Snowflakes
29457A

29457A

Engineering Notebook
High School

HIGH STAKES 24-25

St Chris Snowflakes

Table of Contents

Entries

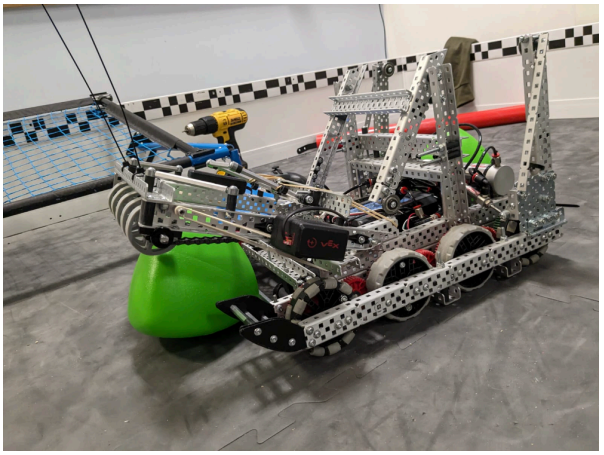
Appendix

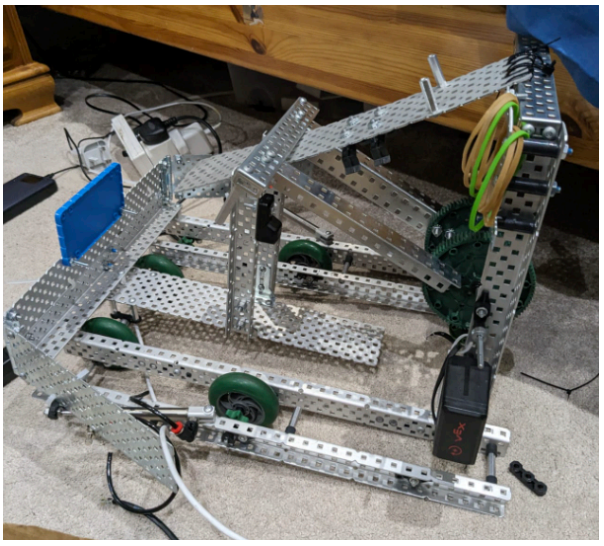
Team Introduction

Who are the Snowflakes?

We are the St Chris Snowflakes; we started out as a VRC team in September 2023, in our first season as a team in 'Over Under'. We set out to do our very best and we quickly found that we all loved the challenge of VEX and wanted to excel as far as we could; after a struggle, we qualified for UK nationals and siezed the oppurtunity to become the best we could. Our hard work payed off, and we went home with design award and a spot in the VEX World Championships; where we went in April and gained key experience to start this season.

Off of the back of Over Under we have acquired some very important skills necessary for competitive representation on both a national and global scale such as good building and coding practices which we hope to utilise in order to become even more competitive in all aspects of the V5RC competition where we weren't last year. We are also moving to a smaller team size as some of our members are moving on. This will mean we may have to work harder but also means a larger range of experience in other areas.





Team Introduction

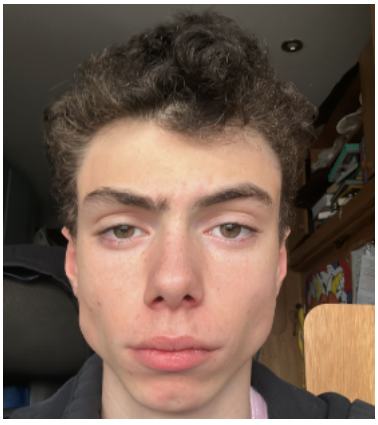
The members

Jonah Fitchew
- Co-Head Builder
- Driveteam

Hello I am Jonah Fitchew and my role within the team is the physical construction of the robot, documentation and assisting the driver during matches.

Aubert Seysess
- Co-Head Builder

Hello I am Aubert Seysses and my role within the team is to help design and virtually CAD the robot, also, I aid with the physical construction of the robot.



Daniel Dew
- Head Programmer
- Driver
- Driveteam

My role within the team primarily resides in the programming of the robot's code including autonomous and driver control. I also aid with the design and CAD phases of building. I am also the primary driver.

Daniel DaSilva
- Programmer

Hi I'm Daniel da Silva and my role within the team is to support the development of the robot's software and to help organise it into the logbook.

Team Introduction

Thomas Robb

- Head Tactics
- Driveteam

My role within the team is to brainstorm tactics and communicate with the other members to ensure that designs and tactics align. During competitions I am also responsible for taking note of performances, both our's and other team's; to help find possible alliances.

How To Use This Notebook

About this Notebook

TL;DR

For this season, we decided to deviate from the standard process for making engineering notebooks. We decided that, with the loss of our main logbooker we would have to share the notebooking duties; this meant that formatting could become inconsistent and we immediatly found that it took too long to format everything to the desired (exceptional) standard. Therefore, to cut down on time and improve the notebook's readability and functionality, we decided to adopt the *Notebookinator* template, which is an extension of the *typst* markup language.

Why Typst?

Several ways of creating notebooks for VEX exist, with most adopting visual editors such as google slides or hand writing their notebooks.
When deciding what we wanted to use for this season, we quickly ruled out hand writing the notebooks as mistakes could take valuable time to correct; neatness and clarity is often sacrificed; and the need for online collaboration is great. We previously used google slides with good results, however the formatting (e.g. colour coding, table of contents etc.) takes a significant ammount of time to maintain and can be very difficult to keep consistent when we all share equal role in notebook creation (as opposed to 1 person overseeing all notebook formatting).
We then landed on the possibility of using a markup language; and with the lack of flexibility from LaTex, Typst seemed like the best option. We had also noted a few teams success with using Typst, especially when using Notebookinator alongside it - for example team 53E (also the creators of Notebookinator) had a great Over Under notebook[1] using Typst.

Features

- Uniform formatting
- Notebookinator template
  - Easy cohesion with engineering design process
  - Built in components i.e. pros/cons tables
- Code blocks
- Built in table of contents
- Fully Digital
  - Neatness
  - Modern tooling
  - Easy submission
  - Cohesion with version control

[1] *Link to notebook*

The Snowflakes' Enginnering Ethos

Our Engineering Ethos

At its core, VEX Robotics is nothing but an engineering problem. It provides a goal, and the materials to get there. We believe that the key to success in Robotics intrinsicly lies in how you approach each problem; with open-mindness and the willingness to learn but most importantly the deterination to find the best solution possible.

> Engineering: The art of organizing and directing men, and of controlling the forces and materials of nature for the benefit of the human race.
>
> — Henry Gordon Stott

Our Engineering Design Process

For every new problem, we try to stick to an engineering method (similar to a scientific method) where different phases are used to maintain organisation. The process applies to all forms of design, including programming and sometimes even tactics.

Phases of Design

For each of the phases in our EDP, a corresponding icon is provided, these are used throughout the notebook to label a phase.

Identify Problem[1]
Each solution starts with a problem, this ranges drastically – for example, from 'We need a drivetrain' to 'this mechanism causes instability'. Problems can be raised by all members and, regardless of severity, it is something that must at least be addressed.

Brainstorm Solutions
Once the problem has been properly analysed, with root causes found, the team can move on to brainstorming solutions. Here, every team member can put forward possible solutions or fixes to the problem at hand, this is often accompanied with rough concept drawings. Additionally, finding use cases where each possible solution has been used effectively can be key to display a solution's viability.

Decide Solution
Once all possible solution have been brought to the table, one possible solution is picked to move to the next phase; to ensure that the decision is definitely the best one available, additional proceses can be used to decide the best (i.e. decision matrices). Ideally, all members offer their thoughts on the solutions.

Implement Solution
With one solution in mind, the solution can be expanded, now taking into account the smaller details while creating a plan of action for the designing and implementation of the solution. The solution is then designed and built – either physically or as a program.
Note that both the build and program icons are used during this phase.

Test Solution
Once the solution has been implemented onto the robot, we can begin testing the solution to find out how effective it is. This is a key phase as it shows us how the solution up to different scenarios.
Depending on its effectiveness, the results of a test may prompt us to move back into the implement phase, as changes sometimes have to be made. This creates a feedback loop that iteratively improves the solution until it meets our desired standards.

For our robot, we decompose the larger problem into a set of smaller, approachable problems. From there, each and every problem is approached using this EDP; this allows us to stay organised and avoid decision paralysis.

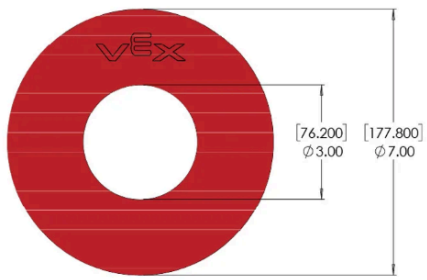
[1]Sometimes, if the problem is obvious (i.e. the need for a drivetrain), this phase is skipped due to mutual understanding.

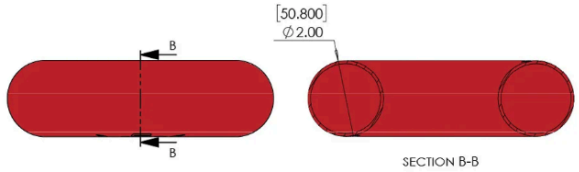
? The High Stakes Problem 2024/08/01

The Game

(All images found from the manual [1])

Scoring & Game Objectives



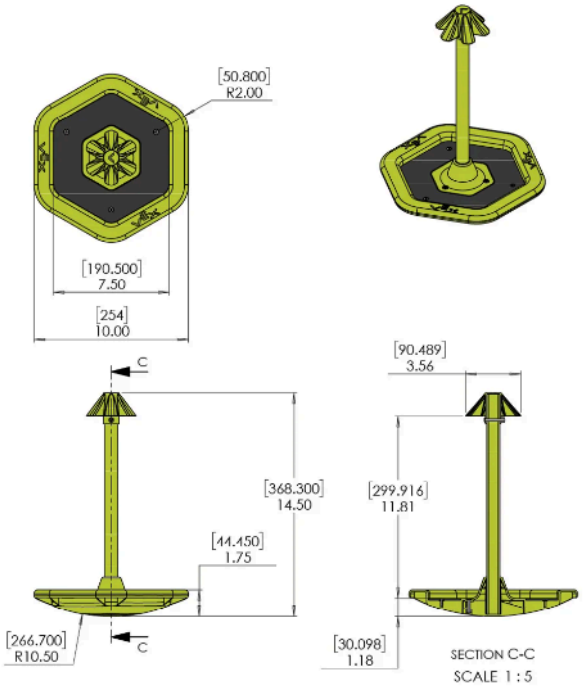


Game Elements: Rings
The majority of the scoring is done via these coloured rings.

- Outer diameter 7''
- Inner diameter 3''
- Height 2''

Potential Challenges

- Rings cannot roll on the floor, so manipulation is strictly contact based
- Large surface area in contact with floor so potential difficulty in manipulation



Scoring Object: Mobile Stakes/Goals There are 5 mobile goals ('mogos') on the field, and they can be freely manipulated by teams.

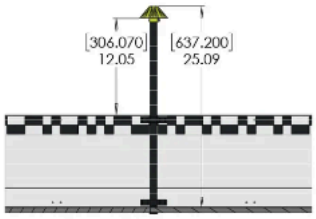
- 10'' diameter Hexagonal bird's eye view profile
- 14.5'' height
- Rubber cap to make descoring more difficult

Potential Challenges

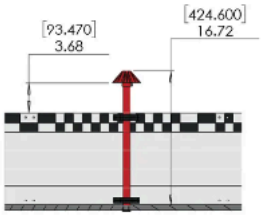
- Weighted bottom could make manipulation difficult
- Scoring would require an elevated mechanism.
- Rubber caps mean force must be required to score/descore

Designed by: Daniel Dew
Witnessed by: Jonah Fitchew

NEUTRAL STATIONARY GOALS:



ALLIANCE SPECIFIC STATIONARY GOALS:



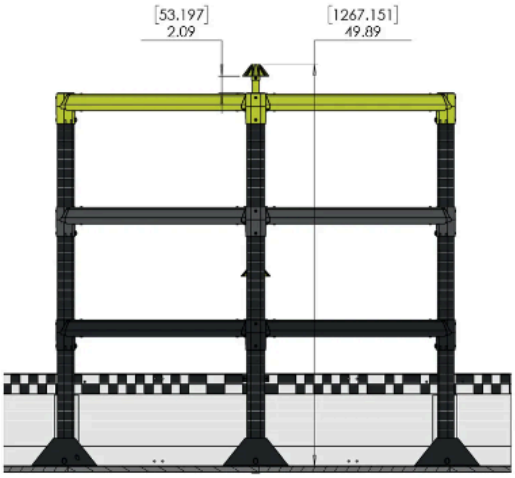
Elevated Stakes: Neutral and Alliance

There are 2 neutral stakes and 2 alliance stakes allowing for further scoring.

- Neutral stake 25.09'' tall
- Alliance stake 16.72'' tall
- Rubber caps (identical to mogo)
- Alliance stakes can only be scored by the corresponding alliance

Potential Challenges

- Stakes differ in height from each other (also from the mogo) meaning different or morphing mechanism to score on all.
- Placement (field perimeter) risks throwing rings out of the field (risking S1 infringement)
- Rubber caps mean force must be required to score/descore



High Stake and Ladder

In the center of the field, there is a 4′ ladder that teams can climb in the endgame to gain extra points. It also has a stake that can fit 1 ring at the very top.

- 49.89'' (4.165′) tall
- 3 tiers/rungs
- 4 sides

Potential Challenges

- Climbing structures requires lots of power and/or time
- High Stake would require extreme precision

Scoring Takeaways

- All scoring requires vertical capability
- Employing multiple methods of scoring (mogo, neutral/alliance stake) would require multiple systems or 1 complex system such as a lift
- Emphasis must be put on precision and reliability as there is little room for error

---

Designed by: Daniel Dew
Witnessed by: Jonah Fitchew

Rules Analysis

Scoring Table

Ultimately, scoring can be summarized into one table, called a scoring table. While it will not allow us to gain insight on the specific intricacies of each rule and the challenge it presents – it can allow us to quickly see what aspects of the game score more points, therefore allowing to adjust our game strategy.

Rule name	Points scored	Associated Rule
Autonomous Bonus	6 Points	*<SC2>*
Ring Scored on Stake	1 Points	*<SC3>*
Each *Top* Ring on Stake	3 Points	*<SC4>*
Ring Scored on High Stake	- (Ring is considered as the 2 above, and may apply bonus as seen in <SC9>)	*<SC9>*
Climb Level 1/2/3	3/6/12 Points	*<SC7>*

Format

To avoid simply regurgitating the rules (to people who already understand them), we are going to simply list some rules with a paraphrased description; then how it affects us; then potential solutions – if a rule presents no problem, we will not cover it.

e.g.

<RULE NUMBER>

- *Paraphrased rule description*

Problems

- This rule affects us like this
- It also affects us like this

Potential Solution

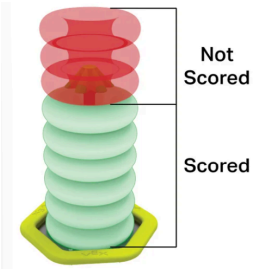
- This is one way we can mitigate the risk of infringement…
- This is another…

> **Note**
>
> Inspection, safety and general rules will not be covered, due to their relative simplicity.

Scoring rules

<SC3>

- To be considered scored on a stake, the ring must meet certain criteria:
  1. Ring must not be contacting robot *of same alliance*
  2. Ring is not contacting foam tile
  3. Ring is encircling the stake[1]
  4. Total ring count must not exceed max ring count of the stake (mobile & neutral: 6, alliance: 2, high: 1)

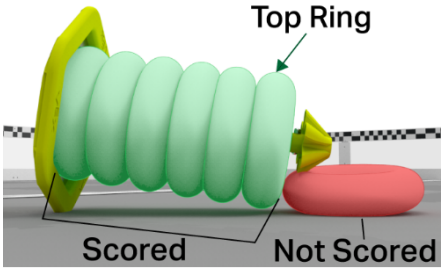


Problems

- Mogos with our rings on can be tipped to effectively descore some rings

Potential Solutions

- Driver may have to take care when scoring on neutral/ alliance stake
- Driver may have to guard or defend filled mogos



<SC5>

- A mobile goal is considered placed in a corner when it meets the following criteria:
  1. Mogo is contacting floor/foam tile
  2. Mogo is upright
  3. Contact with robot is irrelevant

> **Note**
>
> Only 1 mogo can be considered placed in each corner, even if 2 meet the requirements.

Problems

- Mogos can be knocked over to mitigate effect of corner

Potential Solutions

- Driver can guard/defend the corner, especially as robot contact is irrelevant.

<SC6>

- A mobile goal that has been placed in a corner will result in the following modifiers being applied to its scored rings:
  - Placed in **positive** corner:
    1. Values of all scored rings will be doubled
  - Placed in a **negative** corner:
    1. Values of all scored rings will be set to 0
    2. For each ring, an equivalent amount of points will be effectively removed from that alliance's score

[1]Long description omitted, see *https://www.vexrobotics.com/high-stakes-manual#sc3*

3. Points scored from auton bonuses and climbing cannot be removed

Examples included *here* .

