FRC Programming Guide

By: Rohan Patel

Preface:

This guide aims to make FRC programming easier to comprehend and apply. It serves as a supplement of official documentation, such as WPILib and CTRE docs, by providing simple, beginner-friendly explanations.

When I first began learning FRC programming on Team 5199, I faced challenges with understanding and searching through documentation. This inspired me to compile all major concepts into one place, making them easily accessible for anyone.

Thank you to my mentors, seniors, fellow programmers, and parents for continuously guiding, teaching, and inspiring me throughout my journey.

Table of Contents

Introduction

Tools

Installation of WPILib: Go to the WPILib Installation Guide and follow the steps listed there. This installs VS Code with the WPILib extension, Shuffleboard, different roboRIO tools, and much more.

Installation of NI Game Tools: Go to the FRC Game Tools Download and download the latest version after logging in or creating a new account. This installs Driver Station.

Balena Etcher: Balena Etcher is not part of the tools included in the WPILib installation or NI Game Tools and will have to be installed from BalenaEtcher's Website. It allows firmware to be flashed onto SD cards. This can be used for the roboRIO or for devices running PhotonVision.

Phoenix Tuner: Phoenix Tuner has to be installed separately and can be found on the Microsoft Store. It allows IDs to be set for CTRE products and configure them, although configurations set in Phoenix Tuner are not permanent and are overwritten by code. It also allows different control requests to be applied to motors, allowing control of motors without deploying any code. Lastly, if using a drivebase made with CTRE motors, it allows generation of drive code by following these steps.

RoboRIO Team Number Setter: Make sure that the roboRIO's SD card already has the latest firmware by either using Balena Etcher or the RoboRIO Imaging Tool. After, plug into the roboRIO using an ethernet cable, type in the desired team number, and click set.

Shuffleboard: Shuffleboard is used for logging all types of values such as motor positions, velocities, etc. It registers even while the robot is disabled so it can be used while debugging or calibration during events. Many teams also use it to select their autonomous routine as it has drop down menus as well.

VS Code: VS Code (with the WPILib extension) is the main code IDE used. To create a project, click on the W icon in the top right corner, and then select Create a new project. Click Select a Project Type, then Template, Java, and Command Robot or Timed Robot, depending on if the project is command-based or timed-based. After selecting the folder, naming the project, and setting the team number, generate the project. This will set up the Robot and Main classes with some pre-generated methods, along with RobotContainer if in command-based.

DriverStation

Operation Tab: On the very left, it allows selection of what mode the robot will enable in. TeleOperated is the standard mode, Autonomous will run the robot's autonomous command if it has one, Practice will run TeleOperated and Autonomous modes with specific times set in the Settings tab and then automatically disable once time is finished, and Test will run a different set of initialization and periodic methods defined in the code. To enable the robot, click Enable or press "[" + "]" + "\". To disable the robot, click Disable or press Enter. Pressing space while the robot is enabled will cause Emergency Stop the robot, requiring a power cycle of the robot as well as restarting the driver station. Communications displays if the computer is connected to the robot, Robot Code displays if there is functional code on the robot, and Joysticks displays if there is a controller connected to the computer.

Diagnostics Tab: Here, it displays the Connection info. Communications can be restarted here, as well as restart the code or reboot the roboRIO.

Settings Tab: Here, set the team number that is registered on the robot's roboRIO. If the team number of the DriverStation is different from the team number set on the roboRIO, the DriverStation will not connect. Timings for each section of the Practice Model can also be set here.

USB Devices Tab: Here, all the different controllers connected to the computer are displayed and what port they are in. Make sure that the port of each controller matches the port set in code.

Overview of Starter Code

Command-Based vs. Timed-Based

When creating a project in VS Code, the user can select Timed Robot or Command Robot as their template. In timed-based code, actions are run in methods that are called iteratively. For example, a method with a conditional such as a check for a button press will be called continuously, and then when the condition is triggered, the action, such as a subsystem movement, will occur. Command-based code provides an alternative structure that tends to be cleaner. It allows users to define the action taken when a certain condition is met without the need to check the condition iteratively. It is important to note that command-based is an extension of timed-based and no functionality is lost when switching from timed-based to command-based.

Robot Methods

robotInit(): Runs once when the robot first turns on and the roboRIO boots up. The RobotContainer constructor should be called here if using command-based.
robotPeriodic(): Runs continuously every 20 ms while the robot is initialized.

autonomousInit(): Runs once when the robot is enabled in autonomous mode.
autonomousPeriodic(): Runs continuously every 20 ms while the robot is enabled in autonomous mode.

teleopInit(): Runs once when the robot is enabled in teleoperated mode.
teleopPeriodic(): Runs continuously every 20 ms while the robot is enabled in teleoperated mode.

testInit(): Runs once when the robot is enabled in test mode.
testPeriodic(): Runs continuously every 20 ms while the robot is enabled in test mode.

simulationInit(): Runs once when the simulation starts through VS Code.
simulationPeriodic(): Runs continuously every 20 ms while simulation occurs.

disabledInit(): Runs once when the robot is disabled regardless of mode.
disabledPeriodic(): Runs continuously every 20 ms while the robot is disabled.
disabledExit(): Runs once when the robot is enabled in any mode.

