***Mason's Guide to FRC Programming***

This document serves as a guide for all future programmers of FRC team 8046, the LakerBots (and really anyone else that wants to read it). It covers topics such as basic Java syntax and programming fundamentals, as well applied robotics programming and the components and principles of what really goes into the code of a competition robot. This guide also covers the specific programming paradigm of the LakerBots, and how we do things differently from other teams. With the combination of all these topics, you will too be able to be a master programmer.

**Table of Contents**

[Prologue: The Bookmarks 2](#_4fwxqanhi8z)

[Build Season Timelines 2](#_hdc67e4ubzhc)

[Starting a project 3](#_usj7utv4bol3)

[Java Programming Basics and Syntax 4](#_atft1gljwkj2)

[Classes 4](#_q8uf2ohr4d43)

[Objects 4](#_lkkteqvjp4cy)

[Dot Notation, The Power of the Dot! 6](#_gaa1faac3s7q)

[Data types 7](#_fqbcvpy1z5f2)

[Modifiers 9](#_tpyk15rhyfin)

[Additional Context on the Meaning of Static 10](#_121ui47abqpj)

[The LakerBots Secret Sauce 10](#_mbpv6y5pn88q)

[Subsystems 11](#_4jnbk2lbg96w)

[Commands 11](#_fmgflyljpbq4)

[SmartDashboard 15](#_5r979tec0krl)

[Vendor Dependencies 17](#_sc4d0fo0tvkn)

[Reading the Docs 18](#_a8em4zdl4lk8)

[Proportional Integral Differential Control (PID) 19](#_6ls4o8yihxaw)

[Control Theory 19](#_jwf7vy2kj4za)

[Implementation 21](#_kf9aatfu43ua)

[Example Tuning Graphs 26](#_52ddv7ppaby9)

[Swerve Drivetrain Code 27](#_poycbtdylf5g)

[Git Version Control 29](#_mnb19jx4l4zr)

[Appendix 32](#_vasomgpgfrv0)

[The Field Coordinate System 32](#_939n9z68wgrz)

[Debugging Cheat Sheet 33](#_eu4nzodk4rzc)

# Prologue: The Bookmarks

This guide contains a number of bookmarks associated with the Lakerbots 2025 codebase. They can be found in the format of “Bookmark-xx”. To navigate to the place in code which a bookmark refers to, use the global search tool in VScode by using Ctrl+Shift+F, and type the name of the bookmark verbatim.

# Build Season Timelines

The task of creating a codebase for an FRC robot is a big undertaking. There is a lot that needs to happen before you connect to your RoboRio for the first time. A programmer's job is to be ready, with testable and tunable code, when mechanical team passes the robot over to them. This doesn't mean the programmer should plug the computer in and magically make the robot come to life, but rather that they are writing the majority of their code after the robot has been constructed.

For the management of a project, a timeline can be used to mark specific goals for specific dates. Making a project timeline lays out what should be accomplished throughout the development process. Everyone will look different, but here is an example timeline:

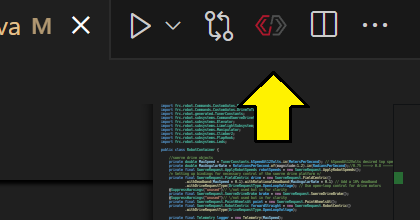
|  | week 1 | week 2 | week 3 | week 4 | week 5 | week 6 |
| --- | --- | --- | --- | --- | --- | --- |
| Code/Programming | began programming drivetrain | began subsystem code |  | communication system debugging | subsystem code and implementation | creating set positions for scoring |
|  | set up project infrastructure and architecture | simple commands | programmed new communication radios | started programming field positioning with limelight | position control loops |  |
|  | set up programming computer with required software |  | went to practice field to test field positioning | coded laserCAN for intake | apriltag positioning |  |

Having a task list is also useful

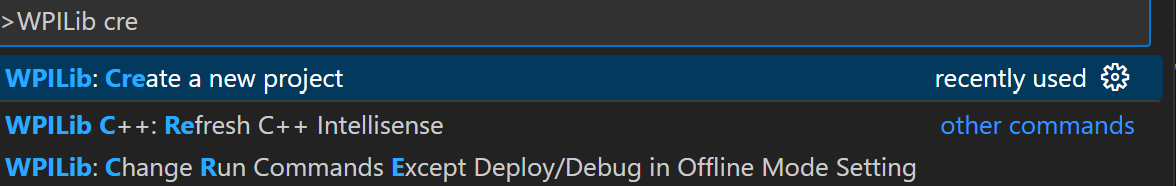


# Starting a project

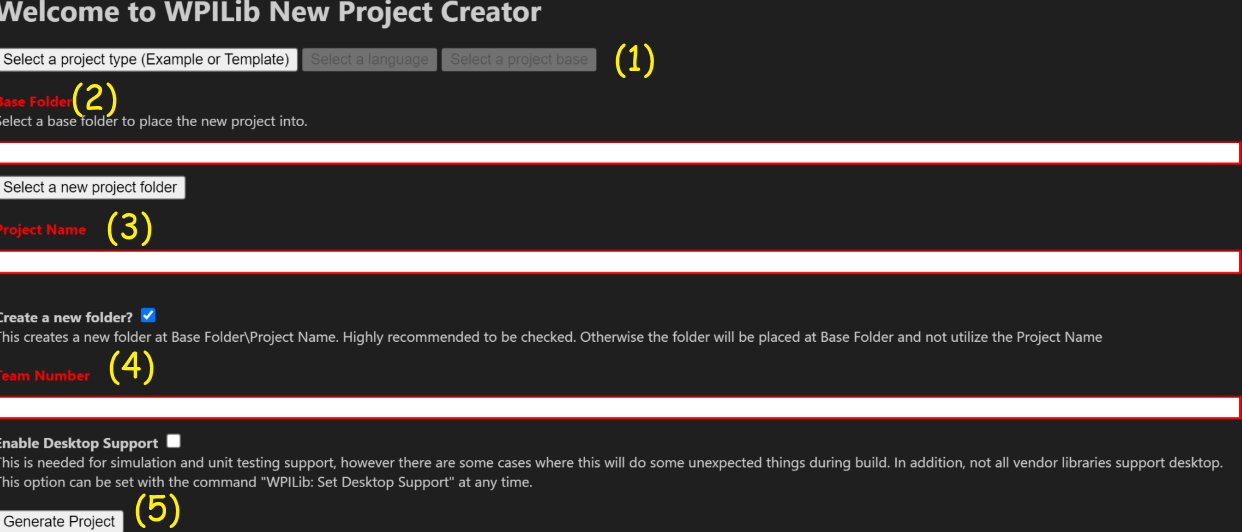
Before we get started with all the technical knowledge, let's generate a project that we can work and code in. Generating a robot project in VS code is pretty easy, simply press the WPI logo in the top right corner of VScode to open the command palette.



Click “create a new project”.



A GUI configuration screen will then appear.



(1) Select Template Project, then java, then Command Robot for the project type, language, and project base respectively.

(2)Select the Path you want to store the project in, in the LakerBots case this path is:

C:\Users\jderr\Documents\workspace\_code



Then make sure the Create a new folder box is checked, set a team number (4), and click generate (5), then you're good to go with a new robot project!

# Java Programming Basics and Syntax

Java is an object oriented programming language (OOP). Object oriented programming is a paradigm used by a variety of programming languages, such as C++, Ada, and most importantly Java. OOP can be described as an underlying structure in programming that uses the idea of objects and classes to support the programming techniques of Data Abstraction, Encapsulation, Inheritance, and Polymorphism. Not only is OOP useful for traditional programming, but it is especially helpful for Robotics oriented programming because of how the different mechanisms of our robot can be translated logically and seamlessly into objects, allowing them to utilize the techniques described above.

## Classes

To understand objects and OOP, the idea of classes must also be understood. Classes are used to represent a group of objects with similar properties. For example, a class named “cars” might contain the objects of “BMW e36”, “Honda Prelude”, and “Toyota Supra”. An example of classes in our codebase would be the “Manipulator” class, containing objects such as “manipulatorWrist” and “lazerCAN”, parts of the physical manipulator subsystem.

## Objects

Objects are used to represent real life things (mostly), bridging the gap between hardware and software. As classes are simply blueprints for objects, objects are simply instances of a class. Objects have 3 underlying properties, an Identity, a State, and a Behavior.

Identity: Name of the object

State: The objects attributes

Behavior: The methods of the object, how the object interacts with the code

The following example code demonstrates all three of these properties of objects, as well as the relationship between classes and objects

//class declaration

public class House{

//Instance Variables

int bedrooms;

int bathrooms;

String location;

//class constructer, aka the "object maker"

public House(int bedrooms, int bathrooms, String location){

//the "this." syntax signifies that we are setting the instance variables to have the data of the parameters (the varaibles contained within the parentheses), as they are the same name, we must use "this." to avoid having "bedrooms = bedrooms;" which would simply set the parameter to itself, effectively doing nothing"

this.bedrooms = bedrooms;

this.bathrooms = bathrooms;

this.location = location;

}

//Basic methods

public int NumberOfBedrooms(){

return bedrooms;

}

public int NumberofBathrooms(){

return bathrooms;

}

public String whereIsIt(){

return location;

}

//main method of the program, in FRC programming there is little interaction with the main method as it simply starts the robot code (Bookmark-1)

public static void main(String args[]){

//calling the class constructer to make objects

House myHouse = new House(3, 2, "Meredith, NH");

House mrDerricksHouse = new House(10, 3, "Alton, NH");

System.out.println(

"My house has " +

toString(myHouse.NumberOfBedrooms()) +

" bedrooms while Mr. Derricks has " + toString(mrDerricksHouse.NumberofBedrooms())

);

System.out.println(

"My house is in " +

myHouse.whereIsIt() +

" while Mr. Derricks is in " +

mrDerricksHouse.whereIsIt()

);

}

}

OUTPUT:  
"My house has 3 bedrooms while Mr. Derricks has 10"

"My house is in Meredith, NH while Mr. Derricks is in Alton, NH"



Identity is demonstrated above by the two objects having different names, state is demonstrated by the number of bathrooms and bedrooms and the location of the house, and properties are demonstrated by the class methods.

