int target, float input, int integralKI, int maxI, int slew, bool m_slew_on){
        prev_power = power; //previous power is equal to power (0 when beginning the log)
        prev_error = error; //previous power is equal to the error immediately (error_{n-1})
        error = target - input; //error is the distance between where we are and where we want to be

        //only if the error is within a certain bound (abs(error) < integralKI) will the integral start to kick in
        std::abs(error) < integralKI? integral += error : integral = 0;
        //limiting the integral
        integral >= 0? integral = std::min(integral, maxI) : integral = std::max(integral, -maxI);

        derivative = error - prev_error; //the derivative can be approximated to delta error

        power = m_kp*error + m_ki*integral + m_kd*derivative; //total PID power

        //statement that limits the acceleration: if the change in power is greater than a certain amount, then the power
will only increment by a max change
        //the max change is equal to the slew rate
        if(t_slew_on && m_slew_on){
            if (std::abs(power) <= std::abs(prev_power) + std::abs(slew)){
                t_slew_on = false;
            }
            else{
                power = prev_power + slew;
            }
        }
        return power;
    }
};
```

```cpp
/*
* File Name: "Movement_Functions.cpp"
*   The appropriate header file for this source file is not included within the documentation
*   as it would be redunant. With this, only the code definitions are shown.
*/

#include "main.h"
#include "../include/PID.h"
#include "../include/Movement_Functions.h"

#define LINEAR_STRAIGHTENING_CONSTANT 0.2f
#define INTEGRAL_KICK_IN 50
#define STRAIGHT_KP 0.57f
#define STRAIGHT_KI 0.13f
#define STRAIGHT_KD 0.1f
#define MAX_INTEGRAL 25
#define COUNT_CONST 28
#define TURN_KI 0.38f
#define TURN_KP 1.8f
#define TURN_KD 0.0f

//maximum power that will be assigned to the linear and turn functions respectively
int linear_movement_max_voltage = 127;
int turn_movement_max_voltage = 127;

/*
Function: "void move": relative movement to move the robot linearly
    Param "int target": this is the setpoint that we want the robot to be at
    Param "bool ask_slew": this bool controls whether or not the robot is using slew
    Param "int slew_rate": this value is the maximum acceleration of the robot

*/
void move(int target, bool ask_slew, int slew_rate){
    PID straight(STRAIGHT_KP, STRAIGHT_KI, STRAIGHT_KD); //creating PID class instance using linear tuned constants

    //voltage is analagous to power, encoder_average is the average of both the sides, and heading is the difference in
encoder count
    float voltage, encoder_average, heading;
    int count = 0; //this is the settle count
    reset_encoders(); //reseting the position in the beginning so the movements are relative

    while(true){
        //encoder average is equal to the current translational position of the robot
        encoder_average = (rDriveT.get_position() + lDriveT.get_position()) / 2;
        //calculating the PID power
        voltage = straight.calc(target, encoder_average, INTEGRAL_KICK_IN, MAX_INTEGRAL, sgn(target) * slew_rate, ask_slew);
        //calculating the heading correction power
        heading = (rDriveT.get_position() - lDriveT.get_position()) * LINEAR_STRAIGHTENING_CONSTANT;
        //putting a cap on the power of the voltage
        voltage = std::min((float) abs(voltage), (float) linear_movement_max_voltage) * sgn(voltage);

        chas_move(voltage - heading, voltage + heading); //moving the chassis at the speed of voltage with the correction
power
        //conditions for breaking the while loop
        if (abs(target - encoder_average) <= 3) count++;
        if (count >= COUNT_CONST) break;
        delay(10); //delay to prevent hogging of resources
    }
    chas_move(0,0); //stopping the chassis
}


/*
Function: "void turn": relative turn using the IMU sensor
    Param "int target": this is the yaw target of the robot
    Param "bool ask_slew": this bool controls whether or not the robot is using slew
    Param "int slew_rate": this value is the maximum acceleration of the robot

*/
void turn(int target, bool ask_slew, int slew_rate){
    PID rotate(TURN_KP, TURN_KI, TURN_KD);
    //the variable names mean the same thing as within the previous method
    float voltage, position, imu_start;
    int count = 0;

    imu_start = imu.get_rotation(); //finding where the IMU begins

    while(true){
```

```
        position = imu.get_rotation() - imu_start; //should always be equal to zero when starting the method