RobotContainer Methods

configureBindings(): All button bindings should be placed in this method. This method should only be called once in the RobotContainer constructor.

getAutonomousCommand(): Returns a command/command composition that is scheduled when the robot is enabled in autonomous mode.

Controllers

Timed-Based: XboxController is the class that represents an xbox controller. Its constructor takes in one parameter, which is the port that the controller is connected to in DriverStation. To use this object, call the method that corresponds to the desired button and then call one of the trigger methods.

- **rising()**: Returns true when the button state has just changed from false to true.
- **falling()**: Returns true when the button state has just changed from true to false.
- **getAsBoolean():** Returns the state of the button.

```
XboxController xboxController = new XboxController(port);

public void teleopPeriodic() {
     //Prints true when a is pressed and false when a is released
     if (xboxController.a().rising()) System.out.println(true);
     if (xboxController.a().falling()) System.out.println(false);
}
```

Command-Based: CommandXboxController is a command-based representation of the XboxController class. The parameter its constructor takes in is the port that the controller is connected to in DriverStation. In the configure bindings method in RobotContainer, configure all of the controls to each button. To bind a command to the Xbox controller, call the method that corresponds to the desired button to and then call one of the trigger methods.

- **onTrue()**: Calls the command once when the button is pressed.
- **onFalse()**: Calls the command once when the button is released.
- **whileTrue()**: Continuously calls the command while the button is pressed.
- **whileFalse()**: Continuously calls the command while the button is not pressed.

```
CommandXboxController commandXboxController =
     new CommandXboxController(port);

public void configureBindings() {
     //Prints true once when a is pressed and false once when a
     is released
     commandXboxController.a().onTrue(new PrintCommand(“true”))
          .onFalse(new PrintCommand(“false”));
}
```

Subsystem Files

When creating a subsystem class, the class should extend SubsystemBase. This allows the class to inherit and override the periodic method, giving each subsystem use of their own periodic method separate from the one in the Robot class. This periodic method runs similarly but independently to the periodic methods in the Robot class and is called every 20 ms.

```
public class ExampleSubsystem extends SubsystemBase {
     public void periodic() {

     }
}
```

Control Loops

DutyCycle Control

This runs the motor at a specified percentage, 0 being off and 1 being maximum power. It is typically used for velocity control for rollers or climbers. Usually, motors have a set() method that only needs to be called once when setting the speed and do not have to be continuously called.

```
motor.set(.5);
motor.set(0);
```

Feedback Loops

Overview

The main form of feedback control is PID control. A WPILib PIDController object calculates the percentage needed to reach a given position based on the motor's current position. Based on the difference between the current and goal position, the percentage applied to the motor is adjusted based on three values: Proportional (P), Integral (I), and Derivative (D).

P: P directly reacts to error. As the error between two points decreases, such as when a motor's current position approaches the goal position, less voltage is applied to the system by P causing the subsystem to slow down as it reaches the goal. Having P too high causes oscillations as the subsystem overshoots the goal position by applying too much voltage. Having P too low causes the subsystem to move slowly and it may stop before reaching the goal point.

I: I reacts to error over time. For example, the mechanism may stop slightly short of its goal with just P alone. As the total amount of error over time accumulates, the voltage applied by I will increase, fixing this small amount of error. Increasing P may not be able to solve this problem since it may just lead to oscillations. I is usually low compared to P and is not always needed, especially when using feedforward and motion profiling. If I is too high, it may cause oscillations as well. Since I can be slightly volatile, a PIDController has two methods that help control it:

setIntegratorRange(double minimumIntegral, double maximumIntegral): This limits the total output I can contribute to the PID calculation (default is -1.0 to 1.0).

setIZone(double iZone): This limits the range I can be active by setting the maximum amount of error where I will be applied. For example, setIzone(5) would configure the PIDController object to only apply I when the error is 5 or less.

D: D reacts to how fast the error is changing. If the error is changing at different speeds, meaning that the mechanism is not moving smoothly, it attempts to limit this. This usually helps curb overshoot by P. D is also low compared to P and is also not always needed, especially when using feedforward and motion profiling. If D is too high, the mechanism may become jittery while moving.

Code Description

A PIDController object has a calculate() method that calculates how much voltage a motor should apply to reach a given point. It takes in two parameters: the motor's current position and the goal position. As the motor's position approaches the goal position and the error decreases, the PIDController will apply less voltage, allowing the motor to gradually slow down as it reaches its goal position.

```
PIDController pidController = new PIDController(P, I, D);

public void periodic() {
     motor.setVoltage(pidController
          .calculate(motor.getPosition(), goalPosition));
}
```

It is important to note that the PID constants themselves do not have any units and can be used with percentage or voltage control. However, the magnitude of these constants change depending on what type of control is used.