Problems

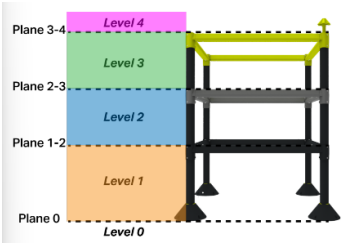
- Opposing rings scored in a positive corner can drastically change outcome of game due to 2x multiplier
- Ring scoring can be easily countered by placing them in negative corner

Potential Solutions

- Once again, large emphasis must be placed on defending scored rings and preventing them from being placed in a negative corner
- Putting emphasis on scoring on the elevated stakes could mitigate dependance on mogo scoring and corner defence/offence

<SC7>

- A robot has climbed to a level when the following criteria is met:
  1. Robot is contacting the ladder
  2. Robot is not contacting any other field elements
  3. Robot is not contacting any mobile goals
  4. The robot's lowest point is above that level's minimum height



Problems

- Climbing must be completely independant, it cannot rely on lower rungs or the floor

Potential Solutions

- When considering climbing, large power consumption – due to independant climbing – must be considered, possibly with use of a winch and/or a PTO[2]

<SC8>

- **Autonomous Win point** is awarded to *any* alliance that have completed the following tasks (as long as they have not broken any rules):
  1. At least 3 scored rings of that alliance's colour
  2. A minimum of two 2 stakes on the alliance's side of the autonomous line with at least 1 ring of the alliance's color scored
  3. Neither robot contacting or breaking plane of alliance's starting line
  4. At least 1 robot contacting ladder

Problems

- Even if we can complete as many tasks as possible, AWP is still reliant alliance teammate, especially with no. 3

Potential Solution

- Ensure prior coordination with teammate to ensure that they move off the line at the start[3]

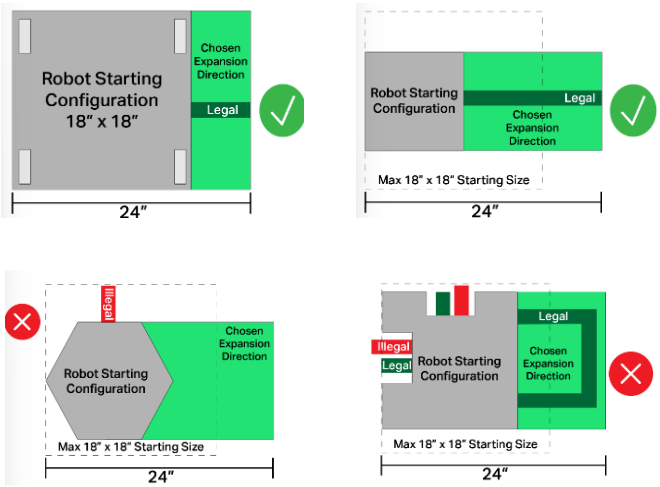
[2]PTO: Power Take-Off

Specific Game Rules

<SG2>

- Horizontal expansion is limited to an <u>additional</u> 6'' on **one** side.

> **Note**
>
> 6'' expansion is based on an 18'' x 18'' starting size, therefore robot can expand to the limit in **1** direction, then 6'' in the same direction.



Problems

- Mechanisims that rely on expansion must be contained within the footprint of the robot, or not expand over 6'' on one side only

Potential Solutions

- Design all expanding mechanisms to expand on one side only
- Use as little space of the 18'' x 18'' to maximise expansion capability

<SG3>

- **Vertical expansion is limited**; vertical expansion cannot break <u>2</u> or more levels of the ladder

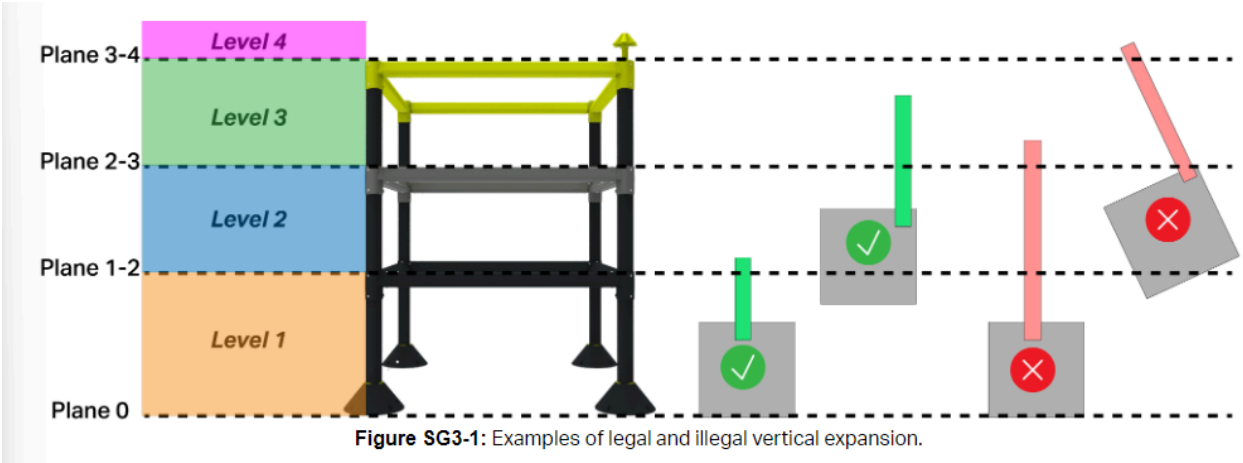


Figure SG3-1: Examples of legal and illegal vertical expansion.

[3]If the team does not have a (working) autonomous, advice/technical help can be given to simply move off the line, ensuring AWP

Problems

- This rule makes climbing to the top with 1 movement impossible – unlike in Over-Under – instead teams have to climb the ladder like… a ladder, using each rung and not skipping levels

Potential Solutions

- When designing climbing mechanisms, multi-stage movements must be incorporated; making sure that the robot does not break 2 or more planes no matter the rotation[4]

<SG4>

- Keep scoring elements **in the field**, rings that exit the field will be given to the corresponding alliances to reintroduce into the game.

Problems

- Accidentally removing rings from when, for example, scoring on wall stakes results in a minor violation

Potential Solutions

- Driver can take extreme care when attempting to score on wall stakes
- Line-up guides can be designed to aid the driver
- Lots of time on tuning the mechanisms to ensure they are not too powerful

<SG6>

- Possesion is limited to 2 rings and/or 1 mobile goal
  - Where rings scored on a stake do not count towards possesion count
  - Plowing multiple mobile goals is legal only when no mobile goals are possessed

Problems

- When attempting to rapidly score rings, this rule may be broken due to more than 2 rings being possessed
- Accidentally plowing a mogo while possessing one will result in a violation

Potential Solutions

- For both problems, driver care can be applied to avoid SG6 infringement
- A distance/colour sensor could be used in conjunction with an algorithm to stop manipulating rings once at the possesion limit
  - Using a colour sensor could allow for a colour sorting algorithm to only intake alliance's rings

<SG7>

- Dont cross the autonomous line during autonomous
  - Robots must not contact or break the plane of the autonomous line[5] during autonomous

Problems

- Accidentally crossing line due to lack of tuning or planning of the autonomous movements would result in the loss of ABP and AWP

[4]This is because the planes are measured from the perspective of the field (see long explanation *here*)
[5]basically the halfway line

Potential Solutions

- Extreme care and consideration must be used when planning out the autonomous movements

Primary Takeaways

Certain solutions appear more than once, meaning we can prioritise them to mitigate more risks at lower time/complexity costs.

Driver Skill

We have concluded that driving is a factor in nearly all the rules specifically targeted defence and offence, High Stakes is a skillful game that requires lots of practice from the driver. Putting emphasis on training our driver, using drills, friendly matches etc. must be a priority.

Control and Precision

We have also concluded that prescicion is key to avoiding rule infringement and also to maintain effectiveness. All mechanisms must be designed with extreme precision with lots of time allocate for fine-tuning to a) maximise effectiveness of mechanism and b) avoid breaking rules such as SG4 and SG7.

The Plan

This game and rule analysis has allowed us to form a plan on how we will approach the coming weeks as we organise ourselves to tackle the season.

Timeline Considerations

- The emphasis on driver practice means we will try our best to allow for plenty of driver practice
- The further emphasis on autonomous tuning means we will have to make ample room for autonomous testing in the timeline

Careful Design

- We will also be making sure that all our designs our designed with strength, prescicion and effectiveness in mind during all stages of the design process – this is especially prominent during the CAD phase

Teamwork Wins Championships

To ensure that we achieve our maximum, we must organise ourselves into a team that works hard to solve the problems presented to us. We can do this by sorting ourselves by roles and skillsets.

> **Quote**
>
> Talent wins games. Teamwork wins championships.
>
> — Michael Jordan

Goals & Objectives

Before we begin to design and build a robot for High Stakes, we must first define clear goals of what we *as a team* want to achieve. This will allow all members to work towards common objectives.

Our Goals For High Stakes

- A Force To Be Reckoned With
  - We want to become a team that is competitive and 'formidable on the field'
  - We can quantify this goal mostly by how high we place in qualifiers – we want to aim for top 5 seed in most competitions
- Nationals
  - We also would like to qualify for the UK National Championship
  - We achieved this goal last season so would love to get this oppurtunity again.
- Worlds/Internationals
  - This is a very difficult goal to achieve, with only ~5 spaces from the UK
  - We were also good enough to get this oppurtunity last season, with us going to the 2024 World Championships in Dallas

Organising Ourselves

Roles

We can first organise ourselves by our skillsets; we can do this by asigning roles[1]:

- Head Tactics: Thomas Robb
- Programming Support & Notebook: Daniel da Silva
- Head Programmer & Driver: Daniel Dew,
- Co-Head Builder: Aubert Seysses
- Co-Head Builder: Jonah Fitchew

[1]Roles can also be seen in the *Team Introduction* pages

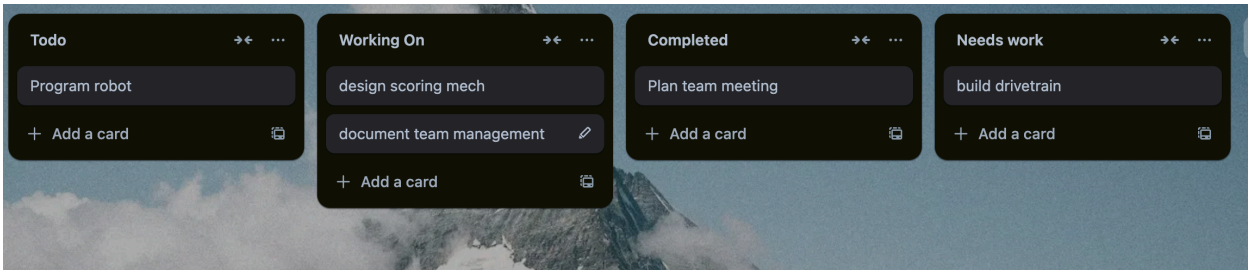
Orginising ourselves like this ensures that when a problem arises, we all know how to approach the challenge with our unique skillsets; it can also be helpful when introducing the team, as it helps judges and peers understand what we do within the team.

Project Assignments & To-Dos

Throughout the season, there will be lots to do over various projects – from designing a new robot (or subsystem), to writing a couple lines of code. To manage all these tasks, we will generally asign a project or to-do to a few members – using the roles as guidelines (e.g. buidling tasks asigned to builders, programming to programmers etc.); progress on said tasks will be discussed in in-person meatings[2], or using our channels of online communication.

Trello

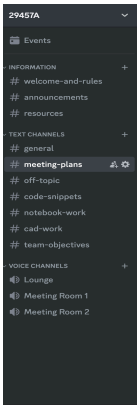
Trello is a project management software that allows teams to create dynamic to-do lists. 'Cards' can be dragged into different columns to display their state (e.g. 'todo', 'Working on' or 'completed'), we can asign members to certain tasks, give them deadlines, or add a checklist for each. We found it to be exceptionally useful last season, and we plan on using it for this season.



An example of a trello todo list

Discord & Online Communication

As a less formal method of communication, we use both Discord and Whatsapp to discuss team objectives and progress. Discord is especially helpful as it helps us to organise our chats into channels (see right). Another useful application to discord is sharing files and resources amongst ourselves – for example we can share small code changes on the discord for testing.

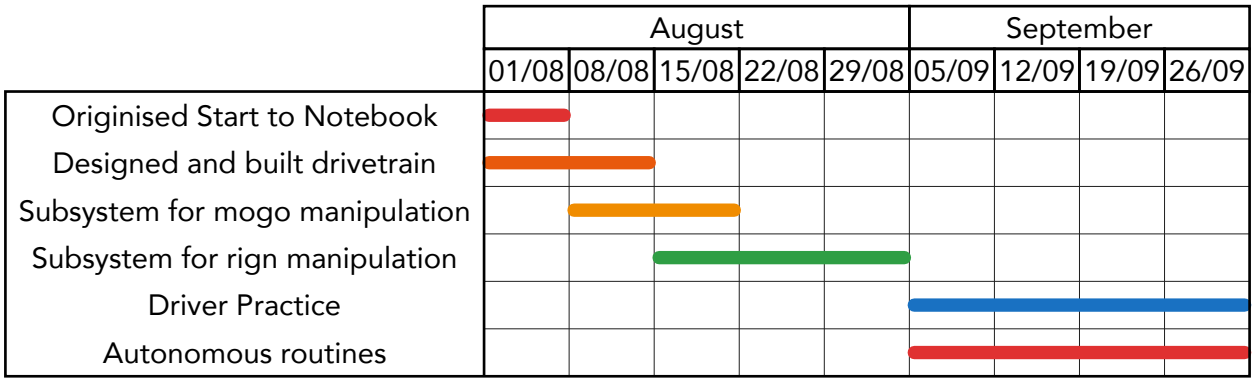


[2]Due to holdays, no in-person meetings have happened yet – our first full in-person meeting is scheduled for 03/09/24

Early Season Projects

We have a list of goals we want to acomplish for early season:

- Organised notebook
- Designed and built drivetrain
- Designed and built subsystem for mogo manipulation
- Designed and built subsystem for ring manipulation
- Substantial ammount of driver practice
- Reliable and semi-complex autonomous routines



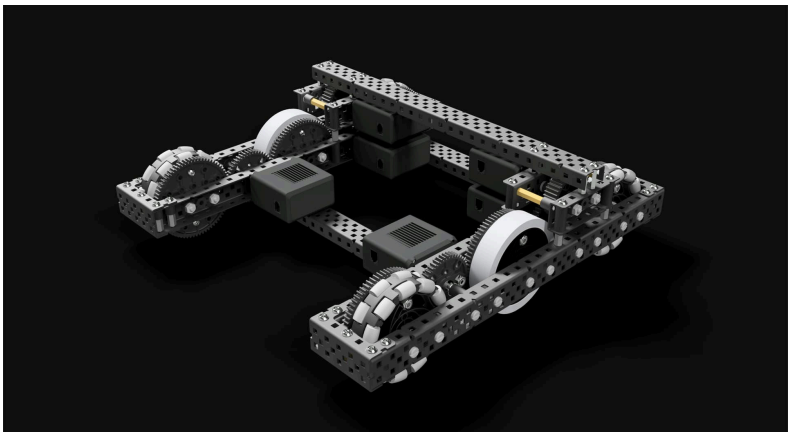
Brainstorming Drivetrain 2024/08/04

What are the different types of Drive?

The first solution to our somewhat obvious problem "We need a drivetrain"[1] is the most conventional VEX drivetrain of them all - the Differential Drive, which features mirrored sides in a typical "tank" formation and allows for independent manipulation of each side in order to achieve the movement we want to see. Second is the Omni Drive which features one omni wheel in the centre of the drivetrain and allows for horizontal movement as well as the conventional forward and backwards. Third is the Mecanum Drive which takes advantage of the VEX mecanum wheels which allow for strafing and a limited form of horizontal movement.

Differential Drive

Pros	Cons
• Simplicity - Easy to build, program, and maintain • Powerful - Excellent for pushing and traction-heavy tasks. • Stability - Will be balanced and support all sorts of mechanisms	• Limited manoeuvrability - Only supports forward, backward, and turning; no lateral (sideways) movement. • Not agile - Slow to make fine, precise movements or quick direction changes



From this forum thread [2]

Holonomic Drives

Pros	Cons

[1]The typical 'identify problem' page is omitted due to the obvious problem of needing a drivetrain.

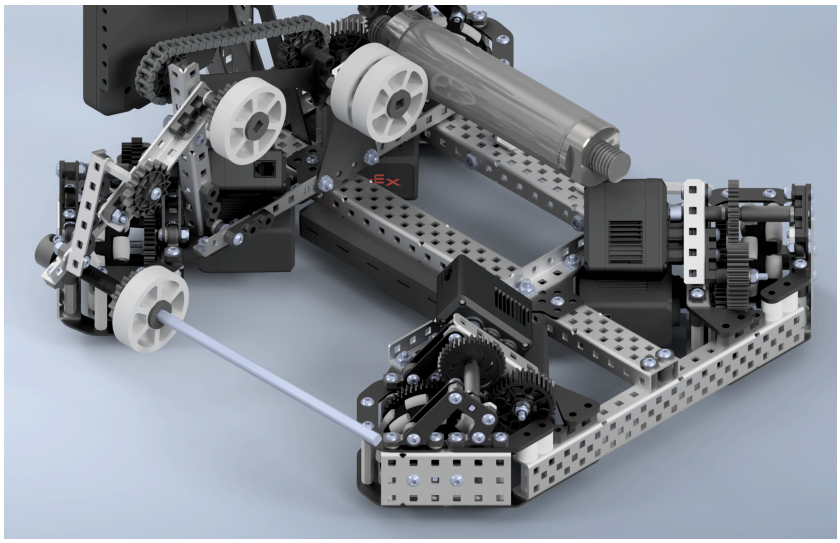
- Holonomic Movement - Can move in any direction (forward, backward, sideways, diagonal) with ease.
- Manoeuvrability - good for accurate and small adjustments
- Agility - Quick, fluid directional changes without rotating the robot.

- Lower traction - Limited pushing power due to low traction of the wheel in the middle
- Complex programming - Requires more advanced coding for full control.

