# 4907 THUNDERSTAMPS SWERVE DRIVE

Programming Tutorial rev. 1.0

Scott Whitlock

Adam Kafka

# Table of Contents

# Introduction

This document explains how to program the 4907 ThunderStamps swerve drive. The software is broken up into modules (each module is to be written as a class in Java) and each module has one straightforward purpose.

## What is a Swerve Drive?

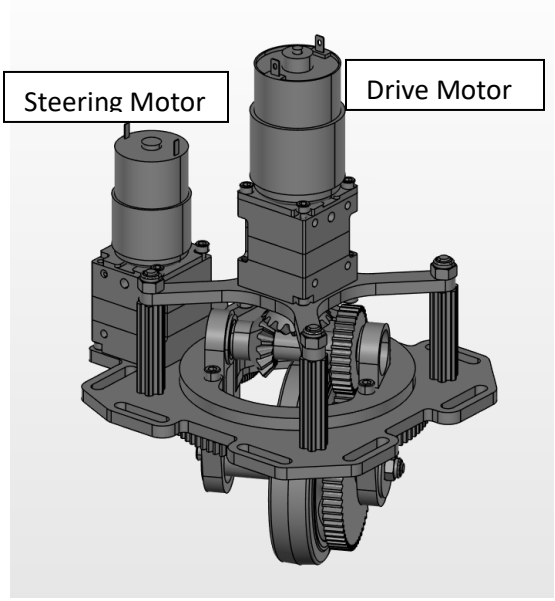
Robots in the FIRST Robotics Competition are wheeled robots. Most are some variation of a "tank drive," which means there are two sets of drive wheels (one set on the left and one set on the right). By driving the left and right wheels both in the forward or reverse direction, the robot moves forward or backwards. However, by moving the left and right wheels at different speeds (such as moving the left motor forwards and the right motor backwards) then the robot can be made to turn and face a different direction. The obvious limitation of a tank drive is that the robot can't move left or right.

Other drive systems have been tried, most notably the mecanum drive, and the swerve drive. Both allow some form of "omni-directional" movement, meaning movement in any direction.

With a swerve drive, each wheel can steer independently. If you pointed all the wheels forward and drove the wheels forward, then the robot would move forwards, but you could also point all wheels left or right to "strafe" the robot from side to side. You can also orient all the wheels into a circular arrangement and spin the robot in place. Most importantly, using some math, you can move the robot across the field in one direction (called translating) and spin the robot to face a different orientation (called rotating) at the same time. A swerve drive separates translation from rotation. This is unlike a robot with a tank drive, which can only translate in the direction it's pointing at.

## Swerve Drive Mechanical Design

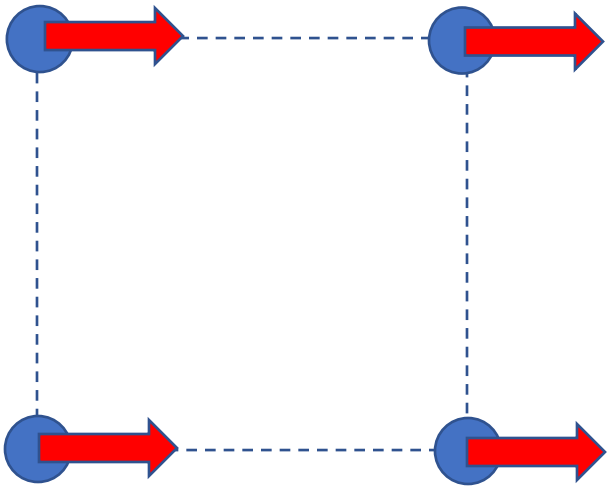
The ThunderStamps swerve drive design was copied from Team 1533's swerve design. Minor changes have been made, mostly to replace the motors with more powerful (and more efficient) NEO brushless DC motors, and to replace the steering motor with one that has a built-in encoder to read the position.



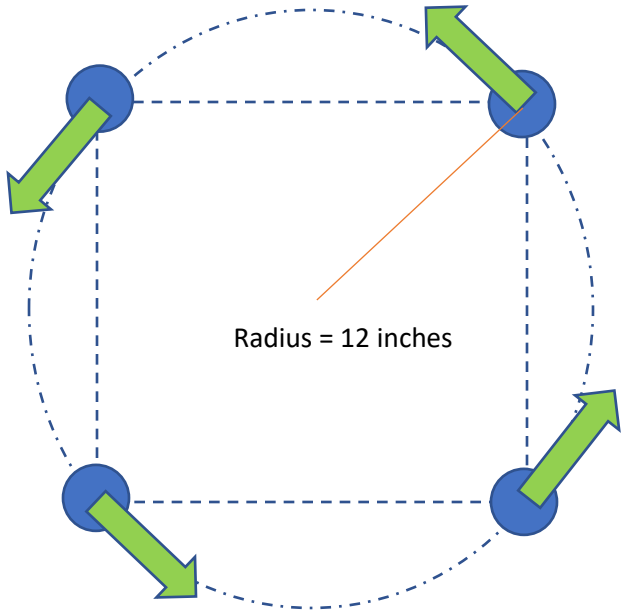
## Vector Math

To understand how a swerve drive really works, you must understand vectors. (Thankfully we only need to understand 2-dimensional vectors).

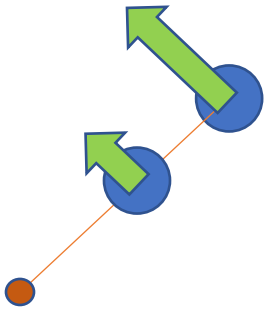
Assume for a moment that we want the robot to translate (move across the field) towards the right at 20 inches per second. That's simple enough… we want all the wheels to steer to the right and drive at the same speed (the blue circles are the wheels, and the arrows represent their speeds):



Now consider a different case where we want the robot to rotate counterclockwise at some rotational speed (let's say 90 degrees per second). The wheels must steer into a circle to accomplish this, but to understand how fast the wheels have to drive, we need to know how far they are from the center of the robot:

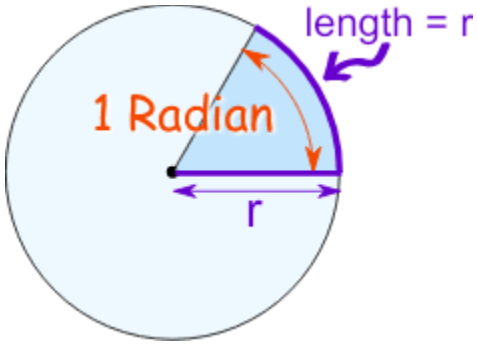


The further the wheel is from the robot, the faster they have to spin to rotate the robot at the desired speed:



Center of robot

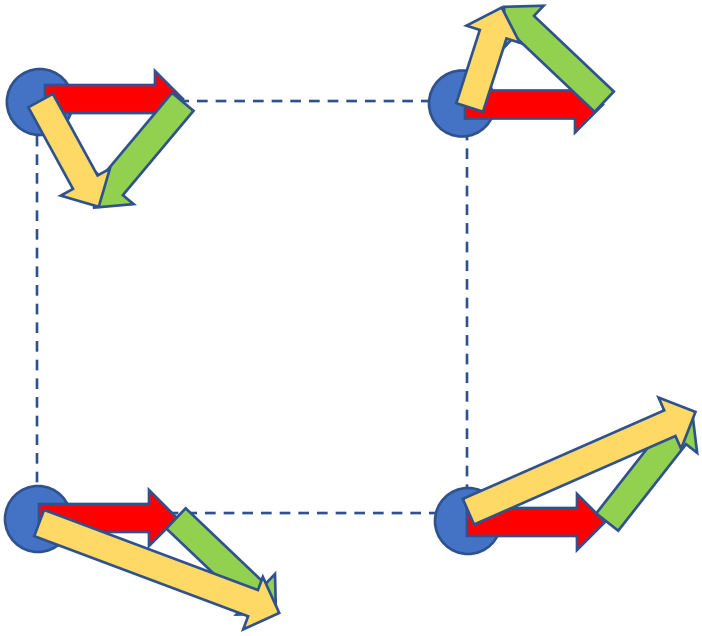
Instead of measuring the angle in degrees, it's helpful to measure the angle in radians. If you recall, there are 2π radians in a full circle. I know, that's hardly helpful, nor is knowing that there are roughly 57.29 degrees in a radian. The usefulness comes from the fact that if you travel around the circumference of a circle by a distance equal to the radius of that circle, you will have travelled one radian of angle around the circle:




So instead of thinking in degrees per second of rotating speed, from now on we're going to use radians and radians per second. So instead of saying we want to turn the robot at 90 degrees per second, let's convert that to radians per second (2π x 90 / 360 = about 1.57 rad/s). That means to rotate the robot counterclockwise at 90 degrees per second, then the wheel needs to spin fast enough to cover 1.57 radiuses of length in 1 second. Since the radius, as stated above, is 12" long in this example, then we can calculate that the wheel must drive at 1.57 x 12 = 18.84 inches per second.

Now comes the clever part. If we want the robot to both translate right at 20 inches per second, and also rotate counter-clockwise at 90 degrees per second, then each wheel needs to do both things simultaneously. Thankfully, all we need to do is add the vectors (arrows).

A vector has two components (a length, also called the magnitude) and the direction, sometimes called a heading. To add two vectors graphically, we just put them tip to tail and the result is the arrow we draw from the first tail to the last tip. The yellow arrows are the result:



You may notice that the bottom two wheels (in this case the back wheels of the robot) have to drive much faster than the front (top) wheels. That's why the yellow arrows are longer. This makes sense when you think about it. In order to rotate counter-clockwise while moving right, the rear of the robot must move to the right faster than the front of the robot.

The interesting thing about this motion is that even though all the wheels will be pointing in a different direction and driving at different speeds, the center of the robot will move towards the right while the robot also rotates at 90 degrees per second counter-clockwise.

## Robot vs. Field Orientation

It's important to realize, however, that as the robot rotates, the direction of "right" is relative to the front of the robot, so the robot will still travel in a curved path as it rotates, moving always towards its right side. This type of motion is called "robot-oriented swerve drive."

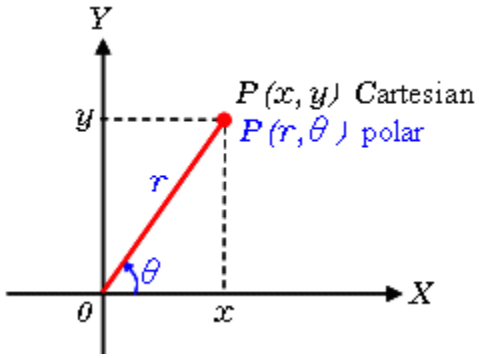
If we add a sensor to our robot called a Gyro, we can detect the direction the robot is facing on the field (relative to the direction we started when we power up, which is always field-forward). Using the Gyro sensor, we can change the direction of our translation (the red arrows) as the robot rotates on the field, and then we can translate in a straight line on the field, while rotating the robot at the same time. This type of motion is called "field-oriented swerve drive." Driving a robot with field-oriented swerve drive feels a little like magic. Programming a field-oriented swerve is the point of this document.

# Vector Arithmetic

## Polar vs. Cartesian Coordinates

As we said above, a vector is a mathematical quantity with a magnitude (length) and a direction (angle). Since each wheel on a swerve drive can steer independently, you can already see how a vector is a useful way to represent the steering direction and speed in a single mathematical quantity.

There are two ways to express the same vector. Specifying the magnitude and angle is called "polar" notation, but we can also express vectors in "cartesian" notation:




If you put a vector with its tail at the origin (0, 0) then the tip of the vector (arrow head) will be at some point on the cartesian plane. These X and Y values are called the cartesian form. The X and Y values are sometimes called the "components".

Given X and Y, we can calculate r and theta:

r = sqrt(x*x + y*y)

theta = atan2(y, x)

Given r and theta, we can calculate x and y:

x = r * cos(theta)

y = r * sin(theta)

It's easiest to add vectors using the X and Y components. Given 2 vectors (x1, y1) and (x2, y2), we can add them to get a new result vector (x, y) like this:

x = x1 + x2

y = y1 + y2

On the other hand, we may want to rotate a vector, which would require us to manipulate theta, or change the magnitude, which means manipulate "r". Therefore, having a class that allows us to manipulate vectors easily will be useful. Let's start by making a set of classes that allow us to work with 2D vectors.

## Vector2D Classes

Let's start by defining an interface that represents an abstract 2D vector. We'll call it `IVector2D`:

```
public interface IVector2D {
    double getX();
    double getY();
    double getMagnitude();
    double getAngleRadians();
}
```

We'll use this interface any time we want to accept a vector from outside our swerve drive modules (such a translation command from the main robot program when the driver moves the Xbox control stick).

Now let's make an actual Vector2D class:

```
public class Vector2D implements IVector2D {
}
```

To satisfy the interface, we'll need to add some internal member variables, and getters:

```
    private double x = 0;
    private double y = 0;
    private double magnitude = 0;
    private double angleRadians = 0;

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getMagnitude() {
        return magnitude;
    }

    public double getAngleRadians() {
        return angleRadians;
    }
```

There are 2 ways to specify a vector. The first is with cartesian coordinates, so let's create a `setXY()` method:

```
    public void setXY(double x, double y) {
        this.x = x;
        this.y = y;
        this.magnitude = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
        this.angleRadians = Math.atan2(y, x);
    }
```

The second way is with polar coordinates, so create a `setPolar()` method:

```
    public void setPolar(double magnitude, double angleRadians) {
        this.x = magnitude * Math.cos(angleRadians);
        this.y = magnitude * Math.sin(angleRadians);
        this.magnitude = magnitude;
        this.angleRadians = angleRadians;
    }
```

It's useful to have a method to set this vector as a copy of another:

```
    public void copy(IVector2D vector) {
        this.x = vector.getX();
        this.y = vector.getY();
        this.magnitude = vector.getMagnitude();
        this.angleRadians = vector.getAngleRadians();
    }
```

Next, create methods to add and subtract vectors (the result is stored in this class and the two vectors being added aren't modified):

```
    public void add(IVector2D vector1, IVector2D vector2) {
        var x = vector1.getX() + vector2.getX();
        var y = vector1.getY() + vector2.getY();
        this.setXY(x, y);
    }

    public void subtract(IVector2D vector1, IVector2D vector2) {
        var x = vector1.getX() - vector2.getX();
        var y = vector1.getY() - vector2.getY();
        this.setXY(x, y);
    }
```

Create two more methods that allow us to manipulate the magnitude and angle:

```
    public void setMagnitude(double magnitude) {
        this.setPolar(magnitude, this.angleRadians);
    }

    public void rotate(double thetaRadians) {
        this.setPolar(this.magnitude, this.angleRadians + thetaRadians);
    }
```

There is a special type of vector called a “unit vector” that always has a length of 1. It’s useful in many formulas, so create a helper function to change this vector into a unit vector:

```
    public void toUnitVector() {
        this.setMagnitude(1.0);
    }
```

Finally, rather than having to use new to create a new `Vector2D` and then call `setXY` or `setPolar`, create some static helper functions to make life a little easier:

```
    public static Vector2D FromXY(double x, double y) {
        var result = new Vector2D();
        result.setXY(x, y);
        return result;
    }

    public static Vector2D FromPolar(double magnitude, double  angleRadians) {
        var result = new Vector2D();
        result.setPolar(magnitude, angleRadians);
        return result;
    }
```

That allows you to write code like:

```
Vector2D v = Vector2D.FromPolar(speed_in_s, direction_rad);
```

## Unit Testing

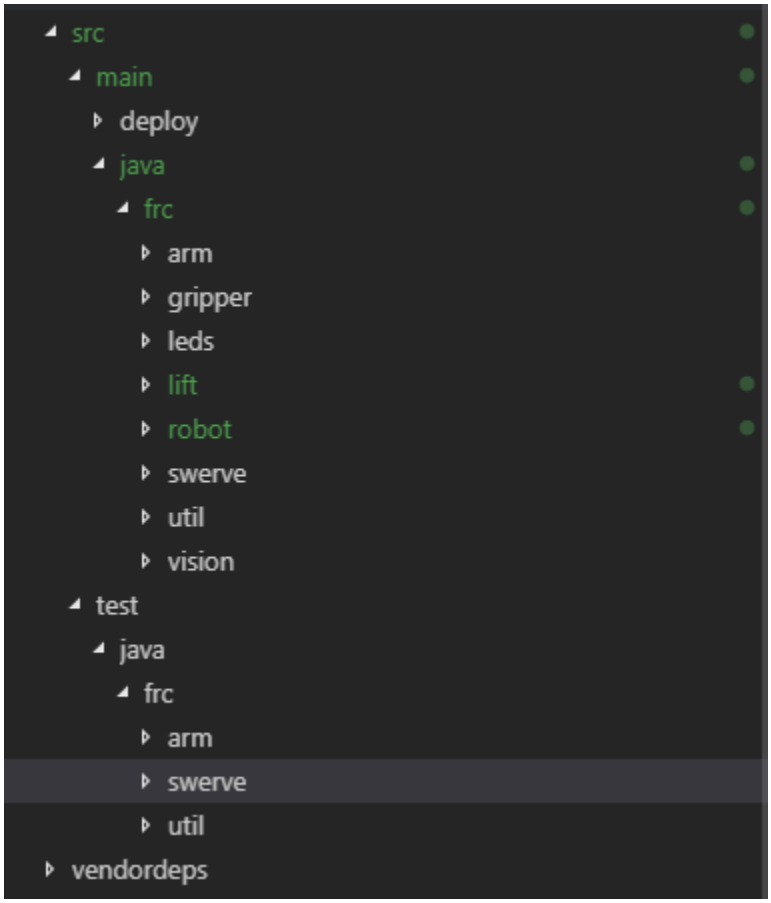
When programming a robot, or any machine for that matter, you're often programming it while it's being built. That means you can't directly test your code on the robot itself until after it's assembled. Wouldn't it be nice to debug your code before someone hands you a robot and looks at you expectantly?