Velocity Control

A PIDController object can calculate voltages for velocity control as well. The only difference is that instead of passing in the motor's position and goal position, the motor's velocity and goal velocity is.

```
public void periodic() {
     motor.setVoltage(pidController
          .calculate(motor.getVelocity(), goalVelocity));
}
```

Tuning

PID tuning is very trial and error based. However, it is a common practice to first set I and D to 0 and tune P first. The value that P can be varies based on each mechanism, so it is a good practice to start with very low values such as 0.01. Tune P till the mechanism is moving at an acceptable speed without oscillating back and forth continuously. If it is overshooting, add a small amount of D. Lastly, if there is a small amount of error once it has finished moving, add a very small amount of I. Keep in mind that feedforward and motion profiling usually allows PID values to be much higher than usual.

Feedforward Loops

Overview

Feedforward loops use information about the subsystem to help predict the voltage necessary for the desired action. It typically cannot be used alone for positional control, but it helps lighten the load of a feedback loop by making it account for less. It uses four constants, kS, kG, kV, and kA, to predict these voltages.

kS: This represents the amount of voltage required to overcome static friction and see any noticeable movement.

kG: This represents the amount of voltage required to counteract gravity for elevators and arms. It does not apply to rollers.

kV: This represents the amount of voltage required to achieve a desired velocity. The relationship between voltage and velocity is typically linear, allowing kV to represent the slope or relationship between the two. Its units are voltage / distance (in any units) for linear mechanisms or voltage / radians for pivot mechanisms.

kA: This represents the amount of voltage required to achieve a desired acceleration. It is generally unnecessary and can be omitted.

Code Description

Since there are generally three types of subsystems, rollers, linear mechanisms (elevators), and pivoting mechanisms (arms), there are three different feedforward classes: SimpleMotorFeedForward, ElevatorFeedForward, and ArmFeedForward. The main distinction between these is how they handle gravity. All three objects have a calculate() method that calculates the feedforward voltage based on the given constants, similarly to the calculate() method in the PIDController class.

SimpleMotorFeedForward: This is used for rollers and will not take a kG parameter. The calculate() method takes in two parameters: the desired velocity and acceleration for the mechanism to travel at.

ElevatorFeedForward: This is used for anything that moves linearly, similar to an elevator. It will apply kG uniformly across all positions. The calculate() method takes in two parameters: the desired velocity and acceleration for the mechanism to travel at.

ArmFeedForward: This is used for anything that pivots along a certain point, similar to an arm. It will scale kG based on the mechanism's current position since the voltage needed to counteract gravity changes as the mechanism moves. The calculate() method takes in the current position in radians along with the desired velocity and acceleration in radians.

The units for the desired velocity and acceleration should be in the same units as kV. Also, the acceleration parameter can be omitted if there isn't a specific desired acceleration value.

```
SimpleMotorFeedForward rollerFF =
     new SimpleMotorFeedForward(kS, kV);
ElevatorFeedForward linearFF =
     new ElevatorFeedForward(kS, kG, kV);
ArmFeedForward pivotFF =
     new ArmFeedForward(kS, kG, kV);

public void periodic() {
     rollerMotor.setVoltage(rollerPIDController
          .calculate(rollerMotor.getPosition(), goalPosition) +
          rollerFF.calculate(desiredVelocity));

     linearMotor.setVoltage(linearPIDController
          .calculate(linearMotor.getPosition(), goalPosition) +
          linearFF.calculate(desiredVelocity));

     pivotMotor.setVoltage(pivotPIDController
          .calculate(pivotMotor.getPosition(), goalPosition) +
          armFF.calculate(pivotMotor.getPositionRadians(),
          desiredVelocity));
}
```

Velocity Control

Feedback and feedforward loops can be used for velocity control for rollers as well with a similar implementation. Instead of the PIDController calculating between the current and desired position, it calculates between the current and desired velocity.

```
PIDController rollerPIDController = new PIDController(P, I, D);
SimpleMotorFeedForward rollerFF =
     new SimpleMotorFeedForward(kS, kV);

public void periodic() {
     rollerMotor.setVoltage(rollerPIDController
          .calculate(rollerMotor.getVelocity(), desiredVelocity)
          + rollerFF.calculate(desiredVelocity));
}
```

Tuning

kG: For linear mechanisms, this will be the lowest amount of voltage that will hold the mechanism up. Place the mechanism in any point where it is not supported by anything and test voltages to find the lowest amount of voltage that prevents it from falling. For pivot mechanisms, place the mechanism in a position where the force of gravity will be the strongest. This is usually when the mechanism is parallel to the ground. After that, test voltages to find the lowest amount of voltage that prevents it from falling. Roller mechanisms do not have a kG.

kS: Test voltages to find the lowest amount of voltage that causes any noticeable movement from the mechanism. When finding the kS for linear or pivot mechanisms, subtract kG from the lowest amount of voltage that causes movement as some voltage is simply allowing the mechanism to remain stationary. Make sure to keep pivot mechanisms parallel to the ground, similar to the tuning process of kG.

