

# Software is Eating Our Team - Final

An exploration of preemptive software design, taking advantage of AI systems, and their effects on raising the skill floor of FRC

By FRC Team 1716, Redbird Robotics, De Pere, Wisconsin

All software features described in this document can be found here: <https://github.com/Brown-County-FIRST-Robotics/1716-2023-robot>





# Table of Contents

- 1 - Foreword
- 2 - Motivations
- 4 - Experiment vs Architect
  - 4 - Laying the Foundation
  - 4 - GitHub
  - 5 - Code License
  - 5 - Advanced Programming Concepts
    - 6 - C++
    - 6 - Python
  - 7 - Challenges
- 8 - Key Software Features
  - 8 - DEATH\_STARE
  - 9 - AprilTag Awareness
  - 11 - Auto Balancer
  - 13 - KoneKube
- 17 - Kernel Patches and USB Ports
- 20 - Electrical Diagrams
- 24 - CAD
- 26 - NEO Motor Curve Data
- 31 - Manipulating the YouTube Algorithm
- 35 - Using ChatGPT for Fun and Donations
  - 35 - Using ChatGPT to get Sponsors
  - 39 - Using ChatGPT to Maintain Sponsor Relations
  - 43 - Automatically Finding Camera Device IDs on Linux.
  - 45 - Using ChatGPT to Write an Obscenely Huge Engineering Document
    - 46 - Synthesizing Student Scripts
    - 50 - Using ChatGPT to Create Visual Aids
    - 54 - Documentation Cleanup
  - 55 - Two AI's are Better than One
- 58 - Why We Document
  - 59 - Command Documentation
    - 59 - Table of Contents
    - 59 - Command Concept Overviews
    - 61 - How to make a...
    - 61 - Subsystem
    - 63 - Command
    - 64 - Button Binding

- 65 - Decorators
- 65 - Command Composition
- 65 - Lambda Use
- 66 - Default Commands
- 66 - Autonomous
- 68 - Generalized Solenoid Subsystem and Command
- 71 - WPILib Reference Document
  - 71 - Table of Contents
  - 71 - Component Reference
  - 72 - Naming Conventions
  - 72 - 3rd Party Vendor Libraries
  - 72 - Dashboard Values
    - 73 - SmartDashboard
    - 73 - Shuffleboard
    - 73 - Reading Values
  - 74 - NetworkTables
  - 75 - Solenoids
- 76 - What's Next?

# **Foreword**

This document is meant to be read similar to an encyclopedia, not a book. If one section bores you to death then feel free to move on to a section that interests you more.

Not everything we worked on made it into our final robot for competition, but we felt it important to document our process anyways.

In some cases spacing between words may look weird -- this is a result of the text being justified. We've done what we can to fix it in most places.

# Motivations

*More and more major businesses and industries are being run on software and delivered as online services — from movies to agriculture to national defense. Many of the winners are Silicon Valley-style entrepreneurial technology companies that are invading and overturning established industry structures. Over the next 10 years, I expect many more industries to be disrupted by software, with new world-beating Silicon Valley companies doing the disruption in more cases than not.*

— Marc Andreessen, August 20, 2011

In "Software is Eating the World," Marc Andreessen argued that software was driving innovation and disruption across all industries, leading to the creation of new opportunities and the destruction of old ones. He stated that software was increasingly being used to automate and improve traditional industries such as retail, finance, and healthcare, and that it was also creating entirely new industries such as social media and mobile apps. Andreessen predicted that this trend would continue and that software would play an even greater role in shaping the future of business and society.

The same has been happening to our team over the past year. From chatbots writing sponsor requests to computer vision, planning out our electrical layout to manipulating social media algorithms to do our research for us, this paper explores several methods to leverage modern software tools to our advantage. Some of these methods are just the result of months of dedicated and consistent effort on behalf of the team, others use tools that simply haven't been available until this year. We discuss the motivations, benefits, deficits, pitfalls, and context around which each of the methods were developed and put into practice.

The first motivation is that for the last several years our robot has been thrown together, sometimes literally, without much concern for the overall design of it. We've had the philosophy of "Duct tape solutions to it until it's nothing but duct tape", mainly because by doing things it felt like we were accomplishing things. We were going nowhere. Our process needed to change and we needed to start planning rather than just doing. The second motivation is that our team this year is lopsided in terms of manpower and skill. Before the season started, our team of programmers were experienced in using Linux, interacting with the robot, and knew several programming languages. On the other hand, our mechanical/electrical team members were new to CAD and were incorporating it into our planning process for the first time.



2022 Robot



2023 Robot

# Experiment vs Architect

## Laying the Foundation

When the 2023 FRC season started there was only one student who had programmed an FRC robot before. The previous season's robot was programmed using Python which was not a supported language at that time. The robot was functional for the 2022 season, but did not have any advanced features and used very few portions of WPilib.

At the start of the 2023 season the programming team decided that C++ should be used as the language to run on the robot rather than Python. By deciding to use C++ we were able to draw on the existing WPilib documentation as well as many existing examples on the internet. The majority of robot programming in C++ has been conducted by the person who worked on the Python code from last year. The existing knowledge of the roboRIO appears to have made progress easier.

The team also decided that a co-processor should be used and programmed in Python. There is functionality such as AprilTag and color detection that is better suited to a computing device that is faster than a roboRIO and can run the specific code and libraries. The co-processor will also return the camera images via a web server, viewable in a web browser, for the benefit of the robot driver.

At the start of the season the team laid out the goals of what we wanted to accomplish. We knew we needed a robot that could be controlled by humans. We also wanted the robot to be controlled by the co-processor with cameras connected. The major areas of development were broken into the roboRIO C++ programming, AprilTag detection in Python, and color detection in Python.

Students picked an area and conducted research and experiments to understand the feasibility of their area of interest. Each project put their efforts into a GitHub repository, in a separate directory. The work done was best described as a collection of test scripts. Once we understood how the various libraries worked in the way that we needed them to work, to accomplish what we wanted, it was time to architect the software that would make the robot function.

The students had never worked on a multi-person project before. Creating simple small scripts by yourself is nothing like working on a team to solve a common problem. While building software as part of a team is common in industry, it is not a skill set that comes naturally to new developers.

## GitHub

GitHub was the collaboration tool used for working together. GitHub has a program where non-profits can gain access to the Teams tier of features which we took advantage of. While GitHub was used last year, there was only one returning programmer. Again, the students had used

GitHub themselves for their projects and were familiar with the basics, they had never worked as part of a larger team.

GitHub has a large amount of functionality, not all of which we are using. We are using one repository to capture all of the work of the programmers. Within that repository we are logging tasks and ideas as issues and markdown documents.

We are using branches heavily for all the code. GitHub has the ability to protect the main branch to prevent adding code without first receiving an approval from another team member. A simplified description of our work flow is:

1. Create a branch
2. Commit changes to branch
3. Create pull request to add changes to main branch
4. Resolve any conflicts in the pull request
5. Get an approval from another team member
6. Merge changes into main branch.

This is how GitHub is used by software professionals in the industry. As all learning experiences, it took time to understand this workflow, but after a few weeks there were no issues.

## Code License

The software written by the team is in a public GitHub repository. Making the repository public solved some problems around needing access to the repository on various test systems, but most importantly it was an opportunity to explain open source licenses to the students. All source code in the repository has been licensed under version 2.0 of the Apache license. Apache 2.0 was chosen due to its current popularity by open source in industry. It also allows the code to be used by others with few restrictions, and contains patent protection provisions to reduce potential risk from using open source software in commercial products.

While many open source libraries are used in the programming of a FRC robot, how open source works is not a point of common understanding. For example, the source code behind WPilib is covered by the MIT open source license. Having a basic understanding of how open source works as well as ensuring our software is covered by an open source license was an important consideration. If a repository is public on GitHub and does not explicitly specify an open source license, the code is “all rights reserved” by the copyright holders. This “all rights reserved” prevents others from legally using the source code as part of another project.

## Advanced Programming Concepts

In addition to using GitHub as a development team, we also included some other advanced industry software development concepts.

In many software teams today there is a concept called pair programming. The idea behind this is to have two people work on one computer to build software. One person uses the keyboard while the other person sits next to them and helps work through writing the software. Bugs can be caught quickly and very difficult problems can be solved by the two people working as one. While we have not been able to use this process consistently, when team members need to work together it is a natural and productive way to solve problems.

Our Python project is using a software development process known as test driven development. The idea behind test driven development is to add an automated test for every feature and bug fix. By relying on automation it is easier to identify unexpected problems in the software being written. We have implemented a basic test framework and find we can use the tests to find very simple problems in our python project. We are not currently writing robust tests, but the students quickly came to understand the advantages to test driven development. We hope to leverage it in more detail next year.

We are also trying to use simulation whenever possible. WPilib contains a simple simulator we can use to verify certain features work as expected. We are able to ensure that the C++ code does what we expect and won't crash the roboRIO. For example if we are trying to move a motor with a joystick, we can use the simulator to ensure the joystick does in fact make a motor move.

## C++

The C++ code that runs on the roboRIO is considerably more complicated than the Python code used last year. Last year the robot used what is known as a "timed robot" which sequentially runs commands in a loop. This is a relatively easy to understand model we know as synchronous programming. Everything happens in an expected order.

This year the students decided that they would use the event driven model known as a "command based robot" in the WPilib documentation. This is a concept known as asynchronous programming where tasks are not executed in an order, but rather executed by external events such as timers and button presses. While asynchronous programming is more difficult to architect and debug, we have found that when adding more advanced features later in development it was the correct decision. In very complicated systems asynchronous programming results in less complexity than synchronous programming would due to the huge number of possible input and output scenarios.

## Python

Recognizing the limitations of the roboRIO as well as the added difficulty of programming C++, we decided to use a raspberry pi 4 running Python code to be a coprocessor. The Python code will process camera images, then using those images to detect color, AprilTags, and send the images to the robot driver on a web page. This has resulted in a complex Python project that required multiple components and cooperation. The radio on the robot has a very limited amount of bandwidth it can return to the driver station computer, so most of the images

captured never leave the Raspberry Pi. We are able to process camera frames at a much faster rate than we can return to the humans.

Python has a concept called a “module” which is a way to write a shared library that allows code to be segregated and shared across the application. By using modules the students are encouraged to write more specialized code. By keeping these specializations in modules, it makes working as a team easier as in many instances the students are working in different files which helps reduce merge conflicts.

We have built modules to handle functionality such as reading from the camera, processing camera images for AprilTags and objects, as well as sending camera images to a web server that can then be shown with a web browser. The tests are also modules that exist in their own directory.

## Challenges

While the programming team has done a great job of coming together and architecting a functional robot, this was not without challenges. All of the students had spent their entire software development time working alone on small projects. Never before did they have to communicate and cooperate to solve software problems. There are still difficulties around understanding that the code isn’t owned by a single person but is part of the larger project. It is often difficult to create something then stand by idly as others change your creation. This is a mindset that takes many years to accept.

We have also seen some of the more formal development steps getting ignored as development progresses. We are no longer creating and updating GitHub issues as often as was being done at the beginning of the build season. This is partially due to the time crunches of late stage robot development, and partially due to the students having a solid understanding of what needs to be done.

# Key Software Features

## DEATH\_STARE

DEATH\_STARE (Driver Extension App To Hopefully See The Action Really Easily) is our attempt to display camera feeds to the driver in a web browser to make driving on the game field easier. The inspiration for this application came when we competed in a practice competition and our robot was not easily controlled due to lack of a good first person view. Inspiration also came from our mentor's backup cameras on their vehicle which gives them a top down view of a virtual car with a surround view of their environment.



*Surround View of a Nissan Leaf*

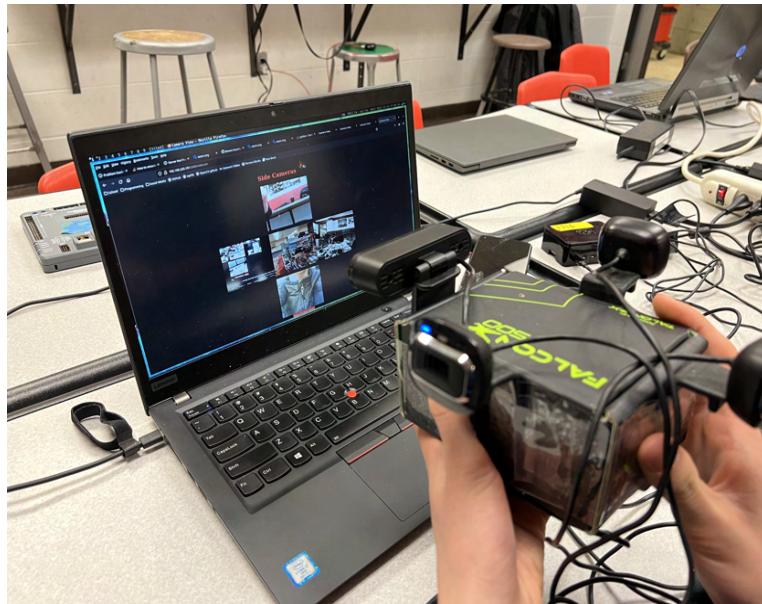
Four cameras streaming 4 images at even  $640 \times 480$  resolution is a lot of data to send to a USB and is lot of data for a program to process all at once. Having it all stream the camera input into a single USB dongle caused bandwidth problems and the computer could not handle all the data. Our workarounds are discussed in the "Kernel Patches and USB Ports" section, but the high level of how we stream data is that we have a web server on our driver station that

sends requests to our camera module on the raspberry pi. That camera module responds with frames taken from OpenCV, with which the web server renders to a browser window.



*Our test board for our cameras*

On the robot, we will physically place 4 cameras facing in a different direction. Those cameras are associated with a certain position in the web browser (top / bottom / left / right). This doesn't form a perfect view of the robot's surroundings as the corners are not filled in. We fix this by distorting the image with CSS by rotating it about the x-axis/y-axis to get the trapezoid effect. While this might not actually be the perfect way to distort the image to be viewed, it works mostly fine for our purposes.



*Early version of the DEATH\_STARE module*

## AprilTag Awareness

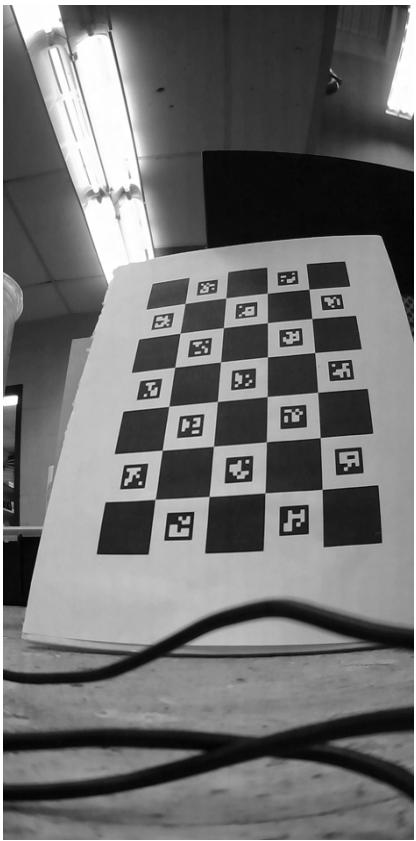
We believe that the apriltag detector is a critical component of our robot throughout the game. In order to navigate the field, it is necessary for us to know the position and orientation of our robot. Apriltags proved to be the best solution for this purpose, as they enable us to drive onto the platform during the autonomous round and end game while being aligned with the charging station, as well as place game pieces accurately.

Initially, we attempted to calculate the distance between the apriltag and our robot by holding the apriltag a known distance away from the camera and extrapolating out. However, we soon realized that this method only provided the distance and not the full picture of the apriltag's position and rotation on the screen relative to the center of the captured image.



*Testing the max range of our AprilTag detector -- 30 feet*

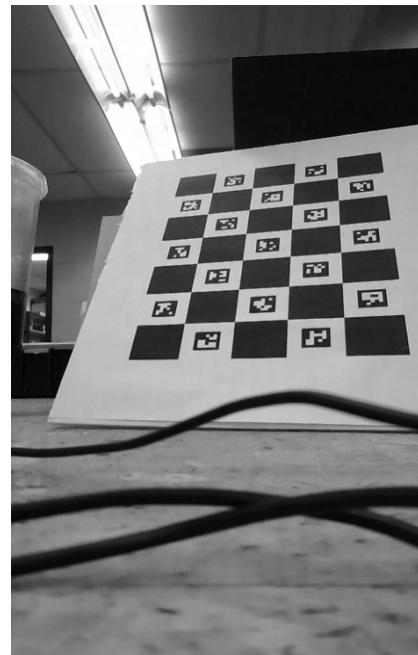
To obtain a complete understanding of the apriltag's location and orientation, we discovered OpenCV's `solvePnP` function, which could take an image of the apriltag and calculate its exact position and orientation on the screen. The only requirement was that our camera needed to be calibrated. We accomplished this using a camera calibration script from the OpenCV website. However, during testing, we encountered an issue with the camera output appearing distorted. After careful examination, we determined that this was due to the camera capturing at a different resolution than it was calibrated for. To resolve this issue, we added the calibrated resolution to the calibration file and set the camera to that resolution at startup, which corrected the math error caused by undistorting a normal image. The `solvePnP` function was then able to accurately calculate the apriltag's position and orientation on the screen.



