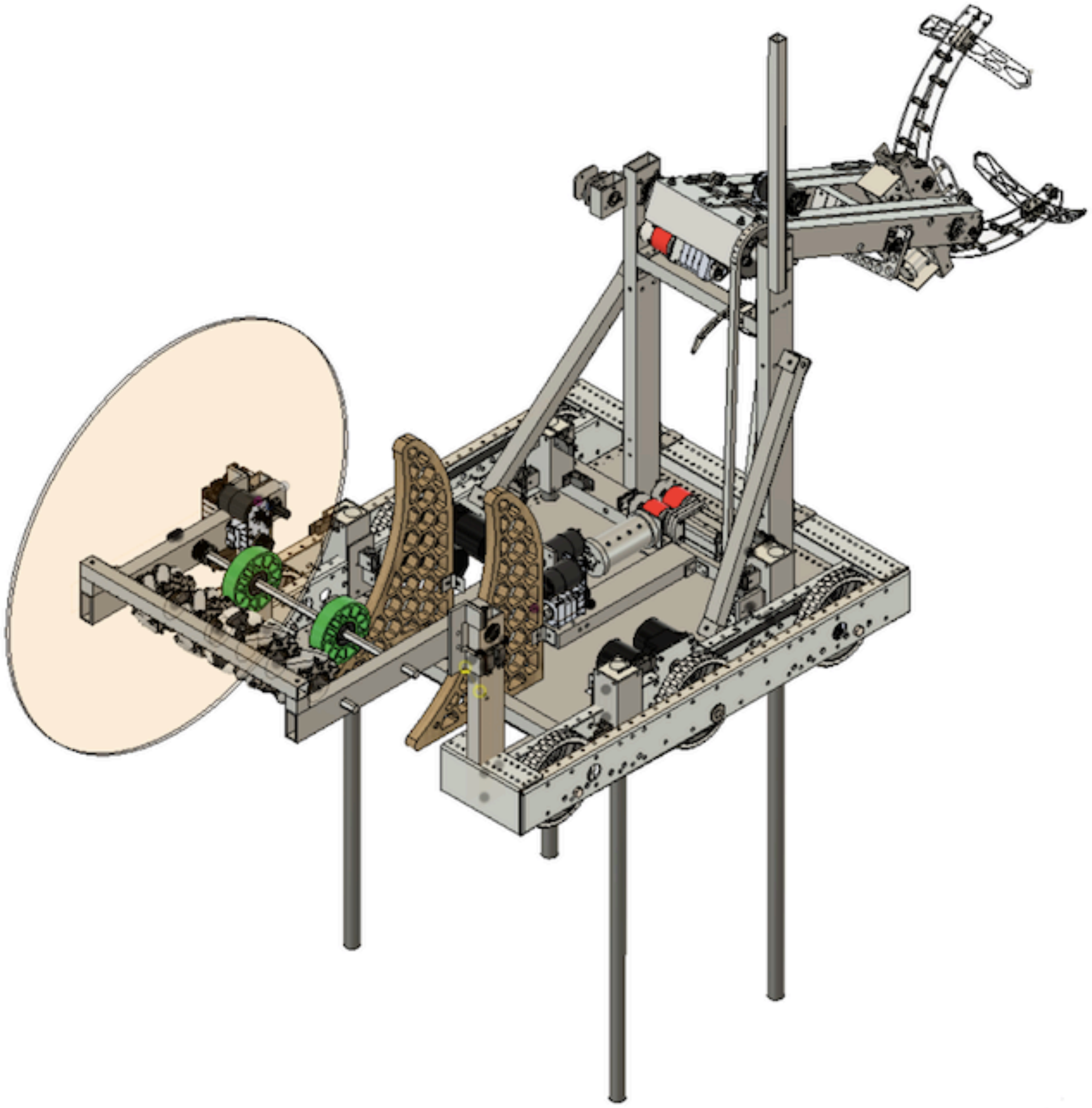


Homestead Robotics (670)

Software Binder 2019

Table of Contents

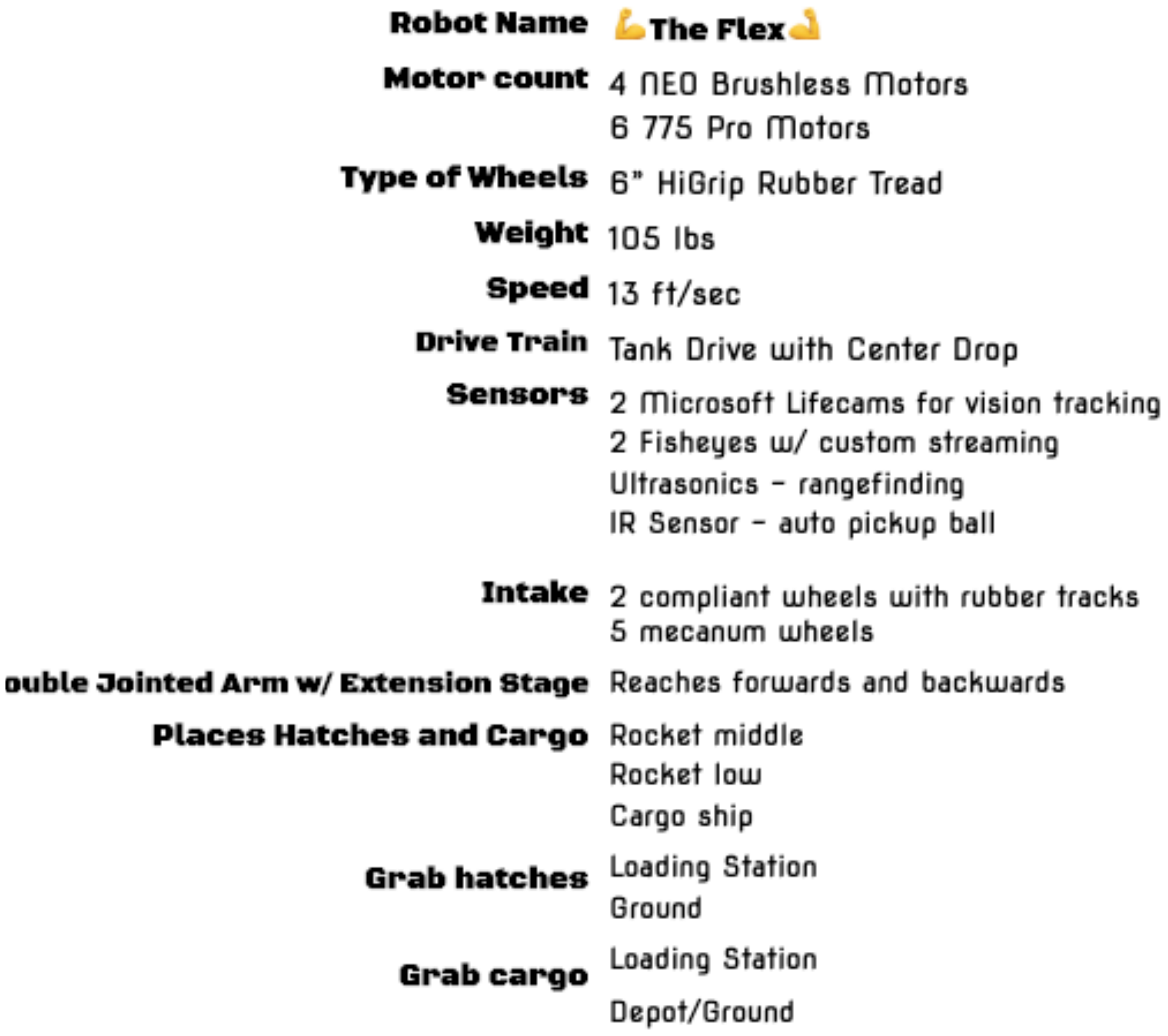
Introduction

Introduction

Welcome to the Homestead Robotics Software Binder for the 2019 FRC Season – Destination: Deep Space.

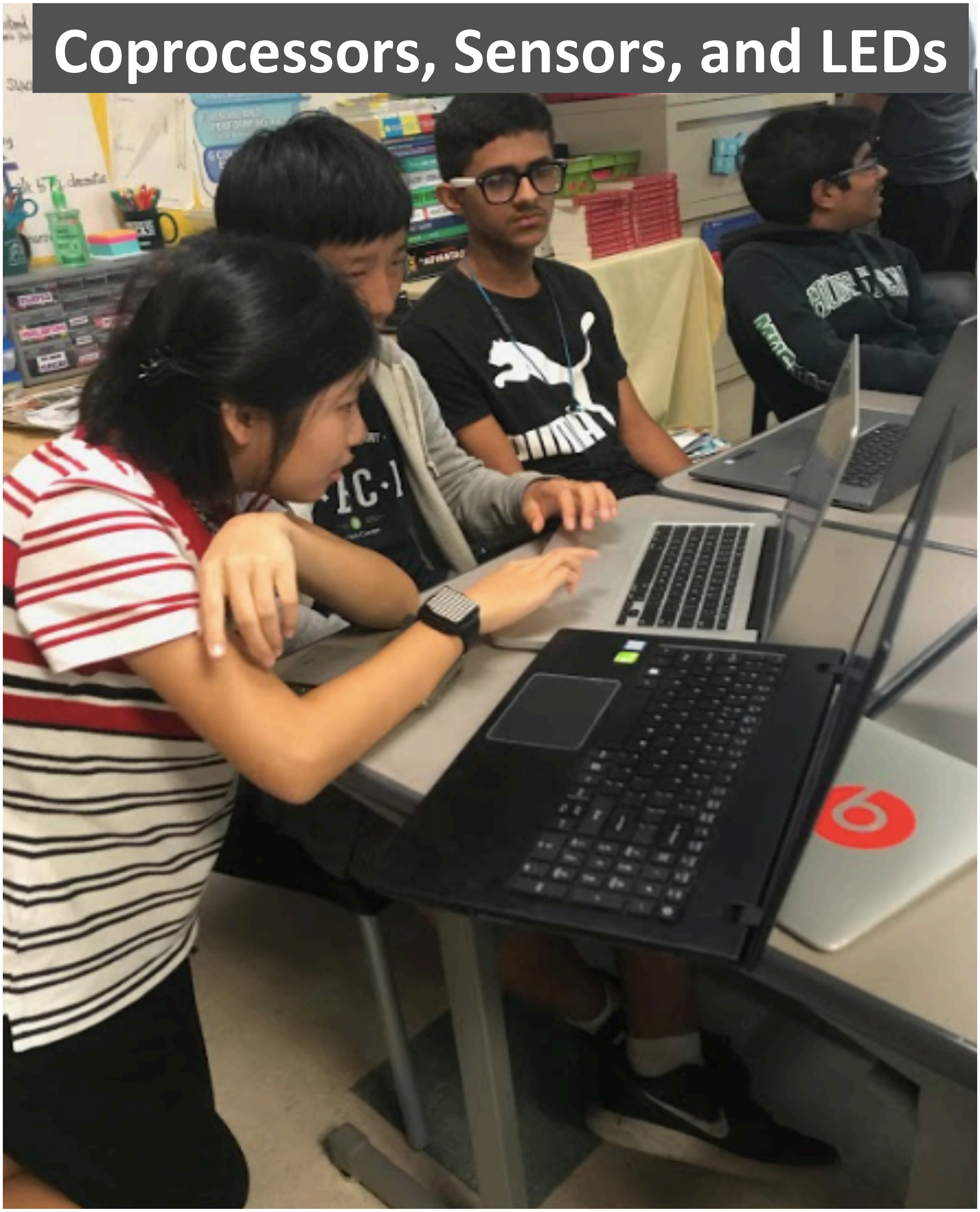
- Homestead Robotics (Team 670) is based out of Homestead High School in Cupertino, California.
- Our team programs in Java, Python, and Arduino C++.
- Our website is homesteadrobotics.com and any questions can be sent to team670@homesteadrobotics.com.

Quick Robot Specs



Robot Name	💪**The Flex**💪
Motor count	4 NEO Brushless Motors 6 775 Pro Motors
Type of Wheels	6" HiGrip Rubber Tread
Weight	105 lbs
Speed	13 ft/sec
Drive Train	Tank Drive with Center Drop
Sensors	2 Microsoft Lifecams for vision tracking 2 Fisheyes w/ custom streaming Ultrasonics - rangefinding IR Sensor - auto pickup ball
Intake	2 compliant wheels with rubber tracks 5 mecanum wheels
ouble Jointed Arm w/ Extension Stage	Reaches forwards and backwards
Places Hatches and Cargo	Rocket middle Rocket low Cargo ship
Grab hatches	Loading Station Ground
Grab cargo	Loading Station Depot/Ground

Coprocessors, Sensors, and LEDs



Coprocessors, Sensors, and LEDs

Our 2019 Robot features the following elements:

Coprocessors

- 1 Raspberry Pi
- 1 Odroid XU4 – Later Removed

Sensors:

- 2 Microsoft LifeCams
- 2 Fisheye Cameras
- 3 Ultrasonic Sensors
- 1 IR Sensor
- 1 NavX Gyroscope

LEDs

- 2 LED Rings
- Decorative LED Strips

Coprocessors

Raspberry Pi

The Raspberry Pi is being used to run driver camera streams as well as a python script that puts the vision error code to network tables if appropriate cameras are not found.

