Day 14 CS570

DriveStraight Command

Goal

By the end of the next class you should be able to create an autonomous command that makes the ROMI drive straight for more than 5 seconds. By driving straight we mean that we don't see the natural drift that seems to occur with most ROMI's given the drive code that we generally use.

Starter Code You can find starter code for this project here: https://github.com/mbardoeChoate/DriveStraight2024

Let's go through the starter code. There are three files, `robot.py`, `subsystems/drivetrain.py`, `pyproject.toml`. You may need to make sure that set up the project to be a robotpy project. You can do that by running the following command:

```
python3 -m pip install robotpy
python3 -m robotpy init
python3 -m robotpy sync
```

Here is the code for the robot.

```
import os
from commands2 import CommandScheduler, TimedCommandRobot
from wpilib import Joystick, Spark
from subsystems.drivetrain import Drivetrain
import ntcore

os.environ["HALSIMWS_HOST"] = "10.0.0.2"
os.environ["HALSIMWS_PORT"] = "3300"

class LilRedRobot(TimedCommandRobot):

    def robotInit(self):
        self.joystick = Joystick(0)
        self.drivetrain = Drivetrain()
        self.nt=ntcore.NetworkTableInstance.getDefault()
        self.scheduler = CommandScheduler.getInstance()


    def robotPeriodic(self):
        pass

    def autonomousInit(self):
        pass

    def autonomousPeriodic(self):

        pass

    def teleopInit(self):
        pass

    def teleopPeriodic(self):
        forward = self.joystick.getX()
        rotation = self.joystick.getY()
        self.drivetrain.arcadeDrive(forward, rotation)
```

First, notice that this robot inherits from `TimedCommandRobot`. This is a new class that we are using that will allow

us to use the command based system that we will be using in the future. Commands are a way of controlling what the robot is doing at any given time. A `Command` is an abstraction of the actions that the robot can take.

Another new addition in the code is inclusion of the `self.nt=ntcore.NetworkTableInstance.getDefault()` line. This line will allow us to use the NetworkTables to get the data from the ROMI. NetworkTables is a way for information to be shared between the ROMI and the computer that is running the code.

The other new line is the `self.scheduler = CommandScheduler.getInstance()` line. This line will allow us to use the CommandScheduler to run the commands that we will be creating in the future.