Variants of Drives:

Holonomic Drivetrains

X Drive:



Here we see a render of an XDrive done by in_ithica | 3818 on the VEX CAD discord server [3]

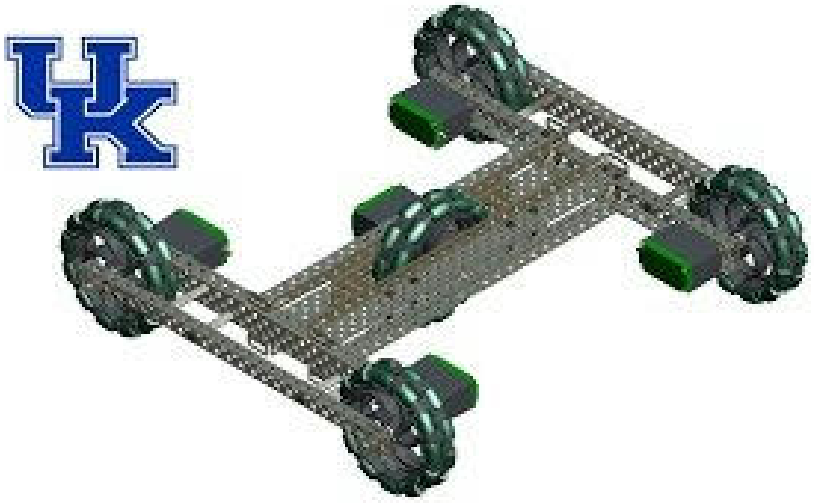
Pros	Cons

1. Higher Speed: X Drive is generally faster than both Mecanum and H-Drive, as it naturally provides more efficient power transfer for quick movements due to the 45-degree wheel orientation
2. Simplicity: Compared to Mecanum, an X Drive is simpler to build and program, as it doesn't require complex motor tuning or algorithms for strafing
3. Better Diagonal Movement: X Drive excels in diagonal motion without the power loss or inefficiencies found in Mecanum, making it smoother for navigating tight angles and corners

1. Lower Pushing Power: X Drive generally lacks traction and pushing force compared to Mecanum, making it less effective in scenarios requiring high torque or pushing resistance.
2. Inefficient Use of Space: X Drive's diagonal wheel layout takes up more space on the robot, which can limit the available area for other mechanisms compared to more compact configurations like H-Drive
3. Complex programming
4. Complex design – especially at the CAD stage

X Drive has traditionally been used by teams for extremely precise programming as it can provide an edge over more traditional drivetrains in terms of accurate movements in things like autonomous skills, however in a more traditional competition format with the goal of Tournament Champions, X Drive does not seem like the most popular choice. X Drives use a compound gear ratio to get around the difficulties in making the wheels diagonal, this can mean that it becomes quite difficult to build and maintain in comparison with more traditional drivetrains. Packaging the brain and pneumatic tank around this drivetrain can be difficult due to the odd motor placements.

H Drive:



Here we see a CAD model of an H Drive found on the website Purdue Sigbots made by the University of Kentucky [4]

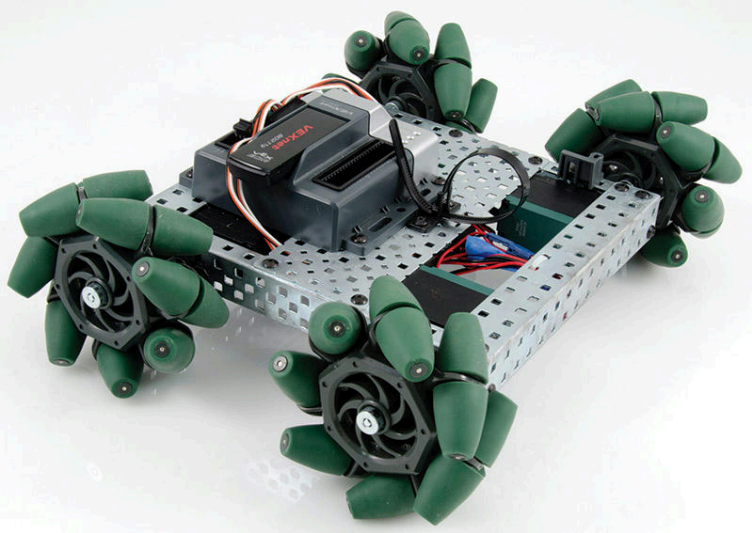
Pros	Cons

- Simplicity: The H Drive is easier to build and program compared to more complex holonomic drives like Mecanum or X Drive
- Good Strafing Ability: The central wheel allows for decent sideways movement (strafing), providing a balance between manoeuverability and straightforward design
- Efficient Use of Space: The parallel wheel layout in H Drive leaves more space for other components on the robot compared to an X Drive

- Limited Lateral Power: The central strafing wheel in H Drive has less traction and pushing power compared to the Mecanum or X Drive, leading to weaker side-to-side movement
- Less Agility: H Drive doesn't handle diagonal movement as smoothly as X Drive or Mecanum, reducing its manoeuverability in accurate and precise movements
- Vulnerability to Central Wheel Issues: The reliance on a single central wheel for strafing means that any failure or inefficiency in that wheel significantly impacts performance
- Programming Complexity: H Drives also consider some (although less than X and Mecanum) programming challenges
- Space: Having the wheel in the middle can result in less space for other subsystems such as odometry pods

H Drive is probably the most simple to build of all holonomic drivetrains with a single strafing wheel in the middle relying on the omni wheels rollers to provide lateral movement, the gear ratios used are the same as any traditional drive but one of the losses coming from the middle wheel is the lack of packaging ability in that area, no longer is the ability to use odometry pods to provide a higher level of accuracy during autonomous and no longer is the ability to have the pneumatic tank as low as possible to the ground due to the need for extra bracing and support of the middle wheel as inconsistencies and friction can have large effects.

Mecanum Drive:

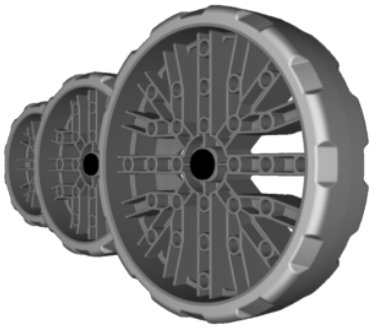


Here is a live model of the Mecanum Drive found on the Servo magazine website about holonomic locomotion [5]

Pros	Cons
• Holonomic Movement: Mecanum wheels allow for full omnidirectional movement, including forward, backward, sideways, and diagonal, giving excellent manoeuverability • Good Pushing Power: Compared to other holonomic drives like X Drive or H Drive, Mecanum maintains relatively good traction and pushing power • Versatility: Mecanum Drive offers a solid balance of movement options while still being able to handle various competition tasks, making it adaptable to different areas of the competition like skills	• Complexity: Mecanum Drives are more complex to build and design than simpler drives like H Drive, requiring precise motor control and alignment for effective movement. • Power Loss: Due to the angled rollers on Mecanum wheels, some power is lost during lateral movement, making it less efficient compared to a tank or X Drive • Programming Complexity: Mecanum Drives are also harder to program

The Mecanum Drive is one of if not the most frowned upon drivetrain in the whole of VEX, as it utilises the otherwise useless Mecanum wheels which take up a large amount of room and are not very versatile. However in a Mecanum Drive you acquire the ability to complete control and assurance that movement is not the problem, however in building this drivetrain the ability to gear is almost completely gone, as in order to maintain the very much wanted 3-4 wide hole gap between the drivetrain C Channels the mecanum wheel leaves no room for a gear, admonishing any gear ratios wanted.

Use of Traction Wheels:



From BRLS Wiki *[4]*

For our drive an important consideration was the inclusion and amount of traction wheels which we wanted to use. Having more traction wheels increases our grip but also reduces our skidding which can mean that our bot will struggle to do tight turns quickly. However, not having enough traction wheels means we may be easier to push around and struggle more to push others.
To maximise grip whilst retaining agility, it is common to use traction wheels in the centre to reduce unwanted pushing forces, whilst combining them with other wheels such as omniwheels on the edges for greater turning ability.

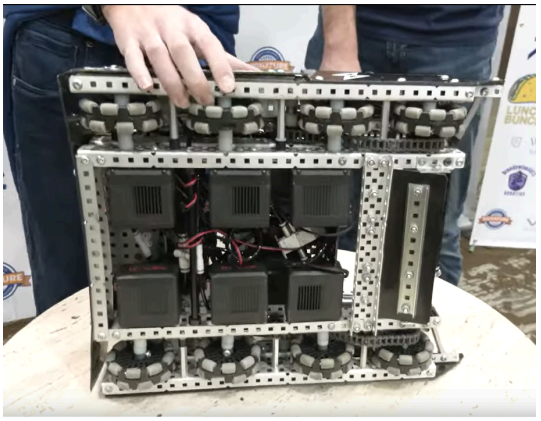
Pros	Cons
• Increased Grip: More grip allows for more power, which means it's easier to push game elements or other bots around. • Resistance: Since they have a single degree of freedom traction wheels make our robot more resistant against being pushed when attacking our defending	• Rigidity: Having one degree of freedom means that the bot will struggle more to perform tight turns, which may be important to perform midmatch or during the autonomous which often relies on turns being consistent and accurate.

Overall, it is important to decide how many traction wheels we use and where. Ultimately, deciding this depends on a teams game strategy and the type of bot they are going for.

All Omniwheel Drive:

Brainstorming Drivetrain 2024/08/04

Having our drive consist of only omniwheels has some advantages such as allowing for a lot more skidding and tight manoeuvering. However, a lack of traction wheels means our bot will have less torque to push and also will be easier to push. This pushing power loss can be compensated for by playing more evasively in order to outmanoeuver opponents. A good example of an all omniwheel drive was 9364H's bot in OU *Pits and Parts [6]*, who additionally engineered their bot for speed and agility to control the arena and punish double zoning in that game. See image on right for example (This image was cropped from Pits and Parts video cited above).



Number of Wheels:

Another important consideration is the amount of wheels, which we want on our drive. Similar to traction wheel ratio, having more wheel affects our torque-agility ratio. Having more wheels increases surface area in contact with the ground, which correlates with more traction. An increase in traction means that we have less skidding, which can be a detriment since skidding can be useful for turning, but it will also mean that the power is transferred more efficiently, which allows us to have more rpm or torque. See explanation for this on the right.

Energy = Power * Time
E=Pt
Energy = Force * Distance
E=Fs
Therefore
Fs=Pt
P=Fs/t
P=Fv
Power = Torque * RPM

4 Wheels

Having 4 wheels is the minimum number of wheels a drivetrain can have. It sacrifices power for manoeuverability but is easier to implement since it doesn't require motor stacking.

Pros

- Agility: Slides more so it has greater theoretical turning ability
- Easy to implement: Since there are fewer wheels there's fewer gears to deal with allowing for more space for game specific components such as wings, intakes or lifts
- Lighter: Less components so robot will weigh less meaning motors have less load to move

Cons

- Less power: Fewer wheels so there's less traction, which means less pushing power
- Lighter: Less weight so it's easier to push around

6 Wheels

6 wheels is a balance between manoeuverability and power. However, it's trickier to implement since it requires motor stacking.

Pros	Cons
• Agility: Still slides but not as much as a 4 Wheel Drive • Power: Balanced power but still less traction than an 8 Wheel Drive	• More complex: Requires motor stacking so its design as a drivetrain is more complex, which can mean it's more difficult to implement game scoring subsystems.

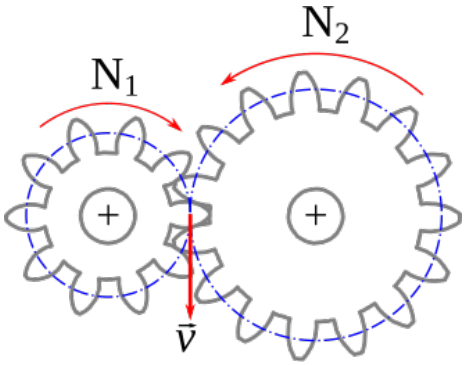
8 Wheels

An 8 Wheel Drive provides the most traction of the three options, which gives it the most power but means it will slide less and so will struggle with turns more.

Pros	Cons
• Most Power: Has the most power because of the greater traction • Heavier: More wheels and more gearing weigh more so it's more resistant to pushing and will push with more force	• Less Agile: The higher traction stops the robot from sliding as much so it will have less turning ability. • Heavier: The greater weight will put more strain on motors, which leads to higher chance of motor burnout. However, this can be circumvented by quickswaps or cooling the motors with a fan.

Overall, it's important to consider the strengths and weaknesses of the wheels chosen and amount of wheels, since the power lost from using fewer wheels can be compensated by using some traction wheels to create overall high performance or similarly an 8 wheel drive can become more manoeuverable if the drive includes some omniwheels or is entirely made of them. Therefore by considering what the wheels and wheel amount can do for the drivetrain it is possible to create a drivetrain, that doesn't suffer from any major weakness within the scope of what our team is trying
to achieve.

A Look at Gear Ratios



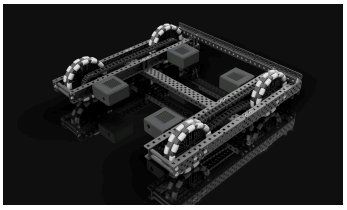
Another factor which contributes to the balance between torque and agility is the gear ratio chosen. If the force applied on the driven gear is further from its pivot compared to the distance from the driving gears pivot then the driven gear will turn with more force but it will turn slower compared to the driving gear. If this coupling is reversed then the driven gear will turn with less force but it will turn faster compared to the driving gear. The gear ratio is often described as either $\frac{\text{driven gear } \Omega}{\text{driving gear } \Omega}$ where Ω is angular velocity or $\frac{\text{driven gear teeth}}{\text{driving gear teeth}}$. Like with wheel type and amount the gear ratio of the drive can be used to further optimise a drive to have greater speed or greater pushing power. On the left is an image [7] of a 2:3 gear coupling. It's also important to consider that wheel size also acts like a gear and so affects torque and speed.

Note

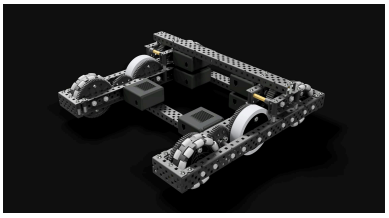
Whilst it's not very commonly used in the context of V5, it's theoretically possible to have a gear transmission to switch between high torque and high speed, which would allow for dynamic gear ratio changing.

Direct Drives

It is also important to consider the viability of direct drives, where power is drawn directly from the motors. This can be useful as it has a lower build complexity allowing for more space and parts for other game scoring subsystems. They can be considered when you want to go with specific rpms such as 200 or 600, in order to have a unique torque-speed ratio compared to the rest of the competition. Below is a comparison between a direct drive and a 6 motor geared



Direct Drive [8]



Geared Drive [2]

Drivetrain Selection Process

The process of selecting a drivetrain is a teams first thoughts when considering the needs for a robot in any given VEX season. The benefits and drawbacks of all types of drivetrain are considered at this point of the design cycle, in the context of the current game of course, many different options are considered as the different parts that VEX provides can be combined in many different ways in order to give each drivetrain a different characteristic as it were.

Considerations for any given drivetrain

At the beginning of any season the needs of the three main aspects of a drivetrain are pitted against each other in the context of the needs of the current game, these three main features are:

- Torque
- Speed
- Size

	Manoeuvrabili	Stability	Ease of build	Size of Wheels	Gear ratio achievable	Total
Differential	3	5	5	4	5	22
Holonomic	4	4	2	4	3	17

For our current purposes and team requirements, a Differential Drive is the most optimal drive to consider using. This is because of their simplicity; which we feel is valuable this early in the season since most other drives suffer from odd motor placements or other restrictions which make designing around them more complex, the power they provide; which is always helpful for pushing tasks, and their stability means we are less likely to tip over mid match. In addition, we have deemed that the downsides of a Differential Drive are either not substantial or can be compensated for by other parts of the drive. For example the lost manoeuverability can be made up for by incorporating omniwheels into our drivetrain.

Choosing Wheel Amount

Choosing the correct amount of wheels is an important consideration when designing the drivetrain. It is an extra way to achieve an optimal balance between torque and agility. The three main considerations for wheel ammount are:

- Traction
- Manoeuvrability
- Complexity

	Manoeuvrability	Traction	Complexity	Gear ratio achievable	Total
4 Wheels	5	2	5	3	15
6 Wheels	4	3	4	3	14
8 Wheels	3	5	3	5	16

From our analysis we decided that an 8 Wheel Drive is our best choice, since it synergises most strongly with a Differential Drive. Combining these two gives us great freedom for gearing, allowing greater control over torque and speed. Additionally, both elements will contribute to a higher power drivetrain. The simplicity of the Differential Drive compensates for the higher complexity of having 8 wheels. Unfortunately, neither of these design elements give us the greatest manoeuverability, which is the greatest problem with our design at this current stage. This can be addressed with the wheels we choose, since having 8 Wheels gives us the most flexibility over combinations of wheel types.

Wheel types

The combination of an 8 Wheel Drive and Differential Drive gives our bot great power, whilst also being relatively simple to build. The main consideration for the combination of wheels, which we choose is that they give our bot greater manoeuverability since that is the main problem with our current solution.

	Manoeuvrability	Traction	Total
All Traction Wheels	1	4	5
Mixture	3	3	6
All Omniwheel	4	1	5

We have decided to go for a combination of 6 Omniwheels and 2 Traction Wheels, positioned third from the front. Having this wheel in this position means we still resist pushing from the sides but allows us to have greater manoeuverability thanks to the 6 omniwheels. Our experience from last season told us that placing the traction wheels there was best for turning and according to *purduesigbots* central placements helps reduce unwanted sideways movement. Another consideration was to place the wheels in a diagonal configuration, but experience told us that middle back was ultimately the superior choice.

Drive Gearing

The final consideration for our drive train will be the gearing we go for. This is the final way in which we can alter our drive train in order to get its torque and speed within desired parameters. We decided to use blue cartridge motors to give us 600 rpm, which we can gear down to 450rpm, with 3:4 gearing, to further increase torque but still have a good speed. We chose 69.85mm (2.75in) wheels to have greater power, so we don't compromise acceleration from the higher rpm. We didn't go for direct drive since we didn't want 200rpm or 600rpm.