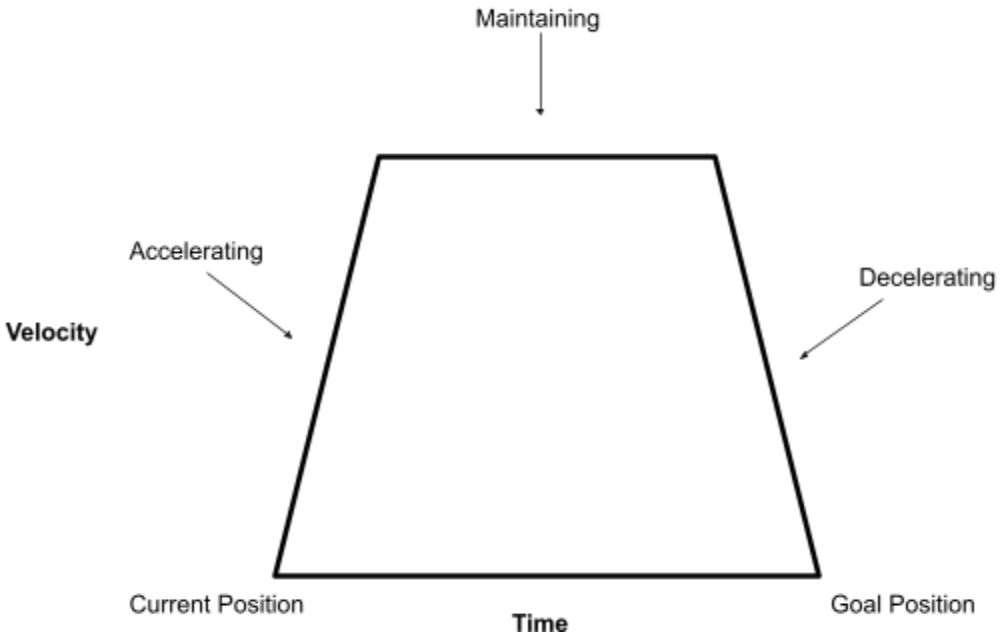
kV: Before tuning kV, tune kS and kG first. After obtaining those values, add any amount of voltage to kS and kG that moves the mechanism at a reasonable speed (usually 1 volt works) and record the general velocity it's moving at. This can be done through Phoenix Tuner for CTRE motors, REV Hardware Client for REV motors, print statements, or other logging softwares such as Advantage Scope. kV will be the amount of voltage added to kS and kG / velocity. Velocity can be any units for linear mechanisms as long as it stays consistent but should be in radians / sec for pivot mechanisms. For more accurate results, multiple different voltages can be added and the kV values can be averaged out.

Motion Profiling

Overview

Motion profiling refers to the process of generating a series of steps for the subsystem to go to in between the current and goal point. Not only does this greatly improve accuracy, but it also makes the mechanism's movement much more controlled. It is generally unnecessary for velocity control and primarily used for positional control.

The most common profile is a trapezoidal profile, which represents a graph of the desired velocity. The first side shows the subsystem accelerating till it reaches max velocity, maintaining it until it gets close to the goal point, and then smoothly decelerating until it hits the goal position.



A trapezoidal profile is made up of many different states. Each state holds a desired position and velocity. Instead of moving directly to the desired position, the mechanism can instead move to the position held by each state. Also, the velocity for each state varies based on the max velocity and acceleration. These can be customized for each profile. Max velocity affects the height of the profile, where increasing max velocity increases the height. Max acceleration affects the slopes of the sides of the profile, where increasing max acceleration increases slope.

Code Description

Apart from the TrapezoidProfile class itself, it has two subclasses: Constraints and State.

TrapezoidProfile.Constraints: The TrapezoidProfile.Constraints constructor takes in two parameters: max velocity and max acceleration. This is passed into the constructor of the TrapezoidProfile class when creating a TrapezoidProfile object.

```
TrapezoidProfile.Constraints constraints = new
     TrapezoidProfile.Constraints(maxVelocity, maxAcceleration);
TrapezoidProfile profile = new TrapezoidProfile(constraints);
```

TrapezoidProfile.State: The TrapezoidProfile.State constructor takes in two parameters: position and velocity. Typically, there are two variables that hold the current state and the goal state. The current state holds the current position and velocity of the motor while the goal state holds the desired position. The velocity for the goal state is usually 0 to specify that the mechanism should come to a complete stop when reaching the goal position. Also, the velocity of the current state and the acceleration calculated using two states next to each other can be used for the feedforward calculations.

```
TrapezoidProfile.State currentState =
     new TrapezoidProfile.State(0, 0);
TrapezoidProfile.State goalState =
     new TrapezoidProfile.State(0, 0);
```

