Hyperdrive Manual

By Brach Knutson

V. 2021

3695 FOXIMUS PRIME

Table of Contents

Overview

Hyperdrive is An easy-to-implement autonomous driving library for FRC robots, developed by FRC team 3695, Foximus Prime. It was able to produce a cumulative time of 28.3 seconds on the Auto-Nav challenge and a 9.9 second cumulative time on the Galactic Search challenge in the 2021 FRC season. In its current state, this library only provides support for tank-style robots. Hyperdrive works off of a “record” - “emulate” paradigm. A human driver manually drives the robot through a path, “recording” it, and Hyperdrive can make the robot drive the same path autonomously, “emulating” it. Hyperdrive tracks the robot’s position on the field, and makes its calculations on the spot rather than pre-compiling them.

This manual will provide documentation for the different classes and methods of this library, as well as a guide to getting it to work with your robot. As you read this guide, keep in mind that the project’s test environment (which can be found at github.com/BTK203/Hyperdrive) can be used as a working example of the library in action.

Class and Method Reference

This chapter overviews the different classes and libraries that are available to programmers for use, as well as some other classes that exist within the library.

```
public class Hyperdrive
```

A tool that can autonomously drive the robot around. This tool provides the resources necessary for programmers to record and emulate paths that the robot drives, all with a few simple lines of code. In addition, it handles all communication with the PathVisualizer app.

An instance of this class can go anywhere in the robot code, but it is recommended that it is either placed in the `RobotContainer` class or the drivetrain subsystem class.

```
public Hyperdrive(
     final Units.LENGTH lengthUnit,
     final double motorUnitsPerUnit,
     final Units.FORCE weightUnit,
     final double robotWeight,
     int pvPort)
```

Creates a new `Hyperdrive` object. This is the “main” constructor.

Arguments	
`final Units.LENGTH lengthUnit`	The units of length (inches, meters, yards, etc) to use.
`final double motorUnitsPerUnit`	The number of motor position units (ticks, rotations, etc) that equal 1 `lengthUnit`. For example, team 3695’s 2021 robot had a value of 0.472, which is the number of motor rotations required to move it forward 1 inch. This value will vary depending on the diameters of the drive wheels or the gear ratio of the drive gearboxes.

	To measure this value for your robot, drive your robot forward a certain distance (preferably 10 feet or more for greatest accuracy), and record the displacement of the drive motors. This value is equal to the expression $\frac{motor\ displacement}{distance\ driven}$.
`final Units.FORCE weightUnit`	The units of weight that the robot is measured in.
`final double robotWeight`	The weight of the robot. This must be measured in the same unit as `weightUnit`.
`int pvPort`	The port number to use to communicate with the PathVisualizer app. Before deciding on this value, check the season's game manual for the list of ports that are blocked by FMS.

```
public Hyperdrive(
     final Units.LENGTH distanceUnits,
     final double motorUnitsPerUnit,
     final Units.FORCE weightUnit,
     final double robotWeight)
```

Creates a new `Hyperdrive` object. This is an overload of the "main" constructor which uses the default port, 3695, to communicate with PathVisualizer.

Arguments	
`final Units.LENGTH lengthUnit`	The units of length (inches, meters, yards, etc) to use.
`final double motorUnitsPerUnit`	Number of motor units per 1 `lengthUnit`. For the full description of this argument, see the "main" `Hyperdrive` constructor documentation.
`final Units.FORCE weightUnit`	The units of weight that the robot is measured in.
`final double robotWeight`	The weight of the robot. This must be measured in the same unit as `weightUnit`.

```
public Hyperdrive(
     final Units.LENGTH lengthUnit,
     final double motorUnitsPerUnit)
```

Creates a new `Hyperdrive` object. This is an overload of the "main" constructor. which uses the default robot weight, 125 pounds, and the default PathVisualizer port, 3695.

Arguments	
`final Units.LENGTH lengthUnit`	The units of length (inches, meters, yards, etc) to use.
`final double motorUnitsPerUnit`	Number of motor units per 1 `lengthUnit`. For the full description of this argument, see the "main" `Hyperdrive` constructor documentation.

```
public void update(
     double distanceTravelled,
     double direction)
```

Updates the Hyperdrive. The Hyperdrive will use the distance that the robot has travelled and the direction that it has travelled in to calculate the robot's position on the field. That information can be viewed on SmartDashboard or Shuffleboard as "Current Robot Position" and accessed in code with `getRobotPositionAndHeading()`.

It is recommended that this method be called in one of the robots `periodic()` methods. It is possible to call this method in a `while` loop in a separate thread, but programmers should be mindful of how often they reference encoder values from their motor controllers, as this can overload the CAN network and cause undesired motor behavior.

Arguments	
`double distanceTravelled`	The total displacement of the robot since the last zero, in motor units (ticks, rotations, etc).
`double direction`	The direction in which the robot is currently travelling, in degrees.

```
public void update(
     double leftDistance,
     double rightDistance,
     double heading)
```

Updates the Hyperdrive. The Hyperdrive will use the distance that the robot has travelled and the direction that it has travelled in and calculate the robot's position on the field. That information can be viewed on SmartDashboard or Shuffleboard as "Current Robot Position" and can be accessed in code with `getRobotPositionAndHeading()`.

Teams with tank-style robots will find this overload of the method is easier to use than the other one because they can simply pass the positions of their drive motors as arguments.

It is recommended that this method be called in one of the robots `periodic()` methods. It is possible to call this method in a `while` loop in a separate thread, but programmers should be mindful of how often they reference encoder values from their motor controllers, as this can overload the CAN network and cause undesired motor behavior.

Arguments	
`double leftDistance`	The current position of the left motors.
`double rightDistance`	The current position of the right motors.
`double heading`	The direction in which the robot is currently travelling, in degrees.

```
public Point2D getRobotPositionAndHeading()
```

Returns the current position and heading of the robot in field space. The coordinates will be reported in the same units that were specified in the `Hyperdrive` constructor.

Returns: The robot's position and heading on the field.

```
public void zeroPositionAndHeading(
     boolean waitForEncoders)
```

Sets the position and heading of the robot to 0. This method should be called when the robot is standing still to ensure accurate and reliable results.

Arguments	
`boolean waitForEncoders`	This should be `true` if the drivetrain encoders are also being zeroed, otherwise `false`. This is because the motors do not zero immediately when the method is called; the message has to be sent through the can network and registered with the motor controller itself. This lag could cause a jump in encoder position which would ruin any recently set positional zero (the position would zero immediately, but the motor position would zero later, changing the motor position, which un-zeros the robot position). To counter that, this method can wait for the motor positions to zero before actually setting the position to 0.

```
public void zeroPositionAndHeading()
```

Sets the position and heading of the robot to 0 without waiting for the drivetrain motors to zero first. This is equivalent to calling `zeroPositionAndHeading(false)`.

```
public void setPositionAndHeading(
     Point2D newPositionAndHeading)
```

Sets the position and heading of the robot. This method is useful as it can be called at any time during the robot’s operation, even while it is moving. You can use this method to set the robot’s position relative to an object or use it to set and maintain the robot’s true position on the field.

Arguments	
`Point2D newPositionAndHeading`	The new position and heading of the robot.

```
public Units.LENGTH getLengthUnits()
```

Returns the units of length that the Hyperdrive has been configured to use.

Returns: The Hyperdrive's units of length.

```
public void sendPath(
     Path path,
     String name)
```

Sends a path to the PathVisualizer client, if one is connected. If PathVisualizer is connected to the robot, and the "Live" option is enabled, the path being sent with this method will appear on the screen under the name that it was given.

Arguments	
`Path path`	The Path to send to PathVisualizer.
`String name`	The name of the path to send. The Path will appear on the manifest with this name.

```
public void initializeRecorder(
     String file)
```

Starts the path recorder to record a path that the robot will drive. After this method is called, the recorder will automatically record the robot's movement until the `stopRecorder()` method is called.

Arguments	
`String file`	The file to record the path to.

```
public void initializeRecorder()
```

Starts the path recorder to record a path that the robot will drive. After this method is called, the recorder will automatically record the robot's movement until the `stopRecorder()` method is called.

```
public void stopRecorder()
```

Stops the path recorder if it is currently active. The path being recorded will be saved to file and the Recorder will no longer record a Path automatically. The recorded Path can be accessed using `getRecordedPath()` until the recorder is started again. This method will also send the Path that was just recorded to PathVisualizer for viewing. If PathVisualizer is connected, and the "Live" option is enabled, then the Path will appear on the screen.

```
public Path getRecordedPath()
```

Returns the Path that the recorder has just recorded. If no such path exists, then the returned Path's `isValid()` method will return `false`.

Returns: The most recently recorded path.

```
public void loadPath(
     Path path,
     IEmulateParams parameters)
```

Loads a path for Hyperdrive to emulate.

Arguments	
`Path path`	The path that the robot should drive.
`IEmulateParams parameters`	The parameters to use while driving the path.

```
public void loadPath(
     String filePath,
     IEmulateParams parameters)
```

Loads a path for Hyperdrive to emulate.

Arguments

`String filePath`	The file location of the path that the robot should drive.
`IEmulateParams parameters`	The parameters to use when driving the path.

```
public void loadPath(
     Path path)
```

Loads a path for Hyperdrive to emulate. Sets the emulation parameters to the defaults.

Arguments	
`Path path`	The path that the robot should drive.

```
public void loadPath(
     String filePath)
```

Loads a path for Hyperdrive to emulate. Sets the emulation parameters to the defaults.

Arguments	
`String filePath`	The file location of the path that the robot should drive.

```
public void specifyResultsFile(
     String filePath)
```

Forces Hyperdrive to record the results of the driven path to the file location provided.

Arguments	
`String filePath`	The location of the new results file.

```
public void performInitialCalculations()
```

Performs some important path calculations prior to actually driving the path. This method MUST be called after `loadPath()` and before the first call to `calculateNextMovements()` or else the path driving may not work.

```
public Trajectory calculateNextMovements()
```

Calculates the target velocity of the robot and the magnitude of any turn that it needs to take as it drives through a path, based on its position and the path being driven.

From this point, teams with tank-style robots can call `getTankTrajectory()` to perform additional calculations to get the target velocity of their robot's left and right wheels. Those velocities can then be set as setpoints for motor controller PIDs. However, in order to set the right target velocity for the specific motor controller your robot uses, be sure to use the `TankTrajectory`'s `convertTime()` method. See the documentation on that method for more information.

Returns: The trajectory for the robot to take.

```
public int getTotalPoints()
```

Returns the total number of points in the path that the robot is emulating.

Returns: The number of points in the current path.

```
public int getCurrentPoint()
```

Returns the index of the point in the path that the robot is trying to achieve.

Returns: Index of the current point.

```
public boolean pathFinished()
```

Returns whether or not the Hyperdrive is done driving the robot through a Path.

Returns: `true` if the robot is done driving through its path, and `false` otherwise.

```
public void finishPath()
```

Frees up resources used while emulating the Path. Also sends the robot's actual Path to PathVisualizer for viewing. If PathVisualizer is connected and the "Live" option is enabled, then the Path will appear on the screen.

```
public double toMotorUnits(
     double commonUnitMeasurement)
```

Converts a measurement made in common units (inches, meters, etc) to motor units (ticks, revolutions, etc) using the motorUnitsPerUnit measurement provided to the Hyperdrive's constructor.

Arguments	
`double commonUnitMeasurement`	Measurement, in common units.

Returns: Equivalent length, in motor units.

```
public double toCommonUnits(
     double motorUnitMeasurement)
```

Converts a measurement made in motor units (ticks, revolution, etc) to common units (inches, meters, etc) using the motorUnitsPerUnit measurement provided to the Hyperdrive's constructor.

Arguments	
`double motorUnitMeasurement`	Measurement, in motor units

Returns: Equivalent length, in common units.

```
public class TankGyro
```

An experimental utility that calculates the theoretical heading angle of a tank-style robot using the positions of the motors. In its current state of development, this class has shown to be reliable for constant velocity and slightly varied acceleration cases, but drifts as far as 20 degrees in 5 seconds when motor speed and robot direction is frequently varied. Therefore, this class should not replace an actual gyro on competition robots but can be used for tests and experiments.

This class' functionality is dependent on the positions reported by the drivetrain motor encoders. That being said, in order for the heading reported by this class to stay accurate, the wheels cannot drift. When the robot drifts or slides, the positions reported by the encoders can no longer truly calculate the linear displacement of that wheel from its starting position. When that happens, this class cannot accurately calculate the robot's angular displacement from its starting heading.

```
public TankGyro(
     final double wheelBaseWidth,
     final double motorUnitsPerUnit,
     boolean inverted,
     double leftPosition,
     double rightPosition)
```

Creates a new `TankGyro`. This is the "main" constructor.

Arguments	
`final double wheelBaseWidth`	The length between the left and right wheels of the robot, measured parallel to the width of the robot. This measurement MUST be taken in the same common length units (inches, meters, etc) that are specified in the next parameter, `motorUnitsPerUnit`. For example, if that value is in motor units per inch, then this value must be in inches. If that value is in motor units per meter, then this value must be in meters.
`final double motorUnitsPerUnit`	The number of motor units per one length unit. This value should be the same as the one supplied to the `Hyperdrive` constructor.

`boolean inverted`	`true` if the gyro should be inverted, `false` otherwise.
`double leftPosition`	The current position of the left motors in motor units. This parameter can be useful if the motor positions are nonzero but you want this gyro to initiate to a 0 heading.
`double rightPosition`	The current position of the right motors in motor units. This parameter can be useful if the motor positions are nonzero but you want this gyro to initiate to a 0 heading.

```
public TankGyro(
     final double wheelBaseWidth,
     final double motorUnitsPerUnit,
     boolean inverted)
```

Creates a new `TankGyro`, initializing the left and right positions as 0.

Arguments	
`final double wheelBaseWidth`	The length between the left and right wheels of the robot, measured parallel to the width of the robot. See the “main” constructor for more information.
`final double motorUnitsPerUnit`	The number of motor units per one length unit. This value should be the same as the one supplied to the `Hyperdrive` constructor.
`boolean inverted`	`true` if the gyro should be inverted, `false` otherwise.

```
public TankGyro(
     final double wheelBaseWidth,
     final double motorUnitsPerUnit)
```

Creates a new, un-inverted `TankGyro`, initializing the left and right positions to be 0.

Arguments	
`final double wheelBaseWidth`	The number of motor units per one length unit. This value should be the same as the one supplied to the `Hyperdrive` constructor.
`final double motorUnitsPerUnit`	`true` if the gyro should be inverted, `false` otherwise.

```
public double getHeading()
```

Returns the theoretical heading of the robot, in degrees, as calculated using the drivetrain's motor positions. This method's return value will only change when `update()` is called.

Returns: The current theoretical heading of the robot in degrees.

```
public void setInverted(
     boolean inverted)
```

Inverts or un-inverts the gyro.

Arguments	
`boolean inverted`	`true` if the gyro should be inverted, `false` otherwise.

```
public void update(
     double leftPosition,
     double rightPosition)
```

Updates the heading of the robot based on the positions of the drivetrain motors. This method should be called in one of the robot's `periodic()` methods. If this object is being used with the rest of Hyperdrive, then this method would be well-placed before `Hyperdrive`'s `update()` method. The return value of `getHeading()` will only change when this method is called.

Arguments	
`double leftPosition`	The position of the drivetrain's left wheels in motor units.
`double rightPosition`	The position of the drivetrain's right wheels in motor units.

```
public void setHeading(
     double heading)
```

Sets the heading of the gyro.

Arguments	
`double heading`	The new heading of the gyro, in degrees.

```
public void zeroHeading(
     boolean waitForEncoders)
```

Sets the heading of the gyro to 0. This method offers the option to wait for the drivetrain encoders to read out a positional value of 0 before it zeros for the same reason that `Hyperdrive`'s `zeroPositionAndHeading()` method does. For more about what that reason is, see that method's documentation.

Arguments	
`boolean waitForEncoders`	This should be `true` if the motor controllers are also being zeroed. This will cause the gyro to wait until the motor encoders read out a positional value of 0 before actually setting the heading to 0.

```
public void zeroHeading()
```

Zeros the heading of the gyro, but does not wait for the drivetrain encoders to be zero beforehand. Calling this method is equivalent to the call `zeroHeading(false)`.

```
public interface IEmulateParams
```

An interface for providing parameters that Hyperdrive should use while driving the robot through a path. These parameters include a turn inhibitor, the maximum and minimum speeds, a positional correction inhibitor, a minimum distance off of the path for positional correction to happen, the coefficient of static friction between the drive surface and the robot’s tires, the number of points ahead of the robot that Hyperdrive should look ahead, and the number of points that Hyperdrive should use when doing its calculations.

```
public double getOverturn()
```

Returns the number of degrees that the robot must turn at any point during the path will be multiplied by this value to get the final turn amount. If one finds that the robot consistently does not turn enough, this value should be increased, likewise, if the robot is turning too much, this value should be decreased.

Returns: The overturn value to use while driving through the path.

```
public double getMinimumSpeed()
```

Returns the minimum speed in units per SECOND that the robot can be going at any time while driving through a path. No matter the tightness, the robot will always take any turn with a speed greater than or equal to this value. This value is measured in units per second.

Returns: The minimum speed of the robot while driving a path.

```
public double getMaximumSpeed()
```

Returns the maximum speed in units per SECOND that the robot can be going at any given time while driving through a path. The robot will always try to achieve this speed on straightaways. This value must be greater than the minimum speed and should be less than the robot’s actual maximum speed.