CADing the Drivetrain

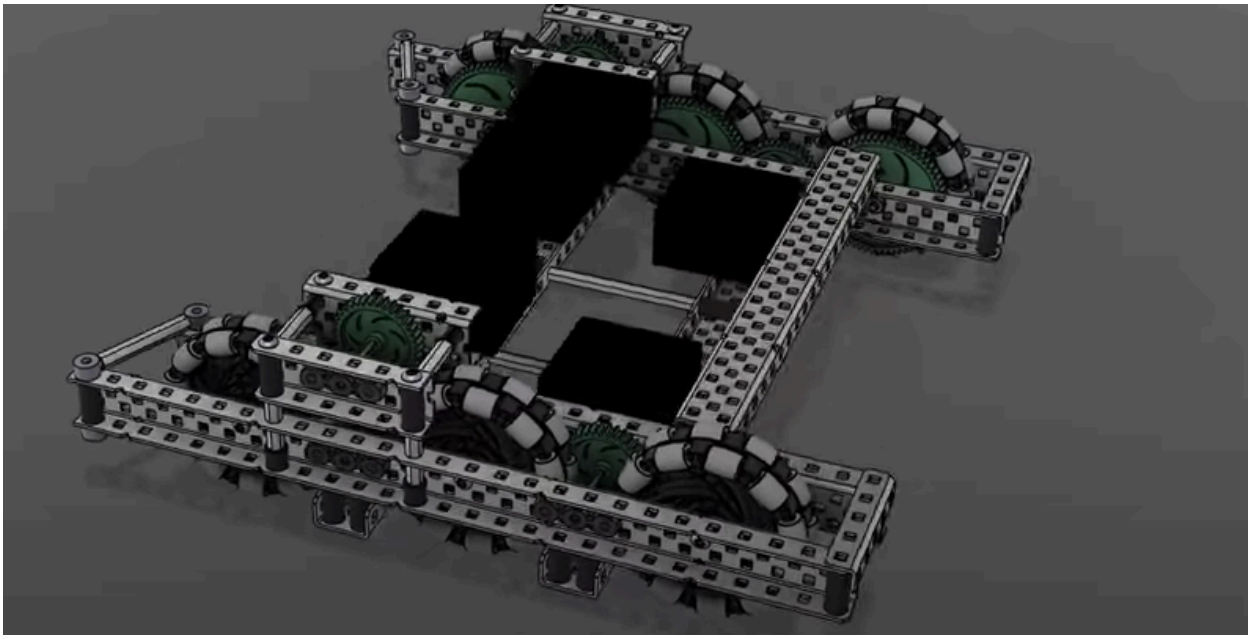
Before building anything in VEX, we must first utilise Computer-Aided Design software (CAD) to visualise what the robot is going to look like and whether or not the ideas we have will work. CAD allows us to use any and all VEX parts that can be bought, and it allows for outlandish ideas to be visualised on a screen before any time or money is wasted on something that will not be viable in competition.

Because of this wonderful resource, we always start any part of the robot that we aim to build in the virtual space, using **Autodesk Fusion 360**, which allows us to import the VEX parts library and take advantage of the range of features it offers.

Our Aim with CAD

Our aim is to have a fully virtual version of our drivetrain so that we can build it in real life with the closest accuracy possible and in the cleanest, most sustainable way. This should help us build an error-free robot and avoid having to rebuild.

> **Quote**
>
> *CAD twice, build once.*



Aan example of a finished CAD model of a drivetrain done by team 29295A [9].

Starting Off

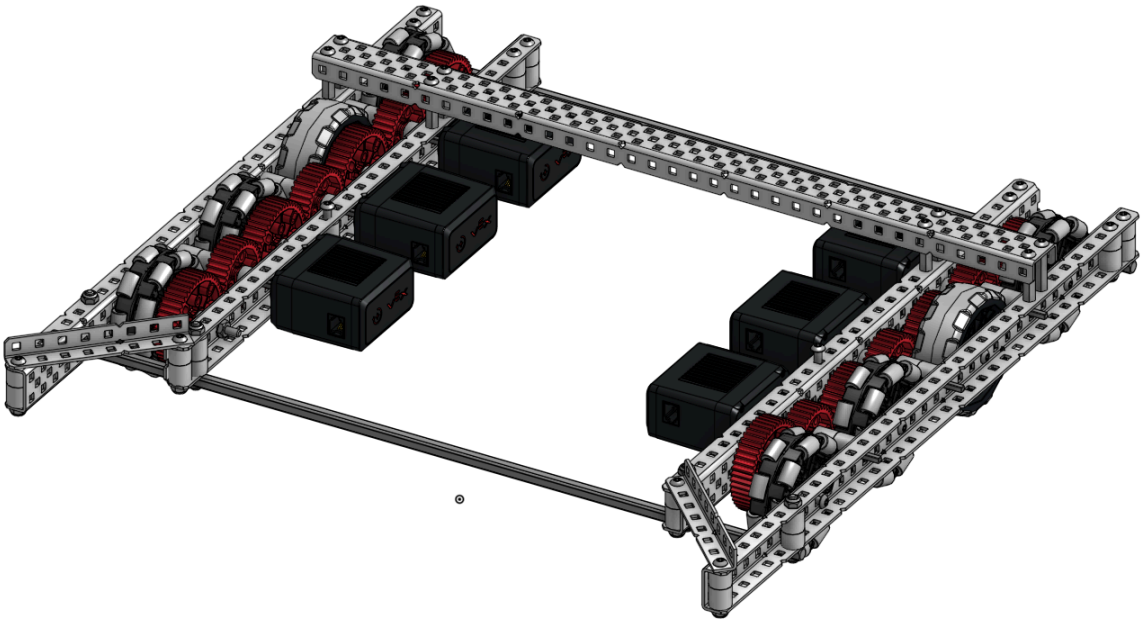
The simplest thing to CAD is probably a drivetrain, as it does not often differ from season to season, being fairly evolved in its entirety. We have decided on the following specifications for our drivetrain:

- 450 RPM at a 3:4 gearing
- 8 wheels
- 6 motors in a differential format

Therefore, it is fairly easy to create a general CAD model that could provide an idea of what to build with the real metal. However, to create a complete replica of what is going to be built, one must spend a little more time on the intricacies of the CAD model, taking time to add each individual spacer and screw to ensure clarity when the time comes to build.

Our CAD Model

A few members of our team are fairly proficient in CAD, especially Fusion 360, due to needs outside of robotics. The process of creating a CAD model exactly to our wants and needs did not take very long.



Here is the CAD model of our drivetrain that we will use to create the drivetrain for our first iteration robot. This CAD model will help us with actual building and hopefully speed up the whole process.

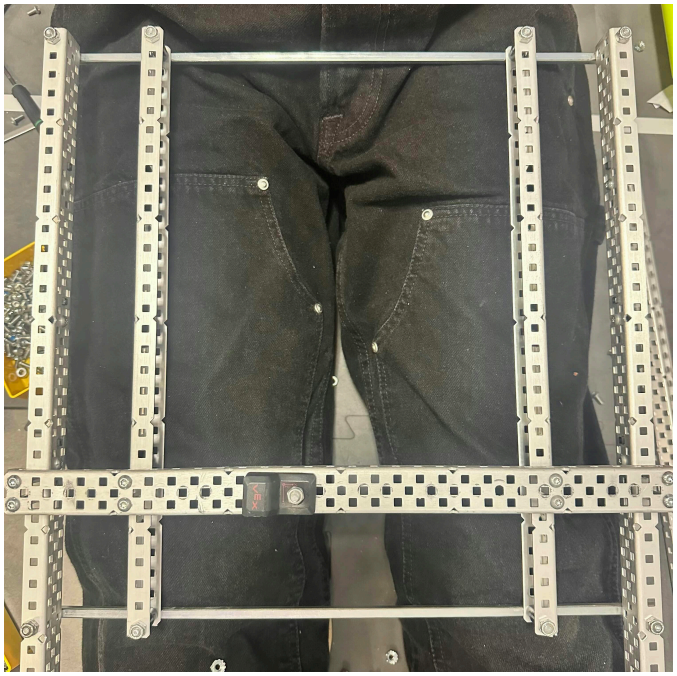
Building the Drivetrain

Step 1: Building the Chassis

The first step when building a drivetrain is to construct the chassis. This consists of the **4 C-channels** that hold the wheels together, plus the bracing that keeps them aligned, allowing for smooth driving with minimal friction.

Unfortunately, due to inconsistencies and tolerances within the screw-C-channel contact, these 4 C-channels are often misaligned, leading to friction and instability. To minimise these inconsistencies, we used a technique called **"squaring"** when building the chassis. This involves attaching the 4 main drive C-channels to 2 other C-channels to create a "box" where all components are either perpendicular or parallel to each other.

We then used an engineer's square to ensure that each angle was **90 degrees**. Afterward, we fitted the braces to ensure that our drivetrain remains straight, reducing friction as much as possible. After fitting these, we removed the temporary braces, leaving us with this:



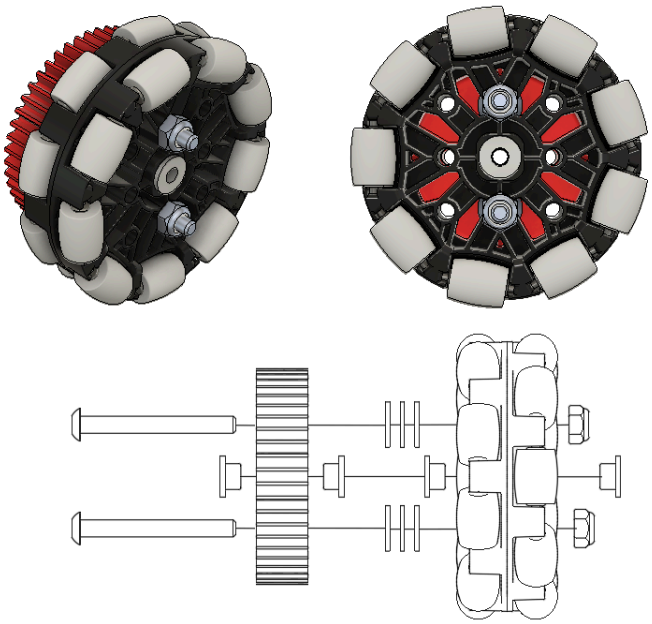
Step 2: Screwjoints and Friction

Now that the drive is braced and sturdy, we can attach the wheels and test the friction both for the wheels themselves and when they are attached to the gears in the motors. To do this, we used the function on the brain that allows us to see how much power each motor is pulling at any given time. Typically, a VEX motor outputs about **0.2W of friction** on its own, meaning this is the target for friction for the entire drivetrain, allowing for peak efficiency and an acceleration curve that can beat anyone on the field.

To attach the wheels with minimum friction, we used something called a **screwjoint**, which utilises the fact that a VEX screw is much sturdier and less prone to bending than an axle. When working with a geared drivetrain, this is particularly helpful as the motors do not need to power the wheels directly.

Step 3: Wheel “Pods”

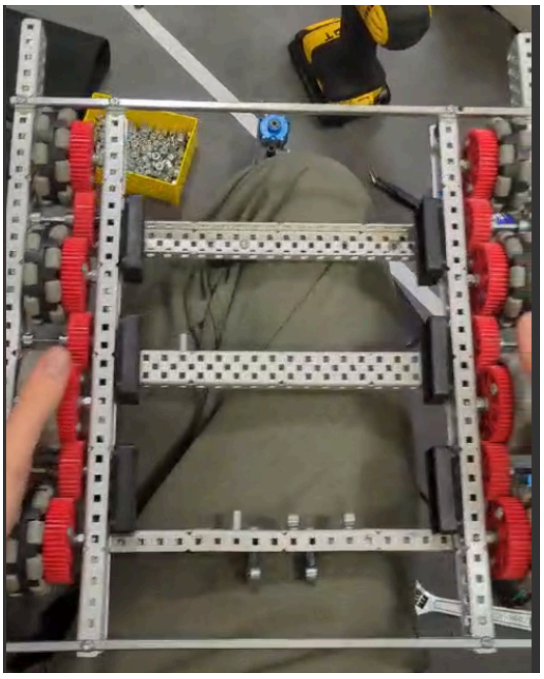
For this to work, we used the **VEX round inserts** in each of our wheel “pods,” which contain a gear and wheel screwed together. On a screwjoint, they would otherwise spin independently.



As seen from this exploded view of the wheel pod, this is how our wheels come together with the gear to create a seamless geared drivetrain with minimal friction, utilising screw joints.

Step 4: Attaching the Wheel Pods

The next step is to attach the wheel pods to the drivetrain so that it becomes driveable.



In this photo, we can see that the wheels are attached in conjunction with the **36t gears** that allow us to have **450 RPM** on our drivetrain. You can also see another technique we use when

building, which is to have **“hotswap” motors**. This means they can be easily switched out in the event of a burnout during competition.

What this entails is that the motor itself, along with the gear cartridge, has been removed from the “cap.” The cap has been solely attached to the robot, and the cartridge and motor are attached with a zip tie or rubber band so they are secure during driver control.

What is a Programming 'Approach'

When tackling any project, thorough planning and thought is required to allow the team to effectively solve it. Defining a standard approach to certain aspects of the challenge can allow certain beneficial procedures to become instinctual, therefore allowing the team to become more efficient.
This is especially prominent when looking at programming, code on large projects can become completely disorganised and hard to read – which is particularly disastrous if you are in a team (where some or most of the members might not intuitively understand the programming language itself).

Common Approaches To Programming

Some aspects of code are typically defined by some form of rule or guideline, examples include:

- Variable naming conventions
- Usage of subprograms or classes
- Choosing a language
- Version or project management
- Program structure
- Safe and secure programming

TL;DR

Standardised programming – through usage of the above – helps to improve:

- Team communication
- Debugging
- Team/project management
- Readability of code
- Enjoyment of writing code

Brainstorming Approaches

Here, we will define what approach(es) we will use when coding, aiming to improve useability, team communication and debugging capabilities.

Language and Software

One of the most important decisions to make is the programming langauge and its accompanying software and libraries. For robotics, the two most common langauges are C++, Python and now, Rust which is making an appearance with *vexide*[1], but the software used alongside it is also very important.

Python with VEXCode V5 or RoboMesh Studio:

- The only current way to code with python is using VEXCode V5 or RoboMesh Studio

Pros	Cons
• Python is exceptionally easy to read and write thanks to its variable type declaration and readable syntax • Devices can be set up using the built in GUIs	• Using GUI for devices can be restricting • There is no allowances for *any* libraries, something that python is known for • Multiple files are not supported, therefore less orginisation through program structure is possible • No choice in editor, therefore no choice for extensions, themes, formatting etc. • Fairly limited device API, less control over devices

C++ with VEXCode or VEXCode Pro

- VEX offers an alternative to python with C++ in the IDEs VEXCode and VEXCode Pro
- VEXCode Pro allows users to use multiple files, along with header files and other c++ functionalities.

Pros	Cons
• C++ offers much more raw functionality as it is a lower level language[2] • GUIs are still available to configure devices	• Still no access to custom libraries • Rigid IDE, no access to extensions to improve workflow • Fairly limited device API, less control over devices • Cannot declare devices both in code and with GUI, must choose one or the other

C/C++ with PROS

[1] An open source Rust runtime for v5 robots
[2] A language that is closer to manipulating raw memory, giving users more control over memory

- *PROS* is an open source development envirionment for VEX founded by Purdue University
- Allows users to write code in C++ or C (C++ is generally pereferred)
- Integrated within existing IDEs, e.g. VSCode or Atom

Pros

- Allows full C/C++ functionality
- Multiple files supported, including header files
- Full template and library functionality
- Hot/Cold linking: only changed code is uploaded, allowing for fast uploads even wirelessly
- Built into existing IDEs, so extensions, formatting themes etc. is all supported and controlled by user
- Significantly improved device API, allowing for fine control over all devices/ components, including serial inputs/ outputs and direct control over radio
- Easy to get started, but with in depth capabilities for niche applications
- Huge ammounts of documentation e.g. *Purdue SIGBots wiki* or *API docs*

Cons

- Can be harder to understand concepts

Rust with Vexide

- *Vexide* is an open source 'no-std'[3] Rust runtime for vex V5
- It is a successor to *pros-rs* , which binds Rust code to the PROS API.
- Allows users to code using Rust, while supplying a CLI to manage projects or interact with a device
- Vexide will be included in a family of vex based applications, such as *vex-v5-qemu* , a CPU level simulation for PROS and Vexide code (includes node-based GUI for device configuration).

⚠ Warning

Vexide is still considered experimental, it has a small base of contributors that are working to make it more and more usable.

[3]'no-std' is a type of package that limits the use of standard libraries. The V5 brain runs without an OS, meaning std libraries impossible to use.

Pros	Cons
• Rust is a langauge designed around memory safetly, including things like variable 'ownership' to avoid memory leaks. • Vexide will eventually work seemlessly with a range of other projects developed by the vexide team.	• Rust is not an easy or intuitive language to learn – we have very limited experience with Rust • Because of it being so new, vexide does not have a stable base of users – meaning less documentation and support • No-std means basic mathematical functions are not accessable[4] • CLI is still limited especially when comparing it to PROS

Variable Naming

Naming conventions are very useful when writing *and* reading code. They can make long, complicated names easy to read; or can help clarify the intent or context around a variable.

Variable requirements:

Most languages (C++ included) require variables to adhere to certain rules:

- Must not start with a digit
- Only alphanumeric values or underscores
- No spaces/whitespaces
- No isolated keywords (e.g. 'if', 'for', 'import' etc.)[5]

Common types of variable naming:

camelCase:

Explanation:

- Variable names start with lowercase
- All new words within the variables start with an uppercase

Example:

```cpp
int dateNow() {
  // Get date
}
bool thisIsAVariable = true;
int currentDate = dateNow();
```

Pros/Cons:

Pros	Cons

[4]There are ways around this
[5]Dependant on language

- Easy to understand multiword variables
- Some programs recognise camelCase, allowing them to display variables with whitespaces
- Satisfying variable 'shape'

- Variable names can become long
- Some words can look confusing e.g 'A' in 'thisIsAVariable' ('A' can be hard to see)

snake_case

Explanation:

- Using underlining to represent whitespace
- Words typically start with lowercase

Example:

```cpp
int date_now() {
  // Get date
}
bool this_is_a_variable = true;
int current_date = date_now();
```

Pros/Cons:

Pros

- Very easy to understand understand multiword variables
- Very easy to see where whitespace is supposed to be

Cons

- Variable names can get very long
- Somewhat difficult to program with due to frequent use of '_'

Note

The difficulty from frequently using '_' can be circumvented by using a program such as Auto HotKey to rebind '_' to something such as "Shift" + "Space". However, this work around may not be worth it for the express purpose of making a naming convention easier.