TrapezoidProfile: The TrapezoidProfile constructor takes in a TrapezoidProfile.Constraints objects. Similarly to PIDControllers, it also has a calculate() method that returns the next state in the profile using three parameters: how far in the profile to travel (represented in seconds), the current state of the subsystem (should be continuously updated), and the goal state. The time usually is .02 since that is the default time in between calls to periodic, but it can also be measured with a timer for further accuracy.

```
TrapezoidProfile.Constraints constraints = new
     TrapezoidProfile.Constraints(maxVelocity, maxAcceleration);
TrapezoidProfile profile = new TrapezoidProfile(constraints);
TrapezoidProfile.State currentState =
     new TrapezoidProfile.State(0, 0);
TrapezoidProfile.State goalState =
     new TrapezoidProfile.State(0, 0);

public void setPosition(double position) {
     goalState = new TrapezoidProfile.State(position, 0);
     currentState = new TrapezoidProfile.State(
          motor.getPosition(), motor.getVelocity());
}

public void followLastProfile() {
     TrapezoidProfile.State nextState =
          profile.calculate(.02, currentState, goalState);
     double acceleration = (nextState.velocity -
          currentState.velocity) / .02;

     motor.setVoltage(pidController.calculate(
          motor.getPosition(), nextState.position) +
          feedforward.calculate(nextState.velocity,
          acceleration);

     currentState = nextState;
}

public void periodic() {
     followLastProfile();
}
```

Similarly to PIDControllers, the trapezoidal profile and constraints can be in any units as long as it is consistent. However, for pivot mechanisms, it may be necessary to convert the units into radians for the feedforward calculations.

CTRE Subsystem

Motors

Overview

CTRE motors are represented by the TalonFX class and require an ID parameter. Here are some common methods:

set(double speed): Runs the motor with duty cycle (percentage) control and takes in a value from -1 to 1.

setVoltage(double volts): Runs the motor at the specified voltage.

setControl(ControlRequest request): Runs the motor with the specified control request.

setPosition(double newValue): Sets the current recorded position of the motor to the specified value. Usually, the motor regards the position it was at when the robot first turns on as 0 rotations. This can be called when the robot is initializing to properly reset motors when code is restarted/redeployed.

getRotorPosition(): Returns the current position of the motor.

getPosition(): Returns the current position of the mechanism, which is influenced by gear ratios set in configurations.

getVelocity(): Returns the current velocity of the motor.

Configuration

A TalonFXConfiguration object can be applied to a motor to configure it in various ways such as if it is inverted, neutral mode (brake or coast), current limits, and slot configs. A slot config holds PID values, feedforward values, and the gravity type, which depends on if the subsystem is a pivot or linear mechanism and can be omitted for rollers.

```
TalonFX motor = new TalonFX(id);
TalonFXConfiguration motorConfig = new TalonFXConfiguration();

public void configureMotor() {
     motorConfig.MotorOutput.Inverted =
          InvertedValue.ClockwisePositive;
     //or InvertedValue.CounterClockwise_Positive

     motorConfig.MotorOutput.NeutralMode = NeutralModeValue.Brake
     //or NeutralModeValue.Coast;

     motorConfig.CurrentLimits.SupplyCurrentLimit =
          supplyCurrentLimit;
     motorConfig.CurrentLimits.StatorCurrentLimit =
          statorCurrentLimit;

     motorConfig.CurrentLimits.SupplyCurrentLimitEnable = true;
     motorConfig.CurrentLimits.StatorCurrentLimitEnable = true;

     Slot0Configs slot0Configs = new Slot0Configs()
          .withKP(P)
          .withKI(I)
          .withKD(D)
          .withKS(kS)
          .withKG(kG)
          .withKV(KV)
          .withGravityType(GravityTypeValue.Arm_Cosine);
          //or GravityTypeValue.Elevator_Static
     motorConfig.Slot0 = slot0Configs;

     motor.getConfigurator().apply(motorConfig);
     motor.setPosition(0); //Unnecessary but allows motor to
     resets its position when the robot initializes
}
```

Control Requests

As referenced previously, the setControl() method of a TalonFX can accept many different control request objects. The most common control requests are PositionVoltage, VelocityVoltage, and Follower.