*The original distorted image*



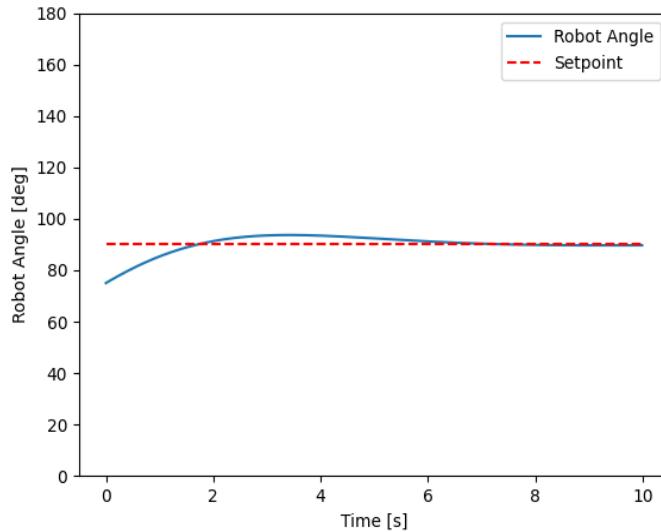
*Applying the distortion  
correction algorithm*



*Crop of the usable space*

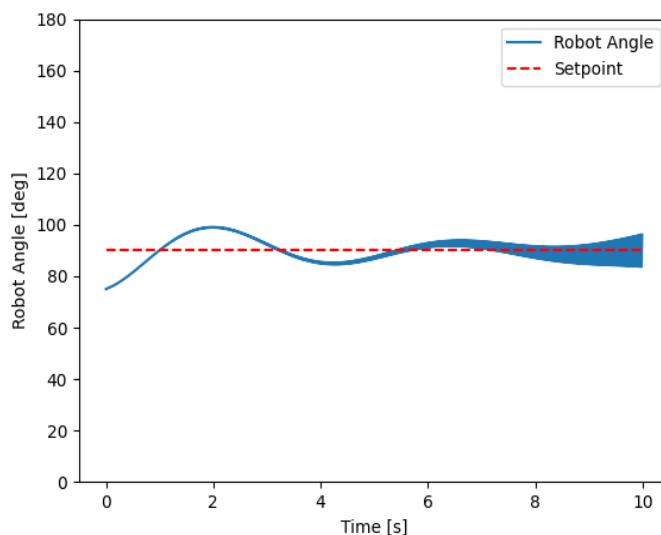
## Auto Balancer

Our team decided right from the beginning to create an auto balancer to make us more valuable during competitions, both to us and our alliance. To achieve this we opted to use a PID loop, a mathematical algorithm that attempts to reach a set point (such as angle of a robot) in as short a time as possible. With minimal oscillations, and using only one input and one output, which is the pigeon's pitch and the motor output percent, we are able to balance our robot onto a charging station extremely quickly without human intervention.



*An example of a robot using its IMU to detect and correct its angle while balancing on the charging station*

A PID loop has three coefficients that control its functionality: Proportional, Integral, and Derivative. The proportional term responds directly to the input value and provides an output that is proportional to the input, which makes it react quickly but prone to oscillations. The integral term adds to itself over time, which essentially represents the past, making it slow to react but eventually enough to help the robot overcome any obstacle. Finally, the derivative takes the slope of the input to reduce overshoot, which corrects the robot's sudden leans and reduces the speed of the other two terms. However, too high of a derivative can lead to overcorrection and violent oscillations.



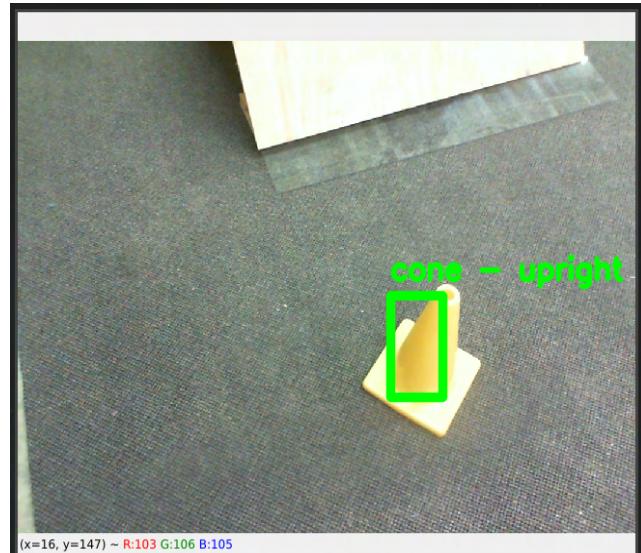
*An example of a poorly configured PID loop. The line gets thicker towards the end because the robot is violently oscillating back and forth.*

## KoneKube

KoneKube is our computer vision program to detect the cones and cubes on the field. It uses OpenCV's color and contour detection under the hood. Giving the robot the ability to detect the locations of cones and cubes on the field can offer massive advantages, since we can use those as inputs to our various other subsystems.



Detecting a cube



Detecting a cone

While the color detection works very well with consistent flat shapes, lighting causes some parts of the cone to be dark while other parts are very bright. This caused the color filter to be messed up and the cone to not be completely detected by the color detection. Similar problems were faced with the cube in that purple is very similar to blue which led to the program detecting both colors which could be a problem when blue bumpers come into the camera's FOV.

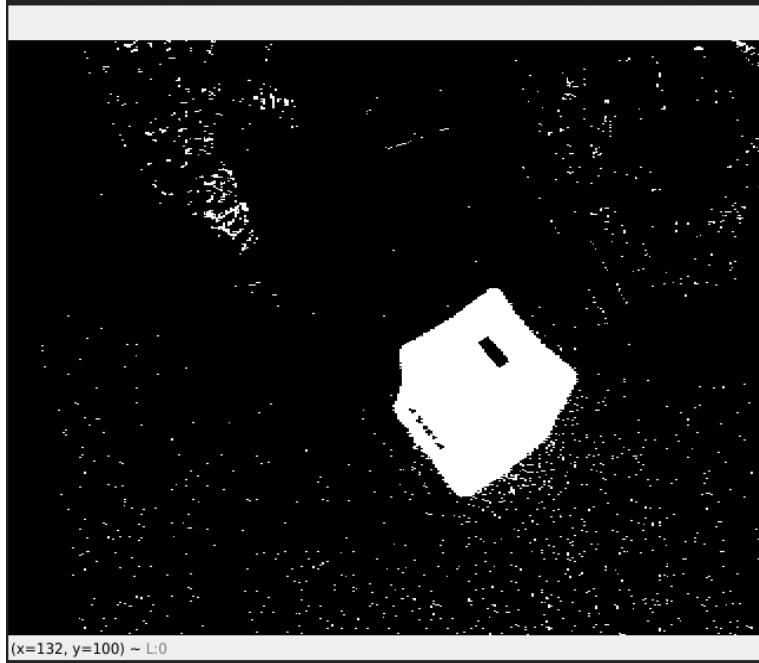


*Color similarity between a purple cube and a blue recycling bin*



*Bright lights on a cone show how they can affect the color*

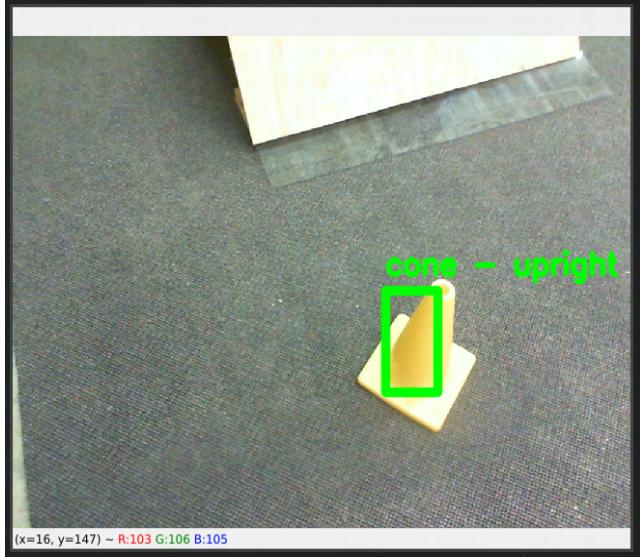
Our first attempt at a solution was a flood fill algorithm where we checked the ratio of adjacent pixels in the image. We determined that if the range of colors for a group of pixels are within a certain range, it is very likely that they are a part of the same object. We then repeated it for the rest of the pixels. This only had to check each pixel in the image once but there are a lot of pixels in the image ( $640 \times 480 = 307200$  pixels!). For the time being, we decided to scrap the flood fill algorithm due to its incredibly slow processing speed.



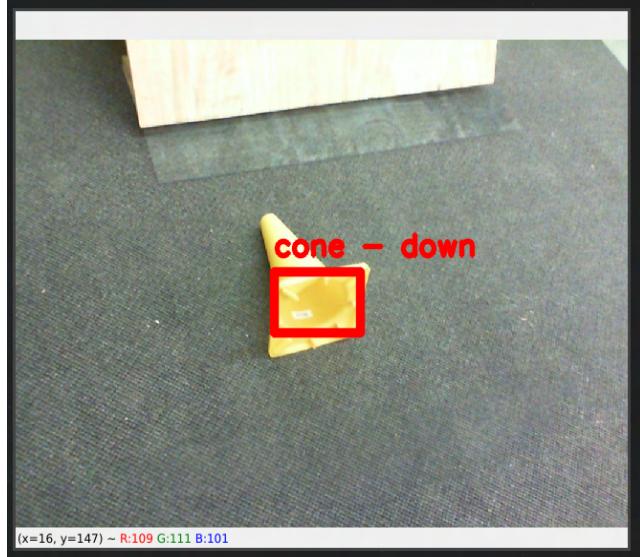
*Using our color filter to highlight all purple colored things*

Our simple solution that we are working with now is to do manual testing and adjust the range of color until it is mostly accurate at a decent range. Additionally, we changed our process so that we primarily look at hue for cubes which is fairly good at coloring the cube and avoiding lighting issues (it did pick up colors such as black and blue but it seems to mostly be effective at detecting the cube).

Detecting if a cone is upright through computer vision is fairly important. Our solution to do this was to first check the aspect ratio of the bounding rectangle for the cone. If the ratio of the width to height is greater than one, then the cone is likely downed and we can mark it as such. However, doing just this isn't always accurate as it doesn't account for some common edge cases where the robot is directly facing a downed cone from its top or bottom. To handle for this, we bisect the bounding box into a top and bottom half. Then we check to see how much yellow is in the top vs bottom half. If there is more cone on the bottom and the width to height ratio is less than one, we can be reasonably sure the cone is upright. This likely could still be improved to be more accurate but it works for now.



Detecting an upright cone



Detecting a downed cone

## Kernel Patches and USB Ports

As a part of creating our computer vision suite, we needed to figure out how to livestream multiple camera feeds into a raspberry pi simultaneously. We needed to keep the data rate below 480 Mb/s since the raspberry pi only has one USB 2.0 controller and our cameras are USB 2.0. Another restriction was that we needed to be able to process the camera feeds using OpenCV and Python so that it would fit into our existing pipeline. We ran into many problems along the way, some of which were self-induced due to cost constraints, others due to lazy and/or deceptive manufacturers. Either way, we made sure to document the process here so that others might avoid the problems that we ran into.



*Raspberry Pi USB layout*

To start us off, each camera individually streamed data at over 200 Mb/s without compression -- that isn't even enough to have 2 cameras streaming simultaneously, since we need to have a bit of a buffer (say, 80 Mb/s) to handle for extraneous functions on the pi. There were several ways we tried to limit that bandwidth, including changing the compression format, changing the USB mode, reducing the frame rate, reducing the resolution, and plugging our cameras into an external USB 3.0 hub, all of which came with a whole host of issues.

The cameras we bought are cheap but still decent in terms of visual quality. Because we didn't go all out though in our camera purchase (e.g., hundreds of dollars per camera), our cameras only support the MJPEG video compression format. You'd think this would help, except that even when we specified to the camera to compress using MJPEG, they were still streaming at over 200 Mb/s!

One way people might intuitively think about cameras streaming data is that if they have data they send data. This is not always true, like in our case. When our cameras initialize, they initialize in "isochronous" mode which reserves a data slot on a bus. This reservation takes into account resolution but ignores compression. There is a parameter that we can specify when

initializing the UVC Camera Driver that attempts to calculate the bandwidth requirement of the camera based on frame rate, resolution, and color depth, but still fails to take into account compression.

We tried to fix this issue by plugging our cameras into a USB 3.0 expansion hub, then plugging that USB 3.0 expansion hub into the USB 3.0 port of the raspberry pi. We expected that the USB 3.0 bus (which operates at 5 Gb/s) would be enough to handle our four camera streams. The first time we did this, we found out that the "USB3.0-7Port SuperSpeed" hub that we had was actually only USB 2.0. The second time we did this (with another USB 3.0 Hub), we found out that while the USB 2.0 and USB 3.0 controllers share the same port, there is no interoperability between the two standards and they are effectively two separate ports crammed into the same form factor. This meant that even though our USB 2.0 cameras were plugged into USB 3.0 ports, only the USB 2.0 lines were communicating information throughout the entire pipeline. As a fun side effect, this means that the raspberry pi technically has four USB 2.0 ports and two USB 3.0 ports, but that still doesn't solve our problem.



Fake USB 3.0 hub that is actually just USB 2.0

```
mark@opencv:~$ lsusb -t
[...]
/: Bus 02:Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/4p, 5000M
/: Bus 01:Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
    |__ Port 1: Dev 2, If 0, Class=Hub, Driver=hub/4p, 480M
        |__ Port 1: Dev 15, If 0, Class=Hub, Driver=hub/4p, 480M
            |__ Port 1: Dev 16, If 0, Class=Video, Driver=uvcvideo, 480M
            |__ Port 1: Dev 16, If 1, Class=Video, Driver=uvcvideo, 480M
            |__ Port 2: Dev 17, If 0, Class=Video, Driver=uvcvideo, 480M
            |__ Port 2: Dev 17, If 1, Class=Video, Driver=uvcvideo, 480M
            |__ Port 3: Dev 18, If 0, Class=Video, Driver=uvcvideo, 480M
            |__ Port 3: Dev 18, If 1, Class=Video, Driver=uvcvideo, 480M
            |__ Port 4: Dev 19, If 0, Class=Hub, Driver=hub/4p, 480M
                |__ Port 4: Dev 20, If 0, Class=Video, Driver=uvcvideo, 480M
                |__ Port 4: Dev 20, If 1, Class=Video, Driver=uvcvideo, 480M
mark@opencv:~$
```

Showing all the USB 2.0 ports using the command 'lsusb -t'

The source of the problem ended up being that our cameras specify way too high of a bandwidth reservation. A solution to this is to tell the UVC Video driver to ignore the camera's bandwidth needs and manually specify it elsewhere. A patch was submitted to the kernel developers over a year ago that implements this, but it was rejected for being too "hacky". Thankfully we don't need to care about that, so we copied the kernel patch to our local system, brought it up to date with the current kernel source code, built the new UVC Video module on the raspberry pi, and swapped it out. This gave us a new parameter when initializing our compressed MJPEG stream that the camera actually adheres to. After initializing with this new setting, our cameras stream at approximately 80 Mb/s per device.

```
[2835.096037] usb 1-1.1.4.4: Adding mapping 'Power Line Frequency' to cont
[2835.096047] usb 1-1.1.4.4: Added control 00000000-0000-0000-0000-00000000
[2835.097039] usb 1-1.1.4.4: Adding mapping 'White Balance Temperature, Au
[2835.097047] usb 1-1.1.4.4: Scanning UVC chain:
[2835.097052] OT 5 <- XU 3 <- PU 2 <- IT 1
[2835.097070] usb 1-1.1.4.4: Found a valid video chain (1 -> 5)
[2835.110277] usb 1-1.1.4.4: Bandwidth capped from 3072 to 1000 B/frame.
[2835.136607] usb 1-1.1.4.4: Bandwidth capped from 3072 to 1000 B/frame.
[2835.137253] Input: HD USB Camera: HD USB Camera as /devices/platform/scb
[2835.137330] 1.4.4:1.0/input/input17
[2835.137330] usb 1-1.1.4.4: UVC device initialized
[2835.144113] usbcore: registered new interface driver uvcvideo
[2835.144416] usb 1-1.1.4.4: uvc_v4l2_open
[2836.031910] usb 1-1.1.4.4: uvc_v4l2_release
[2836.031935] ush 1-1.1.4.4: Suspending interface
```

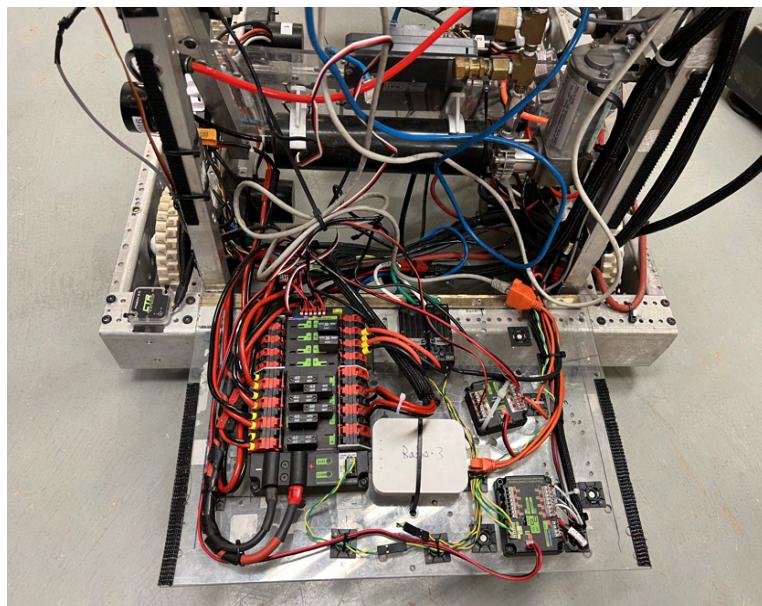
*UVC Kernel Driver logs showing the bandwidth adhering to our cap*

Something to note is that this problem of streaming cameras over USB is not specific to the pi. It applies to any system that has a single USB controller, because the core issue here is that there is not a ready and available method for limiting camera bandwidth for MJPEG streams. It's just that most people find it on the pi because they hear it's good for hobby projects, see four USB ports, and assume that they can plug it full of cameras and it will just work.

The Kernel Driver patch as well as the process to implement it can be found on the GitHub page listed on the cover of this document.

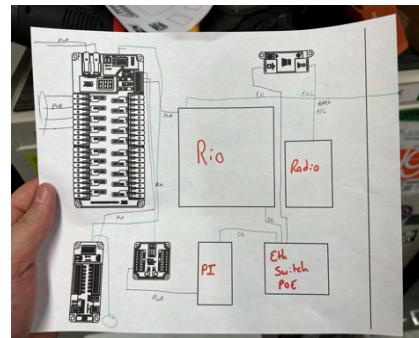
# Electrical Diagrams

Last year, we encountered significant difficulties with our electrical board. The board was poorly designed and had disorganized wiring, which led to difficulties with troubleshooting and disconnect issues, as well as a messy appearance. This is approximately what our robot's wiring looked like last year, though this is after lots of tidying up:

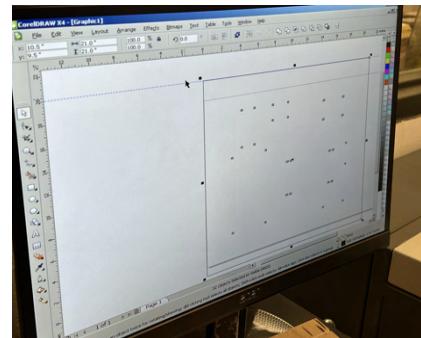


*The wiring on our 2022 robot*

We wanted to create a functional and aesthetically pleasing board this year, so we took proactive steps during the pre-season and current build season. To ensure that the wiring would be organized and efficient, we created a wiring diagram using a program called Mermaid and designed the physical board in CAD. We then cut out the necessary holes with a laser cutter and mounted all the electronics.