Boolean 'is' naming

Explanation:

- Start all booleans with 'is'
- Often times subprograms that return booleans start with 'get' ('getIs…')

Example:

```cpp
// (Using camelCase)
bool getIsSaturday() {
  // is it a saturday?
}
bool isSaturday = getIsSaturday();
```

Pros/Cons:

Pros	Cons
• Easy to differentiate between regular procedures and functions that return a boolean value • Works best with camelCase but can also work with other variable naming conventions	• Variable names can get very long • Doesn't help identify between procedures and functions that return other data types

> **Note**
>
> Procedures are defined as subprograms, which don't return **any** value, whilst functions are defined as subprograms, that return a value of a **specified** data type.[6]

Pronouncable names

Explanation:

- Using abreviated words that are pronunciable and easy to make sense of

Example:

```cpp
int getCurrDate() {
  // Get date
}
bool thisIsAVar = true; // variable = var
int currDate = getCurrDate(); // current = curr
```

Pros/Cons:

[6]Languages such as Python do not require such precision in subprogram declaration mitigating the difference between functions and procedures. However, most other languages require procedures to be declared with the **void** keyword to specify they don't return a value.

Pros	Cons
• Drastically shortens variable names	• Not all abbreviations will make sense to everyone • Not using standard english can make documentation and understanding code harder: text autocompletion in modern IDEs means that long names aren't a problem to type.

Unit classification

Explanation:

- Suffixing all variable names (where applicable) with a unit (e.g. rpm, lbs, kgs etc.)
- Using an underscore to seperate unit
- Best used with snake_case

Example:

```cpp
int getMotorSpeed() { // Dont need unit classification for subprograms
  // Get RPM
}
int motorSpeed_rpm = getMotorSpeed(); // suffixed with '_rpm'
```

Pros/Cons:

Pros	Cons
• Units are always known • Conversions can be easier because input/output is documented in the name	• Takes longer to document all variables' types • Increases variable length

Using Subprograms and Classes

- Subprograms are smaller blocks of code that can be run anywhere in user code
- Classes are structures that allow for variables to be 'owned' and can drastically help organisation[7]

Pros	Cons

[7]We may do a deep dive on classes at a later point

- Organised and readable code
- Code can be run multiple times using less lines
- Classes allow for even further organisation
- Classes can help mitigate developer mistakes

- Classes can take time to write
- Variables defined in different files can lead to null pointers during initialising[8]

Version Control

An equally important issue for programming as a discipline is Version Control. This is important because it allows multiple people to work in parallel on the project from anywhere to develop one feature or several and then merge changes into one main branch of the project. In addition, Version Control softwares allow developers to mess with experimental changes without the fear of ruining the project, and it also acts as safety net incase a changeset that enters the main branch ends up breaking the project, since it can easily be rolled back to a stable build. Another advantage of Version Control is that it allows for new and old code to compared for performance over whatever parameters we test - allowing us to easily prove if our new solution is an improvement.

Examples of Version Control Software

There are many different Version Control Software. For this logbook, we are using GitHub - which is built off Git, since it has direct support for Typst and Notebookinator. Other examples include:

- Beanstalk
- PerForce
- Apache Subversion
- Mercurial

Program Structure

As well as using subprograms and classes for organisation, the usage of libraries and splitting the code up into different files helps keep the software for the robot structured and helps prevent accidental changes when working on different parts of the code. Our experience from last year taught us that its best practise to have the code for driver control, match autonomous and autonomous skills on seperate files. In last years game OU this principal was taken further since we had different routes for our left and right side autonomous for matches.

Warning

Global variables have to be initialised in 1 file to avoid Null Pointer Exceptions since the C++ compiler doesn't specify initialisation order.

[8]Basically, the C++ compiler does not have a specified order for intialising variables in different files, meaning if 1 variable depends on an unititialised variable (from another file) it can throw a memory error

Safe Programming

When working in medium level languages such as C++, the necessity for optimising your code and ensuring you handle computer memory and pointers properly increases. Failing to do so effectively, will cause crashes and problems on your machine and at a large scale can even throw the world into *disarray* .
If we use C++ again this year, we must make sure to initialize variables in the right order and ensure we handle memories and pointers properly. General things we can do ensure this are: using smart pointers, using tools such as static analyser; and finally using STL containers.

Smart Pointers

These help reduce the amount of manual memory management. The Microsoft *documentation* states that they are crucial to the RAII or Resource Acquisition Is Initialization programming idiom. The basic premise of RAII is to ensure finite resources are not wasted and so must control their usage and destroyed once its no longer useful - ie when a variable goes out of scope. The usage of Smart Pointers greatly reduces the chance of bugs and memory leaks since memory is automatically deallocated when the resource is no longer used.
Below is a comparison of a raw pointer vs smart pointers.

```cpp
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}

void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

> **Note**
> For the majority of C++ programming, smart pointers aren't necessary to manage especially within the context of VEX programming.

How Do We Decide Our Programming Approach?

While we typically use quantative methods for deciding aspects of our robot, such as the drivetrain (for example using descision matrices); we will avoid using such methods when deciding our programming approach. While we believe taking care when picking a programming approach is *very* important, much of it is still personal preference and can be decided by simply evaluating what fits us best.

Deciding Our Programming Approach

Here we will display the choices we made on the different aspects of the programming approach.

Language and Software

Final Decision

We ultimately decided to code in C++ using PROS

There were many factors involved with this, but experience played a big part in us choosing this: we used Python with VEXCode for the majority of last year, but we found it was very limiting, with the lack of multiple file support and library support; we also used PROS/C++ last year for nationals and worlds, and we found it to be very reliable, even if we didn't utilise the libraries available to the fullest.
I did a significant ammount of research on Rust with vexide, talking to the main developers about the state of the project – I found that while it seemed like a very interesting project, there was not currently enough surrounding support to justify using the more unfamiliar language: Rust.

Variable Naming

Final Decision

We decided to use the following variable naming conventions:
- camelCase
- 'is' boolean variable naming
- Pronounceable names

Variable naming is mostly inconsequential, so we chose what we were most familiar with. It is important to stick with this, as collaborative programming is much easier when the conventions are stuck to.
Arguments can be made to say that stricter rules must be followed, for example this *video* states why long variable names dont matter, and that unit classification is important. However, in VEX, code changes are often made in a hurry, and its easier to focus on a few instinctual rules than to have strict ones that force the programmer to pour over the code after rushed

changes are made to fix formatting – we are also unlikely to go above 1-2 people making changes to the code, so punctuality matters less.

Version Control

As programmers get more experienced, it becomes more and more apparent that version control is a ***must have***; as projects grow larger, mistakes become easier, making the ability to revert code into a stable state is essential.

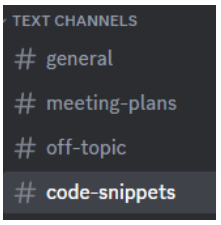
Final Decision

We decided to use the most common form of version control: Git

Git, using *GitHub* , allows us to easily collaborate and use code accross multiple devices, with branching functionality and the ability to manage multiple projects as an organisation[1].

Note

Small changes that we need to implement on another device can be shared in a discord channel, to avoid cluttering the version control.



Small changes are shared in our "code snippets" channel.

Other approaches:

These are approaches that don't require decisions (you either use them or you don't)

- Subprograms and classes will be used where applicable[2]
- We will prioritise uisng multiple files to aid with organisation
- We will adhere to general safe programming rules, especially avoiding unititialised variables, as the compiler doesn't catch them and these issues can be extremely difficult to debug.

Sticking To The Rules

With all these 'rules' it may seem like it will be impossible to follow these in practise; however we can implement procedures to help aid this process. We have taken inspiration from how many companies maintain their huge codebases.

[1]Our organisation: *https://github.com/29457A-SNowflakes*
[2]Classes, in the context of VEX can become more of a nuisance than an aid, so they must be used sparingly

Example

Fun Fact: all of Meta's code (including Instagram, Facebook, Meta VR etc.) is all kept on the same Git codebase, meaning Zuckerburg's original Facebook code is available to *all* meta developers

- Before commiting to the Git, we will make sure that all code follows the formatting rules
- If code changes have to be made in a rush – for example during a competition – commits to the Git will be flaged as bad code (often using the emoji[3]), and will be formatted at a later date.

[3]This is actually an industry standard flag for bad code

The Need For a Mobile Goal Mechanism

Scoring in High Stakes is a complex engineering challenge. This is because, unlike in last years game[1], it requires a dedicated and precise mechanism in order to put rings on the stakes - both mobile and static. Additionally, from analysis of the rules and games played [insert relevant footage] we have realised the importance of being able to manipulate and hold mogos.

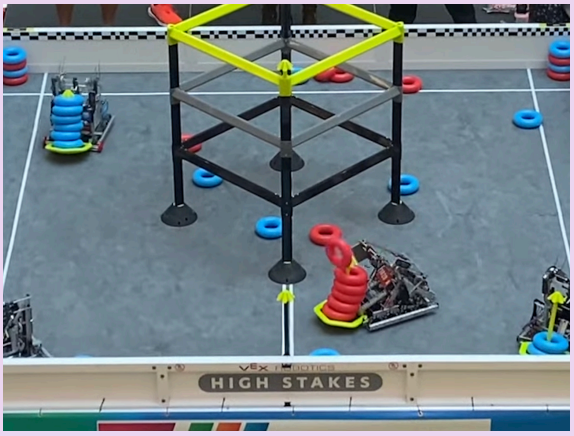
Therefore we need to design a subsystem that can:

- Grab and release mogos
- Securely hold grabbed mogos

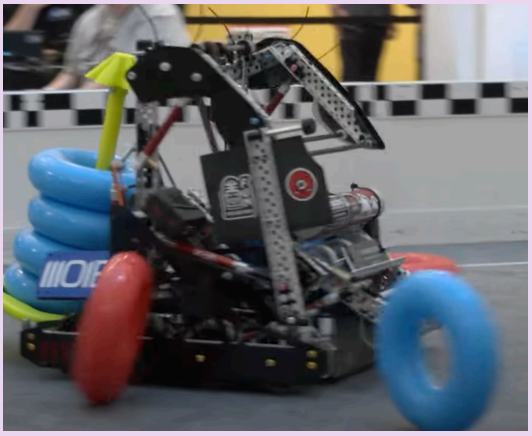
Example

Further Motivation

One case study for the need of mogo manipulation is the early season 'Mall of America' (MOA) signiture event. Throughout this competition, the manipulation of filled or semi-filled mogos was key to winning a match – in most, if not all, it was actually the deciding factor. During our analysis of the key games during MOA, and our analysis and familiarisation of the manual, it became increasingly obvious that leaving this aspect of High Stakes out of our design would be a bad descision.



A snapshot from a video taken during the MOA finals match 1 [10], showing all robots manipulating mogos to their advantage.

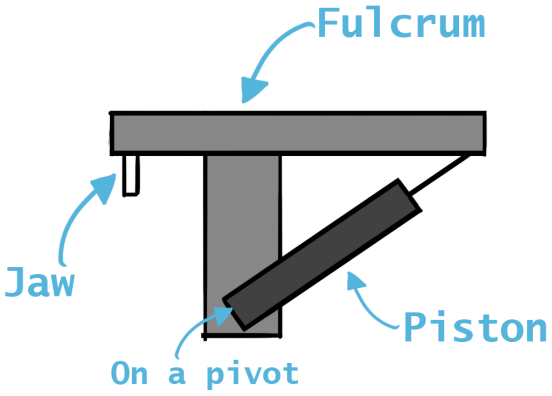


One of the Tournament Champions from MOA, 11101B Barcbots, using a mogo & clamp. (Snapshot from Pits & Parts showcase [11]).

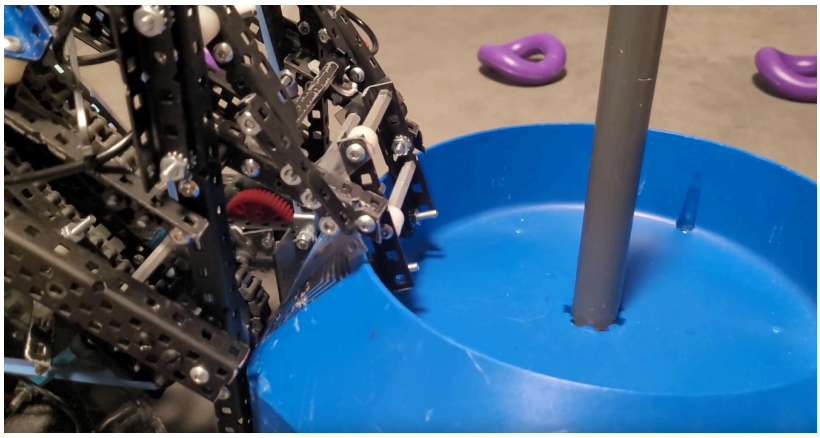
[1] In OU even basic push bots could see relative success at scoring since triballs could be simply pushed to score.

Piston based clamping

The two most common ways to clamp things in VEX is by either using pistons or motors. We will start by looking at piston based clamping. In general, piston based clamping works using a lever applying effort to one side to support a load on the other. Below is a basic concept sketched in photoshop.



Looking at clamp designs for mobile goals, we came across two designs, which shared various similarities. The first design comes from *@6047OSSemicolon* [12] for the game Tipping Point, a game which also features mogos with a similar design. The premise of the design uses a piece of laser cut polycarb, two pistons and two rubber tips to pin the goal to the side of the bot. Notably, HS mogos don't have wide walls and are instead much flatter. However, the concept of this design could be reworked for a High Stakes bot.

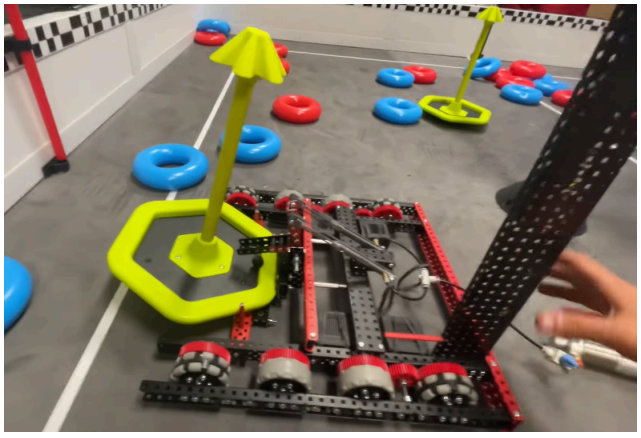


60470S *mogo mech [12]*

An example of a mech from this years game is this prototype by 23851A [13], which uses two pieces of C-Channel and some standoffs to align and angle the mogo. The actual clamp is made up of two cut C-Channels attached together with a gusset. Similarly it uses two pistons, which when extended causes it to pivot and clamp down on the mogo base.

Brainstorming Mogo Mech

2024/09/08



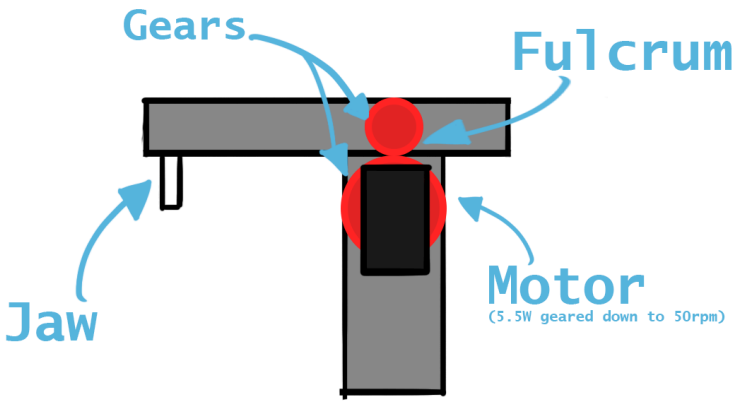
23851A *mogo mech [13]*

Here's a list of advantages and disadvantages for a piston based solution.

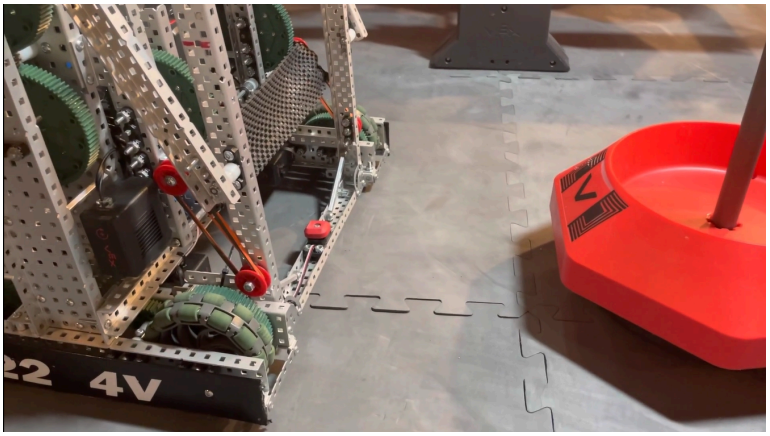
Pros	Cons
• Fast extention time • Doesn't contribute to power budget • No burn out • Better gripping • Lighter mechanism	• Has much more limited actuations than motor based solution • Pneumatic tanks have to be filled before each match

Motor based clamping

An alternative to piston clamping is using motors as a power source. Motor based clamps generally work by using motors as a pivot. Below is a sketched example of how such a thing could work.



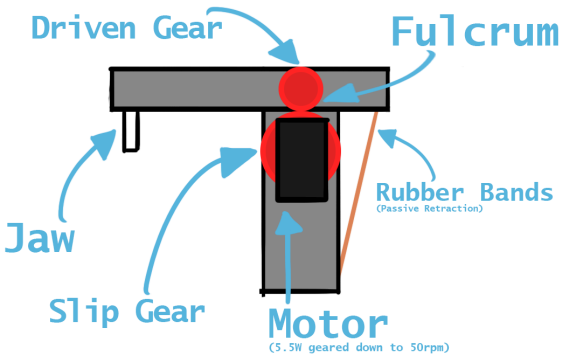
Looking around for inspiration, the best design we came across was made by *@22204V* [14] for Tipping Point, which uses a single motor with rubber bands to tension it, giving the design passive retraction and active extension. However, this design acts more like a scoop than a clamp relying on supporting the clamp from below and the sides rather than pinning it to the side.



Example from the video . [14]

Here's a list of advantages and disadvantages for a motor based solution.

Pros	Cons
• Avoids the sometimes volatile pneumatics • More conistent actuation, independant from previous actuations	• Takes 5.5/11W from 88W limited • Typically weaker than pneumatics • Typically slower actuation than pistons

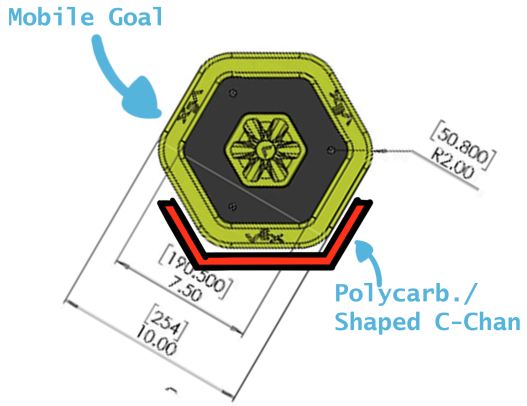


A more experimental concept, which could use slip gears and rubber bands for active grabbing and passive retraction, based on the ideas from the example above.

Passive mechanism

Another alternative to clamping is creating a mechanism to align the mogo to plough it. The benefits of this solution is that it is less complex to implement meaning less chance of failure,

in addition it doesn't use pistons or motors so it doesn't come with either of their drawbacks. Below is an example of how that could look.



Here's a list of pros and cons for this solution.

Pros

- Doesn't contribute to power budget
- Doesn't use up PSI or need pneumatic tanks
- Simple

Cons

- No gripping power
- Mogos can be lost on turns
- Mogos can be stolen by opposing bots
- Little control on positioning for ring mech
- Restrictive: can't go full speed or reverse with mogo

Deciding on a Mogo Mech 2024/09/15

Deciding on the mechanism

To decide on a mogo mech architype we will compare each of the qualities of each design.

- Grip
- Actuation amount
- Complexity
- Manoeuvrability
- Activation speed
- Weight

And additionally the requirements of each design will be considered seperately.

	Grip	Actuation amount	Ease of build	Manoeuvrabil	Activation speed	Weight	Total
Piston	5	3	4	5	4	4	25
Motor	3	4	3	4	3	3	20
Plough	1	5	5	1	5	5	22

Piston requirements:

- Pneumatic tank

Motor requirements:

- 5.5W/11W from power budget

Plough requirements:

- Polycarb or standard metal

Ultimately we have decided to discard the plough as a solution since it drastically reduces our manoeuvrability since we can not drive or turn at high speeds with a mogo or reverse while trying to control it. In addition, not holding the mogo in a fixed position and angle means that

it is much more difficult to score rings on stakes. These factors mean that choosing a plough design will greatly limit our match performance and jeopardise our chances of success. However, the usage of a cut polycarb piece for alignment still seems like a useful mechanism to help us manipulate the mogo and therefore more accurately score rings on mogos.

Final Decision

Between a piston based solution and a motor based solution, we have decided that a piston based solution is the superior choice. This is because in general pistons are able to perform the task better in almost everyway, with the greatest draw of motors being more actuations. However, this is still a non issue since if we have two tanks with our two piston design then we will have at least 20 actuations per game, which from game analysis [insert relevant page/web link here] is much more than needed in match conditions. Therefore, we will iterate on a piston based solution.

The Problem

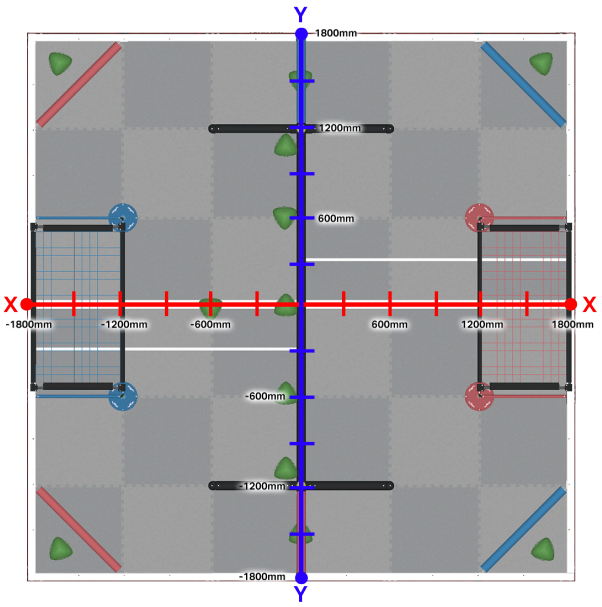
During every match, there is a 15 second autonomous period; there is also a minute-long autonomous skills challenge – which contributes towards our skills score. Ultimately both autonomous challeneges boil down to 1 simple challenge: finding where the robot is.

Finding The Robot

To know where to go, the robot must first know where it is; generally there are a lot of ways to find this – or the robot can simply guess based on rudimentry time inputs.

Where Is This Applicable?

Knowing where the robot is can be useful in a range of scenarios, but of course the main one being during designated autonomous periods (either at the start of a match or during auton skills run). It can also be applicable during driver periods – for example to run macros that require moving the robot.
On the right, there is an image – taken from one of VEX's guides to autonomous [15] from over under – showing the global[1] dimensions of the field, these can be used to program autonomous routines.



Field Dimensions defined uisng the centre as the origion(12'x12') [15]

Other Nescesarry Aspects

To make full autons, we also need to know the state of various other aspects of the robot – including:

- State of pistons
- State of subsystem motors
- (Sometimes) time elapsed from start

Easy solutions

Some of the aforementioned aspects are very easily handled (and therefore will be ommited from the brainstorm process), more specifically, they can be easily handled within the code with minimal extra knowledge.

[1] aka a birdseye view

State of Pistons & Motors