Returns: The maximum speed of the robot while driving a path.

```
public double getPositionalCorrectionInhibitor()
```

Returns the inhibitor for the robot's position correction. The greater this value, the more the robot will try to remain exactly on the path that it is driving. Setting this value too low could cause the robot to veer off of its path. However, setting this value too high could cause the robot to oscillate from left to right which would affect the speed and visual appeal of the robot as it drives through the path.

Returns: The positional correction inhibitor to use.

```
public double getPositionalCorrectionDistance()
```

Returns the minimum distance from the path that the robot must have in order to start correcting its position. This value should be measured in the unit that was specified for the `Hyperdrive` constructor.

Returns: Minimum distance from the path required for positional correction to take effect.

```
public double getCoefficientOfStaticFriction()
```

Returns the approximate coefficient of static friction between the robot's wheels and the surface that the robot is driving on. This value is used to determine the speed at which the robot takes its turns and can be used as a way to tune that speed.

Returns: The coefficient of static friction between the robot's wheels and the surface that the robot is driving on.

```
public int getPointSkipCount()
```

Returns the number of points directly ahead of the robot that will be skipped by the path following algorithm. The points immediately following the skipped points will then be used to determine how much the robot needs to turn and how fast it should make that turn. Skipping points ensures that the angle between the robot's trajectory and the path is true and not being made larger by a small distance between the robot and the next point.

Returns: The number of points to skip while making calculations for path driving.

```
public int getImmediatePathSize()
```

Returns the number of points immediately ahead of the robot (after skipped points) that will be used to calculate the robot’s target trajectory. A larger value will make the robot drive smoother and take turns sooner, while a small value will make the robot’s driving rather choppy, and the robot will take turns later.

Returns: The number of points ahead of the robot, after skipped points, to use to determine future turns.

```
public class ConstantEmulationParams implements IEmulateParams
```

A set of parameters that Hyperdrive will use to drive the robot through a path.

```
public ConstantEmulationParams(
     double overturn,
     double minimumSpeed,
     double maximumSpeed,
     double positionalCorrectionInhibitor,
     double positionalCorrectionDistance,
     double coefficientOfStaticFriction,
     int pointSkipCount,
     int immediatePathSize)
```

Creates a new `ConstantEmulationParams`. For more information about any of the arguments for this constructor, see the `IEmulateParams` documentation.

Arguments	
`double overturn`	Increases or decreases the magnitude of future turns by multiplying the turn by this amount.
`double minimumSpeed`	The minimum speed of the robot at any time while driving the path.
`double maximumSpeed`	The maximum speed of the robot at any time while driving the path.
`double positionalCorrectionInhibitor`	Inhibits the robot from performing positional corrections while driving the path.
`double positionalCorrectionDistance`	The minimum distance from the path that the robot can have before positional correction takes effect
`double coefficientOfStaticFriction`	The coefficient of static friction between the robot's wheels and the surface that it is driving on.
`int pointSkipCount`	The number of points ahead of the robot that Hyperdrive should ignore.

`int immediatePathSize`	The number of points ahead of the robot, after the skipped ones, to use to determine the robot's target trajectory.

```
public static ConstantEmulationParams getDefaults(
     Units.LENGTH units)
```

Returns the default values for robots to use in whatever unit is supplied.

Arguments	
`Units.LENGTH units`	The units of length that are being used.

Returns: A `ConstantEmulationParams` containing the default parameters for path driving.

****This class has other methods that are inherited from `IEmulateParams`. See that class' documentation for more information.**

```
public class PreferenceEmulationParams implements IEmulateParams
```

A class that uses the Preferences table on the dashboard to determine Hyperdrives path emulation parameters. Use this class as an `IEmulateParams` to tune any paths that may need to be tuned. When tuning is done, this class can be replaced by a `ConstantEmulationParams`.

```
public PreferenceEmulationParams(
     Units.LENGTH units)
```

Creates a new `PreferenceEmulationParams`.

Arguments	
`Units.LENGTH units`	The units of length to use. This should be the same unit that you provided to the `Hyperdrive` constructor.

****This class has other methods that are inherited from `IEmulateParams`. See that class' documentation for more information.**

public class Units

A class that describes units of measurement of length, time, and force.

public static enum LENGTH

Describes units of length or distance.

Value	Description
`INCHES`	Specifies the units of inches.
`FEET`	Specifies the units of feet.
`YARDS`	Specifies the units of years.
`CENTIMETERS`	Specifies the units of centimeters.
`METERS`	Specifies the units of meters.

public static enum TIME

Describes units of time.

Value	Description
`DECASECONDS`	Specifies the unit of decasconds (tenths of seconds). This is the common time unit for CTRE motor controllers.
`SECONDS`	Specifies the unit of seconds. This is regular, human-readable time.
`MINUTES`	Specifies the unit of minutes. This is the common time unit for REV motor controllers.

public static enum FORCE

Describes units of force or weight.

Value	Description
`NEWTON`	Specifies the unit of Newtons.
`POUND`	Specifies the unit of pounds.
`KILOGRAM_FORCE`	Specifies the unit of kilograms used as a unit of force, not mass.

`public class Trajectory`

Represents the desired path of movement of the robot at any point while driving through a path. This class is the main trajectory class which holds only the crude turn and speed values. This cannot be used to set the target velocity of any drivetrain motors without further calculations. Users should use this class to calculate target velocities of their motors by converting instances of this class to more specific forms by using methods such as the `getTankTrajectory()` method. As support for different types of drivetrains gets added, more methods similar to this one will be added to this class.

```
public Trajectory(
      double velocity,
      double distance,
      double turn,
      double maxSpeed,
      double minSpeed,
      final double motorUnitsPerUnit)
```

Creates a new `Trajectory`, defining that the robot should drive at the given velocity for a certain distance ahead while turning to a certain angular displacement.

Arguments	
`double velocity`	The target velocity of the robot.
`double distance`	The length of the path immediately ahead of the robot, after skipped points.
`double turn`	The magnitude of the turn immediately ahead of the robot, in radians.
`double maxSpeed`	The maximum speed of the robot, in length units per second, as it drives its path.
`double minSpeed`	The minimum speed of the robot, in length units per second, as it drives its path.
`final double motorUnitsPerUnit`	The number of motor units per one length unit. See the main `Hyperdrive` constructor for more information.

```
public double getVelocity()
```

Returns the target velocity of the robot as it goes through this trajectory.

Returns: Target velocity of the robot.

```
public double getDistance()
```

Returns the length of the path immediately ahead of the robot, after skipped points.

Returns: Length of the trajectory.

```
public double getTurn()
```

Returns the angular displacement, in radians, of the robot as it moves through the trajectory.

Returns: Angular displacement of robot over trajectory, in radians.

```
public double getMaxSpeed()
```

Returns the maximum speed of the robot as it drives through its path.

Returns: maximum speed, in length units per second.

```
public double getMinSpeed()
```

Returns the minimum speed of the robot as it drives through its path.

Returns: Minimum speed, in length units per second.

```
public TankTrajectory getTankTrajectory(
     double wheelBaseWidth)
```

Returns the trajectory information in a format that can drive tank-style robots.

Arguments	
`double wheelBaseWidth`	The distance between the left and right wheels of the robot, measured parallel to the width of the robot.

Returns: A `TankTrajectory`, which contains the left and right wheels velocities required to keep the robot on its path.

```
public final double getMotorUnitsPerUnit()
```

Returns the motor units per unit value that was passed into the constructor.

Returns: Motor units per unit scalar.

```
public class TankTrajectory extends Trajectory
```

Represents the trajectory of a tank-style robot as it moves through a path. This class provides programmers with the target velocities of the left and right motors.

```
public TankTrajectory(
     Trajectory trajectory,
     double wheelBaseWidth)
```

Creates a new `TankTrajectory`.

Arguments	
`Trajectory trajectory`	The `Trajectory` to calculate from.
`double wheelBaseWidth`	The distance between the left and right wheels, measured parallel to the width of the robot.

```
public TankTrajectory convertTime(
     Units.TIME desired)
```

Converts the wheel velocities to a specified unit of time. The velocities are originally in units per second. This method returns a `TankTrajectory` so that method calls that modify the trajectory can be chained together.

This method MUST be called if using a CTRE, REV, or other motor controller whos velocities are reported in something other than units per second.

Arguments	
`Units.TIME desired`	The units of time to convert the velocities to.

Returns: The `TankTrajectory` after the modification is complete.

```
public TankTrajectory invertTurn()
```

Inverts the turns that the robot takes during the path. Call this method if the robot turns the wrong direction while driving.

Returns: The `TankTrajectory` after the modification is complete.

```
public TankTrajectory invertDirection()
```

Inverts the direction that the robot drives. Call this method if the robot drives the wrong direction when driving through a path.

Returns: The `TankTrajectory` after the modification is complete.

```
public double getRawLeftVelocity()
```

Returns the value of the left velocity in its original units per second.

Returns: Target left velocity in units per second.

```
public double getRawRightVelocity()
```

Returns the value of the right velocity in its original units per second.

Returns: Target right velocity in units per second.

```
public double getLeftVelocity()
```

Returns the target velocity of the left drive motors needed to stay on course with the path currently being driven. This value is returned in the raw motor controller units and can be directly set as a setpoint for a velocity PID running on the motor controllers. In order for the unit to be correct, however, the `convertTime()` method must be called with the correct unit of time (decaseconds for CTRE controllers, minutes for REV controllers).

Returns: Target velocity of left motors in proper motor velocity units.

```
public double getRightVelocity()
```

Returns the target velocity of the right drive motors needed to stay on course with the path currently being driven. This value is returned in the raw motor controller units and can be directly set as a setpoint for a velocity PID running on the motor controllers. In order for the unit to be correct, however, the `convertTime()` method must be called with the correct unit of time (decaseconds for CTRE controllers, minutes for REV controllers).

Returns: Target velocity of right motors in proper motor velocity units.

```
public class HyperdriveUtil
```

A collection of utilities that Hyperdrive relies on to function.

```
public static double getAndSetDouble(
     String key,
     double backup)
```

Retrieves a value from the Preferences table on the dashboard. If that value does not exist, it is set to the backup value given.

Arguments	
`String key`	The name of the value to get
`double backup`	The value to set if the value does not exist

Returns: The value of the entry at `key`.

```
public static double roundTo(
     double value,
     int places)
```

Rounds a value to *n* places.

Arguments	
`double value`	Original Value
`int places`	Number of places after the decimal point to round to

Returns: Rounded value.

```
public static double convertDistance(
     double value,
     Units.LENGTH original,
     Units.LENGTH desired)
```

Converts a measurement between units of length.

Arguments	
`double value`	The value of the measurement to convert.
`Units.LENGTH original`	The current unit of the measurement.
`Units.LENGTH desired`	The unit to convert the measurement to.

Returns: The value of the measurement in the desired unit.

```
public static double convertTime(
     double value,
     Units.TIME original,
     Units.TIME desired)
```

Converts a value between units of time.

Arguments	
`double value`	The value to convert between.
`Units.TIME original`	The original units of the value.
`Units.TIME desired`	The units to convert the value to.

Returns: The equivalent value in the desired unit.

```
public static double convertForce(
     double value,
     Units.FORCE original,
     Units.FORCE desired)
```

Converts a force value between units.

Arguments	
`double value`	The value of the original force.
`Units.FORCE original`	The original units of the force.
`Units.FORCE desired`	The units to convert the value to.

Returns: The equivalent value in the desired unit.

```
public static double massKGFromWeight(
     double weight,
     Units.FORCE units)
```

Calculates the mass of an object in kilograms from its weight.

Arguments	
`double weight`	The weight of the object, in some unit.
`Units.FORCE units`	The units of `weight`.

Returns: The mass of the object, in kilograms.

```
public static double weightFromMassKG(
     double mass,
     Units.FORCE desiredWeightUnit)
```

Calculates the weight of an object based on its mass.

Arguments	
`double mass`	The mass of the object, in kilograms.
`Units.FORCE desiredWeightUnit`	The unit of weight to return the result in.

Returns: The weight of the object in the desired unit.

```
public static double closestToZero(
     double[] set)
```

Returns the number in the set that is closest to zero.

Arguments	
`double[] set`	An array of `double`s to parse.

Returns: The value of the number in the set with the least absolute value.

```
public static double getAngleToHeading(
     double angle,
     double heading)
```

Gets the angle that the robot needs to turn through to achieve a heading.

Arguments	
`double angle`	The current angle of the robot, in degrees.
`double heading`	The desired heading to be achieved, in degrees.

Returns: The angle that the robot needs to turn through to have its desired heading, in degrees.

```
public static boolean assertEquals(
     String assertionName,
     Object item1,
     Object item2)
```

Tests the equality of two values, then prints and returns the result. On RioLog, this result will appear as an error reading “Assertion [assertionName] SUCCEEDED/FAILED.” If the test fails, this error will also include the values of the two items. This method can be used to test utility methods in the robot’s Test mode.

Arguments	
`String assertionName`	The informational name of the assertion being made. This will appear on the printout in RioLog.
`Object item1`	The first item to test.
`Object item2`	The second item to test.

Returns: `true` if `item1 == item2`, `false` otherwise.

```
public static String[] getFilesInDirectory(
     String directory,
     boolean specifyDirectories)
```

Returns a list of files and, optionally, directories in the RoboRIO's file system.

Arguments	
`String directory`	The directory to search.
`boolean specifyDirectories`	If `true`, the method will add a tag to the end of each string containing the name of a directory to specify it as a directory.

```
public class Path
```

Represents a Path that can be driven by the robot.

```
public Path(
     String file)
```

Creates a new `Path`.

Arguments	
`String file`	The absolute file path of the file that contains the path data.

```
public Path(
     Point2D[] points)
```

Creates a new Path.

Arguments	
`Point2D[] points`	An array of points that compose the path.

```
public Point2D[] getPoints()
```

Returns the `Path`'s points.

Returns: An array of `Point2D` objects.

```
public boolean isValid()
```

Returns `true` if the `Path` is valid and driveable, `false` otherwise.

Returns: The validity of the path.

```
public String toString()
```

Converts the object into a user and computer-readable `String`.

Returns: `String` representation of the path.

`public class Point2D`

Simple class structure that represents a point in the XY plane.

```
public Point2D(
     double x,
     double y,
     double heading)
```

Creates a new `Point2D`.

Arguments	
`double x`	The X-coordinate of the point.
`double y`	The Y-coordinate of the point.
`double heading`	The heading of the robot at the point.

```
public double getX()
```

Returns the X-coordinate of the point.

Returns: X-coordinate of the point.

```
public double getY()
```

Returns the Y-coordinate of the point.

Returns: Y-coordinate of the point.

```
public double getHeading()
```

Returns the heading of the robot at the point.

Returns: Heading of the robot at the point.

```
public double getDistanceFrom(
     Point2D point)
```

Calculates the distance from a certain point.

Arguments	
`Point2D point`	The point to measure distance to.

Returns: The distance between this point and the given point.

```
public double getHeadingTo(
     Point2D point)
```

Returns the angle of the line between this point and the given point.

Arguments	
`Point2D point`	The point to calculate heading to.

Returns: The angle of the line between this point and the given point. This is the heading that the robot would need to drive in a straight line from this point to the given point.

```
public String toString()
```

Returns a representation of the point in a `String` format.

Returns: Human and computer-readable representation of the point.

```
public static Point2D fromString(
     String input)
```

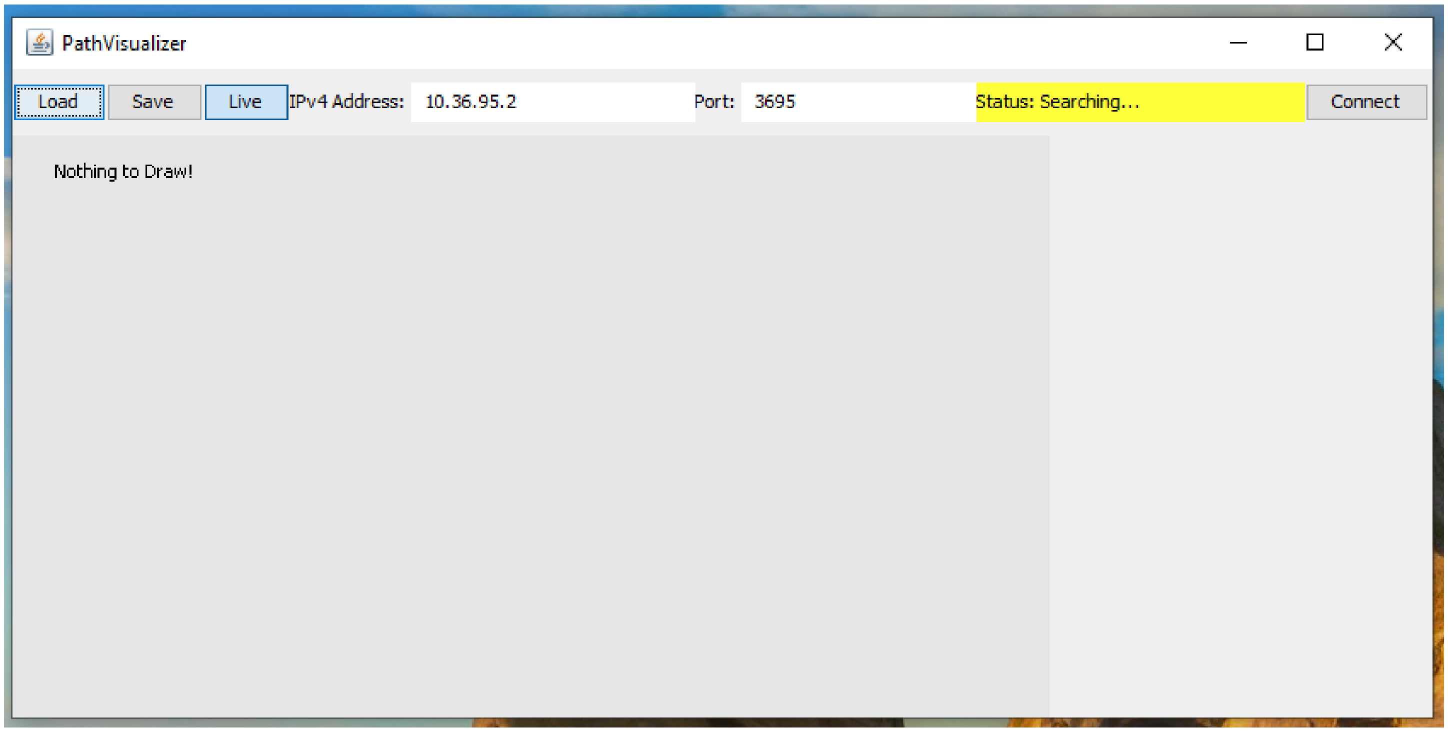
Creates a new `Point2D` using the given input string.
The string must be formatted as such: “[x],[y],[heading]”. This is the same format that is outputted by `toString()`.