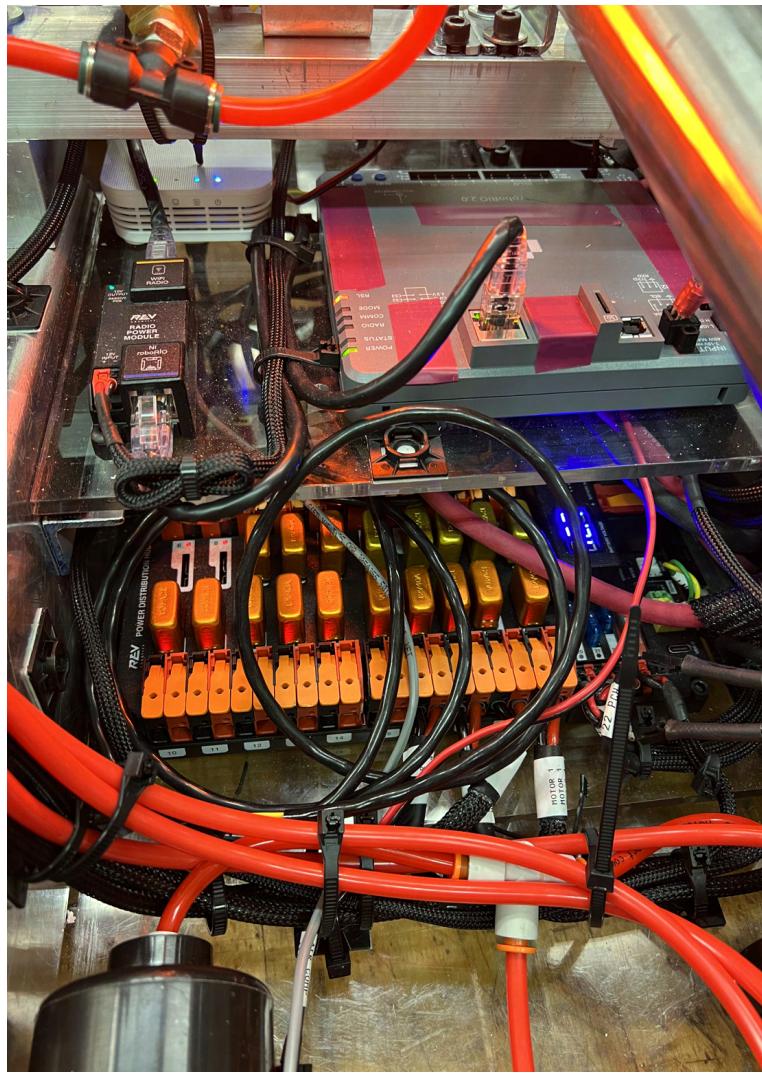
CAD Printout with approximate wiring paths



Cutting the board out on the laser cutter

*Planning the layout*

Our goal was to provide proper protection for the electronics during matches and make the board easily accessible. Given our frame constraints, we decided to go with a "double decker" design, splitting our electrical board into two pieces. Both boards are mounted with hook and loop fasteners so that we can remove them easily if the need arises.

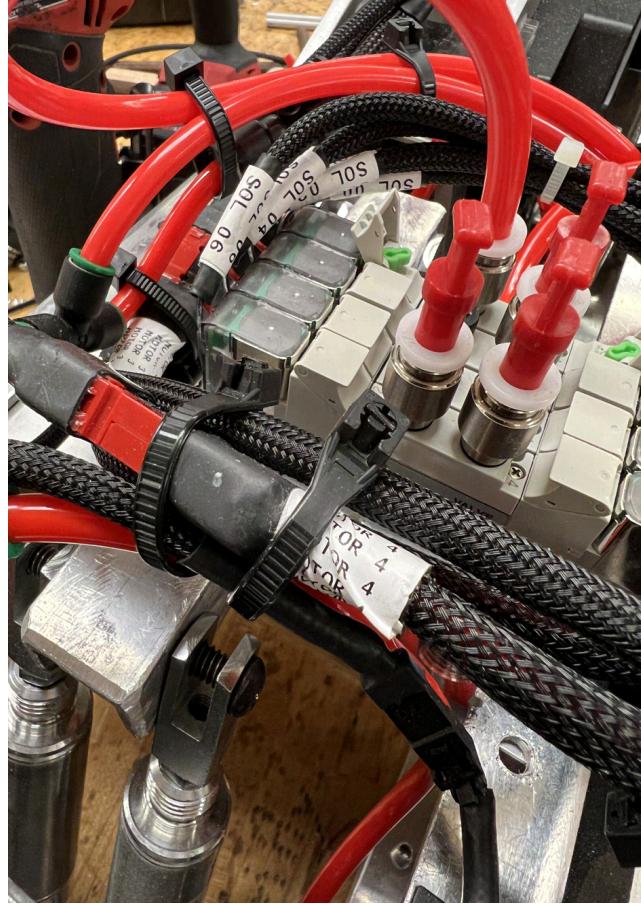


*Our double decker design*

We also took care to use latching connectors and higher quality cable protection. During the pre-season we researched many different kinds of latching connectors. We settled on testing three different styles -- Molex Micro-Fit, Molex SL, and AMPMODU MTE. We ended up using a Molex SL clone with an AMPMODU MTE crimper. To protect our cables, we are using braided cable sleeves and dual wall heat shrink on all of our wires to help to prevent wear and chafing.



*Our latching connectors*



*Our tech flex, heat shrink, and labeled motors*

## CAD

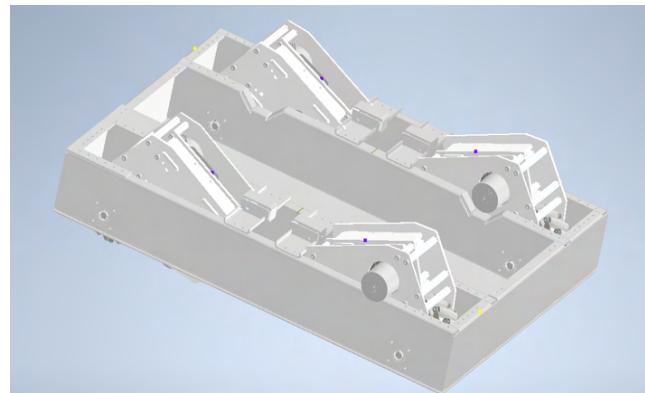
CAD is an important part of any robotics workflow. We would love to show you a comparison of last year's CAD vs this year's, but we didn't deem it important enough last year to have anything to show. This year we decided that if we didn't have the time to CAD it, we didn't have the time to build it, so during the pre-season we took about a month of meetings to have our mechanical team learn the basics of CAD.

Armed with this new tool, the mechanical team had a number of advantages compared to prior years of development. Team members were able to quickly iterate their designs and work with each other to virtually integrate different parts of the robot. Interferences could be foreseen and designs could be put on a virtual field to ensure the robot could interact with its environment as intended. In addition the development of the robot in a CAD environment enabled the use of CNC fabrication of components. Instead of taking the time to mark out and machine each feature, parts could be quickly prototyped through the use of acrylic and a laser cutter which saved a considerable amount of time.

During our competition season last year we chose mecanum. It was a useful drive base up until other robots came into the equation and started pushing us around. We had zero stopping power. To avoid that problem while also taking advantage of the mobility that holonomic motion offers, we chose to build our own butterfly drive.

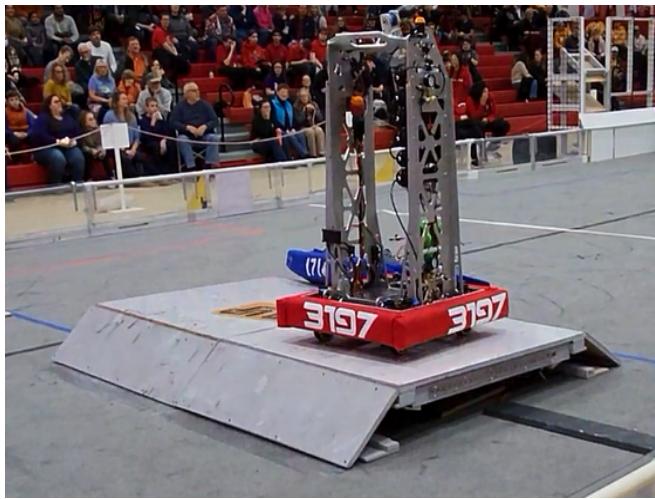


*Early prototype of a butterfly module*



*Early CAD of our frame*

One thing we noticed early on was that with the bumpers on it was very difficult to climb the charging station with the standard wheel distance from the outside of the frame (such as in the kit of parts). It was one of our design considerations when designing our custom chassis and has proved itself useful. At a practice regional we were consistently able to drive over the charging station while many other robots were having trouble due to how much material was in between the wheels and the charging station. For reference to other teams, the distance from the end of our wheel to the outer perimeter of our chassis is 1.25" vs the kit of parts' 3" (for a 6" wheel).



*Our robot (in blue) trying to get onto the charging station at a low speed*



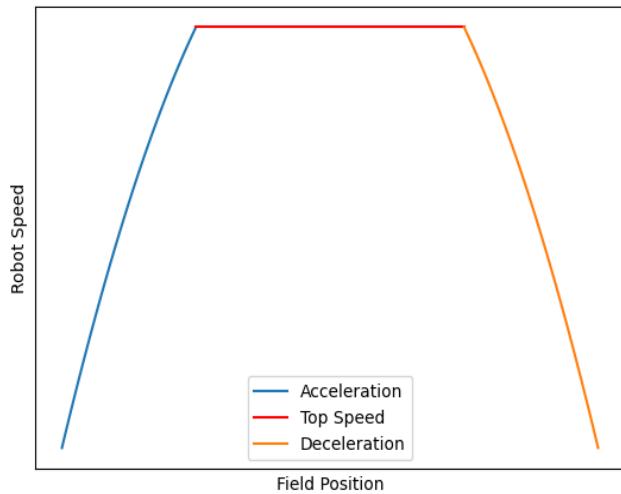
*A moment later, we successfully made it onto the platform to balance*

# NEO Motor Curve Data

Charged Up has a very open field. It seemed obvious to us that any serious contender will need to reduce cycle times as much as possible. To that end, we set out to find (and prove!) the optimal gearing ratio to allow our robot to cross the field as fast as possible. There are three main aspects of traveling across the field:

- Speeding up
- Maintaining max speed
- Slowing down

For our purposes, we assumed the acceleration and deceleration curves were identical.

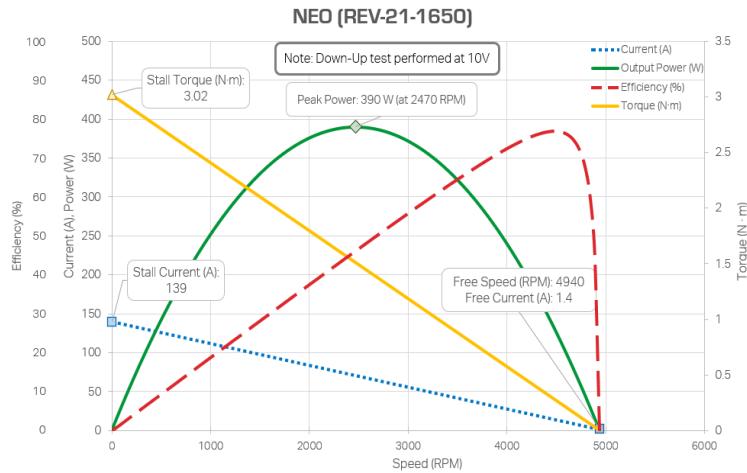


*Example of the different parts of moving a robot across the field*

To start off, we needed a formula that considered several things:

| Data                    | Value  |
|-------------------------|--|
| Mass                    | 150 lbs / 68.0388 kg                           |
| Wheel Radius            | 3 in / 0.0762 m                                |
| Gear Ratio              | COTS options (5.95/7.31/8.45/10.71/12.75 to 1) |
| Field distance          | 582.26 in / 14.7894 m                          |
| Motor Type              | NEO  |
| Stall Torque            | 2.6 Nm   |
| Torque Falloff Constant | -0.0004502164502 Nm/RPM                        |

We got our motor specs from the NEO motor curve data provided by the manufacturer and related material:



NEO Motor Curve

Using this information, a simplified linear relationship could be derived between the motor torque output and the motors rotational velocity. The torque would start at the stall torque when the robot was at rest and would continue to decrease as the robot got up to speed. The rate that the torque decreased per RPM (torque falloff constant) was estimated and listed in the table above.

With this data we used an Excel spreadsheet to simulate the acceleration of the robot and the leveling off at a constant speed. This helped us understand how the torque made the robot accelerate at a certain rate, and how as the robot sped up, the wheel speed increased while the torque speed decreased. We ran this simulation for 10 seconds, with timesteps of 0.1 second, and we could see where the robot speed leveled off.

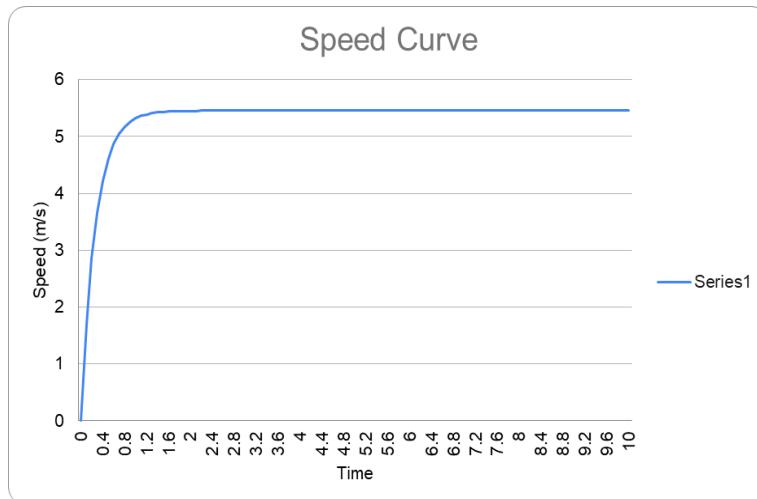
| Time (s) | Position (m) | Velocity (m/s) | Acceleration (m/s <sup>2</sup> ) |
|----------|--------------|----------------|----------------------------------|
| 0        | 0            | 0              | 16.95034069                      |
| 0.1      | 0.084751703  | 1.695034069    | 11.68194975                      |
| 0.2      | 0.312664859  | 2.863229043    | 8.051044661                      |
| 0.3      | 0.639242987  | 3.66833351     | 5.548673085                      |
| 0.4      | 1.033819703  | 4.223200818    | 3.824071819                      |
| 0.5      | 1.475260144  | 4.605608       | 2.635499524                      |
| 0.6      | 1.948998442  | 4.869157952    | 1.81635128                       |
| 0.7      | 2.444995993  | 5.05079308     | 1.251805186                      |
| 0.8      | 2.956334327  | 5.175973599    | 0.862727514                      |
| 0.9      | 3.478245325  | 5.26224635     | 0.594580348                      |

| Time (s) | Position (m) | Velocity (m/s) | Acceleration (m/s^2) |
|----------|--------------|----------------|----------------------|
| 1        | 4.007442861  | 5.321704385    | 0.409776881          |

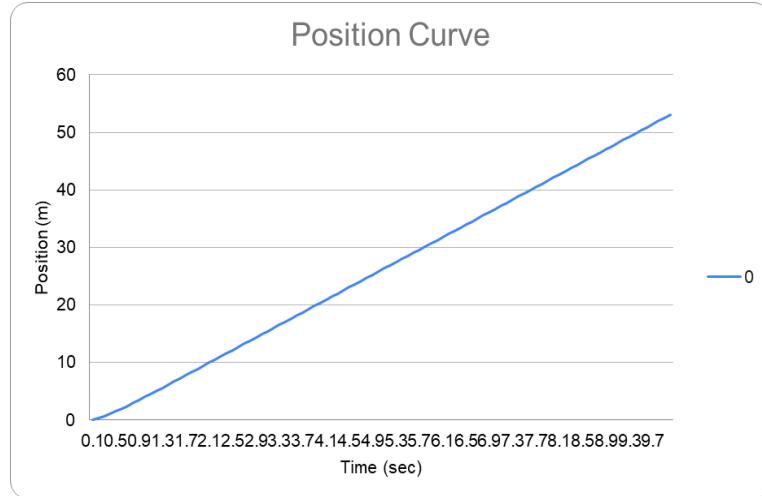
| Output_Wheel_Speed (RPM) | Output Wheel Torque (Nm) | Motor_Speed (RPM) | Motor_Torque (Nm) |
|--------------------------|--------------------------|-------------------|-------------------|
| 0                        | 21.97                    | 0                 | 2.6               |
| 212.419725               | 15.1414323               | 1794.946676       | 1.791885479       |
| 358.8165791              | 10.4352741               | 3032.000093       | 1.234943681       |
| 459.7113472              | 7.19185236               | 3884.560884       | 0.851106788       |
| 529.2466818              | 4.956529159              | 4472.134461       | 0.586571498       |
| 577.1695111              | 3.415974087              | 4877.082369       | 0.404257288       |
| 610.1972888              | 2.354243986              | 5156.167091       | 0.278608756       |
| 632.9595947              | 1.622513697              | 5348.508575       | 0.192013455       |
| 648.6470737              | 1.118214898              | 5481.067773       | 0.132333124       |
| 659.4586759              | 0.770658861              | 5572.425812       | 0.091202232       |
| 666.9098886              | 0.531127855              | 5635.388559       | 0.062855367       |

The table was too wide, so we had to split it. Short sample of our simulation with a gearing ratio of 8.45:1, the optimal gearing ratio for our COTS gearboxes

We created two graphs to visualize the motion of the robot. The first graph plots the speed of the robot over time, and the second graph plots the position of the robot over time.