```cpp
pros::adi::Pneumatics thisPiston; // declare piston using in-built 'Pneumatics' struct
thisPiston.toggle(); // toggle piston extension
bool extended = thisPiston.extended; // access state of piston

pros::Motor thisMotor; // declare motor with pros 'Motor' struct
moveMotor1Sec(&thisMotor) // arbitrary function to move motor for 1 second
float motorPos = thisMotor.get_position_relative(); // access state/position of motor
// Other telemetry functions are included for the motor struct.
```

Elapsed Time

```cpp
float time = pros::millis(); // time in milliseconds from start program
```

Aproaching Sensors

As always, it is best to have a plan on how we are going to approach picking which sensors and methods to use. There are many factors involved, but some are much more important than others; for instance, we are very keen on reliability – and, for early season, we want to focus on high scoring 'support' autons – therefore we can bias our search in favour of reliabilty and (for high scoring) speed. Cost is also a big factor, so we must be careful to do thourough research on all possible solutions to avoid misusing our budget.

Brainstorming Sensors

Ultimately, the combinations that are possible in VEX are almost limiteless – for that reason, we will only brainstorm what we feel is likely to be effective or has been proven to work.

> **Note**
>
> **Odometry**
> Odometry may be mentioned throughout the brainstorm process, it is method of absoloute positioning using encoders and/or IMUs. If we move forward with odmetry, we may cover it in seperate page(s).

Internal Sensors

IMEs

Not to be confused with IMU, an IME stands for 'Internal Motor Encoder', essentially, they measure how many rotations the motor has done; they can be reset/tared to 0, and various different measurements can be pulled from them, such as rotation in degrees, radians or full rotations.

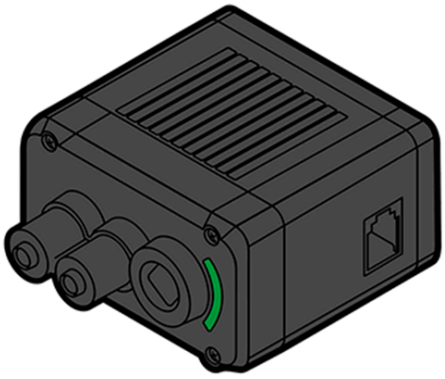


Image of a standard 11W VEX Smart Motor (includes IME) [16]

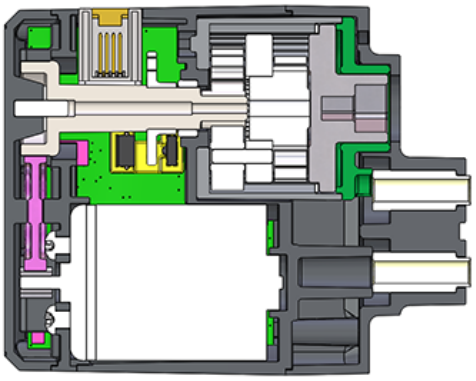


Image of the VEX Smart Motor's (11W) internal structure [16]

IMEs can be used in a range of cases – from finding the state of a motor-based subsystem (see page before), or used in the drivetrain as an input to an odometry program.

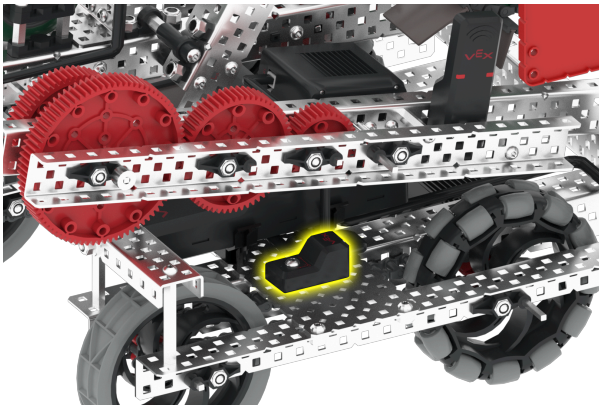
Pros	Cons
• Build into the motors, no extra hardware needed • No extra space needed • Accurate on slow moving systems	• Inaccurate after prolonged use • Encoders attatched to powered input: cannot account for wheel slippage, gear skipping or other linkage-based inconsistensies

External Sensors

IMUs

IMUs stand for 'Inertial Measurement Units,' they are widely used sensors that can collect a range of data:

- Gyroscopic measurements (acceleration in 3 axes)
- Heading (relative heading from calibration – clockwise, degrees or radians)



An inertial sensor installed on a robot (OU Striker/Hero Bot) [15].

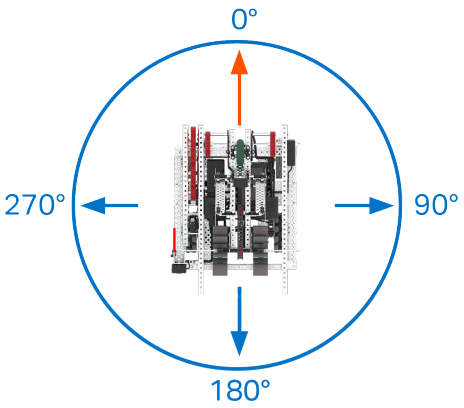


Diagram showing heading directions North, East, South and West in degrees [15].

Ways to use IMUs

- Heading: heading is of course very useful data during autons, they not only allow the robot to 'know' which way it is facing – but can also be used to drastically improve the accuracy of an odometry program.
- INS: 'Inertial Navigation Systems' is another absoloute position system that integrates acceleration into velocity, then into position.

fx Equation

$$A_f(t) = \begin{pmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{pmatrix} \cdot A_b(t),$$

$$P_f(t) = \int_0^t \int_0^t A_f(t)$$

Where:

- $A_b(t)$ is the IMU acceleration vector in form $\begin{pmatrix} x \\ y \end{pmatrix}$
- $A_f(t)$ is the global[1] acceleration transformed from $A_b(t)$ in form $\begin{pmatrix} x \\ y \end{pmatrix}$
- $P_f(t)$ is the current field orientated position of the robot (integrated from $A_f(t)$)

The process of using the IMUs data in IMU odometry (INS)

[1]Field orientated

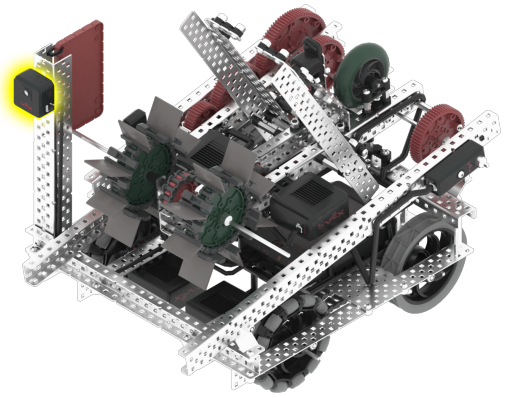
Pros	Cons
• Very accurate heading • Consistent and reliable • Very useful in odometry	• Extremely noisy gyroscopic measurements (INS unusable)

GPS Sensors

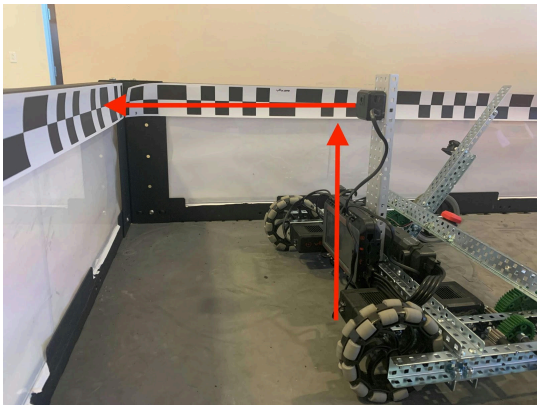
GPS stands for Game Positioning System; the VEX GPS Sensor (not to be confused with sattelite-based GPS) is a small box that has a camera, the camera views a QR Code-like pattern on the field perimeter, and computes the global position based on the pattern, this data is streamed to the brain as coordinates with the centre of the field as the origin (see image on page 69).

⚠ Warning

GPS Must maintain a constant line of sight with the field perimeter, and they must be at the correct height.



A GPS Sensor installed onto the OU Striker/ Hero Bot [15]



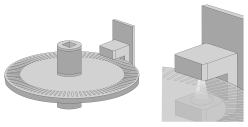
An example of a GPS Sensor with correct positioning [17]

Pros	Cons
• Easy to use (in code) • Somewhat accurate position	• Slow update rate (25Hz in theory – much slower in practice [18]) • Innacturate Heading measurements in practice [19]

Optical Shaft Encoders

> **Note**
> OSEs are now discontinued, in favour of the Rotation Sensors, we however bought some during OU season.

Optical Shaft Encoders (OSEs) are sensors that use 2 ADI ports to stream the current position of a low stregth shaft (inserted into hole in sensor). They have a resolution of 1 degree per tick[2].



The disc contained within the OSE for optically measuring rotation [20]

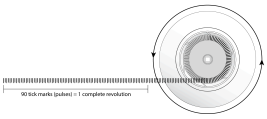
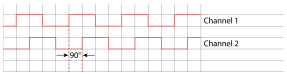
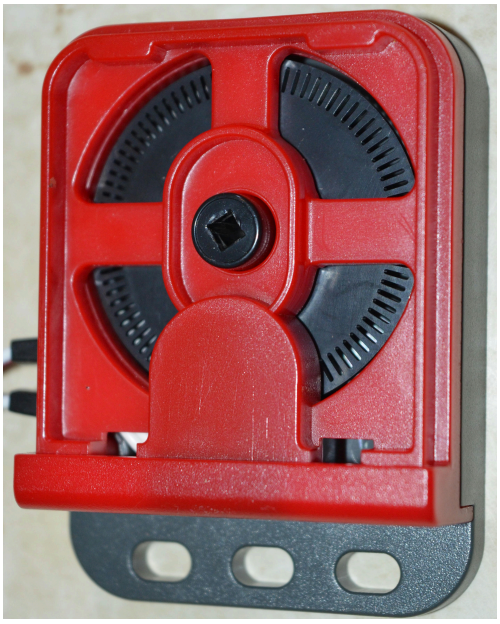


Diagram explaining how ticks are converted to rotations [20]



The signal phase produced from the ticks[3] [20]



The inside of a OSE [20]

Using OSEs

State of Subsystems:

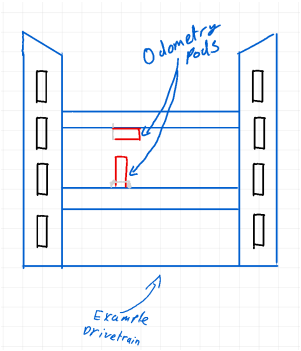
Some subsystems that rely on rotation can be measured using an OSE, either to replace the sometimes inconsistent IME, or to measure rotation on a piston based rotational mechanism.

Pros	Cons
• Can measure rotation on piston based subsystem • Can be used after gearing to counteract gear skipping affects	• IMEs typically suffice for motor based application • Piston based mechanism rarely need to be measured, as they cant be controlled

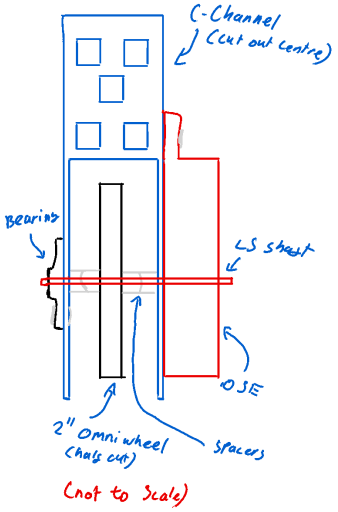
[2]'Ticks' are digital pulses sent when the shaft moves a set ammount
[3]Phase indicates direction

Odomety Pods

Odometry Pods are essentially a wheel attatched to an OSE, they can transform rotations into distance travelled. When they are attatched in a certain orientation, they can measure distance travelled in either direction[4].



Example placement of Odom Pods (perpendicular)



A labeled diagram showing a space efficient odometry pod (requires half-cut omni wheels)

fx Equation

Encoder rotation to distance travelled:

$$D = \frac{\Delta\,\omega \cdot C}{2}$$

Where:

- D: Distance travelled
- $\Delta\,\omega$: Change in rotation of encoder in radians
- C: Circumference of wheel

Note

While other orientations of odom pods exists (using 3 of them), we are not looking at them due to their decreased performance gain (and the fact we would have to buy more OSEs, which is impossible)

Note

To find absoloute position of robot using odometry pods, an odometry algorithm needs to be used – the maths is out of the scope of this research and can be viewed *here* [4].

[4]Works significantly better with an IMU

Pros	Cons
• Very versatile • Relaible • Very Low resolution (1 degree, ~0.023'' on 2.75'' wheels) • Pre-built libraries allow for fast programming with odom pods	• Difficult to program without libraries • Can require half-cutting omni wheels (lower strength) • Uses 2 ADI ports

Rotation Sensors

Rotation Sensors are practically identical to OSEs, with the main difference being size and connection type (smaller and uses smart port)

Image of rotation sensor from the VEX Store [21]

They can be used in the same configuration and context as OSEs, and are often more favourable for odom pods due to their reduced size.

Pros	Cons
• Smaller than OSEs • Lower resolution (0.088 degrees) than OSEs	• We don't have them • They are £43.19 each (we would need at least 2)

Positioning Takeaways

From our research, these sensors can be used in the ways depicted to find position, either relatively from starting point or absoloutely from the centre of the field.

Complimentry Sensors

When implementing autonomous routines, there are a range of complimentry sensors that can be used alongside the positioning sensors to further improve the complexity or reliabilty of the routines.

Limit Switches & Bumper Switches

Limit switches are small sensors that detect whether a small, flexible metal lever is being compressed.
Similarly, Bumper Switches also detect compression – but uses a button-like structure instead.

An image of a limit switch [22]

An image of a bumper switch [22]

Uses:

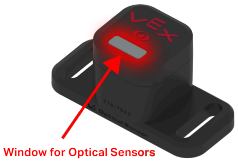
- Stop action once limit is reached, i.e. a lift reaching minimum extension
- Stop robot once contacting wall

Pros	Cons
• Hardware solution *can* be simpler[5]	• Uses 3-wire ADI port • Code based solutions exist and are generally easier

Optical and AI Sensors

Optical Sensors

Optical sensors are small sensors that detect 2 things: distance and colour. Distance will only tell the program roughly hgow far the object is (e.g. near or far) and so cannot be used as a replacement for the distance sensor; however, the colour sensing can be a very useful tool, especially in High Stakes where game objects are colour coded for alliances.



An image of the V5 Optical Sensor and it's light window [23]

A screenshot of the Optical Sensor's menu [23]. Note the distance is not *classified by a value.*

[5]Although most likely not.

Example

Example Optical Sensor Usage
One example of how an optical sensor can be used is **colour sorting**; if the robot has an intake, where rings are picked up from the floor and transporeted onto a mogo, an optical sensor can be used to trigger something that interupts the rings' travel – therefore not scoring it on a mogo.

Note

While this section is on *autonomous sensors*, optical sensors can be used (like the above) during driver control to aid the driver.

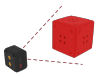
Pros
- Colour sorting mechs can take significant ammount of work off driver
- Colour soting allows for some built in tolerance to autonomous

Cons
- Requires more complex programming
- Can be somewhat inconsistent, especially at longer ranges

AI Sensors

AI Sensors are the more complex, bigger brothers of the opical sensors. Similarly to an oprical sensor, they can detect colours, however, they also have a larger field of view, are caplable of detecting multiple contrasting objects, including their size, distance and angle – and can even be trained to detect certain objects.

Image of the AI vision sensor [24]



An example of how AI vision sensors can detect 3D objects [24]

Pros
- Aligning with objects during autonomous
- More precise than colour and distance sensors

Cons
- Much more complex programming
- Some objects have to be trained, therefore much **more tuning**

Brainsorm Summary

In summary, many (oh so many) sensors can be used in autonomous, to provide the basis of routines, or to aid with preripheral functions.

Going Forward

We can now begin to narrow down the options based on what we as a team want to target, and how we can most efficiently improve out autonomous routines.

Programming Objecives

As covered in our 'Programming Approach' design process, there are certain objectives that a programmer must consider when creating a robot's program. These include:

- Driver/Operator Control
  - Somewhat self explanatory, the inputs from the controller must influence how the robot actuates its systems during the 1:45min drier control period
- Autonomous
  - Autonomous routine(s) must be programmed into the robot to be used during autonomous period
- Ease of setup
  - Before a match, the correct autonomous routines and possibly adjustable values must be set up – the programmer must consider how this is done (e.g. GUI, through code changes etc.)
- Ease of testing
  - During testing (not in competition) the robot must be bleto be easily tested – this could simply mean it is easy to change the key values in code, or could be more complex in the form of a GUI or CLI.
- Maintainability
  - This has been discussed in detail in preious pages, essentially, the programmer must keep ease of use and maintainabilty to a high standard when programming.

Orginisation is key.

> **Quote**
>
> Code is read much more often than it is written.
>
> — Guido Van Rossum, Creator of Python

The Blank Project

After creating a blank project using PROS, it supplies some defualt functions that can be used to create the program.