If the input string is invalid, then `null` is returned.

Arguments	
`String input`	Formatted string representing a `Point2D` object.

Returns: The `Point2D` represented by the given string, or `null` if the string is invalid.

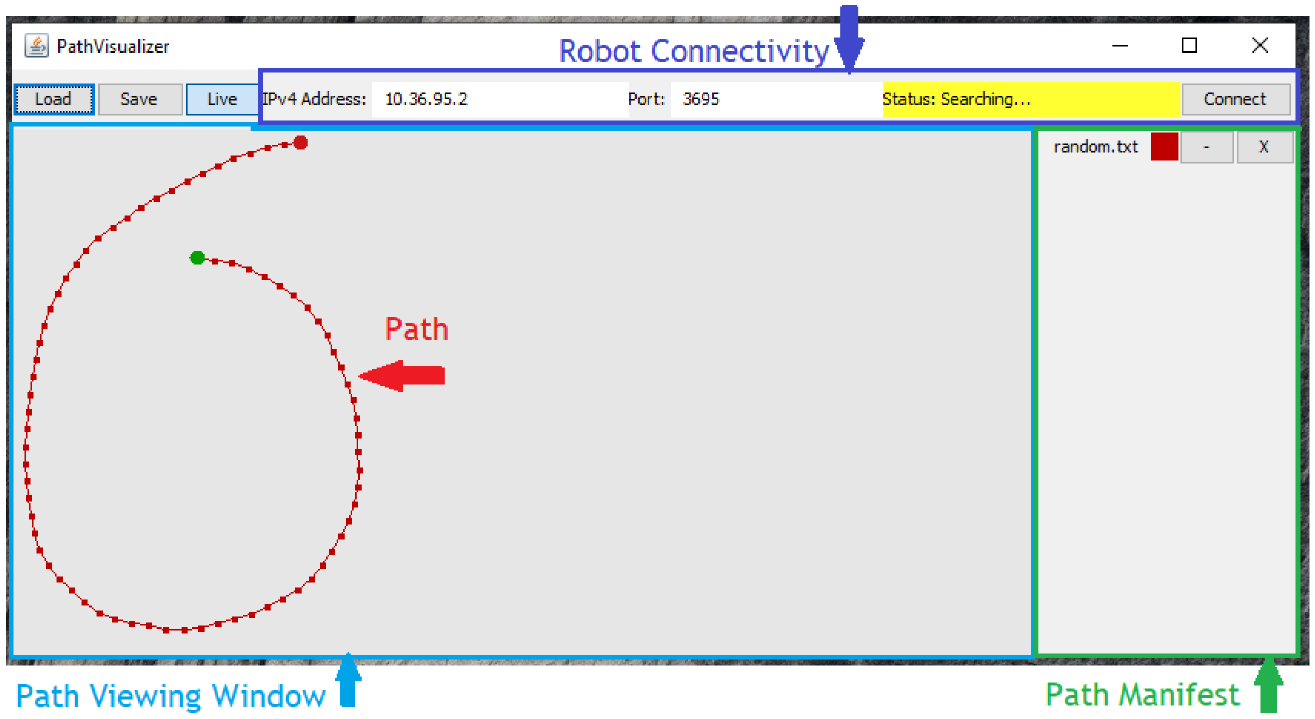
PathVisualizer



This chapter will cover how to use PathVisualizer with your robot. PathVisualizer is a Java application for viewing paths generated by Hyperdrive. With this tool, it is easy to identify and fix problems with your robot's autonomous driving. PathVisualizer can render paths to scale, display the robot's current position relative to those paths, and read paths from and save paths to the computer, as well as the robot's file system.

In order for PathVisualizer to successfully communicate with the robot, an instance of the `Hyperdrive` class must be present in the robot code and its `update()` method must be called in one of the robot's `periodic()` methods. PathVisualizer's robot-related functions will not work on robots where these conditions are not met.

Anatomy



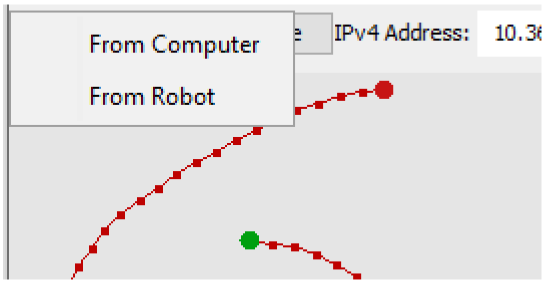
The above image shows the general layout of PathVisualizer. The majority of the window is taken up by the Path Viewing Window, where all paths and the robot's position on the field (if the robot is connected) will be rendered. To the right of the Path Viewing Window is the Path Manifest, where the names of all paths and their respective colors are shown. Here, the user can press the [-] button to hide the path and the [X] button to delete the path. Above the manifest is the Robot Connectivity area where the user can specify the IP Address and port to use to connect to the robot. This area also features a connection status indicator, which will read out the status of the connection to the robot. Finally, to the left of that region are the "Load", "Save", and "Live" buttons. Using these buttons, the user can read a path from the computer or robot, save paths to the computer or robot, and toggle on or off the ability of PathVisualizer to receive live path data from the robot.

Viewing Paths

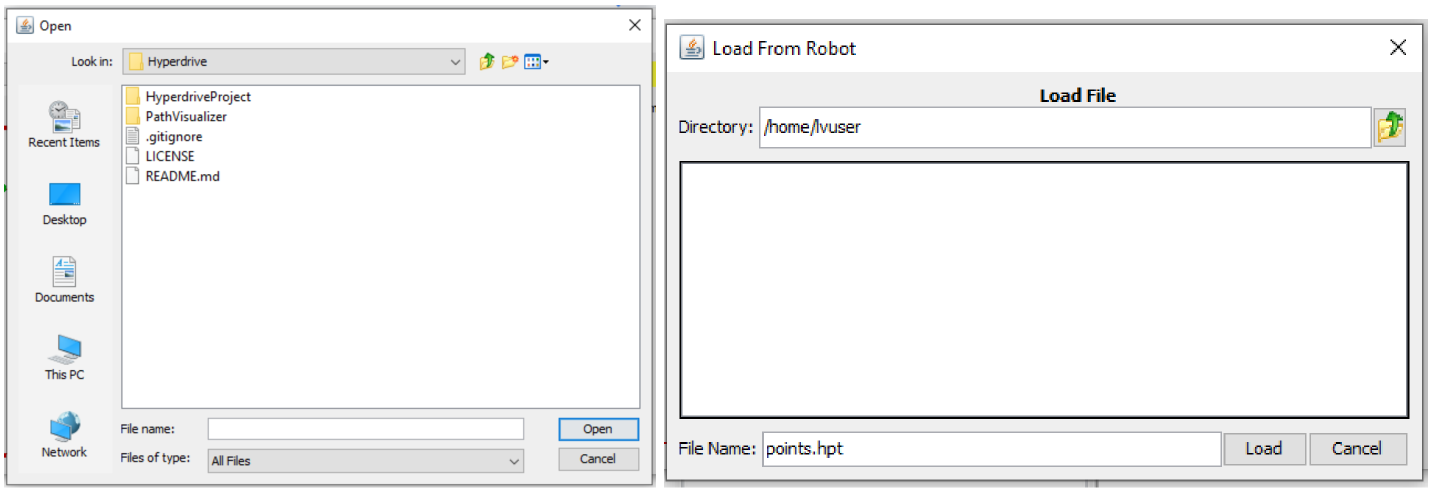
PathVisualizer's main function, which gives it its name, is to view paths that the robot can drive or has already driven. This section will cover a few ways to do that.

- *Using the "Load" button*

  PathVisualizer can load paths from either the computer or the robot's file system. To do either of these, click on the Load button. Then, use the context menu to select whether or not to load from the computer or from the robot.



  A file dialog should pop up, prompting you to select the file to load.



Directory: /home/lvuser

File Name: points.hpt

Computer File Dialog (left), and Robot File Dialog (right)

  After selecting the file to load, PathVisualizer should load that file and the path should become visible on the Path Viewing Window.

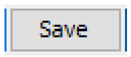
- *From Live Robot Data*

  PathVisualizer also has the ability to automatically load paths that the robot sends. However, in order for PathVisualizer to accept the paths being sent, the Live option must be enabled. When live data is enabled, PathVisualizer will display the following:
    - Newly recorded paths every time the robot finishes recording a new path.

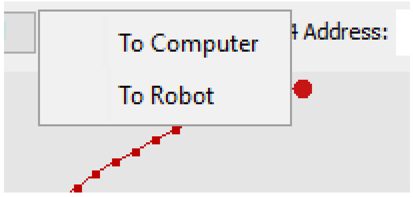
- Newly driven paths every time the robot finishes driving a path.
- Other paths sent with `Hyperdrive`'s `sendPath()` method.

Saving Paths

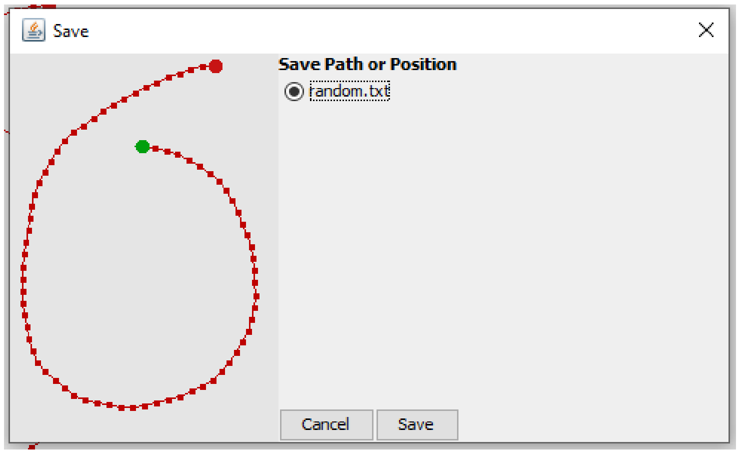
PathVisualizer also offers an option to save paths to the computer or the robot.

In PathVisualizer, click the 

button. Using the context menu, select whether you wish to save a path to the computer or to the robot.



Use the dialog to select which path to save.

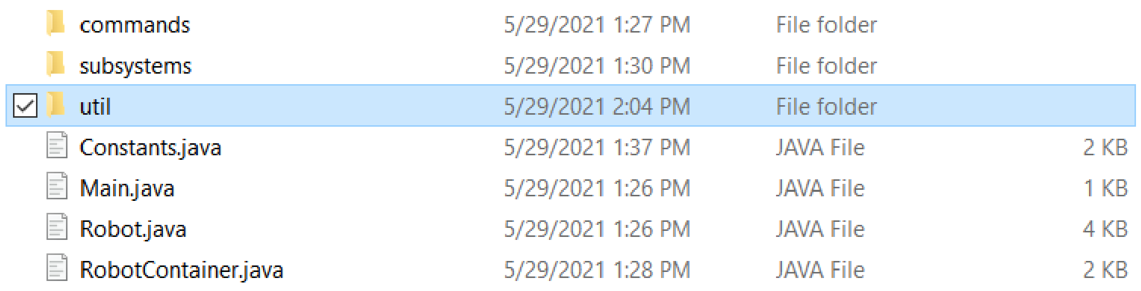


Finally, use the file dialogs to choose where to save the path or position.

Example Project

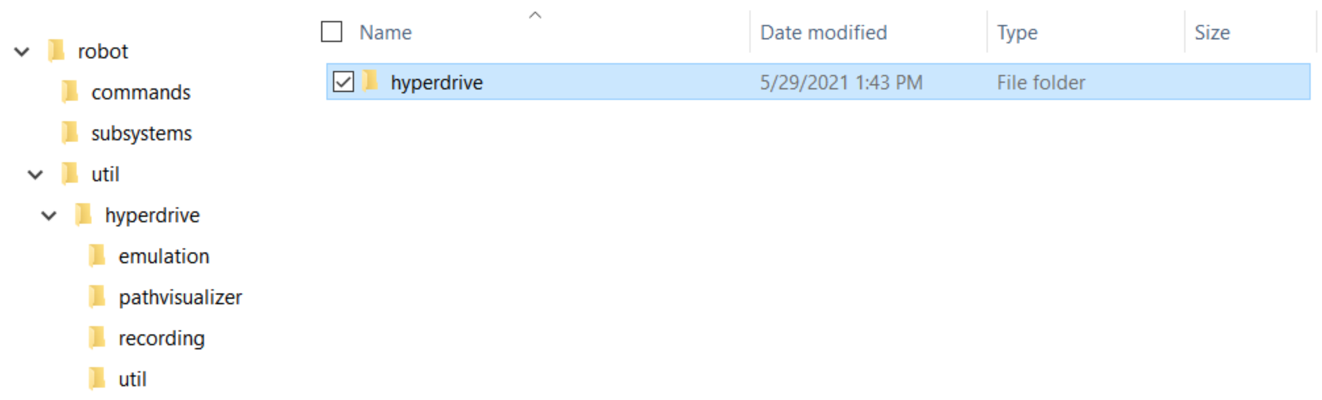
The following example project will guide you through how to add autonomous driving capabilities to your robot using Hyperdrive. In this example, we will add autonomous driving abilities to Foximus Prime's 2019 competition robot. This robot uses 4 NEO motors (2 on each side) which are controlled by Spark MAX motor controllers, 2-stage shifting gearboxes (though for this example, the higher gear will be used), and 6 inch hi-grip wheels on its drivetrain. It also uses a Nav-X gyro for navigation.

To start, create a new WPILib project. We will add Hyperdrive to this project right away. To do this, create a folder called “util” in the base of the project (where Robot.java is located).



commands	5/29/2021 1:27 PM	File folder	
subsystems	5/29/2021 1:30 PM	File folder	
util	5/29/2021 2:04 PM	File folder	
Constants.java	5/29/2021 1:37 PM	JAVA File	2 KB
Main.java	5/29/2021 1:26 PM	JAVA File	1 KB
Robot.java	5/29/2021 1:26 PM	JAVA File	4 KB
RobotContainer.java	5/29/2021 1:28 PM	JAVA File	2 KB

Copy the hyperdrive library folder that you downloaded from the releases (github.com/BTK203/Hyperdrive/releases) and paste that folder into the new util folder.