Start scripts on boot

The streaming script (written in bash) and the Python script are set to run when the Pi turns on. The script called startup.sh calls both scripts and is set to run on bootup with the crontab command. To set up a script to run on boot:

1. Open up a terminal window and type crontab -e
2. Select your editor of choice
3. Add the following line to the bottom of the file: @reboot followed by the name of your script (for us this line looks like @reboot /home/pi/startup.sh)

Static IP Address

A static IP address was given to the Pi so that its IP address is the same regardless of which router it is connected to. This is useful especially for testing purposes since you may not always connect the Pi to the same router, in which case you would have to scan for its IP address each time. Also, since we are using mjpg-streamer for driver camera streams, which requires us to know the IP address of the device the cameras are plugged into, the static IP was necessary for the streams to function seamlessly. Use the instructions at https://www.modmypi.com/blog/how-to-give-your-raspberry-pi-a-static-ip-address-update to set a static IP address for your Pi.

Refer to the page on Driver Cameras for details on camera streaming

Disable WiFi

1. Open a terminal window and run systemctl disable wpa_supplicant
2. Open /boot/config.txt in an editor of your choice
3. Add dtoverlay=pi3-disable-wifi to the file

Credit to https://irulan.net/disable-wifi-and-bluetooth-on-raspberry-pi-3/

Use this process to clone the contents of one SD card onto another

This can be used to create two copies of an SD card for use on two Pi's at once

NOTE: this process requires Win32 Disk Imager to be installed

SD1 = the SD card being copied from (the original)
SD2 = the SD card being copied to (the new one)

Do this with SD1:
Plug into laptop using an adapter if necessary
Read the contents of the [D:] drive of the card to a convenient local destination using Win32 Disk Imager

Unplug SD card from laptop

Then do this with SD2:
Plug into laptop using an adapter if necessary
Windows should suggest formatting the disk, go ahead and do that
Follow the instructions on this page to remove all partitions on the SD card: http://oddsnsods.net/blog/?p=100
Use Win32 Disk Imager to write the file read from SD1 in the previous part to the [E:] drive of the SD card

Insert SD2 into the Pi, turn it on, and run 'ls' in the root directory to ensure all contents were copied over

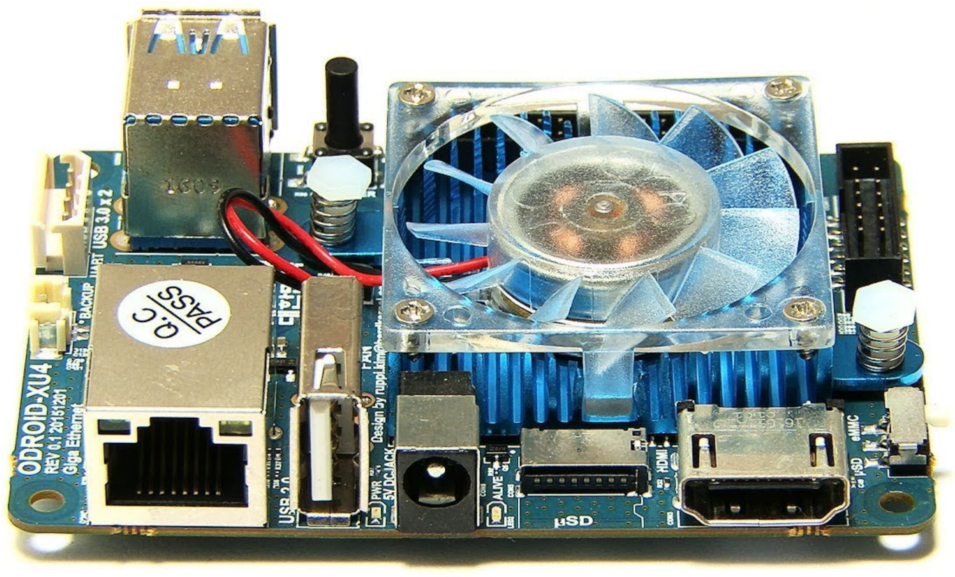
ODroid XU4 Coprocessor

The ODroid XU4 coprocessor is a powerful coprocessor for vision when you want a continuous feed for vision, such as if you were feeding its data directly into a PID loop. However, it is a large current draw, and not necessary if using the option we went for with vision this year: taking a single picture for targeting using a camera connected to a Raspberry Pi, then using a Pure Pursuit algorithm to drive to the target.

The way we chose to power it was to plug it directly into one of the 20 amp sections of the PDP, then step down the voltage using an adjustable power supply off of Amazon.

Beware that the ODroid seems to get extremely hot when running for an extended period. It has a fan for cooling, but make sure the case has holes in it and that it is not placed in an area with little air flow.

For a good metric of power of the Odroid compared to the raspberry pi, check out the Liger Bots white paper which we used as reference: https://www.chiefdelphi.com/t/a-step-by-step-run-through-of-frc-vision-processing/341012



Sensors

IR Sensor

Our robot has a single IR sensor mounted on its intake. This IR sensor has a threshold set so that when any object is within a certain distance of the sensor, it is tripped. In this competition, it is used to register when a Cargo game piece has entered the intake.

Setup
The sensor is plugged directly into the RoboRIO DIO ports. Simply instantiate the IR Sensor as a DigitalInput with the DIO port as the construction parameter. Use the get() method to get the sensor's output as a boolean.

NavX Gyroscope

Placement
The NavX should be bolted directly to the drive base through the holes in its 3D printed case, and should be as close to the center of the robot as possible. We have not been using any of its magnetic capabilities so far, but if you were

to use them, it should not be located near any running motors or its readings will be off.

Reset vs. Zero
The reset() method should be called on the NavX at the beginning of the match (inside autonomousInit()). This zeros the NavX device itself, making it field centric for the match. From there on, the NavX should not be reset for the rest of the match. Instead, zero it from within code so that both a zeroed and field centric angle can be pulled off of it.

Holding a Field Centric Angle
Simply hold an offset value, and when zero() is called, subtract that offset value from the NavX reading for a zeroable yaw. This means that the NavX can be zeroed before autonomous driving or pivoting Commands to make the math for those simpler. The field centric yaw can still be accessed if needed by returning the direct yaw from the NavX without subtracting the offset

Ultrasonic Sensors

Our robot features 3 HC-SR04 ultrasonic sensors which are used to output the range to the nearest object it detects. In our case, these sensors were used to find distances nearby targets for use in our “Vision Drive” routine.

Setup

The ultrasonic sensors plug in directly to the RoboRIO, with two signal pins being used for the Echo and Trigger, respectively. In code, the Trigger Pin is instantiated as a DigitalOutput and the Echo Pin as a DigitalInput. These can then be passed in as parameters for a WPI Ultrasonic object. With one or more ultrasonic sensors, setAutomaticMode(true) has to be called once only on the last object instantiated. Once instantiated, the getRangeInches() method will return the output of the ultrasonic in inches.

LEDs

Decorative LEDs

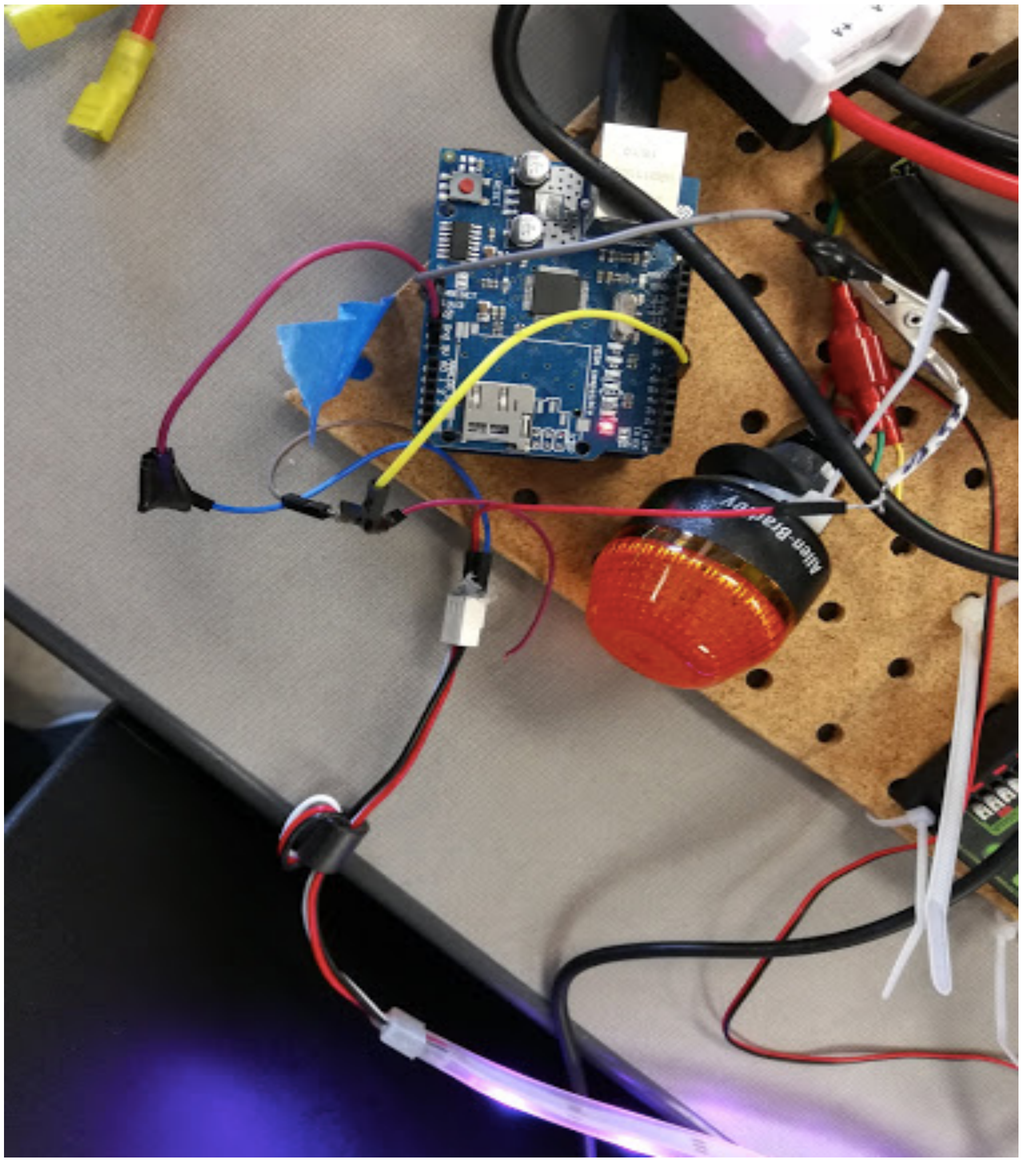
What You Need

- RoboRIO
- Arduino with Ethernet. This can be an Arduino with an Ethernet shield, or something like the Yun which has it built in

- Neopixel strip (WS2812 Integrated Light Source)
- Optional: Ethernet switch for extra Ethernet ports, so you can use Ethernet on RoboRio both for all your necessities and the fancy lights too

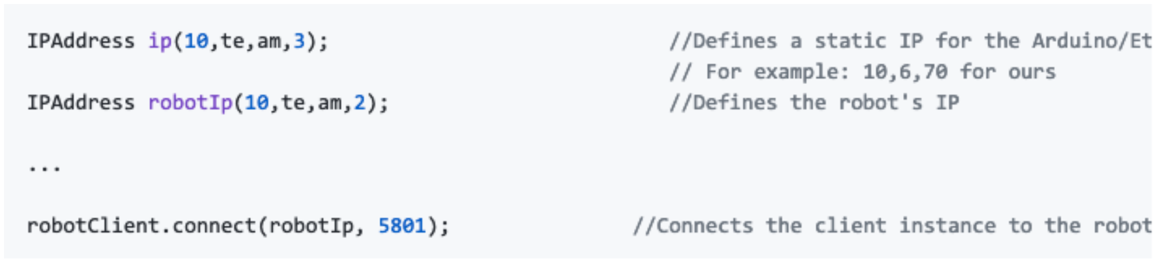
Setting Up for Your Team/Code Checklist

- Neopixel DATA pin connected to Arduino pin 7 (to use a different pin for data, change the value in the Arduino code). The Neopixels use RGB format at 800KHZ bitstream.