## Dot Notation, The Power of the Dot!

Dot notation is seen throughout all java code. It is one of if not the most used of the Java operators. It is used to access and reference the inner fields of a class member or object. Essentially a Elevator Object with a moveUp method is accessed by referring to moveUp as a part of the Elevator, by chaining the two together with a (.)

Elevator elevator = new Elevator();

elevator.moveUp(); //moves the elevator up



Dot Notation is hierarchical, you can chain dot notation if there are fields within fields.

//Referencing the position of the encoder of the elevator

int currentElevatorPosition = elevator.elevatorEncoder.positionReading();



## Data types

Data types are what they sound like, types of data. Data types describe variables by specifying the kind of values it can hold. Java is a statically typed language, meaning that you must tell the Java program what type of data you are working with (aka, Java does **not** know 7 is a number or “fish” is a word **unless you tell it**), this makes understanding Data types imperative to Java programming and coding in general.

There are two major types of data types, primitive and non-primitive. Essentially, primitive data types are the simple ones and non-primitive data types are more complicated.

In Java there are 8 different primitive data types.

byte: Whole numbers (-128 to 127)

short: Whole numbers (-32768 to 32767)

int: Whole numbers (-2147483648 to 2147483647)

long: Whole numbers (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

float: Rational numbers (6 to 7 digits out)

double: Rational numbers (15 to 16 digits out)

boolean: True or False value

char: Single letter or ASCII character

This might seem overwhelming at first, but the first four primitive data types are essentially the same, just allowing for different amounts of data. This is the same for the next two, float and double. **The typical LakerBots convention is to default to int for whole numbers and double for rational numbers.**

Non Primitive data types are data types that refer to objects. Thus they are often called reference types or object references. They must be user defined and constructed.

Primitive data type examples:

String: words or multiple ASCII characters

CommandXboxController: xbox controller

LimelightSubsystem: limelight camera

//creating a new class Color

public class Color{

Double[] colors = {0.0, 0.0, 0.0};//Array of doubles

int red;

int blue;

int green;

public Color(int red, int green, int blue){

this.red = red;

this.blue = blue;

this.green = green;

}

public Double[] getColorAsArray(){

colors[0] = red;

colors[1] = blue;

colors[2] = green;

return colors;

}

}

---Different files---

public class Person{

int age;

int height;

Color eyeColor;

public Person(int personsAge, int personsHeight, Color personsEyeColor){

age = personsAge;

height = personsHeight;

eyeColor = personsEyeColor;

}

}



These above classes can be referenced as objects and non primitive data types

Color masonsEyeColor = new Color(37, 145,125);

Person mason = new Person(18, 65, masonsEyeColor);



See Bookmark-2a and Bookmark-2b for an example of non primitive data construction

## Modifiers

Just Like Data types, we can divide modifiers into two groups, Access Modifiers and Non Access Modifiers.

Access Modifiers determine “visibility” or whether or not parts of the code outside of a given class can access given classes, attributes, methods, or constructors within said class. In general in FRC and LakerBots code, only the public and private modifiers are used.

Access Modifier Types:

public: class can be accessed outside itself

private: class can only be referenced **within itself**

There are also default and protected modifiers, although they are rarely used.

Non Access Modifiers provide other characteristics about a class, attribute, method, or constructor

final: attributes and methods that cannot be overridden or redefined

static: attributes that can be accessed **just by referring to a class**, without reference to an object

There are also abstract, transient, synchronized, and volatile modifiers, but they are again rarely used.

## Additional Context on the Meaning of Static

Accessing attributes or variables of a class without referring to an object has applications. Sometimes you don't always want to make an object in your class. The LakerBots use the static modifier for every variable in our Constants.java file, as we don't want to create a Constants object for every place in code we want to reference one of the constants. This example can be found in Bookmark-3a and Bookmark-3b. An additional example can be found in Bookmark-3c and Bookmark-3d, notice how we don't create a RobotContainer object instance in ComplexCommands.java, but we can still access and reference its subsystems statically.

# The LakerBots Secret Sauce

Over the past couple of years, we have developed our own kind of programming paradigm. A set of standards and practices that when implemented into our code, introduces structure, format, and functionality. These practices were found via a process of iterative development. Each year of competition code, we have taken what we liked about the previous year's code, and introduced additional techniques and modes of thinking. This is expected to continue to happen, and over the years this section of the guide will become less and less relevant.

## Subsystems

Each mechanism of the robot is separated into different subsystems. This architecture allows for logical reference to each of the robots mechanisms individually. You wouldn't want the code for your elevator to be in the same place as your launcher code, right? Each subsystem file contains a constructor of itself, so we can reference it in RobotContainer. The subsystem file is where the motor and sensor objects are created. Low level control methods and commands are also defined in the subsystem.

## Commands

In our code, all robot actions are encapsulated by commands. Commands give simple methods and code pieces another layer of abstraction and packaging. All commands are controlled by the Command Scheduler, a routine that runs automatically in WPILib code. The Command Scheduler dictates what commands are running at a given time and when to stop or interrupt them.

We have two different methods of making commands, constructing and composing

Constructed Commands are the traditional way of creating commands. To create a constructed command, it must have its own class. Constructed commands have four main body methods.

initialize: ran once at the beginning of the command

execute: runs repeatedly as the command is running

end: ran once at the end of the command

isFinished: returns boolean on end condition of command

Having separated body methods is versatile as it allows for higher level logic. Pieces of the command can be executed at different times and stages of the runtime schedule.

Example Constructed Command Code:



**public** **class** ScoreGamePiece **extends** Command {

**private** **final** LauncherSubsystem m\_launcherSubsystem;

**public** ScoreGamePiece(LauncherSubsystem subsystem) {

m\_launcherSubsystem = subsystem;

addRequirements(m\_launcherSubsystem);

}

@Override

**public** void initialize() {

m\_launcherSubsystem.warmUpLauncher();

}

@Override

public void execute(){

m\_launcherSubsystem.aim();

m\_launcherSubsystem.shootIfLinedUp();

}

@Overide

public void end(){

m\_launcherSubsystem.stopLauncher();

}

@Override

**public** boolean isFinished() {

**return** m\_launcherSubsystem.isGamePieceLaunched();

}

}



This command can now be called by being constructed.

//assumes launcher instance already created

driverXbox.x().onTrue(new ScoreGamePiece(launcher));//<-construction!



Composed Commands are a way of making commands inline, they do not require their own class to be created. There are many different types of composed commands, each with its own logical operation.

Composed Commands can be defined using the following lambda expressions.

runOnce: runs once and then finishes

run: runs every command scheduler iteration until interrupted

startEnd:runs one lambda once, and then another upon interruption

Composed commands are used mainly to interface with the command composition structure. Command composition is a way to apply additional logical decorators to composed commands.

andThen: decorates one command with another, runs the decorated command and then the decorator command

repeatedly: runs a command over and over until interrupted

alongWith: decorates one command with another, runs both in parallel, stops when all members finish

raceWith: decorates one command with another, runs both in parallel, stops when the first member finishes

deadlineWith: decorates one command with another, runs both in parallel, stops when a specific member finishes

until: runs a command until decorator condition is met or is interrupted

withTimeout: runs a command for a specific time, then interrupts

finallyDo: decorates one command with another, runs the decorated command after the end method of the decorated command

unless:runs a command unless a decorated condition is met

Here is an example of using the andThen decorator in an example robot commands class. Notice how each command is contained in the ComplexCommands class. This structure is used by the LakerBots. We define all the complex actions of the robot in their own class to abstract the logic from RobotContainer. As each “complex command” uses a number of our subsystems, we must reference them statically through RobotContainer to avoid creating a new instance of our subsystems. Referencing the subsystems in a static way avoids having to create an instance of RobotContainer in ComplexCommands.

import frc.robot.RobotContainer;

public class ComplexCommands{

//Static Reference from RobotContainer!

public static LauncherSubsystem launcher = RobotContainer.launcher;

public static IntakeSubsystem intake = RobotContainer.intake;

public static TurretSubsystem turret= RobotContainer.turret;

public ComplexCommands{}

//each command that references the static subsystems also must be static

public static Command collectGamePiece(){

//turning methods into commands using runOnce

return Commands.runOnce(() -> {turret.moveToIntake();})

.andThen(Commands.runOnce(() -> {intake.collect();}));

//andThen decoration!

}

public static Command spitOutGamePiece(){

return Commands.runOnce(() -> {intake.runBackwards();});

}

}



To avoid having to surround everything in runOnce lambdas, we can redefine our subsystem methods to return commands

//--in turret subsystem---

public Command moveToIntake(){

return Commands.run(/\*.....\*/);

}

//--in intake subsystem--

public Command collect(){

return Commands.run(/\*.....\*/);

}

//--in complex commands--

//we can then redefine collectGamePiece for better structure and readability

public static Command collectGamePiece(){

return turret.movetoIntake().andThen(intake.collect());

}



Additional Decorator Examples

//warms up the launcher and aims the turret at the same time, and when both of those commands end it launches the gamepiece

public Command scoreGamePiece(){

return launcher.warmUpLauncher()

.alongWith(turret.aim())

.finallyDo(launcher.launch());

}

//spins intake until sensor value true, then scooches the launcher

public Command indexGamePiece(){

return intake.collect().until(() -> intake.sensor.value() == true)

.andThen(launcher.scoochUp());

}

//alternate way of doing things in parallel

public Command prepareToClimb(){

return Commands.parallel(turret.home(), intake.moveIn());

}



## SmartDashboard

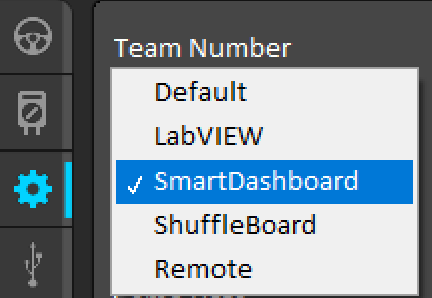
Controlling the robot can be accomplished in several ways. Certain control methods have different pros and cons. Autonomous control does not always work as intended *(wink wink)* and Xbox controllers only have so many buttons on them. Sometimes alternate ways of human input are required. This is where SmartDashboard comes into play.