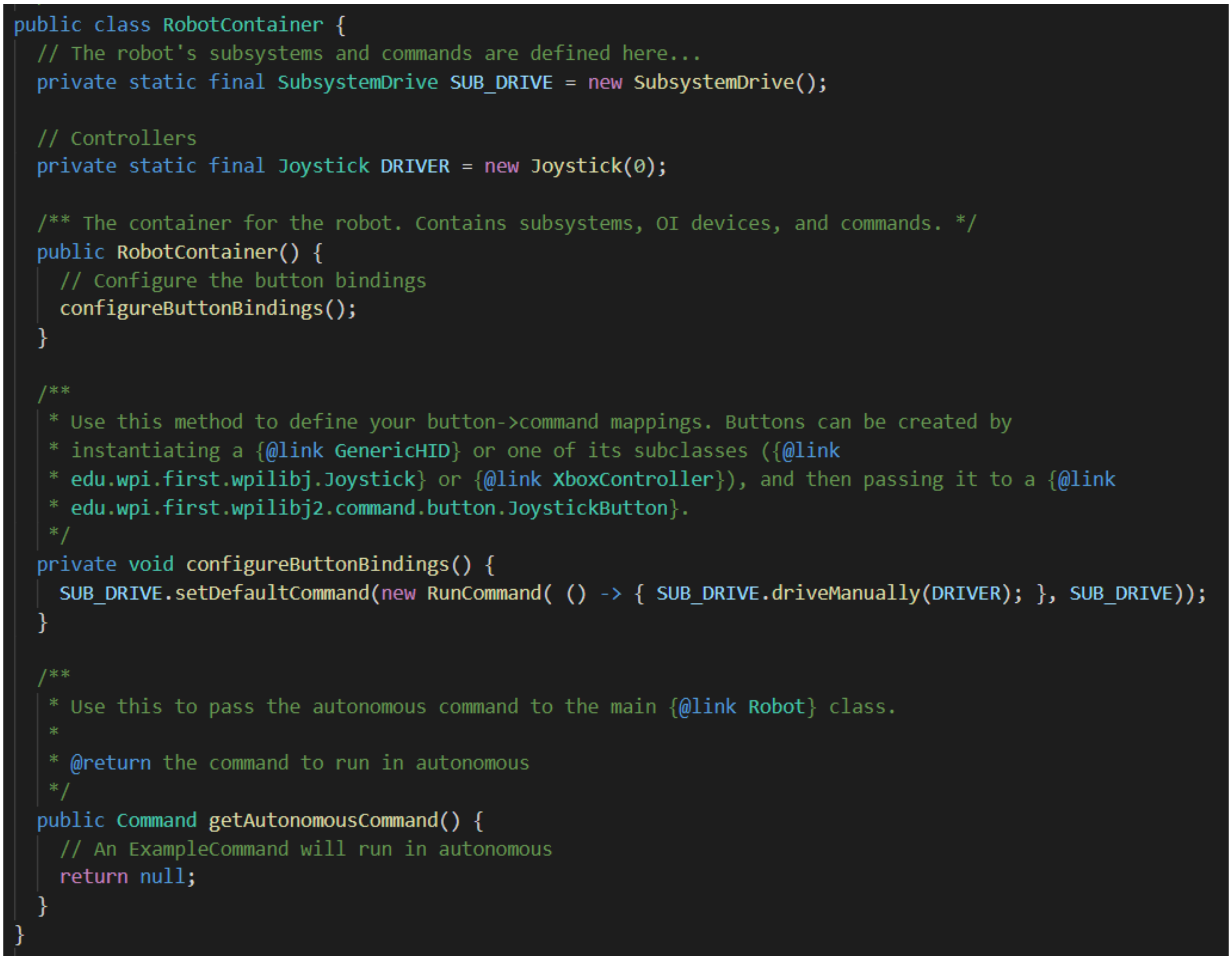


- robot
  - commands
  - subsystems
  - util
    - hyperdrive
      - emulation
      - pathvisualizer
      - recording
      - util

Name	Date modified	Type	Size
hyperdrive	5/29/2021 1:43 PM	File folder	

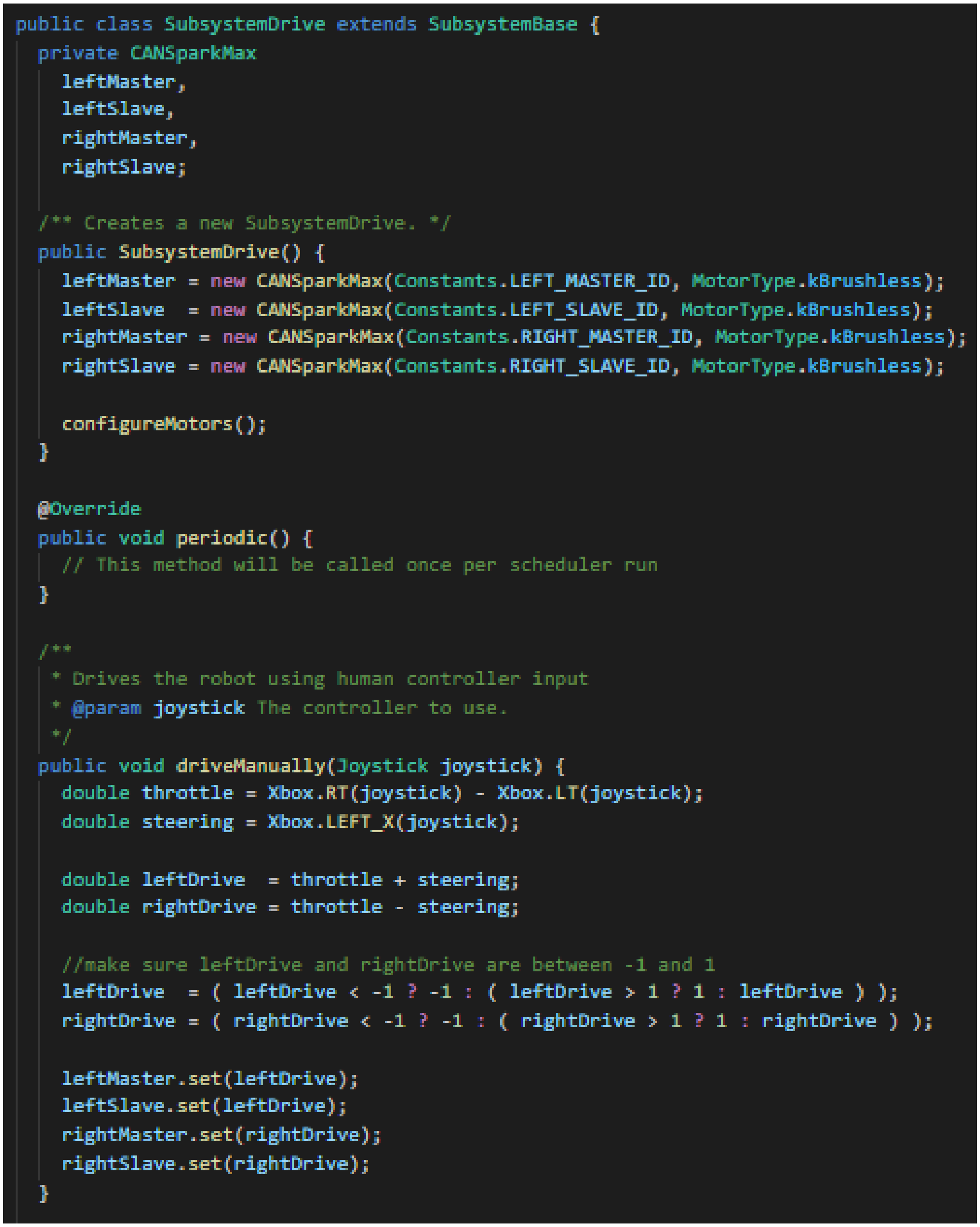
After adding the library to the project, go through some of the files to make sure that the `package` create a basic drivetrain subsystem and add it to the `RobotContainer` class.

RobotContainer.java



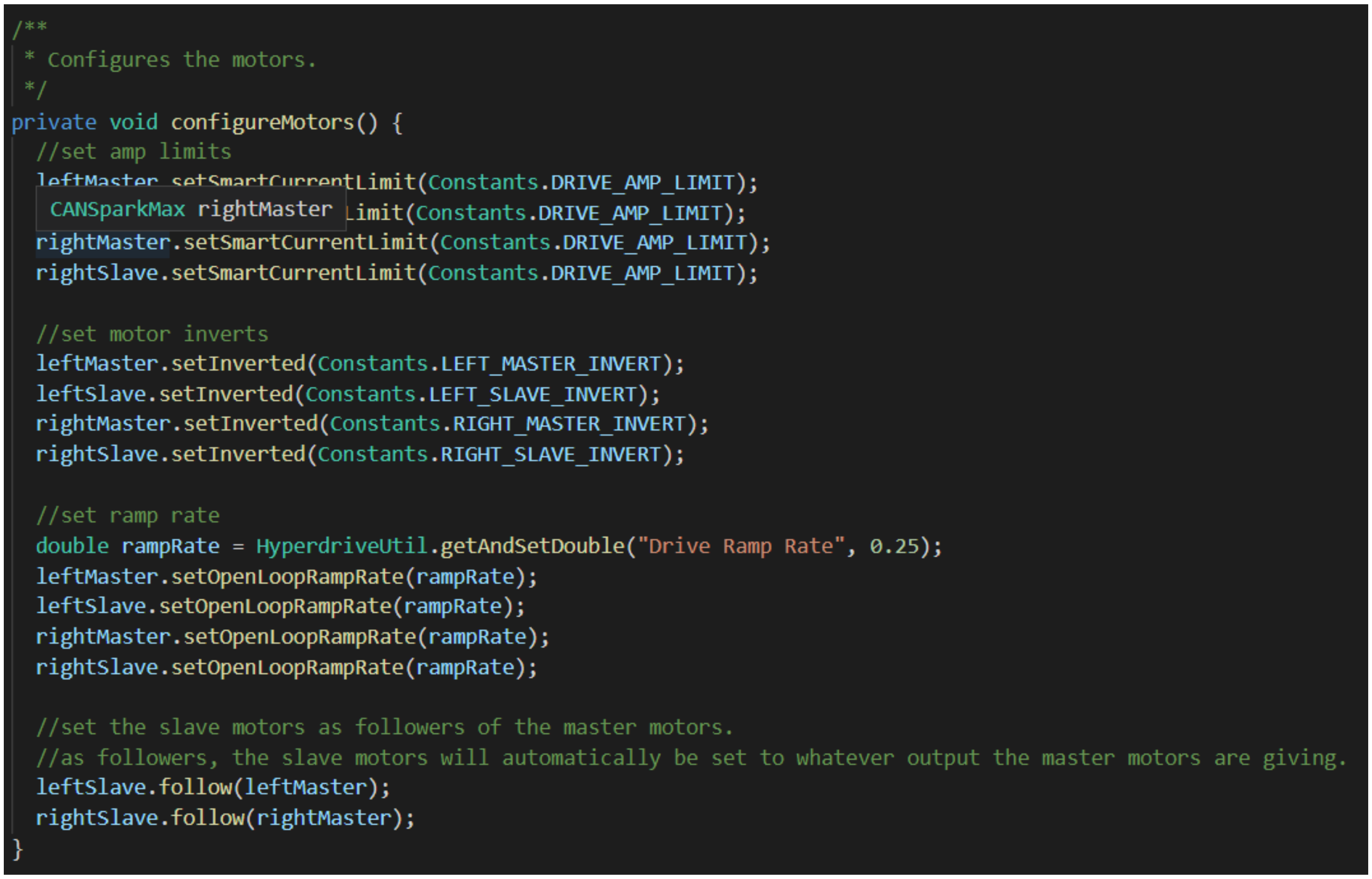
```java
public class RobotContainer {
  // The robot's subsystems and commands are defined here...
  private static final SubsystemDrive SUB_DRIVE = new SubsystemDrive();

  // Controllers
  private static final Joystick DRIVER = new Joystick(0);

  /** The container for the robot. Contains subsystems, OI devices, and commands. */
  public RobotContainer() {
    // Configure the button bindings
    configureButtonBindings();
  }

  /**
   * Use this method to define your button->command mappings. Buttons can be created by
   * instantiating a {@link GenericHID} or one of its subclasses ({@link
   * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it to a {@link
   * edu.wpi.first.wpilibj2.command.button.JoystickButton}.
   */
  private void configureButtonBindings() {
    SUB_DRIVE.setDefaultCommand(new RunCommand( () -> { SUB_DRIVE.driveManually(DRIVER); }, SUB_DRIVE));
  }

  /**
   * Use this to pass the autonomous command to the main {@link Robot} class.
   *
   * @return the command to run in autonomous
   */
  public Command getAutonomousCommand() {
    // An ExampleCommand will run in autonomous
    return null;
  }
}
```

SubsystemDrive.java - (in “subsystems” folder)



```
public class SubsystemDrive extends SubsystemBase {
  private CANSparkMax
    leftMaster,
    leftSlave,
    rightMaster,
    rightSlave;

  /** Creates a new SubsystemDrive. */
  public SubsystemDrive() {
    leftMaster = new CANSparkMax(Constants.LEFT_MASTER_ID, MotorType.kBrushless);
    leftSlave  = new CANSparkMax(Constants.LEFT_SLAVE_ID, MotorType.kBrushless);
    rightMaster = new CANSparkMax(Constants.RIGHT_MASTER_ID, MotorType.kBrushless);
    rightSlave = new CANSparkMax(Constants.RIGHT_SLAVE_ID, MotorType.kBrushless);

    configureMotors();
  }

  @Override
  public void periodic() {
    // This method will be called once per scheduler run
  }

  /**
   * Drives the robot using human controller input
   * @param joystick The controller to use.
   */
  public void driveManually(Joystick joystick) {
    double throttle = Xbox.RT(joystick) - Xbox.LT(joystick);
    double steering = Xbox.LEFT_X(joystick);

    double leftDrive  = throttle + steering;
    double rightDrive = throttle - steering;

    //make sure leftDrive and rightDrive are between -1 and 1
    leftDrive  = ( leftDrive < -1 ? -1 : ( leftDrive > 1 ? 1 : leftDrive ) );
    rightDrive = ( rightDrive < -1 ? -1 : ( rightDrive > 1 ? 1 : rightDrive ) );

    leftMaster.set(leftDrive);
    leftSlave.set(leftDrive);
    rightMaster.set(rightDrive);
    rightSlave.set(rightDrive);
  }
```

SubsystemDrive.java - continued



```java
/**
 * Configures the motors.
 */
private void configureMotors() {
  //set amp limits
  leftMaster.setSmartCurrentLimit(Constants.DRIVE_AMP_LIMIT);
   CANSparkMax rightMaster Limit(Constants.DRIVE_AMP_LIMIT);
  rightMaster.setSmartCurrentLimit(Constants.DRIVE_AMP_LIMIT);
  rightSlave.setSmartCurrentLimit(Constants.DRIVE_AMP_LIMIT);

  //set motor inverts
  leftMaster.setInverted(Constants.LEFT_MASTER_INVERT);
  leftSlave.setInverted(Constants.LEFT_SLAVE_INVERT);
  rightMaster.setInverted(Constants.RIGHT_MASTER_INVERT);
  rightSlave.setInverted(Constants.RIGHT_SLAVE_INVERT);

  //set ramp rate
  double rampRate = HyperdriveUtil.getAndSetDouble("Drive Ramp Rate", 0.25);
  leftMaster.setOpenLoopRampRate(rampRate);
  leftSlave.setOpenLoopRampRate(rampRate);
  rightMaster.setOpenLoopRampRate(rampRate);
  rightSlave.setOpenLoopRampRate(rampRate);

  //set the slave motors as followers of the master motors.
  //as followers, the slave motors will automatically be set to whatever output the master motors are giving.
  leftSlave.follow(leftMaster);
  rightSlave.follow(rightMaster);
}
```

Next, define two new constants in the `Constants` class. One constant will describe the number of motor units (motor rotations, encoder ticks, etc) per 1 length unit (inches, meters, yards, etc). This value is what Hyperdrive will use to figure out how far it has driven based on the position of the motors. The other constant will be the wheelbase width of the robot.

To measure the first value (motor units per length unit), lay a tape measure on the ground. Position the robot at the very beginning of the tape measure. **Zero the drive motor encoders** and use the tape measure to drive the robot a known distance. In the pictures below, a tape measure is being used to drive the robot 10 feet.

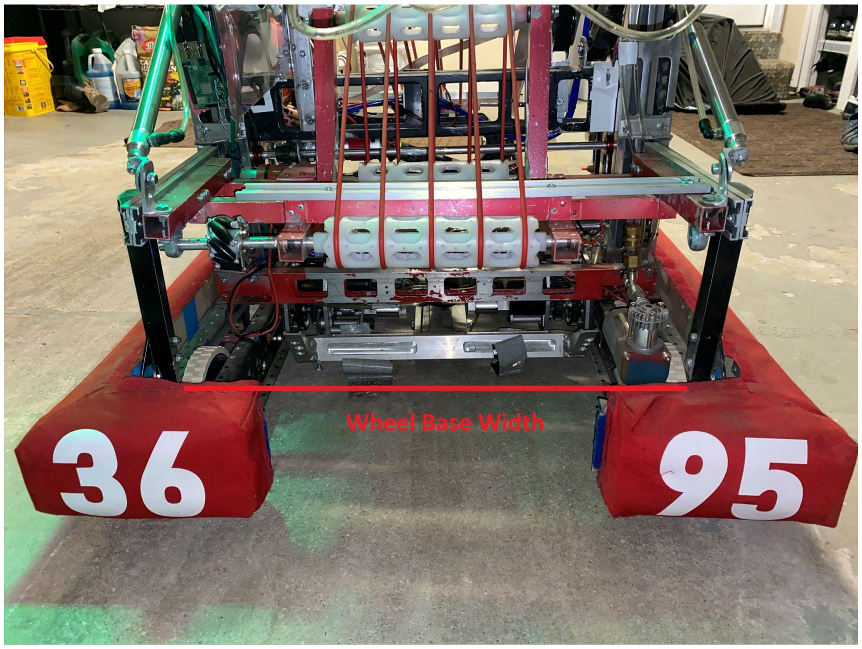
Now, our motor units per unit value can be calculated as such:

$$motor\ units\ per\ unit\ = \frac{motor\ displacement}{distance\ driven}$$

For this example, the robot's drive motors were zeroed and the robot was driven forward 10 feet. After this, the motor position was 49.5 rotations. So, our value was calculated to be

$$\frac{motor\ displacement}{distance\ driven} = \frac{49.44\ rotations}{120\ inches} = 0.412\ rotations\ per\ inch$$