- Pin 7 (or whatever you defined previously) on Arduino/Ethernet shield to white wire (DATAIN) on Neopixel strip
- 5v pin on Arduino/Ethernet shield to red wire (VIN) on Neopixel strip
- GND pin on Arduino/Ethernet shield to black wire (GND) on Neopixel strip
- Arduino Ethernet shield to RoboRIO Ethernet (as long as it's somehow connected to Ethernet on the robot side)
- Arduino to 5V 500 mA power supply on the VRM. An alternative to powering from USB

In the Arduino code, to set up IP addresses:

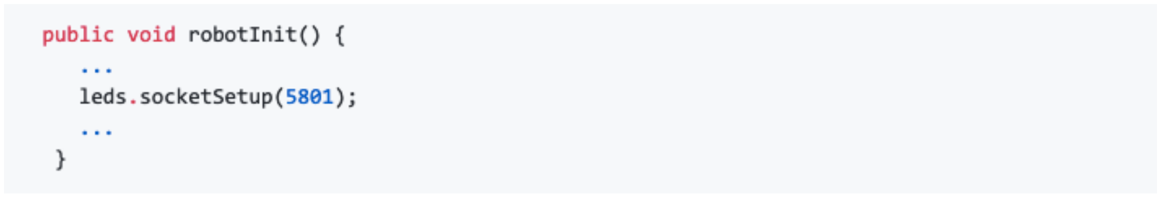


```
IPAddress ip(10,te,am,3);                         //Defines a static IP for the Arduino/Et
                                                  // For example: 10,6,70 for ours
IPAddress robotIp(10,te,am,2);                    //Defines the robot's IP

...

robotClient.connect(robotIp, 5801);             //Connects the client instance to the robot
```

On the RIO-side, make sure you have this in robotInit():

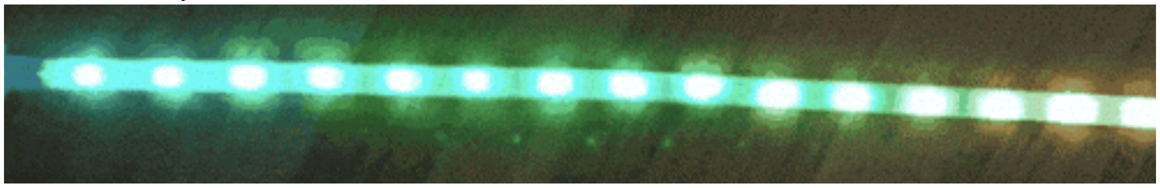


```
public void robotInit() {
   ...
   leds.socketSetup(5801);
   ...
 }
```

To run on RIO, deploy robotside code with gradle ./gradlew deploy and also upload the Arduino sketch to the Arduino.

Animations Used on 2019 Robot

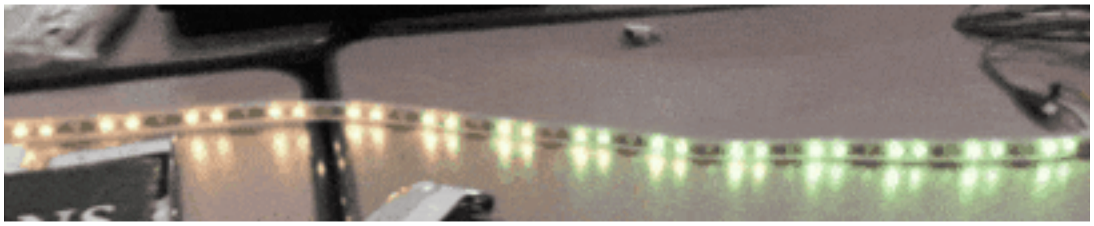
Rainbow Cycle



Theatre Chase

Meteor Rain

Rainbow Theatre Chase



LED Rings

Powering Through PCM

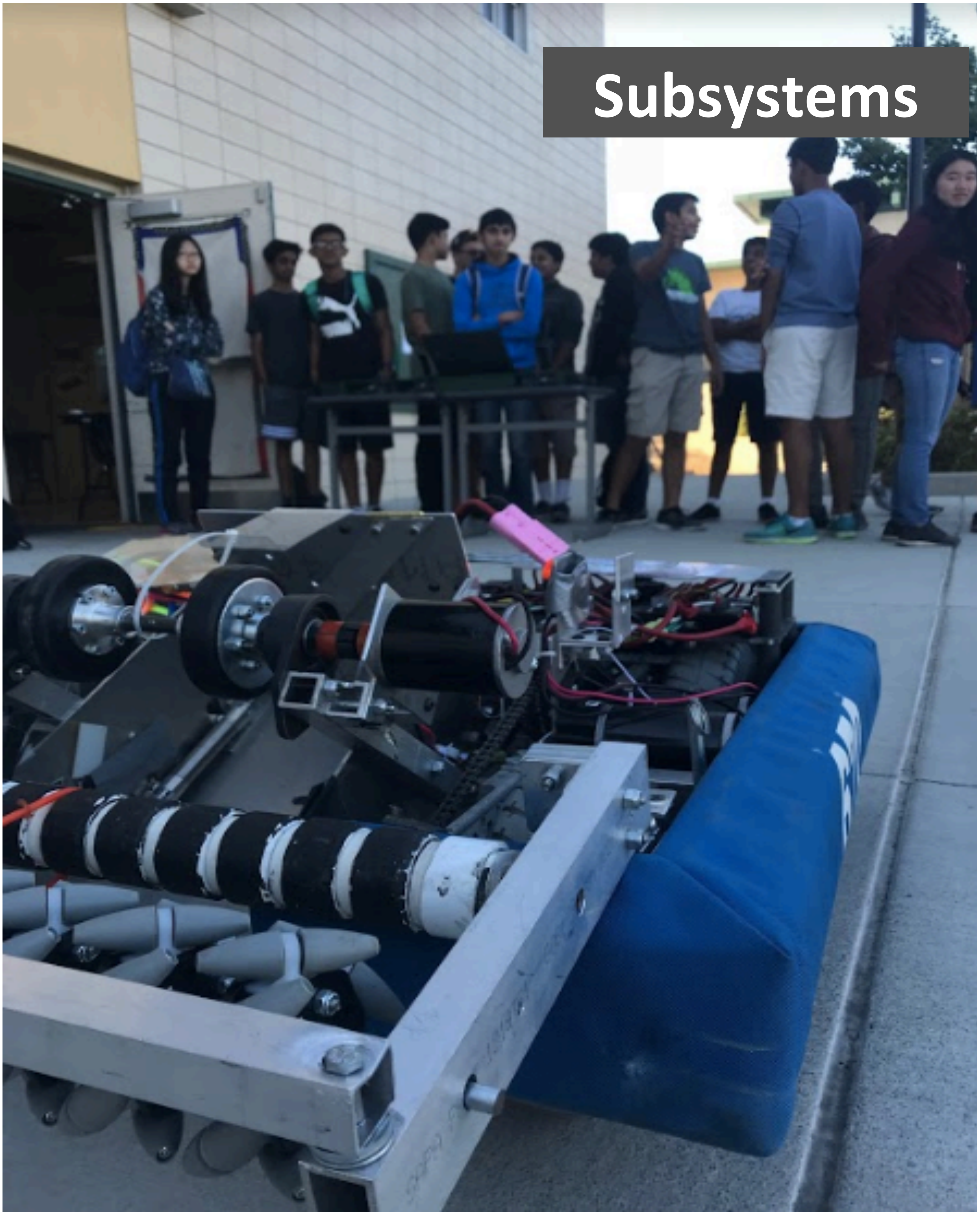
The LED ring we used from SuperBrightLEDs this year was extremely bright, in a 180 degree cone, and thus extremely annoying. Some judges may decide that it is too bright to be on constantly, and if not, at the very least it is annoying to the team when working on the robot while it remains on.

The easiest way to solve this dilemma is to set up a system to turn the LED ring on and off. There are two ways to do this: use a relay connected to the RoboRIO, or simply treat the LED like a Solenoid from the PCM. This year we chose to treat it as a Solenoid for simplicity since we decided to implement this after the build season ended.

The current 2019 PCM outputs two possible voltages from each channel: 12 or 24. Our LED ring uses 12V, but our LED ring requires 12, so we need to adjust it. The easiest way to do this is to plug a voltage adjuster similar to this: https://www.amazon.com/LOCHI-1-25V-28V-Regulator-Adjustable-Step-down/dp/B07L6CKXJN/ref=sr_1_1?keywords=voltage+adjuster+board&qid=1551848924&s=gateway&sr=8-1 into the PCM, and then plug the LED ring straight into that.

From there it is simple to instantiate a Solenoid object at the port the LED ring is plugged into and simply enable/disable it in code using the set() method when the LED ring is needed for targeting.

The one disadvantage to this is that the LED ring cannot be on when the robot is disabled since at that time the PCM will not output power, though this has no effect on actual competition matches, only testing.

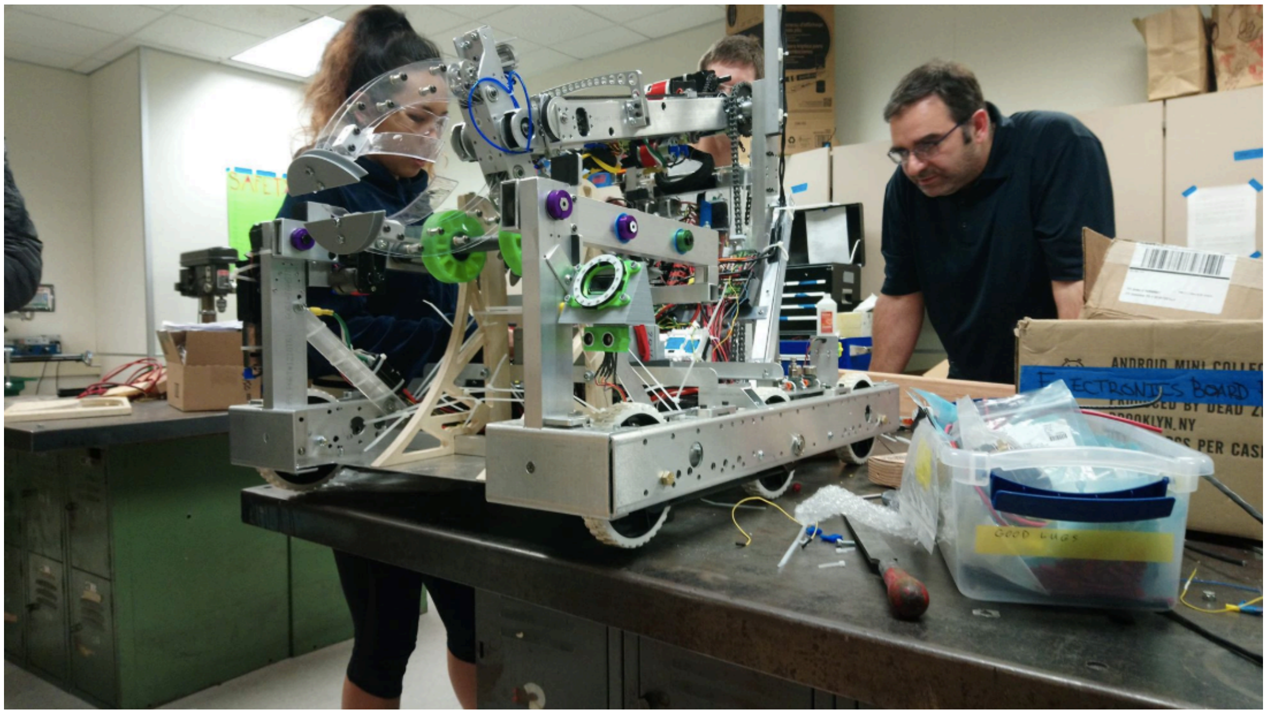


Subsystems

Subsystems

Our robot features the following subsystems:

- Arm
    - Elbow
    - Extension
    - Wrist
    - Claw
- Intake
- Drivebase
- Climber – Not Used (Code written)



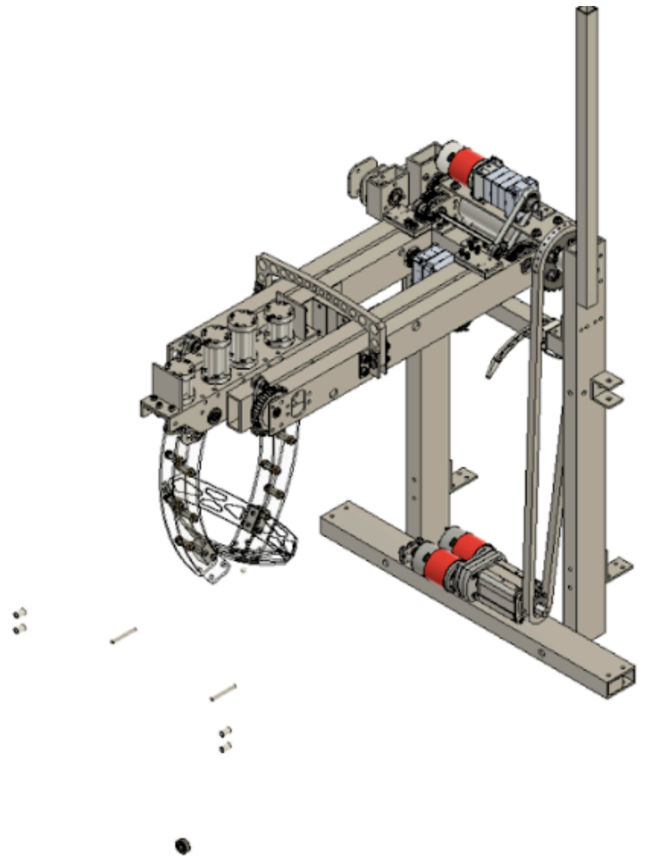
Arm

Arm State Machine

In order to maintain control of the arm on this robot, it is kept within a State Machine at all times to keep track of it while changing position. When a button is pressed to move the arm, the code looks at the current ArmState, performs a search for a path to the desired ArmState, and then executes it. This prevents it from colliding with the intake and allows it to maintain position and be moved around with simple button clicks rather than unwieldy joysticks.