Speed Curve for a gearing ratio of 8.45:1, reaching a max speed of ~5.45m/s in about 1 seconds

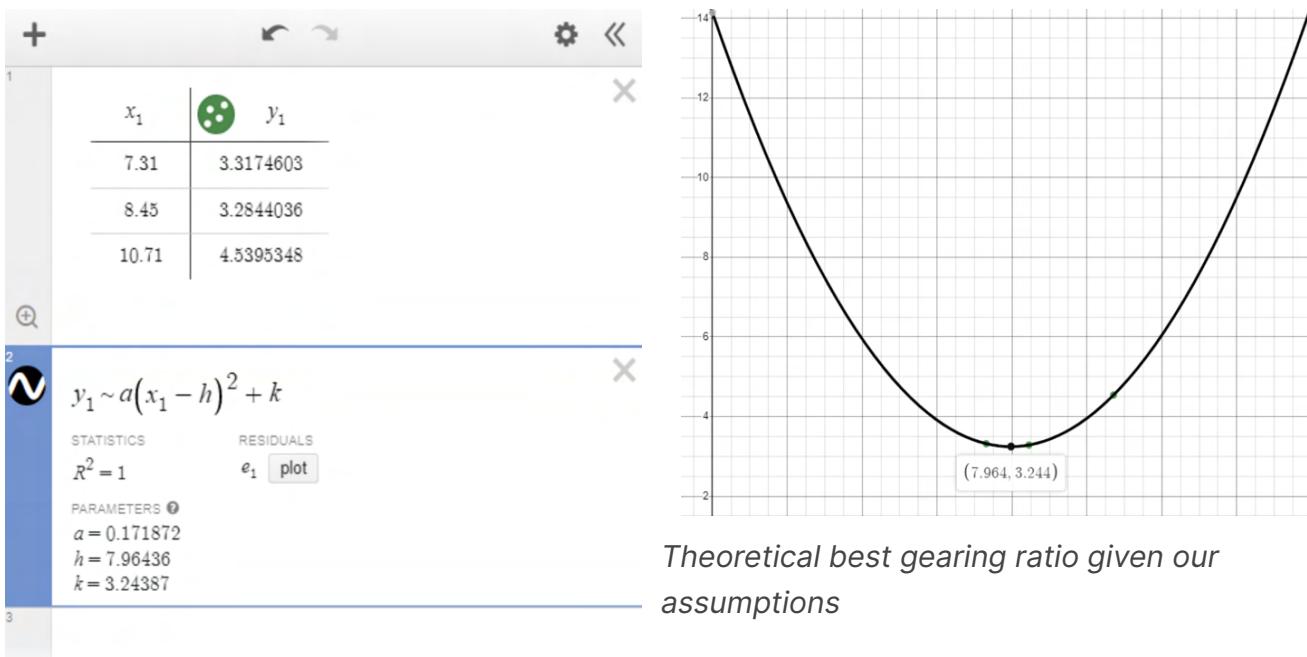


*Robot position on field for a gearing ratio of 8.45:1*

Here is a table of all the COTS gearing ratios we had access to:

| Gear Ratio | Max Speed Time (s) | Max Speed (m/s) | Remaining Distance (m) | Max Speed Distance (m) | Remaining Time (s) | Total Time (s) |
|------------|--------------------|-----------------|------------------------|------------------------|--------------------|----------------|
| 5.95       | 3                  | 7.75            | 0                      | 18.6                   | 0                  | 6              |
| 7.31       | 1.5                | 6.3             | 2                      | 7                      | 0.317460317        | 3.317460317    |
| 8.45       | 1                  | 5.45            | 7                      | 4                      | 1.28440367         | 3.28440367     |
| 10.71      | 0.6                | 4.3             | 10.92                  | 1.94                   | 2.539534884        | 4.539534884    |

The key point of this chart is to look at the "Total Time" column. That shows, in seconds, how long it takes for each gearing ratio to get across the field. You can see that the sweet spot for COTS is 8.45:1, though as an exercise we wanted to see where the optimal gearing ratio is. We found that according to our simulation, the theoretical perfect ratio is around 8:1.



### Desmos Calculations

With all that being said, there are some things we could have done better. For starters, these calculations assume the wheels are traction wheels while we have mecanum, though in practice this effect seems minimal. It also doesn't take into account some variables such as friction coefficients or that the acceleration and deceleration curves may not be exactly equal.

Our full spreadsheet and calculations can be found here:

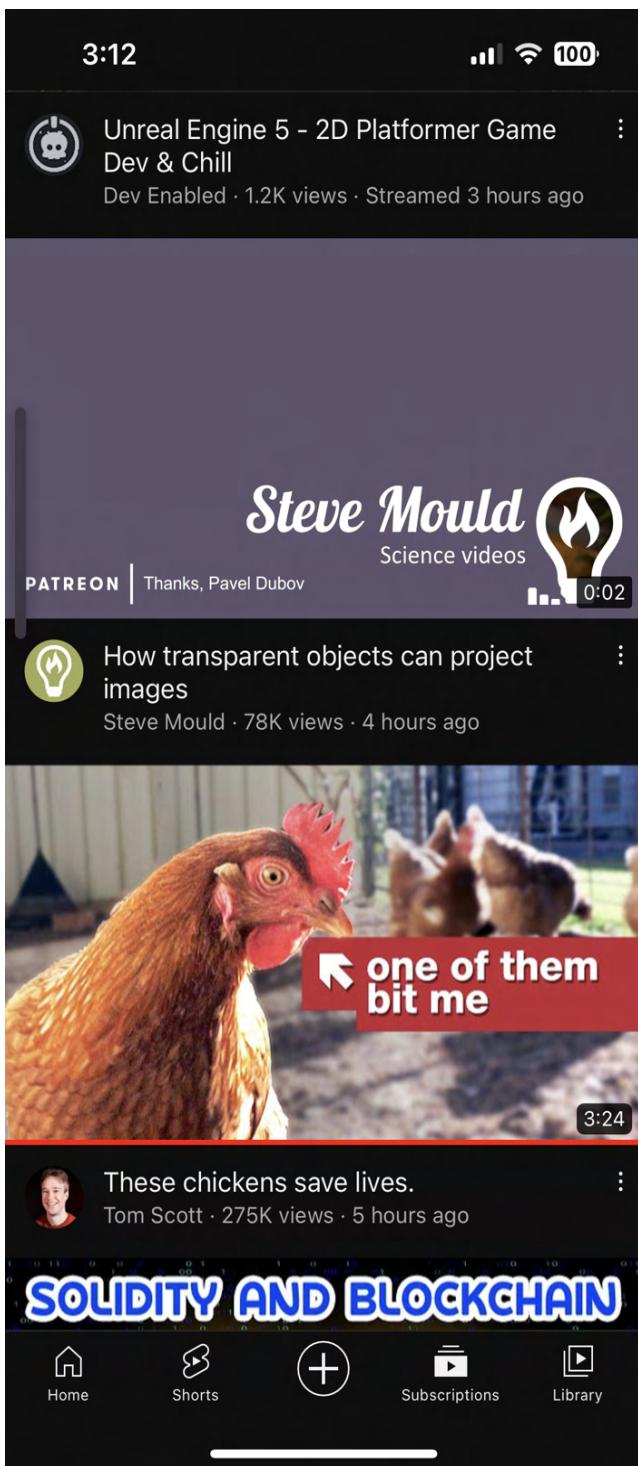
[https://docs.google.com/spreadsheets/d/1p0qtzqADmQgesjxbGEKo\\_7EsibrtspHn2\\_ffg3v\\_oE/edit#gid=0](https://docs.google.com/spreadsheets/d/1p0qtzqADmQgesjxbGEKo_7EsibrtspHn2_ffg3v_oE/edit#gid=0)

# Manipulating the YouTube Algorithm

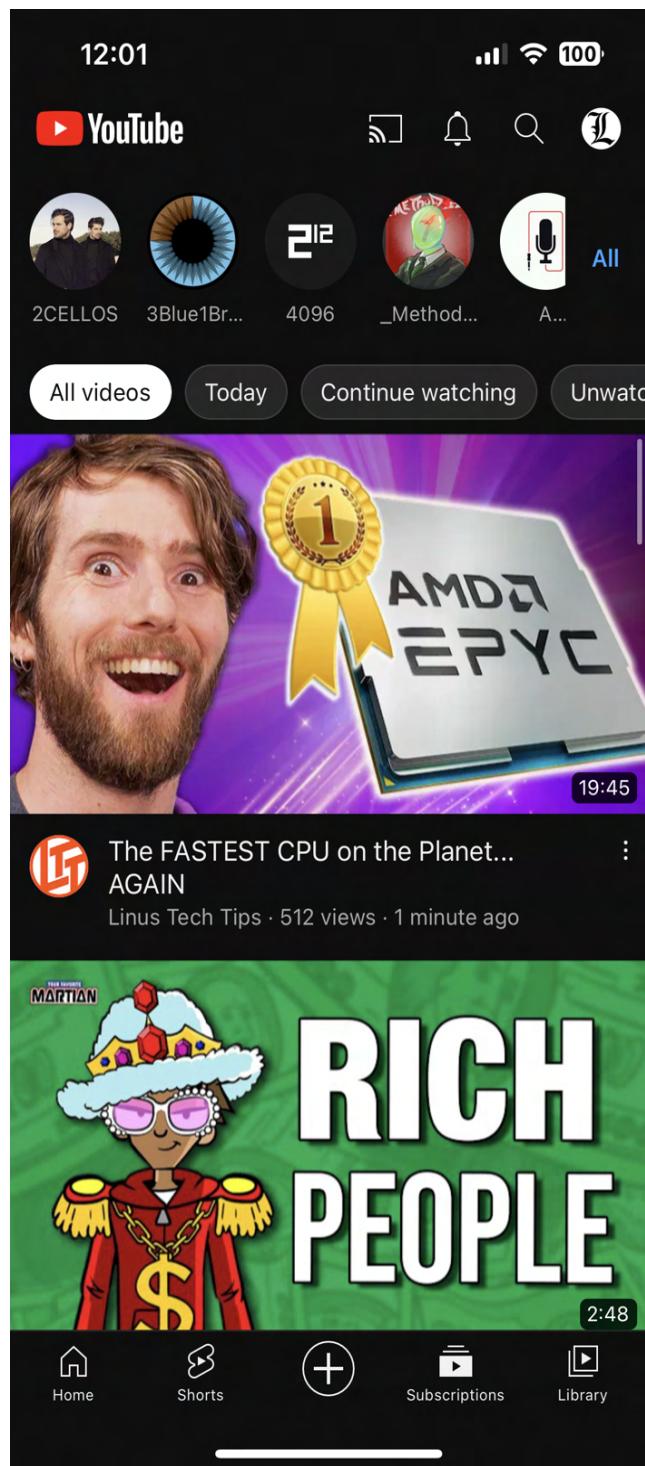
This section examines the ways in which the YouTube algorithm can be utilized for the purpose of discovering new ideas and insights by recommending videos based on specified interests. Our experiment finds that by specifying interests, such as FIRST robotics videos, the algorithm will recommend videos from small and obscure teams from across the globe that otherwise would not have been seen. The videos often have almost no views, almost no comments, almost no likes or dislikes, were posted within the past week, were posted by channels that almost nobody has ever heard of, and are about incredibly niche topics (e.g. short videos of FRC prototype mechanisms from 2023). The research also includes an examination of Mr. Beast's advice to YouTubers and how it can be used to the viewer's advantage, as well as strategies that other teams can replicate to game the algorithm. We include examples of oddities discovered through the process, such as an increased recommendation of low quality gaming videos and videos about guns, and ways the viewer can influence the algorithm intentionally.

To start off, we need to establish a baseline of how YouTube recommends videos, so let me introduce you to The Algorithm. The Algorithm is a machine learning based system created by YouTube that only ever shows you the most relevant content to you, with the sole goal of increasing your watch time and interaction with the platform. It chooses which videos to recommend to you based on how likely you are to watch an entire video, whether you like or dislike a video, what videos you click on in the side bar after your current video, whether you comment on the video, and many more subtle ways. With over 500 hours of video uploaded to YouTube every single minute, The Algorithm needs to be very good at recognizing your tastes so that you will only ever engage with content that is relevant to you.

By default one of our team member's personal algorithm consisted of popular tech channels, comedy music videos, as well as various learning channels:



Example 1



Example 2

Everybody has their own personal algorithm, and the recommended feed is the coveted position of many YouTubers since that is how virality occurs. Mr. Beast, a YouTuber well renowned for his over the top viral videos and one of the most subscribed YouTube channels, is very open about the techniques he uses. Beyond simply "Make good videos and thumbnails", he mentions things that creators should be optimizing for, such as:

- Making a video that's interesting enough for people to watch all the way through

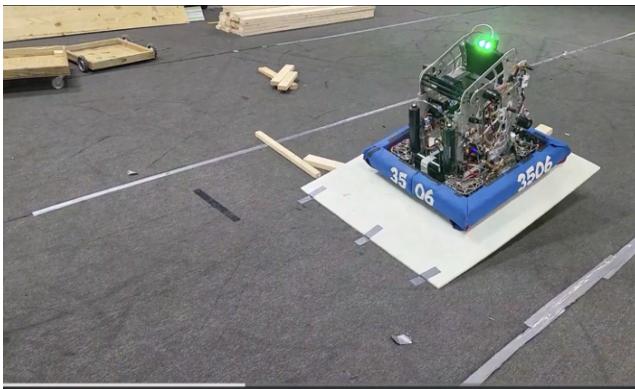
- Making a thumbnail that makes people want to click on the video in the recommendations bar
- Having a title that is short enough that it doesn't get cut off on small screens
- Having an interesting title

By intentionally searching for FRC mechanism prototype videos and applying Mr. Beast's techniques as a viewer, we can effectively manipulate the algorithm to surface relevant and high-quality content. This allows us to easily find and learn from other teams' designs and innovations in the field.

More explicitly, for the first day or so of the experiment we searched for prototype videos on YouTube. Often times they wouldn't be exactly what we were looking for, being videos of previous years' prototypes, but they were close enough for our purposes. We would:

- Watch the entire video, even if it was only necessary to watch the first couple of seconds to understand what they were doing
- Like the video
- Share the video (We have a channel in our discord exclusively to be a place to dump these video shares)
- If ANY of the videos that showed up in the recommended feed were a combination of being from the same channel, were only a couple minutes long, or mentioned robots/FRC/etc. adjacent things, click it.
  - Match replays should not be clicked on, otherwise we will not get recommended videos of close ups of WIP mechanisms
  - Documentaries should not be clicked on because they often do not contain the detail necessary for prototyping mechanisms
  - Robot reveals should not be clicked on because they are more often about the flash and fanfare than the technical details of how something works
- Any surveys that show up regarding FRC videos asking how much you enjoyed a video, answer with the highest rating
- Anything that does not fall into the above categories you need to either tap the "Do not recommend channel" button or the "Do not recommend this content" button

Wash, rinse, and repeat. You only need to do this a little bit each day for a couple days before The Algorithm takes over and begins to serve you ridiculously niche videos. Videos from teams that have zero virality or traction, have next to no views, have almost zero likes, and you have never watched will be served to you within hours or days of them being posted. As long as you keep watching these niche videos, they will keep being served to you:



Amazon Shopping Ad · 4.8★ FREE Install

### Charging Station with Normal Swerve lock sliding

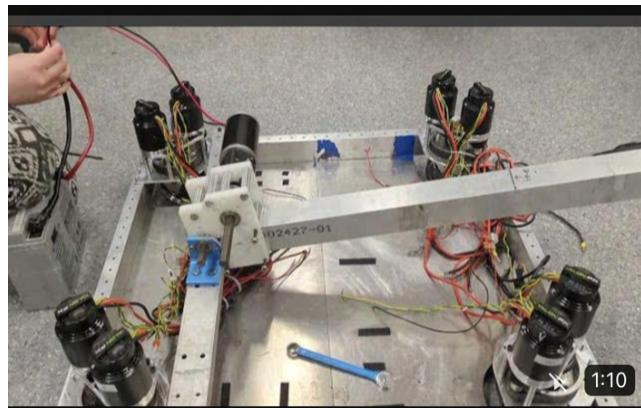
299 views 7d ago ...more

Try YouTube Kids An app made just for kids Open app

YETI Robotics 3506 293 Subscribe

Like 6 | Dislike Share Download Save

299 Views, 7 days ago, 293 Subscribers, 6 Likes



FRC 2023 team 3134 week 1 update part 1

FRC Teams 3134 and 3275 · 391 views · 4 days ago

391 Views, 4 days ago

One oddity that occurred while running this experiment is that along with an increase in niche robotics videos, there was a noticeable uptick in low quality gaming content and videos about guns being recommended, though this subsided after a week or so. We're assuming on YouTube's backend the categories of "robots", "video games", and "guns" must be semantically connected and draw similar audiences, but it's difficult to tell for certain without more data. After a few more days of signaling to The Algorithm that we were not interested in these kinds of videos, it stopped recommending them.

# Using ChatGPT for Fun and Profit Donations

ChatGPT ([chat.openai.com](https://chat.openai.com)) is an AI chat bot released by OpenAI in late November of 2022. We could describe it for you, but we think that it might be best if ChatGPT describes itself:

The screenshot shows a conversation interface. A user prompt is at the top: "How would you describe ChatGPT, an advanced chat bot by OpenAI?". Below it is a response from ChatGPT: "ChatGPT is a large language model developed by OpenAI that can generate human-like text. It is trained on a vast amount of internet data, allowing it to understand and respond to a wide range of topics and questions. It can be used for a variety of applications such as language translation, text summarization, and conversation. Additionally, it can also be fine-tuned for specific use cases such as customer service, content generation and more." There are upvote and downvote buttons to the right of the response.

*ChatGPT describing itself*

That response was not pre-programmed -- it is a response that is generated based on the prompt that was given to it by a human. Common use cases for ChatGPT are using it as an assistant for writing poetry, creating fictional short stories, and SEO content. We, as engineers, have figured out how to get it to help us with short scripts, debugging, providing new students with problem-specific context that they otherwise wouldn't have had, and even getting it to write sponsorship letters. We show how to use conversational AI to go further than either human or computer can go individually.