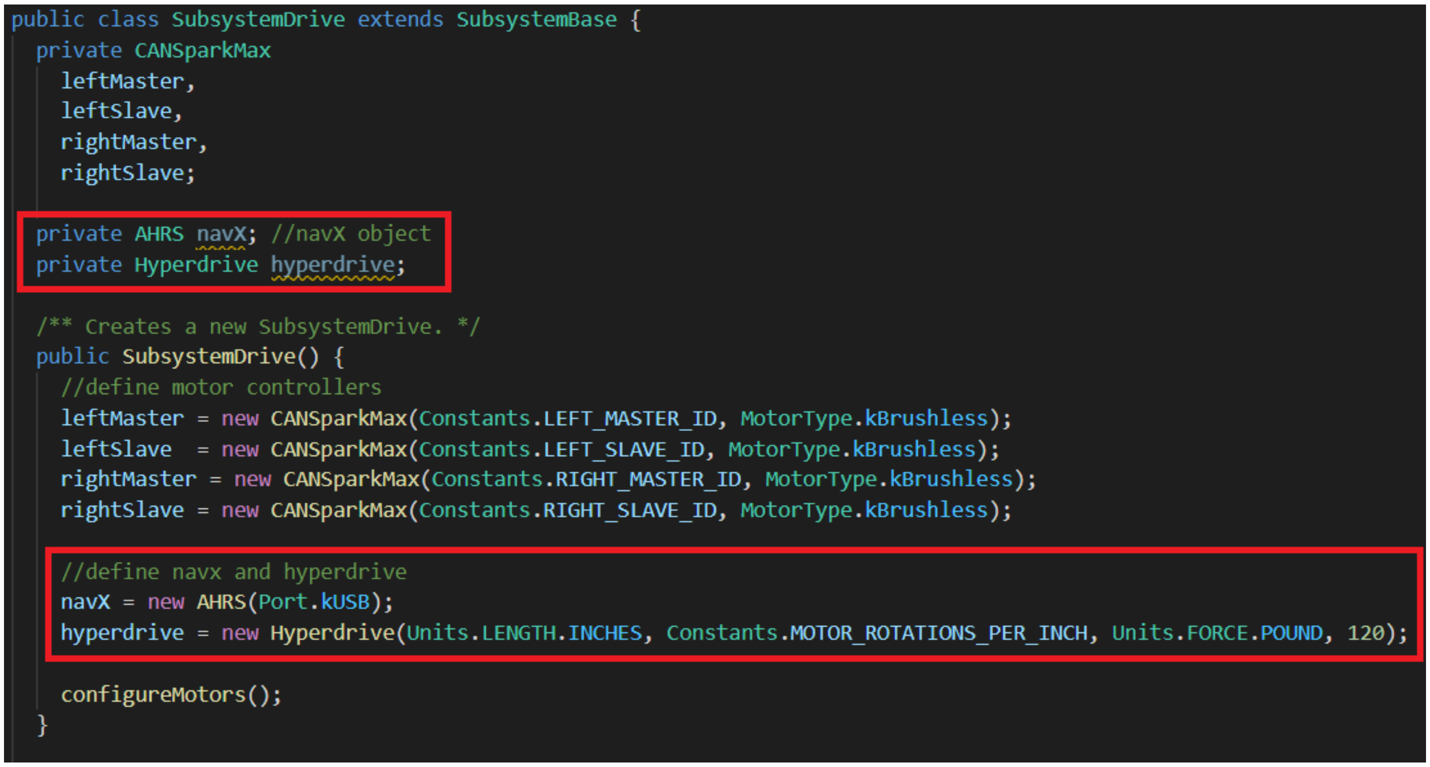
To measure the next value (wheelbase width), simply measure the length between the left and right wheels of the robot. This length should be measured in the same unit that the previous value converts between. For example, the value used in the example is 0.412 motor rotations per inch, so the wheelbase width should be measured in inches. If that value was 0.412 motor rotations per meter, then the wheelbase width should be measured in meters. In this example, the wheelbase width is 24 inches.



After measuring these two values, they should be put into your **Constants.java** file.

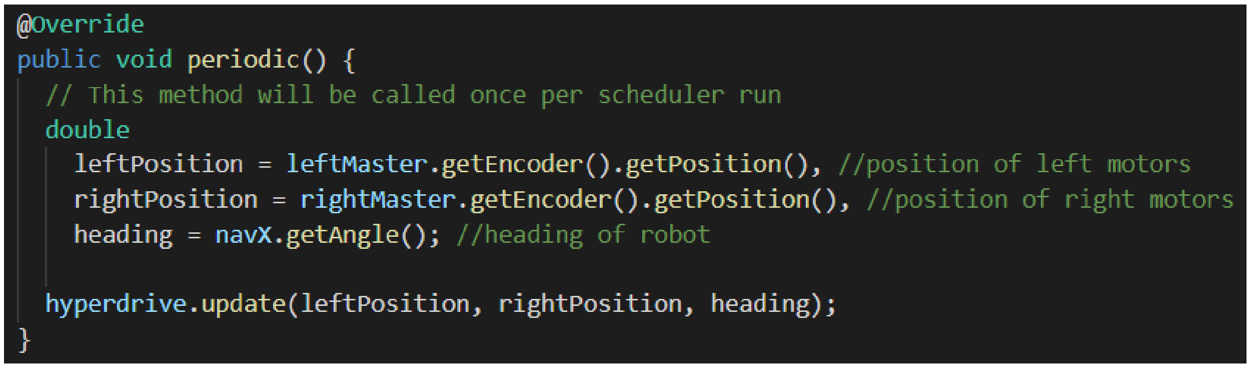
```
public static final double
    MOTOR_ROTATIONS_PER_INCH = 0.412,
    ROBOT_WHEELBASE_WIDTH = 24; //inches
```

Now, add our gyro and a `Hyperdrive` object to the `SubsystemDrive` class.



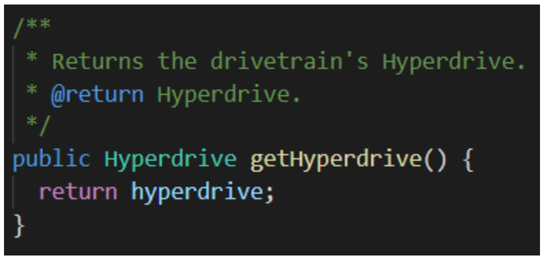
```java
public class SubsystemDrive extends SubsystemBase {
  private CANSparkMax
    leftMaster,
    leftSlave,
    rightMaster,
    rightSlave;

  private AHRS navX; //navX object
  private Hyperdrive hyperdrive;

  /** Creates a new SubsystemDrive. */
  public SubsystemDrive() {
    //define motor controllers
    leftMaster = new CANSparkMax(Constants.LEFT_MASTER_ID, MotorType.kBrushless);
    leftSlave  = new CANSparkMax(Constants.LEFT_SLAVE_ID, MotorType.kBrushless);
    rightMaster = new CANSparkMax(Constants.RIGHT_MASTER_ID, MotorType.kBrushless);
    rightSlave = new CANSparkMax(Constants.RIGHT_SLAVE_ID, MotorType.kBrushless);

    //define navx and hyperdrive
    navX = new AHRS(Port.kUSB);
    hyperdrive = new Hyperdrive(Units.LENGTH.INCHES, Constants.MOTOR_ROTATIONS_PER_INCH, Units.FORCE.POUND, 120);

    configureMotors();
  }
```

In the drivetrain's `periodic()` method, call the `update()` method on your newly-created `Hyperdrive`.



```java
@Override
public void periodic() {
  // This method will be called once per scheduler run
  double
    leftPosition = leftMaster.getEncoder().getPosition(), //position of left motors
    rightPosition = rightMaster.getEncoder().getPosition(), //position of right motors
    heading = navX.getAngle(); //heading of robot

  hyperdrive.update(leftPosition, rightPosition, heading);
}
```

Finally, add an accessor that returns the `Hyperdrive` object. This will be used later in the example.

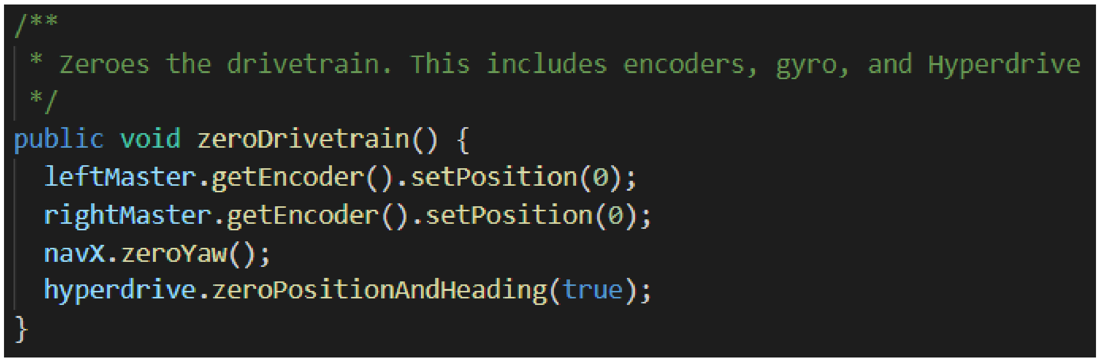


```java
/**
 * Returns the drivetrain's Hyperdrive.
 * @return Hyperdrive.
 */
public Hyperdrive getHyperdrive() {
  return hyperdrive;
}
```

SubsystemDrive.java

At this point, Hyperdrive is set up. Your robot will now connect to PathVisualizer and you can view your robot's position on the field. However, there are a few more steps required to make Hyperdrive functional.

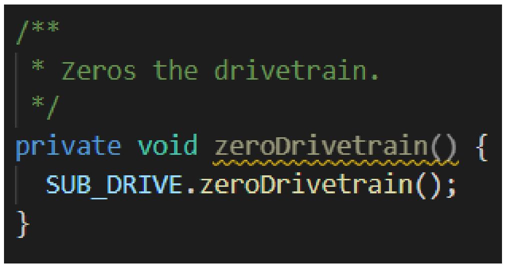
First, you will need to have a command on your dashboard that you can use to zero the robot's drivetrain encoders and gyro as well as the robot's position on the field. This command might be called at the start of autonomous or just used to zero the robot's position before testing a path. To start, define a method in your drivetrain class. In this example, we will call it `zeroDrivetrain()`. This method should zero the drivetrain encoders, the gyro, and Hyperdrive. Note that `Hyperdrive`'s `zeroPositionAndHeading()` method should only have an argument of `true` if the drivetrain encoder positions are being zeroed.



```java
/**
 * Zeroes the drivetrain. This includes encoders, gyro, and Hyperdrive
 */
public void zeroDrivetrain() {
  leftMaster.getEncoder().setPosition(0);
  rightMaster.getEncoder().setPosition(0);
  navX.zeroYaw();
  hyperdrive.zeroPositionAndHeading(true);
}
```

SubsystemDrive.java

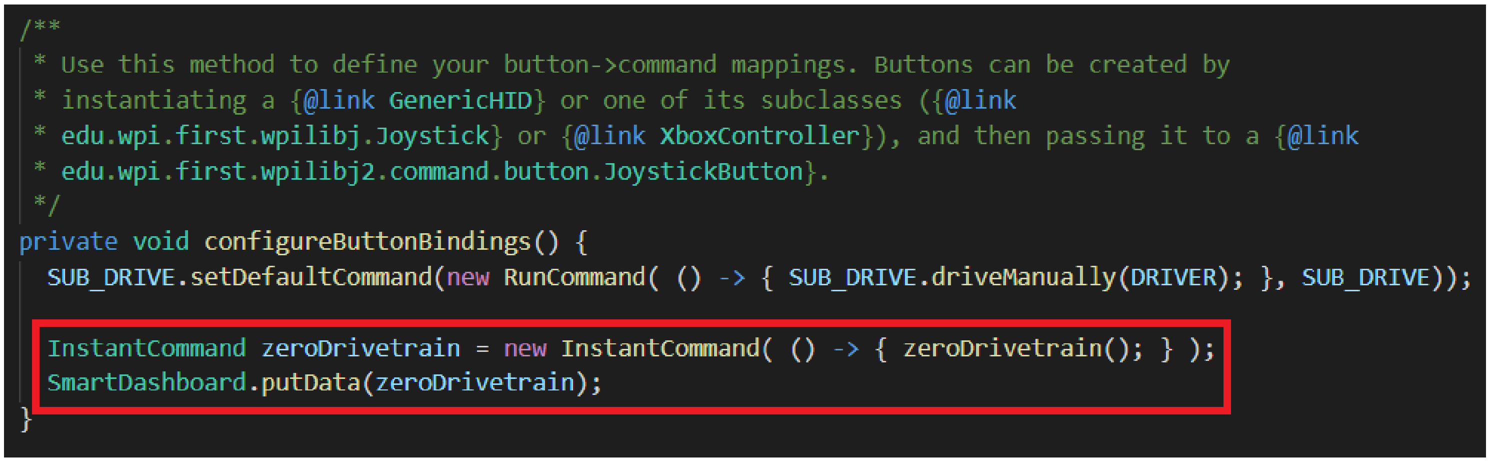
After creating the method in the drivetrain class, create a method in your `RobotContainer` class that will call the new method in the drivetrain class.



```java
/**
 * Zeros the drivetrain.
 */
private void zeroDrivetrain() {
  SUB_DRIVE.zeroDrivetrain();
}
```

RobotContainer.java

Now, in `RobotContainer`'s `configureButtonBindings()` method, send a command to the dashboard that calls this method.

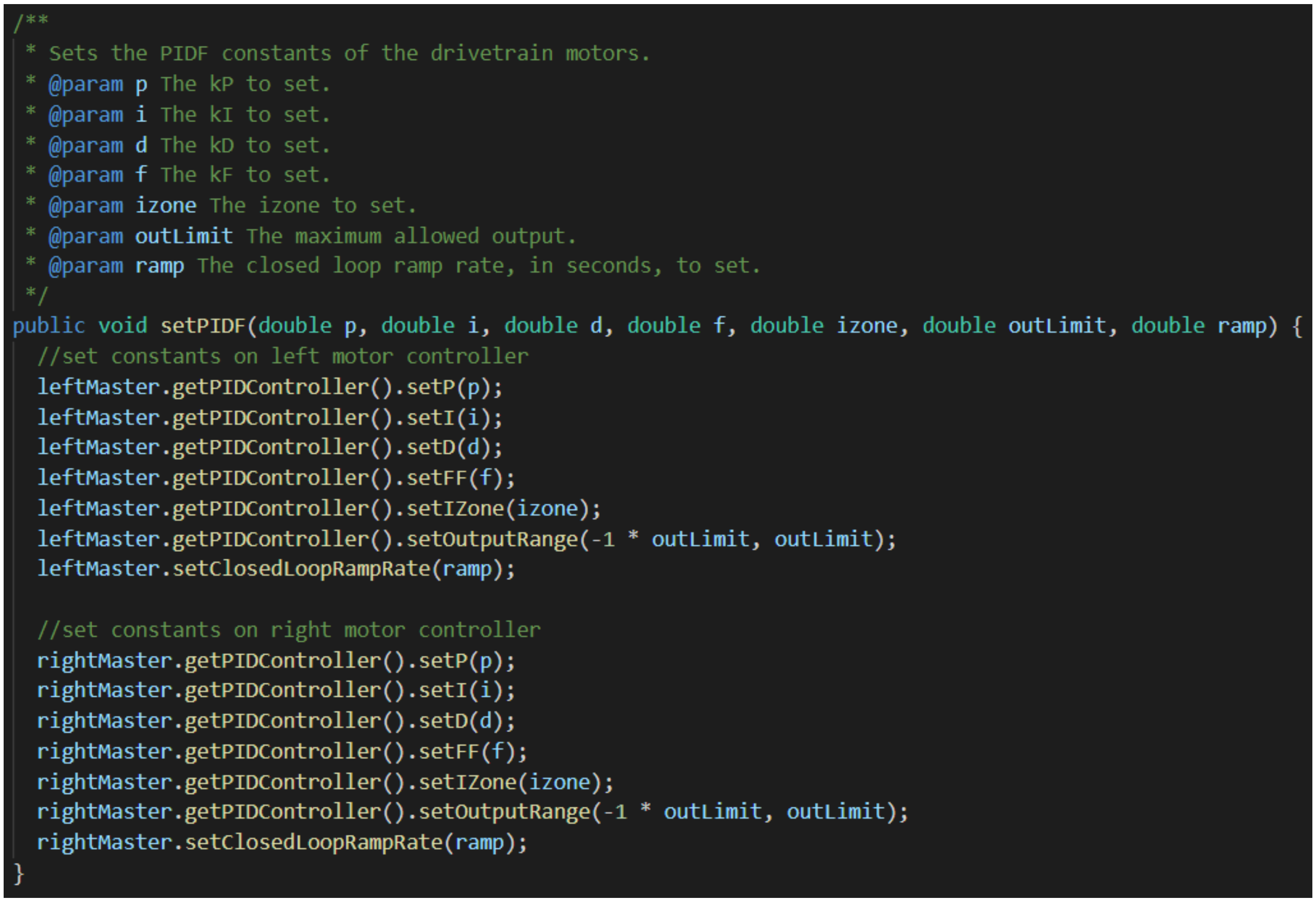


```java
/**
 * Use this method to define your button->command mappings. Buttons can be created by
 * instantiating a {@link GenericHID} or one of its subclasses ({@link
 * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it to a {@link
 * edu.wpi.first.wpilibj2.command.button.JoystickButton}.
 */
private void configureButtonBindings() {
  SUB_DRIVE.setDefaultCommand(new RunCommand( () -> { SUB_DRIVE.driveManually(DRIVER); }, SUB_DRIVE));

  InstantCommand zeroDrivetrain = new InstantCommand( () -> { zeroDrivetrain(); } );
  SmartDashboard.putData(zeroDrivetrain);
}
```

RobotContainer.java

You will now be able to put a button on the dashboard that will zero the drivetrain encoders, gyro, and Hyperdrive when clicked.

Now, we can work on getting Hyperdrive to drive the robot around autonomously. To start, we need to tune a PID loop that drives the robot forward at a constant velocity, which means that we need to add some methods to our drivetrain subsystem. In your drivetrain class, create a method that sets the PID constants on the master motors. In this example, it will be called `setPIDF()`. In addition to setting the kP, kI, kD, and kF constants, it is also useful to set an IZone, maximum allowed percent output, and closed loop ramp rate.