Arm State Machine

- Arm
- LegalState
- ArmState - implements Node
- ArmTransition - implements Edge



Arm
Represents the actual Arm as a Subsystem. Instantiates a static HashMap containing all of the ArmStates and stores the current ArmState, its HeldItem, and the arm's coordinate position.

LegalState
A boolean representing each potential position for the arm that is used to look up each ArmState that each LegalState corresponds to.

ArmState
The aggregate of the angle of the elbow and wrist, the extension length, and the intake position. Examples are positions for the Low Hatch, Ball Intake, Cargoship Ball, etc. About 35 total were calculated and put in code to be used to track the arm and keep it in the correct position. One, and only one, subclass of this is instantiated for each possible position of the arm, each of which work as a Node for the breadth-first search and are passed around to be used in the search. Also stores an array of ArmTransitions that can be used to get from this ArmState to other ArmStates. If a PlaceOrGrab state, contains extra data that pertains to how far it extends from the robot for use with vision alignment.

ArmTransition
One, and only one, subclass of this is instantiated for each allowed transition between states of the arm. Many of these are CommonTransition, which simply moves each piece of the arm directly, while others can be defined specifically to prevent the arm from colliding with other parts of the robot. Contains a starting ArmState and an ending ArmState for use in the search. It also has to keep track of the intake's position to prevent it from colliding with the arm and completely destroying it while keeping it out the way of driving and placing. This ArmTransition is then executed using a well-tuned Motion Magic trapezoidal motion profile that prevents jerky movement. This had to be tuned for each of the 4 components separately, and when combined creates a smooth movement. Additionally, movement accounts for the extra weight of either game piece which may be held. A final parameter is the 30 inch size constraint outside the frame perimeter, which specific arm transitions keep the arm within if it is physically possible for it to leave the space.

Breadth First Search
A simple implementation of a breadth first search that finds the most optimized path from one ArmState to another by looking along the ArmTransitions extending out from the current ArmState and then continuing from there. It is generalized using the Node and Edge classes so that it could

be used for other purposes. Node is a point on a 2D graph, and Edge is a line between two Nodes on the graph.

Arm Tuning

Below is a verrrry messy copy-paste of the actual test plan used for the 2019 season.

Phoenix Tuner is a pretty useful tool! Find out how to use it here

0. Run through all gear ratios, encoder counts, and conversion equations with Ben and triple check them for the Wrist, Elbow, and Extension. Otherwise, you might have all kinds of uncontrollable bugs that will break the robot. If Ben is busy, hunt him down and hold him down because this is the most important thing.

0.5. Update all TalonSRX and VictorSPX Firmware, set CAN IDs properly, and update the CAN IDs to match in RobotMap

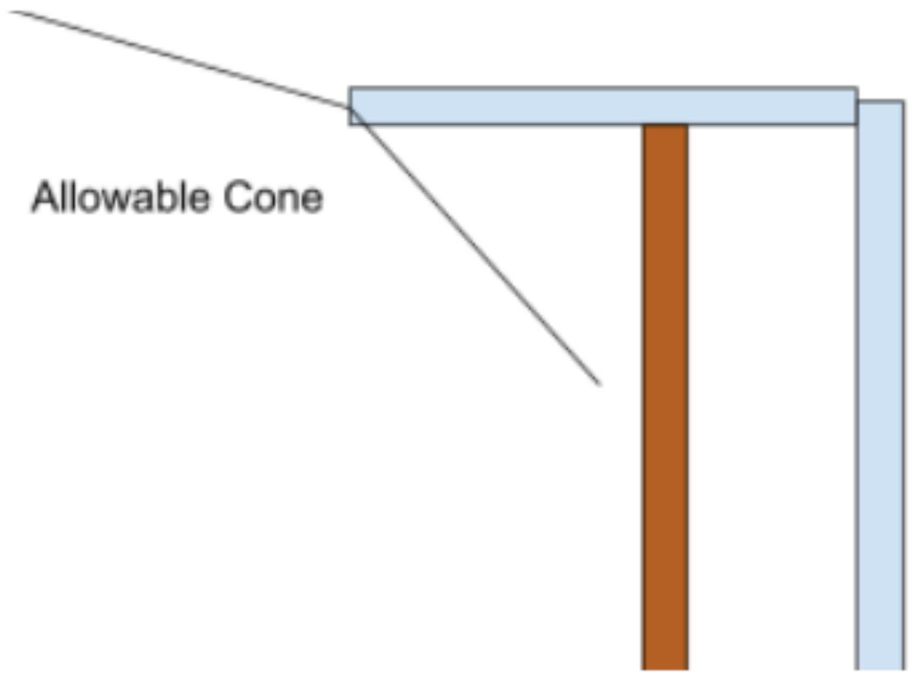
0.55 MAKE SURE THE CONSTRUCTOR IS GIVING ALL THE RIGHT VALUES

0.6 SET P TO 0 AND RUN WHILE HOLDING UP TO CHECK THAT FEEDFORWARD IS IN THE CORRECT DIRECTION

1. Tune Wrist with no Weight

- Make sure the constructor is passing in all the correct values to the super constructor. to do that:
- Find the absolute encoder value when wrist is flat - that's OFFSET_FROM_ENCODER_ZERO
- Find the limits that we want to have in each direction from 0 - that's FORWARD_SOFT_LIMIT and REVERSE_SOFT_LIMIT.
- Let's write down the true limits in the code in a comment and initially we'll restrict ourselves to some subset.
- Check RobotMap for CAN Bus numbers

- Determine the SensorPhase and MotorInversion
- Place the arm at a horizontal angle and prop it up with a block
- Limit the wrist's movement from going very far past vertical



- Place pool noodles on the top of the arm and on the block to protect in case it blows past limits
- Make sure the Wrist is in Brake Mode
- Use Arbitrary Feed Forward Calculator Command to figure out arbitrary forward to hold the wrist at horizontal
- SET P TO 0 AND RUN WHILE HOLDING UP TO CHECK THAT FEEDFORWARD IS IN THE CORRECT DIRECTION
- Tune PID Values until smooth motion is achieved that reaches and holds at the target
- Increase soft limits to allow wider range of motion. Test/adjust tuning
- Increase soft limits to full range of motion. Test/adjust tuning
- Estimate for max shaft rotation speed: 18730rpm * (1 minute/60 second) * (1/26 gearing) = 12 full (360 degree) rotations per second.

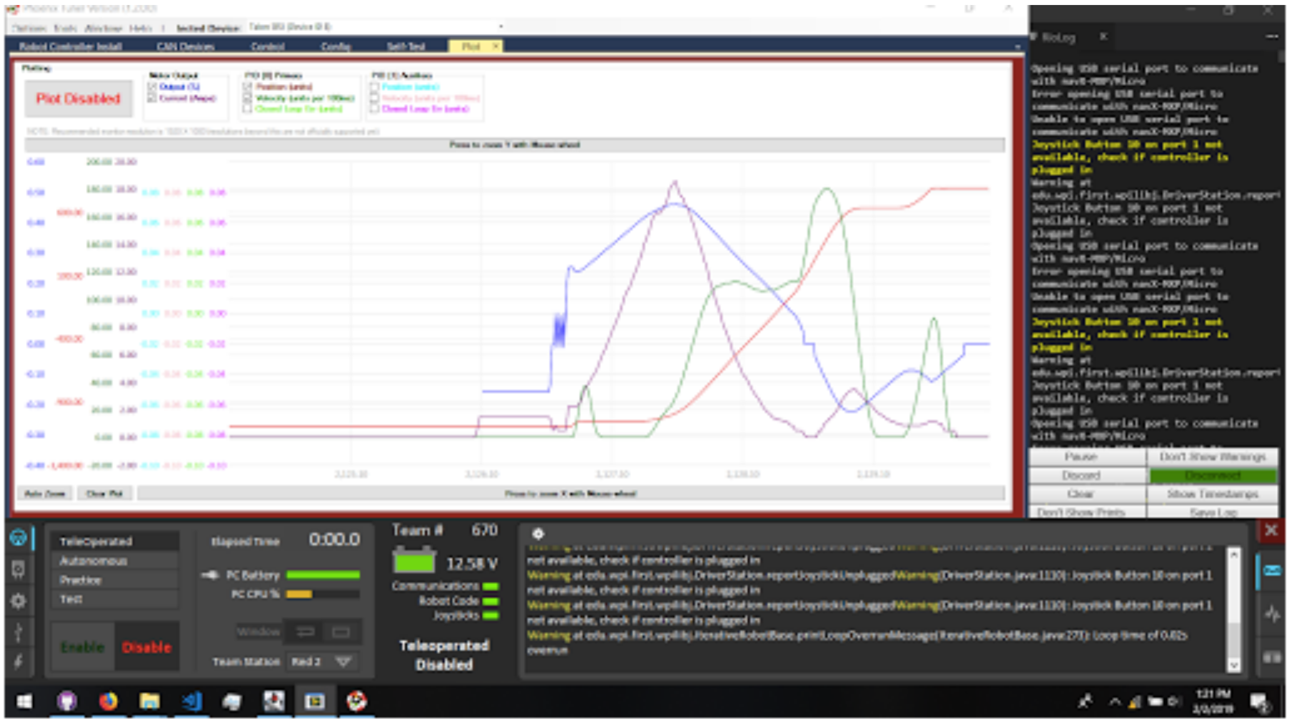
The wrist probably has a usable range of motion of somewhere between 180-270 degrees. We'll need far less than max speed.

- Approximation for max velocity: 1/8 of max speed = 600 units per 100ms
- Approximation for max acceleration: 6000 units per second?

2. Tune Elbow with no Weight and no Extension

- Make sure the constructor is passing in all the correct values to the super constructor
- Check RobotMap for CAN Bus numbers
- Determine the SensorPhase and MotorInversion
- Limit the arm's movement to a small cone near vertical using soft limits
- Place a stack of Pool noodles on top of a block on each side of the arm in case it blows past the soft limits
- Make sure the Elbow is in Brake Mode
- Get Arbitrary Feed Forward using the Arbitrary Feed Forward finding Commands
- **SET P TO 0 AND RUN WHILE HOLDING UP TO CHECK THAT FEEDFORWARD IS IN THE CORRECT DIRECTION**
- Tune PIDF Values until smooth motion is achieved
- Increase soft limits to allow wider range of motion. Test/adjust tuning
- Increase soft limits to full range of motion. Test/adjust tuning
- Estimate for max shaft rotation speed TO BE VERIFIED WITH BEN: 18730rpm * (1 minute/60 second) * (1/390 gearing) = 0.8 full (360 degree) rotations per second
- Approximation for max velocity: absolute max is 327.68 units per 100 ms. We can start with 150 units per 100ms.
- Approximation for acceleration per second: 3000 units per second?