Many people are of the opinion that ChatGPT is either completely useless or is going to take over all of our jobs. Reality, as it often is, is somewhere in the middle. ChatGPT is a tool, not for automating everything, but for automating boilerplate operations such as a first draft or writing the general structure of a program. The part that the human plays in this instance is more of a teacher and an editor. ChatGPT was only trained on text, so if we want it to properly help us we need to give it as much context about our problem as possible. If the problem gets too big in scope, ChatGPT can't remember all of the context that it needs to be effective. ChatGPT doesn't always get it right, as you will soon see, but the speed with which it synthesizes and generates content far outweighs those deficits.

## Using ChatGPT to get Sponsors

Using ChatGPT to get sponsors is fairly straightforward. When creating a sponsorship request letter, we know what the format we are trying to adhere to is, and we are just using ChatGPT to get there. In this case, we want our format to be as follows:

- Paragraph 1: Who we are
- Paragraph 2: What we want
- Paragraph 3: What's in it for you

We will be cutting out pieces of this conversation for the sake of brevity, but all the main ideas will be shown. When making the first paragraph, we start off with something easy:

Human:

Give me a 5 sentence description of what FIRST robotics is

ChatGPT:

FIRST Robotics is a student-centered program that combines science, technology, engineering, and math (STEM) education with competitive robotics. Students in grades K-12 participate in teams, design, build, and compete with robots they build using a standard kit of parts and a common set of rules. The program aims to inspire students to pursue careers in STEM fields, and to develop valuable life skills such as problem-solving, teamwork, and leadership. The program includes local, regional, and international competitions, with the ultimate goal of competing in the FIRST Robotics Competition World Championship. The program is organized and run by the non-profit organization FIRST (For Inspiration and Recognition of Science and Technology).

Looks like a good start. Let's start customizing it to our team, Redbird Robotics:

Human:

Rewrite it to also include a mention of "Redbird Robotics", a high school robotics team run out of De Pere High School

ChatGPT:

FIRST Robotics is a student-centered program that combines science, technology, engineering, and math (STEM) education with competitive robotics. One of the teams participating in the program is Redbird Robotics, a high school robotics team run out of De Pere High School. Students in grades.....

Human:

Redbird Robotics should be mentioned from the first person, as in "we are Redbird Robotics"

ChatGPT:

FIRST Robotics is a student-centered program that combines science, technology, engineering, and math (STEM) education with competitive robotics. We, Redbird Robotics, are a high school robotics team run out of De Pere High School, and one of the teams participating in the program. Students in grades.....

Even minor edits such as changing the personal pronouns are important to specify, since it is context that is added to the conversation which ChatGPT can use later. If this step was not taken, the context and sentence structure would be different and would drift away from the intended feelings we are trying to evoke with our intended message. Moving onto the second paragraph, we asked it to write a paragraph about everything we wanted and what it would go towards:

Human:

Write a paragraph to a sponsor about what we want. Specifically mention:

- We are looking for \$500 in funds
- Our season runs until the end of March
- The money goes towards providing our students with reduced pricing for competitions
- Purchasing an advanced drive base
- Supplies for our pits at competition
- Any in-kind donations of food for our Saturday meetings

ChatGPT:

We are reaching out to you today as we are currently seeking support for our team in the form of donations. Specifically, we are looking to raise \$500 to help cover costs for our upcoming build season, which runs until the end of March. The money you would donate could potentially go towards providing our students with reduced pricing for competitions, purchasing an advanced drive base for our students to learn on for next year's bot, and purchasing supplies for our pit at competitions. We would also greatly appreciate any in-kind donations of food for our Saturday meetings during the build season. Your support would be greatly appreciated and would go a long way in helping our team succeed. Thank you for your consideration.

Human:

Reword the first sentence so that it sounds less like a corporate email

ChatGPT:

We hope this letter finds you well. We are reaching out to you today as a local high school robotics team, Redbird Robotics, is currently seeking support in the form of donations. Specifically, we are looking to raise \$500 to help cover costs for our upcoming build season, which runs until the end of March. The money you would donate could potentially go towards providing our students with reduced pricing for competitions, purchasing an advanced drive base for our students to learn on for next year's bot, and purchasing supplies for our pit at competitions. We would also greatly appreciate any in-kind donations of food for our Saturday meetings during the build season. Your support would be greatly appreciated and would go a long way in helping our team succeed. Thank you for your consideration.

As we progressed in creating our sponsorship, ChatGPT must have gotten the impression that it should sound like a formal email between colleagues. Simply mentioning that the letter should be written in the style of something else is enough for it to understand its mistake and correct it. The third paragraph was much easier, in part because we had a bullet point list of things we wanted ChatGPT to address:

Human:

Now write a new paragraph. It should include these things:

What they get out of it:

- A sponsorship mention/logo placement on Brown County FIRST Robotics' website (still in the making)
- Weekly/Semi-Weekly updates on the robot's progress during our build season
- A shout out in that email that we send out to sponsors
- Your logo will be put on our robot so that people can see it when we show it off at events around the community (such as Rock the Dock)

ChatGPT:

We understand the importance of supporting local businesses and we would like to extend our gratitude by offering you a number of benefits for your support. By becoming a sponsor, your company will receive a sponsorship mention and logo placement on the Brown County FIRST Robotics' website which is currently under construction. Furthermore, you will receive weekly/semi-weekly updates on the robot's progress during our build season, so you can see the impact of your contribution. We will also give you a shout out in the email that we send out to sponsors, to show our appreciation for your support. Lastly, your logo will be placed on our robot, so people can see it when we show off the robot at events around the community such as Rock the Dock.

From this point we had what we needed to put together a sponsorship request. These were not ready to send out as is, but they were 90% of the way there. In the end, with some editing, we came up with this:

FIRST Robotics is a student-centered program that combines science, technology, engineering, and math (STEM) education with competitive robotics. We, Redbird Robotics, are a high school robotics team run out of De Pere High School, and one of the teams participating in the program. Students in grades K-12 participate in teams, design, build, and compete with robots they build using a common set of rules. The program aims to inspire students to pursue careers in STEM fields, and to develop valuable life skills such as problem-solving, teamwork, and leadership. The program includes regional and international competitions, with the ultimate goal of competing in the FIRST Robotics Competition World Championship.

We are reaching out to you today as we are seeking support for our team in the form of donations. Specifically, we are looking to raise \$500 to help cover costs for our build season, which is currently in progress and runs until the end of March. The money you would donate would go towards providing our students with reduced pricing for competitions, purchasing an advanced drive base for our students to learn on for next year's bot, and purchasing supplies for competitions. We would also greatly appreciate any in-kind donations of food for our Saturday meetings during the build season. Your support would be greatly appreciated and would go a long way in helping our team succeed.

We understand the importance of supporting local businesses and we would like to extend our gratitude by offering you a number of benefits for your support. By becoming a sponsor, your company will receive a sponsorship mention and logo placement on the

Brown County FIRST Robotics' website which is currently under construction. Furthermore, you will receive weekly/semi-weekly updates on the robot's progress during our build season, so you can see the impact of your contribution. We will also give you a shout out in the email that we send out to sponsors, to show our appreciation for your support. Lastly, your logo will be placed on our robot, so people can see it when we show off the robot at events around the community such as Rock the Dock.

## Using ChatGPT to Maintain Sponsor Relations

Getting sponsorships is a critical aspect to any successful FRC team. Without the support of the industries around us, many teams would not exist. As such, it's imperative that teams not only bring on new sponsors to help grow teams and increase the presence of STEM, but also to maintain the relationships with the sponsors they have. The problem is that with everything that goes on during the build season, it can sometimes be difficult to effectively and clearly communicate everything that a robotics team is doing with that support.

One way we have tried to mitigate this issue is by providing regular build season updates to our sponsors, offering a more granular look at our progress each week. At the end of each day, we go around and get updates from each sub-team (Electrical / Mechanical / CAD / Programming). At the end of each week, we get a list that looks something like this:

### Programming

- Working on fixing bugs with the new motors
- working on updating hardware
- Got REV motors working
- Got all hardware updated to latest drivers
- 2023 Software WPILib installed on all computers
- We got motor ramping working so that they have a smooth acceleration and deceleration curve
- Familiarizing ourselves with the newest WPILib updates
- Worked on CAN IDs
- Set up a new industrial camera to test our distortion correction algorithm
- The first version of this year's drive code is ready to be tested on last year's robot, pending electrical issues being solved
- Got pneumatics to work on our test board
- rewired, added, and reorganized parts on the test board

### CAD

- Collected parts for the butterfly module
- Motor simulation for NEO's posted in mech chat
- Come up with an idea for a manipulator prototype
- Started working on the frame

- sides and back done
- tomorrow to CAD the front and start on holes
- Figured out CAD constraints for assemblies for the frame
- back of frame CADed
- Most of butterfly drive CADed

## Electrical

- Swapped out CAN bus
- Created and sent electrical parts to be purchased
- Halfway through swapping out Falcons for NEOs
- Organized parts and components on last year's robot
- Finished swapping out last year's Falcons for this year's NEOs
- Added NEO controllers

That works fine for a checklist, but it does a pretty poor job of telling a story to sponsors who have not been at our weekly meetings. For the storytelling aspect of our progress, we turn to ChatGPT. As a reminder, ChatGPT is a large language model that was only trained on text data, so it has no context for telling the kind of story we are looking for. In the previous section we started from scratch and worked towards a specific format. Using ChatGPT to maintain sponsors works in the exact opposite way -- we start with a list of items and are using that to work towards something that sounds nice, but we have no idea what the structure will look like. We must first give it some context:

Human:

I am a part of an FRC team called Redbird Robotics. It's a part of the global organization called FIRST Robotics. We have a list of everything we've done this week here:

(Checklist goes here)

Cluster the list of completed tasks by similarity. Present the clusters as a short list. Feel free to combine concepts from different points in the list if it makes the list shorter, as long as they are related ideas.

We cluster the ideas here to try to combine any similar ideas and events that have occurred throughout the week. If we don't do this, the AI has a propensity to repeat similar information that just generally does not sound right. As a response we get:

ChatGPT:

1. Programming
  - Fixed bugs with new motors
  - Updated hardware and drivers
  - Installed WPILib on all computers

- Familiarized with newest updates
- Motor ramping and CAN ID work
- Tested new drive code on old robot
- Tested pneumatics on test board

## 2. CAD

- Collected parts for butterfly module
- Motor simulation for NEO's posted
- Came up with manipulator prototype idea
- Started frame design and assembly
- Butterfly drive CAD work

## 3. Electrical

- Swapped out CAN bus
- Purchased electrical parts
- Swapped out Falcons for NEOs
- Organized components on old robot
- Added NEO controllers

Looks pretty good. Let's start telling a story!

Human:

Now convert these clusters into paragraphs

ChatGPT:

Our FRC team, Redbird Robotics, had a productive week with a focus on programming. We fixed bugs with the new motors and updated our hardware and drivers. To ensure that we were all up to date, we installed WPILib on all of our computers and familiarized ourselves with the newest updates. We also worked on motor ramping and CAN ID assignments. Our first version of this year's drive code is ready to be tested on last year's robot, and we were happy to find that we got pneumatics to work on our test board.

In terms of CAD, we had a number of exciting developments. We collected the necessary parts for the butterfly module and performed a motor simulation for NEO's. We came up with an idea for a manipulator prototype and started to work on the frame. By the end of the week, we had completed the sides and back of the frame and were preparing to CAD the front and start on the holes. We also figured out the CAD constraints for the assembly of the frame, and we have made significant progress on the butterfly drive CAD.

Finally, our electrical team has been busy this week. We swapped out the CAN bus, and we created and sent electrical parts to be purchased. We were also able to successfully swap out the Falcons for NEOs and add the NEO controllers. In addition, we organized the parts

and components on last year's robot. It was a great week for our team, and we are excited to continue to make progress on all fronts.

Not bad for a first try, but some parts of it are a little off. First of all, if we're presenting information to companies who already sponsor us, we don't need to introduce ourselves. We also didn't "collect necessary parts for a butterfly module" -- we downloaded 3D models from vendor websites to make a CAD of our butterfly module. Another major point is that we didn't "create" any electrical parts -- all of our electronics are COTS. It's assuming context that it doesn't have, so we have to help it out:

Human:

More context:

- The people this is being sent to are already aware of us, so we don't need to introduce ourselves as "Redbird Robotics"
- We did not collect parts for a butterfly module, we downloaded 3D CAD parts from manufacturers' websites to use in our CAD assemblies
- Our "simulation" is entirely numerical in a Google sheet, not something like static analysis. What you have currently implies static analysis. Rework it so that it reflects what actually happened better.
- Specify that the holes that we will be CADing are for riveting the frame together
- We did not create any electrical parts -- we just sent electrical parts to be purchased

By specifying it as a list, we make it easier for ChatGPT to separate the concepts we want it to understand. It responds with:

ChatGPT:

This week, we had a strong focus on programming, fixing bugs with our new motors and updating our hardware and drivers. Our team members installed WPILib on all of our computers and familiarized ourselves with the newest updates. We worked on motor ramping and CAN ID assignments, and we are excited to test the first version of this year's drive code on our old robot. Additionally, we made progress with pneumatics and were able to get them to work on our test board.

In terms of CAD, we downloaded 3D CAD parts from manufacturers' websites to use in our CAD assemblies. We simulated the motors numerically in a Google Sheet and came up with an idea for a manipulator prototype. Our team started working on the frame by completing the sides and back of the frame and are in the process of preparing to CAD the front and holes for riveting the frame together. We also made significant progress on the butterfly drive CAD.

Our electrical team accomplished quite a bit this week as well. We swapped out the CAN bus and sent electrical parts to be purchased. We were able to successfully swap out the Falcons for NEOs and added the NEO controllers. Additionally, we organized the parts and

components on last year's robot. It was a great week for our team, and we are excited to continue making progress on all fronts.

Much better! It's not perfect, but it's 90% of the way there. From here we can make simple corrections, add some images that we've taken throughout the week, and send the email out.

But it kind of begs the question, why bother with offering more context at all? If the first set of paragraphs were close enough, why not just edit them by hand? While the latter may seem like the easier option, providing additional context can actually be more efficient in the long run. Critiquing a piece of writing is often easier than fixing it, especially when dealing with poorly written text. It's easy to identify what's wrong with the text, but transitioning from criticism to making specific improvements can be challenging. This is where ChatGPT can be beneficial. By providing extra context to ChatGPT, we can simplify the process and get a more effective outcome. Rather than spending time and effort fixing the text ourselves, we can rely on ChatGPT to make the necessary changes based on the provided context. This not only saves time but can also lead to better results.

## Automatically Finding Camera Device IDs on Linux.

An important part of calibrating our robot's cameras to fix lens distortion is being able to match up their calibration files with a specified camera, since even between two copies of the same camera, their manufacturing defects can cause distortions from reality. These distortions only amplify when an object (like an AprilTag) are at the edge of a camera's frame or are far away. One issue we were running into was that we were not able to match up a camera with its configuration file since the camera's device ID would change on every boot sequence and every re-plug of the camera device. We needed a reliable way to match the camera configuration file with its associated camera, so we turned to ChatGPT. We explained our problem and asked for a bash script for our solution:

```
#!/bin/bash

# Get a list of all /dev/video* devices
devices=$(ls /dev/video*)

# Iterate through the list of devices
for device in "${devices[@]}"
do
    # Get the ID_SERIAL value for the current device
    serial=$(udevadm info -q all -n $device | grep ID_SERIAL | awk '{print $2}')

    # Print the device and its ID_SERIAL value
    echo "$device : $serial"
done
```

While this initially seemed like it was giving us errors and was detecting video devices that we were not able to even access, we noticed that it also mentioned this at the end of its response:

Please note that the `ID_SERIAL` key may not be available in some cases, in this case the script will return nothing for that device.

The script was still returning devices that it wasn't supposed to, so we asked it to handle for instances where it wasn't able to find `ID_SERIAL`:

```
#!/bin/bash

# Get a list of all /dev/video* devices
devices=$(ls /dev/video*)

# Iterate through the list of devices
for device in "${devices[@]}"
do
    # Get the ID_SERIAL value for the current device
    serial=$(udevadm info -q all -n $device | grep ID_SERIAL | awk '{print $2}')

    # If ID_SERIAL value is not available
    if [ -z "$serial" ]; then
        echo "$device : Not available"
    else
        # Print the device and its ID_SERIAL value
        echo "$device : $serial"
    fi
done
```

This script gave us a number that was different for every camera and stayed the same between computers. From here we used the subprocess module in Python to parse the output of `udevadm info -q all -n $device | grep ID_SERIAL | awk '{print $2}'` and integrate the resulting information into our existing scripts.

Unfortunately the cheap cameras we ended up working with are different from the cameras we started off with and do not have serial IDs that we can reference. We ended up using a hardcoded path to their physical port and just made sure to plug in each camera in the same spot every time. An example of it looks like this:

```
/dev/v41/by-path/platform-fd50000.pcie-pci-0000:01:00.0-usb-0:1.4.1.0-video-index0
```



*Modern problems require modern solutions. Our method for making sure we plug the cameras into the correct port every time.*

## Using ChatGPT to Write an Obscenely Huge Engineering Document

At this point you are probably wondering if we used ChatGPT to write this document. The short answer is "Yes", though the full answer is a little more complicated. It's easy to get the impression that we could have just used ChatGPT to write this entire document. While we won't say it's impossible, it's definitely impractical. The trend among all examples in this document is that they are diverse in application but are extremely limited in scope:

- The "Getting a Sponsor" letter requires follow up by an actual human to drive the deal home (such as a call or a meeting)
- The Sponsorship Update letter is a once and done kind of thing and only needs to make sense in the context of what happened that week, not necessarily the entire season
- The camera script doesn't take into account our entire software pipeline, nor is it a drop in solution to our problem (even if we had better cameras)

None of these push a production solution across the finish line. Varying amounts of work are still required to turn these into something that actually progresses our team. Likewise, due to the scope of this document it is not possible to use an AI to write it as coherently as it is with current technology. We could have split the sections up piecemeal like with functions, methods, and classes in programs, but even then you would run into issues of consistency between the separate sections. Some sections would make subtle false claims that would be easy to miss just by skimming through it or would flat out contradict other sections. The technology to automatically do something on this scale simply does not exist yet.