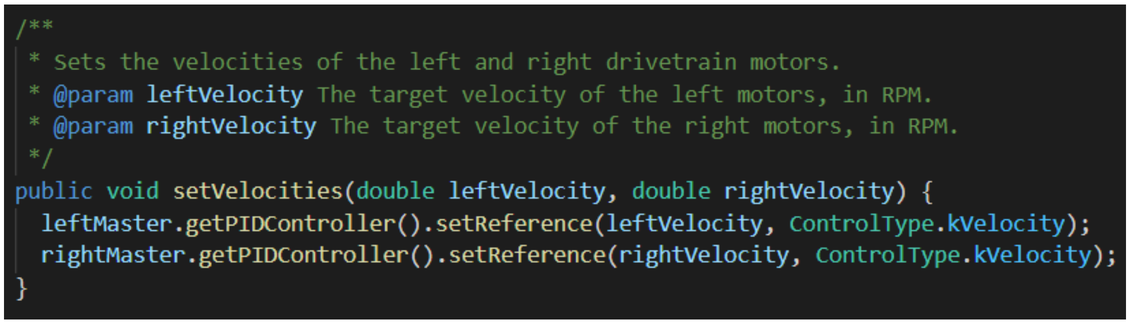


```java
/**
 * Sets the PIDF constants of the drivetrain motors.
 * @param p The kP to set.
 * @param i The kI to set.
 * @param d The kD to set.
 * @param f The kF to set.
 * @param izone The izone to set.
 * @param outLimit The maximum allowed output.
 * @param ramp The closed loop ramp rate, in seconds, to set.
 */
public void setPIDF(double p, double i, double d, double f, double izone, double outLimit, double ramp) {
  //set constants on left motor controller
  leftMaster.getPIDController().setP(p);
  leftMaster.getPIDController().setI(i);
  leftMaster.getPIDController().setD(d);
  leftMaster.getPIDController().setFF(f);
  leftMaster.getPIDController().setIZone(izone);
  leftMaster.getPIDController().setOutputRange(-1 * outLimit, outLimit);
  leftMaster.setClosedLoopRampRate(ramp);

  //set constants on right motor controller
  rightMaster.getPIDController().setP(p);
  rightMaster.getPIDController().setI(i);
  rightMaster.getPIDController().setD(d);
  rightMaster.getPIDController().setFF(f);
  rightMaster.getPIDController().setIZone(izone);
  rightMaster.getPIDController().setOutputRange(-1 * outLimit, outLimit);
  rightMaster.setClosedLoopRampRate(ramp);
}
```

SubsystemDrive.java

This method sets all PID constants needed for a well-tuned loop. It only needs to set the constants on the master motors because the slave motors will automatically “follow” the master motors (see `configureMotors()` for more).

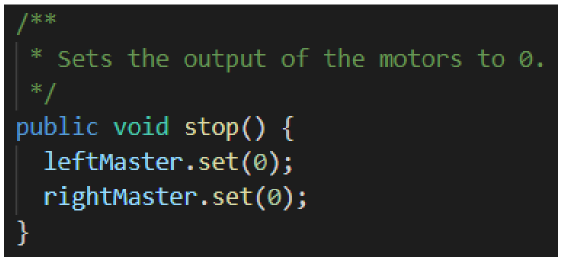
After creating the `setPIDF()` method, we need to create a method that sets the target velocity of the drivetrain motors.



```java
/**
 * Sets the velocities of the left and right drivetrain motors.
 * @param leftVelocity The target velocity of the left motors, in RPM.
 * @param rightVelocity The target velocity of the right motors, in RPM.
 */
public void setVelocities(double leftVelocity, double rightVelocity) {
  leftMaster.getPIDController().setReference(leftVelocity, ControlType.kVelocity);
  rightMaster.getPIDController().setReference(rightVelocity, ControlType.kVelocity);
}
```

SubsystemDrive.java

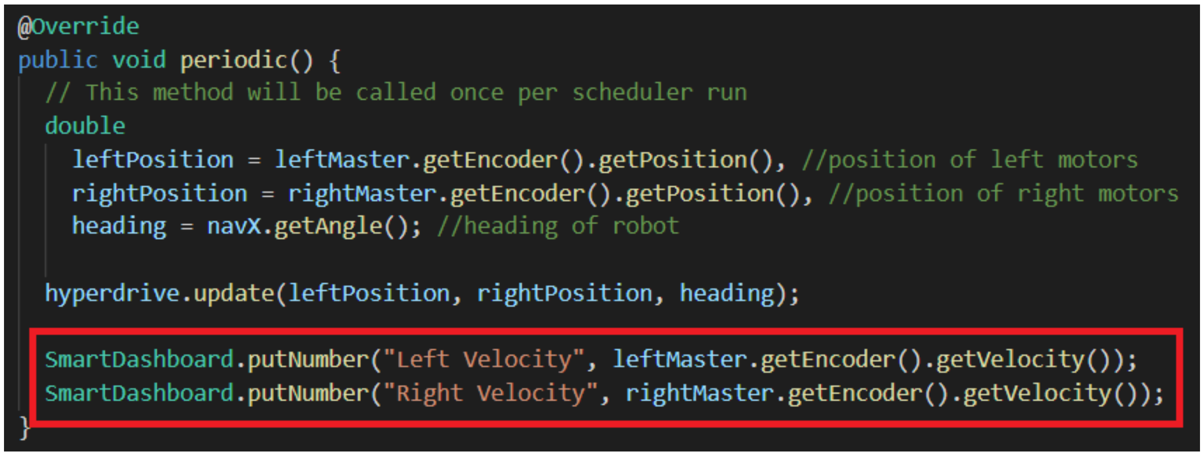
We should also make a method that stops the drivetrain by setting the percent output of the left and right motors to 0.



```java
/**
 * Sets the output of the motors to 0.
 */
public void stop() {
  leftMaster.set(0);
  rightMaster.set(0);
}
```

SubsystemDrive.java

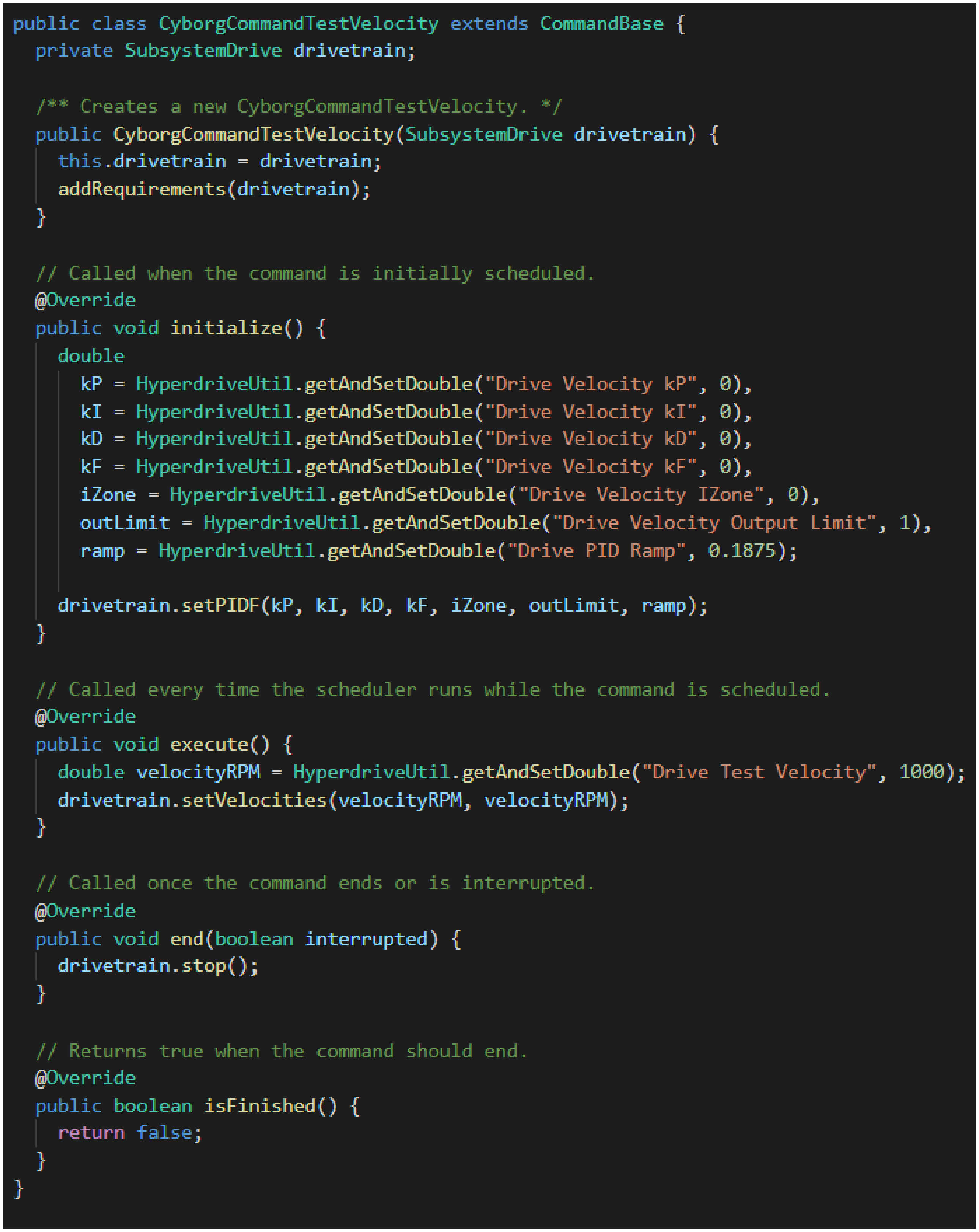
Finally, to make PID tuning easier, we should output the velocity of the robot to the dashboard. We will do that in the drivetrain's `periodic()` method.



```java
@Override
public void periodic() {
  // This method will be called once per scheduler run
  double
    leftPosition = leftMaster.getEncoder().getPosition(), //position of left motors
    rightPosition = rightMaster.getEncoder().getPosition(), //position of right motors
    heading = navX.getAngle(); //heading of robot

  hyperdrive.update(leftPosition, rightPosition, heading);

  SmartDashboard.putNumber("Left Velocity", leftMaster.getEncoder().getVelocity());
  SmartDashboard.putNumber("Right Velocity", rightMaster.getEncoder().getVelocity());
}
```

SubsystemDrive.java

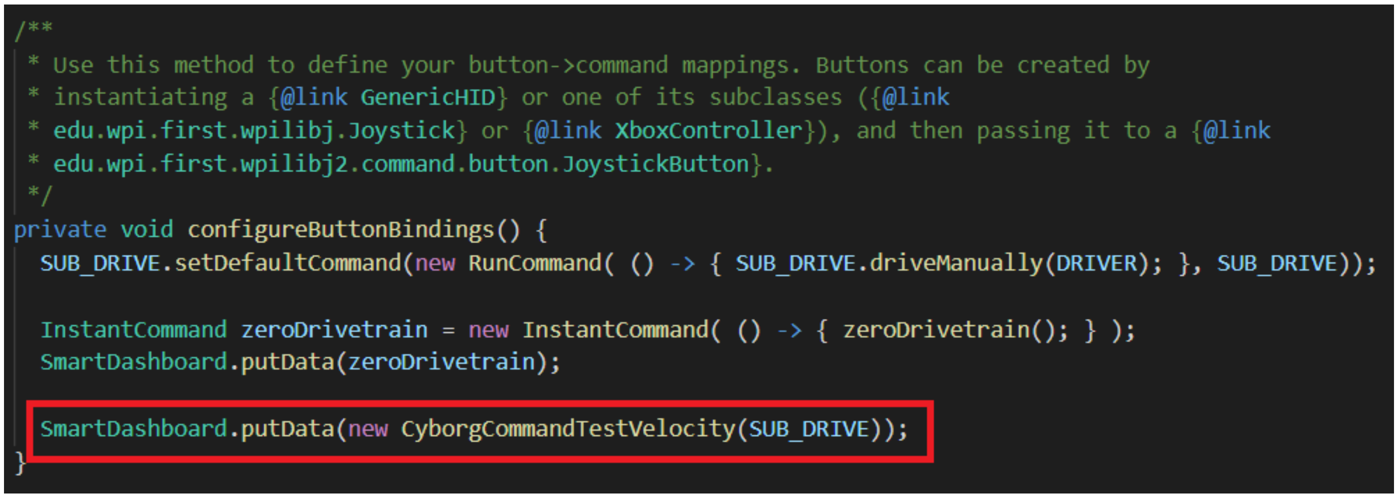
Now, we must create a command that we can use to test the velocity PID loop as we tune it. For this example, we will create a new command called `CyborgCommandTestVelocity`. This command's only purpose will be to drive the robot forwards at a user-specified speed.



```java
public class CyborgCommandTestVelocity extends CommandBase {
  private SubsystemDrive drivetrain;

  /** Creates a new CyborgCommandTestVelocity. */
  public CyborgCommandTestVelocity(SubsystemDrive drivetrain) {
    this.drivetrain = drivetrain;
    addRequirements(drivetrain);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {
    double
      kP = HyperdriveUtil.getAndSetDouble("Drive Velocity kP", 0),
      kI = HyperdriveUtil.getAndSetDouble("Drive Velocity kI", 0),
      kD = HyperdriveUtil.getAndSetDouble("Drive Velocity kD", 0),
      kF = HyperdriveUtil.getAndSetDouble("Drive Velocity kF", 0),
      iZone = HyperdriveUtil.getAndSetDouble("Drive Velocity IZone", 0),
      outLimit = HyperdriveUtil.getAndSetDouble("Drive Velocity Output Limit", 1),
      ramp = HyperdriveUtil.getAndSetDouble("Drive PID Ramp", 0.1875);

    drivetrain.setPIDF(kP, kI, kD, kF, iZone, outLimit, ramp);
  }

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {
    double velocityRPM = HyperdriveUtil.getAndSetDouble("Drive Test Velocity", 1000);
    drivetrain.setVelocities(velocityRPM, velocityRPM);
  }

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
    drivetrain.stop();
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return false;
  }
}
```

CyborgCommandTestVelocity.java

In order to be able to run the command, send it to the dashboard in `RobotContainer`'s `configureButtonBindings()` method.

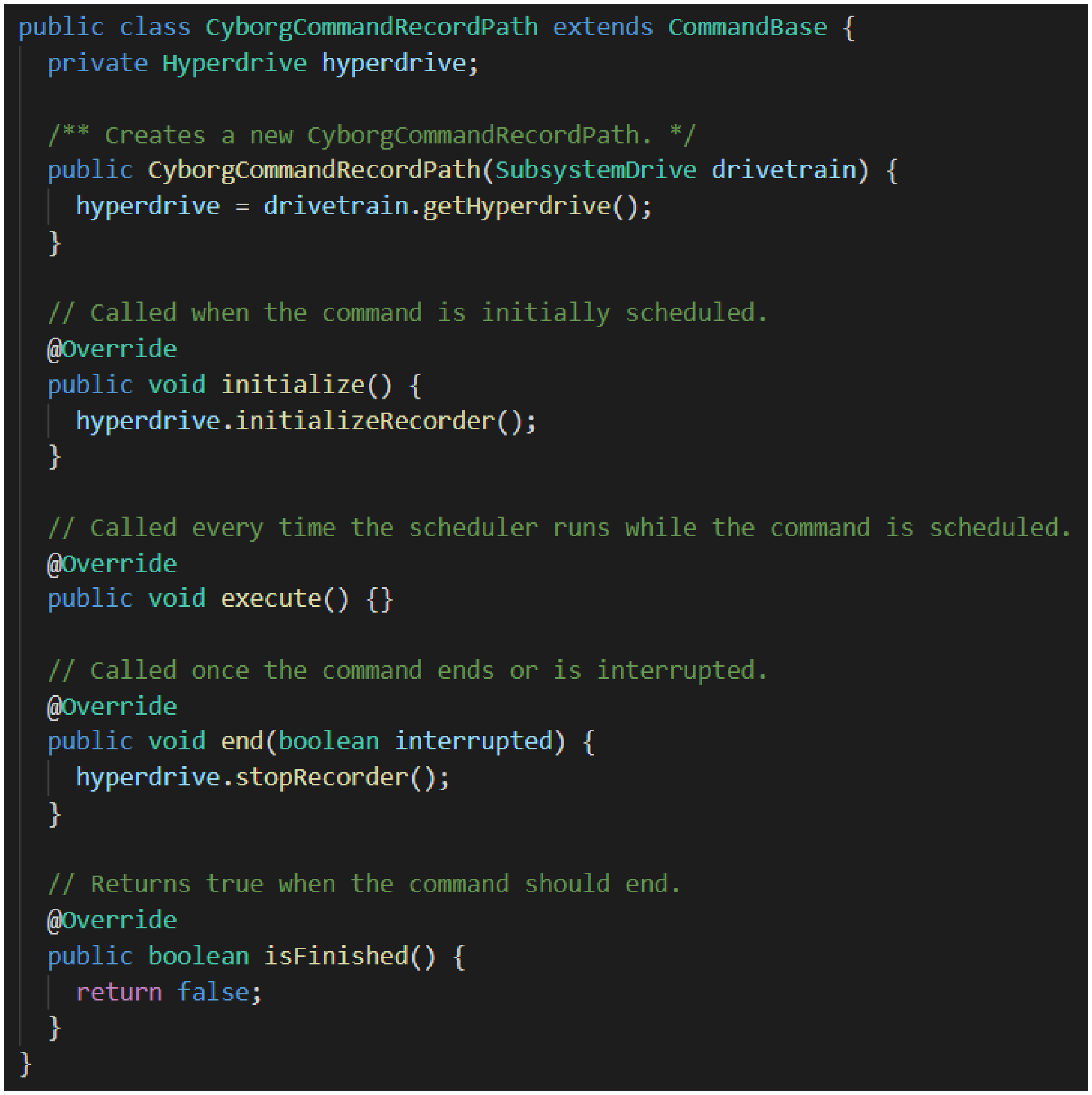


```java
/**
 * Use this method to define your button->command mappings. Buttons can be created by
 * instantiating a {@link GenericHID} or one of its subclasses ({@link
 * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it to a {@link
 * edu.wpi.first.wpilibj2.command.button.JoystickButton}.
 */
private void configureButtonBindings() {
  SUB_DRIVE.setDefaultCommand(new RunCommand( () -> { SUB_DRIVE.driveManually(DRIVER); }, SUB_DRIVE));

  InstantCommand zeroDrivetrain = new InstantCommand( () -> { zeroDrivetrain(); } );
  SmartDashboard.putData(zeroDrivetrain);

  SmartDashboard.putData(new CyborgCommandTestVelocity(SUB_DRIVE));
}
```

RobotContainer.java

At this point, you can push the code to your robot and use the command you just created to test the velocity PID loop of your robot. The kP, kI, kD, kF, IZone, and output limit constants can be modified in the Preferences table on the dashboard. As you tune your PID loop, make sure that it fully reaches the velocity target with minimal oscillation (which means you will probably need to add some kI). Oscillation about the velocity setpoint makes for messy path driving.