After tuning, you should get a fancy output on Phoenix Tuner that looks something like this:



```
Opening USB serial port to communicate
with navX-MXP/Micro
Error opening USB serial port to
communicate with navX-MXP/Micro
Unable to open USB serial port to
communicate with navX-MXP/Micro
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Warning at
edu.wpi.first.wpilibj.DriverStation.report
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Opening USB serial port to communicate
with navX-MXP/Micro
Error opening USB serial port to
communicate with navX-MXP/Micro
Unable to open USB serial port to
communicate with navX-MXP/Micro
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Warning at
edu.wpi.first.wpilibj.DriverStation.report
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Opening USB serial port to communicate
with navX-MXP/Micro
```

```
not available, check if controller is plugged in
Warning at edu.wpi.first.wpilibj.DriverStation.reportJoystickUnpluggedWarning(DriverStation.java:1130): Joystick Button 10 on port 1
not available, check if controller is plugged in
Warning at edu.wpi.first.wpilibj.DriverStation.reportJoystickUnpluggedWarning(DriverStation.java:1130): Joystick Button 10 on port 1
not available, check if controller is plugged in
Warning at edu.wpi.first.wpilibj.IterativeRobotBase.printLoopOverrunMessage(IterativeRobotBase.java:273): Loop time of 0.02s
overrun
```

3. Tune Extension

- Make sure the constructor is passing in all the correct values to the super constructor
- Check RobotMap for CAN Bus numbers
- Determine the SensorPhase and MotorInversion
- Make sure the Extension is in Brake Mode
- Get Arbitrary Feed Forward using the Arbitrary Feed Forward finding Commands
- **SET P TO 0 AND RUN WHILE HOLDING UP TO CHECK THAT FEEDFORWARD IS IN THE CORRECT DIRECTION**
- Point the arm vertical and prop it up in that position
- Add in Soft Limits just inside the physical limits of the Extension
- Figure out: what max velocity and max acceleration do we want to target?
- Figure out the Arbitrary Feed Forward needed to hold up the Extension

- Use this Arbitrary FeedForward in updateArbitraryFeedForward(), note that this piece works somewhat opposite of the rotating pieces in that it needs more feed forward as it moves towards vertical, so use sine instead of cosine (based on angle of elbow)
- Tune PIDF Values until smooth motion is achieved
- Test both Reset Extension Commands
- Estimate for max shaft rotation speed TO BE VERIFIED WITH BEN: 18730rpm * (1 minute/60 second) * (1/35 gearing) = 8.9 full (360 degree) output shaft rotations per second. Question is how many rotations to move the extension from one limit to the other. Per Ben: 5.906 inches per rotation of the output shaft So that's a max speed of 52.68" per second, which is obviously way too much for an extension that is currently right about 12" long Let's aim to get to about 1 second end-to-end. And let's start at half of that. With the current limit switches (which still aren't wired so let's remember that), we have about 8000 ticks so call it 800 units per 100ms. So we'll start with 400 units per 10ms. Accel: 6000 units per second

4. Tune Elbow at Full Extension by Adjusting only Arbitrary Feed Forward

- Get new Arbitrary Feed Forward using the Arbitrary Feed Forward finding Commands
- Use this Arbitrary Feed Forward and the one at no extension to create a linearized equation for Arbitrary Feed Forward based on the extension length (use this calculation in updateArbitraryFeedForward() or alternatively just use it to scale the result of getArbitraryFeedForwardAngleMultiplier())
- Check that smooth motion is now achieved at full, half, and no extension

5. Run all of the MoveArm Commands using XKeys

- Set up Foam Bumpers on pieces of wood (many, many pool noodles) so if the arm blows past limits it will not smash itself to pieces

- Keep in mind that some of the Commands will not be bound by default to XKeys (the actual place commands and the ReadyToClimb state, and start states), so bind these to Xbox Controller buttons or something to run them. Full List of all ArmStates is in ArmStates.txt in the Subsystem package NOTE: The current method of using CommonTransition to move between all of the states by going through Netural probably works for now, but keep track of where it might hit the intake or the climber tubes and make note to edit those specific transitions
- Turn the peak motor output all the way up and run the move Commands again

6. Run everything with a hatch held. If it creates an issue or you miraculously have extra time(????), repeat Wrist tuning with a hatch held in the wrist. Then repeat Extension and Elbow the same way and have a check that changes the values when grabbing a hatch. If we need to do anything for the hatch or cargo, it's most likely/hopefully just computing a different arbitrary feedforward value for each component while the robot is holding a hatch or cargo. If that works correctly, the PID tuning should remain the same.

7. If Holding cargo also causes issues, repeat the same thing just with a cargo ball.

Driver Station

Dashboard

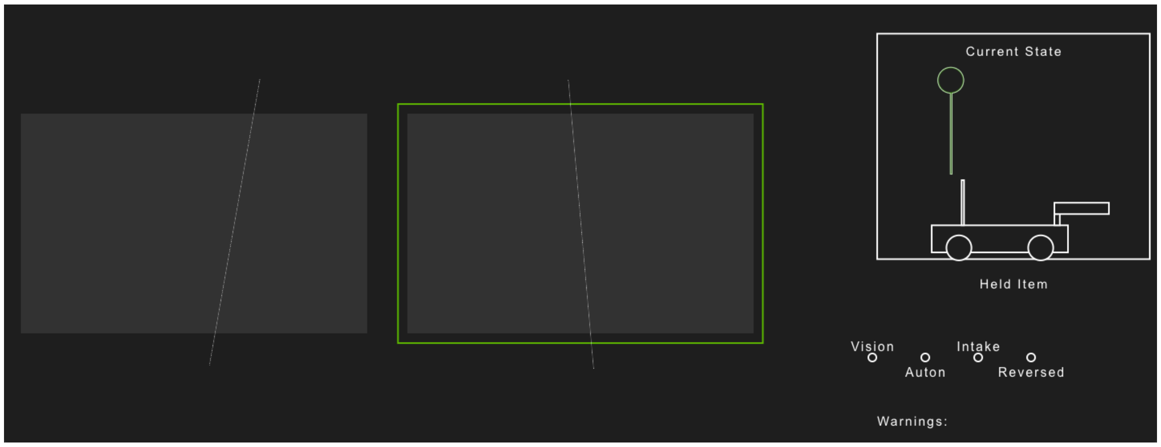
The Dashboard is a custom dashboard which is being used in place of SmartDashboard/ShuffleBoard.

We have a large space for viewing driver cameras which are streamed via mjpg-streamer from the Raspberry Pi. (*see the page on Driver Cameras for more details*)

In the top right is a real-time diagram of the robot. This is accomplished by continuously sending over network tables the elbow angle, the extension length of the arm, and the angle of the intake, all measured by encoders, reading those values in the dashboard, and updating the diagram accordingly.

The status of the claw is indicated by filling the claw in the diagram with different colors to represent the different states. These values are also sent over network tables and read by the dashboard.

The current ArmState is also displayed as text as is the current HeldItem, which represents what the claw thinks it is currently holding.



Driver Cameras

We are using 2 fisheye cameras placed on either side of the robot for driver vision. These cameras are plugged into the Raspberry Pi, and streaming starts when the Pi turns on (see the Raspberry Pi page for details). The cameras are streamed to the dashboard (see the page on Dashboard) using mjpg-streamer.

To set up mjpg-streamer:

1. Follow the instructions on https://github.com/jacksonliam/mjpg-streamer

Before using this setup in competition:

Set up a static IP address for the Pi

Refer to the Raspberry Pi page for details

Make sure the number in /dev/video for your camera is consistent

1. Plug in the desired camera to the desired USB port on the Pi
2. Run `ls -l /dev` in a terminal window - you should see an item listed video# where # is a single digit number, usually 0
3. Run `ls -l /dev/v4l/by-path` - you should see a long string that is symlinked to the video# you saw above
4. Set a symlink from the new location you want the camera to be to the string you see in by-path (we wanted our front driver camera to be `/dev/video40` and saw `platform-3f980000.usb-usb-0:1.2:1.0-video-index0` under by-path so we ran the following command to set a symlink between the two `sudo ln -s /dev/v4l/by-path/platform-3f980000.usb-usb-0:1.2:1.0-video-index0 /dev/video40`)
   Check that the symlink worked by running `ls -l /dev` again - you should see the new link you just created

Note that this link is deleted when the Pi reboots. To use this link multiple times, run the* `ln -s` *command either in any script that starts on bootup or in the* `~/.bashrc` *file

Also note that the link* `/dev/video40` *points to the specific USB port, and not the camera itself. So, make sure you plug the same cameras into the same ports each time. While this may seem like it defeats the purpose of this process, it is important to the streamer code.

Operator Controller – Not Used

Intro

Although the operator joystick was never used, code for it was still written. This code was held within JoystickElbow.java, JoystickExtension.java, JoystickWrist.java, and JoystickClimb.java.

ControlOperatorController.java

A command called ControlOperatorController was initialized in robotInit() within Robot.java. The ControlOperatorController command was uninterruptible and never finished. The command held an enum for three separates states that the operator controller could be in: ARM, CLIMB, and NONE.

Transitioning between these states was purposely made to be slightly difficult in order to make sure that the operator did not switch between these states accidentally.

Once the operator did switch into a state, the joysticks were mapped to different subsystems. In the ARM state, the right stick controlled the elbow, the left controlled the wrist, and the triggers controlled the extension. In the CLIMBER state, the right stick controlled the back pistons and the left stick controlled the front pistons.

Joystick Commands

Each joystick command had similar logic. The command was instant and took in a squared input from the ControlOperatorController class. This input was then used to scale the maximum/minimum output of its respective subsystem in order to control the subsystem's movement.

A tolerance is set at a specified distance away from the soft limits set. Once the encoder hits this tolerance, a linear scalar is multiplied by the output, slowing the movement until it hits the soft limit.

Intake

Auto Intake Routine

Intro

The AutoIntake sequence is used to take in a ball from the front of the robot. It utilized both the intake and the arm, as well as an IR sensor mounted on the intake itself.

Routine

The sequence for AutoIntake is controlled using a CommandGroup: AutoPickupCargo.java. The steps it uses are as follows:

- If the claw is closed, it is opened in parallel.
- The arm is then moved to a state in which it is ready to receive a ball from the intake in sequential. This state also calls for the intake to be deployed.
- The command RunIntakeWithIR is called in sequential.
    - This command runs the intake rollers in until an IR sensor mounted on the intake is tripped. Following this, the intake continues to run for 0.55 seconds, effectively keeping the ball at the top of the intake for the claw to finish picking up the ball.

- A TimedRunIntake command is called, which runs the rollers in for 0.6 seconds, the pre-determined time for the claw to open or close. This is called in parallel at a reduced speed.
- The PickupBall command is called in sequential, which closes the claw and sets the HeldItem enum to BALL.
- The Intake rollers are stopped
- The arm is moved back into Neutral, which also brings the intake back in.

Climber – Not Used

Climbing with Tilt Control – Not Used

Intro
The climber for the 2019 Destination: Deep Space robot was ultimately not finished in time for competition season. However, code for the subsystem was still written in advance. The climber subsystem, as seen below, featured two sets of pistons (front and back), controlled by motors connected to Talon SRX's.

Climbing Routine
The climber was designed to reach both HAB Level 2 and HAB Level 3. The routine for climbing is as follows:

- The driver drives the robot up to the platform and aligns it properly by "ramming" into the platform itself
- The pistons the motors drive both pistons down until the robot reaches the appropriate height (Level 2 or Level 3)
- The arm on the robot comes down onto the platform and pulls in, bring the front wheels of the robot onto the platform (this step is repeated as many times as necessary until the front wheels are safely on)
- The front pistons are retracted and the driver drives the wheels until the back pistons hit the platform
- The back pistons are retracted and the driver finishes driving the robot onto the platform

User Control of the Climb
Control of the climb is done through CycleClimb.java. The stages of the climbing process were controlled through an enum: ClimbStage. The operator was given a button to cycle through these stages. Each time the

button was pressed called the respective command and switched the enum to the next state. These states included:

- DEPLOY_PISTONS: Moved both pistons down to bring the robot up to the correct height.
- ARM_CLIMB: Executed the arm movement to drag the robot onto the platform. (This movement would be called repeatedly until the operator called the "CANCEL_ARM_CLIMB" command
- RETRACT_FRONT_PISTONS_AND_STOW_ARM: Retracted the front pistons and moved the arm into the "STOW" position
- RETRACT_BACK_PISTONS: Retracts the back pistons

Climbing with Tilt Control

Climbing with tilt control was called with the command: PistonClimbWithTiltControl.java. This command set setpoints on the two WPI PIDControllers instantiated with the front and back TalonSRX's. A third PIDController was used to minimize "tilt error" based on input from the NavX sensor mounted on the drivebase. This PIDController returned outputs that were used to adjust the maximum and minimum outputs of the PIDControllers on the front and back pistons. For example, if the NavX returned a pitch of -2 degrees (meaning that the front of the robot dipped downward), the tilt PIDController would return a value to increase the maximum range of the front PIDController, increasing it's maximum output to compensate for the forward tilt.