With that established, we'd like to mention some places within this document that we've used ChatGPT as a writing and visual aid.

## Synthesizing Student Scripts

To achieve a document as extensive and clear as this it takes a team effort. For each student's specialty we asked them to write:

- What were you trying to accomplish or what was your pain point?
- Why were the currently available solutions not good enough?
- Talk about the process of building the thing you built.
- What did you learn over the course of building this thing? Anything surprising or insightful?

We are engineers, not authors, so naturally not all of the explanations submitted were presentation worthy. They would often come in looking something like this:

The board began as the brainchild of me and a mentor during a meeting somewhat early in the build season. The goal was to incorporate function and aesthetic. We wanted the board to be easily accessible but still allow for proper protection of our electronics during the matches. The protection aspect was very important for Rapid React as shooting projectiles was the main focus of the game. Quite simply the easiest way to protect the electronics from flying balls was to completely close them off and have them completely facing away from anything that could hit them. Solving this problem was easy enough to sit the board facing "into" the robot. How would it be accessible? Once again an easy problem to solve, put the board on a hinge. This allowed the board to be flipped down when we needed to access the wires and flip up when it needed to be protected during matches, testing, etc. The basic board was perfect, the issue came in with the actual wiring of the electronics on the board. The wiring was sloppy and poorly planned leading to a rat's nest of wires and an absolute mess. This led to troubleshooting troubles, disconnect issues, and overall grossness. This year is different. First we planned the wiring diagram using a program called mermaid. This let us write out what we wanted and gave us a visual of what our wiring would look like. Next we went to CAD and designed the physical board. After the board was designed and the holes were placed in the correct space we exported the file onto a USB thumb drive, transferred them to our laser cutter and laser cut our mounting holes for all of the electronics on the board. Once that was complete we proceeded to mount all electronics save the raspberry pi. The final steps were to plan wiring, mount zip tie mounts to hold all wiring to the board and finally run wiring. Finally, we mounted the boards to the robot and wired in the robot mounted electronics such as motors, cameras, and solenoids.

or, if just talking about it was easier, we would listen to them ramble and take very disorganized notes:

- We did a time series simulation of a robot acceleration event
  - If you have an imaginary robot with 4 NEO motors with an imaginary switch
  - at t=0 it flips on and the robot accelerates from standstill and drive straight

- What does the velocity and position of the robot look like
- ultimate goal of how quickly can we make it cross the field
- Crossing the field is 3 stages
  - Speed up, constant, slow down
    - up and down are assumed to be the same
    - At a constant speed
      - pos = vel x time
- Motor
  - We looked at torque curve data sheet
    - we can derive that if a motor is driving at a certain RPM, it will have a specified torque
      - That torque makes the robot accel at a certain rate, as it speeds up the wheel speed goes up, the torque speed goes down
        - This continues until the robot balances out at a constant speed
        - Excel spreadsheet runs that loop for 10 seconds
          - We can see where it levels off with the speed
          - Once we see where it levels off we can see our 3 stages
  - 1 graph is speed of robot over time, 1 is position over time
    - When speed levels off, we take a look at the position of our robot, so we take that distance off on either end, the remaining distance is a constant velocity
      - Add up the time of all 3 sections to determine the total time to get across the field
        - From here select gear ratios from a list that minimizes that time across the field
  - Compares the speed of the motor vs the speed of the wheel

Using our clustering technique we talked about earlier, we can transform these into something like this:

Human:

I have an unorganized bullet pointed list of something one of our team members did this season on our robotics team:

(the bullet pointed list above)

Cluster it into similar ideas and present them as a bullet pointed list

ChatGPT:

Here's a possible way to cluster the ideas in the original bullet pointed list:

- Time series simulation of robot acceleration event:
  - Imaginary robot with 4 NEO motors and an imaginary switch
  - Goal of quickly crossing the field
  - 3 stages of crossing the field: speed up, constant, slow down

- Position and velocity of robot over time
- Motor:
  - Torque curve data sheet
  - Deriving torque and acceleration from RPM and motor specifications
  - Excel spreadsheet simulation of acceleration and leveling off at constant speed
- Graphs:
  - Speed of robot over time
  - Position of robot over time
- Analysis:
  - Determining time to cross the field from the 3 stages
  - Selecting gear ratios to minimize time
- Comparison:
  - Comparing motor speed and wheel speed

And here's the same information presented as a new bullet pointed list with these clusters:

- Time series simulation of robot acceleration event:
  - Create imaginary robot with 4 NEO motors and an imaginary switch
  - Goal: quickly cross the field
  - Divide crossing into 3 stages: speed up, constant, slow down
  - Plot position and velocity of robot over time
- Motor:
  - Use torque curve data sheet to derive acceleration from RPM and motor specifications
  - Simulate acceleration and leveling off at constant speed with Excel spreadsheet
- Graphs:
  - Plot speed of robot over time
  - Plot position of robot over time
- Analysis:
  - Determine time to cross the field from the 3 stages
  - Select gear ratios to minimize time
- Comparison:
  - Compare motor speed and wheel speed
- Clustering with NEO motors
- Matplotlib graphs
- References in the documentation

Human:

Expand the clustered data points into paragraphs that explain what happened.

ChatGPT:

Sure! Here are expanded explanations of each cluster of ideas:

Time series simulation of robot acceleration event:

Our robotics team conducted a time series simulation of a robot acceleration event with the ultimate goal of determining how quickly we could make the robot cross the field. To do this, we created an imaginary robot with 4 NEO motors and an imaginary switch. We then divided the process of crossing the field into three stages: speed up, constant speed, and slow down. We plotted the position and velocity of the robot over time to determine how the robot would move during each stage.

Motor:

To determine the acceleration of the robot, we looked at the torque curve data sheet and derived torque and acceleration from RPM and motor specifications. We then used an Excel spreadsheet to simulate the acceleration of the robot and the leveling off at a constant speed. This helped us understand how the torque made the robot accelerate at a certain rate, and how as the robot sped up, the wheel speed increased while the torque speed decreased. The simulation ran for 10 seconds, and we could see where the robot leveled off with the speed.

Graphs:

We created two graphs to visualize the motion of the robot. The first graph plotted the speed of the robot over time, and the second graph plotted the position of the robot over time. When the speed leveled off, we could take a look at the position of the robot and calculate the distance covered. We subtracted this distance on either end, and the remaining distance was covered at a constant velocity.

Analysis:

To determine the time it would take the robot to cross the field, we added up the time of all three sections: speed up, constant speed, and slow down. We then selected gear ratios that would minimize the time it took for the robot to cross the field. This analysis helped us optimize the robot's speed and acceleration, making it as efficient as possible.

Comparison:

To ensure that the motor was running optimally, we compared the speed of the motor with the speed of the wheel. By comparing these two speeds, we could determine if the motor was providing enough power to the wheel to make it move at the desired speed. This comparison helped us make sure that the robot was performing as expected.

In summary, our team used a time series simulation to study the acceleration and motion of a robot with 4 NEO motors. By analyzing the torque and acceleration of the motor, and plotting the position and velocity of the robot over time, we were able to optimize the

robot's speed and acceleration to minimize the time it took to cross the field. We also compared the speed of the motor and wheel to make sure the robot was running as expected.

ChatGPT is able to find the patterns in the data and organize it into a much better presentation. There are some inaccuracies, but we have both the original information as well as the person who gathered the data to understand the broader context to fix it. If you look at our section on NEO motor curves, you'll find that we ended up rewording and reorganizing nearly the entire output of ChatGPT. In this context, ChatGPT functioned more as an inspiration for how the data could be talked about and presented.

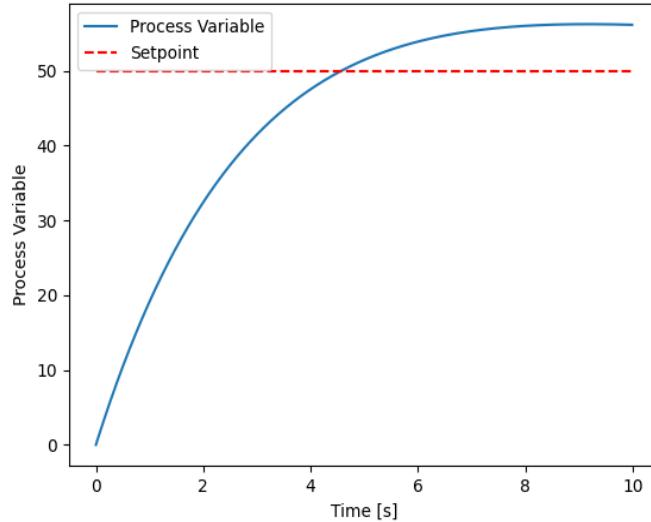
## Using ChatGPT to Create Visual Aids

Throughout this document we have provided many visual aids. In most cases we could pull from pictures taken throughout the season. In some select few cases however, we needed specific graphs or diagrams to help illustrate an idea. Rather than grabbing a random image from the internet (which can come with its own copyright issues), we decided to use ChatGPT to create our own graphs.

In our section about our autobalancer we wanted some visual explanations of how a PID loop works. There is a Python library called `matplotlib` which allows users to plot lines on a graph, so we told it:

Human:

Write me a python script that generates a matplotlib example of a PID loop



*Generated image*

It's on the right track of what we were looking for, but not quite there. I give it more context:

Human:

Modify the code so that:

- The Y axis goes from `0` degrees to `180` degrees
- The "Process Variable" should be renamed to the "Robot Angle"
- `y = [90]` is `75` instead
- Change the PID values so that they wobble a little before settling at the setpoint
- It saves the generated graph

and it provides me with this script:

```
import matplotlib.pyplot as plt
import numpy as np

# Define the setpoint and initial conditions
setpoint = 90
y = [75]
t = [0]

# Define the PID parameters
kp = 0.9
ki = 0.01
kd = 0.1

# Define the time step and simulation time
dt = 0.01
sim_time = 10
```

```

# Define the error
error = [setpoint - y[0]]

# Define the integral and derivative terms
integral = 0
derivative = 0

# Loop for the simulation time
for i in np.arange(dt, sim_time, dt):
    # Calculate the error
    error.append(setpoint - y[-1])
    integral += error[-1] * dt
    derivative = (error[-1] - error[-2]) / dt

    # Calculate the control signal
    control = kp * error[-1] + ki * integral + kd * derivative

    # Update the state
    y.append(y[-1] + control * dt)
    t.append(t[-1] + dt)

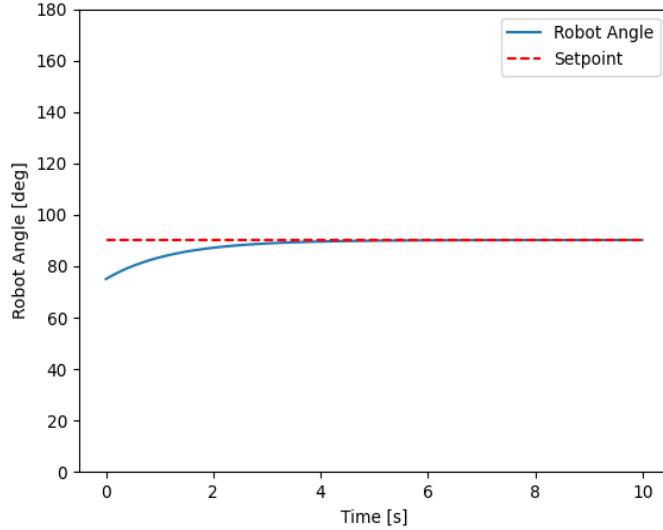
# Plot the results
plt.ylim(0, 180)
plt.plot(t, y, label='Robot Angle')
plt.plot(t, [setpoint]*len(t), 'r--', label='Setpoint')
plt.xlabel('Time [s]')
plt.ylabel('Robot Angle [deg]')
plt.legend()

# Save the plot
plt.savefig('pid_loop.png')

plt.show()

```

which generates a graph that is close enough to what we want so that we can start playing around with PID values until we get something that we like:



*ChatGPT generated good PID loop*

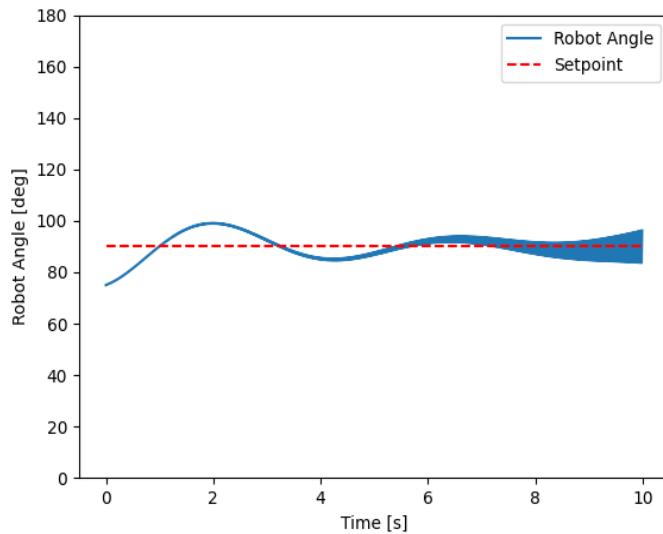
For the bad PID loop, something interesting happens. We tell it:

Human:

Modify the code so that:

1. The robot angle violently oscillates around the setpoint and never settles

And it created this image:



*ChatGPT generated bad PID loop*

The reason why this in particular is interesting is that in order for ChatGPT to create this, it had to understand that in the context of a PID loop, the word "violently" simply means changing a couple of numbers. There are other contexts where the word "violently" means much different things, many of which ChatGPT knows about but is restrained from talking about. But within

the context of a PID loop, it understood how to modify the numbers in just the right way so as to achieve the concept of "violently oscillating".

## Documentation Cleanup

At the end of this paper we provide significant portions of our team's personal documentation (all of which is available at the github link mentioned on the cover). In that documentation we have hyperlinks to more in depth resources where applicable. This is very useful for a web browsing experience, but hyperlinks don't really work on a physical sheet of paper. To remedy this, we asked ChatGPT for help:

Human:

Write me a python script that loops through the lines in a markdown file that removes all hyperlinks and adds them as references at the bottom of the file

As is the pattern, it kind of gets things right except for a few things. The hyperlinks at this point looked like `you can view extra documentation here [27 documentation here | <url> ]`. The number wasn't the reference's position relative to the position of all the other references, it was the line number the reference was found on. The references at the end of the page were also in the format of `[27] documentation here <url>`. It was also citing titles of sections as references. The end result is something that looks kind of disorganized and hard to read, so we make some requests for changes:

Human:

Modify the script so that:

- It reformats the hyperlinks to be like previously hyperlinked text [0]
- Modify the reference so that it doesn't include the hyperlink text
- The index is the index of hyperlinks found so far, not the line number they were found on
- Only grabs hyperlinks that start with http

The final result is a script that converts markdown hyperlinks to references that are more easily accessible offline:

```
import re

# Define the pattern for identifying hyperlinks in markdown
hyperlink_pattern = re.compile(r'\[(.*?)\]\((http.*?)\)')

# Open the markdown file for reading and store the lines in a list
with open('your_markdown_file.md', 'r') as f:
    lines = f.readlines()

# Create an empty list to store the references and a counter for the number of hyperlinks
# found
```

```

references = []
counter = 0

# Loop through each line in the markdown file
for line in lines:
    # Check if the line contains a hyperlink starting with "http"
    if hyperlink_pattern.search(line):
        # Extract the hyperlink text and URL
        hyperlink_text = hyperlink_pattern.search(line).group(1)
        hyperlink_url = hyperlink_pattern.search(line).group(2)
        # Add the reference to the list of references
        counter += 1
        reference = f"[{counter}] {hyperlink_url}\n"
        references.append(reference)
        # Replace the hyperlink in the original line with the reference number
        lines[lines.index(line)] = hyperlink_pattern.sub(f"{hyperlink_text} [{counter}]",
                                                       line)

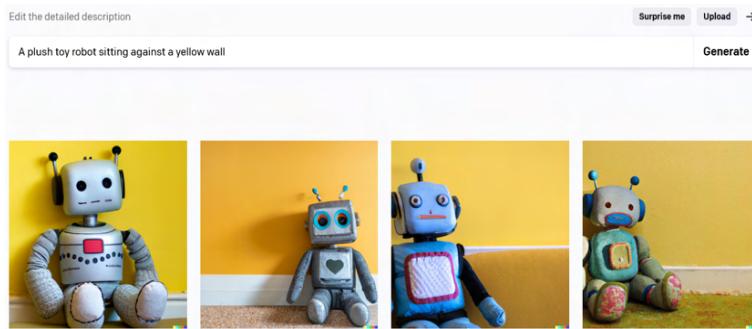
    # Append the list of references to the end of the file
    lines.append("\n\n" + "\n".join(references))

# Write the modified markdown file to disk
with open('your_modified_markdown_file.md', 'w') as f:
    f.writelines(lines)

```

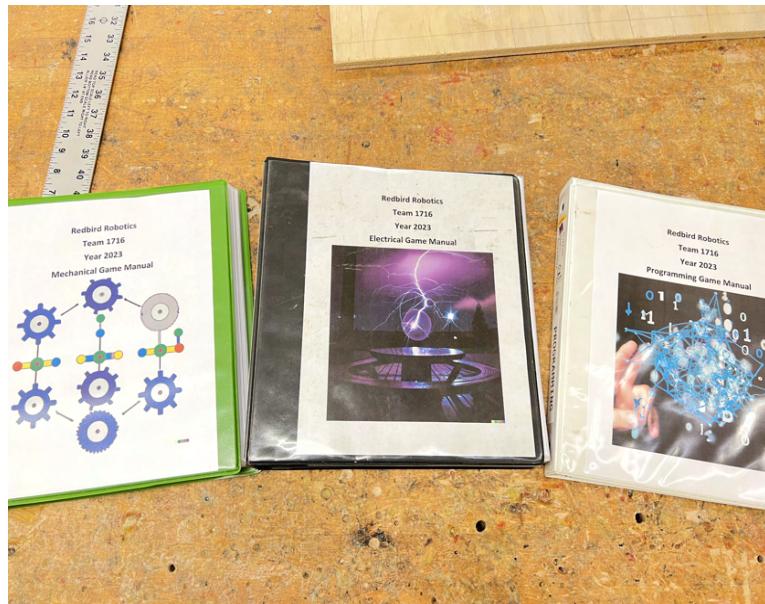
## Two AI's are Better than One

As a bonus section on ChatGPT, another famous AI called DALL-E 2 was released to the public mid-last year. DALL-E 2 is a text-to-image generator that can create any image given a text prompt. For example, giving it a description of "A plush toy robot sitting against a yellow wall" generates these images:



*DALL-E 2 in action*

Every image is a uniquely generated original. We wanted to have some original artwork on each of our sub-teams' game manuals, so we used ChatGPT to generate the text prompts that would create theme appropriate images:



*From left to right, Mechanical, Electrical, Programming*

The format for getting these kinds of results is pretty simple:

Human:

Create a list of prompts that I can feed into a text to image AI generator that create an abstract image of electricity or lightning

ChatGPT:

Here are five prompts for creating abstract images of electricity or lightning using a text to image AI generator:

1. "Generate an image of electricity as a swirling, vibrant energy field."
2. "Create an abstract representation of a lightning bolt, using vibrant colors and dramatic brushstrokes."
3. "Illustrate the concept of electrical current as a flowing stream of light or energy."
4. "Create a surreal landscape featuring a sky full of swirling, crackling electricity."
5. "Generate an image of a lightning storm, with the lightning depicted as bright, jagged lines of energy."