Drivetrain The drivetrain code is as follows:

```
import math

import commands2
import ntcore
import wpilib
import wpilib.drive
import romi


class Drivetrain(commands2.Subsystem):
    kCountsPerRevolution = 1440.0
    kWheelDiameterInch = 2.75591

    def __init__(self) -> None:
        super().__init__()

        # The Romi has the left and right motors set to
        # PWM channels 0 and 1 respectively
        self.leftMotor = wpilib.Spark(0)
        self.rightMotor = wpilib.Spark(1)

        # The Romi has onboard encoders that are hardcoded
        # to use DIO pins 4/5 and 6/7 for the left and right
        self.leftEncoder = wpilib.Encoder(4, 5)
        self.rightEncoder = wpilib.Encoder(6, 7)

        # Set up the differential drive controller
        self.drive = wpilib.drive.DifferentialDrive(self.leftMotor, self.rightMotor)

        # Set up the RomiGyro
        self.gyro = romi.RomiGyro()

        # Set up the BuiltInAccelerometer
        self.accelerometer = wpilib.BuiltInAccelerometer()

        # Use inches as unit for encoder distances
        self.leftEncoder.setDistancePerPulse(
            (math.pi * self.kWheelDiameterInch) / self.kCountsPerRevolution
        )
        self.rightEncoder.setDistancePerPulse(
            (math.pi * self.kWheelDiameterInch) / self.kCountsPerRevolution
        )
        self.resetEncoders()

    def arcadeDrive(self, fwd: float, rot: float) -> None:
        """
```

```
        Drives the robot using arcade controls.

        :param fwd: the commanded forward movement
        :param rot: the commanded rotation
        """
        self.drive.arcadeDrive(rot, fwd)

    def resetEncoders(self) -> None:
        """Resets the drive encoders to currently read a position of 0."""
        self.leftEncoder.reset()
        self.rightEncoder.reset()

    def getLeftEncoderCount(self) -> int:
        return self.leftEncoder.get()

    def getRightEncoderCount(self) -> int:
        return self.rightEncoder.get()

    def getLeftDistanceInch(self) -> float:
        return self.leftEncoder.getDistance()

    def getRightDistanceInch(self) -> float:
        return self.rightEncoder.getDistance()

    def getAverageDistanceInch(self) -> float:
        """Gets the average distance of the TWO encoders."""
        return (self.getLeftDistanceInch() + self.getRightDistanceInch()) / 2.0

    def getAccelX(self) -> float:
        """The acceleration in the X-axis.

        :returns: The acceleration of the Romi along the X-axis in Gs
        """
        return self.accelerometer.getX()

    def getAccelY(self) -> float:
        """The acceleration in the Y-axis.

        :returns: The acceleration of the Romi along the Y-axis in Gs
        """
        return self.accelerometer.getY()

    def getAccelZ(self) -> float:
        """The acceleration in the Z-axis.

        :returns: The acceleration of the Romi along the Z-axis in Gs
        """
        return self.accelerometer.getZ()

    def getGyroAngleX(self) -> float:
        """Current angle of the Romi around the X-axis.

        :returns: The current angle of the Romi in degrees
        """
        return self.gyro.getAngleX()

    def getGyroAngleY(self) -> float:
```

```
        """Current angle of the Romi around the Y-axis.

        :returns: The current angle of the Romi in degrees
        """
        return self.gyro.getAngleY()

    def getGyroAngleZ(self) -> float:
        """Current angle of the Romi around the Z-axis.

        :returns: The current angle of the Romi in degrees
        """
        return self.gyro.getAngleZ()

    def resetGyro(self) -> None:
        """Reset the gyro"""
        self.gyro.reset()

    def periodic(self) -> None:
        self.nt_drivetrain=ntcore.NetworkTableInstance.getDefault().getTable("Drivetrain")
        self.nt_drivetrain.putNumber("Left Encoder", self.getLeftEncoderCount())
        self.nt_drivetrain.putNumber("Right Encoder", self.getRightEncoderCount())
        self.nt_drivetrain.putNumber("Left Distance", self.getLeftDistanceInch())
        self.nt_drivetrain.putNumber("Right Distance", self.getRightDistanceInch())
        self.nt_drivetrain.putNumber("Average Distance", self.getAverageDistanceInch())
        self.nt_drivetrain.putNumber("Accel X", self.getAccelX())
        self.nt_drivetrain.putNumber("Accel Y", self.getAccelY())
        self.nt_drivetrain.putNumber("Accel Z", self.getAccelZ())
        self.nt_drivetrain.putNumber("Gyro X", self.getGyroAngleX())
        self.nt_drivetrain.putNumber("Gyro Y", self.getGyroAngleY())
        self.nt_drivetrain.putNumber("Gyro Z", self.getGyroAngleZ())
```

The drivetrain class is a subsystem that will allow us to control the ROMI. The drivetrain class has a number of methods that will allow us to control the ROMI. The `arcadeDrive` method will allow us to drive the ROMI using arcade controls. The `resetEncoders` method will reset the encoders on the ROMI. The `getLeftEncoderCount` and `getRightEncoderCount` methods will get the counts from the encoders. The `getLeftDistanceInch` and `getRightDistanceInch` methods will get the distance in inches from the encoders. The `getAverageDistanceInch` method will get the average distance from the encoders. The `getAccelX`, `getAccelY`, and `getAccelZ` methods will get the acceleration in the X, Y, and Z directions. The `getGyroAngleX`, `getGyroAngleY`, and `getGyroAngleZ` methods will get the angle of the ROMI in the X, Y, and Z directions. The `resetGyro` method will reset the gyro. The `periodic` method will update the NetworkTables with the data from the ROMI.