Furthermore, once you have your code debugged and you want to make a change, wouldn't it be nice to be able to prove you didn't break anything you already wrote? This is especially important if re-testing meant you needed access to the robot again.

This is why we create unit tests. A unit test is a small program you write to test one small part of your code. You give it inputs, and check that you get the correct outputs. Then you write unit tests to test all the parts of your code independently, which gives you lots of confidence that your code is *mostly* working correctly when you download it the first time.

The VSCode environment has support for unit tests. Just go into the extensions and install the Java Test Runner extension (WPILib suggests that you install the Java Extension Pack which comes with the Java Test Runner extension).

You normally organize your unit tests into a parallel set of classes named after the classes they're testing. So, if you had a `SwerveModule` class, you'd put the tests for that in a `TestSwerveModule` class, and it would go into a separate source tree (typically in `src/test` rather than `src/main`):



## Using Junit

When you're creating a new unit test class (such as `TestSwerveModule`) you need to import some references at the top of the file:

```
import org.junit.Test;
import static org.junit.Assert.*;
```

That will import the references you need for your unit test framework (JUnit).

Then you can write methods and mark them as tests with the `@Test` attribute before them:

```
public class TestSwerveModule {

    @Test
    public void One_and_one_make_two() {
        assertEquals(1+1, 2);
    }

}
```

That's not a very useful test for us, but it shows how they work. The test runner will run all the methods marked with the `@Test` attribute, and check if they pass or fail. They can fail if an exception is thrown, or if one of your assertions fails. In this case we're asserting that `1+1` is equal to `2`. Since they are equal, the assertion will pass, and the test will pass.

## Testing Vector2D

Let's say we wanted to write some tests for our `Vector2D` class, from earlier. We should test the static methods `Vector2D.FromXY()` and `Vector2D.FromPolar()`. We might write something like this:

```
public class TestVector2D {
    @Test
    public void Can_construct_FromXY() {
        Vector2D test = Vector2D.FromXY(3, 4);

        assertEquals(3, test.getX(), 0.0);
        assertEquals(4, test.getY(), 0.0);
        assertEquals(5, test.getMagnitude(), 0.0);
        assertEquals(53.13, Math.toDegrees(test.getAngleRadians()), 0.02);
    }
    @Test
    public void Can_construct_FromPolar() {
        Vector2D test = Vector2D.FromPolar(Math.sqrt(2.0), Math.toRadians(45.0));

        assertEquals(1.0, test.getX(), 0.01);
        assertEquals(1.0, test.getY(), 0.01);
        assertEquals(Math.sqrt(2.0), test.getMagnitude(), 0.0);
        assertEquals(Math.toRadians(45.0), test.getAngleRadians(), 0.0);
    }
}
```

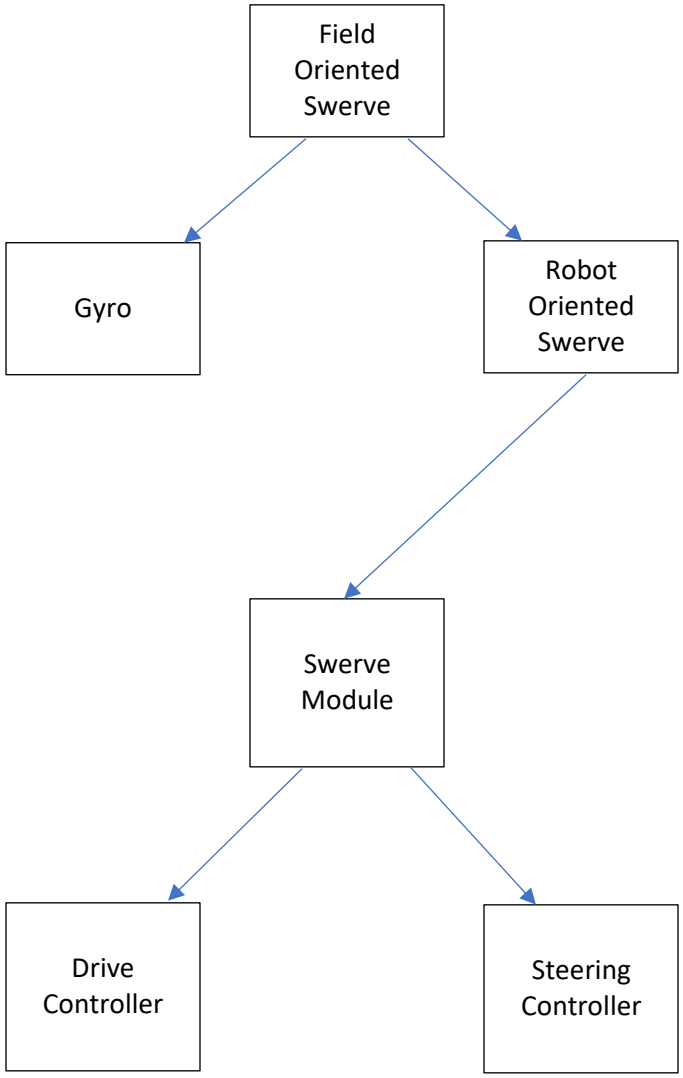
Of course, we should probably test more than one set of values for each function. The simplest way is to just do something like this (create a helper method and call it multiple times):

```
    @Test
    public void Can_construct_FromXY() {
        can_construct_fromXY(3, 4, 5, 53.13);
        can_construct_fromXY(4, 3, 5, 36.87);
        can_construct_fromXY(1, 1, Math.sqrt(2.0), 45);
    }

    private void can_construct_fromXY(
            double x, double y, double magnitude, double angleDegrees) {
        Vector2D test = Vector2D.FromXY(x, y);

        assertEquals(x, test.getX(), 0.0);
        assertEquals(y, test.getY(), 0.0);
        assertEquals(magnitude, test.getMagnitude(), 0.0);
        assertEquals(angleDegrees, Math.toDegrees(test.getAngleRadians()), 0.02);
    }
```

# Architecture

Software architecture refers to the overall layout of the software modules, and how they interact.

## Diagram



## Modules

What follows is a brief description of what each module does. In object-oriented software, such as Java, we often try to follow the "single responsibility principle." This means each module (or class in this case) should perform a single conceptual function. Don't get overly hung up on this rule, as we often violate it out of practical concerns, but it's a guiding principle.

Mostly we're trying to break down the software into pieces that we can easily understand what they have to do, and most importantly, easily test them.

The modules are listed here from the "bottom up."

### Drive Controller

The drive controller is responsible for all the communication with the motor controller that drives the wheel. In this case, we expect it to be a NEO motor with a SparkMAX motor controller. One important function of this class is to enclose all the functionality in place where we could easily replace it with another equivalent class if we had to change motor controllers. Therefore, anything that is specific to the motor controller we've chosen should be in this class, and code that would be the same regardless of the motor controller we chose should not be in this class.

Since this class deals directly with the motor controller, this isn't a class that we can unit test. In order to test it, we would have to connect it up to a real motor controller and spin the motor. Since this class is fundamentally difficult to unit test, we want to make it as simple as possible.

This class will include all initialization code for the SparkMAX motor controller, and its primary function is to convert speed and direction commands from the Swerve Module class into commands for the SparkMAX motor controller.

### Steering Controller

The steering controller is responsible for all the communication with the motor controller that drives the steering motor. In the 2019 season, this was a Talon SRX motor controller with a PG27 gearmotor. However, this is likely to change in 2020, either to a Falcon 500, or a NEO with SparkMAX. Like the Drive Controller, this class needs to include all the code that's specific to the motor controller we've chosen, but nothing extra because it's inherently untestable with unit tests.

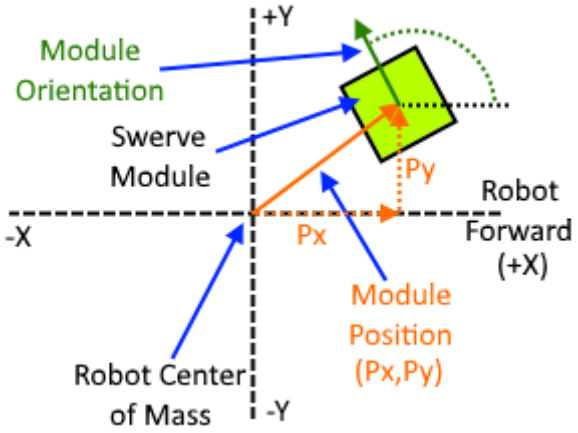
The Steering Controller class will initialize the motor controller, handle the logic to "home" or "reference" the position to a known angle, and convert the commands from the Swerve Module (a steering angle command) into commands for the chosen motor controller.

Additionally, the Steering Controller class must report on the status, such as whether or not the referencing step is complete, and the current steering angle.

## Swerve Module

The Swerve Module class contains the code for one swerve wheel module. When it's constructed, we will provide it with its "dependencies" in its constructor:

- A reference to a Drive Controller
- A reference to a Steering Controller
- An X & Y **module position**, in inches, relative to the center of mass of the robot
- A **module orientation** (in radians) relative to the front of the robot
    - When steering controller is at zero angle, and drive motor running forward




The Swerve Module functionality will be tested by unit tests. That means we want to be able to test the code inside the Swerve Module class without actually having to hook it up to a robot. During testing, we won't provide it with a reference to an actual Drive Controller or a Steering Controller. Instead, we'll create interfaces (`IDriveController` and `ISteeringController`). The Swerve Module will require objects implementing these interfaces passed into its constructor. For testing we will create fake implementations of these interfaces called "Mocks" (`MockDriveController` and `MockSteeringController`). The Swerve Module class won't know the difference, but it'll allow us to test it without needing to hook it up to motors. In fact, the tests can run on the Windows programming PC, rather than on the RoboRIO.

The Swerve Module class's main responsibility is to accept a velocity command of direction and speed (in radians an inches/second) from the Robot Oriented Swerve Module and command the Drive Controller and Steering Controller.

Remember that the wheel can spin in either direction. If the wheel is currently pointed in the backwards direction (towards the back of the robot) and the command from the Robot Oriented Swerve class is to drive forwards, the Swerve Module class will be smart enough to tell the Steering Controller to maintain the current direction (backwards) and tell the Drive Controller to spin the wheel in reverse. That is, the Swerve Module is responsible for calculating the *shortest path* to steer the module onto the correct heading, and then choose the appropriate direction for the Drive Controller to spin the wheel.

## Robot Oriented Swerve

The Robot Oriented Swerve class will be constructed with a collection of Swerve Modules as its dependencies. Many online examples of swerve drives will assume 4 swerve modules in a rectangular layout, but we won't assume any particular layout or number of modules. One could imagine a tricycle robot with 3 swerve modules, etc.

Each Swerve Module will report its position (X and Y) relative to the center of mass of the robot. The responsibility of the Robot Oriented Swerve Module is to accept robot translation and rotation commands (either from the Xbox controller or from the Field Oriented Swerve module) and convert those commands into velocity (speed and direction) commands for each Swerve Module using the vector arithmetic explained earlier.

Each Swerve Module will also provide its maximum speed. The Robot Oriented Swerve module is responsible for checking that no velocity command will exceed any swerve module's maximum speed, and if it does, then it will scale all velocity commands down accordingly.

The Robot Oriented Swerve class will assume rotation around the center of mass of the robot. Rotation about an arbitrary point is possible, but is outside the scope of this document.

To facilitate unit testing, the Robot Oriented Swerve class will expect a collection of interfaces (`ISwerveModule`) rather than the actual Swerve Module class. We will construct a fake class (`MockSwerveModule`) that implements the `ISwerveModule` interface so we can test the Robot Oriented Swerve class in isolation.

## Gyro

In order to create a Field Oriented Swerve module, the robot needs to know its current orientation on the field. To do this, we use a Gyro sensor (also called an inertial measurement unit). While the sensor can report on acceleration, we really only care about the heading reading, which tells us the robot direction relative to the direction we were facing when the sensor powered on.

Since we want to test the Field Oriented Swerve class without hooking it up to the Gyro board, we need a way to provide a fake Gyro sensor for testing. Therefore, this Gyro class exists just to be a "wrapper" for the Gyro sensor, so we can create a `MockGyro` class for testing the Field Oriented Swerve class.

## Field Oriented Swerve

When we construct the Field Oriented Swerve class, we'll provide it with dependencies in the constructor:

- A Robot Oriented Swerve object
- A Gyro object
- Maximum robot speed (inches per second)
- Maximum acceleration (inches per second per second)
- Maximum rotation rate (radians per second)
- Scan time (seconds)

Like the previous classes, we want to be able to unit test the Field Oriented Swerve class, so during testing we'll inject a `MockRobotOrientedSwerve` class, and a `MockGyro` class.

During operation, the Field Oriented Swerve module takes many continuous inputs:

- Robot oriented translation command vector (in/s and radians)
- Rotation command (rad/s)
- Field oriented translation command vector (in/s and radians relative to field)
- Heading command (radians relative to field)
- Gyro input (robot heading relative to field, gyro status enabled/disabled)

The Field Oriented Swerve module deals with two components of motion: translation and rotation.

### *Translation*

The Field Oriented Swerve converts the robot-oriented and field-oriented translation commands into a single field-oriented translation command vector. It first applies a maximum speed, and then applies an acceleration limit to the rate of change of this command vector. We apply an acceleration limit here to prevent the robot from tipping over. It then converts this limited field-oriented translation vector back into a robot-oriented command, and sends that to the Robot Oriented Swerve module. The translation between Robot and Field orientation requires the Gyro input.

### *Rotation*

The Field Oriented Swerve takes the Heading command and compares that to the Gyro heading to determine how far it needs to turn the robot to face the desired direction. It then applies a PID loop to figure out how fast it should be trying to rotate the robot right now. Then it adds that to the Rotation command input, and sends the result to the Robot Oriented Swerve module.