Human:

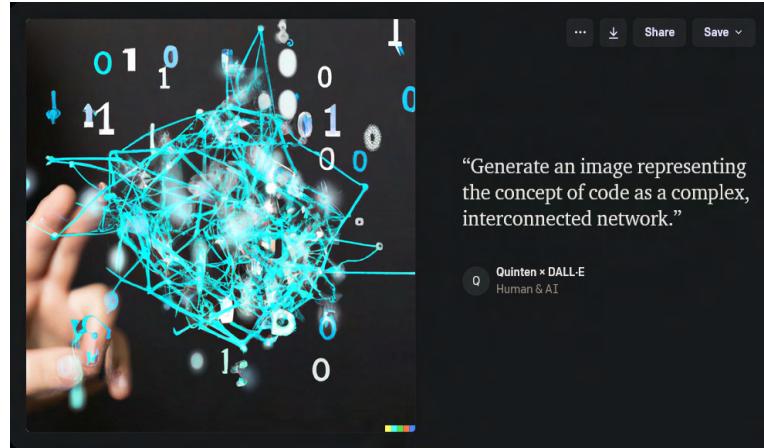
now do a list for programming

ChatGPT:

Here are five prompts for creating abstract images related to programming:

1. "Generate an image representing the concept of code as a complex, interconnected network."
2. "Create an abstract representation of the process of debugging, using vibrant colors and dynamic shapes."

3. "Illustrate the idea of data as a flowing stream of information, with different data types represented by different colors or patterns."
4. "Create a surreal landscape featuring floating islands of code or data."
5. "Generate an image of a computer's hardware and software as a cohesive, interconnected system."



*Example text prompt and the requisite image it generated*

# Why We Document

Our documentation was created because we are working with a more complex language this year, C++, and using a more complex system, command-based. We chose command-based because it's much more organized than timed. Rather than having a single loop where all code is run once per frame, which can be a restraint when building more complicated programs, each piece of functionality has its own spot in the file system. Command based programs are organized based on two main constructs:

- Subsystems, which provide access to individual components by abstracting away individual values for components
- Commands, which abstract subsystem methods away into specific actions for the robot to do, such as picking something up.

To write good documentation, you need to actually understand the system, how it works, and why it works in this way. One of the best ways to further understand and learn about a system is by writing down what you know so you can find where you have gaps in your understanding. We started by reading through WPILib's command-based docs, which are quite long and take a decent amount of time to read through. Then we wrote down what we knew in organized sections. Over the next few months, we rewrote the docs as we gained further understanding and experience with command-based, and we added sections to cover topics that we thought should be covered.

The docs that we have now are much more approachable for beginners and are generally faster to look to for reference than the WPILib docs (although they are riddled with relevant links to WPILib's docs for more depth and detail). They are now in a state where someone relatively new to programming could read through them and gain a reasonable understanding of the command-based system, as well as several other topics. Additionally, the docs now contain many code examples and simple explanations, so that current programmers can use them as a quick reference for a wide variety of topics, from how to post values to NetworkTables to how to add double solenoid functionality properly.

The docs also help experienced programmers who, instead of working on the code directly for the RoboRIO, write code for co-processors (such as raspberry pi's) that will go on the robot to aid in automation of control over the robot. Occasionally, one will work on some direct robot code or need to read and understand a section of it, and the docs are very useful for that.

Bad documentation can be a problem in situations like these. When someone doesn't put enough effort into writing the docs, they can actually be detrimental to the team, causing them to spend more time in the team docs than the official ones just trying to understand what they are saying. Additionally, team docs can often be inaccurate if the writer isn't careful with accuracy. This can, again, waste precious time for the programmer, as they check their broken code against the docs, which say their code should be working.

We acknowledge that using the full URL can be an inconvenience to type out, but we are much more concerned with the long term health of the information presented. If we were to use a URL shortener and the company that provides them goes out of business, suddenly all of our references are no longer accessible. For a more convenient experience, feel free to check out our documentation at the GitHub link on the cover of this document.

With that being said, here is our documentation so that others may learn from our experience.

## Command Documentation

### Table of Contents

1. [Command Concept Overviews](#)
2. [How to make a...](#)
  1. [Subsystem](#)
  2. [Command](#)
  3. [Button Binding](#)
3. [Decorators](#)
4. [Command Composition](#)
5. [Lambda Use](#)
6. [Default Commands](#)
7. [Autonomous](#)
8. [Generalized Solenoid Subsystem and Command](#)

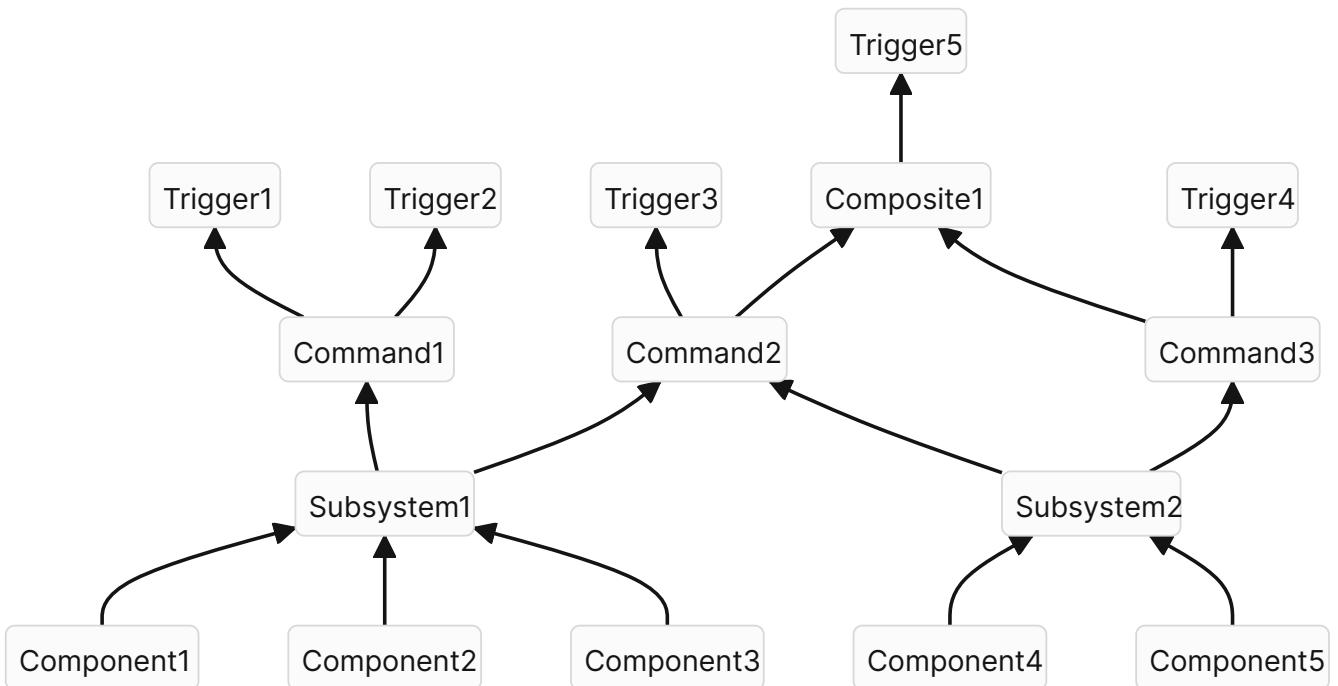
### Command Concept Overviews

- A **component** is a single device on the robot
  - Accessible via a class either built into wpilib or a 3rd party vendor library (not user-defined)
  - Lowest level concept we work with
  - Examples: `TalonFX`, solenoid, limit switch
- A **subsystem** is a group of components on the robot
  - The only way to access a component
    - Must contain all functionality needed to interface with the components contained in it
  - Each component should be contained in exactly **one** subsystem
    - Otherwise, the built-in protection against multiple commands being scheduled for one subsystem at once will not properly protect the component (this is handled by command requirements [1])
  - No components in a subsystem should need to do different tasks at the same time
    - A subsystem can't run multiple commands at once, so even if two tasks require different components and won't interfere, you will need to find a way to have the

- tasks in the same command or break your subsystem into multiple pieces
- If you have your drive train in the same subsystem as your shooter, you can't drive and shoot at once, at least not without an extremely confusing and unintuitive command; it's much simpler to have separate subsystems and separate commands for driving and shooting
- Read more on the subsystem docs [2]
- Examples: a drivetrain, a shooter, an intake
- A **command** does a task
  - The only way to access subsystems
    - User input (bindings) is abstracted into actions for the robot to do using its various subsystems
  - Handled by the Command Scheduler [3], which determines what commands are running when through a system of scheduling a command when a binding is pressed in most cases and descheduling it when it completes or is interrupted
  - Each subsystem can be used in multiple commands
    - WPIlib handles conflicts, ensuring that a subsystem doesn't try to do two things at once using command requirements [4]
    - Each command has a list of requirements, the subsystems that it uses
    - You can make a command non-interruptible, meaning that if another command with a shared requirement is scheduled (or the command is interrupted in some other way), the current command will not be cancelled (interrupted)
      - You can detect if your command was ended due to an interruption using the boolean `interrupted` parameter the command's `End()` method is called with
  - Commands can be formed out of other commands in command grouping, these commands are called compositions
    - There are several different types of command group to control the order that the composed commands are run in, read more here [5]
    - As composites are commands, they can be used in the formation of more composites
  - Read more on the command docs [6]
  - Examples: drive, shoot out a ball, take in a ball
- A **trigger** is an input that interacts with the command scheduler to control when commands are scheduled and descheduled
  - The most common type of trigger is a button input
    - These can be formed by declaring a `frc2::CommandXboxController` and calling a button method (`A()`, `B()`, `Y()`), then a binding on that (`OnTrue()`, `WhileTrue()`)
      - Ex: `commandXboxControllerObject.A().OnTrue([CommandPtrToBind])`
  - Commands are generally bound to triggers in the `ConfigureButtonBindings` class of `RobotContainer`
  - Alternatively, you can make custom triggers [7], which, rather than being a button input, take a boolean value that you provide

- Example: 

```
frc2::Trigger([&objectYourBoolIsIn] { return objectYourBoolIsIn.GetBoolValue(); }).Binding(Command(&subsystem).ToPtr());
```
- There are different bindings [8] available, which allow for different logical connections between triggers and commands
  - First, you make a trigger (such as a button), then call a binding on it, then place your command in the form of a `CommandPtr` as the argument for the binding
- Triggers can be composed [9] so that `&&`, `||`, or `!` can be applied to them
- Bindings return the original trigger, so you can chain together bindings [10]
- Read more on the docs [11]



## How to make a...

The following section covers how to write a new subsystem and command, and how to bind a command to a trigger. All file paths are in the `src > main` directory.

### Subsystem

1. Create your files: you'll need a `include/subsystems/SubsystemName.h` file and a `cpp/subsystems/SubsystemName.cpp` file. (from now on, `SubsystemName` represents the name of your subsystem)
2. In `SubsystemName.h` declare methods and components:

```
#pragma once
```

```

#include <frc2/command/SubsystemBase.h>
#include <ctre/Phoenix.h>

class SubsystemName : public frc2::SubsystemBase {
public:
    SubsystemName(); //Constructor

    //Method declarations go here:
    void ExampleMethod1(); //Declare your own methods like so
    void ExampleMethod2(double exampleParameter);

private:
    //subsystem components such as motors:
    WPI_TalonFX exampleTalonFX;
    WPI_TalonSRX exampleTalonSRX;
};

```

3. In `SubsystemName.cpp` define methods and components:

```

#include "subsystems/SubsystemName.h"

SubsystemName::SubsystemName() : exampleTalonFX{0}, exampleTalonSRX{1} {} //constructor,
make sure to include all components with their constructor arguments

void SubsystemName::ExampleMethod1() {
    exampleTalonFX.Set(0.5);
}

void SubsystemName::ExampleMethod2(double exampleParameter) {
    exampleTalonSRX.Set(exampleParameter);
}

```

4. In `include/RobotContainer.h`, declare your subsystem:

```

#pragma once

#include <frc2/command/Command.h>
#include <frc2/command/button/CommandXboxController.h>

#include "subsystems/SubsystemName.h" //add this line

class RobotContainer {
public:
    RobotContainer();

private:
    frc2::CommandXboxController controller{0}; //controller used for bindings
}

```

```

    SubsystemName subsystemName; //use pascalCase for the instance

    void ConfigureButtonBindings();
};


```

## Command

1. Create your files: you'll need a `include/commands/CommandName.h` file and a `cpp/commands/CommandName.cpp` file. (from now on, `CommandName` represents the name of your command)
2. In `CommandName.h`, declare constructor and methods you plan to use:

```

#pragma once

#include <frc2/command/CommandBase.h>
#include <frc2/command/CommandHelper.h>

#include "subsystems/SubsystemName.h" //Subsystem requirement

class CommandName : public frc2::CommandHelper<frc2::CommandBase, CommandName> {
public:
    explicit CommandName(SubsystemName* subsystem); //constructor, it must take all
required subsystems as parameters

    void Initialize() override; //Called once at the beginning, defaults to doing nothing

    //void Execute() override; //Called every frame that the command is scheduled,
defaults to doing nothing

    void End(bool interrupted) override; //Called once at the end, defaults to doing
nothing, `interrupted` parameter tells whether the command was ended because it was
interrupted

    //bool IsFinished() override; //you return whether the command has completed, run
once per frame that the command is scheduled, defaults to false (no end until
interrupted)

private:
    SubsystemName* subsystemName; //each subsystem used needs its own pointer
};


```

3. In `CommandName.cpp`, define methods and constructor:

```

#include "commands/CommandName.h"

CommandName::CommandName(SubsystemName* subsystem) : subsystemName(subsystem) {
//constructor, takes all required subsystems and adds them to command requirements and
instance variables
}


```

```

    AddRequirements(subsystem); //subsystems must be added to the command's requirements
}

void CommandName::Initialize() {
    subsystemName->ExampleMethod1();
    subsystemName->ExampleMethod2(0.5);
}

/*void CommandName::Execute() {
    subsystemName->ExampleMethod1();
}*/
```

```

void CommandName::End(bool interrupted) {
    subsystemName->ExampleMethod2(0);
}
```

```

/*bool SpinTfx::IsFinished() { //return whether the command should finish this frame
}*/
```

## Button Binding

1. Ensure that you have `frc2::CommandXboxController controller{0};` and `void ConfigureButtonBindings();` under `private` in `RobotContainer.h` (you'll need to include `<frc2/command/button/CommandXboxController.h>`). This instantiates a controller to use for button triggers and declares the method used for bindings.
2. In `RobotContainer.cpp`, declare your button bindings (read the docs [12] to find the correct binding for the functionality you want to achieve):

```

#include "RobotContainer.h"

#include <frc2/command/button/JoystickButton.h>

using namespace frc2;

RobotContainer::RobotContainer() {
    ConfigureButtonBindings();
}

void RobotContainer::ConfigureButtonBindings() {
    // bind your button triggers to commands here
    controller.A().WhileTrue(CommandName(&subsystemName).ToPtr()); //declare a gamepad
button trigger by calling the method named for the button you want on the controller,
then call a binding on that, such as `WhileTrue()`, which schedules the command when you
hit the button and deschedules it when you release it. The binding takes a `CommandPtr`,
which you can make by declaring a command (which takes a subsystem instance with an "&" as
the parameter), then calling `ToPtr()` or a decorator on it.
```

```

controller.B()
    .OnTrue(CommandName(&subsystemName).ToPtr())
    .WhileTrue(Command2Name(&subsystemName).ToPtr()); //bindings return the original
trigger, so bindings can be chained together

(controller.X() || controller.Y()).WhileFalse(CommandName(&subsystemName).ToPtr());
//trigger composition, if neither are pressed, the command `CommandName` will be
scheduled
}

```

## Decorators

Decorators are a feature of WPILib that allow you to add additional functionality to commands during binding. They can be used by calling the decorator method on the command of choice inside of the binding method. So, for example:

```
Jcontroller.Y().WhileTrue(ToggleSolenoid(&solenoidSubsystem).WithTimeout(3_s));
```

Read more about types of decorators on the docs [13].

## Command Composition

Commands can be made by combining other commands using command composition. There are two main types of command groups: the *Sequence*, which runs a list of commands in sequence, and the *Parallel*, which runs a list of commands at the same time, provided they don't share any requirements (there are also two sub-types of parallel command group, listed here [14]). If you need a more complex command structure, it can be built by composing command compositions, since command groups are themselves commands, they can be composed into more command groups. Commands can be composed together by including `<frc2/command/Commands.h>` and then calling the factory for the group you want from `frc2::cmd` in `RobotContainer.cpp`:

```

controller.A().WhileTrue(frc2::cmd::Parallel(std::move(command1), std::move(command2),
std::move(command3)));
controller.B().OnTrue(frc2::cmd::Sequence(std::move(command1), std::move(command2)));

```

Each command included in the composition needs to have `std::move` called on it from the `<utility>` library.