We can interface through smart dashboard through defining some buttons

SmartDashboard.putData("process algea", ComplexCommands.goToProcessorPose());



Then just set the Dashboard type to SmartDashboard in the driver station



and the button should populate in the SmartDashboard program upon code deployment and the enablement of the robot



This button can then be used to control the robot!

In addition to human input, SmartDashboard can also be used to collect and display robot output. Using the same syntax, we can define an output box

//elevator periodic

//see Bookmark-4

SmartDashboard.putNumber("elevator postion" elevatorLead.getPosition().getAsDouble());

SmartDashboard.putBoolean("elevatorAtPosition", elevatorAtPosition());



these values populate and update as the periodic method of elevator executes

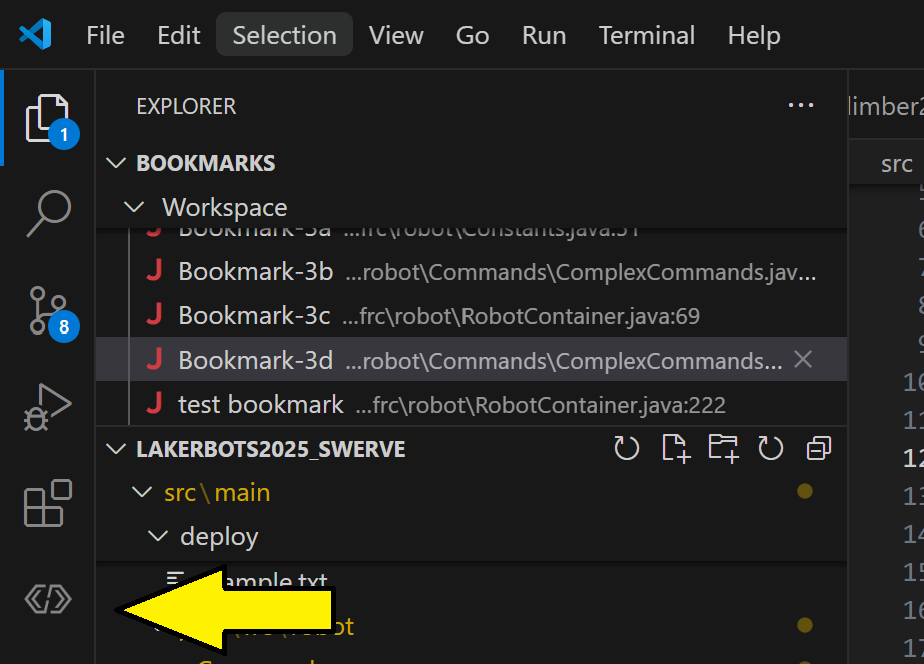


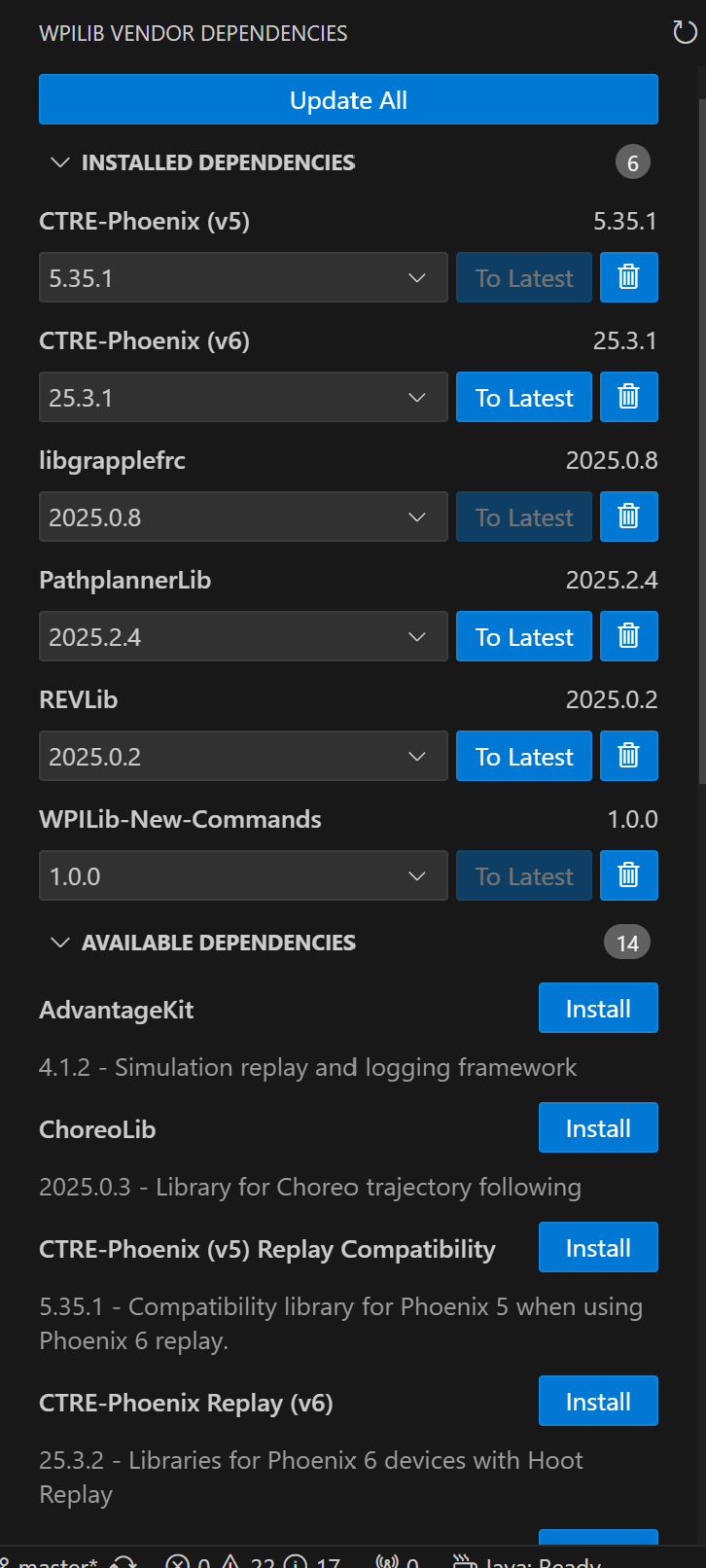
# 

# Vendor Dependencies

Vendor Dependencies are code libraries supplied by the manufacturers of electronics used in FRC. They assist in controlling said electronics by having pre-programmed classes, constructors, and methods that are essentially “plug and play” with FRC Java. Such code helps us interface with the various motors and sensors on the robot without having to create code for them from the ground up.

Vendor Dependencies can be installed using the tab in the bottom left of VScode





A screen will then appear with any currently installed dependencies and what version they are, as well as options to update them to the latest version if they are not already updated. At the bottom will be the other available dependencies that can be installed, simply select the ones that are required for your project and it will install them automatically.

# 

# Proportional Integral Differential Control (PID)

PID control is a method of controlling motors in a precise and algorithmic manner using control coefficients. PID control operates in a closed loop feedback system. A closed loop system takes continuous input from a source, and provides constant feedback with respect to the value of the input or process variable, the current time, the elapsed behavior of the process variable, and the target value or setpoint/reference variable.

## Control Theory

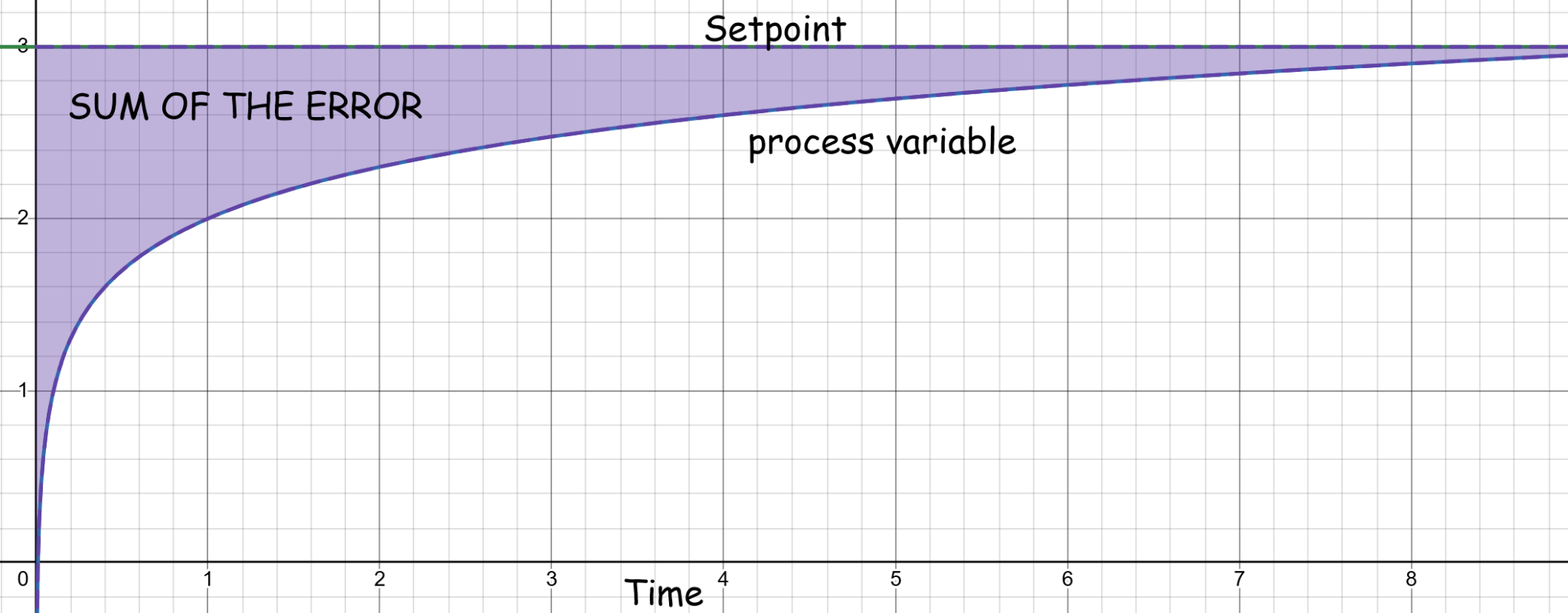
In practice, PID theory is applied to control motors. Position control of a mechanical system is achieved by a motor running on a PID controller. For a given setpoint position, the controller is notified of its position target once, and constantly updated with the current position. The output of the controller is used to control the motor and move the mechanism to the given position, which then feeds back position values to the controller, creating a closed loop. This can also be applied in terms of a motor's velocity instead of position. Disturbance in the system or external forces acting on the mechanism can be corrected for using this method. A “well tuned” PID loop minimizes the effect of disturbances on the process variable.