## Interfaces

The following interfaces define each module:

### IDriveController

```
public interface IDriveController {

    // Call this to put the drive motor into speed-controlled
    // mode and attempt to run at the given target speed.
    void executeVelocityMode(double targetMotorRpm);

    // Actual motor speed
    double getMotorRpm();

    // Call this to put the drive motor into position-controlled
    // mode and move to the target position (units of motor revolutions)
    void executePositionMode(double targetMotorRev);

    // Actual motor distance travelled
    double getMotorRevCount();

    // A parameter of the swerve module, this is the gear ratio
    // of motor turns for full revolution of the wheel (~8.0:1.0)
    double getMotorRevsPerWheelRev();

    // A parameter of the swerve module, this is the wheel size
    double getWheelDiameterInches();

    // A parameter of the swerve drive, max motor speed in RPM
    double getMaxSpeedRpm();
}
```

### ISteeringController

```
public interface ISteeringController {

    // The steering controller will automatically "home" when it starts
    // This property reports if the homing sequence is complete.
    boolean isHomed();

    // Call this to move to the target position (units of motor revolutions)
    void executePositionMode(double targetMotorRev);

    // Actual motor distance in full motor revolutions relative to
    // reference (home) position
    double getMotorRevCount();

    // A parameter of the swerve drive, this is the gear ratio
    // of motor turns for full revolution of the steering module
    double getMotorRevsPerSteeringRev();
}
```

### ISwerveModule

```
public interface ISwerveModule {

    // This status property echoes the property
    // of ISteeringController.isHomed()
    boolean isHomed();

    // Call this to execute the module in velocity mode with the given command
    // The command is a vector with magnitude in in/sec and direction in radians
    // relative to forward being the front of the robot
    void executeVelocityMode(IVector2D velocityCommand_in_s_rad);

    // Returns the actual swerve module velocity in inches per second
    // with direction in radians relative to the front of the robot
    IVector2D getCurrentVelocity_in_s_rad();

    // Starts a precise module move of a given distance in inches,
    // in a given direction relative to robot forward, in radians
    IPreciseModuleMove startPreciseMove(IVector2D positionCommand_in_rad);
    boolean isAligned(IPreciseModuleMove preciseModuleMove);
    void executePreciseMove(IPreciseModuleMove preciseMove, boolean allAligned);

    // Returns the swerve module location relative to the center of mass
    IVector2D getModulePos_in();

    // Returns the swerve module orientation relative to robot forward
    double getModuleOrientation_rad();

    // Returns the maximum wheel speed allowed in inches per second
    double getMaxSpeed_in_s();
}
```

### IRobotOrientedSwerve

```
public interface IRobotOrientedSwerve {

    // Call this to execute the swerve drive in velocity mode,
    // providing a translation vector and a rotation rate.
    // Optionally provide a vector to specify the center of rotation.
    void execute (
        IVector2D translationCommand_in_s_rad,
        double targetRotationRate_rad_s);

}
```

### IGyro

```
public interface IGyro {

    // Returns the heading of the robot relative to the field
    double getFieldOrientation_rad();

    // Returns the rate of change of heading relative to field
    double getRotationRate_rad_s();

    // Set the Field Orientation to zero (useful during practice)
    void resetOffsetToZero_rad();

    // Modify the Field Orientation by a small adjustment
    // Typically hooked up to Xbox buttons for small heading corrections
    void adjustOffset_rad(double adjustment_rad);

    // If the gyro malfunctions, we can manually disable it
    // which will put us into robot-oriented control only
    boolean getEnabled();
    void setEnabled(boolean enabled);
}
```

### IFieldOrientedSwerve

```
public interface IFieldOrientedSwerve {

    // Call this to execute the swerve drive in velocity mode,
    // providing robot- and field-oriented translation vectors and
    // a rotation rate.
    void execute(
        IVector2D fieldOrientedTranslationCommand_in_s_rad,
        double fieldOrientedHeadingCommand_rad,
        IVector2D robotOrientedTranslationCommand_in_s_rad,
        double targetRotationRate_rad_s);
}
```

# Motor Controller Classes

Now we're going to start writing code. The motor controller classes (`DriveController` and `SteeringController`) are based on example code from the vendor's website, plus experimentation with the motor and controller plugged into a RoboRIO for testing.

## DriveController

Start by creating a new `DriveController` class:

```
package frc.swerve;

import com.revrobotics.*;
import com.revrobotics.CANPIDController.*;
import com.revrobotics.CANSparkMaxLowLevel.*;

public class DriveController implements IDriveController {

}
```

In the constructor, we need to pass in one parameter: the CANBUS address of the motor controller. With that information, we can create the motor controller objects and store them in class variables. We're going to assume we're using the same motor controller and motor from last year (the SparkMAX controller and NEO brushless DC motor):

```
    private final int sparkMaxCanBusAddress;
    private final CANSparkMax sparkMax;
    private final CANEncoder encoder;
    private final CANPIDController pidController;

    public DriveController(int sparkMaxCanBusAddress) {
        this.sparkMaxCanBusAddress = sparkMaxCanBusAddress;
        this.sparkMax = new CANSparkMax(
            this.sparkMaxCanBusAddress,
            MotorType.kBrushless);
        this.encoder = this.sparkMax.getEncoder();
        this.pidController = this.sparkMax.getPIDController();
    }
```

Following that, still in the constructor, we need to initialize all the parameters of the motor controller:

```
        double kP, kI, kD, kIz, kFF;
        double SPARKMAX_CRUISE_VELOCITY_RPM = 5000.0;

        // PID Coefficients
        kP = 7e-5;
        kI = 1.8e-6;
        kD = 0.0;
        kIz = 200;
        kFF = 1/SPARKMAX_CRUISE_VELOCITY_RPM * 5.0/6.0;

        this.pidController.setP(kP);
        this.pidController.setI(kI);
        this.pidController.setD(kD);
        this.pidController.setIZone(kIz);
        this.pidController.setFF(kFF);
        this.pidController.setOutputRange(-1,1);
        this.sparkMax.setOpenLoopRampRate(0.1);
        this.sparkMax.setClosedLoopRampRate(0.1);

        this.pidController.setSmartMotionMaxVelocity(5000.0, 0);
        this.pidController.setSmartMotionAccelStrategy(
            AccelStrategy.kTrapezoidal,
            0);
        this.pidController.setSmartMotionMinOutputVelocity(0.0, 0);
        this.pidController.setSmartMotionMaxAccel(2500.0, 0);
        this.pidController.setSmartMotionAllowedClosedLoopError(0.2, 0);

        this.sparkMax.setCANTimeout(200);
```

To see where these values were obtained, reference [the examples from REV robotics](#).

Now start writing the methods that implement the `IDriveController` interface. Start with the methods that just return parameters of the swerve drive modules:

```
    @Override
    public double getMotorRevsPerWheelRev() { return 8.0; }

    @Override
    public double getWheelDiameterInches() { return 4.0; }

    @Override
    public double getMaxSpeedRpm() { return 5000.0; }
```

Getting the motor speed and position is simply a call to the encoder object:

```
    @Override
    public double getMotorRpm() {
        return encoder.getVelocity();
    }

    @Override
    public double getMotorRevCount() {
        return encoder.getPosition();
    }
```

If this seems overly simple, remember that the purpose of the `DriveController` class is to be a simple “wrapper” around the motor controller with no more code in it than absolutely necessary, because we can’t easily test this code without running it on the RobotRIO with a motor attached.

Finally, write the code to run the motor controller in velocity or position mode:

```
    @Override
    public void executeVelocityMode(double targetMotorRpm) {
        CANError err = pidController.setReference(
            targetMotorRpm, ControlType.kVelocity);
        if (err != CANError.kOk) {
            System.out.println("Swerve Drive MC at CanBus "
            + this.sparkMaxCanBusAddress + " error: " + err.toString());
        }
    }

    @Override
    public void executePositionMode(double targetMotorRev) {
        CANError err = pidController.setReference(
            targetMotorRev, ControlType.kSmartMotion);
        if (err != CANError.kOk) {
            System.out.println("Swerve Drive MC at CanBus "
            + this.sparkMaxCanBusAddress + " error: " + err.toString());
        }
    }
```

As you can see, we’re using the built-in functions of the motor controller to run the motor at a given target speed, or to move to a target position.

## SteeringController

At this point, we don't know which motor controller we're going to use for steering yet, except that it likely won't be the Talon SRX and PG27 gear motor from last year. It may be the Falcon 500, or the SparkMAX with NEO motors. Skip this class for now.

One of the nice things about using interfaces and "mock" classes for unit testing is that we can continue programming the swerve drive without knowing this low-level detail yet.

## Gyro Class

Last year we used the navX MXP gyro board, and we'll assume we can use it again this year. It's important to understand that if, for some reason, we can't use the same gyro board, we just need to create a new class that implements the same `IGyro` interface and we can use that instead, and the rest of our swerve drive code won't need to change.

### SwerveUtil

It's often necessary to create "utility" functions which are functions that aren't really tied to any classes, because they don't require any saved state to operate. A good example of this is the `Math` class in Java. We can call `Math.toRadians()` to convert from degrees to radians, but we don't want to create a new class to call it. However, Java requires us to put all code into a class, so we have to create a utility class. For our purposes, let's create a new class called `SwerveUtil` for our helper functions:

```
package frc.swerve;

public class SwerveUtil {

}
```

To write the gyro class, we're going to need a helper function that can take any angle in radians and return the equivalent angle limited from -pi to pi (this would be equivalent to limiting the angle to -180 to 180 in degrees). Create the method stub like this:

```
    public static double limitAngleFromNegativePItoPI_rad(double angle_rad) {

    }
```

To get this function right, we need to write unit tests for it, that test the following inputs and results:

| angle_rad | result |
|---|---|
| 0 | 0 |
| pi/2 | pi/2 |
| pi | Pi (or -pi) |
| 3*pi/2 | -pi/2 |
| -pi/2 | -pi/2 |
| -pi | -pi (or pi) |
| -3*pi/2 | pi/2 |
| -2*pi | 0 |
| 2*pi | 0 |
| 5*pi/2 | pi/2 |
| -5*pi/2 | -pi/2 |

Now, write the function. You should be using the modulo function (%) and the constant `Math.PI`.

## NavXMxpGyro

Start by creating an empty class:

```
package frc.swerve;

import com.kauailabs.navx.frc.AHRS;
import edu.wpi.first.wpilibj.SPI;

public class NavXMxpGyro implements IGyro {

}
```

In the constructor, we simply need to create an instance of the class that the vendor provides us to use the gyro board:

```
    private final AHRS gyro;

    public NavXMxpGyro() {
        this.gyro = new AHRS(SPI.Port.kMXP);
    }
```

Start by writing code to implement the enabled property of the `IGyro` interface:

```
    private boolean enabled = true;

    @Override
    public boolean getEnabled() { return this.enabled; }

    @Override
    public void setEnabled(boolean enabled) { this.enabled = enabled; }
```

For the rotation rate in radians per second, we can read the rotation rate directly from the gyro module (in degrees per second) and convert that to radians per second:

```
    @Override
    public double getRotationRate_rad_s() {
        return Math.toRadians(gyro.getRate());
    }
```

Finally, implement the code to get the field orientation, in radians. The gyro board will give us the current orientation in degrees, relative to the orientation when we powered up. We can reset this power-up angle by calling `gyro.reset()` and `gyro.resetYaw()`, but we also want to be able to make small adjustments to the heading reference over time. We do that by introducing a class variable called `offsetToZero_rad`.

```
    private double offsetToZero_rad = 0.0;

    @Override
    public double getFieldOrientation_rad() {
        double rawFieldOrientation_rad =
        Math.toRadians(gyro.getYaw())
        - offsetToZero_rad;
        return SwerveUtil.limitAngleFromNegativePItoPI_rad(
            rawFieldOrientation_rad);
    }

    @Override
    public void resetOffsetToZero_rad() {
        gyro.reset();
        gyro.zeroYaw();
        this.offsetToZero_rad = 0.0;
    }

    @Override
    public void adjustOffset_rad(double adjustment_rad) {
        this.offsetToZero_rad += adjustment_rad;
    }
```

## SwerveModule Class

Start with an empty `SwerveModule` class:

```
package frc.swerve;

public class SwerveModule implements ISwerveModule {

}
```

To construct this class, we need a reference to a steering controller, a drive controller, the location of this module relative to the robot center of mass, and the orientation of this module relative to the robot forward direction:

```
    private final ISteeringController steeringController;
    private final IDriveController driveController;
    // position of this swerve module relative to the center of mass
    private final IVector2D modulePosition_in;
    // orientation of this module relative to robot forward
    private final double moduleOrientation_rad;

    public SwerveModule(
            ISteeringController steeringController,
            IDriveController driveController,
            IVector2D modulePosition_in,
            double moduleOrientation_rad) {
        this.steeringController = steeringController;
        this.driveController = driveController;
        this.modulePosition_in = modulePosition_in;
        this.moduleOrientation_rad = moduleOrientation_rad;
    }
```

### Delegated Properties

To start implementing the `ISwerveModule` interface, fill in the following simple methods that just return the data we already have (tests aren't needed if we're just returning existing data):

```
    @Override
    public boolean isHomed() { return this.steeringController.isHomed(); }

    @Override
    public IVector2D getModulePos_in() { return this.modulePosition_in; }

    @Override
    public double getModuleOrientation_rad()
        { return this.moduleOrientation_rad; }
```

## Unit Testing the Swerve Module with Mocks

To create unit tests for the SwerveModule class, we need to create some mock classes. Remember that your mock classes go in your src/test folder tree, not in your src/main tree.

### MockDriveController

Start by creating an empty `MockDriveController` class:

```
package frc.swerve;

public class MockDriveController implements IDriveController {

}
```

When you want to mock a simple get method, you have to provide an internal value to return, and a way for the user of your mock (your test) to set the value you want it to return when called. Something like this is the simplest and easiest way to do this:

```
    private double maxSpeedRpm = 5000.0;

    public void setMaxSpeedRpm(double maxSpeedRpm) {
        this.maxSpeedRpm = maxSpeedRpm;
    }

    @Override
    public double getMaxSpeedRpm() {
        return this.maxSpeedRpm;
    }
```

First, we create an internal variable called `maxSpeedRpm`, and give it a sensible default value, like 5000 RPM. We also create a setter called `setMaxSpeedRpm()` to set the value in the test code. Finally, we implement the actual method from the interface called `getMaxSpeedRpm()` and return the value.

We will use the mock class like this:

```
    @Test
    public void Some_test() {
        MockDriveController driveController = new MockDriveController();
        driveController.setMaxSpeedRpm(3450.0);

        SwerveModule swerveModule = new SwerveModule(
            driveController,
            ...);
    }
```

So, let's finish writing the `MockDriveController`:

```
    private double targetMotorRpm = 0.0;

    @Override
    public void executeVelocityMode(double targetMotorRpm) {
        this.targetMotorRpm = targetMotorRpm;
    }

    public double getTargetMotorRpm() {
        return this.targetMotorRpm;
    }

    private double targetMotorRev = 0.0;

    @Override
    public void executePositionMode(double targetMotorRev) {
        this.targetMotorRev = targetMotorRev;
    }

    public double getTargetMotorRev() {
        return this.targetMotorRev;
    }

    private double motorRpm = 0.0;

    public void setMotorRpm(double motorRpm) {
        this.motorRpm = motorRpm;
    }

    @Override
    public double getMotorRpm() {
        return this.motorRpm;
    }

    private double motorRevCount = 0.0;

    public void setMotorRevCount(double motorRevCount) {
        this.motorRevCount = motorRevCount;
    }

    @Override
    public double getMotorRevCount() {
        return this.motorRevCount;
    }
```

```
    private double motorRevsPerWheelRev = 8.0;

    public void setMotorRevsPerWheelRev(double motorRevsPerWheelRev) {
        this.motorRevsPerWheelRev = motorRevsPerWheelRev;
    }

    @Override
    public double getMotorRevsPerWheelRev() {
        return this.motorRevsPerWheelRev;
    }

    private double wheelDiameterInches = 4.0;

    public void setWheelDiameterInches(double wheelDiameterInches) {
        this.wheelDiameterInches = wheelDiameterInches;
    }

    @Override
    public double getWheelDiameterInches() {
        return this.wheelDiameterInches;
    }

    private double maxSpeedRpm = 5000.0;

    public void setMaxSpeedRpm(double maxSpeedRpm) {
        this.maxSpeedRpm = maxSpeedRpm;
    }

    @Override
    public double getMaxSpeedRpm() {
        return this.maxSpeedRpm;
    }
```

### MockSteeringController

Similar to `MockDriveController`, write the `MockSteeringController`:

```
package frc.swerve;

public class MockSteeringController implements ISteeringController {

    private boolean homed = true;

    public void setHomed(boolean homed) {
        this.homed = homed;
    }

    @Override
    public boolean isHomed() {
        return this.homed;
    }

    private double targetMotorRev = 0.0;

    @Override
    public void setTargetMotorRev(double targetMotorRev) {
        this.targetMotorRev = targetMotorRev;
    }

    public double getTargetMotorRev() {
        return this.targetMotorRev;
    }

    private double motorRevCount = 0.0;

    public void setMotorRevCount(double motorRevCount) {
        this.motorRevCount = motorRevCount;
    }

    @Override
    public double getMotorRevCount() {
        return this.motorRevCount;
    }

    private double motorRevsPerSteeringRev = 9.0;

    public void setMotorRevsPerSteeringRev(
            double motorRevsPerSteeringRev) {
        this.motorRevsPerSteeringRev = motorRevsPerSteeringRev;
```

```
    }

    @Override
    public double getMotorRevsPerSteeringRev() {
        return this.motorRevsPerSteeringRev;
    }
}
```

## Max Speed

Part of the `ISwerveModule` interface is to return the max speed of the module in inches per second:

```
    @Override
    public double getMaxSpeed_in_s() {

    }
```

To compute this, we need to use the parameters provided by the drive controller. The reason we don't have the drive controller class return this directly is because we can't easily test the drive controller, so we want to put the math in the swerve module so we can write unit tests.