PositionVoltage

This serves as an alternative to using a PIDController object for positional control. It takes in one parameter: the desired position. The setControl() method does not need to be called periodically and only needs to be called when changing the position.

withPosition(double position): Changes the position attribute held by the PositionVoltage object.

withSlot(int newSlot): Selects which slot the PositionVoltage will draw feedback and feedforward constants from as motors can have multiple in different slots.

withEnableFOC(boolean newEnableFOC): Enables FOC which increases total motor power (Phoenix Pro feature, only available on licensed devices).

withFeedForward(double newFeedForward): Uses a feedforward calculation which can be obtained from WPILib's feedforward classes.

```
PositionVoltage positionVoltage =
     new PositionVoltage(0).withSlot(0).withEnableFOC(true);

public void setPosition(double position) {
     motor.setControl(positionVoltage.withPosition(position)
          .withFeedForward(feedforward
          .calculate(desiredVelocity)));
}
```

Integration with Motion Profiling

A PositionVoltage object can be used in conjunction with trapezoid profiling similar to how a PIDController is used. However, it will need to be called periodically since the profile needs to be constantly updated.

```
TrapezoidProfile.Constraints constraints = new
     TrapezoidProfile.Constraints(maxVelocity,
     maxAcceleration);
TrapezoidProfile profile = new TrapezoidProfile(constraints);
TrapezoidProfile.State currentState =
     new TrapezoidProfile.State(0, 0);
TrapezoidProfile.State goalState =
     new TrapezoidProfile.State(0, 0);
PositionVoltage positionVoltage =
     new PositionVoltage(0).withSlot(0).withEnableFOC(true);
TalonFX motor = new TalonFX(id);

public void setPosition(double position) {
     goalState = new TrapezoidProfile.State(position, 0);
     currentState = new TrapezoidProfile.State(
          motor.getPosition(), motor.getVelocity());
}

public void followLastProfile() {
     TrapezoidProfile.State nextState =
          profile.calculate(.02, currentState, goalState);
     double acceleration = (nextState.velocity -
          currentState.velocity) / .02;

     motor.setControl(positionVoltage
          .withPosition(nextState.position)
          .withFeedForward(feedforward
          .calculate(nextState.velocity, acceleration));

     currentState = nextState;
}

public void periodic() {
     followLastProfile();
}
```

VelocityVoltage

This serves as an alternative to using a PIDController object for velocity control and functions similarly to PositionVoltage. It takes in one parameter: the desired velocity. The setControl() method does not need to be called periodically and only needs to be called when changing the velocity.

withVelocity(double newVelocity): Changes the velocity attribute held by the VelocityVoltage object.

withSlot(int newSlot): Selects which slot the VelocityVoltage will draw feedback and feedforward constants from as motors can have multiple in different slots.

withEnableFOC(boolean newEnableFOC): Enables FOC which increases total motor power (Phoenix Pro feature, only available on licensed devices).

withFeedForward(double newFeedForward): Uses a feedforward calculation which can be obtained from WPILib's feedforward classes.

```
VelocityVoltage velocityVoltage =
     new VelocityVoltage(0).withSlot(0).withEnableFOC(true);

public void setVelocity(double velocity) {
     motor.setControl(velocityVoltage.withVelocity(velocity)
          .withFeedForward(feedforward
          .calculate(velocity));
}
```

To stop a motor once it is spinning using VelocityVoltage, use the set(0) or setVoltage(0) methods to stop the motor instead of calling the withVelocity(0) method.

Follower

A motor can also be set to follow another motor, meaning that it will continuously copy another motor's output. It is primarily used when two motors are controlling the same subsystem and there isn't a need to have each motor act independently. Once the follower motor is set up, it is no longer necessary to apply any other control requests to it.

The Follower constructor takes in two parameters: the id of the master motor and whether it should be inverted compared to the master motor. Usually, the second parameter is set to true since follower motors are often facing different directions but it depends on each subsystem.

```
TalonFX masterMotor = new TalonFX(masterId);
TalonFX followerMotor = new TalonFX(followerId);
Follower follower = new Follower(masterId, true);

public void configureFollowerMotor() {
     followerMotor.setControl(follower);
}
```

Utility Methods

These are a few methods that are unnecessary but may be helpful to include in a subsystem class.

Check for Reached Goal Position

It may be helpful to have a method that returns whether the subsystem has reached a goal position or not. This helps especially when actions are run in sequence and subsystem movement needs to wait for previous movement to finish. It is also important to have reasonable tolerances as the subsystem will never exactly reach the goal position.

```
double position;

public void setPosition(double position) {
     this.position = position;
     //Rest of code to move the motor
}

public boolean isMechAtGoal() {
     return motor.getPosition() >= goal - lowerTolerance
          && motor.getPosition() <= goal + upperTolerance;
}
```

If using trapezoid profiling, this method can also include a check for if the profile itself has finished.

```
public boolean isMechAtGoal() {
     return motor.getPosition() >= goal - lowerTolerance
          && motor.getPosition() <= goal + upperTolerance
          && currentState.position == goalState.position
          && currentState.velocity == goalState.velocity;
}
```

Units

Keeping everything in motor rotations may be confusing while creating setpoints. For pivot mechanisms, it may be easier to input positions in terms of degrees of the mechanism and then convert it to motor rotations. Similarly, it may be easier to use distance measurements like meters for positions for linear mechanisms.