There are three parameter coefficients that control the PID controller, P, I, and D. Consider the controller as a function of elapsed time,

P is the proportional gain. It takes the product of the magnitude of the error signal (the difference between the current value of the process variable and the reference variable) and the P coefficient to produce response proportional to the error of the system. In short; the more error, the more response to try to correct it, and the P value is how much it does this. If a system has a magnitude of error of 10 and has a proportional gain of 2, it will produce a proportional response of 20. A controller using only proportional gain is defined as:

where is a function of the error signal with respect to time

I is the integral response. It will sum the error in the system over time. This is useful when steady state error is encountered because the integral response will still be increasing upon asymptoting below or above the setpoint.



Considering our controller function, adding integral response looks like this:

where serves as an integration variable, as the time is the upper bound of the integral and we cannot integrate with respect to it.

D is the derivative response. It essentially acts as a damping factor. It is proportional to the derivative of the process variable with respect to time. The signal generated by the derivative component combined with the controller response is equal to:

This is the complete definition of a simple PID controller. Don't let all this math scare you! It's simply to help visualize how the signals of a PID system are generated and what each component does.

## Implementation

Implementation of PID control in an FRC setting is fairly straightforward, it's the tuning that's the hard part. Depending on what family of motor controllers are used for PID control, the syntax may look slightly different.

In the REV family of motors, we can use the MAXMotion framework from the vendor. MAXMotion allows for additional control over a motors velocity and acceleration, it is a second degree PID controller. To utilize MAXMotion, the following syntax is used:

//This is our Manipulator Subsystem trucated for the relevent code

//See Bookmark-5

public class Manipulator extends SubsystemBase {

public SparkMax manipulatorWrist;

public SparkClosedLoopController positionController;

public SparkMaxConfig wristConfig;

public targetPos;

public Manipulator(){

manipulatorWrist = new SparkMax(3, MotorType.kBrushless);

positionController = manipulatorWrist.getClosedLoopController();

wristConfig = new SparkMaxConfig();

//configure first degree PID values

wristConfig.closedLoop

.feedbackSensor(FeedbackSensor.kPrimaryEncoder)

.p(0.4)

.i(0.001)

.d(0)

.outputRange(-1,1)

.iZone(0.5);

//configure second degree PID values

wristConfig.closedLoop.maxMotion

.maxVelocity(1500)

.maxAcceleration(1500)

.allowedClosedLoopError(.1);

manipulatorWrist.configure(

wristConfig,

ResetMode.kResetSafeParameters,

PersistMode.kNoPersistParameters);

}

//position control method

public Command manipulatorWristGoToPosition(double targetPos){

return runOnce(

() -> {

this.targetPos = targetPos;

positionController.setReference(

targetPos,

ControlType.kMAXMotionPositionControl

);

});

}

}



In the CTRE family of motors, we can use the MotionMagic framework. MotionMagic is very similar to MAXMotion, being a second degree PID controller.

MotionMagic example:

//Elevator Subsystem Truncated for relevent code

//See Bookmark-6

public class Elevator extends SubsystemBase {

public TalonFX elevatorLead = new TalonFX(13, "canivore");

private final MotionMagicVoltage m\_mmReq = new MotionMagicVoltage(0);

public TalonFxConfiguration cfg = new TalonFXConfiguration();

public double setpoint;

public Elevator(){

//Second degree PID values

MotionMagic mm = cfg.MotionMagic;

mm

.withMotionMagicCruiseVelocity(RotationsPerSecond.of(15))

.withMotionMagicAcceleration(RotationsPerSecondPerSecond.of(15))

.withMotionMagicJerk(RotationsPerSecondPerSecond.per(Second).of(0))

//First degree PID values

Slot0Configs slot0 = cfgf.Slot0;

slot0.kS = 0;

slot0.kV = 0.275;

slot0.kA = 0.0;

slot0.kP = 1.0;

slot0.kI = 0.0;

slot0.kD = 0.0;

slot0.kG = 0.5;

}

//position control method

public Command elevatorGoToPosition(double setpoint){

return Commands.runOnce(() -> {

this.setpoint = setpoint;

elevatorLead.setControl(m\_mmReq.withPosition(setpoint));

});

}

}



MotionMagic also allows for on the fly switching of second degree profiles with the purchase of a Phoenix Pro license. Being able to switch between second degree PID profiles is useful when encountering a mechanism such as a linear elevator. In the 2025 codebase, there are implementations of both dynamic and non-dynamic MotionMagic present. The ladder being our first implementation of position control on the mechanism, which we found to work, but turning up the speeds for efficiency was a dangerous game, as the elevator had different forces acting on it depending on the direction of travel (it was fighting gravity on the way up, and being assisted by it on the way down). So we used two different profiles, one on the way up, and one on the way down.

Dynamic Motion Magic Example:

//Elevator Subsystem Truncated for relevent code

//See Bookmark-6

public class Elevator extends SubsystemBase {

public TalonFX elevatorLead = new TalonFX(13, "canivore");

private final DynamicMotionMagicVoltage dynamicReq =

// position, velocity, accel, jerk

new DynamicMotionMagicVoltage(0, 80, 400, 4000);

public double setpoint;

public Elevator(){

//Second degree PID values

MotionMagic mm = cfg.MotionMagic;

mm

.withMotionMagicCruiseVelocity(RotationsPerSecond.of(15))

.withMotionMagicAcceleration(RotationsPerSecondPerSecond.of(15))

.withMotionMagicJerk(RotationsPerSecondPerSecond.per(Second).of(0))

//First degree PID values

Slot0Configs slot0 = cfgf.Slot0;

slot0.kS = 0;

slot0.kV = 0.275;

slot0.kA = 0.0;

slot0.kP = 1.0;

slot0.kI = 0.0;

slot0.kD = 0.0;

slot0.kG = 0.5;

}

//Simple void methods not commands!

public void elevatorUpDynamic(double setpoint){

this.setpoint = setpoint;

dynamicReq.Velocity = 45;

dynamicReq.Acceleration = 50;

dynamicReq.Jerk = 0;

elevatorLead.setControl(dynamicReq.withPosition(setpoint));

}

public void elevatorDownDynamic(double setpoint){

this.setpoint = setpoint;

dynamicReq.Velocity = 26;

dynamicReq.Acceleration = 26;

dynamicReq.Jerk = 0;

elevatorLead.setControl(dynamicReq.withPosition(setpoint));

}

}



//ElevatorMoveDynamic Command

//See Bookmark-7

public class ElevatorMoveDynamic extends Command {

public Elevator elevator;

public double setpoint;

public ElevatorMoveDynamic(Elevator elevator, double setpoint) {

this.elevator = elevator;

this.setpoint = setpoint;

addRequirements(elevator);

}

@Override

public void initialize() {

//If the diff between the setpoint and the pos is <0, move with the up profile

//(as elevator pos starts at 0 and goes nrgative as it moves up)

//else use the down profile

if(setpoint - elevator.elevatorLead.getPosition().getValueAsDouble() < 0){

elevator.elevatorUpDynamic(setpoint);

}else{

elevator.elevatorDownDynamic(setpoint);

}

}

@Override

public void execute() {

}

@Override

public void end(boolean interrupted) {

}

@Override

public boolean isFinished() {

return elevator.elevatorAtPosition();