Motor Controllers

SparkMAX Drivebase Controller Setup

Before using the SparkMAX, make sure to flash the correct firmware and set the ID properly on the controller. This can be done using the client application. Of course, before doing that make sure to plug the Spark properly into the PDP and put it in the CAN loop properly. Once this is done, the Spark be solid cyan. Refer to the blink code chart below.

Instantiation and Setup

Instantiate each controller as a CANSparkMax object with the Spark ID and either CANSparkMaxLowLevel.MotorType.kBrushless or CANSparkMaxLowLevel.MotorType.kBrushed as parameters depending on the motor being connected. **It is extremely important to set brushed or brushless properly as doing this improperly may damage either the controller, the motor, or both.** Once this is done, follow the steps below:

- **SetInverted**: The method setInverted() inverts the output of the controller.

- **PID Control**: Each controller has built-in PID controllers. Position and velocity control can both be used by utilizing multiple slots. Output ranges should be set with the method setOutputRange() on the PID Controller.

- **Ramp Rate**: Ramp rate can be set with the method setRampRate(). Use this to prevent stripping out gearboxes. If it is 0, the robot will stop almost on a dime if it is light and in brake mode, but gear boxes will suffer. We recommend at least 0.25 and more if the robot is heavy.

- **Built-In Encoder**: Each SparkMAX Controller has a built in encoder which can be called with the method getEncoder(). Calling getPosition() on this encoder will return position in revolutions and getVelocity() returns RPM but this can be scaled using setVelocityConversionFactor().

- **Speed Controller Group/Differential Drive**: The controllers on either side can be put into a WPI SpeedControllerGroup, which together can be used to make a WPI DifferentialDrive object. Any drive commands can then be called on the DifferentialDrive object such as TankDrive or CurvatureDrive. A max output can be set on the Differential Drive object with the method setMaxOutput().

TalonSRX Setup

Before using the TalonSRX, make sure to flash the correct firmware and set the ID properly on the Talon. This can be done using the Phoenix Tuner. Of course, before doing that make sure to plug the Talon properly into the PDP and put it in the CAN loop properly. Once this is done, the Talon should flash orange. Refer to the blink code chart below. *Note: if two Talons are set to the same ID, they may show up as only Talon. As such, when making a new electronics board, it might make more sense to add each Talon on individually.*

Instantiation and Setup

Instantiate a new TalonSRX object and pass in its ID as a parameter. Following this, the rest of the recommended setup steps are listed below:

- **Factory Default**: The Talon can have all of its configurations set to the default factory values using the method configFactoryDefault(). This effectively takes out the need to set every parameter on the Talon.

- **SelectedFeedbackSenso**r: The connected feedback sensor can be set with the method (configSelectedFeedbackSensor). Passing in either FeedbackDevice.CTRE_MagEncoder_Relative or FeedbackDevice.CTRE_MagEncoder_Absolute corresponds to a relative or absolute CTRE Mag Encoder respectively. This can then be used later for Motion Magic or other control loops.

- **Current Limits**: Both a continuous and a peak current limit can be set on the TalonSRX. The continuous limit serves as a lower limit for the current and can be set with the method configContinuousCurrentLimit(). The current is allowed to exceed this limit set for the amount set with the method configPeakCurrentDuration(). The current will never be allowed to exceed the peak current set with configPeakCurrentLimit(). If the peak current limit is set to 0, the continuous current limit will act as the maximum current allowed.

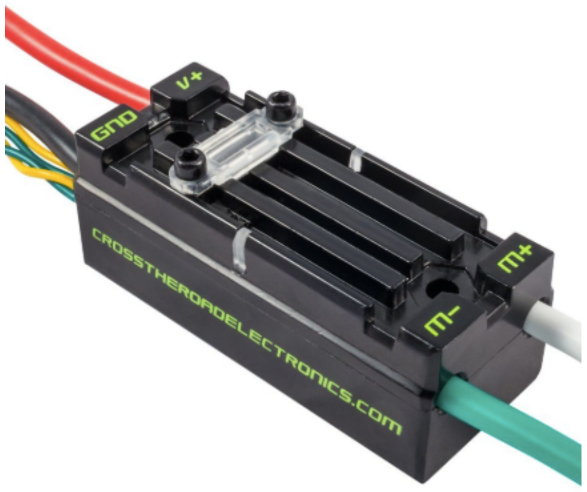
- **Soft Limits**: Forward and reverse soft limits can be set with configForwardSoftLimitThreshold() and configReverseSoftLimitThreshold(), respectively with the forward limit being the more positive tick value. These soft limits should then be enabled with the methods configForwardSoftLimitEnable(true) and configReverseSoftLimitEnable(true).

- **Follower**: If this Talon is being set as a follower, this can be done with the method follow() with the master Talon passed in as a parameter. More information about followers can be found on in the "Control Mode" section.

- **Closed Loop Control**: For closed loop control (in our case, Motion Magic), first a profile slot has to be set with the method, selectProfileSlot() with the slot number passed in. Following that, the PID, Feed Forward, and I-Zone constants can be set for that slot. For Motion Magic, cruise velocity and acceleration should be set. Peak and nominal outputs can then be set, with peak output representing the strongest output allowed and nominal representing the weakest. Finally, the allowable error for the control loop can be set with configAllowableClosedloopError().

- **Set Sensor Phase**: The feedback sensor set above should increase when the Talon gives a positive output. If the two don't correlate then the setSensorPhase method should be called with either false or true such that both the Talon and the sensor are in sync.

- **Selected Sensor Position**: The method getSelectedSensorPosition() should be used to get the output of the encoder connected to each respective Talon. Use this instead of getQuadraturePosition().



TalonSRX Control Modes

Note: all control modes run at a rate of 1000 times per second by default, so using Onboard Talon control modes will always be more accurate than doing stuff from the RoboRIO if tuned well.

Percent Output

- Uses PWM control, the most basic way to control a motor
- Good for linear movement, likely the best option for using Joystick input to control a mechanism

Position

- Uses encoders (or any other type of sensor you can plug into the TalonSRX) and tries to servo the mechanism to the position using a PIDF loop (make sure to tune these and set them on the motor)
- Often not very well controlled unless the loop is tuned well
- An issue we have often had with this method involved being unable to set the sensor phase of the encoder (reverse it). Make sure you use talon.getSelectedSensorPosition() when getting encoder position and not talon.getSelectedSensor.getQuadraturePosition() and you will avoid this and many other issues!!!!

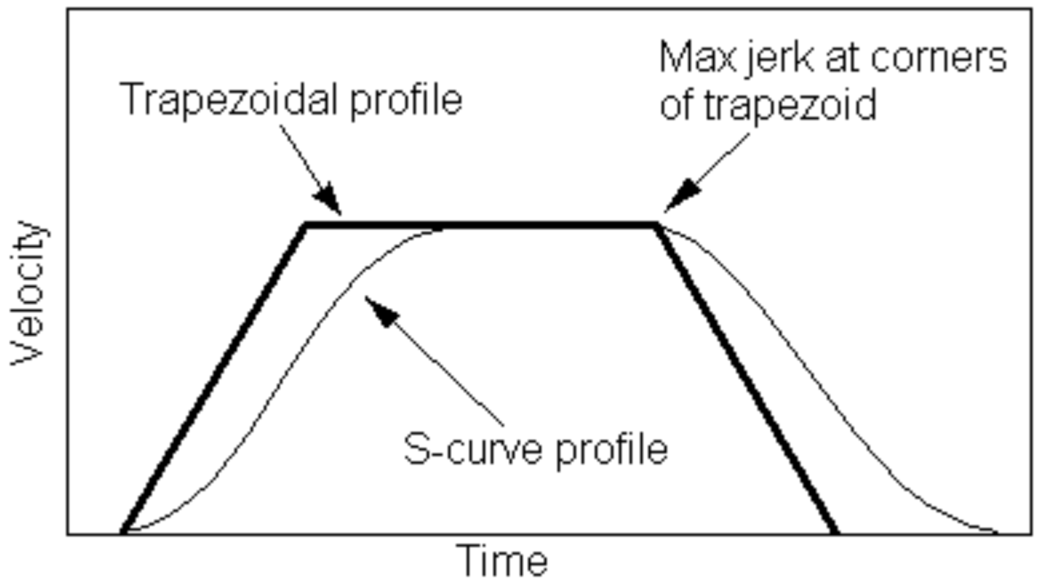
Arbitrary Feedforward (also useful for basically all other control modes)

- Arbitrary Feedforward is a value that is always output to the motor no matter what (use this to counteract a constant force like gravity)
- The easiest way to do this if the mechanism is moving is to have a WPI Notifier object (essentially a separate Thread) that runs constantly and updates the Arbitrary Feedforward on the motor. If you were counteracting gravity, you would calculate and store how much you Motor output [0, 1] you need to hold the mechanism perpendicular to

the ground (most strength you need), and then multiply it by the appropriate trig function (sine/cosine) of the angle to the ground based on where zero is so it scales the feedforward appropriately.

Motion Magic

- This is our control mode of choice for moving the arm and intake on the 2019 robot, and should be used for any task that requires precisely moving a component. Essentially, the TalonSRX uses a PIDF loop to move a component, extremely similarly to Position control. The main difference is that this mode either uses a Trapezoidal Profile or an S-Profile (depending on how you set it up). This means that the output of the motor ramps up smoothly, maintains, and then ramps down as it nears its target. This is nice for not slamming fragile components around and creating smoother movement with less likelihood of chains or belts slipping. Almost all of it is handled by the controller, you just pass in values and tune it (see our Arm Tuning page on this wiki).



- An issue we have often had with this method involved being unable to set the sensor phase of the encoder (reverse it). Make sure you use talon.getSelectedSensorPosition() when getting encoder position.

Motion Profile

- I never got this working well and ended up using Pathfinder instead, however it is there. The concept is that you stream a bunch of different waypoints to the Talon and it tries to essentially follow them. You can either load all points and then start driving them, or load as you drive once you've given it a certain number so it won't blow past how many you are streaming.

Velocity

- Controls the motors to maintain a certain velocity using feedback from a sensor. This is useful for something like a PurePursuit driving algorithm, where you calculate a speed to maintain and then tell the motors to maintain it to follow the path.

Current

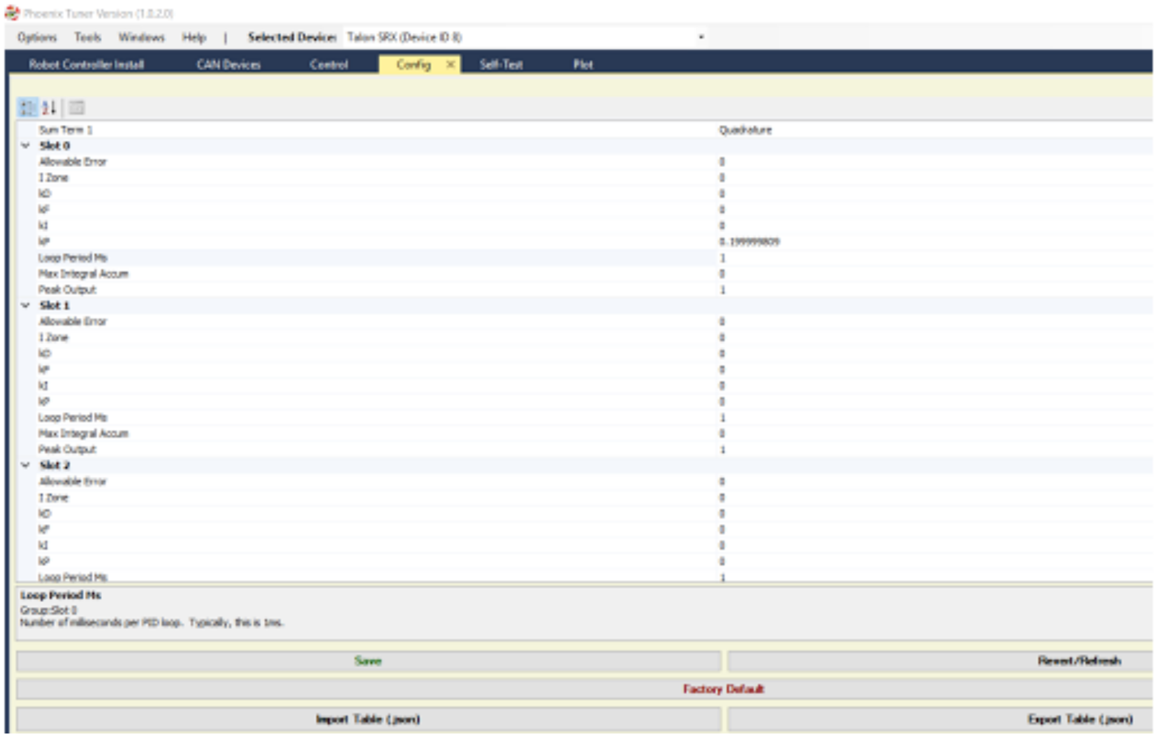
- Current control is very nice for exerting a certain amount of force from a mechanism, or performing linear movement without burning out the motor (though you should use a current limit to do this anyways)
- Giving a negative value for current makes the motor run backwards, while positive will go forwards

Follower

- Extremely simple, the motor outputs exactly the same as the motor who's ID is passed into the set() method when Follower is given. Note that a VictorSPX can be slaved to a TalonSRX when you want the SRX only features (Current Control) while still using a cheaper controller, however the reverse is not currently possible.