The drive controller provides 3 methods that we'll use to compute max speed:

1. `getMaxSpeedRpm()`
2. `getWheelDiameterInches()`
3. `getMotorRevsPerWheelRev()`

Every time the motor spins one rotation, the wheel spins fewer rotations due to the gear ratio, and the `getMotorRevsPerWheelRev()` method gives us that ratio. Once we have wheel rotations per minute, we need to convert that into inches per minute using the formula for the circumference of a circle (pi * wheel diameter in inches), and finally, we need to convert from inches per minute to inches per second.

- max wheel speed in rpm is max motor speed in rpm / motor revs per wheel rev
- max speed in inches per minute is max wheel speed in rpm * PI * diameter in inches
- max speed in inches per second is max speed in inches per minute / 60.0

Note that you can safely assume all three parameters used in the calculation don't change while the program is running, so you could compute the max speed in inches per second in the constructor, save it in a class variable, and always return the saved value.

To test that this function is working correctly, we need to create unit tests. Create a class in your test folder tree called `TestSwerveModule`. Add a unit test to validate that `getMaxSpeed_in_s()` returns correct values:

```
package frc.swerve;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestSwerveModule {
    @Test
    public void Test_getMaxSpeed_in_s() {
        test_getMaxSpeed_in_s(60.0, 1.0, 1.0/Math.PI, 1.0);
        test_getMaxSpeed_in_s(5000.0, 8.0, 4.0, 130.9);
        // check a zero gear ratio returns 0 rather than divide by 0
        test_getMaxSpeed_in_s(5000.0, 0.0, 4.0, 0.0);
    }

    private void test_getMaxSpeed_in_s(
            double maxMotorRpm,
            double motorRevsPerWheelRev,
            double wheelDiameterInches,
            double expected) {
        MockDriveController driveController = new MockDriveController();
        driveController.setMaxSpeedRpm(maxMotorRpm);
        driveController.setMotorRevsPerWheelRev(motorRevsPerWheelRev);
        driveController.setWheelDiameterInches(wheelDiameterInches);

        SwerveModule test = makeSwerveModule(driveController);
        double actual = test.getMaxSpeed_in_s();
        assertEquals(expected, actual, 0.01);
    }

    private SwerveModule makeSwerveModule(
            IDriveController driveController) {
        ISteeringController steeringController = new MockSteeringController();
        IVector2D modulePosition = Vector2D.FromPolar(1.0, 0.0);
        double moduleOrientation = 0.0;
        return new SwerveModule(
            steeringController,
            driveController,
            modulePosition,
            moduleOrientation);
    }
}
```

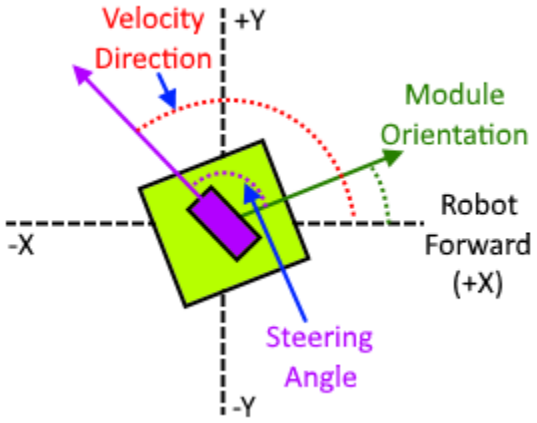
## Get Current Velocity

The method for returning the velocity of this swerve module looks like this:

```
    @Override
    public IVector2D getCurrentVelocity_in_s_rad() {

    }
```

As any physics teacher will tell you, “velocity” is “speed” and “direction.” In our case, speed is in inches per second, and direction is in radians relative to robot forward.

So, we need to return a vector (magnitude and direction). The magnitude is the speed from the `DriveController` module, and the direction is from the `SteeringController` module. The direction is supposed to be relative to the front of the robot, so we have to take the swerve module orientation into account. Finally, we want to return a positive magnitude for our vector, so we should take the wheel driving direction (forward or backward) into account.



## Calculating Velocity Direction

In the diagram above, robot forward is to the right. Our steering controller will report the number of motor revolutions from zero (`getMotorRevCount()`), where zero is when the wheel is pointed towards the Module Orientation (green) vector. We have to take the number of steering motor revolutions, and divide that by the gear ratio given to us by `getMotorRevsPerSteeringRev()` to find how many steering revolutions (full circles) we are from the green vector. Then we need to convert those full steering revolutions into radians by multiplying by 2 x PI radians per full circle.

Note that the swerve module is capable of continuous rotation. For a concrete example, suppose the `SteeringController` reports a position of 36.0 and it has a gear ratio of 10.0:1.0, then we can calculate that the wheel is oriented 3.6 full steering revolutions from the green vector. Our direction needs to be in radians, and must be limited to a value from -PI to PI (negative half a rotation to positive half a rotation). We can ignore full rotations, so 3.6 rotations is the same as 0.6 rotations, but 0.6 is still more than half a rotation. The correct answer is -0.4 rotations, which is -144 degrees, or -0.8 PI radians. That would be the *Steering Angle* (purple).

Finally, we can add in the Module Orientation (green angle) to get the Velocity Direction (red angle). This is the direction that the wheel is pointing relative to the front of the robot.

We then look at the motor speed from the `DriveController`, and if the speed is negative (the motor is rotating in reverse) then we want to add (or subtract) half a circle (PI radians) from the Velocity Direction because the module is actually driving in the opposite direction.

## Calculating Speed (Velocity Magnitude)

To determine the speed (velocity magnitude), the `SwerveModule` reads the motor speed in RPM from the `DriveController` and still needs to use the gear ratio and wheel diameter parameters of the `DriveController` to calculate the magnitude of the speed in inches per second. It also needs to read the steering motor position from the `SteeringController` and use the steering gear ratio to calculate the actual steering angle.

### Unit Test Helper Function

Inside our `TestSwerveModule` class, build a method just for testing the `getCurrentVelocity_in_s_rad()` method:

```
private void test_getCurrentVelocity_in_s_rad(
        double actualDriveMotorSpeedRpm,
        double driveControllerMotorRevsPerWheelRev,
        double driveControllerWheelDiameterInches,
        double actualSteeringMotorPositionRevs,
        double steeringControllerMotorRevsPerSteeringRev,
        double swerveModuleOrientation_rad,
        double expectedMagnitude_in_s,
        double expectedDirection_rad) {

    MockDriveController driveController = new MockDriveController();
    driveController.setMotorRpm(actualDriveMotorSpeedRpm);;
    driveController.setMotorRevsPerWheelRev(driveControllerMotorRevsPerWheelRev);
    driveController.setWheelDiameterInches(driveControllerWheelDiameterInches);

    MockSteeringController steeringController = new MockSteeringController();
    steeringController.setMotorRevCount(actualSteeringMotorPositionRevs);
    steeringController.setMotorRevsPerSteeringRev(
        steeringControllerMotorRevsPerSteeringRev);

    SwerveModule test = new SwerveModule(
        steeringController,
        driveController,
        Vector2D.FromXY(1.0, 1.0),
        swerveModuleOrientation_rad);

    IVector2D actual = test.getCurrentVelocity_in_s_rad();
    assertEquals(expectedMagnitude_in_s, actual.getMagnitude(), 0.01);
    assertEquals(expectedDirection_rad, actual.getAngleRadians(), 0.01);
}
```

This helper function takes 8 parameters. The first 6 parameters define the scenario (the first 3 are used to build the `MockDriveController` and the 4th and 5th are used to construct the `MockSteeringController`, the 6th is the Swerve Module Orientation), and the last 2 parameters represent the expected answer (velocity magnitude and direction).

## Unit Tests

To construct unit tests, we need to create a set of inputs for which we can compute the expected answer ourselves. Let's start with a simple one:

- `actualDriveMotorSpeedRpm`: **60.0** (which is one motor revolution per second)
- `driveControllerMotorRevsPerWheelRev`: **1.0** (which is a one-to-one gear ratio)
- `driveControllerWheelDiameterInches`: **1 / pi** (which gives a circumference of 1 inch)
- `actualSteeringMotorPositionRevs`: **0.0** (align steering to module forward or zero)
- `steeringControllerMotorRevsPerSteeringRev`: **1.0** (one-to-one gear ratio)
- `swerveModuleOrientation_rad`: **0.0** (align module with robot forward)

From there we can calculate our expected values quite easily:

- `expectedMagnitude_in_s`: **1.0**
  - This is because the drive motor is spinning at 60 RPM, or 1 rotation per second, there's a one-to-one gear ratio between the motor and the wheel, so the wheel is rotating once per second, and the wheel has a circumference of 1 inch, so it's moving at 1 in. per sec.
- `expectedDirection_rad`: **0.0**
  - This is because the steering is at zero position, and there's no difference between module forward and robot forward, so the wheel is pointed robot forward (zero angle).

Using our unit test helper function from above, that would look like this:

```
    test_getCurrentVelocity_in_s_rad(
        60.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 1.0, 0.0);
```

Great. Now we can start writing more tests by simply varying one of the values and changing the expected result. For instance, if we doubled the drive motor RPM from 60.0 to 120.0 (the first parameter), then we should see the expected magnitude double from 1.0 inch per second to 2.0 inches per second (second last parameter):

```
    test_getCurrentVelocity_in_s_rad(
        120.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 2.0, 0.0);
```

For the 3rd test, we can double the gear ratio from one-to-one to two-to-one (second parameter) and that should cause the expected magnitude go from 1.0 inch per second to ½ inch per second:

```
    test_getCurrentVelocity_in_s_rad(
        60.0, 2.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 0.5, 0.0);
```

…and so on…

Using the same procedure, let's flesh out a bunch of test scenarios:

```
@Test
public void Test_getCurrentVelocity_in_s_rad() {
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 1.0, 0.0);
    test_getCurrentVelocity_in_s_rad(120.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 2.0, 0.0);
    test_getCurrentVelocity_in_s_rad(60.0, 2.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 0.5, 0.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 2.0/Math.PI, 0.0, 1.0, 0.0, 2.0, 0.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, 0.25, 1.0, 0.0, 1.0, Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, -0.25, 1.0, 0.0, 1.0, -Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(-60.0, 1.0, 1.0/Math.PI, 0.25, 1.0, 0.0, 1.0, -Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(-60.0, 1.0, 1.0/Math.PI, -0.25, 1.0, 0.0, 1.0, Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, 0.75, 1.0, 0.0, 1.0, -Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, -0.75, 1.0, 0.0, 1.0, Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, 1.75, 1.0, 0.0, 1.0, -Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, -1.75, 1.0, 0.0, 1.0, Math.PI/2.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, 0.5, 3.0, 0.0, 1.0, Math.PI/3.0);
    test_getCurrentVelocity_in_s_rad(60.0, 1.0, 1.0/Math.PI, -0.5, 3.0, 0.0, 1.0, -Math.PI/3.0);
}
```

## Implementing the Method

Now that we've written our tests (and you should run them once to make sure they fail), now it's time to write the actual `getCurrentVelocity_in_s_rad()` method.

First, we start with the drive motor RPM from the `DriveController`:

```
double driveMotorRpm = this.driveController.getMotorRpm();
```

Next, we need to convert that wheel speed in RPM into inches per second. It's helpful to create a small helper function (that can also be used for the `getMaxSpeed_in_s()` method created earlier):

```
    private double calculateWheelSpeed_in_s(double motorRpm) {
        double gearRatio = this.driveController.getMotorRevsPerWheelRev();
        if(gearRatio <= 0.0) {
            return 0.0;
        }
        double wheelRpm = motorRpm / gearRatio;
        double inchesPerMinute = wheelRpm
            * this.driveController.getWheelDiameterInches() * Math.PI;
        double inchesPerSecond = inchesPerMinute / 60.0;
        return inchesPerSecond;
    }
```

Notice that this function takes a drive motor RPM and converts it into inches per second, but it preserves the sign, so if the motor is running backwards, we'll get a negative speed in in/s.

So, in the `getCurrentVelocity_in_s_rad()` method, now we can take `driveMotorRpm` and compute wheel speed:

```
double speed_in_s = this.calculateWheelSpeed_in_s(driveMotorRpm);
```

But what we really want for velocity is the magnitude, which is always positive, so we'll take the absolute value of the speed:

```
double positiveSpeed_in_s = Math.abs(speed_in_s);
```

Good, that's half of our answer (magnitude). Next, we need to calculate the direction. As we saw previously, the steering controller will give us a number of motor rotations and the gear ratio between the motor turns and the steering revolutions. With this and the module orientation, we can calculate the robot oriented steering angle in radians limited from -PI to PI. For this I suggest creating another little helper function:

```
private double getRobotOrientedSteeringAngle_rad() {
    double gearRatio = this.steeringController.getMotorRevsPerSteeringRev();
    if(gearRatio <= 0.0) {
        return 0.0;
    }
    double steeringMotorRotations = this.steeringController.getMotorRevCount();
    double steeringRotations = steeringMotorRotations / gearRatio;
    double steeringAngle_rad = steeringRotations * 2.0 * Math.PI;
    double robotOrientedSteeringAngle =
        steeringAngle_rad
        + this.moduleOrientation_rad;
    return SwerveUtil
        .limitAngleFromNegativePItoPI_rad(robotOrientedSteeringAngle);
}
```

Using this helper function, we can calculate the velocity direction. Notice that if the wheel speed is negative, we reverse the direction by adding PI (half a circle) and limiting to within -PI to PI again:

```
    double direction_rad = getRobotOrientedSteeringAngle_rad();
    if(speed_in_s < 0.0) {
        direction_rad += Math.PI; // adds half a circle
        direction_rad = SwerveUtil
            .limitAngleFromNegativePItoPI_rad(direction_rad);
    }
```

Finally, we can just return the result:

```
return Vector2D.FromPolar(positiveSpeed_in_s, direction_rad);
```

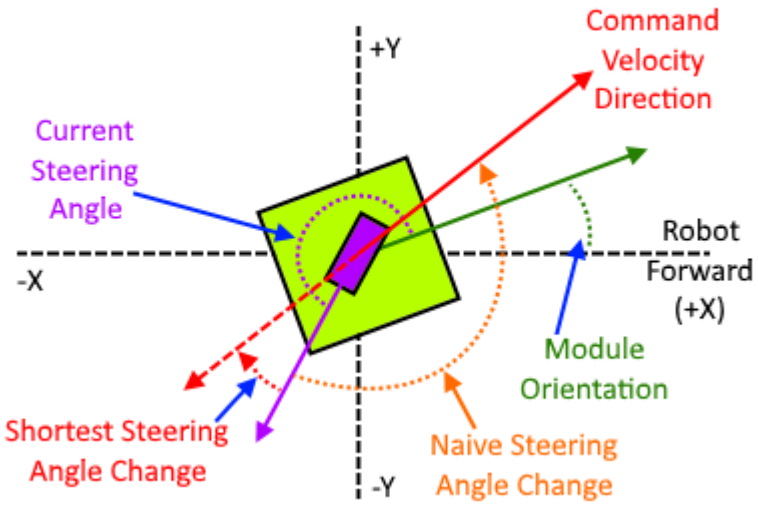
## Execute (Velocity Mode)

The job of the `execute()` method is to take a velocity command *relative to robot forward* and convert that into a rotational position command for the steering controller, and a speed command for the drive controller.