Look through that code and see if you can understand each line. The goal of this class is to make it easy to control the motion of the ROMI, but also access the data connected to the movement of the ROMI. So rather than deal with Encoders, Gyros, or Accelerometers individualy, we can use the `Drivetrain` class to access all of that data. We can still use the drivetrain properties `.gyro`, `.accelerometer`, `.leftEncoder`, and `.rightEncoder` to access the individual sensors if we need to.

DriveStraight Command

The goal of this class is to create a command that will make the ROMI drive straight autonomously for more than 5 seconds. To do that we create a new folder, and a new file. The folder is called `commands` and the file is called `drivestraight.py`. We don't really need a new folder in this project, but it is a good practice to keep the commands in a separate folder.

Let's start with the skeletal structure for most commands.

```
import commands2
from wpimath.controller import PIDController
```

```
class DriveStraight(commands2.CommandBase):

    def __init__(self, required_subsystem):
        super().__init__()
        self.subsystem = required_subsystem
        self.addRequirements(required_subsystem)

    def initialize(self):
        pass

    def execute(self):
        pass

    def isFinished(self):
        pass

    def end(self):
        pass
```

The `DriveStraight` class is a subclass of `CommandBase`. The `__init__` method takes in a required subsystem and adds that subsystem as a requirement for the command. The `initialize` method is called when the command is first started. The `execute` method is called repeatedly while the command is running. The `isFinished` method is called to determine if the command is finished. The `end` method is called when the command is finished.

This is the structure of most basic commands. There are other more complex commands that use the basic builing blocks of the `CommandBase` class to create more complex behaviors. But driving straight is a pretty simple behavior.

The last weird piece is in the imports the `PIDController` class is imported from `wpimath.controller`. This is a new class that we will be using to control the motion of the ROMI.

init The `__init__` method is where we will set up the command. We will need to set the subsystem that we are using, and add that subsystem as a requirement for the command.

```
    def __init__(self, drivetrain, speed:float, time:float):
        super().__init__()
        self.drivetrain = drivetrain # we are going to use the drivetrain subsystem
        self.addRequirements(self.drivetrain) # Tell other commands that the drivetrain is being used
        self.speed = speed # The speed that we want the robot to drive at
        self.pidController = PIDController(.2, 0.0, 0.0)
```

The lines:

```
        self.drivetrain = drivetrain # we are going to use the drivetrain subsystem
        self.speed = speed # The speed that we want the robot to drive at
```

Simply make it so that we can keep reference to these essential qualities of the command. The line:

```
        self.addRequirements(self.drivetrain) # Tell other commands that the drivetrain is being used
```

is important because it tells the command scheduler that this command is using the drivetrain subsystem. This is important because the command scheduler will make sure that no other command is using the drivetrain subsystem at the same time. This is important because we don't want two commands to be trying to control the drivetrain at the same time.

The line:

```
        self.pidController = PIDController(.02, 0.0, 0.0)
```

is important because it sets up a PIDController. A PIDController is a way of controlling the motion of the ROMI. The PIDController takes in three values, the P, I, and D values. The P value is the proportional value, the I value is the integral value, and the D value is the derivative value. These values are used to control the motion of the ROMI. We will be using the PIDController to control the motion of the ROMI. PIDControllers are a way of controlling

any system with feedback. In general, you consider how far the system is from the desired state and you use that information to adjust the system to get closer to the desired state.

In FRC robotics we mostly will use the P and D values. The P value is used to measure how responsive the system is deviations from the set point. You might think of it as the stiffness of a spring that pulls the robot back in line if there is deviation from the set point. The D value is a dampening value that makes it so that the a very stiff spring doesn't create too much oscillation.