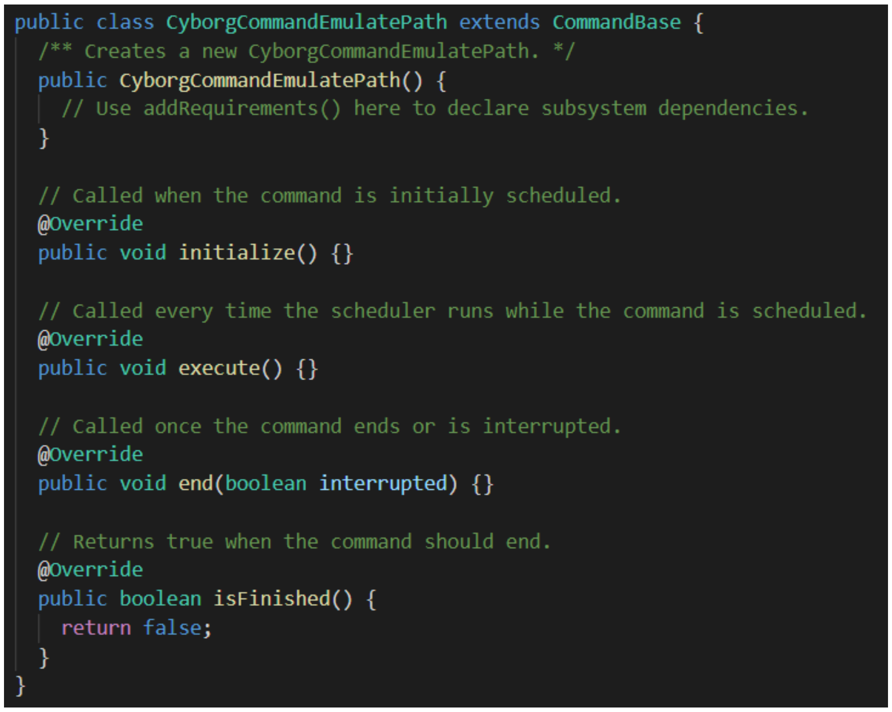
After tuning your PID loop, you can now create the commands to record and emulate paths using Hyperdrive. We will start with the "record" command. In this example, it will be called `CyborgCommandRecordPath`. This command is very simple: initialize Hyperdrives recorder in the command's `initialize()` method, and stop the recorder in the command's `end()` method.



```java
public class CyborgCommandRecordPath extends CommandBase {
    private Hyperdrive hyperdrive;

    /** Creates a new CyborgCommandRecordPath. */
    public CyborgCommandRecordPath(SubsystemDrive drivetrain) {
        hyperdrive = drivetrain.getHyperdrive();
    }

    // Called when the command is initially scheduled.
    @Override
    public void initialize() {
        hyperdrive.initializeRecorder();
    }

    // Called every time the scheduler runs while the command is scheduled.
    @Override
    public void execute() {}

    // Called once the command ends or is interrupted.
    @Override
    public void end(boolean interrupted) {
        hyperdrive.stopRecorder();
    }

    // Returns true when the command should end.
    @Override
    public boolean isFinished() {
        return false;
    }
}
```

CyborgCommandRecordPath.java

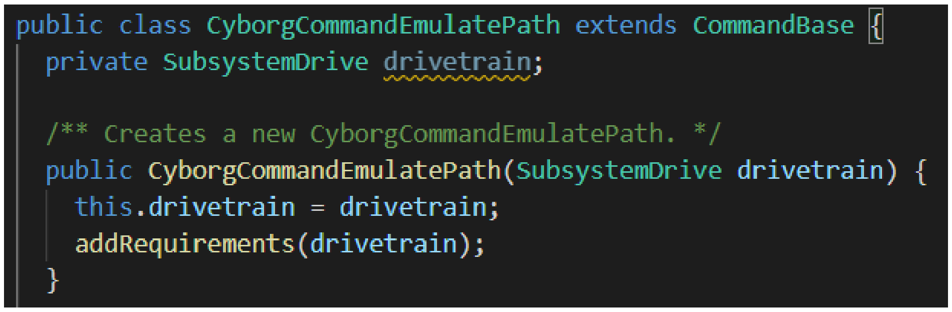
Now, create the "emulate" command. In this example, it will be called `CyborgCommandEmulatePath`.



```java
public class CyborgCommandEmulatePath extends CommandBase {
  /** Creates a new CyborgCommandEmulatePath. */
  public CyborgCommandEmulatePath() {
    // Use addRequirements() here to declare subsystem dependencies.
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {}

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {}

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {}

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return false;
  }
}
```

CyborgCommandEmulatePath.java

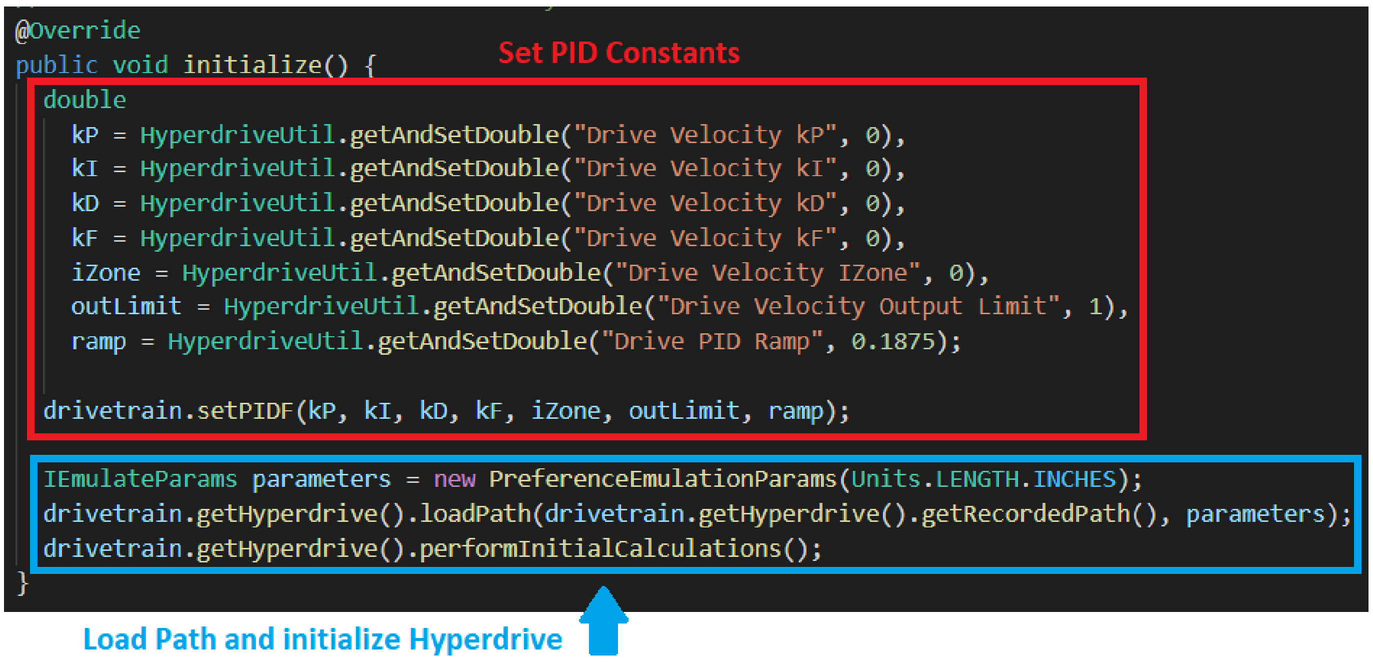
Because the command will use the drivetrain subsystem to drive the robot around, the command needs a private instance of the subsystem and it should be added as a requirement in the constructor.



```java
public class CyborgCommandEmulatePath extends CommandBase {
  private SubsystemDrive drivetrain;

  /** Creates a new CyborgCommandEmulatePath. */
  public CyborgCommandEmulatePath(SubsystemDrive drivetrain) {
    this.drivetrain = drivetrain;
    addRequirements(drivetrain);
  }
```

CyborgCommandEmulatePath.java

In the command's `initialize()` method, the drivetrain's PID constants should be set. In addition, the Path being driven should be loaded into Hyperdrive using the `loadPath()` method and Hyperdrive should be initialized using the `performInitialCalculations()` method. The path loaded can be any path you would like (it can even be passed as an argument to the constructor of the command which is a very good idea if you need to emulate paths during autonomous), but this command will be programmed to simply emulate the path that Hyperdrive has recorded most recently.

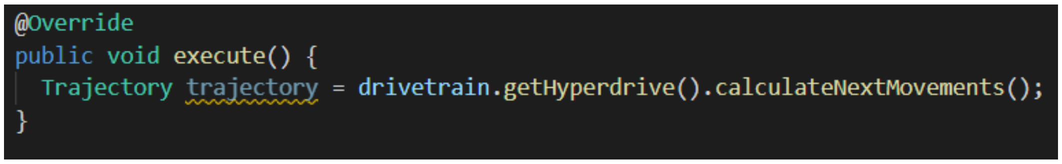


```java
@Override
public void initialize() {
    double
        kP = HyperdriveUtil.getAndSetDouble("Drive Velocity kP", 0),
        kI = HyperdriveUtil.getAndSetDouble("Drive Velocity kI", 0),
        kD = HyperdriveUtil.getAndSetDouble("Drive Velocity kD", 0),
        kF = HyperdriveUtil.getAndSetDouble("Drive Velocity kF", 0),
        iZone = HyperdriveUtil.getAndSetDouble("Drive Velocity IZone", 0),
        outLimit = HyperdriveUtil.getAndSetDouble("Drive Velocity Output Limit", 1),
        ramp = HyperdriveUtil.getAndSetDouble("Drive PID Ramp", 0.1875);

    drivetrain.setPIDF(kP, kI, kD, kF, iZone, outLimit, ramp);

    IEmulateParams parameters = new PreferenceEmulationParams(Units.LENGTH.INCHES);
    drivetrain.getHyperdrive().loadPath(drivetrain.getHyperdrive().getRecordedPath(), parameters);
    drivetrain.getHyperdrive().performInitialCalculations();
}
```

CyborgCommandEmulatePath.java

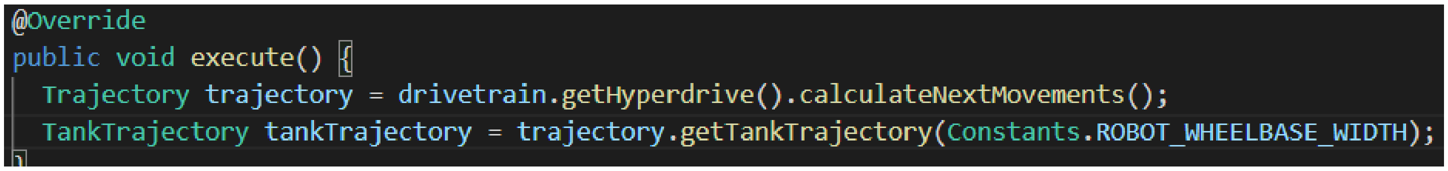
In addition, an `IEmulateParams` object was defined. This object defines all of the parameters that Hyperdrive will follow as it drives the robot through a path. `PreferenceEmulationParams` is a type of `IEmulateParams` that stores its values in the Preferences table so that the user can modify them and tune the robot's path driving. The other type of `IEmulateParams` is `ConstantEmulationParams` which uses constant values that cannot be modified. `ConstantEmulationParams` should only be used with values that have been confirmed to work. See the documentation for `IEmulateParams` for more about what the different parameters are and what they do.

In the command's `execute()` method, Hyperdrive must calculate the trajectory that the robot should take and then calculate the left and right wheel velocities needed to make that trajectory happen. To calculate the trajectory that the robot needs to take, use `Hyperdrive`'s `calculateNextMovements()` method.



```java
@Override
public void execute() {
    Trajectory trajectory = drivetrain.getHyperdrive().calculateNextMovements();
}
```

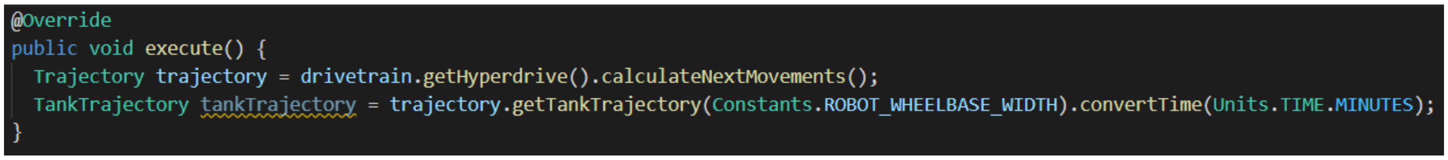
To calculate the left and right wheel velocities needed to drive that trajectory, it must be converted to a `TankTrajectory` object. This can be done by calling the `getTankTrajectory()` method.



```
@Override
public void execute() {
  Trajectory trajectory = drivetrain.getHyperdrive().calculateNextMovements();
  TankTrajectory tankTrajectory = trajectory.getTankTrajectory(Constants.ROBOT_WHEELBASE_WIDTH);
}
```

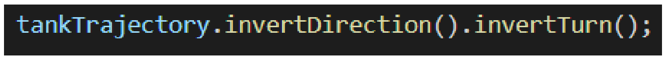
Now, the wheel velocities needed to drive the robot through that trajectory have been calculated. However, the velocities are calculated as values of motors units per SECOND. Spark Max motor controller PID loops take setpoints of rotations per MINUTE, and CTRE motor controllers take setpoints in ticks per 100 MILLISECONDS. In order for the velocities to be correct, they must be converted. This can be done by chaining on the `convertTime()` method with the correct time unit.

- REV motor controllers: use `Units.TIME.MINUTES`
- CTRE motor controllers: use `Units.TIME.DECASECONDS`



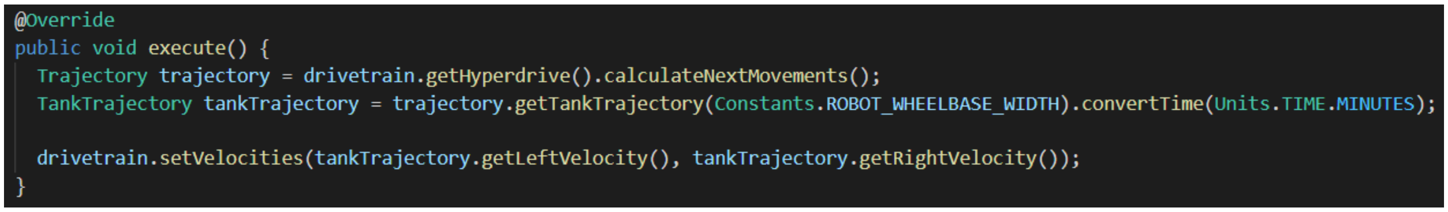
```
@Override
public void execute() {
  Trajectory trajectory = drivetrain.getHyperdrive().calculateNextMovements();
  TankTrajectory tankTrajectory = trajectory.getTankTrajectory(Constants.ROBOT_WHEELBASE_WIDTH).convertTime(Units.TIME.MINUTES);
}
```

There are also other modifiers that can be used to change the way that the robot drives its path. The `invertTurn()` method switches the left and right velocities so that the robot's turns are inverted. The `invertDirection()` method multiplies the velocities by -1 so that the robot drives backwards. These can be used if the robot does not drive its path correctly. They are also chainable so that they can be called in one line as such:



```
tankTrajectory.invertDirection().invertTurn();
```

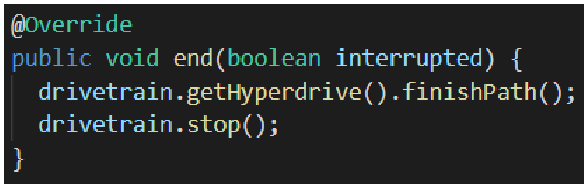
After converting our velocities to the correct time unit, they can be directly set on the motor controllers.



```
@Override
public void execute() {
  Trajectory trajectory = drivetrain.getHyperdrive().calculateNextMovements();
  TankTrajectory tankTrajectory = trajectory.getTankTrajectory(Constants.ROBOT_WHEELBASE_WIDTH).convertTime(Units.TIME.MINUTES);

  drivetrain.setVelocities(tankTrajectory.getLeftVelocity(), tankTrajectory.getRightVelocity());
}
```