Since the command is relative to robot forward and our swerve module may be oriented differently, we need to take the module orientation into account. For example, if the module is oriented pointing to the right side of the robot, and the command is to drive forward, then we need to tell the steering controller to steer at 90 degrees (positive is counter-clockwise).

Additionally, since the wheel can spin both forwards and backwards, we want to choose an optimal combination of steering direction and drive motor direction. For example, if the steering controller is currently pointed at 10 degrees and moving forward, and the new command is to move in the direction of 170 degrees, then it's better to steer to -10 degrees and run the motor backwards. That means we have to take the current steering angle into account.

For example, in this diagram, the current steering angle is almost directly opposite the desired Command Velocity Direction. In this case, it's shorter to steer towards the opposite direction (dashed red arrow) and run the drive motor in the reverse direction, than to try to steer all the way to the solid red arrow (see the orange Naïve Steering Angle Change angle). Our goal is to align the wheel to the Command Velocity direction, *or its opposite*, whichever is shorter.



## Unit Test Helper Function

Inside our `TestSwerveModule` class, build a method just for testing the `execute()` method:

```
private void test_execute(
        double steeringControllerMotorRevsPerSteeringRev,
        double steeringControllerMotorRevCount,
        double driveControllerMotorRevsPerWheelRev,
        double driveControllerWheelDiameterInches,
        double swerveModuleOrientationRadians,
        double commandVelocityMagnitudeInchesPerSecond,
        double commandDirectionRadians,
        double expectedSteeringTargetMotorRevs,
        double expectedDriveTargetMotorRpm) {

    MockSteeringController steeringController = new MockSteeringController();
    steeringController.setMotorRevCount(steeringControllerMotorRevCount);
    steeringController.setMotorRevsPerSteeringRev(
        steeringControllerMotorRevsPerSteeringRev);

    MockDriveController driveController = new MockDriveController();
    driveController.setMotorRevsPerWheelRev(driveControllerMotorRevsPerWheelRev);
    driveController.setWheelDiameterInches(driveControllerWheelDiameterInches);

    SwerveModule test = new SwerveModule(
        steeringController,
        driveController,
        Vector2D.FromXY(1.0, 1.0),
        swerveModuleOrientationRadians);

    IVector2D velocityCommand = Vector2D.FromPolar(
        commandVelocityMagnitudeInchesPerSecond,
        commandDirectionRadians);
    test.execute(velocityCommand);

    assertEquals(expectedSteeringTargetMotorRevs, steeringController.getTargetMotorRev(), 0.001);
    assertEquals(expectedDriveTargetMotorRpm, driveController.getTargetMotorRpm(), 0.001);
}
```

The function has 9 parameters. The 1st and 2nd setup the steering controller (gear ratio and current steering motor revolutions from zero), the 3rd and 4th setup the drive controller (gear ratio and wheel diameter), the 5th is the swerve module orientation relative to robot forward. The 6th and 7th parameters are the commanded velocity magnitude and direction (the input to the `execute()` method). Finally, the 8th and 9th parameters are the expected result (the target steering motor revs sent to the steering controller, and the target motor RPM sent to the drive controller).

## Unit Tests

Let's start by creating a simple unit test scenario that we can easily work out in our heads:

- `steeringControllerMotorRevsPerSteeringRev`: **1.0** (one-to-one gear ratio)
- `steeringControllerMotorRevCount`: **0.0** (currently pointed towards module forward)
- `driveControllerMotorRevsPerWheelRev`: **1.0** (one-to-one gear ratio)
- `driveControllerWheelDiameterInches`: **1 / pi** (the circumference will then be 1 inch)
- `swerveModuleOrientationRadians`: **0.0** (module forward aligned to robot forward)
- `commandVelocityMagnitudeInchesPerSecond`: **1.0** (1 in. per sec. command velocity)
- `commandDirectionRadians`: **0.0** (direction forward, relative to robot forward)

Using those inputs, it's simple to see that the `SwerveModule` should command the `SteeringController` to target a position of 0.0 motor revolutions, and should command the `DriveController` to spin at 60.0 RPM (one rotation per second). Our first unit test scenario would therefore look like this:

```
    test_execute(1.0, 0.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 0.0, 60.0);
```

Using that test as a template, we can then write some more simple variants of this test by changing one number and making sure the result changes as we expect. For instance, change the `driveControllerMotorRevsPerWheelRev` (drive gear ratio) from 1.0 to **2.0**, and make sure that the `expectedDriveTargetMotorRpm` change from 60.0 to **120.0**. It has to spin twice as fast to maintain the same wheel speed with a two-to-one gear ratio:

```
    test_execute(1.0, 0.0, 2.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 0.0, 120.0);
```

Likewise, if we double the command magnitude, we should see the motor RPM double:

```
    test_execute(1.0, 0.0, 1.0, 1.0/Math.PI, 0.0, 2.0, 0.0, 0.0, 120.0);
```

If we double the wheel diameter (from 1/pi to **2/pi**) then the drive motor target RPM should change from 60.0 to **30.0** because you now travel twice as many inches per motor revolution, so you only have to run the motor half as fast:

```
    test_execute(1.0, 0.0, 1.0, 2.0/Math.PI, 0.0, 1.0, 0.0, 0.0, 30.0);
```

Now, let's create a scenario where we spin the actual steering angle around by half a rotation (from 0.0 to **0.5**) so the wheel is now pointed backwards to our desired command direction. We want it to leave the steering in that direction, but reverse the target motor speed sent to the `DriveController` (from 60.0 to **-60.0**):

```
    test_execute(1.0, 0.5, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, 0.5, -60.0);
```

Now let's create the same scenario, but with the steering rotated ½ rotation in the negative direction:

```
    test_execute(1.0, -0.5, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, -0.5, -60.0);
```

Remembering that the swerve module can continuously rotate, we should get the same result given any number of full revolutions. So, let's replace those last 2 tests with a for loop that checks that we get the same result regardless of the number of full steering revolutions:

```
    for(double r = -2.0; r <= 2.0; r += 1.0) {
        test_execute(1.0, r+0.5, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.5, -60.0);
        test_execute(1.0, r+0.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.0, 60.0);
        test_execute(1.0, r-0.5, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r-0.5, -60.0);
    }
```

Next, apply the same test, but with a 10:1 steering gear ratio. Notice that the values to the 1st, 2nd, and 2nd last parameters are multiplied by 10 because the steering motor has to turn 10 times as many revolutions for the wheel to steer to the same steering angle:

```
    for(double r = -20.0; r <= 20.0; r += 10.0) {
        test_execute(10.0, r+5.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+5.0, -60.0);
        test_execute(10.0, r+0.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.0, 60.0);
        test_execute(10.0, r-5.0, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r-5.0, -60.0);
    }
```

In all of the existing tests, the current steering angle has always been equal to the expected target steering angle, and we're just testing that the `SwerveModule` continues to tell the `SteeringController` to stay at the same position. Next, let's start varying the current steering angle. As long as we only vary it by just less than 1/4 revolution then the target should still be the same:

```
    for(double r = -2.0; r <= 2.0; r += 1.0) {
        test_execute(1.0, r+0.7, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.5, -60.0);
        test_execute(1.0, r+0.6, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.5, -60.0);
        test_execute(1.0, r+0.4, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.5, -60.0);
        test_execute(1.0, r+0.3, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.5, -60.0);

        test_execute(1.0, r+0.2, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.0, 60.0);
        test_execute(1.0, r+0.1, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.0, 60.0);
        test_execute(1.0, r-0.1, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.0, 60.0);
        test_execute(1.0, r-0.2, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.0, 60.0);

        test_execute(1.0, r-0.3, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r-0.5, -60.0);
        test_execute(1.0, r-0.4, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r-0.5, -60.0);
        test_execute(1.0, r-0.6, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r-0.5, -60.0);
        test_execute(1.0, r-0.7, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r-0.5, -60.0);
    }
```

It's always important to test the boundary conditions. In this case, when the current steering direction is exactly ¼ revolution away from the command direction, the swerve module can legitimately steer either direction by ¼ revolution, as long as it also sets the drive motor direction correctly.

We could write a test that checked that one of two valid results were returned, but we'd have to write a new unit test helper function to do that. Instead, let's constrain the expected result a bit more, and say that if the steering angle needs to change by exactly ¼ revolution, then it should default to turning the direction that would result in the motor spinning in a forward direction, rather than reversing:

```
    for(double r = -2.0; r <= 2.0; r += 1.0) {
        test_execute(1.0, r+0.25, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r+0.0, 60.0);
        test_execute(1.0, r-0.25, 1.0, 1.0/Math.PI, 0.0, 1.0, 0.0, r-0.0, 60.0);
    }
```

The last parameter that we haven't varied is the 5th parameter, module orientation. Let's start with our first test scenario, and add a small "delta" (or change) of **0.1** radians (about 5.7 degrees). If we set the module orientation to +0.1 radians (positive is counter-clockwise), we should see the steering motor target move in the opposite direction. We can calculate the number of motor revolutions by dividing the radians by 2 x PI:

```
double delta_rad = 0.1;
double delta_rev = delta_rad / (2.0 * Math.PI);
test_execute(1.0, 0.0, 1.0, 1.0/Math.PI, delta_rad, 1.0, 0.0, -delta_rev, 60.0);
```

Likewise, if we set the module orientation to a *negative* angle, we should see the steering motor target move in the *positive* direction to compensate:

```
test_execute(1.0, 0.0, 1.0, 1.0/Math.PI, -delta_rad, 1.0, 0.0, delta_rev, 60.0);
```

If we continue increasing the module orientation angle, this relationship should stay the same until we get to ¼ rotation. At that point (once the module orientation is more than 90 degrees from the front of the robot) then since the steering angle is current at zero (module forward) and the command is at zero (robot forward) then it's a shorter distance to steer the opposite direction and reverse the drive motor direction (from 60.0 to **-60.0** RPM):

```
double d2_rad = Math.toRadians(95.0);
double d2_rev = Math.toRadians(85.0) / (2.0 * Math.PI);
test_execute(1.0, 0.0, 1.0, 1.0/Math.PI, d2_rad, 1.0, 0.0, d2_rev, -60.0);
test_execute(1.0, 0.0, 1.0, 1.0/Math.PI, -d2_rad, 1.0, 0.0, -d2_rev, -60.0);
```

So that's a pretty list of tests we've created. It doesn't hurt, with a complicated system like this, to plug in some real-world scenarios just as a sanity check. Let's start by picking some real-world values for our swerve drive parameters, like gear ratios and wheel diameters:

- `steeringControllerMotorRevsPerSteeringRev`: **9.0**
- `driveControllerMotorRevsPerWheelRev`: **8.0**
- `driveControllerWheelDiameterInches`: **4.0**
- `swerveModuleOrientationRadians`: **0.0**

Then these 3 parameters determine the bulk of the scenario:

- `steeringControllerMotorRevCount`: **9.0 / 8.0** (1/8$^{th}$ of a full rev is 45 degrees)
- `commandVelocityMagnitudeInchesPerSecond`: **5.3**
- `commandDirectionRadians`: **-PI/2** (robot right, which is negative Y direction)

Given those parameters, we expect the swerve module to steer an additional 8$^{th}$ of a turn (9/8 motor revolutions) in the positive direction, and run the motor ***backwards*** at this speed:

Drive motor speed = 60.0 * 8.0 * 5.3 in/s / (4.0 x PI) = **202.445** RPM

```
test_execute(9.0, 9.0/8.0, 8.0, 4.0, 0.0, 5.3, -Math.PI/2.0, 9.0/4.0, -202.445);
```

## Implementing the Method

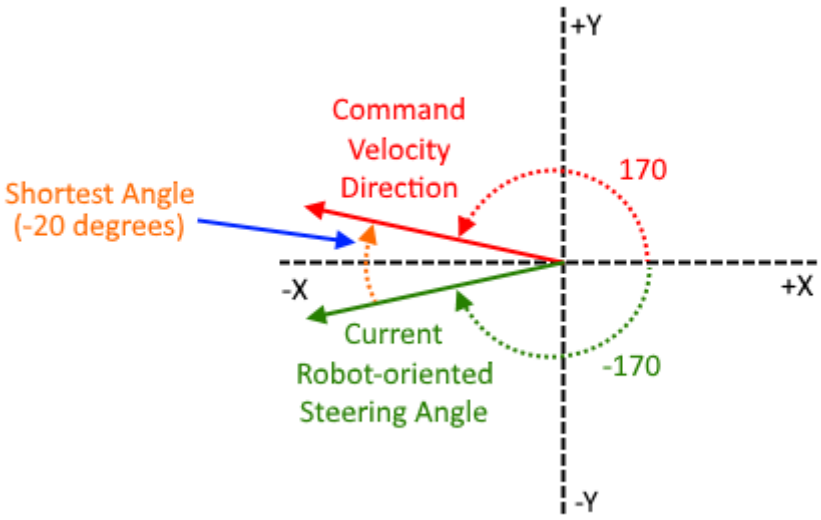
Start by adding an empty `execute()` method to the `SwerveModule` class like this:

```
    @Override
    public void execute(IVector2D velocityCommand_in_s_rad) {

    }
```

Our strategy will be:

1. Figure out how far we have to steer to get to the command direction
2. If the steering angle change is greater than a ¼ revolution:
    a. Re-compute the steering angle change the other direction
    b. Remember that we have to reverse the drive motor direction
3. Convert the steering angle change into a change of steering motor revolutions
4. Add that to the current position of the steering motor and set that as the target
5. Convert the velocity magnitude from inches/second to motor RPM
6. Optionally multiply the motor RPM by -1 if we decided to do so in 2(b) above
7. Set this as the target drive motor speed

Our first step is to figure out how far the current robot-oriented steering angle is from the velocity command direction. This is complicated by the fact that both angles are in the range of -PI to PI (-180 degrees to 180 degrees) but if, for example, the velocity command was 170 degrees and the current steering angle was -170 degrees, then the difference would be +20 degrees:



Since this is a fairly complicated function to write by itself, let's write a helper function in the `SwerveUtil` class called `shortestDiffToTargetAngle_rad()`:

```
    public static double shortestDiffToTargetAngle_rad(
            double currentAngle_rad,
            double targetAngle_rad) {

    }
```

As you can see, we take 2 parameters: the current angle, and the target angle, both in radians. To make sure we get this function right, let's test it by creating some unit tests. Begin by creating a unit test helper function:

```
    private void test_shortestDiffToTargetAngle_rad(
            double currentAngle_rad,
            double targetAngle_rad,
            double expected_rad) {
        double result = SwerveUtil.shortestDiffToTargetAngle_rad(
            currentAngle_rad,
            targetAngle_rad);
        assertEquals(expected_rad, result, Math.toRadians(0.01));
    }
```

This is all in radians, and it's often easier to think in degrees, so let's create another helper function so we can specify the angles in degrees:

```
    private void test_shortestDiffToTargetAngle_deg(
            double currentAngle_deg,
            double targetAngle_deg,
            double expected_deg) {
        test_shortestDiffToTargetAngle_rad(
            Math.toRadians(currentAngle_deg),
            Math.toRadians(targetAngle_deg),
            Math.toRadians(expected_deg));
    }
```

Now we can just create a list of tests that lists the expected outcomes from a variety of inputs:

```
    @Test
    public void Test_shortestDiffToTargetAngle_rad() {
        test_shortestDiffToTargetAngle_rad(0.0, 0.0, 0.0);
        test_shortestDiffToTargetAngle_rad(Math.PI/2.0, Math.PI/2.0, 0.0);
        test_shortestDiffToTargetAngle_rad(-Math.PI/2.0, -Math.PI/2.0, 0.0);
        test_shortestDiffToTargetAngle_rad(Math.PI, Math.PI, 0.0);
        test_shortestDiffToTargetAngle_rad(-Math.PI, -Math.PI, 0.0);
        test_shortestDiffToTargetAngle_rad(Math.PI, -Math.PI, 0.0);
        test_shortestDiffToTargetAngle_rad(-Math.PI, Math.PI, 0.0);

        test_shortestDiffToTargetAngle_deg(10.0, 20.0, 10.0);
        test_shortestDiffToTargetAngle_deg(-10.0, -20.0, -10.0);
        test_shortestDiffToTargetAngle_deg(-10.0, 10.0, 20.0);
        test_shortestDiffToTargetAngle_deg(170.0, -170.0, 20.0);
        test_shortestDiffToTargetAngle_deg(180.0, -170.0, 10.0);
        test_shortestDiffToTargetAngle_deg(170.0, -180.0, 10.0);
        test_shortestDiffToTargetAngle_deg(180.0, 170.0, -10.0);
        test_shortestDiffToTargetAngle_deg(-170.0, -180.0, -10.0);
    }
```