Conversion between mechanism degrees and motor rotations:

```
double motorRot = degrees / 360.0 / gearRatio;
```

Conversion between meters and motor rotations:

```
//drumCircumfrence refers to the spool that a rope goes around
for an elevator
double motorRot = meters / drumCircumfrence / gearRatio;
```

Changing Max Speeds

When using trapezoid profiling, it may be useful to be able to customize the speed of each action with a different max velocity and acceleration. The easiest way to do this is to recreate the profile with a different set of constraints.

```
TrapezoidProfile.Constraints constraints = new
     TrapezoidProfile.Constraints(maxVelocity, maxAcceleration);
TrapezoidProfile profile =
     new TrapezoidProfile(constraints);

public void changeProfileConstraints(double maxVelocity,
                                     double maxAcceleration) {
     profile = new TrapezoidProfile(new TrapezoidProfile
          .Constraints(maxVelocity, maxAcceleration);
}
```

Command-Based

Overview

In command-based, actions are run as commands when certain conditions are fulfilled. They run based off of triggers instead of having to check a condition each iteration. Commands are run by the Command Scheduler, which coordinates when to start, finish, and interrupt commands.

Each command has four parts, the initialization, execute, end, and isFinished condition. The initialization runs once when the command starts. The execute portion runs constantly while the command is being run. The end portion runs once right before the command ends, and it is triggered by the end condition. Each command also belongs to a certain subsystem. However, each subsystem can only run one command at a time, so if another command is triggered from one subsystem when one is already active, the active command will be interrupted.

WPILib provides numerous Command classes that command objects can be created from. They usually require lambdas to convert methods into Runnable objects or booleans into BooleanSupplier objects. These command objects can be run through a binding to a CommandXboxController object set up in the configureBindings() method or in the getAutonomousCommand() method, both in the RobotContainer class.

Types of Commands

InstantCommand: the command starts, executes the action, and finishes instantly.

```
InstantCommand command = new InstantCommand(() →
     System.out.println(“I run instantly”));
```

WaitCommand: Waits the desired number of seconds.

```
WaitCommand command = new WaitCommand(5);
```

PrintCommand: Prints a message, similarly to a system print, but only takes in strings.

```
PrintCommand command = new PrintCommand(“Hello World”);
```

FunctionalCommand: This has five parameters, initialization, execute, end condition, end, and the subsystem(s) required. The end parameter also takes in a boolean which is true if the command was interrupted. This gives the most control apart from declaring an entire command class as each part can be defined individually.

```
FunctionalCommand command = new FunctionalCommand(
     () → { //initialize
          System.out.println(“I print once at the start”);
     },
     () → { //execute
          System.out.println(“I print continuously”);
     },
     (interrupted) → { //end
          System.out.println(“I print once at the end”);
     },
     () → condition, //isFinished - any boolean
     exampleSubsystem);
```

Custom Commands

A class can also extend the Command class. This allows it to inherit the initialize(), execute(), end(), and isFinished() methods and customize them. When extending the Command class, the addRequirements class should be called with an object of a class that extends SubsystemBase. This allows the CommandScheduler to properly schedule command objects created from this class.

```
public class ExampleCommand extends Command {
     public ExampleCommand() {
          addRequirements(exampleSubsystem);
     }

     public void initialize() {

     }

     public void execute() {

     }

     public void end(boolean interrupted) {

     }

     public boolean isFinished() {

     }
}
```

To use this class, simply create an object of this class, similarly to the default command classes. After, it functions the same as any other command object and can be run by binding it to a button to a CommandXboxController object.

```
ExampleCommand exampleCommand = new ExampleCommand();
```

Types of Command-Groups

SequentialCommandGroup: Runs all of the commands in sequence and will wait for each one to finish before executing the next one.

```
SequentialCommandGroup commandGroup =
     new SequentialCommandGroup(
          new PrintCommand(“1”),
          new PrintCommand(“2”),
          new PrintCommand(“3”));
```

ParallelCommandGroup: Runs all of the commands at the same time. If multiple commands use the same subsystems, it will throw an exception

```
ParallelCommandGroup commandGroup =
     new ParallelCommandGroup(
          new PrintCommand(“1”),
          new PrintCommand(“1”),
          new PrintCommand(“1”));
```

ConditionalCommand: Runs one of the two commands based on the condition passed.

```
ConditionalCommand command = new ConditionalCommand(
     new PrintCommand(“true”),
     new PrintCommand(“false”),
     () → condition)
```

SelectCommand: Runs one of the multiple commands based on the state of an enum. It is similar to a switch-case statement.

```
public enum Enum {
     ONE,
     TWO,
     THREE
}

ExampleEnum enum = ExampleEnum.ONE;
SelectCommand command = new SelectCommand<>(
     Map.ofEntries(
          Map.entry(ExampleEnum.ONE, new PrintCommand(“1”),
          Map.entry(ExampleEnum.TWO, new PrintCommand(“2”),
          Map.entry(ExampleEnum.THREE, new PrintCommand(“3”)
     )),
     () → enum);
```