}

}



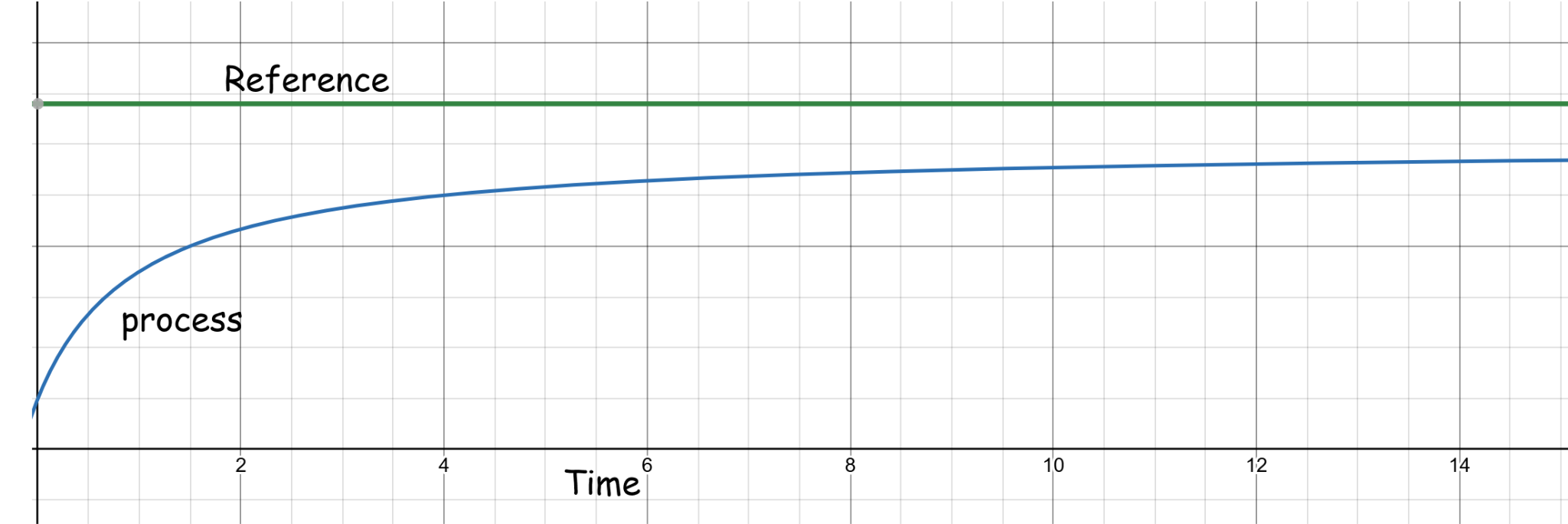
Simple PID controllers can also be generated without the use of vendor dependencies:

PIDController pid = **new** PIDController(kP, kI, kD);

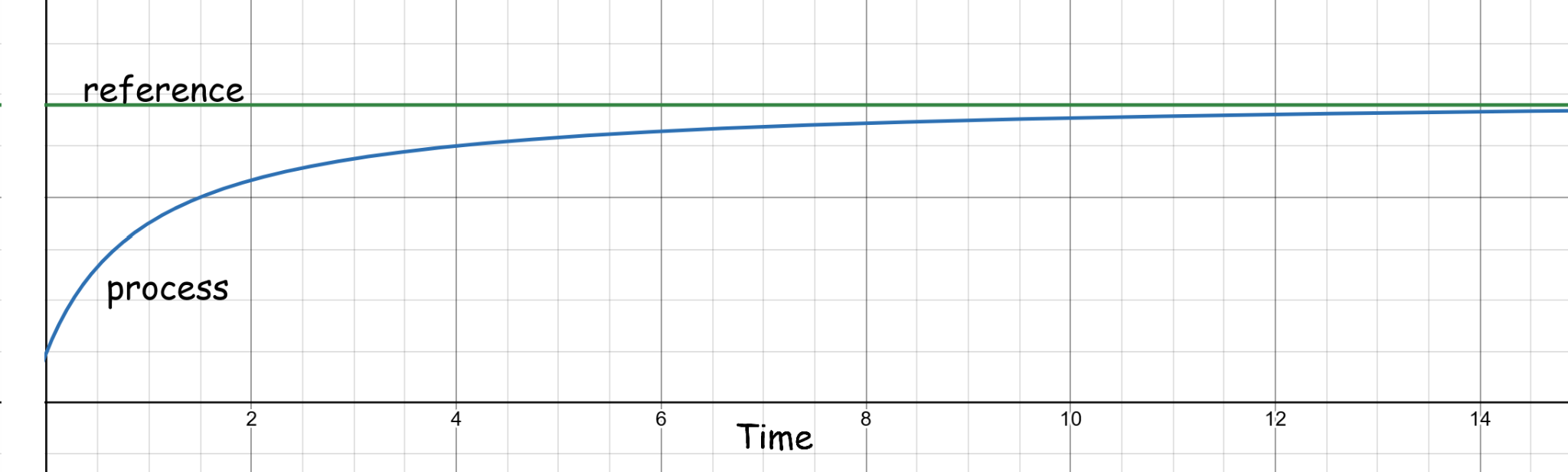


## Example Tuning Graphs

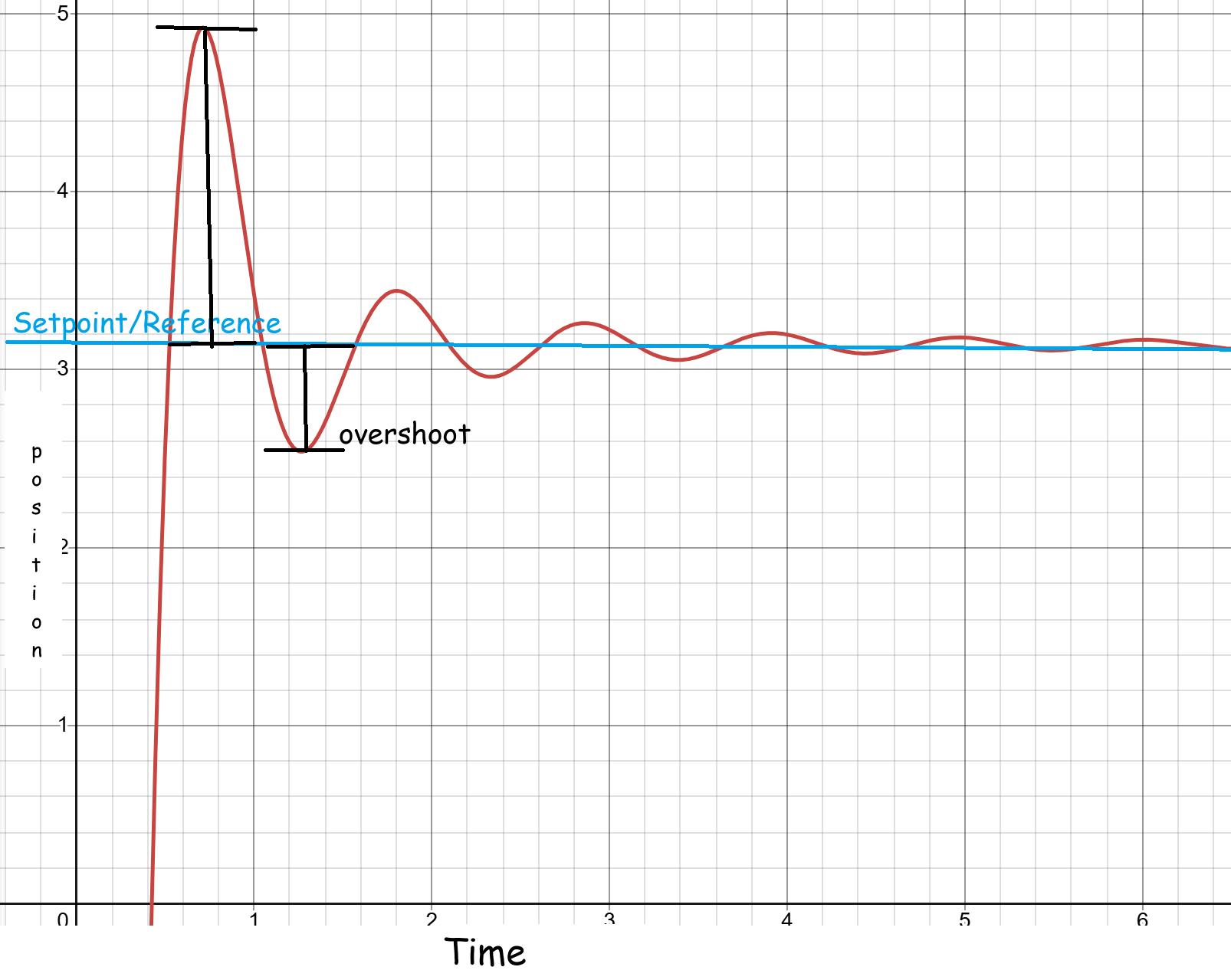
Undertuned PID system, increase P or I



Well Tuned System



Over Tuned System, decrease P or increase D.