Now we can implement `SwerveUtil.shortestDiffToTargetAngle_rad()`. We should be able to assume that the two inputs are in the range of -PI to PI, but it doesn't hurt to sanitize our inputs just to make sure. Let's run both through our existing `limitAngleFromNegativePItoPI_rad()` function and subtract them to get an initial difference:

```
        double diff =
            limitAngleFromNegativePItoPI_rad(targetAngle_rad)
            - limitAngleFromNegativePItoPI_rad(currentAngle_rad);
```

If the difference is between -PI to PI, we're done, but there's a chance that `diff` is outside of that range, and we need to go the other direction. To determine if we need to go the other direction, let's break `diff` into a magnitude and a direction:

```
        double diffMagnitude = Math.abs(diff);
        double diffDirection = diff >= 0 ? 1.0 : -1.0;
```

Use an if statement to check if we need to go the other direction, and if we do, re-calculate the magnitude, and invert the direction:

```
        if(diffMagnitude > Math.PI) {
            diffMagnitude = TWO_PI - diffMagnitude;
            diffDirection *= -1.0;
        }
```

Finally, we can then re-constitute our angle difference from `diffMagnitude` and `diffDirection`:

```
        return diffMagnitude * diffDirection;
```

Back to our implementation of `execute()`, now we can use the new helper method we just created, and the `getRobotOrientedSteeringAngle_rad()` method we created when implementing the `getCurrentVelocity_in_s_rad()` method to calculate how far we need to steer to reach the target velocity direction:

```
double steeringAngleDiff_rad = SwerveUtil.shortestDiffToTargetAngle_rad(
    this.getRobotOrientedSteeringAngle_rad(),
    velocityCommand_in_s_rad.getAngleRadians());
```

The result of the calculation is an angle from -PI to PI. If it's outside the range of -PI/2 to PI/2, then we should steer in the other direction, and reverse the direction of rotation of the drive wheel. Start by decomposing the `steeringAngleDiff_rad` into a magnitude and a direction:

```
double steeringAngleDiffMagnitude_rad = Math.abs(steeringAngleDiff_rad);
double steeringAngleDiff_direction = steeringAngleDiff_rad >= 0.0 ? 1.0 : -1.0;
```

Let's default the drive motor direction to forward (positive):

```
double driveMotorDirection = 1.0;
```

If the magnitude of the steering angle diff is more than PI/2, then we need to re-compute the magnitude, and invert the direction (and also remember to invert the drive motor direction):

```
if(steeringAngleDiffMagnitude_rad > Math.PI/2.0) {
    steeringAngleDiffMagnitude_rad = Math.PI - steeringAngleDiffMagnitude_rad;
    steeringAngleDiff_direction *= -1.0;
    driveMotorDirection = -1.0;
}
```

Now we can reconstitute the `steeringAngleDiff_rad` variable from the magnitude and direction:

```
steeringAngleDiff_rad =
    steeringAngleDiffMagnitude_rad
    * steeringAngleDiff_direction;
```

Convert the steering angle change from radians to steering revolutions by dividing by 2 x PI:

```
double steeringAngleDiff_revs = steeringAngleDiff_rad / (2.0 * Math.PI);
```

Then apply the gear ratio to get the steering angle change in steering motor revolutions:

```
double steeringMotorDiff_revs =
    steeringAngleDiff_revs
    * this.steeringController.getMotorRevsPerSteeringRev();
```

Now that we have the change in steering motor revs, add it to the current steering motor position to get the new target position for the steering controller:

```
double steeringCommand_revs =
    this.steeringController.getMotorRevCount()
    + steeringMotorDiff_revs;
```

… and apply that target to the `SteeringController` itself:

```
this.steeringController.setTargetMotorRev(steeringCommand_revs);
```

That's the end of computing the steering. Now let's move on to computing the target drive motor speed. Thankfully it's a bit more straightforward. Begin by computing the wheel circumference, in inches (which is simply PI x diameter):

```
double inchesPerWheelRev =
    this.driveController.getWheelDiameterInches() * Math.PI;
```

In the next step, we need to divide by the circumference, and any time we do a division operation in a computer we risk a divide by zero exception. The only way this could happen is if the drive controller returned a diameter of zero inches, and even though this is unlikely, a divide by zero error is bad enough that we really should avoid it at all costs. One question is, what do we do in that case. Since our main goal is to avoid a divide by zero, let's just substitute a sensible wheel diameter instead:

```
if(inchesPerWheelRev <= 0.0) {
    inchesPerWheelRev = 4.0 * Math.PI;
}
```

Using our circumference, convert from our command velocity magnitude in inches per second, to a wheel RPM (revolutions per minute):

```
double commandWheelSpeed_rpm =
    60.0 * velocityCommand_in_s_rad.getMagnitude() / inchesPerWheelRev;
```

Then apply the drive controller gear ratio to get drive motor RPM:

```
double commandMotorSpeed_rpm =
    commandWheelSpeed_rpm
    * this.driveController.getMotorRevsPerWheelRev();
```

Finally, remember to take into account whether we need to reverse the drive motor direction, and set the result as the drive motor target speed:

```
this.driveController
    .executeVelocityMode(commandMotorSpeed_rpm * driveMotorDirection);
```

That's it! Assuming your new method passes all the unit tests, then we can have pretty good confidence that the `execute()` method is working correctly.

### Refactor

Once you've completed a class, and you have unit tests, you're now safe to (carefully) improve the code without changing the functionality. When you do this, it's called "refactoring."

There are typically 2 goals with refactoring:

1. Make it more readable
2. Improve the performance

Typically, we don't worry too much about performance, but since this is an embedded controller with limited processing power, it's worth taking a look to make sure we're not performing excess calculations. For instance, `inchesPerWheelRev` is a value that only depends on the wheel diameter, and we know that won't change during execution, so we could compute that once in the constructor and save it as a class variable:

```
        double inchesPerWheelRev =
            this.driveController.getWheelDiameterInches() * Math.PI;
        if(inchesPerWheelRev <= 0.0) {
            inchesPerWheelRev = 4.0 * Math.PI;
        }
```

There are several other places where we could pre-calculate values and avoid re-calculating them every time we call `execute()`. I'll leave that as an exercise for the reader.

## RobotOrientedSwerve Class

Start with an empty `RobotOrientedSwerve` class:

```
package frc.swerve;

import java.util.*;

public class RobotOrientedSwerve implements IRobotOrientedSwerve {

}
```

To construct this class, we need to have a list of swerve modules:

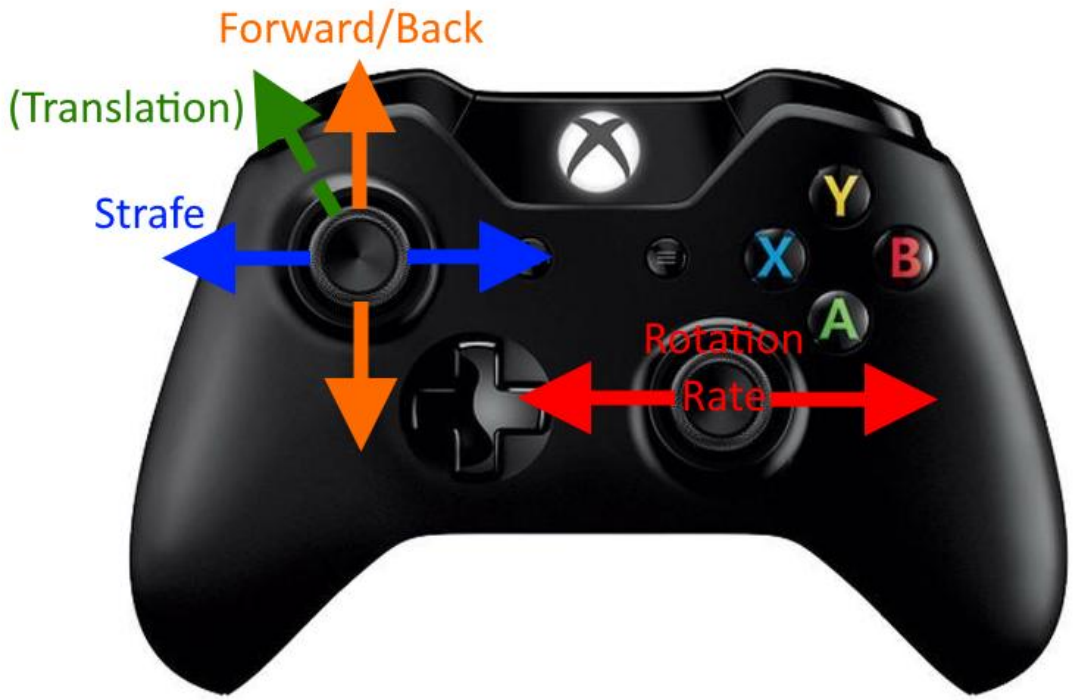
```
    private final ArrayList<ISwerveModule> swerveModules;

    public RobotOrientedSwerve(
            ArrayList<ISwerveModule> swerveModules) {
        this.swerveModules = swerveModules;
    }
```

Then we need to implement the `IRobotOrientedSwerve` interface, which is only a single method:

```
    @Override
    public void execute(
            IVector2D translationCommand_in_s_rad,
            double targetRotationRate_rad_s) {

    }
```

The purpose of the `RobotOrientedSwerve` class is to take the translation command and the target rotation rate, and convert that into a velocity command for each swerve module.

Just to put this into a practical perspective, the translation and rotation commands could come directly from an Xbox controller, and we'd actually be able to drive the robot around. Here's an example (the choice of which thumb stick does what is completely arbitrary):




Each of those controls (orange, blue, and red) can be read in the robot program giving us an "analog" value from -1.0 to 1.0. The Forward/Back (orange axis) would be the X component of our translation vector (positive forward), and the Strafe (blue axis) would be the Y component (positive to the left).

You could use `Vector2D.FromXY()` to create a translation vector from those 2 axes, but the magnitude of that vector would be from -1.0 to 1.0. You'd then have to scale that magnitude to get a speed in inches per second. For instance, if you were testing and wanted to limit the robot speed to 10.0 inches per second, you could scale the vector magnitude by multiplying by 10.0.

Similarly, you could take the Rotation Rate (red axis), which would be a number from -1.0 to 1.0, and scale that to your maximum desired rotation rate in radians per second. Since one radian is about 57 degrees, I'd probably start with a scaling of about 0.5 to 1.0 (divide by 2) to get a reasonably safe rotation rate. Note that you also have to get the direction right. In our math, we've always assumed positive rotation is counter-clockwise (because that's the typical mathematical standard). If the analog thumb stick gives you a positive value when you move to the right, you'll need to multiply by -1 to get the desired rotation rate.

## MockSwerveModule Class

Since `RobotOrientedSwerve` depends on `ISwerveModule`, then to write our unit tests, we need to create a `MockSwerveModule` class that implements `ISwerveModule` and allows us to set and get all the properties:

```
package frc.swerve;

public class MockSwerveModule implements ISwerveModule {

    private boolean homed = true;

    public void setHomed(boolean homed) {
        this.homed = homed;
    }

    @Override
    public boolean isHomed() {
        return this.homed;
    }

    private IVector2D velocityCommand_in_s_rad = null;

    @Override
    public void execute(IVector2D velocityCommand_in_s_rad) {
        this.velocityCommand_in_s_rad = velocityCommand_in_s_rad;
    }

    public IVector2D getVelocityCommand_in_s_rad() {
        return this.velocityCommand_in_s_rad;
    }

    private IVector2D currentVelocity_in_s_rad
        = Vector2D.FromPolar(0.0, 0.0);

    public void setCurrentVelocity_in_s_rad(
            IVector2D currentVelocity_in_s_rad) {
        this.currentVelocity_in_s_rad = currentVelocity_in_s_rad;
    }

    @Override
    public IVector2D getCurrentVelocity_in_s_rad() {
        return this.currentVelocity_in_s_rad;
    }
```

```
    private IVector2D modulePos_in = Vector2D.FromPolar(1.0, 0.0);

    public void setModulePos_in(
            IVector2D modulePos_in) {
        this.modulePos_in = modulePos_in;
    }

    @Override
    public IVector2D getModulePos_in() {
        return modulePos_in;
    }

    private double moduleOrientation_rad;

    public void setModuleOrientation_rad(
            double moduleOrientation_rad) {
        this.moduleOrientation_rad = moduleOrientation_rad;
    }

    @Override
    public double getModuleOrientation_rad() {
        return this.moduleOrientation_rad;
    }

    private double maxSpeed_in_s = 40000.0; // very large, about 1 km/s

    public void setMaxSpeed_in_s(double maxSpeed_in_s) {
        this.maxSpeed_in_s = maxSpeed_in_s;
    }

    @Override
    public double getMaxSpeed_in_s() {
        return this.maxSpeed_in_s;
    }
}
```

## TestRobotOrientedSwerve Class

Next, create a new class in our test source tree to hold the unit tests for the `RobotOrientedSwerve` class:

```
package frc.swerve;

import org.junit.Test;
import static org.junit.Assert.*;

public class TestRobotOrientedSwerve {

}
```

## Testing with a Single Swerve Module

The `RobotOrientedSwerve` class only has one publicly accessible method that we need to test: `execute()`. Next, let's create a helper function in `TestRobotOrientedSwerve` to let us test this method.

### Unit Test Helper Function

This `execute()` method's job is to apply the swerve math to all swerve modules independently, so for most of our tests, it's sufficient to test with just one mock swerve module. While there are physical and mechanical reasons why a robot with only one wheel wouldn't work very well, mathematically this is perfectly legitimate way to test our code.

The inputs to our swerve drive math will be:

- The swerve module's position (X and Y component) relative to the robot center of mass
- Translation command (X and Y component)
- Rotation command

For these tests we're going to ignore the following parameters (we will test their effect in later tests):

- Swerve module `isHomed()` status → we will set this value to `true` for all tests
- Swerve module `getMaxSpeed_in_s()` value → this will be some huge number for all tests

The result that we want to check is the velocity command sent to the swerve module.

Here's the unit test helper function for testing with a single swerve module:

```
    private void test_execute_single_swerve_module(
            double modulePosX_in,
            double modulePosY_in,
            double translationCommandX_in_s,
            double translationCommandY_in_s,
            double rotationCommand_rad_s,
            double expectedVelocityCommandX_in_s,
            double expectedVelocityCommandY_in_s) {

        MockSwerveModule swerveModule = new MockSwerveModule();
        swerveModule.setMaxSpeed_in_s(40000.0); // around 1 km/s
        assertTrue(swerveModule.isHomed()); // assume module is homed
        swerveModule.setModulePos_in(
            Vector2D.FromXY(modulePosX_in, modulePosY_in)
        );

        ArrayList<ISwerveModule> swerveModules = new ArrayList<>(
            Arrays.asList(swerveModule)
        );
        RobotOrientedSwerve test = new RobotOrientedSwerve(swerveModules);

        IVector2D translationCommand_in_s_rad
                = Vector2D.FromXY(
                    translationCommandX_in_s,
                    translationCommandY_in_s);
        test.execute(translationCommand_in_s_rad, rotationCommand_rad_s);

        IVector2D result = swerveModule.getVelocityCommand_in_s_rad();
        assertNotNull(result);
        assertEquals(expectedVelocityCommandX_in_s, result.getX(), 0.01);
        assertEquals(expectedVelocityCommandY_in_s, result.getY(), 0.01);
    }
```

## Unit Tests (Single Swerve Module)

Start by making a method in `TestRobotOrientedSwerve` to hold these tests:

```
    @Test
    public void Test_execute_single_swerve_module() {

    }
```

We're going to start with some simple unit tests. The simplest scenario is when the rotation rate is zero. In that case, the velocity command given to the swerve module should always be equal to the translation command, regardless of the swerve module position.

This is easy to understand, if you consider the actual robot for a moment. If you grab the Xbox controller and give the robot a command to move forward at 5 in/s, with no rotation command, then we want all swerve modules to steer forward and run the drive wheels at 5 in/s. Similarly, if we to strafe right at 10 in/s, then all modules should steer right and run the drive wheels at 10 in/s.