Decorators

Commands can be chained together to create command compositions. This can be done with command groups, as well as decorators which can be added on to the ends of commands or command groups.

andThen(): Will run the specified command after the original command executes. It functions as a SequentialCommandGroup.

```
new PrintCommand("1").andThen(new PrintCommand("2"))
```

and(): Will run the specified command while the original command executes. It functions as a ParallelCommandGroup.

```
new PrintCommand("1").and(new PrintCommand("1"))
```

onlyIf(): Will only run the original command if the condition is met.

```
new PrintCommand("true").onlyIf(condition)
```

unless(): Will run the original command unless the condition is met.

```
new PrintCommand("false").unless(condition)
```

until(): Will only run the original command while the condition is met and will terminate it once the condition is no longer met.

```
new PrintCommand("false").until(condition)
```

beforeStarting(): Will run the specified command before the original command.

```
new PrintCommand("2").beforeStarting(new PrintCommand("1"))
```

repeatedly(): Repeats the command when it ends, causing it to continuously run from start to finish until it is interrupted.

```
new PrintCommand("I print continuously").repeatedly()
```

finallyDo(): Runs the specified command after the end section of the original command.

```
new PrintCommand(“1”).finallyDo(new PrintCommand(“2”))
```

withTimeout(): Ends the command after a certain amount of time measured in seconds.

```
new PrintCommand(“1”).withTimeout(1)
```

Template Commands

Positional Control

A positional command for a CTRE subsystem using motion profiling as defined previously would call setPosition() in the initialize section, followLastProfile() in the execute section, and would finish when it has reached the goal position using the isMechAtGoal() method. However, it is important that the subsystem does not stop profiling once the command ends to prevent it from falling or deviating from its goal position. In the end section, it will begin calling followLastProfile() in the periodic method of the subsystem by toggling on a boolean.

Add these modifications to the methods in CTRE Subsystem section:

```
boolean followLastProfile = false;

public void setFollowLastProfile(boolean followLastProfile) {
     this.followLastProfile = followLastProfile;
}

public void periodic() {
     if (followLastProfile) followLastProfile();
}
```

It is important that the followLastProfile() method is not called in the execute portion of the command and periodic at the same time. However, since it still needs to be called once the command finishes, it still needs to exist in the periodic method. While the command runs, followLastProfile() will be called in the execute portion and once it ends, it will be called in the periodic method by toggling the boolean on.

```
public class PositionCommand extends Command {
     private double goalPosition;
     private ExampleSubsystem exampleSubsystem;

     //ExampleSubsystem extends SubsystemBase and uses the CTRE
     Subsystem methods previously explained
     public PositionCommand(ExampleSubsystem exampleSubsystem,
                            double goalPosition) {
          this.goalPosition = goalPosition;
          this.exampleSubsystem = exampleSubsystem;
```

```
        addRequirements(exampleSubsystem);
    }

    public void initialize() {
        exampleSubsystem.setPosition(goalPosition);

        //Disables the call to followLastProfile in periodic so
        it isn't called in the execute and periodic methods at
        the same time
        exampleSubsystem.setFollowLastProfile(false);
    }

    public void execute() {
        exampleSubsystem.followLastProfile();
    }

    public void end(boolean interrupted) {
        //Enables the call to followLastProfile in the periodic
        method so it continues holding its position once the
        command ends.
        exampleSubsystem.setFollowLastProfile(true);
    }


    public boolean isFinished() {
        return exampleSubsystem.isMechAtGoal();
    }
}
```

Velocity Control

A command for velocity control is much simpler. In the initialize method, the setVelocity() method is called. There is nothing in the execute method since nothing is changing while the command is running. In the end method, set(0) will be called to stop the motor. The isFinished method typically just returns false as velocity commands usually just finish when interrupting instead of when reaching a condition. To prevent them from obstructing SequentialCommandGroups, call them in ParallelCommandGroups as rollers can typically move independently. However, if the roller must spin up before further actions can be taken, an alternate approach where it ends once it has reached the goal velocity and doesn't stop in the end method can be applied.

```
public class VelocityCommand extends Command {
     private double goalVelocity;
     private ExampleSubsystem exampleSubsystem;

     //ExampleSubsystem extends SubsystemBase and uses the CTRE
     Subsystem methods previously explained
     public VelocityCommand(ExampleSubsystem exampleSubsystem,
                            double goalVelocity) {
          this.goalVelocity = goalVelocity;
          this.exampleSubsystem = exampleSubsystem;

          addRequirements(exampleSubsystem);
     }

     public void initialize() {
          exampleSubsystem.setVelocity(goalVelocity);
     }

     public void execute() {

     }

     public void end(boolean interrupted) {
          exampleSubsystem.set(0);
```

```
    }


    public boolean isFinished() {
        return false;
    }
}
```

These classes can be customized to allow for custom velocity and acceleration values, updating goal positions or velocities while the command is running, adding offset control, and more.