```cpp
#include "main.h" // -> includes the PROS API, to be used throughout the program.
/**
 * Runs initialization code. This occurs as soon as the program is started.
 *
 * All other competition modes are blocked by initialize; it is recommended
 * to keep execution time for this mode under a few seconds.
 */
void initialize() {}
```

```
9   /**
10   * Runs while the robot is in the disabled state of Field Management System or
11   * the VEX Competition Switch, following either autonomous or opcontrol. When
12   * the robot is enabled, this task will exit.
13   */
14  void disabled() {}
15  /**
16   * Runs after initialize(), and before autonomous when connected to the Field
17   * Management System or the VEX Competition Switch. This is intended for
18   * competition-specific initialization routines, such as an autonomous selector
19   * on the LCD.
20   *
21   * This task will exit when the robot is enabled and autonomous or opcontrol
22   * starts.
23   */
24  void competition_initialize() {}
25  /**
26   * Runs the user autonomous code. This function will be started in its own task
27   * with the default priority and stack size whenever the robot is enabled via
28   * the Field Management System or the VEX Competition Switch in the autonomous
29   * mode. Alternatively, this function may be called in initialize or opcontrol
30   * for non-competition testing purposes.
31   *
32   * If the robot is disabled or communications is lost, the autonomous task
33   * will be stopped. Re-enabling the robot will restart the task, not re-start it
34   * from where it left off.
35   */
36  void autonomous() {}
37  /**
38   * Runs the operator control code. This function will be started in its own task
39   * with the default priority and stack size whenever the robot is enabled via
40   * the Field Management System or the VEX Competition Switch in the operator
41   * control mode.
42   *
43   * If no competition control is connected, this function will run immediately
44   * following initialize().
45   *
46   * If the robot is disabled or communications is lost, the
47   * operator control task will be stopped. Re-enabling the robot will restart the
48   * task, not resume it from where it left off.
49   */
50  void opcontrol() {}
```

Adding Packages

One of the huge advantages of PROS is the abilty to add community developed packages into the code. We as a team will add 2 packages to significantly improve our program.

> **Warning**
>
> **DISCLAIMER**
>
> We as a team *cannot* claim responsibilty for the development of these packages. They are both developed by individual open source communities.
> However, before we decided to use these packages, we all researched and gained understanding of the core concepts that these packagas utilize, this not only drastically improves our capabilty to use them, but also ensures our adherance to the *Student-Centered Policy* .

The Packages:

- LemLib
  - *LemLib* is a library developed by a large open source community – it allows users to employ odometry alogorithims, as well as movement functions; including PID controllers and Boomerang Controllers.
  - We use a modified fork[1] of lemlib (developed by us), allowing us to implement *gain scheduling* to be used later.
- Robodash
  - Robodash implements a user friendly libvgl[2] interface, that allows us to quickly implement consoles and auton selectors.

Adding The Packages in The Project

To add the packages we can use the newly implemented remote depot feature from the PROS CLI.

For LemLib:

```zsh
pros c add-depot LemLib https://raw.githubusercontent.com/LemLib/LemLib/depot/stable.json # adds LemLib stable depot
pros c apply LemLib # applies latest stable version of LemLib
```

For Robodash:

```zsh
pros c add-depot robodash https://raw.githubusercontent.com/unwieldycat/robodash/depot/stable.json
```

File structure

Our program will be split up into different sections for the sake of orginisation:

[1] A branch, in this case not public
[2] The GUI library that PROS implements for the brain's screen

```
|
|-- [vscode files/folders]
|-- include
|   |-- [defualt pros header folders]
|   |-- lemlib -> lemlib header files
|   |-- robodash -> robodash header files
|   |-- usr -> created by us
|   |   |-- autons.h -> header files for all auton routines
|   |   |-- robot.h -> contains 'robot' class
|   |   |-- utils.h -> header file for utility functions
|-- src
|   |-- robotConfig
|   |   |-- globals.cpp -> where all global values are initialized
|   |   |-- intis.cpp -> initializer functions for robot
|   |   |-- tuning.cpp -> where tuning loops and functions are contained
|   |-- autons.cpp -> where auton routines are defined
|   |-- main.cpp -> all competition functions (opcontrol(), autonomous(), etc.)
|   |-- utils.cpp -> where util functions are declared
```

Class Based Program (robot.h)

To improve maintainabilty and orginisation, we will be implementing a class based program; this involves us using a header file that contains all functions (contained in subclasses) to operate the robot and its subystems. These functions can then be called and used in any other file.

robot.h:

```h
// includes ---
#include "api.h"
#include "lemlib/api.hpp"
#include "lemlib/chassis/chassis.hpp"
#include "lemlib/chassis/trackingWheel.hpp"
#include "pros/adi.hpp"
#include "pros/imu.hpp"
#include "pros/motor_group.hpp"
#include "pros/motors.hpp"
#include "robodash/api.h"
#include "robodash/core.h"
#include "robodash/views/console.hpp"
#include "robodash/views/selector.hpp"
// ---


namespace Types {
    // specifies how intake is actuated
    enum IntakeActionType {
        BOTH,
        FIRST,
        SECOND
```

```
    };
}
using namespace Types;
class Robot {
    public:
        class Auton { // contains autons
            public:
                class Tuning {  // contains functions to tune autonomous (PID) values
                    public:
                        static rd::Console AutonTuning;
                        static const void TuningLogicLoop();
                        static lemlib::ControllerSettings latController;
                        static lemlib::ControllerSettings angularController;

                        //TODO (non-urgent)
                        static const void DriveCurveTuning();
                        static lemlib::ExpoDriveCurve driveCurveLateral;
                        static lemlib::ExpoDriveCurve driveCurveAngular;
                        //--

                        static rd::Selector Tuningselector; // selector for test autons
                };
            static rd::Selector autonSelectorMain; // selector for game autons
        };
        class Motors { // contains all motor-based variables
            public:
                static const lemlib::Drivetrain drivetrain;
                static pros::Motor Intake1st;
                static pros::Motor Intake2nd;
                static pros::MotorGroup fullIntake;
                static bool DTDirection; // direction of 'forward' for drivetrain (not implemented)
            protected:
                static pros::MotorGroup leftMotors;
                static pros::MotorGroup rightMotors;
        };
        class Sensors { // class for sensors
            public:
                static lemlib::OdomSensors sensors;
                static pros::IMU imu;
            protected:
                static lemlib::TrackingWheel verticalTracking;
                static lemlib::TrackingWheel horizontalTracking;
        };
        class Pneumatics { // class for pneumatics
            public:
                static pros::adi::Pneumatics mogoMech;
                static pros::adi::Pneumatics intakeLifter;
                static pros::adi::Pneumatics doinker;
        };
        class Actions { // contains macros/actions for robot
            public:
                static void setMogoFor (bool extended, float time, bool async=true);

```

```
76
77              static void setIntake(int direction, IntakeActionType stage);
78              static void runIntakeFor(int direction, IntakeActionType stage, float time, bool async = true);
79
80
81              static void setIntakeLifterFor(bool extended, float time, bool async=true);
82
83              static void jiggle(float time); // jiggles robot (may be useful?)
84
85          };
86          class Init { // intializer functions
87              public:
88                  static void initAll();
89              protected:
90                  static void initPIDs();
91                  static void initDriveCurves();
92          };
93          class Logging {/* WIP */};
94          static pros::Controller master; // controller
95          static lemlib::Chassis chassis; // main chassis
};
```

Declaring global values (globals.cpp)

To avoid null pointers, global variables must be decalred in one file:

```cpp
#include "main.h"
#include "lemlib/api.hpp"
#include "pros/abstract_motor.hpp"
#include "pros/adi.hpp"
#include "pros/motor_group.hpp"
#include "usr/robot.h"
#include "robodash/api.h"
#include "usr/autons.h"
using namespace pros;
using namespace lemlib;

pros::Controller Robot::master (pros::E_CONTROLLER_MASTER);
rd::Console Robot::Auton::Tuning::AutonTuning ("PID Tuner");

lemlib::ControllerSettings Robot::Auton::Tuning::latController (10, // proportional gain (kP)
                                0, // integral gain (kI)
                                3, // derivative gain (kD)
                                3, // anti windup
                                1, // small error range, in inches
                                100, // small error range timeout, in milliseconds
                                3, // large error range, in inches
                                500, // large error range timeout, in milliseconds
                                20 // maximum acceleration (slew)
);
lemlib::ControllerSettings Robot::Auton::Tuning::angularController(2, // proportional gain (kP)
```

```
                                0, // integral gain (kI)
                                10, // derivative gain (kD)
                                3, // anti windup
                                1, // small error range, in degrees
                                100, // small error range timeout, in milliseconds
                                3, // large error range, in degrees
                                500, // large error range timeout, in milliseconds
                                0 // maximum acceleration (slew)
);
rd::Selector Robot::Auton::Tuning::Tuningselector (
    {
        {"Move Forwards 24''", Autons::Testers::forward24},
        {"Turn 90deg right", Autons::Testers::turn90},
        {"Boomerang 24 24 90", Autons::Testers::boomerang_24_24_90},
        {"Swing 90deg right", Autons::Testers::swing90},
        {"Cricle (exp.)", Autons::Testers::circle}
    }
);

Motor Robot::Motors::Intake1st (-11, v5::MotorGears::blue, v5::MotorUnits::rotations);
Motor Robot::Motors::Intake2nd (9, v5::MotorGears::blue, v5::MotorUnits::rotations);
MotorGroup Robot::Motors::fullIntake ({-11, 9}, v5::MotorGears::blue, v5::MotorEncoderUnits::rotations);

bool Robot::Motors::DTDirection = true; // Defualt direction is forwards

//* Left motors on ports 10, 9, 8; Rights on 1, 2, 3; Using blue cartridges
MotorGroup Robot::Motors::leftMotors (
    {-1, -2, -10},
    MotorGears::blue,
    MotorUnits::degrees
);
MotorGroup Robot::Motors::rightMotors (
    {14, 20, 18},
    MotorGears::blue,
    MotorUnits::degrees
);
ExpoDriveCurve Robot::Auton::Tuning::driveCurveLateral (4, 6, 1.004);
ExpoDriveCurve Robot::Auton::Tuning::driveCurveAngular (4, 6, 1.016);
//* Drivetrain with track width 13.1'', Geard (down) for 450rpm, using horizontal drift of 8 due to traction wheel
usage
//TODO Measure dimensions
const Drivetrain Robot::Motors::drivetrain(
    &leftMotors,
    &rightMotors,
    13.1,
    Omniwheel::NEW_275,
    450,
    6
);

adi::Pneumatics Robot::Pneumatics::mogoMech {'E', false};
adi::Pneumatics Robot::Pneumatics::intakeLifter {'G', true}; // not built
```

```
adi::Pneumatics Robot::Pneumatics::doinker {'h', false}; // not built

Imu Robot::Sensors::imu (21); // IMU on port ?

adi::Encoder vertEncoder('A', 'B');
adi::Encoder horiEncoder('C', 'D');

//TODO measure distances
TrackingWheel Robot::Sensors::verticalTracking (&vertEncoder, Omniwheel::NEW_275_HALF, 0);
TrackingWheel Robot::Sensors::horizontalTracking (&horiEncoder, Omniwheel::NEW_275_HALF, 0);

OdomSensors Robot::Sensors::sensors (&verticalTracking, nullptr, &horizontalTracking, nullptr, &imu);

lemlib::Chassis Robot::chassis (
    Motors::drivetrain,
    Auton::Tuning::latController,
    Auton::Tuning::angularController,
    Sensors::sensors,
    &Auton::Tuning::driveCurveLateral,
    &Auton::Tuning::driveCurveAngular
);
```

Utils

Utils are useful functions that can be used throughout the program to manage certain aspects.

utils.h:

h

```
#include <string>
namespace utils { // namespaces for utils
    void save_value(std::string name, float value); // saves float to file on sd card
    float load_value (std::string name); // loads float from file on sd card
}
```

utils.cpp (declaring):

cpp

```
#include "usr/utils.h"
#include "fmt/core.h"
#include "main.h"
#include <cstdio>
#include <cstring>

void utils::save_value(std::string name, float value) {

    std::string fullPath = fmt::format("/usd/{}.txt", name);

    FILE* value_file = fopen(fullPath.c_str(), "w");
    fputs(std::to_string(value).c_str(), value_file);

```

```
    fclose(value_file);
}

float utils::load_value(std::string name) {

    std::string fullPath = fmt::format("/usd/{}.txt", name);

    FILE* value_file = fopen(fullPath.c_str(), "r");

    if (value_file == nullptr) {
        return 0;
    }
    char buf[50];
    fread(buf, 1, 50, value_file);
    fclose(value_file);

    return std::stof(buf);
}
```

Autons

> **Note**
>
> As of right now, autons are WIP – only test autons are implemented.

autons.h:

```h
namespace Autons {
    class Testers {
        public:
            static void forward24(); // moves robot forward 24 inches
            static void turn90(); // turns 90deg right
            static void swing90(); // 'swings' 90deg right
            static void boomerang_24_24_90(); // uses boomerang controller to go to (24, 24) inches at heading 90deg
            static void circle(); // (in theory) moves bot in full circle.
    };
    // WIP
    void supportTouchLadder();
    void support();
    void rush();
    // TODO: more?
}
```

autons.cpp

```cpp
#include "lemlib/chassis/chassis.hpp"
#include "usr/robot.h"
```

```
#include "api.h"
#include "lemlib/api.hpp"
#include "usr/autons.h"
using namespace pros;