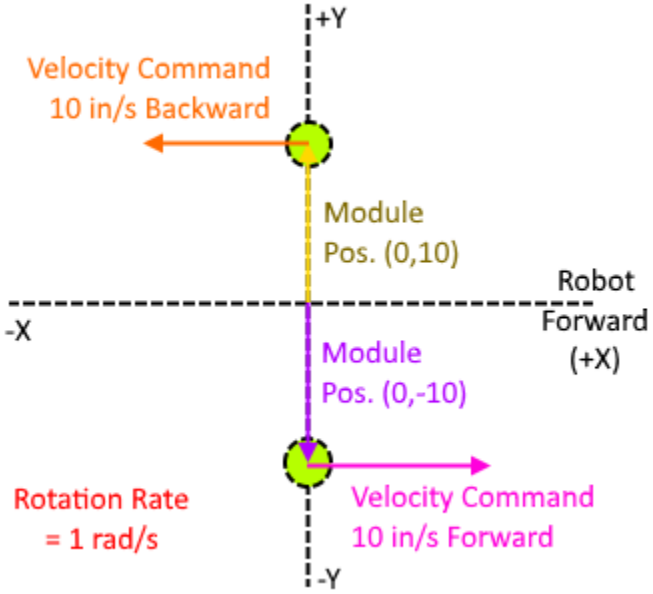
Start with a simple test:

```
        test_execute_single_swerve_module(10.0, 10.0, 5.0, 0.0, 0.0, 5.0, 0.0);
```

We can then write some more tests by varying the 1st and 2nd parameters (to anything), and modifying the 3rd and 4th parameters (translation command), and making sure the expected velocity (last to parameters) are equal:

```
        test_execute_single_swerve_module(-5.0, 20.0, 8.0, -3.5, 0.0, 8.0, -3.5);
        test_execute_single_swerve_module(0.0, -3.33, -2.0, 1.5, 0.0, -2.0, 1.5);
        test_execute_single_swerve_module(-1.0, -1.0, 0.0, 11.7, 0.0, 0.0, 11.7);
```

Now, instead of translation, let's test rotation by itself. Assume for a second that we have a robot with 2 swerve modules. One module is on the left side (+Y direction) and one is on the right side (-Y direction). Let's put them at 10 inches from the center of mass of the robot:




If we wanted to rotate the robot at a rate of 1 radian per second (positive, or counter-clockwise), what velocity commands would we have to give to the swerve modules to cause this rotation?

This is where working in radians actually helps us. If we travel one radius of distance around the circumference of a circle, we will have traveled one radian. Therefore, if we want to rotate the robot at one radian per second, then each swerve module has to travel fast enough to travel the distance of one radius (10 inches) in one second. That means each swerve module needs to be going at 10 in/s.

To cause a rotation, we just need to make sure they go in the right direction. The right swerve module needs to be going forward (+X direction), and the left swerve module needs to be going backward (-X).

Create a unit test for each of those scenarios:

```
        test_execute_single_swerve_module(0.0, 10.0, 0.0, 0.0, 1.0, -10.0, 0.0);
        test_execute_single_swerve_module(0.0, -10.0, 0.0, 0.0, 1.0, 10.0, 0.0);
```

A simple variant of this test is to move the swerve modules closer to the center of mass. If the radius is 5 inches per second, then their speed should now be 5 inches per second (to maintain the same rotation rate of 1 radian per second):

```
        test_execute_single_swerve_module(0.0, 5.0, 0.0, 0.0, 1.0, -5.0, 0.0);
        test_execute_single_swerve_module(0.0, -5.0, 0.0, 0.0, 1.0, 5.0, 0.0);
```

Similarly, if we double the rotation rate, but keep the module positions at 5 inches from the center of mass, then the module speeds should double (to 10 inches per second):

```
        test_execute_single_swerve_module(0.0, 5.0, 0.0, 0.0, 2.0, -10.0, 0.0);
        test_execute_single_swerve_module(0.0, -5.0, 0.0, 0.0, 2.0, 10.0, 0.0);
```

Next, reverse the rotation direction (from 1.0 to -1.0 radians per second) and make sure the swerve module velocities change direction:

```
        test_execute_single_swerve_module(0.0, 5.0, 0.0, 0.0, -2.0, 10.0, 0.0);
        test_execute_single_swerve_module(0.0, -5.0, 0.0, 0.0, -2.0, -10.0, 0.0);
```

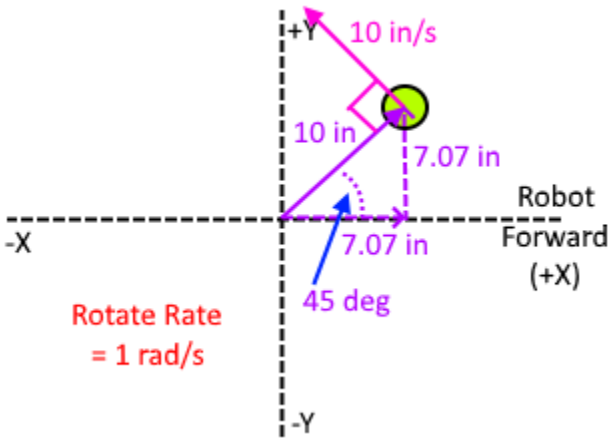
Up to now we've moved the swerve modules along the Y axis, but now let's write some tests with them on the X axis. If we put the module at 10 inches in front of the center of mass, then it needs to get a velocity command of 10 in/s to the left to rotate at a 1 rad/s rate:

```
        test_execute_single_swerve_module(10.0, 0.0, 0.0, 0.0, 1.0, 0.0, 10.0);
```

Likewise, if the swerve module is 10 inches behind (-X) the center of mass, it needs to move in the right (-Y) direction to cause the same rate of rotation:

```
        test_execute_single_swerve_module(-10.0, 0.0, 0.0, 0.0, 1.0, 0.0, -10.0);
```

Let's move on to a more complicated scenario (still just considering rotation). Let's put the swerve module 10 inches away from the center of mass, but out in the +X and +Y quadrant at a 45 degree angle from the X axis:




Since cosine of 45 degrees is 0.7071 (approximately) then we can calculate the X and Y components of the module position to be 7.071 inches, and the X and Y components of the expected velocity command to be 7.071 inches per second.

Let's put that in a unit test:

```
test_execute_single_swerve_module(7.071, 7.071, 0.0, 0.0, 1.0, -7.071, 7.071);
```

Now, create three more variants in each of the other three quadrants:

```
test_execute_single_swerve_module(-7.071, 7.071, 0.0, 0.0, 1.0, -7.071, -7.071);
test_execute_single_swerve_module(-7.071, -7.071, 0.0, 0.0, 1.0, 7.071, -7.071);
test_execute_single_swerve_module(7.071, -7.071, 0.0, 0.0, 1.0, 7.071, 7.071);
```

While we're at it, reverse the rotation direction and make sure the X and Y components of the expected velocity commands change sign:

```
test_execute_single_swerve_module(7.071, 7.071, 0.0, 0.0, -1.0, 7.071, -7.071);
test_execute_single_swerve_module(-7.071, 7.071, 0.0, 0.0, -1.0, 7.071, 7.071);
test_execute_single_swerve_module(-7.071, -7.071, 0.0, 0.0, -1.0, -7.071, 7.071);
test_execute_single_swerve_module(7.071, -7.071, 0.0, 0.0, -1.0, -7.071, -7.071);
```

Another good triangle to use for testing is the 3, 4, 5 triangle. This is a well-known right-angled triangle with hypotenuse of length 5, and shorter sides of length 3 and 4. Let's make a test for that:

```
test_execute_single_swerve_module(3.0, 4.0, 0.0, 0.0, 2.0, -8.0, 6.0);
```

We're going to use that test as the basis for our next test. We need to verify that the `execute()` method will combine our translation command with our rotation command. Thankfully this is simple vector addition, so we can just add the X and Y components to get our result. Take the last test and add a translation command and make sure the result is the addition of the two parts:

```
test_execute_single_swerve_module(3.0, 4.0, 12.5, -3.2, 2.0, 4.5, 2.8);
```

## Partially Implementing the Execute Method

This is a good moment to stop and think about what it would take to write an `execute()` method that would be able to pass all the unit tests we've just created. Let's try that. To start with, we need to iterate through each swerve module:

```
    for (ISwerveModule swerveModule : this.swerveModules) {

    }
```

For each swerve module (inside the for loop), start by getting the vector representing the position of the swerve module:

```
        IVector2D positionVector = swerveModule.getModulePos_in();
```

To calculate the rotation speed in inches per second, we can simply multiply the distance from the center of mass to the swerve module (inches / radius) by the speed (radians / second):

```
        double rotation_speed_in_s
            = positionVector.getMagnitude() * targetRotationRate_rad_s;
```

The direction of the rotation is always at right angles to the position vector, so we an just add 90 degrees, which is PI / 2 radians:

```
        double rotation_angle_rad
            = positionVector.getAngleRadians() + Math.PI / 2.0;
```

Now we can create the rotation component of our swerve module velocity:

```
        IVector2D rotation_in_s_rad = Vector2D.FromPolar(
            rotation_speed_in_s, rotation_angle_rad);
```

Finally, we need to add this rotation component to the translation command:

```
        Vector2D velocityCommand = new Vector2D();
        velocityCommand.add(translationCommand_in_s_rad, rotation_in_s_rad);
```

…and then we just write that to the swerve module:

```
        swerveModule.execute(velocityCommand);
```

That's actually the core functionality of the `RobotOrientedSwerve` class. However, we're not done.

## Testing with Multiple Swerve Modules

There are two other requirements we need to fulfill:

1. Don't give any velocity commands to *any* swerve modules until all swerve modules are homed.
2. If we exceed the maximum speed of *any* swerve module, scale all velocity commands lower so that we don't exceed the capabilities of any swerve modules.

The use of the word *any* in these descriptions means that we need to test with multiple swerve modules at the same time.

### Testing the isHomed Feature

Start by adding a new unit test helper function to the `TestRobotOrientedSwerve` class. This helper function creates two `MockSwerveModule` instances (`swerveModule1` and `swerveModule2`) and sets their homed status based on the first 2 parameters. It then constructs a `RobotOrientedSwerve` instance for testing using those 2 swerve modules. It calls the `execute()` method with a translation and rotation command (all zeros).

Then, based on the 3rd parameter, it tests whether the `execute()` method did or didn't set the velocity command on both swerve modules.

```
    private void test_execute_isHomed(
            boolean swerveModule1isHomed,
            boolean swerveModule2isHomed,
            boolean setsVelocityCommand) {

        MockSwerveModule swerveModule1 = new MockSwerveModule();
        swerveModule1.setHomed(swerveModule1isHomed);
        MockSwerveModule swerveModule2 = new MockSwerveModule();
        swerveModule2.setHomed(swerveModule2isHomed);

        ArrayList<ISwerveModule> swerveModules = new ArrayList<>(
            Arrays.asList(swerveModule1, swerveModule2));
        RobotOrientedSwerve test = new RobotOrientedSwerve(swerveModules);

        test.execute(Vector2D.FromXY(0.0, 0.0), 0.0);

        IVector2D result1 = swerveModule1.getVelocityCommand_in_s_rad();
        IVector2D result2 = swerveModule2.getVelocityCommand_in_s_rad();
        if(setsVelocityCommand) {
            assertNotNull(result1);
            assertNotNull(result2);
        }
        else {
            assertNull(result1);
            assertNull(result2);
        }
    }
```

Now, create an actual test that uses the helper function. Our goal is to make sure that the `execute()` method only sets the velocity command if all swerve modules are homed:

```
    @Test
    public void Test_execute_isHomed() {
        test_execute_isHomed(false, false, false);
        test_execute_isHomed(true, false, false);
        test_execute_isHomed(false, true, false);
        test_execute_isHomed(true, true, true);
    }
```

### Implementing the isHomed Feature

To help implement this feature in the `RobotOrientedSwerve` class, create a helper function inside the class that iterates through the swerve modules and only returns true if all the swerve modules report that they're homed:

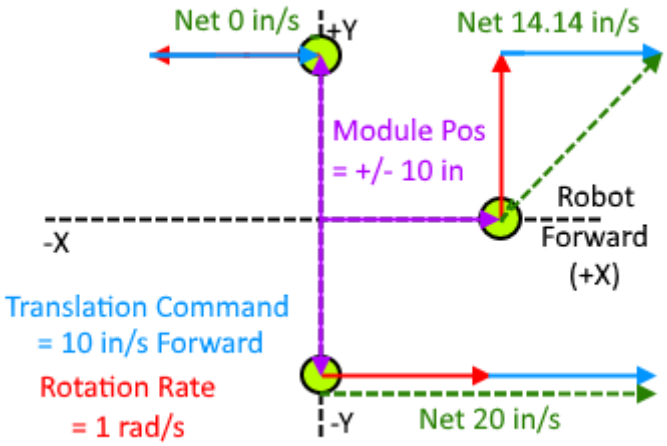
```
    private boolean allHomed() {
        boolean result = true;
        for (ISwerveModule swerveModule : this.swerveModules) {
            if(!swerveModule.isHomed()) {
                result = false;
                break;
            }
        }
        return result;
    }
```

Now, use that helper function to abort execution of the `execute()` method unless all modules are homed:

```
    @Override
    public void execute(
            IVector2D translationCommand_in_s_rad,
            double targetRotationRate_rad_s) {

        if(!allHomed()) {
            return;
        }
        ...
```

## The Maximum Swerve Module Speed Problem

Consider the case where the combined translation and rotation commands exceed the maximum speed of one or more swerve modules. To understand the problem, consider what happens in this 3 wheel system when we have a rotational command of 1 rad/s and a translational command of 10 in/s forward, and all swerve modules are positioned 10 in from the center of mass:




All three swerve modules need to run at a different speed in this situation. The left wheel is stationary, the front wheel is moving at a 45-degree angle at about 14.14 in/s, and the right wheel is moving forward at 20 in/s.

If you look carefully, you'll realize that the wheels are all moving around a circle centered on the left (top) wheel module at 1 rad/s. The left wheel is stationary. The right wheel is 20 inches away from the left wheel, and is moving at 20 in/s perpendicular to it, and the front wheel is 14.14 inches away from the left wheel and is moving at 14.14 in/s perpendicular to the left wheel. All 3 wheels are coordinating to rotate the robot around the left wheel. In order to do this, the front and right wheels need to move at different speeds, proportional to their distance from the center of rotation.

Now let's suppose that the swerve modules are incapable of moving at greater than 10 in/s maximum. If we just limited the speed of each swerve module individually, the left wheel would still stay stationary, but the front and right wheels would both be limited to 10 in/s. The left wheel would want to stay where it was and be the center of rotation, but the front and right wheels are now travelling at the same speed, so they can't be rotating at the same rate around the left wheel point.

So, what happens? In simple terms, the front and right wheels fight each other. There is no motion of the robot that can satisfy all of the conditions, so one of the wheels *must* either slip against the carpet, or if the traction is strong enough, then the drive motor simply might not have enough torque to make it slip. In either case, it wastes power and is hard on the tires and/or motors.

The solution to the problem in this case is to divide all magnitudes by 2. The right wheel will then be travelling at its maximum speed (10 in/s) and the front wheel will slow down proportionally to compensate. The wheels won't fight each other and the overall trajectory will be the same, just slower.