## Lambda Use

A useful technique in command based is using lambda functions to easily and quickly get live input from the controller or some other variable. They are capable of getting the current value of a variable that is accessible from the place they are defined from. This is commonly used in the parameters of commands, such as when driving or shooting when you need the joystick or trigger values. Lambdas can be used in this way like so:

1. Put a parameter of type `std::function<RETURNTYPE()>` in your command's header file, where `RETURNTYPE` is the type of variable you would like to get from your input
  - For example, a joystick axis would have the type `double`, and the lambda would be declared as a `std::function<double()>`
2. Make a private variable of the same type as your lambda so that it is accessible outside of the constructor
3. In your command's source file, make the constructor match the header and place `memberVar(std::move(parameterVar))` after your subsystem's member definition, separated by commas
  - `memberVar` is the name of your member variable, `parameterVar` is the name of your parameter, and `std::move` is accessible in `<utility>`
  - For example:

```
ShootCommand::ShootCommand(Shooter* subsystem, std::function<double()> speed)
    : drivetrain(subsystem), triggerValue(std::move(speed))
{
    AddRequirements(subsystem);
}
```

4. When you bind your command, you can use the following format to define your lambda in the constructor:

```
ShootCommand::ShootCommand(Shooter* subsystem,
    [this] { return controller.GetRightTriggerAxis(); });
```

- The square brackets hold a list of class that will be accessible within the lambda, `this` refers to the current class, `RobotContainer`, where the controller is contained
- Anythin within the curly braces will be run, return the value that you need in as few lines as possible

## Default Commands

Each subsystem can have a default command that is run if no other command is currently using it. This feature is often useful in tandem with lambda functions because they allow functionality to be changed while running. Driving is almost always a default command, and shooting may also be; both use lambdas. You can set the default command of a subsystem in the `RobotContainer.cpp` constructor like so:

```
shooter.SetDefaultCommand(ShootCommand(&shooter));
```

## Autonomous

This guide is based off of this one [15] from the smartdashboard docs.

**Note:** Decorators and inline commands are not supported for autonomous because the sendable chooser does not take CommandPtrs, define commands you need for autonomous in their own files (including compositions)

1. In `RobotContainer.h`, include `<frc/smartdashboard/SendableChooser.h>` and declare a private `SendableChooser` object, which is able to be sent to the smartdashboard as a list of options:

```
frc::SendableChooser<frc2::Command*> autonomousChooser;
```

2. Declare a private object for each command you would like to be available as an autonomous routine (subsystem object must already be declared):

```
Command1Name command1Name{&subsystemName};
```

3. Declare a public method to be accessed by `robot.cpp` that returns the currently selected autonomous command:

```
frc2::Command* GetAutonomousCommand();
```

4. In `RobotContainer.cpp`, in the constructor, set your default command and add the others:

- For the first command, call `SetDefaultOption()`:

```
autonomousChooser.SetDefaultOption("Display Name", &command1Name);
```

- For all other commands, call `AddOption()`:

```
autonomousChooser.AddOption("Display Name", &commandName2);
```

5. Define `GetAutonomousCommand()` to return the currently selected autonomous command at the end of `RobotContainer.cpp`:

```
frc2::Command* RobotContainer::GetAutonomousCommand() {
    return autonomousChooser.GetSelected();
}
```

6. In `Robot.h`, publicly override `AutonomousInit()` and `TeleopInit()`:

```
void AutonomousInit() override;
void TeleopInit() override;
```

7. In `Robot.cpp`, schedule the command if there is one selected in `AutonomousInit()`:

```
void Robot::AutonomousInit() {
    autonomousCommand = robotContainer.GetAutonomousCommand();

    if (autonomousCommand != nullptr) {
        autonomousCommand->Schedule();
    }
}
```

8. Cancel the autonomous command when teleoperated mode begins in `TeleopInit()`:

```
void Robot::TeleopInit() {
    if (autonomousCommand != nullptr) {
        autonomousCommand->Cancel();
        autonomousCommand = nullptr;
    }
}
```

## Generalized Solenoid Subsystem and Command

Most double solenoids should be able to function using basically the same subsystem and command, though you may need to add the solenoid to another subsystem in some cases. Below is a general use command and subsystem for toggling a solenoid.

**IMPORTANT: The below command must be bound with the `WithTimeout(3_ms)` decorator. Failure to do so may result in damage to the solenoid.**

`SolenoidSubsystem.h`:

```
#pragma once

#include <frc2/command/SubsystemBase.h>
#include <frc/DoubleSolenoid.h>
#include <frc/PneumaticHub.h>

class SolenoidSubsystem : public frc2::SubsystemBase {
public:
    SolenoidSubsystem();

    void SetPosition(frc::DoubleSolenoid::Value position);
    frc::DoubleSolenoid::Value GetPosition();
}
```

```

private:
    frc::PneumaticHub hub{0};
    frc::DoubleSolenoid solenoid = hub.MakeDoubleSolenoid(0, 1);
};


```

SolenoidSubsystem.cpp:

```

#include "subsystems/SolenoidSubsystem.h"

SolenoidSubsystem::SolenoidSubsystem() {
    hub.EnableCompressorDigital();
}

void SolenoidSubsystem::SetPosition(frc::DoubleSolenoid::Value position) {
    solenoid.Set(position);
}

frc::DoubleSolenoid::Value SolenoidSubsystem::GetPosition() {
    return solenoid.Get();
}

```

ToggleSolenoid.h:

```

#pragma once

#include <frc2/command/CommandBase.h>
#include <frc2/command/CommandHelper.h>

#include "subsystems/SolenoidSubsystem.h"

class ToggleSolenoid : public frc2::CommandHelper<frc2::CommandBase, ToggleSolenoid> {
public:
    explicit ToggleSolenoid(SolenoidSubsystem* subsystem);

    void Initialize() override;
    void End(bool interrupted) override;

private:
    SolenoidSubsystem* solenoidSubsystem;
    frc::DoubleSolenoid::Value currentPosition;
};


```

ToggleSolenoid.cpp:

```

#include "commands/ToggleSolenoid.h"

ToggleSolenoid::ToggleSolenoid(SolenoidSubsystem* subsystem) :

```

```

solenoidSubsystem(subsystem) {
    AddRequirements(subsystem);
    currentPosition = solenoidSubsystem->GetPosition(); //set the initial position to
whatever the solenoid is currently at
}

void ToggleSolenoid::Initialize() {
    if (currentPosition == frc::DoubleSolenoid::Value::kReverse) { //if reverse, set to
forward
        SolenoidSubsystem->SetPosition(frc::DoubleSolenoid::Value::kForward);
        currentPosition = frc::DoubleSolenoid::Value::kForward;
    }
    else { //if not reverse, set to reverse
        SolenoidSubsystem->SetPosition(frc::DoubleSolenoid::Value::kReverse);
        currentPosition = frc::DoubleSolenoid::Value::kReverse;
    }
}

void ToggleSolenoid::End(bool interrupted) { //should be called 3 ms after `Initialize()`
using `WithTimeout()`` decorator
    SolenoidSubsystem->SetPosition(frc::DoubleSolenoid::Value::kOff); //set the solenoid
to off, the piston will remain where it was last set to
}

```

[1]

<https://docs.wpilib.org/en/stable/docs/software/commandbased/commands.html#getrequirements>

[2] <https://docs.wpilib.org/en/stable/docs/software/commandbased/subsystems.html>

[3] <https://docs.wpilib.org/en/stable/docs/software/commandbased/command-scheduler.html>

[4]

<https://docs.wpilib.org/en/stable/docs/software/commandbased/commands.html#getrequirements>

[5] <https://docs.wpilib.org/en/stable/docs/software/commandbased/command-groups.html>

[6] <https://docs.wpilib.org/en/stable/docs/software/commandbased/commands.html>

[7] <https://docs.wpilib.org/en/stable/docs/software/commandbased/binding-commands-to-triggers.html#arbitrary-triggers>

[8] <https://docs.wpilib.org/en/stable/docs/software/commandbased/binding-commands-to-triggers.html#trigger-bindings>

[9] <https://docs.wpilib.org/en/stable/docs/software/commandbased/binding-commands-to-triggers.html#composing-triggers>

- [10] <https://docs.wpilib.org/en/stable/docs/software/commandbased/binding-commands-to-triggers.html#chaining-calls>
- [11] <https://docs.wpilib.org/en/stable/docs/software/commandbased/binding-commands-to-triggers.html>
- [12] <https://docs.wpilib.org/en/stable/docs/software/commandbased/binding-commands-to-triggers.html#trigger-bindings>
- [13] <https://docs.wpilib.org/en/stable/docs/software/commandbased/command-compositions.html#composition-types>
- [14] <https://docs.wpilib.org/en/stable/docs/software/commandbased/command-compositions.html#parallel>
- [15] <https://docs.wpilib.org/en/stable/docs/software/dashboards/smartdashboard/choosing-an-autonomous-program-from-smartdashboard.html#command-based>

## WPILib Reference Document

### Table of Contents

1. [Component Reference](#)
2. [Naming Conventions](#)
3. [3rd Party Vendor Libraries](#)
4. [Dashboard Values](#)
  1. [SmartDashboard](#)
  2. [Shuffleboard](#)
  3. [Reading Values](#)
5. [NetworkTables](#)
6. [Solenoids](#)

### Component Reference

| Class Name          | Include Path           | Notes   |
|---------------------|------------------------|---|
| WPI_TalonFX         | <ctre/Phoenix.h>       | -   |
| WPI_TalonSRX        | <ctre/Phoenix.h>       | -   |
| frc::DoubleSolenoid | <frc/DoubleSolenoid.h> | Positions are accessible at<br>frc::DoubleSolenoid::Value ; more info available in the <a href="#">#solenoids</a> section |
| rev::CANSparkMax    | <rev/CANSparkMax.h>    | -   |
| frc::PneumaticHub   | <frc/PneumaticHub.h>   | -   |

## Naming Conventions

CC is camelCase, PC is PaskalCase

**Note:** If a single instance of a class is being used, it should be named the same as its class, but in camelCase, assuming this causes no conflicts.

| Structure       | Convention |
|-----------------|------------|
| Class           | PC         |
| Method/Function | PC         |
| Variable        | CC         |
| Parameter       | CC         |

## 3rd Party Vendor Libraries

*3rd Party Vendor Libraries* are libraries written and distributed by component manufacturers for use in interfacing with their components. To install REVLib or the CTRE Phoenix library:

1. Install their respective software.

- ForREVLib, download the Java/C++ API [1] (big orange button with "Download latest Java API" on it) and unzip into the C:\Users\Public\wpilib\2023 directory on Windows or ~/wpilib/2023 directory on Linux.
- For the CTRE Phoenix library, download and install the Phoenix Framework [2].

2. Next, in VSCode, press the wpilib icon in the top right in your project and go to WPIlib:

```
Manage Vendor Libraries then Install new libraries (offline) and select REVLib, Phoenix (v5), or both.
```

**Do not** install the Phoenix (Pro) library, as it is locked behind a paywall and will prevent your code from running. To uninstall a library, go to WPIlib: Manage Vendor Libraries again and select Manage current libraries, then select any libraries you wish to remove and press enter.

## Dashboard Values

One very useful feature of WPILib is the ability to post values to a dashboard. This can be used to show values for debugging, display useful information while driving, etc. Of the several options for dashboards included with WPILib (including making your own), we use Shuffleboard because it looks nice, is simple to use, and is feature rich. There are two methods for putting values on Shuffleboard:

- Using the API for SmartDashboard, an older dashboard with less features
  - Shuffleboard can read values from SmartDashboard's API and will automatically display them on its SmartDashboard tab

- Using Shuffleboard's own API, which supports specifying which tab to place the value on, along with other features

The SmartDashboard API is the simpler option, but does not support tab placement, widget type selection, positioning, etc. As such, the SmartDashboard API should be used for quick testing, but the ShuffleBoard API should be used for production.

## SmartDashboard

You can put Boolean, Numeric, or String values on the dashboard very simply by including `<frc/smartdashboard/SmartDashboard.h>` and calling `frc::SmartDashboard::PutBoolean / PutNumber / PutString` with the parameters (`"Displayed Name"`, `value`). The value should appear on Shuffleboard on the `SmartDashboard` tab. [3]

## Shuffleboard

To put values on Shuffleboard, include `<frc/shuffleboard/Shuffleboard.h>` and declare a `nt::GenericEntry*` for each value that you will need to update. Then, in your constructor, set your variables to the return of this:

```
frc::Shuffleboard::GetTab("Your Tab")
    .Add("Your Value Name", 3.14)
    .GetEntry();
```

This will function creates a tab if it does not exist, then creates a value on that tab if it does not exist, it then returns the NetworkTable entry for setting, note that this will not change already existing values. If you need to specify a widget to use, call `WithWidget()` on `.Add()` with a `frc::BuiltInWidgets` as the parameter for the type of widget, and to modify widget properties, call `WithProperties()` with the properties like so:

```
frc::Shuffleboard::GetTab("Your Tab")
    .Add("Your Value Name", 3.14)
    .WithWidget(frc::BuiltInWidgets::kNumberSlider)
    .WithProperties({ // specify widget properties here
        {"min", nt::Value::MakeDouble(0)},
        {"max", nt::Value::MakeDouble(10)}
    })
    .GetEntry();
```

To specify the size or position, call `WithSize()` or `WithPosition()`, where `WithSize()` takes the height then width, and `WithPosition()` takes the row and column, starting at (0, 0) in the top left corner. You can also specify a layout for the widget by calling `GetLayout("Layout Name", frc::BuiltInLayouts::WantedLayout)` after `GetTab()`. Finally, you can open a tab from code by calling `frc::Shuffleboard::SelectTab("Tab Name")`. Read in more detail on the docs [4]

## Reading Values

On Shuffleboard, you can make a value selectable by calling the type-specific get method on a `nt::GenericEntry*` created in the [Shuffleboard](#) section (make sure to make the widget something modifyable). For example:

```
doubleEntry = frc::Shuffleboard::GetTab("Your Tab")
    .Add("Number", 0)
    .WithWidget(frc::BuiltInWidgets::kNumberSlider)
    .GetEntry();

double number = doubleEntry->GetDouble(0.0); //parameter is the default value

boolEntry = frc::Shuffleboard::GetTab("Your Tab")
    .Add("Boolean", false)
    .WithWidget(frc::BuiltInWidgets::kToggleSwitch)
    .GetEntry();

bool doTheThing = boolEntry->GetBoolean(false);
```

Make sure that you do not set the value too often, as it will make it difficult for you to get input reliably.

## NetworkTables

Another feature of WPILib is NetworkTables, which can be used to communicate values between the driverstation computer, the RoboRIO, and any coprocessors such as Raspberry Pi's the robot may have on it. All NetworkTable values are copied to all devices connected to networktables, and values are organized similarly to a filesystem, where the folders are "subtables" and the files are "topics", which is the NetworkTables term for values. You can reference a value by accessing its table similarly to opening a folder, then declaring a `subscriber` to read values or a `publisher` to write values.

1. In your header, include `<networktables/NetworkTable.h>`, `<networktables/NetworkTableInstance.h>`, and the topic for the data type you are working with, for example a float: `<networktables/FloatingPointTopic.h>`
2. Declare a private `NetworkTableInstance`, a `shared_ptr<nt::NetworkTable>`, and a subscriber or publisher for the data type:

```
nt::NetworkTableInstance networkTableInst; //the default networktables network
std::shared_ptr<nt::NetworkTable> table; //the table to be used

nt::FloatSubscriber subscriber; //able to read values

nt::FloatPublisher publisher; //able to write values
```

3. In the source file constructor, set the values to read from the topic you want:

```
networkTableInst = nt::NetworkTableInstance::GetDefault(); //the network the RoboRIO  
and DriverStation are connected to  
table = networkTableInst.GetTable("YourTable"); //if the table does not exist, it  
will be created  
  
subscriber = table->GetFloatTopic("YourTopic").Subscribe(0.0); //will create the  
topic if it doesn't exist,  
//Subscribe() parameter is the default value  
  
publisher = table->GetFloatTopic("YourTopic").Publish(); //will create the topic if  
it doesn't exist,
```

4. Read or write values:

```
subscriber.Get(); //returns the float value topic is currently set to  
publisher.Set(124.5); //takes a float to set the topic value to
```

## Solenoids

Solenoids are the controllers and valves for air-powered devices, generally pistons. We use double solenoids, which have three settings: Forward, Reverse, and Off. The solenoid should never be in Forward or Reverse for longer than a fraction of a second (about 3 ms), just long enough to move the piston. It must then be set to Off again in order to avoid damaging the piston. There is a sample subsystem and command available for them in the generalized solenoid subsystem and command section in the Command Concept Overviews section above. (docs [5])

- [1] <https://docs.revrobotics.com/sparkmax/software-resources/spark-max-api-information#c++-and-java>
- [2] <https://store.ctr-electronics.com/software/>
- [3] <https://docs.wpilib.org/en/stable/docs/software/dashboards/smartdashboard/displaying-expressions.html>
- [4] <https://docs.wpilib.org/en/stable/docs/software/dashboards/shuffleboard/layouts-with-code/index.html>
- [5] <https://docs.wpilib.org/en/stable/docs/software/hardware-apis/pneumatics/pneumatics.html>

## What's Next?

We should see most of the programming team back next year. Armed with the current understanding and experience we expect to build on the software of this year to solve even harder problems to meet whatever the challenges may be.

We expect April Tags to continue to be used in competition. While our current AprilTag detection is good, without question it will be great with just a little more work in the future. This also applies to our object detection today. It is hard to detect colors and shapes on the field, but being able to build on the solutions of this year will be welcome. As the code is covered by a permissive open source license, others can also learn from, and benefit from our software.

Lastly, there is always talk of what comes next with the team. There is currently an interest in the Rust programming language. There will be opportunities to detect more and different things on the field. More automation will allow for more consistent and reliable robot operations. With expectations of the programming team growing, we will see opportunities for the current members to focus on skills such as software architecture, skilling up other team members, and team leadership.

We offer you our most sincere "Thank You" for reading about everything we've done. Our team has put so much work into this season laying a foundation that will last for years to come. We hope some of the solutions we've found will be useful to other teams, and we hope that the ideas we've presented here will spark even more creative solutions from the FRC community. Until next year, Team 1716, Redbird Robotics.

Hundreds of Alumni

121 Years of Collective Industry Experience

17 Years of Robots

9 Team members

7 Mentors

3 Name Changes

1 Team