Phoenix Tuner

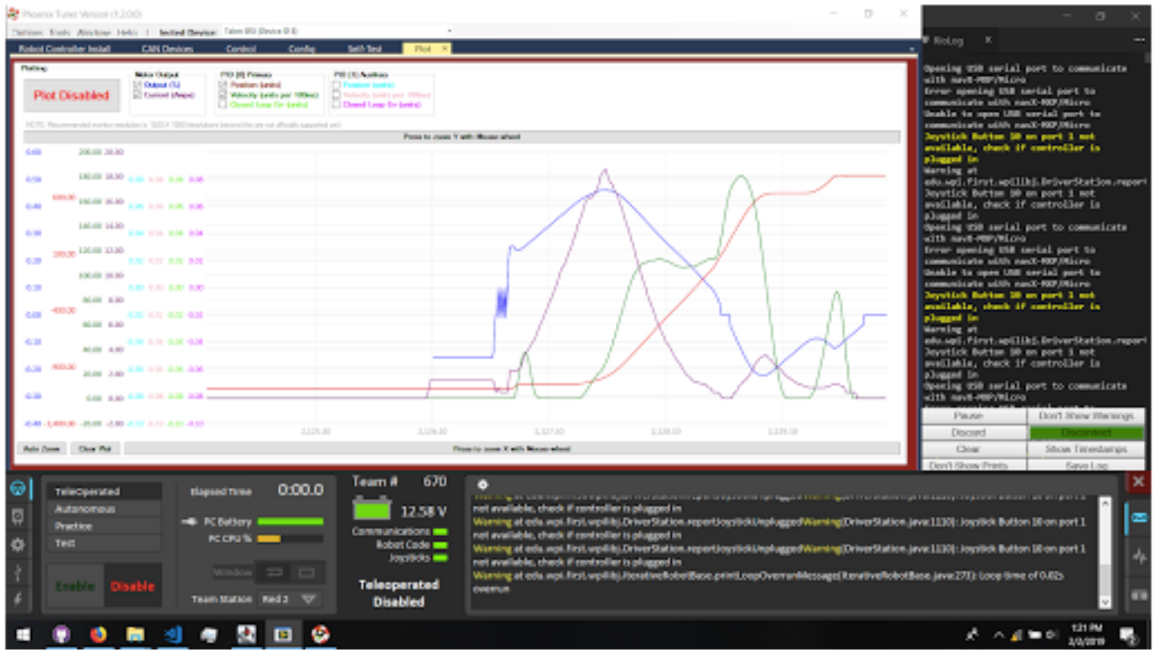
The Basics


Phoenix Tuner Version (1.0.2.0)
Options Tools Windows Help | Selected Device: Talon SRX (Device ID 8)
Robot Controller Install
CAN Devices
Control
Config
Self-Test
Plot

Sum Term 1	Quadrature
Slot 0	
Allowable Error	0
I Zone	0
kD	0
kF	0
kI	0
kP	0.199999809
Loop Period Ms	1
Max Integral Accum	0
Peak Output	1
Slot 1	
Allowable Error	0
I Zone	0
kD	0
kF	0
kI	0
kP	0
Loop Period Ms	1
Max Integral Accum	0
Peak Output	1
Slot 2	
Allowable Error	0
I Zone	0
kD	0
kF	0
kI	0
kP	0
Loop Period Ms	1

Loop Period Ms
Group:Slot 0
Number of milliseconds per PID loop. Typically, this is 1ms.

Here is a nice output for testing with added weight:



```
Opening USB serial port to communicate
with navX-MXP/Micro
Error opening USB serial port to
communicate with navX-MXP/Micro
Unable to open USB serial port to
communicate with navX-MXP/Micro
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Warning at
edu.wpi.first.wpilibj.DriverStation.report
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Opening USB serial port to communicate
with navX-MXP/Micro
Error opening USB serial port to
communicate with navX-MXP/Micro
Unable to open USB serial port to
communicate with navX-MXP/Micro
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Warning at
edu.wpi.first.wpilibj.DriverStation.report
Joystick Button 10 on port 1 not
available, check if controller is
plugged in
Opening USB serial port to communicate
with navX-MXP/Micro
```

```
not available, check if controller is plugged in
Warning at edu.wpi.first.wpilibj.DriverStation.reportJoystickUnpluggedWarning(DriverStation.java:1130): Joystick Button 10 on port 1
not available, check if controller is plugged in
Warning at edu.wpi.first.wpilibj.DriverStation.reportJoystickUnpluggedWarning(DriverStation.java:1130): Joystick Button 10 on port 1
not available, check if controller is plugged in
Warning at edu.wpi.first.wpilibj.IterativeRobotBase.printLoopOverrunMessage(IterativeRobotBase.java:273): Loop time of 0.02s
overrun
```

Steps for Setting Up

1. Download everything you need here

2. Run the application, connect RoboRIO to PC over USB

3. Select 172.22.11.2.:1250 in the Diagnostic Server Address Bar

4. Click "Install Phoenix Library/Diagnostics" and make sure firmware is updated in CAN Devices tab

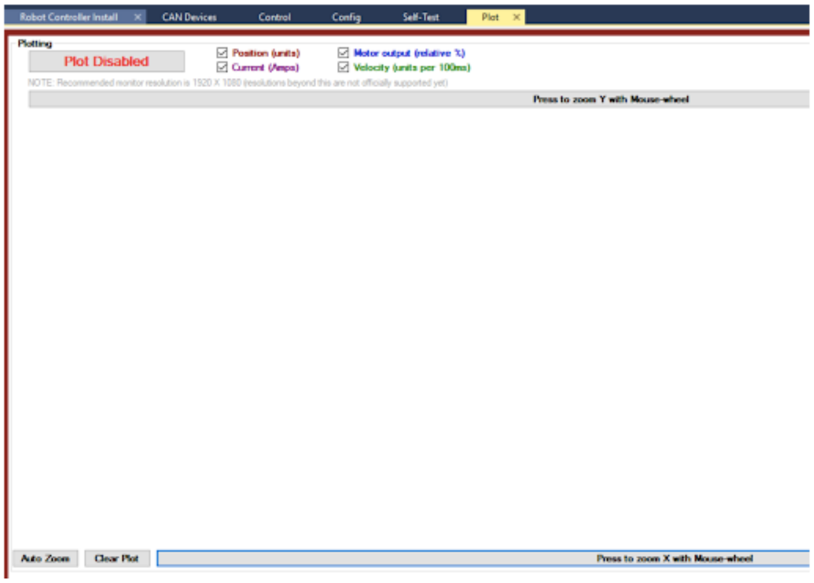
On to tuning:

1. Confirm motor & sensor health, sensor phase

2. Get max sensor velocity, manually move motor to find encoder range (hard limits)

3. If encoder range values are very unreasonable out of bounds, make sure necessary calculations (such as bitwise operations) are done to adjust encoder ranges reasonably. Make sure soft limits are reasonable or not interfering with commands.

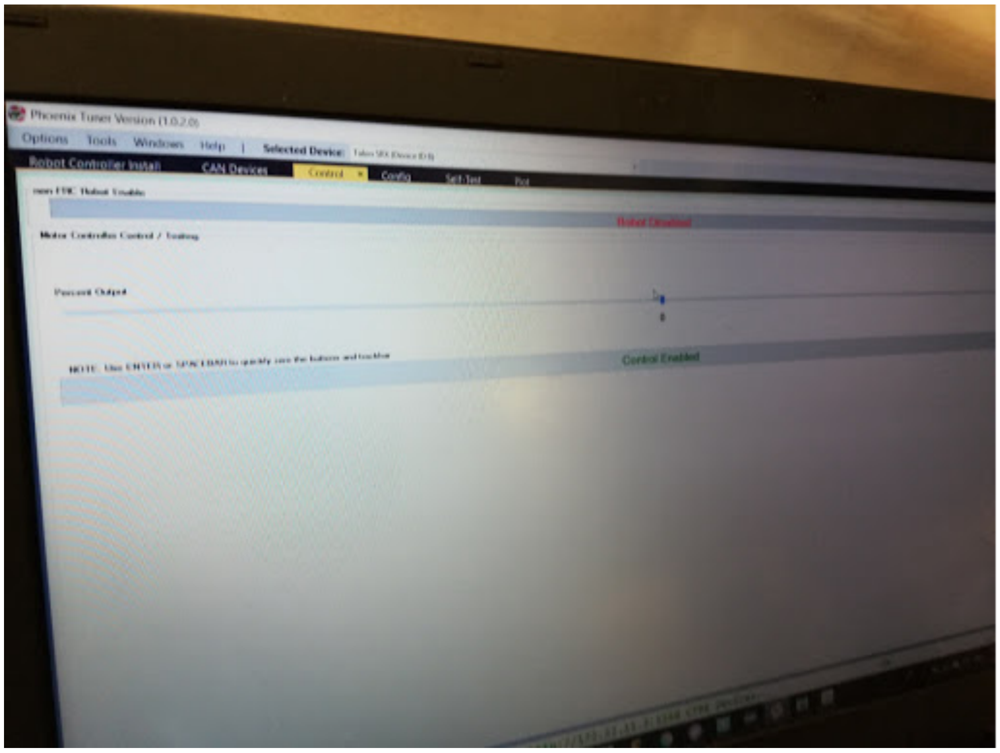
4. Go to the Plotter on Phoenix Tuner to see position, output, etc



NOTE: Recommended monitor resolution is 1920 X 1080 (resolutions beyond this are not officially supported yet)

5. Configure gains
   - Set all gains to zero (make sure the slot is correct (0 for now))
   - If not using position-closed-loop, set kF to your calculated value
   - Set initial cruise velocity and acceleration. A reasonable initial cruise velocity is 1/2 the max
   - units are ticks per 100 ms
   - Initially cap the outputs very low to minimize problems because problems later suck
   - Deploy the application, use command (currently something like MotionMagicSetpoint) to adjust your target ticks

6. If using Motion Magic mode, DO NOT PRESS CONTROL ENABLED (for Percent Output, interferes with Motion Magic). Using command will move motor and motor connected to the Talon will respond accordingly. In newer versions of the Phoenix Tuner, Motion Magic support has been added into the tuner, so you can enable that mode from this screen instead of through the robot code. Make sure control is enabled, should be on this screen:



- If using percent output, each tick on Control is 0.05

All closed-loop modes update every 1ms

7. "Self Test" to get readouts of values (check mode is Motion Magic)

- if motor is not responding to commands, check error values in Self Test from the encoder