CyborgCommandEmulatePath.java

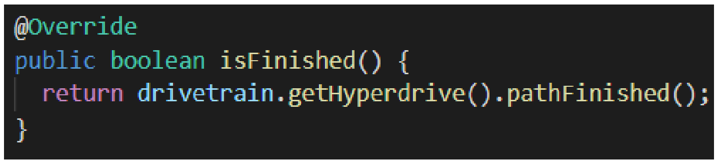
In the command's `end()` method, the Hyperdrive's `finishPath()` method must be called and the drivetrain should be stopped.



```java
@Override
public void end(boolean interrupted) {
  drivetrain.getHyperdrive().finishPath();
  drivetrain.stop();
}
```

CyborgCommandEmulatePath.java

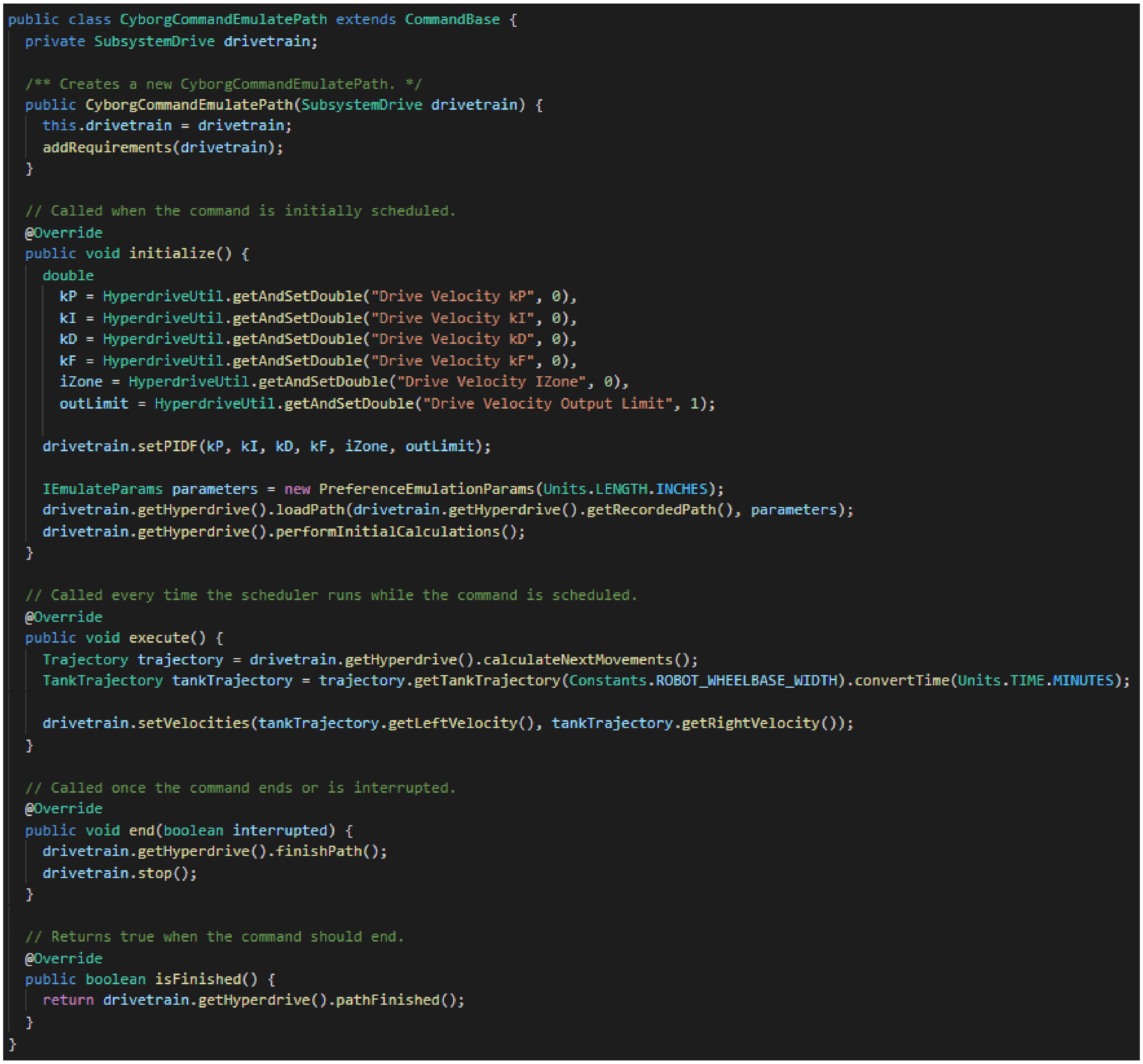
The command’s `isFinished()` method should return the value of Hyperdrive’s `pathFinished()` method.



```java
@Override
public boolean isFinished() {
  return drivetrain.getHyperdrive().pathFinished();
}
```

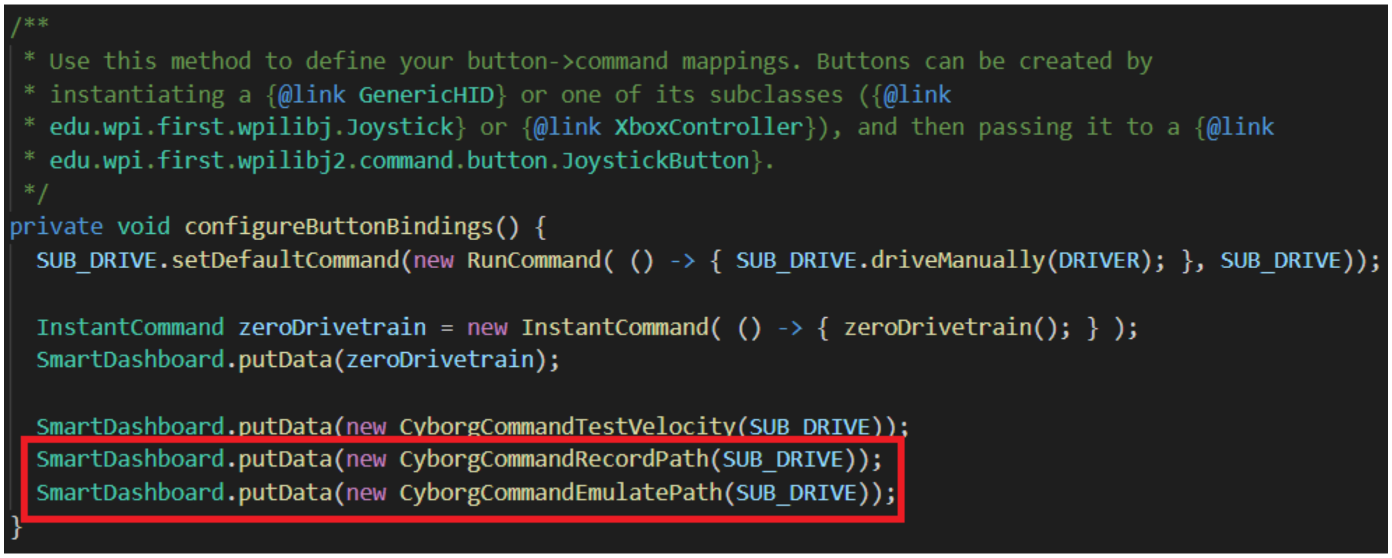
The command, after being written, should look like the command on the next page.

CyborgCommandEmulatePath.java



```java
public class CyborgCommandEmulatePath extends CommandBase {
  private SubsystemDrive drivetrain;

  /** Creates a new CyborgCommandEmulatePath. */
  public CyborgCommandEmulatePath(SubsystemDrive drivetrain) {
    this.drivetrain = drivetrain;
    addRequirements(drivetrain);
  }

  // Called when the command is initially scheduled.
  @Override
  public void initialize() {
    double
      kP = HyperdriveUtil.getAndSetDouble("Drive Velocity kP", 0),
      kI = HyperdriveUtil.getAndSetDouble("Drive Velocity kI", 0),
      kD = HyperdriveUtil.getAndSetDouble("Drive Velocity kD", 0),
      kF = HyperdriveUtil.getAndSetDouble("Drive Velocity kF", 0),
      iZone = HyperdriveUtil.getAndSetDouble("Drive Velocity IZone", 0),
      outLimit = HyperdriveUtil.getAndSetDouble("Drive Velocity Output Limit", 1);

    drivetrain.setPIDF(kP, kI, kD, kF, iZone, outLimit);

    IEmulateParams parameters = new PreferenceEmulationParams(Units.LENGTH.INCHES);
    drivetrain.getHyperdrive().loadPath(drivetrain.getHyperdrive().getRecordedPath(), parameters);
    drivetrain.getHyperdrive().performInitialCalculations();
  }

  // Called every time the scheduler runs while the command is scheduled.
  @Override
  public void execute() {
    Trajectory trajectory = drivetrain.getHyperdrive().calculateNextMovements();
    TankTrajectory tankTrajectory = trajectory.getTankTrajectory(Constants.ROBOT_WHEELBASE_WIDTH).convertTime(Units.TIME.MINUTES);

    drivetrain.setVelocities(tankTrajectory.getLeftVelocity(), tankTrajectory.getRightVelocity());
  }

  // Called once the command ends or is interrupted.
  @Override
  public void end(boolean interrupted) {
    drivetrain.getHyperdrive().finishPath();
    drivetrain.stop();
  }

  // Returns true when the command should end.
  @Override
  public boolean isFinished() {
    return drivetrain.getHyperdrive().pathFinished();
  }
}
```

Now, all you have to do is send your "Record" and "Emulate" commands to the dashboard in your `RobotContainer` class.



```java
/**
 * Use this method to define your button->command mappings. Buttons can be created by
 * instantiating a {@link GenericHID} or one of its subclasses ({@link
 * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it to a {@link
 * edu.wpi.first.wpilibj2.command.button.JoystickButton}.
 */
private void configureButtonBindings() {
  SUB_DRIVE.setDefaultCommand(new RunCommand( () -> { SUB_DRIVE.driveManually(DRIVER); }, SUB_DRIVE));

  InstantCommand zeroDrivetrain = new InstantCommand( () -> { zeroDrivetrain(); } );
  SmartDashboard.putData(zeroDrivetrain);

  SmartDashboard.putData(new CyborgCommandTestVelocity(SUB_DRIVE));
  SmartDashboard.putData(new CyborgCommandRecordPath(SUB_DRIVE));
  SmartDashboard.putData(new CyborgCommandEmulatePath(SUB_DRIVE));
}
```

RobotContainer.java

Now you can push code to your robot and try out its new autonomous driving capabilities. First, zero your robot using the `InstantCommand` you created towards the beginning of the example. Then, activate your `CyborgCommandRecordPath` and use a controller to manually drive the robot through a path. After that, deactivate the command, drive the robot back to its starting position and heading, and zero the robot again. Then, activate your `CyborgCommandEmulatePath` command and watch your robot drive through the path you just made. The robot will probably not drive through the path perfectly, and may have made some major mistakes as it drove through the path that first time. This is why you should use a `PreferenceEmulationParams` to tune your robot's path driving. Use the "Tuning" table in the next chapter to assist you with tuning your parameters for optimal robot function.

Spend some time recording, emulating, and tuning your paths on your robot. At very high speeds, paths may need to have unique parameters because of the increased effect of gyro and positional drift, but at medium and lower speeds, one set of parameters should effectively drive the robot through any path.

The full example project that was just covered can be found at github.com/BTK203/Hyperdrive/tree/main/ExampleProject.

Tips, Tuning, and Troubleshooting

Autonomous driving is complex, so you may come across some trouble while setting up the library. This chapter will cover some tips and troubleshooting strategies that you can use to help you add Hyperdrive to your robot.

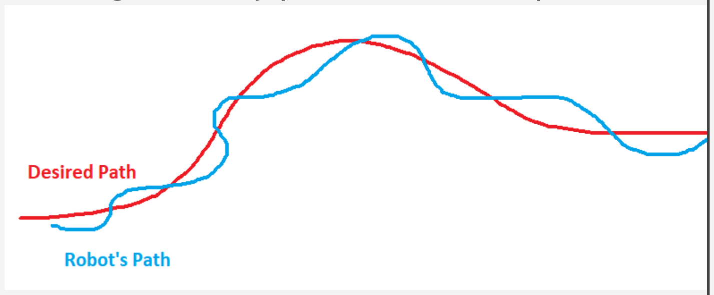
Tips

- **Follow the example project outlined in the previous chapter.** The example project closely guides you through all of the components needed for Hyperdrive to fully work on your robot.
- **Start slow.** It is always less scary when the robot veers off course at slow speeds. In addition, Hyperdrive works better at slow speeds. That being said, as you test your robot's first paths, start it with a low minimum and maximum speed and increase it gradually.
- **Use a real gyro.** The `TankGyro` class, while offering potential for future use, simply is not at the stage where it should be used on a competition robot. For best results, use a gyro such as the NavX mini.
- **Do not let your robot drift or slide.** Hyperdrive's robot position tracker relies on the position of the motors accurately depicting their displacement. As soon as the wheels slip, the motor position no longer has that accuracy. When recording and emulating paths, keep robot drift to a minimum.
- **While recording, avoid turning the robot in place.** Hyperdrive is capable of driving backwards and making 3-point turns, but is not programmed to handle turning the robot large amounts in place.
- **Use PathVisualizer.** Especially when tuning the parameters, PathVisualizer is an excellent tool that you can use to compare the desired path to the robot's actual path. This can help you identify problems with your robot's autonomous driving and fix them easily.

Tuning

This table outlines some symptoms of a Hyperdrive that needs to be tuned, and which parameters to change to tune them out:

Symptom	Fix
Robot takes turns too shallowly	Increase the "Emulate Overturn" value.
Robot takes turns too tightly	Decrease the "Emulate Overturn" value.

When the robot gets off course, it takes a long time to get back on course (if it even tries at all)	Increase the "Emulate Positional Correction Inhibitor" value. If you are using big length units such as meters or yards, this value may need to be increased drastically to as high as 1 or 1.5.
The robot works too hard to stay on course, resulting in a wavy path like the one pictured: 	Decrease the "Emulate Positional Correction Inhibitor" value.
The robot takes turns too late and/or the path that it drives is very rough	Increase the "Emulate Immediate Path Size" value.
The robot takes turns too early and cuts corners too much	Decrease the "Emulate Immediate Path Size" value.
The robot's drive gearboxes make a kind of grinding noise, as if the motors are trying to change directions multiple times per second	Increase your motor controllers' closed loop ramp rate. This ramp rate should be greater than 0 seconds but less than 0.3 seconds.
The robot does not seem very responsive as it drives the path	Decrease your motor controllers' closed loop ramp rate. This ramp rate should be greater than 0 seconds but less than 0.3 seconds. If that does not work, decrease the "Emulate Immediate Path Size" value.
The robot takes corners too fast, resulting in a gyro or positional drift	Increase the "Emulate Coefficient of Static Friction" value.
The robot takes corners too slowly	Decrease the "Emulate Coefficient of Static Friction" value.

Troubleshooting

This table outlines some problems that you may experience while using Hyperdrive and some potential fixes for those problems:

Symptom	Fix
Hyperdrive drives unpredictably, specifically upon reaching the first turn of a path.	Ensure that the gyro is connected to the robot and is functioning properly.
Upon starting a path, the robot drives in the wrong direction or displays some other unpredictable behavior.	First, position the robot back at its starting point and make sure that its position is correct. This can be done using PathVisualizer. Load the path that you wish to emulate and ensure that the black robot position dot is by the start of the path (which is marked with a green dot). If that does not fix the problem, then the robot's direction may be inverted. You can fix that problem by calling the `invertDirection()` method on the `TankTrajectory` object that you use to set your motor velocities.
The robot turns the wrong direction.	Call the `invertTurn()` method on the `TankTrajectory` object that you use to set your motor velocities.
The robot finishes the path, but it is facing the wrong direction.	Re-record the path. While recording, keep in mind that Hyperdrive was not programmed to turn robots in place. Anytime the robot turns, try to make sure that it is moving forwards or backwards as well.
The robot seems to exaggerate a turn that does not exist.	Hyperdrive, depending on the configuration of the parameters, will sometimes exaggerate small turns. To fix this, re-record the path and try to minimize the number of turns that the robot makes. If the robot can drive straight through an area, it should.
The robot does not drive smoothly. The drive motors seem to be giving an inconsistent amount of power.	Ensure that the velocity PID that the drivetrain is running is tuned. The robot should be able to speed up to almost any speed without overshooting or oscillating about the target velocity. Most likely, the kP value is too high.

PathVisualizer says that the robot emulated a path correctly, but it didn't	Ensure that the robot started in the correct place and that the wheels did not drift while the robot was driving the path. Also check to see that the gyro heading is correct. It should be consistent with what it was before the path was driven.

Troubleshooting Strategies

The table above should cover most of the problems that you might come across as you add autonomous driving abilities to your robot. However, there may be more problems that may come up. This list of steps, if followed in order, should help you resolve most of those problems.

- If the problem involves a specific Path, re-record it.
- Slow down the robot by decreasing the minimum and maximum speeds.
- Try starting your robot in a different position or record a different path that achieves the same purpose.
- Make sure that your gyro is plugged in and functioning properly.
- Try recording the path with no changes in direction and minimal turns.
- Make sure that your target velocities are being calculated with the correct time unit (i.e make sure that `convertTime()` is being called on your `TankTrajectory` with the right time unit)
- Make sure that your robot's heading is correctly reported by your gyro.
- Make sure that your robot's position is correctly reported by Hyperdrive.
- Try using the default parameters (`ConstantEmulationParams.getDefaults()`)
- Make sure that your drivetrain velocity PID loop is properly tuned.
- Make sure that Hyperdrive is correctly implemented and functioning on your robot. Use the example project to help.
- Try using a length unit of either inches, feet, or meters (those three are confirmed to work with driving the robot around; the other length units can be correctly converted to and from any other supported unit, but have not been used to drive a robot around)