//* For Tuning
void calibrate () {
    Robot::chassis.calibrate();
    pros::delay(1000);
    Robot::chassis.setPose({0,0,0});
}
void Autons::Testers::forward24() {
    calibrate();
    Robot::chassis.moveToPoint(0, 24, 1000);
    Robot::chassis.waitUntilDone();
}
void Autons::Testers::turn90() {
    calibrate();
    Robot::chassis.turnToHeading(90, 1000);
    Robot::chassis.waitUntilDone();
}
void Autons::Testers::boomerang_24_24_90 () {
    calibrate();
    Robot::chassis.moveToPose(24, 24, 90, 1250, {.lead=0.81});
    Robot::chassis.waitUntilDone();
}
void Autons::Testers::swing90() {
    calibrate();
    Robot::chassis.swingToHeading(90, lemlib::DriveSide::RIGHT, 500);
    Robot::chassis.waitUntilDone();
}
void Autons::Testers::circle() {
    calibrate();
    Robot::chassis.moveToPose(0, 24, 270, 4000, {.lead=0.6});
    Robot::master.rumble(".");
    Robot::chassis.moveToPose(-24, 0, 180, 1000, {.lead=0.81});
    Robot::chassis.moveToPose(0, -24, 90, 1000, {.lead=0.81});
    Robot::chassis.moveToPose(24, 24, 0, 1000, {.lead=0.81});
    Robot::chassis.moveToPose(0, 24, 90, 1000, {.lead=0.81});
    Robot::chassis.waitUntilDone();
}
lemlib::Chassis* chassis = &Robot::chassis;
//* Game autons
void Autons::support() {
    chassis->setPose({-50, 41, 305});
    chassis->moveToPoint(-32, 28, 1000, {.forwards=false, .maxSpeed=110, .minSpeed=40});
    chassis->waitUntilDone();
    Robot::Pneumatics::mogoMech.set_value(true);
    delay(40);
    chassis->turnToPoint(-24, 48, 500);
    Robot::Actions::setIntake(1, Types::BOTH);
    chassis->moveToPoint(-26, 42, 1000, {.maxSpeed=80});
    chassis->waitUntilDone();
```

```
    delay(100);
    chassis->turnToPoint(-3, 50.5, 400);
    chassis->moveToPose(-3, 52, 90, 1000, {.lead=0.9, .maxSpeed=80});
    chassis->waitUntilDone();
    delay(100);
    chassis->moveToPose(-12, 55, 135, 800, {.forwards=false, .lead=0.5, .maxSpeed=70});
    chassis->moveToPoint(-7, 48, 1000, {.maxSpeed=90});
}
```

Robot Actions

robotActions.cpp:

```cpp
#include "lemlib/pose.hpp"
#include "main.h"
#include "lemlib/api.hpp"
#include "pros/rtos.hpp"
#include "usr/robot.h"

using namespace pros;
using namespace lemlib;

void Robot::Actions::setMogoFor(bool extended, float time, bool async) {
    if (async) {
        Task task ([=] {setMogoFor(extended, time, false);});
        delay(10);
        return;
    }
    Pneumatics::mogoMech.set_value(extended);
    delay(time);
    Pneumatics::mogoMech.toggle();
}
void Robot::Actions::setIntake(int direction, IntakeActionType stage) {
    int volts = 12000 * direction;
    if (stage==IntakeActionType::FIRST || stage == IntakeActionType::BOTH) {
        Motors::Intake1st.move_voltage(volts);
    }
    if (stage==IntakeActionType::SECOND || stage == IntakeActionType::BOTH) {
        Motors::Intake2nd.move_voltage(volts);
    }
}
void Robot::Actions::runIntakeFor(int direction, IntakeActionType stage, float time, bool async) {
    if (async) {
        Task t ([=] {runIntakeFor(direction, stage, time, false);});
```

```
        delay(10);
        return;
    }
    setIntake(direction, stage);
    delay(time);
    setIntake(0, stage);
}
void Robot::Actions::setIntakeLifterFor(bool extended, float time, bool async){
    if (async) {
        Task t ([=] {setIntakeLifterFor(extended, time, false);});
        delay(10);
        return;
    }
    Pneumatics::intakeLifter.set_value(extended);
    delay(time);
    Pneumatics::intakeLifter.toggle();
}
void Robot::Actions::jiggle(float time) {
    float start = millis();
    while (millis()-start < time) {
        Pose jigglePoint1 = Pose {chassis.getPose().x-(float)cos(chassis.getPose(true).theta)*2, chassis.getPose().y-(float)sin(chassis.getPose(true).theta)*2};
        Pose jigglePoint2 = Pose {chassis.getPose().x+(float)cos(chassis.getPose(true).theta)*2, chassis.getPose().y+(float)sin(chassis.getPose(true).theta)*2};
        Pose start = chassis.getPose();
        Robot::chassis.moveToPoint(jigglePoint1.x, jigglePoint1.y, 50, {.minSpeed=100});
        Robot::chassis.moveToPoint(jigglePoint2.x, jigglePoint2.y, 50, {.minSpeed=100});
        Robot::chassis.moveToPoint(start.x, start.y, 50, {.minSpeed=100});
        Robot::chassis.waitUntilDone();
        delay(200);
    }
}
```

Tuning

tuning.cpp

```cpp
#include "lemlib/chassis/chassis.hpp"
#include "pros/imu.hpp"
#include "pros/misc.h"
#include "pros/misc.hpp"
#include "pros/motor_group.hpp"
#include "robodash/views/console.hpp"
#include "robodash/views/selector.hpp"
#include "usr/robot.h"
#include <cstdio>
#include <cstring>
#include <string>
#include "usr/utils.h"
using namespace utils;

```

```
15
16  const void Robot::Auton::Tuning::TuningLogicLoop() {
17      AutonTuning.focus();
18      std::cout << "here";
19      float AngularP = angularController.kP;
20      float AngularI = angularController.kI;
21      float AngularD = angularController.kD;
22      float LatP = latController.kP;
23      float LatI = latController.kI;
24      float LatD = latController.kD;
25      float step = 0.1;
26      int index = 0;
27      int minIndex = 0;
28      int maxIndex = 6;
29      while (!master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_R2)) {
30          pros::delay(20);
31          AutonTuning.clear();
32
33          latController.kP = LatP;
34          latController.kI = LatI;
35          latController.kD = LatD;
36
37          angularController.kP = AngularP;
38          angularController.kI = AngularI;
39          angularController.kD = AngularD;
40
41          chassis.changeAngularP(AngularP);
42          chassis.changeAngularI(AngularI);
43          chassis.changeAngularD(AngularD);
44
45          chassis.changeLatP(LatP);
46          chassis.changeLatI(LatI);
47          chassis.changeLatD(LatD);
48
49          AutonTuning.println("AUTON TUNING");
50          AutonTuning.println("Press B to run Auton, R2 to quit, Arrows to navigate/change values, X to save values");
51
52          AutonTuning.printf("Angular P: %f %s\n", AngularP, index==0 ? "<<" : "");
53          AutonTuning.printf("Angular I: %f %s\n", AngularI, index==1 ? "<<" : "");
54          AutonTuning.printf("Angular D: %f %s\n", AngularD, index==2 ? "<<" : "");
55          AutonTuning.printf("Lateral P: %f %s\n", LatP, index==3 ? "<<" : "");
56          AutonTuning.printf("Lateral I: %f %s\n", LatI, index==4 ? "<<" : "");
57          AutonTuning.printf("Lateral D: %f %s\n", LatD, index==5 ? "<<" : "");
58          AutonTuning.printf("Change step: %f %s\n", step, index==6 ? "<<" : "");
59
60          if (master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_DOWN)) index +=1;
61          if (master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_UP)) index -= 1;
62          if (index > maxIndex) index = minIndex;
63          if (index < minIndex) index = maxIndex;
64
65          if (master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_LEFT)) {
66              switch (index) {
67
```

```
            case 0:
              AngularP = ((int)(AngularP*100))-((int)(step*100))/100;
              break;
            case 1:
              AngularI = ((int)(AngularI*100))-((int)(step*100))/100;
              break;
            case 2:
              AngularD = ((int)(AngularD*100))-((int)(step*100))/100;
              break;
            case 3:
              LatP = ((int)(LatP*100))-((int)(step*100))/100;
              break;
            case 4:
              LatI = ((int)(LatI*100))-((int)(step*100))/100;
              break;
            case 5:
              LatD = ((int)(LatD*100))-((int)(step*100))/100;
              break;
            case 6:
              step = (((int)(step*100))-1)/100;
              break;
          }
        } else if (master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_RIGHT)) {
          switch (index) {
            case 0:
              AngularP = ((int)(AngularP*100))+((int)(step*100))/100;
              break;
            case 1:
              AngularI = ((int)(AngularI*100))+((int)(step*100))/100;
              break;
            case 2:
              AngularD = ((int)(AngularD*100))+((int)(step*100))/100;
              break;
            case 3:
              LatP = ((int)(LatP*100))+((int)(step*100))/100;
              break;
            case 4:
              LatI = ((int)(LatI*100))+((int)(step*100))/100;
              break;
            case 5:
              LatD = ((int)(LatD*100))+((int)(step*100))/100;
              break;
            case 6:
              step = (((int)(step*100))+1)/100;
              break;
          }
        }
        chassis.changeAngularP(AngularP);
        chassis.changeAngularI(AngularI);
        chassis.changeAngularD(AngularD);
        chassis.changeLatP(LatP);
        chassis.changeLatI(LatI);
        chassis.changeLatD(LatD);
```

```
121
122         if (master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_B)){
123             master.clear_line(1);
124             AutonTuning.printf("Running auton...");
125             master.print(1, 1, "Running auton");
126             Tuningselector.run_auton();
127         }else {
128             AutonTuning.printf("\n");
129             master.clear_line(1);
130             master.print(1, 1, "Done");
131         }
132
133         if (master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_X)) {
134             AutonTuning.println("Saving");
135             save_value("angP", AngularP);
136             save_value("angI", AngularI);
137             save_value("angD", AngularD);
138             save_value("latP", LatP);
139             save_value("latI", LatI);
140             save_value("latD", LatD);
141         }
142     }
143     Tuningselector.focus();
}
```

Implementing the Primary Code

With all the complimentary functions built into the robot class, I can now imlement the main functions in the main.cpp file – this will include calling the selected autonomous routine and collecting and using user inputs during operator control.

Initialization

We had the foresight to declare initialize functions globally in *inits.cpp*, therefore we can just call '*initAll()*' from the primary code:

```cpp
1  void initialize() {
2      Robot::Inits::initAll(); // runs initialization function
3  }
```

Operator Control

> **Note**
>
> Most functionality within opcontrol is contained in a while loop, this allows the program to constantly update values and assess states.

Moving the drive

Final Decision

For driver control, I (driver and programmer) have decided to use a control scheme called **single stick curvature drive**, it is very similar to single stick arcade, but the turning inputs are scaled differently depending on throttle input (essentially creating an arc with the turning) – I chose this because arcade is typically limited in terms of precise control, and tank, while it technically is easier to provide ultra precise turn circles, is not very intuitive and would be very difficult to learn in such a short timeframe. (We will evaluate this desision as the season progresses.)

Fortunately, LemLib condenses the somewhat long mathematical function into 1 function call:

```cpp
void opcontrol() {
    while (true){
        pros::delay(20) // 20ms delay so brain doesn't crash
        int leftX = Robot::master.get_analog(pros::E_CONTROLLER_ANALOG_LEFT_X);
        int leftY = Robot::master.get_analog(pros::E_CONTROLLER_ANALOG_LEFT_Y);
        Robot::chassis.curvature(leftY, leftX); // lemlib's chassis contains curvature function
        // ...
    }
}
```

Actuating Subsystems

To actuate the other subystems, if statements can be used to access the state of buttons on the controller:

```cpp
void opcontrol() {
    while (true){
        // ...
        if (Robot::master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_B)) {
            Robot::Pneumatics::mogoMech.toggle(); // toggles mogo on B press
        }
        if (Robot::master.get_digital(pros::E_CONTROLLER_DIGITAL_L2)) {
            Robot::Actions::setIntake(-1, Types::BOTH); // sets intake backwards while L2 is pressed
        } else if (Robot::master.get_digital(pros::E_CONTROLLER_DIGITAL_R2)) {
            Robot::Actions::setIntake(1, Types::BOTH); // sets intake forwards while R2 is pressed
        } else Robot::Actions::setIntake(0, Types::BOTH); // stops intake
    }
}
```

Extra features

With plenty of buttons spare, extra features can be added – for exapmple, we plan on having an arm/sweeper that can be added with another button press.
We can also use button presses to actuate autonomous for testing (we should of course avoid during competition). Instead of a single button press, however, it should be a sequence, so we avoid running it during driver practice.

Designed by: Daniel Dew
Witnessed by: Daniel daSilva

Example

An example button sequence would be:
*Down arrow held down **AND** up arrow new press*

Adding to code:

```cpp
void opcontrol() {
    while (true) {
        // ...
        if (
            !pros::competition::is_connected() && // if field control (at a competition) is NOT connected
            Robot::master.get_digital(pros::E_CONTROLLER_DIGITAL_DOWN) && // if down arrow is held down
            Robot::master.get_digital_new_press(pros::E_CONTROLLER_DIGITAL_UP) // if up arrow is newly pressed
        ) autonomous() // starts autonomous
    }
}
```

Starting Autonomous

Robodash fortunately handles picking autonomous via a GUI, so the autonomous function must just run the seleced auton like so:

```cpp
void autonomous() {
    Robot::Auton::autonSelectorMain.run_auton(); // runs selected auton
}
```

Note On Testing & Iteration

Normally after implementing a large feature, such as this code – we would implement a testing campaign to verify the results of the code. However, we managed to very quickly verify that the code worked by simply running it and verifying all features worked.
Throughout the season, we will continue to update the code with autons, subsystems and additional features as we see fit.

Quote

> Every great developer you know got there by solving problems that they were unqualified to solve until they actually did it.
>
> — Patrick Mckenzie, Sofware Enginner

Glossary

'Auton(s)'

An abbreviation for ‘Autonomous’ or ‘Autonomous Routines’; used to abbreviate either general autonomous functions or specific routines e.g. ‘we need to code autons’

'Mogo'

An abbreviation for ‘mobile goal’; sometimes used to abbreviate a mobile goal clamp

'OU'

Abbreviation for ‘Over Under’, the 2023-4 V5RC season.

Bibliography

Bibliography

[1] V. Robotics, “High Stakes Manual.” 2024.

[2] V. F. / 'Littletimmy479', “Gearing on a six motor drive.” [Online]. Available: *https://www.vexforum.com/t/gearing-on-a-six-motor-drive/109357*

[3] in_ithica | 3818, “X Drive Posting with caption: 'x drive? (wall stake mech coming soon)'.” [Online]. Available: *https://imgur.com/a/y8n1rAr*

[4] BRLS, “Purdue SIGBots Wiki — wiki.purduesigbots.com.” [Online]. Available: *https://wiki.purduesigbots.com/*

[5] S. Magazine, “A LOOK AT HOLONOMIC LOCOMOTION.” [Online]. Available: *https://www.servomagazine.com/magazine/article/a-look-at-holonomic-locomotion*

[6] F. R. Network, “Pits & Parts | 9364H Iron Eagles - HailStorm | Over Under Robot.” [Online]. Available: *https://www.youtube.com/watch?v=EMaQuOrPwew*

[7] Wikipedia contributors, “Gear train — Wikipedia, The Free Encyclopedia.” [Online]. Available: *https://en.wikipedia.org/w/index.php?title=Gear_train&oldid=1219685098*

[8] V. F. / 'Xenon27', “Creating a Basic Drive.” [Online]. Available: *https://www.vexforum.com/t/creating-a-basic-drive/106873*

[9] S. Robotics, “How we CAD VEX Robots in Fusion 360 | Part 2 - A Simple Drivetrain.” [Online]. Available: *https://www.youtube.com/watch?v=gaJeJkWVefg*

[10] V. 98807B, “Finals Match #1, Mall of America signature | Vex High Stakes.” [Online]. Available: *https://www.youtube.com/watch?v=ixRf_hRArEQ*

[11] F. R. Network, “11101B Barcbots Getting There | Pits & Parts | High Stakes.” [Online]. Available: *https://www.youtube.com/watch?v=DbVMsuxRuag&t=396s*

[12] 6. 0470S Semicolon, “60470S 2 Piston Mogo Clamp (VEX Robotics Tipping Point).” [Online]. Available: *https://www.youtube.com/watch?v=VgsfGvBJ9-M*

[13] 2. Lucas Whiteaker, “Vex High Stakes Mobile Goal Clamp Prototype (Robot Prototype V1) | Team 23851A.” [Online]. Available: *https://www.youtube.com/watch?v=YkzFfF5XOf0*

[14] 2. 2204V Vangaurd, “22204V 1 Motor Mogo Mech (VEX Robotics Tipping Point).” [Online]. Available: *https://www.youtube.com/watch?v=Iv8quYM1e0I*

[15] V. Robotics, “Understanding Robot Features in V5RC Over Under.” [Online]. Available: *https://kb.vex.com/hc/en-us/articles/15540683424788-Understanding-Robot-Features-in-V5RC-Over-Under*

[16] V. Robotics, “V5 Motor Overview.” [Online]. Available: *https://kb.vex.com/hc/en-us/articles/360035591332-V5-Motor-Overview*

[17] V. Robotics, “Using the V5 GPS Sensor.” [Online]. Available: *https://kb.vex.com/hc/en-us/articles/360035591332-V5-Motor-Overview*

Bibliography

[18] 6210K, "Comment on post: Is GPS a great tool for autonomous?." [Online]. Available: *https://www.vexforum.com/t/is-gps-a-great-tool-for-autonomous/120533/3*

[19] Mentor_335U, "Comment on post: Is GPS a great tool for autonomous?." [Online]. Available: *https://www.vexforum.com/t/is-gps-a-great-tool-for-autonomous/120533/6*

[20] V. Robotics, "Using the V5 3-Wire Optical Shaft Encoder." [Online]. Available: *https://kb.vex.com/hc/en-us/articles/360039512851-Using-the-V5-3-Wire-Optical-Shaft-Encoder*

[21] V. Robotics, "VEX Robotics Store." [Online]. Available: *https://www.vexrobotics.com/*

[22] V. Robotics, "Using the V5 3-Wire Bumper Switch v2 & Limit Switch." [Online]. Available: *https://kb.vex.com/hc/en-us/articles/360038026831-Using-the-V5-3-Wire-Bumper-Switch-v2-Limit-Switch*

[23] V. Robotics, "Using the V5 Optical Sensor." [Online]. Available: *https://kb.vex.com/hc/en-us/articles/360051005291-Using-the-V5-Optical-Sensor*

[24] V. Robotics, "Using the AI Vision Sensor with VEX V5." [Online]. Available: *https://kb.vex.com/hc/en-us/articles/24062385752212-Using-the-AI-Vision-Sensor-with-VEX-V5*