The I value is used to correct for any bias in the system. If the system is always off by a little bit, the I value will correct for that bias. We don't often have that issue, and I can often lead to more oscillation in the system.

initialize The `initialize` method is called when the command is first started. This is where we will set up the PIDController.

```
def initialize(self):
    # Reset encoders and PID controller at the start
    self.drivetrain.resetEncoders()
    self.pidController.reset()
    self.pidController.setSetpoint(0)  # Set the target difference to zero
```

The lines:

```
    self.drivetrain.resetEncoders()
    self.pidController.reset()
    self.pidController.setSetpoint(0)  # Set the target difference to zero
```

are important because they reset the encoders and the PIDController. This is important because we want to make sure that the encoders are set to zero and the PIDController is set to zero at the start of the command.

execute The `execute` method is called repeatedly while the command is running. This is where we will control the motion of the ROMI.

```
def execute(self):
    error=self.drivetrain.getLeftDistanceInch() - self.drivetrain.getRightDistanceInch()
    output=self.pidController.calculate(error)
    self.drivetrain.arcadeDrive(self.speed, output)
```

The line:

```
    error=self.drivetrain.getLeftDistanceInch() - self.drivetrain.getRightDistanceInch()
```

is important because it calculates the error between the left and right encoders. This is important because we want to make sure that the ROMI is driving straight. If the left and right encoders are the same, then the ROMI is driving straight. If the left and right encoders are different, then the ROMI is not driving straight.

The line:

```
    output=self.pidController.calculate(error)
```

is important because it calculates the output of the PIDController. This is important because we want to make sure that the ROMI is driving straight. The PIDController will take in the error and calculate the output. The output will be used to control the motion of the ROMI.

The line:

```
    self.drivetrain.arcadeDrive(self.speed, output)
```

is important because it will drive the ROMI. The `arcadeDrive` method will take in the speed and the output of the PIDController and drive the ROMI.

isFinished The `isFinished` method is called to determine if the command is finished. This is where we will determine if the command is finished.

```
def isFinished(self):
    return False
```

Basically we don't want the command to finish until the time is up. So we will return False until the time is up.

end The `end` method is called when the command is finished. This is where we will clean up the command.

```
    def end(self, interrupted):
        self.drivetrain.arcadeDrive(0, 0)
```

The line:

```
        self.drivetrain.arcadeDrive(0, 0)
```

is important because it will stop the ROMI. We want to make sure that the ROMI stops when the command is finished.

Integrating the Command

We need to make an instance of this DriveStraight command in the `robot.py` file. First, we must import the file at the top.

```
from commands.drivestraight import DriveStraight
```

Then we need to add the command to the scheduler in the `autonomousInit` method.

```
    def autonomousInit(self):
        self.scheduler.schedule(DriveStraight(self.drivetrain,
                                              0.5).withTimeout(5))
```

The line:

```
        self.scheduler.schedule(DriveStraight(self.drivetrain,
                                              0.5).withTimeout(5))
```

is important because it will schedule the DriveStraight command to run for 5 seconds at a speed of 0.5.

Tuning the PIDController

The values of P, I, and D are important because they will control the motion of the ROMI. If the values are too high, the ROMI will oscillate. If the values are too low, the ROMI will not drive straight. We will need to tune the values of P, I, and D to get the ROMI to drive straight. We will need to experiment with the values of P, I, and D to get the ROMI to drive straight.

General Thoughts about Tuning a PID Controller

In general, the thoughts on how to tune (pick the values for P and D) a PID controller are as follows:

1. Start with the P value. Increase the P value until the system starts to oscillate. Then decrease the P value until the system only oscillates a little. It is important that the P value is big enough to make the system respond to changes, but not so big that the system oscillates too much. Imagine that there is a spring that is pulling the system back to the set point. The P value is the stiffness of the spring.
2. Increase the D value until the system stops oscillating or the oscillation is very mild, and the system settles quickly. The D value is like the level of thickness of the fluid that the spring in the P analogy is in. The fluid dampens the oscillation of the spring.
3. Graphing the error over time can be helpful. If the error is oscillating, then the P value is too high. If the error is not responding to changes, then the P value is too low. If the error is oscillating and the oscillation is not dampened, then the D value is too low. If the error is not oscillating, but the system is not responding to changes then the P is too low or the D value is too high.