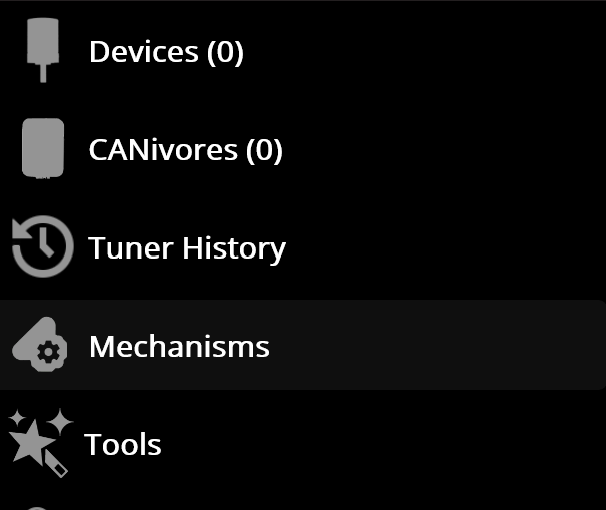


# Swerve Drivetrain Code

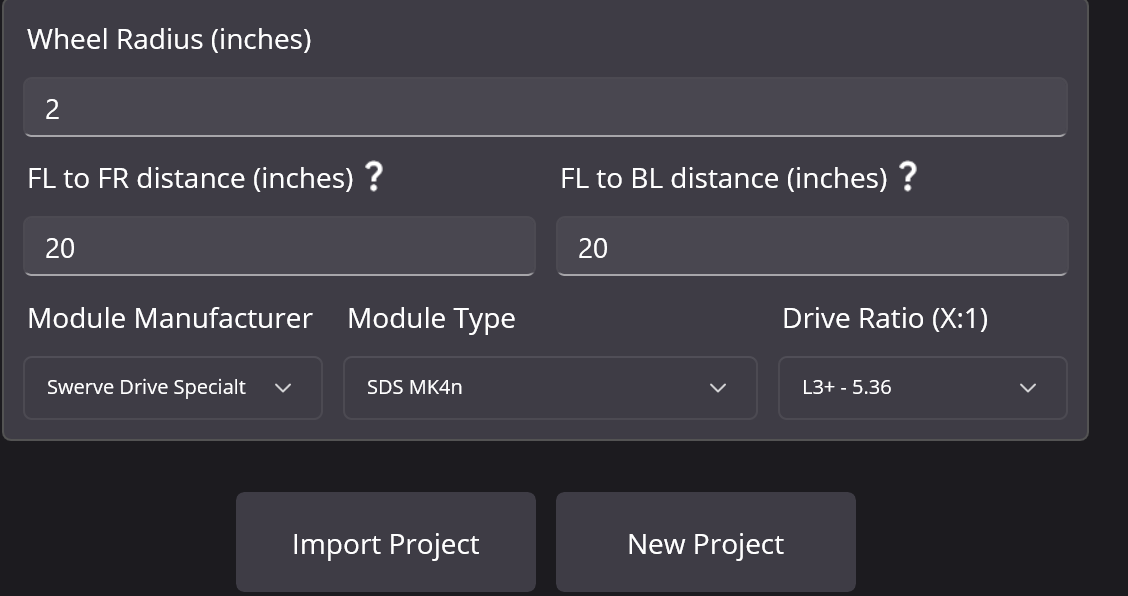
Swerve Drive is a mechanical drivetrain system where there are four wheels that have independent control over their direction and velocity. There are two motors per wheel for a total of eight, one controlling the heading and one spinning the wheel. This mechanical system allows for robots to move in all directions.

In the **dark ages**, programmers had to write their own swerve code or copy from an example project. Now we can use a tool called Tuner X to generate our swerve code, assuming only Phoenix products are being used on the drivetrain, and that the drivetrain is running on a CANivore.

Using the Tuner X Swerve Generator is pretty simple. To start, simply click into the Mechanisms tab

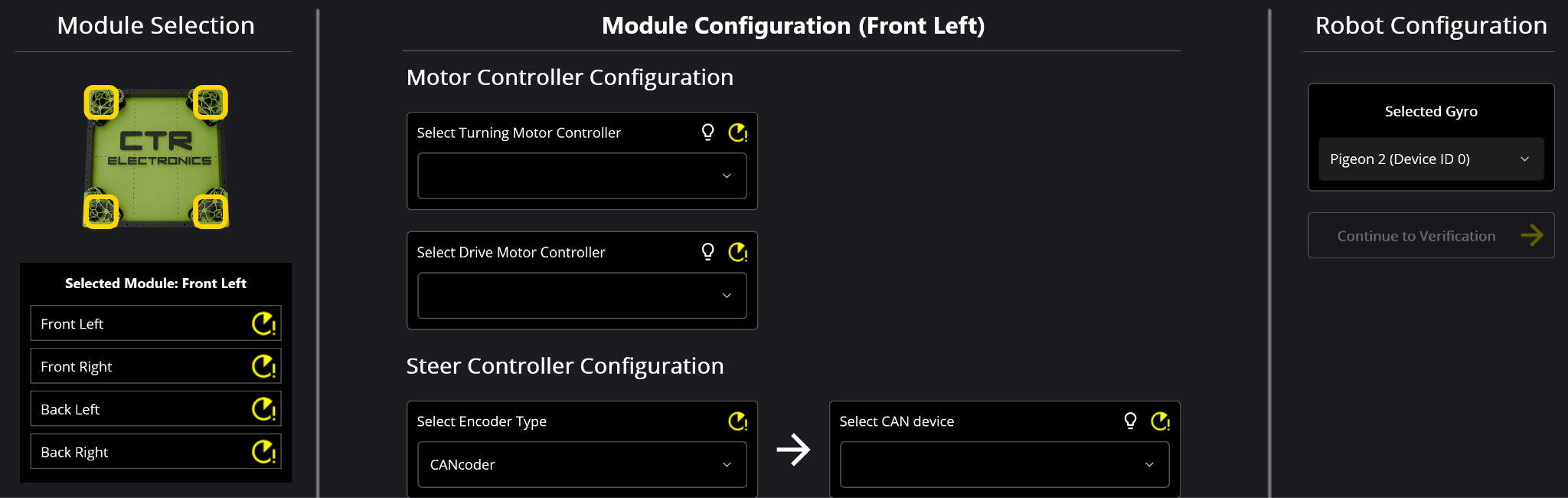


Input the characteristics of your drivetrain into the boxes



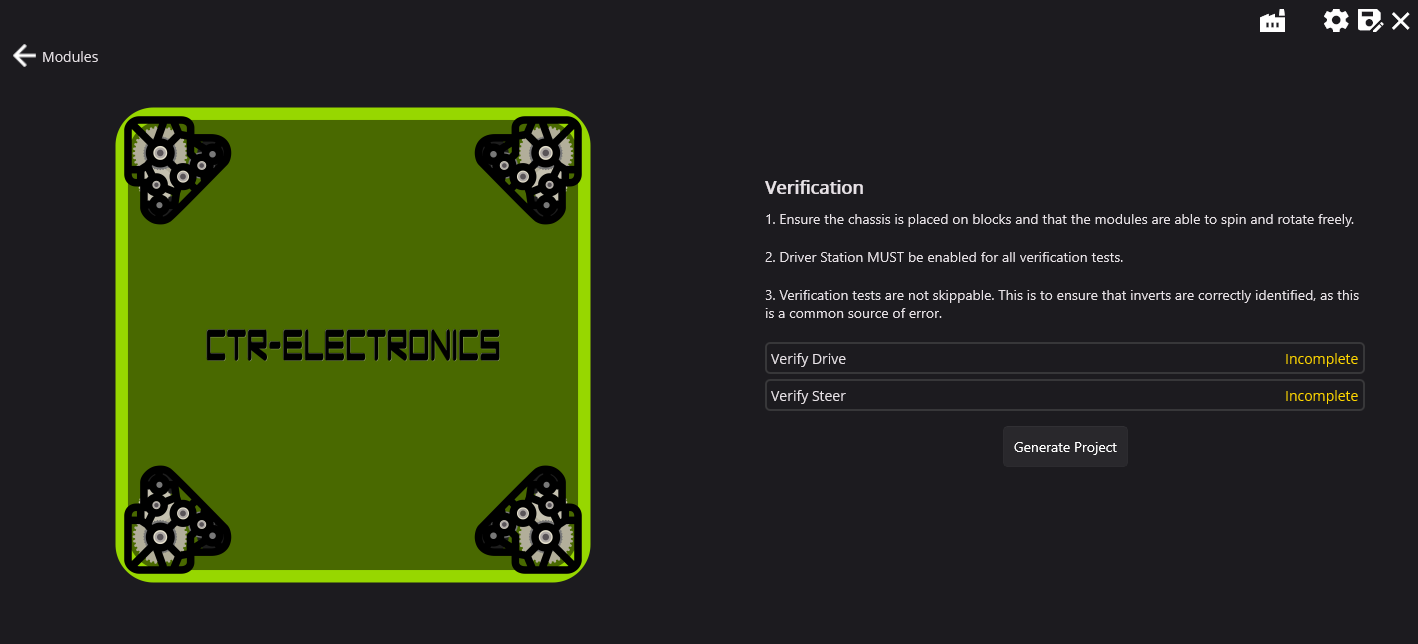
and then either import your current robot project, or have it generate a new one for you.

Then configure each module with the CAN of the associated motors and encoder.



and run the drive/athmuith tests to verify each module was configured correctly.

Then the overall drive and angle tests may be run, if these succeed the drivetrain is configured correctly



Then click generate and it will auto populate your robot codebase with swerve code!

# 

# Git Version Control

Git is a distributed version control system. It is used for controlling source code and collaborative programming. To interface with Git, a developer platform is used. The industry standard platform is called Github.

To understand how Git works, we must define some terms:

Repository: a directory that contains all the files of a project, all versions of said files, and all commits, deletions, and other file history

Branch: a version of a repository that diverges from the main project

Master/Origin: the main branch of a repository

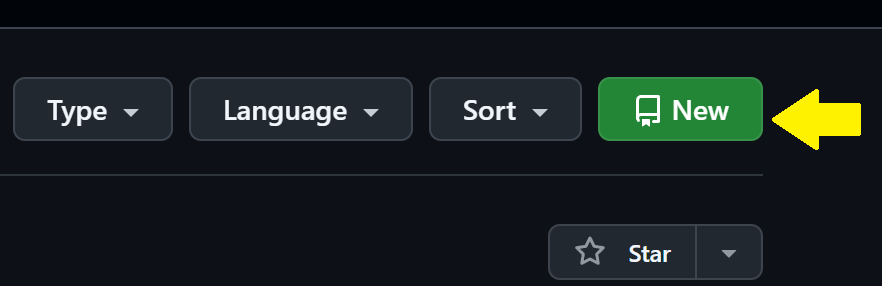
Remote: the common repository, this is what is actually displayed and stored in github

Commit: a “snapshot” of your repository at a current time, tracks all files in a repo and changes to files, also can be thought of as the “steps” in your code development