        //calculating the power of the PID and putting a cap on the power of the PID
        voltage = rotate.calc(target, position, INTEGRAL_KICK_IN, MAX_INTEGRAL, (sgn(target) * slew_rate), ask_slew);
        voltage = std::min((float)abs(voltage), turn_movement_max_voltage);

        chas_move(voltage, -voltage); //moving the chassis: one must be negative

        //break conditions
        if (abs(target - position) <= 1.5) count++;
        if (count >= COUNT_CONST) break;
        delay(10); //delay to prevent hogging of resources
    }
    chas_move(0,0); //stopping the chassis if the while loop is broken
}


/*
Function: "void time_move": will move the robot for a specific period of time with heading correctional code
    Param "int power": power that will be assigned to the motors
    Param "int ms": time that the robot will be moving for
*/
void time_move(int power, int ms){
    int time_count = 0; //beginning the time at zero

    while((time_count * 25) < ms){
        //the difference between the right and left side is the encoder difference
        float encoder_difference = rDriveT.get_position() - lDriveT.get_position();
        //the correctional power (heading power) is equal to the encoder_difference times a constant
        float heading_power =  encoder_difference * LINEAR_STRAIGHTENING_CONSTANT;
        chas_move(power + heading_power, power - heading_power); //moving the chassis
        pros::delay(25);  //delay to prevent hogging of resources
    }
    chas_move(0, 0); //stopping the chassis when the while loop breaks after certain time
}
```

```cpp
#include "../include/PID.h"
#include "../docs/Movement_Functions.cpp"

/*
This is the autonomous that received the 69 point skills
*/
void autonomous(){

                        //First movement: go forward and intake the ball
                        index();
                        move(450, true, 2); //used to be 1 for the slew rate


                        //Second movement: turn towards the goal and stop intaking after turning
                        turn(125, true, 1);
                        sis();


                        //Move forward towards the first corner goal
                        time_move(70, 700); //going at a velocity of 70 for 700 MS
                        time_move(40, 480);
                        pros::delay(100); //Some time to rest to prevent bounce? (this works)
                        autonASI(); //shoot the ball


                        //Move backwards and turn away
                        move(-300, true, 1);
                        turn(146, true, 1);


                        //Go forward, turn, and now you're in position for the second goal
                        move(975, true, 2);
                        turn(-93, true, 1);


                        //Go into the second goal and score
                        time_move(45, 950); //need to modify this
                        time_move(8, 100);
                        autonASNI();  //NO INDEXING


                        //move backwards and turn to face the ball next to the corner goal
                        move(-200, true, 1);
                        turn(87, true, 1);


                        //Go forward and then go forward once again but slower
                        move(400, true,  2);
                        index();
                        move(600, true, 1);
                        pros::delay(50); //Give it some time to rest before cutting the indexing
                        sis();


                        //turn towards the second corner goal and move into it using time_move
                        turn(-35, true, 1);
                        time_move(80, 500); //speed of 80 for 500 MS
                        time_move(30, 400);
                        autonASNI(); //shoot NO INDEX


                        //Move backwards and turn away from the goal and towards the front
                        move(-250, true, 1);
                        turn(127, true, 1);


                        //Go forward, turn, and intake the perimeter ball
                        move(340, true, 2);
                        turn(-88, true, 1);
                        index();
                        time_move(40, 1220);


                        //Move back from the perimeter, turn, and now face the ball in front of the edge goal
                        //Move towards that ball
                        move(-310, true, 1);
                        turn(89, true, 1);
                        move(400, true, 2);
```

```
linear_movement_max_voltage = 25;
            move(450, true, 1);
            sis();


            //Turn towards the edge goal, moving forward, and score it
            turn(-90, true, 1); //turn towards the goal
            time_move(45, 750); //moving towards the fourth goal


            autonASI(); //shoot INDEX


            //Move back from the goal and turn to face forward (same direction as we are standing)
            move(-350, true, 2);


            turn(90, true, 1);


            //Move forward and then make a ~45 degree turn towards the last corner goal
            move(770, true, 2);
            turn(-44, true, 1);


            //Move forward for the goal, descore, and score
            time_move(90, 700);
            time_move(40, 650);
            index(); //intaking the balls to descore while in position to shoot
            delay(1000);
            sis();
            autonASNI();
            move(-350, true, 2);

}
```