Implementing Arbitrary Feed Forward

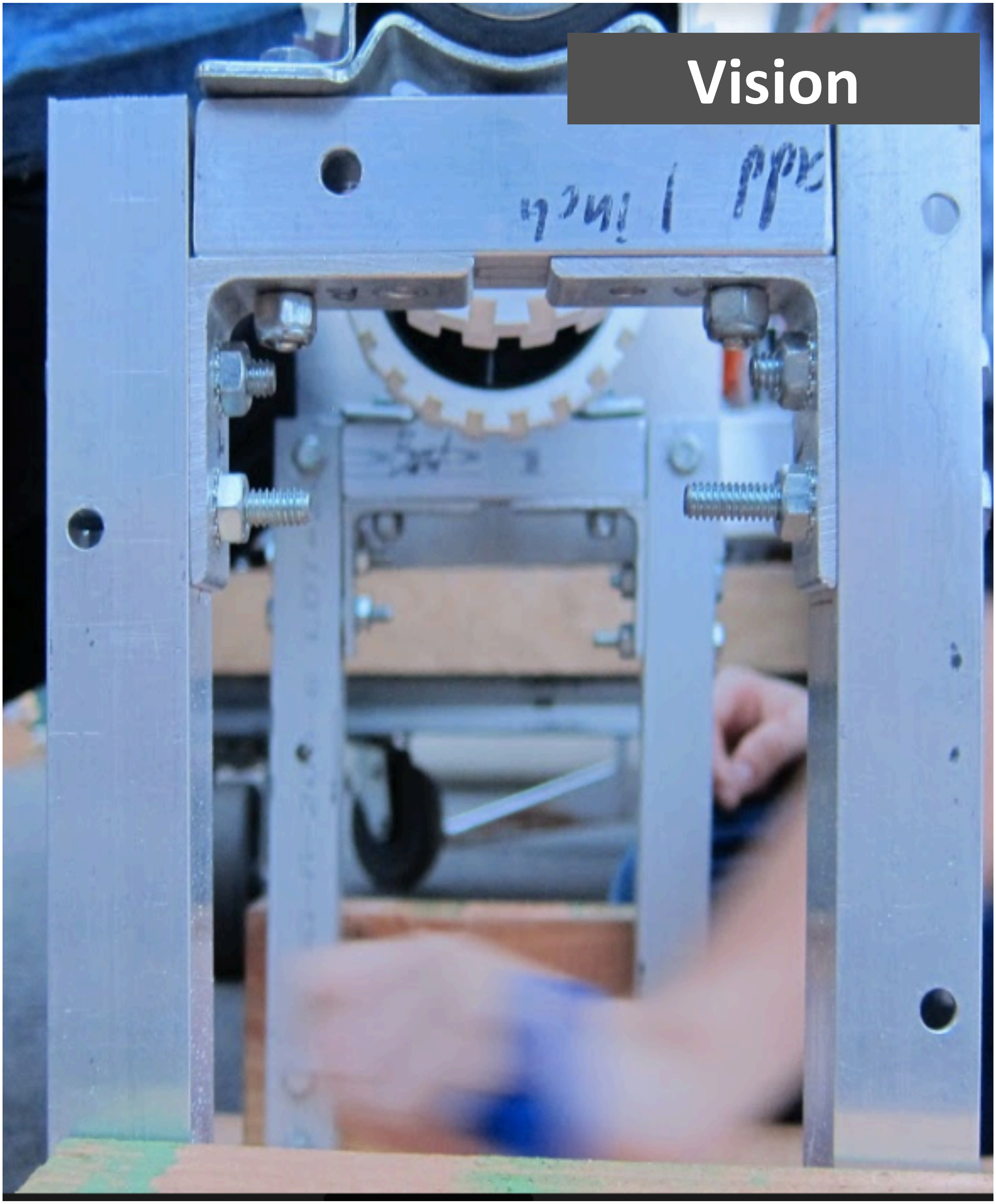
Arbitrary Feed Forward is essentially a value that the motor will take into account to counteract gravity and ensure smooth, controlled movement.

1. Create two commands for increasing/decreasing motor percent output value.
2. Have another command for running the motor at percent output value for short span of time. Continue changing output value until command barely keeps arm from falling.
3. Update constant in appropriate subsystem class with final percent output value, test several timesto ensure constant works well

Things to check

- Make sure the Talon shows up under CAN devices!
- Set the current limit very low when testing because fried parts are no bueno.

Vision



Vision

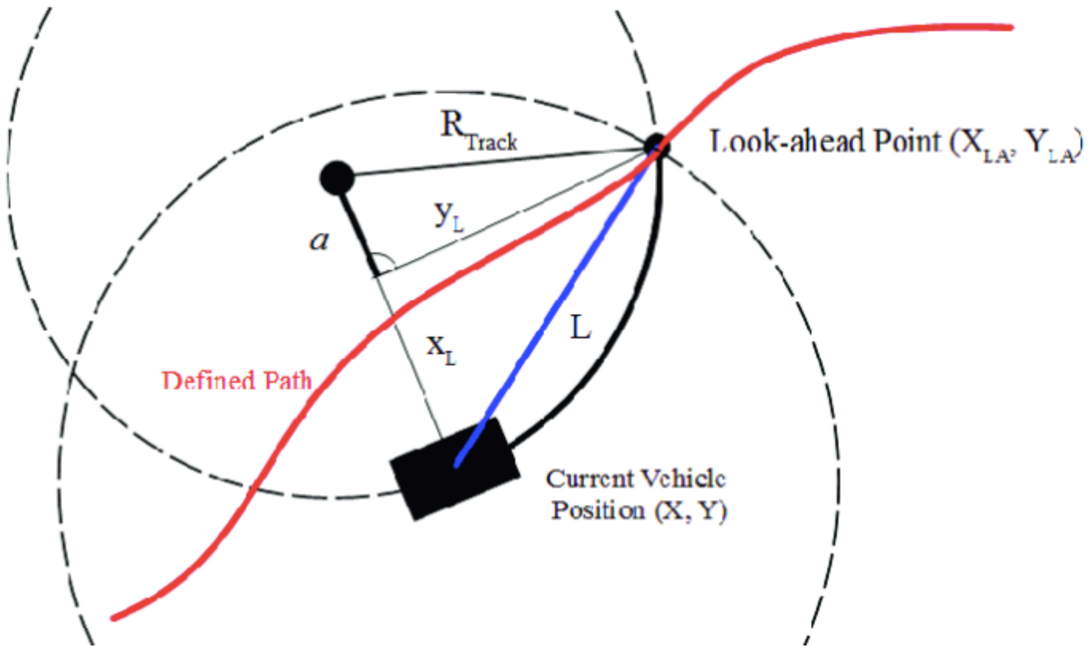
Our robot utilizes vision through OpenCV on our Raspberry Pi in combination with the Pure Pursuit path-tracking algorithm in order to drive toward and align with vision targets.

Pure Pursuit

What is it?

Pure pursuit is a path tracking algorithm. Basically, with pure pursuit, you generate target velocities based on where the robot is, relative to the path it wants to follow. One way to think about it: the robot is "chasing" a moving point ahead of it, "looking ahead" some distance to *pursue* the point. The benefit of this method is that paths can adjust; since it moves based on location in relation to a target path, the robot can correct itself if it is disturbed while moving.

This is what the algorithm is doing (image from P. Sanjeevikumar):



R_{Track}
Look-ahead Point (X_{LA}, Y_{LA})
y_L
a
x_L
L
Defined Path
Current Vehicle Position (X, Y)

Details About Implementation

- lookaheadDistance is how far along the path we should "pursue"
- maxVel and maxAccel are in inches/s and inches/s^2 respectively.
- maxVelk is proportional to how fast we turn
- Feedback is supported, where kP should be around 0.001. With a higher kP, we can better compensate for being off the targeted velocity, but be aware that too large of a kP will cause oscillations
- To smooth out the paths, a = 1-b and both a and b should be between 0 and 1. Ideally b should fall in the 0.7-0.9 range. The higher the b value, the smoother the path.
- Spacing is the distance between points along the path: the path generator will inject points along the coordinates you specify, at the given spacing.
- Tolerance is the maximum change for each "smoothing" operation. If tolerance is exceeded, the smoothing algorithm runs again, continuing until the change in the path is below the tolerance. If tolerance is too low, the smoothing might never converge, so try raising the tolerance if this occurs.

Vision Camera Setup

The Vision Cameras consist of 2 Microsoft HD Lifecam 3000s - one on the front and one on the back. Each one is fitting in a mount with a green LED ring surrounding it

The vision targets this year are retroreflective tape, meaning that they reflect a bright green when the LED ring shines on it, allowing the camera to easily detect it with calibrated settings.

Vision Calculations

This page briefly explains some of the calculations used by the vision system to adjust certain inputs to a more desirable format.

Φ calculation

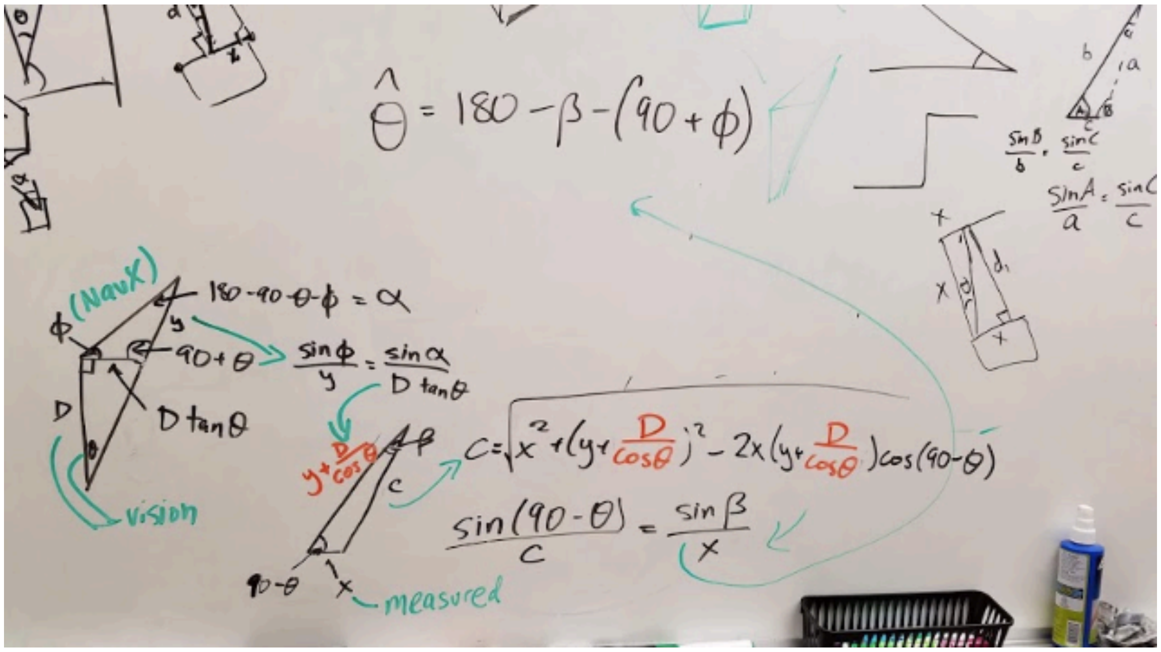
Φ is defined to be the angle between the front edge of the robot and the face of the target. If they are parallel, Φ = 0°, but if the target is to the left of the robot, Φ < 0 and if the target is to the right, Φ > 0.

Φ is calculated by reading in the field-centric angle of the robot from the NavX (which is reset at the start of every match) and subtracting it from the known angles of the various targets on the field. If the angle of the robot is within a threshold of the actual angle of the target, the code assumes the robot is trying to go towards that target and uses that angle.

For instance, if the threshold is 10°, the target is at 60° relative to the field, and the robot is anywhere from 50° to 70° relative to the field, it is assumed that the robot is aiming for that 60° target, and that value is used as the "actual" angle. Φ is determined by subtracting the actual angle from angle of the robot, giving us a "should turn" value which specifies how much the robot should turn to face the target straight on.

In the example above, if the robot were facing 55° relative to the field, the difference is within the threshold, and Φ is calculated to be +5°, meaning the robot needs to turn 5° to the right to face the target. If the robot were facing 67°, Φ would be calculated as -7°.

Horizontal Offset


$\hat{\theta} = 180 - \beta - (90 + \phi)$
(NavX)
$180 - 90 - \theta - \phi = \alpha$
$90 + \theta$
$\frac{\sin\phi}{y} = \frac{\sin\alpha}{D\tan\theta}$
D
$D\tan\theta$
vision
$y + \frac{D}{\cos\theta}$
$C = \sqrt{x^2 + (y + \frac{D}{\cos\theta})^2 - 2x(y + \frac{D}{\cos\theta})\cos(90-\theta)}$
$\frac{\sin(90-\theta)}{C} = \frac{\sin\beta}{x}$
$90 - \theta$
measured
$\frac{\sin B}{b} = \frac{\sin C}{c}$
$\frac{\sin A}{a} = \frac{\sin C}{c}$

This calculates the angle to a target that is not parallel to the front of the robot. Since the vision system returns a distance D and angle θ assuming the target is parallel to the front of the robot, some adjustments need to be made when the robot is not parallel to the target. Returns the adjusted angle θ-hat

A walkthrough of the calculation with example values:
Calculation starts on the left side and follows the green arrows to the subsequent step

Given: D = 30 inches, θ = 10°, Φ = 15°
α = 65°
y = 1.51064 inches
Given: x = 15 inches
c = 32.8746 inches
β = 26.7019°
θ-hat = 48.2981°

Depth Offset
This calculates the actual distance (depth) to the target. Since the vision system assumes the target is parallel to the front of the robot, adjustments

have to be made when it is known that the robot is not parallel to the target. Returns the adjusted depth real_depth

D is obtained in the same way as for the Horizontal Angle calculation, Φ is obtained using the method described above, and x is a known constant of the robot.

```
double phi = Robot.sensors.getAngleToTarget();
double real_depth = depth_offset + (cameraHorizontalOffset * Math.tan(Math.toRadians(phi)));
```

A walkthrough of the calculation with example values:

Given: D = 30 inches, Φ = 15°, x = 15 inches
real_depth = 30 + (15 * tan(15°)) = 34.0192 inches