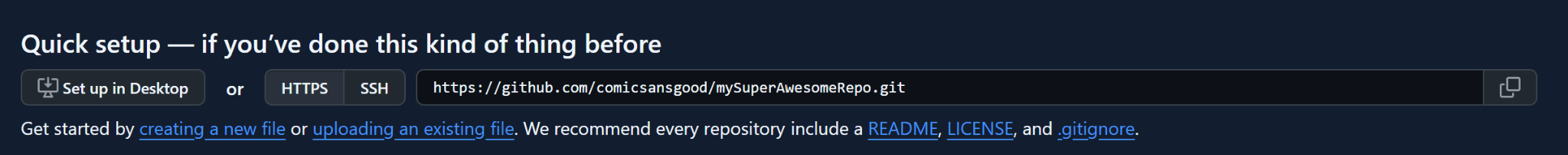
Push: command that updates the remote repository with your current commit, figuratively “pushing” your changes onto the remote

Pull: command that updates your local repository with changes made to the remote

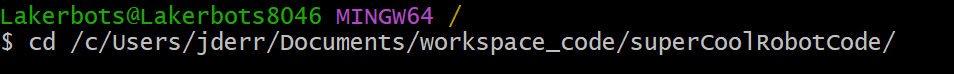
To set up a git repository, first create a remote in github



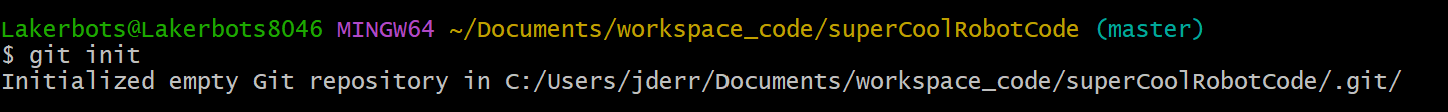
Then copy the generated link, we will need this later



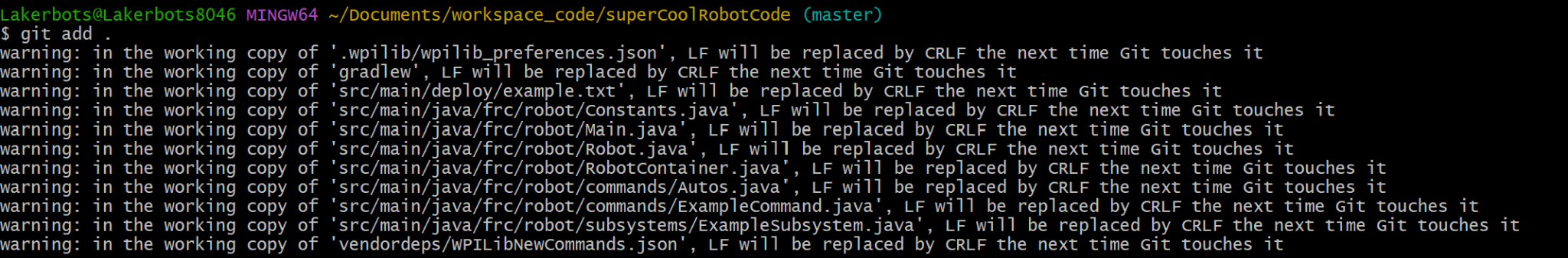
After your remote repository is set up, we need to make a local repo so we can push our code to github. Open Git Bash and change the active directory to that of your project using the cd (short for change directory) command.



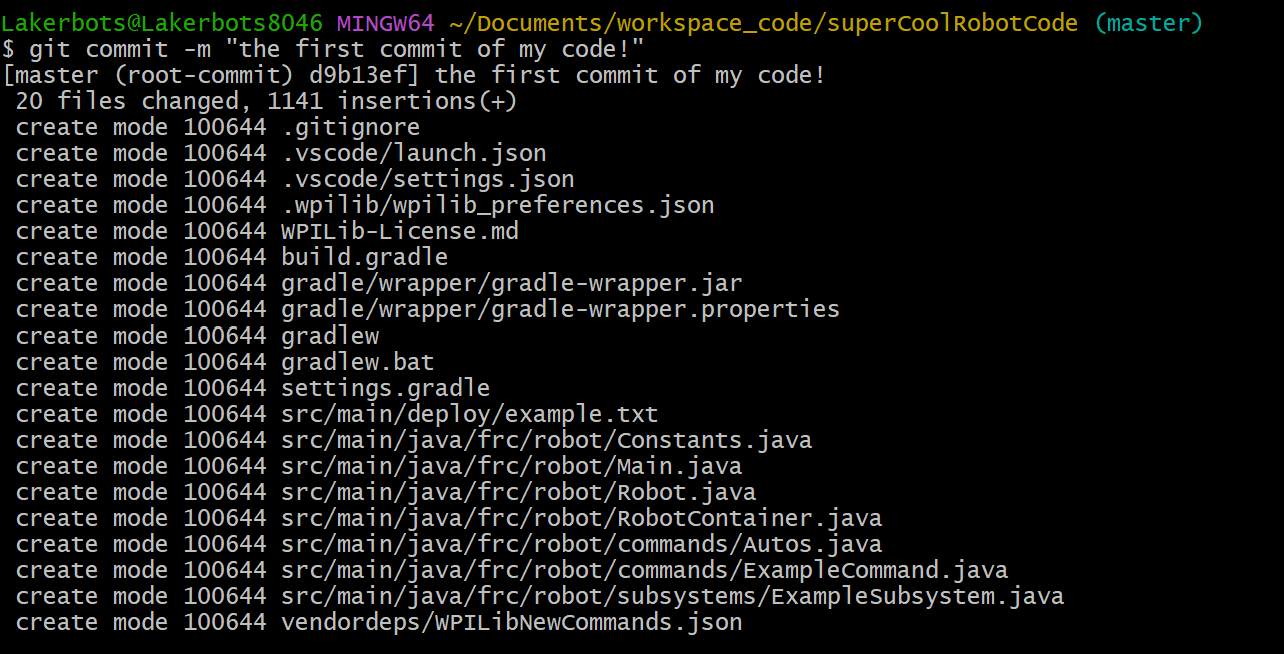
To create a repo of the local project, use the git init command



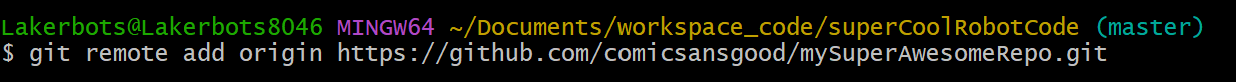
Then by running git status, we can see that all the files are untracked by git, which makes sense because we have not added any files to the remote repo. We can use git add to add the files to a commit.



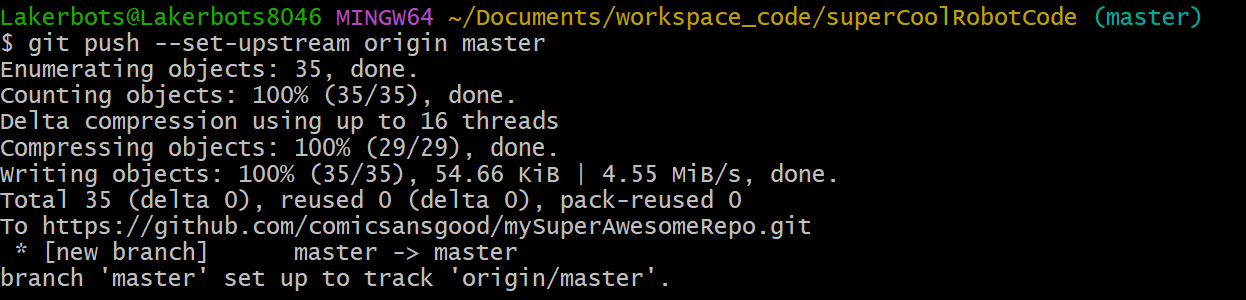
Then a commit message is needed, this message will be displayed in github and should explain what the pushed changes are.



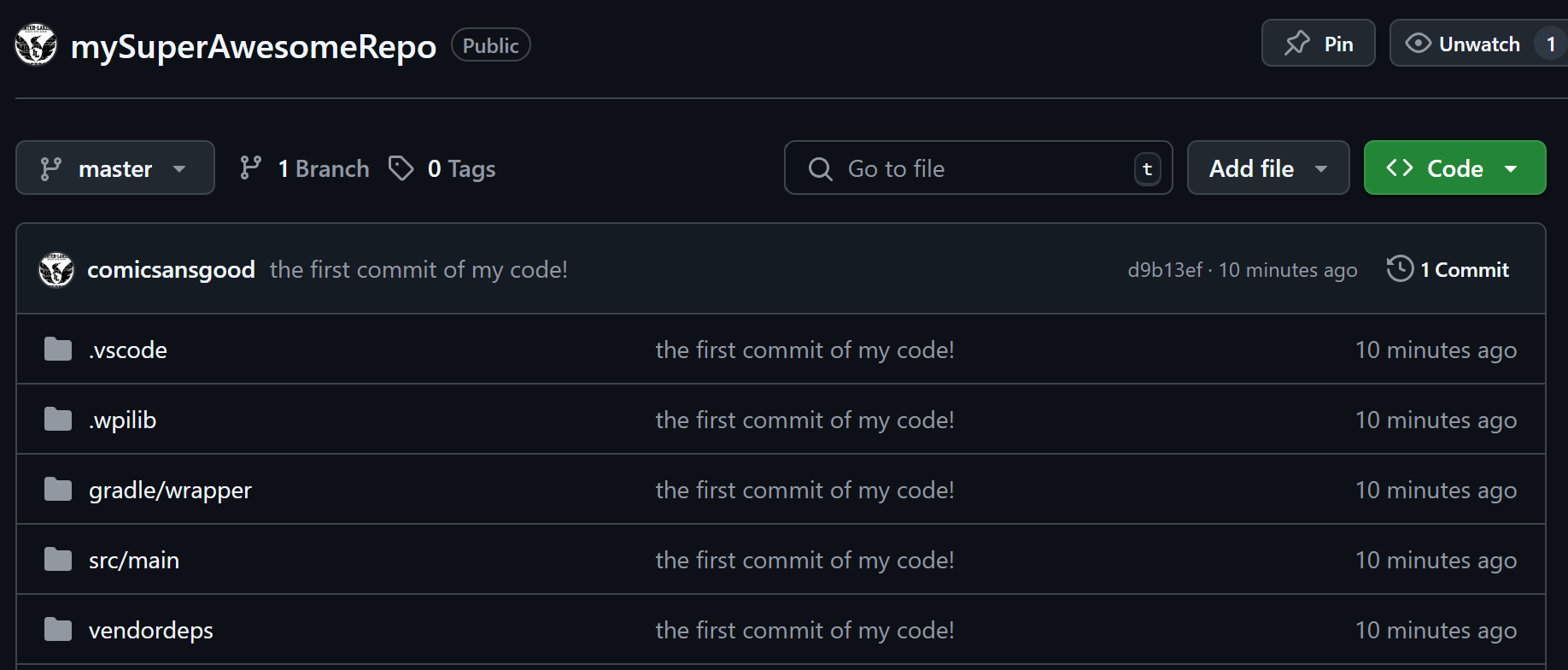
Before we push our changes, we must create a link between the local and remote repo’s. This is where that url comes into play!



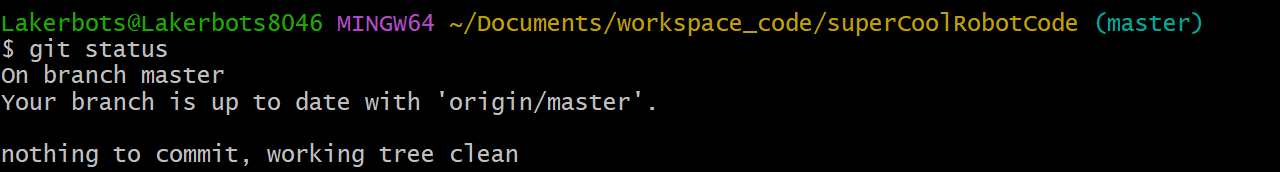
Finally, we can push our changes to the remote. Before the first push, the master(local) branch must have a set upstream branch(origin). The first push of your project should set this using the set-upstream parameter as you are pushing. It might prompt you to enter your github username and password at this step



Then that's it! Navigating back to github we can see our project is now in our remote repo



We can run git status once more to see that our local and remote are the same



To push additional changes, the following commands are executed

cd {path of project}

git status

git add .

git commit -m "{commit messege}"

git push

git status



For additional info on git and how it works, check out this video![Git Visually Explained](https://www.youtube.com/watch?v=-iWaarLI7zI)

# Reading the Docs

Reading Documentation is one of the most valuable tools in the programmers arsenal. They can tell you a lot if you know where to look.

And guess what, you're already doing it! This guide is documentation!

Vendors supply documentation for their vendor dependencies, and WPI has their own documentation for WPILib. Basically, if there is non user supplied code, there are probably docs for it somewhere.

Common Documentation:

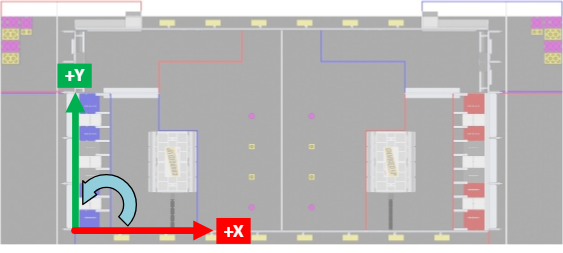
* WPI:<https://docs.wpilib.org/en/stable/index.html>
* CTRE:<https://v6.docs.ctr-electronics.com/en/latest/index.html>
  + Tuner X: <https://v6.docs.ctr-electronics.com/en/latest/docs/tuner/index.html>
  + CTRE Package Overview: <https://api.ctr-electronics.com/phoenix6/release/java/>
  + Phoenix 6 Examples: <https://github.com/CrossTheRoadElec/Phoenix6-Examples>
* Rev:<https://docs.revrobotics.com/revlib>
  + Rev Package Overview: <https://codedocs.revrobotics.com/java/com/revrobotics/package-summary.html>
  + SparkMax Examples: <https://github.com/REVrobotics/SPARK-MAX-Examples/tree/master/Java>
* Grapple Robotics (LaserCAN): <https://github.com/GrappleRobotics/LaserCAN/blob/master/docs/getting-started.md>
  + Java Example: <https://github.com/GrappleRobotics/LaserCAN/blob/master/docs/example-java.md>
* Path Planner: <https://pathplanner.dev/home.html>
* Limelight: <https://docs.limelightvision.io/docs/docs-limelight/getting-started/summary>

# 

# 

# Appendix

## The Field Coordinate System

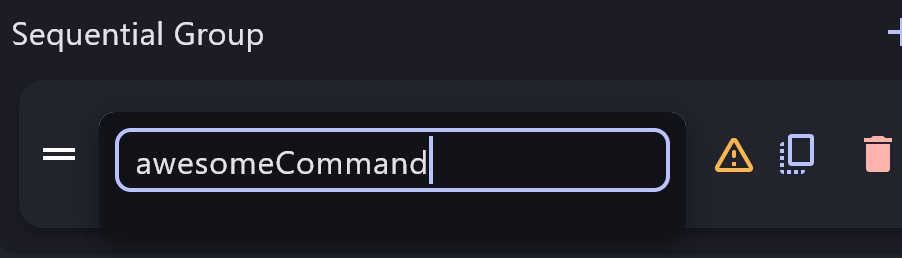
A robot's odometry routine constantly estimates the position of the robot relative to the field. This relative position is based on a specific field coordinate system. This coordinate system is used for swerve drivetrain code as well as autonomous routines. The standard convention in FRC is for (0,0) to be in the bottom left corner of the blue driver stations, closest to the scoring table. The convention is also for positive-x to be moving towards the red driver stations, parallel to the scoring table, and for positive-y to be moving perpendicularly away from the scoring table.

## Debugging Cheat Sheet

| Error Message | Fix |
| --- | --- |
| { } cannot be resolved to a type | Import the type using quickfix |
| Cannot make a static reference to the non static field { } | Either construct a new instance of the object locally(DO NOT do with subsystem objects) or reference statically using the static keyword |
| The method { } in the type { } is not applicable for the arguments { } | Method is being called with incorrect parameters, check the parameter types of the method |
| FAILURE: Build failed with an exception.  What went wrong: Execution failed for task ‘:downloadDepsPreemptively’. | Update your vendor dependencies |
| Unhandled exception instantiating robot { } java.lang.NullPointerException: Cannot invoke "{ }" because "{ }" is null | Object is referenced without being having a state, set the object equal to a value before referencing its identity |

## Pathplanner Tips

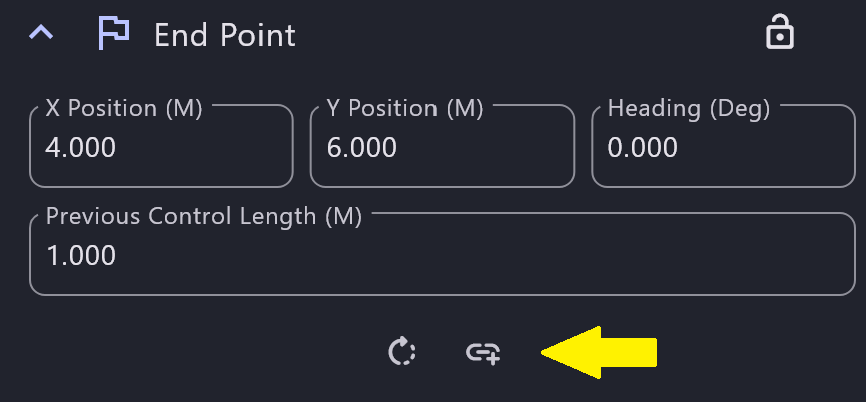
When adding a named command to your auto in pathplanner make sure you type the exact name of the command into pathplanner, it will not pop up in a drop down until you type in a textbox

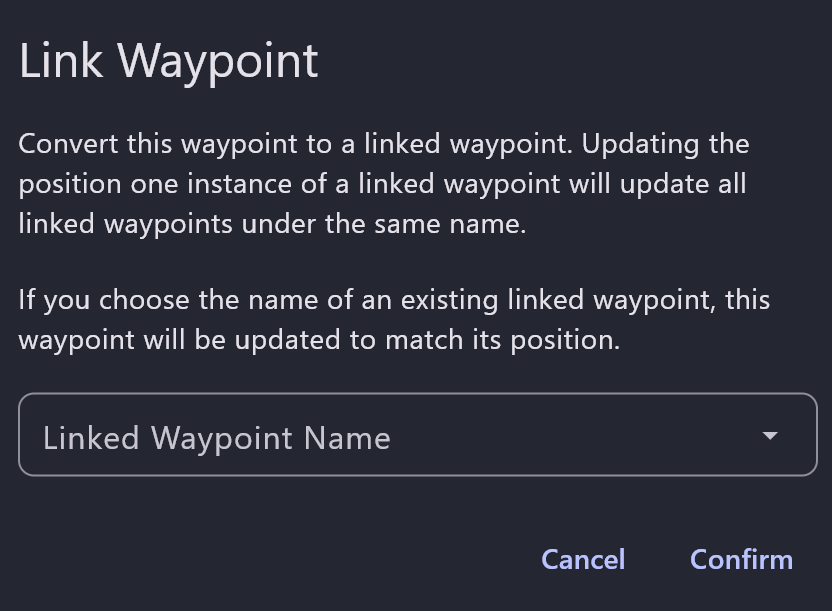


NamedCommands.registerCommand("awesomeCommand",awesomeSubsystem.awesomeCommand());



Linked waypoints in pathplanner are really useful. They link the endpoint of one path to the start of another so if you change one, it changes the other automatically





Same thing as the named command, you have to type the name in once before it appears in the dropdown