## Testing the Max Speed Feature

Here's a unit test that checks the scenario described above. It creates 3 mock swerve modules (left, front, and right) and sets their maximum speed to 10 in/s. It then executes with a translation command of 10 in/s forward and 1 rad/s rotation, and makes sure that the result is limited as expected.

```
@Test
public void Test_execute_MaxSpeed() {
    MockSwerveModule swerveModuleLeft = new MockSwerveModule();
    swerveModuleLeft.setMaxSpeed_in_s(10.0);
    swerveModuleLeft.setModulePos_in(
        Vector2D.FromXY(0.0, 10.0)
    );
    MockSwerveModule swerveModuleFront = new MockSwerveModule();
    swerveModuleFront.setMaxSpeed_in_s(10.0);
    swerveModuleFront.setModulePos_in(
        Vector2D.FromXY(10.0, 0.0)
    );
    MockSwerveModule swerveModuleRight = new MockSwerveModule();
    swerveModuleRight.setMaxSpeed_in_s(10.0);
    swerveModuleRight.setModulePos_in(
        Vector2D.FromXY(0.0, -10.0)
    );

    ArrayList<ISwerveModule> swerveModules = new ArrayList<>(
        Arrays.asList(swerveModuleLeft, swerveModuleFront, swerveModuleRight));
    RobotOrientedSwerve test = new RobotOrientedSwerve(swerveModules);

    test.execute(Vector2D.FromXY(10.0, 0.0), 1.0);

    IVector2D resultLeft = swerveModuleLeft.getVelocityCommand_in_s_rad();
    assertEquals(resultLeft.getMagnitude(), 0.0, 0.001);

    IVector2D resultFront = swerveModuleFront.getVelocityCommand_in_s_rad();
    assertEquals(resultFront.getMagnitude(), 7.071, 0.001);
    assertEquals(resultFront.getAngleRadians(), Math.toRadians(45.0), 0.001);

    IVector2D resultRight = swerveModuleRight.getVelocityCommand_in_s_rad();
    assertEquals(resultRight.getMagnitude(), 10.0, 0.001);
    assertEquals(resultRight.getAngleRadians(), 0.0, 0.001);
}
```

## Implementing the Max Speed Feature

In order to compensate for the maximum speed of the swerve modules, the `execute()` function must now be broken down into 3 steps:

1. Calculate
    a. Compute the "unlimited" velocity command for each swerve module
    b. Compare the unlimited velocity command to each module's max. velocity to compute the "speed factor", where speed factor = unlimited speed / max. speed
2. Find Maximum
    a. Find the maximum speed factor of all values computed in 1b
    b. If the maximum speed factor is less than 1, assume it's 1
3. Apply Limits
    a. Compute "limited" velocity commands for each swerve module by dividing the unlimited speed by the maximum speed factor found in step 2
    b. Send the limited velocity commands to the swerve modules

There are many different ways to accomplish this and all are unfortunately a little bit complicated. To solve this, we're doing to use a pattern called a "wrapper" class. That is, we're going to create a private class inside of `RobotOrientedSwerve` called `SwerveModuleWrapper`, and use it in place of `ISwerveModule` throughout the class. That will make things a little simpler.

Start by creating the new class (remember it's inside `RobotOrientedSwerve`):

```
    class SwerveModuleWrapper {

    }
```

In the constructor, we take a reference to an ISwerveModule, store that in a class variable, and we're also going to call some methods on the swerve module object and cache those values locally since the max speed and location don't change during program execution:

```
        private final ISwerveModule swerveModule;
        private final double maxSpeed_in_s;
        private final IVector2D positionVector;
        private final Vector2D unlimitedVelocity = new Vector2D();

        public SwerveModuleWrapper(
                ISwerveModule swerveModule) {
            this.swerveModule = swerveModule;
            this.maxSpeed_in_s = swerveModule.getMaxSpeed_in_s();
            this.positionVector = swerveModule.getModulePos_in();
        }
```

Note that we also created a new Vector2D class variable that we're going to use to store results from our calculations (between step 1 and step 2).

Since we're going to use this class in place of the existing `ISwerveModule` references, including in the `allHomed()` function, we need to "surface" the `isHomed()` property. This is common when writing wrapper classes:

```
public boolean isHomed() {
    return this.swerveModule.isHomed();
}
```

Now write a method on our new wrapper class that implements step 1, from above (namely calculating the unlimited velocity command). This code is mostly copied from our original implementation of `execute()`:

```
public void calculateUnlimitedVelocityCommand(
        IVector2D translationCommand_in_s_rad,
        double targetRotationRate_rad_s) {

    double rotation_speed_in_s
        = this.positionVector.getMagnitude() * targetRotationRate_rad_s;
    double rotation_angle_rad
        = this.positionVector.getAngleRadians() + Math.PI / 2.0;
    IVector2D rotation_in_s_rad = Vector2D.FromPolar(
        rotation_speed_in_s, rotation_angle_rad);
        unlimitedVelocity.add(translationCommand_in_s_rad, rotation_in_s_rad);
}
```

For step 2, we need to return the "speed factor":

```
public double getSpeedFactor() {
    double speed = this.unlimitedVelocity.getMagnitude();
    if(speed > this.maxSpeed_in_s) {
        return speed / this.maxSpeed_in_s;
    }
    return 1.0;
}
```

… and finally, for step 3 we need to apply the max. speed factor to create a limited velocity command, and call `execute()` on the ISwerveModule with the limited velocity as the parameter:

```
public void applyAndExecute(double maxSpeedFactor) {
    double oldSpeed = this.unlimitedVelocity.getMagnitude();
    double newSpeed = oldSpeed / maxSpeedFactor;
    Vector2D limitedVelocity = new Vector2D();
    limitedVelocity.copy(this.unlimitedVelocity);
    limitedVelocity.setMagnitude(newSpeed);
    this.swerveModule.execute(limitedVelocity);
}
```

Now that we have the `SwerveModuleWrapper` class, it's time to revisit the constructor of the `RobotOrientedSwerve` class and replace our list of `ISwerveModules` with a list of `SwerveModuleWrapper`.

```
    private final ArrayList<SwerveModuleWrapper> swerveModules;

    public RobotOrientedSwerve(
            ArrayList<ISwerveModule> swerveModules) {
        this.swerveModules = new ArrayList<>(swerveModules.size());
        for(ISwerveModule swerveModule : swerveModules) {
            this.swerveModules.add(new SwerveModuleWrapper(swerveModule));
        }
    }
```

Return to the `execute()` method and remove the `for` loop. All that should be left is the `if` block that checks if all the modules are homed. After the `if` block, we're going to add 3 new `for` loops (one loop for each step we defined above). The first loop calls the calculation method:

```
    for(SwerveModuleWrapper swerveModule : this.swerveModules) {
        swerveModule.calculateUnlimitedVelocityCommand(
            translationCommand_in_s_rad,
            targetRotationRate_rad_s);
    }
```

The second loop finds the maximum speed factor:

```
    double maxSpeedFactor = 1.0;
    for(SwerveModuleWrapper swerveModule : this.swerveModules) {
        double speedFactor = swerveModule.getSpeedFactor();
        maxSpeedFactor = Math.max(speedFactor, maxSpeedFactor);
    }
```

The third loop applies the max. speed factor and executes the limited velocity command on all swerve modules:

```
    for(SwerveModuleWrapper swerveModule : this.swerveModules) {
        swerveModule.applyAndExecute(maxSpeedFactor);
    }
```

That's it! The `RobotOrientedSwerve` class is now complete.

## FieldOrientedSwerve Class

Begin by creating an empty `FieldOrientedSwerve` class:

```
package frc.swerve;

import java.util.*;

public class FieldOrientedSwerve implements IFieldOrientedSwerve {

}
```

Then add a constructor, where we have to pass in a reference to an `IRobotOrientedSwerve` and an `IGyro`, along with 3 motion parameters, a scan time parameter, and PID parameters:

```
    private final IRobotOrientedSwerve robotOrientedSwerve;
    private final IGyro gyro;
    private final double maxSpeed_in_s;
    private final double maxAcceleration_in_s2;
    private final double maxRotationRate_rad_s;
    private final double scanTime_s;
    private final double rotationP;
    private final double rotationI;

    public FieldOrientedSwerve(
            IRobotOrientedSwerve robotOrientedSwerve,
            IGyro gyro,
            double maxSpeed_in_s,
            double maxAcceleration_in_s2,
            double maxRotationRate_rad_s,
            double scanTime_s,
            double rotationP,
            double rotationI) {
        this.robotOrientedSwerve = robotOrientedSwerve;
        this.gyro = gyro;
        this.maxSpeed_in_s = maxSpeed_in_s;
        this.maxAcceleration_in_s2 = maxAcceleration_in_s2;
        this.maxRotationRate_rad_s = maxRotationRate_rad_s;
        this.scanTime_s = scanTime_s;
        this.rotationP = rotationP;
        this.rotationI = rotationI;
    }
```

To implement the `IFieldOrientedSwerve` interface, we need to create an execute method:

```
    @Override
    public void execute(
            IVector2D fieldOrientedTranslationCommand_in_s_rad,
            double fieldOrientedHeadingCommand_rad,
            IVector2D robotOrientedTranslationCommand_in_s_rad,
            double targetRotationRate_rad_s) {

    }
```

## MockRobotOrientedSwerve Class

To test the `FieldOrientedSwerve`, we need a `MockRobotOrientedSwerve` class:

```
package frc.swerve;

public class MockRobotOrientedSwerve implements IRobotOrientedSwerve {

    private IVector2D translationCommand_in_s_rad;
    private double targetRotationRate_rad_s = 0.0;

    @Override
    public void execute(
            IVector2D translationCommand_in_s_rad,
            double targetRotationRate_rad_s) {

        this.translationCommand_in_s_rad = translationCommand_in_s_rad;
        this.targetRotationRate_rad_s = targetRotationRate_rad_s;
    }

    public IVector2D getTranslationCommand_in_s_rad() {
        return this.translationCommand_in_s_rad;
    }

    public double getTargetRotationRate_rad_s() {
        return this.targetRotationRate_rad_s;
    }
}
```

## MockGyro Class

Likewise, we need a `MockGryo` class:

```
package frc.swerve;

public class MockGyro implements IGyro {

    private double fieldOrientation_rad = 0.0;

    @Override
    public double getFieldOrientation_rad() {
        return this.fieldOrientation_rad;
    }
    public void setFieldOrientation_rad(
            double fieldOrientation_rad) {
        this.fieldOrientation_rad = fieldOrientation_rad;
    }

    private double rotationRate_rad_s = 0.0;

    @Override
    public double getRotationRate_rad_s() {
        return this.rotationRate_rad_s;
    }
    public void setRotationRate_rad_s(
            double rotationRate_rad_s) {
        this.rotationRate_rad_s = rotationRate_rad_s;
    }

    private boolean enabled = true;

    @Override
    public boolean getEnabled() { return this.enabled; }
    @Override
    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    @Override
    public void resetOffsetToZero_rad() { } // not needed

    @Override
    public void adjustOffset_rad(double adjustment_rad) { }
}
```

## Testing Robot Oriented Commands Pass-through

The `execute()` method expects two sets of parameters. The first pair is field-oriented commands and the second pair is robot-oriented commands. We can start writing simple tests by zeroing out the field-oriented commands and making sure that any robot-oriented commands we issue are passed through directly to the `IRobotOrientedSwerve` instance.

Begin by creating a new `TestFieldOrientedSwerve` class:

```
package frc.swerve;

import org.junit.Test;
import static org.junit.Assert.*;

public class TestFieldOrientedSwerve {

}
```

Then create a unit test helper function that lets us test the robot-oriented pass-through:

```
    private void test_execute_robotOriented_passThrough(
            double translationCommandX_in_s,
            double translationCommandY_in_s,
            double rotationRate_rad_s) {

        MockRobotOrientedSwerve robotOrientedSwerve
            = new MockRobotOrientedSwerve();
        MockGyro gyro = new MockGyro();

        final double MAX_SPEED_IN_S = 40000.0; // about 1 km/s
        final double MAX_ACCELERATION_IN_S2 = 1000000.0; // very high
        final double MAX_ROTATION_RATE_RAD_S = 100.0; // very high
        final double SCAN_TIME_S = 0.02; // typical
        final double ROTATION_P = 0.0;

        FieldOrientedSwerve test = new FieldOrientedSwerve(
            robotOrientedSwerve,
            gyro,
            MAX_SPEED_IN_S,
            MAX_ACCELERATION_IN_S2,
            MAX_ROTATION_RATE_RAD_S,
            SCAN_TIME_S,
            ROTATION_P);

        Vector2D translationCommand_in_s_rad = Vector2D.FromXY(
            translationCommandX_in_s, translationCommandY_in_s);
```

```
        test.execute(
            Vector2D.FromXY(0.0, 0.0),
            0.0,
            translationCommand_in_s_rad,
            rotationRate_rad_s);

        IVector2D translationResult
            = robotOrientedSwerve.getTranslationCommand_in_s_rad();
        double resultRotationRate
            = robotOrientedSwerve.getTargetRotationRate_rad_s();

        assertNotNull(translationResult);
        assertEquals(translationCommandX_in_s, translationResult.getX(), 0.001);
        assertEquals(translationCommandY_in_s, translationResult.getY(), 0.001);
        assertEquals(rotationRate_rad_s, resultRotationRate, 0.001);
    }
```

Notice that we set the maximum speed, acceleration, and rotation rate to very high values. That's because we don't want to test the function of those parameters in this test. By setting the limits very high, they should have no effect.

Next, write a test that uses this helper function to verify that any robot-oriented command we issue will be sent directly to the `IRobotOrientedSwerve` instance:

```
    @Test
    public void Test_execute_robotOriented_passThrough() {
        test_execute_robotOriented_passThrough(0.0, 0.0, 0.0);
        test_execute_robotOriented_passThrough(10.0, 0.0, 0.0);
        test_execute_robotOriented_passThrough(0.0, -5.0, 0.0);
        test_execute_robotOriented_passThrough(0.0, 0.0, 1.0);
        test_execute_robotOriented_passThrough(-8.2, 1.3, -0.86);
    }
```

## Testing Field Oriented Converts to Robot Oriented

The first parameter of the `execute()` method is a field-oriented translation command. The `FieldOrientedSwerve` class converts this field-oriented translation into a robot-oriented translation by applying the field orientation value provided by the `IGyro` instance.

Inside of the `TestFieldOrientedSwerve` class, create a new unit test helper function to let us test this feature:

```
private void test_execute_fieldOrientedConversion(
        double fieldOrientedTranslationCommandX_in_s,
        double fieldOrientedTranslationCommandY_in_s,
        double gyroFieldOrientation_rad,
        double expectedTranslationX_rad_s,
        double expectedTranslationY_rad_s) {

    MockRobotOrientedSwerve robotOrientedSwerve
        = new MockRobotOrientedSwerve();
    MockGyro gyro = new MockGyro();
    gyro.setFieldOrientation_rad(gyroFieldOrientation_rad);

    final double MAX_SPEED_IN_S = 40000.0; // about 1 km/s
    final double MAX_ACCELERATION_IN_S2 = 1000000.0; // very high
    final double MAX_ROTATION_RATE_RAD_S = 100.0; // very high
    final double SCAN_TIME_S = 0.02; // typical
    final double ROTATION_P = 0.0;

    FieldOrientedSwerve test = new FieldOrientedSwerve(
        robotOrientedSwerve,
        gyro,
        MAX_SPEED_IN_S,
        MAX_ACCELERATION_IN_S2,
        MAX_ROTATION_RATE_RAD_S,
        SCAN_TIME_S,
        ROTATION_P);

    Vector2D translationCommand_in_s_rad = Vector2D.FromXY(
        fieldOrientedTranslationCommandX_in_s,
        fieldOrientedTranslationCommandY_in_s);
    test.execute(
        translationCommand_in_s_rad,
        gyroFieldOrientation_rad, // tell it to face the direction it's facing
        Vector2D.FromXY(0.0, 0.0),
        0.0);

    IVector2D translationResult
        = robotOrientedSwerve.getTranslationCommand_in_s_rad();
    double resultRotationRate
        = robotOrientedSwerve.getTargetRotationRate_rad_s();
```

```
    assertNotNull(translationResult);
    assertEquals(expectedTranslationX_rad_s, translationResult.getX(), 0.001);
    assertEquals(expectedTranslationY_rad_s, translationResult.getY(), 0.001);
    assertEquals(0.0, resultRotationRate, 0.001);
}
```

Next, create a new unit test function to begin writing unit tests for this feature:

```
    @Test
    public void Test_execute_fieldOrientedConversion() {

    }
```

Our first and simplest unit test is no translation command with the gyro at zero (meaning the robot is facing “field forward”):

```
        test_execute_fieldOrientedConversion(0.0, 0.0, 0.0, 0.0, 0.0);
```

As long as the field orientation angle from the gyro (3rd parameter) is zero, then the `FieldOrientedSwerve` class should just pass through the field-oriented commands as robot-oriented commands to the `IRobotOrientedSwerve` instance. Let’s write a few tests to confirm that:

```
        test_execute_fieldOrientedConversion(10.0, 0.0, 0.0, 10.0, 0.0);
        test_execute_fieldOrientedConversion(0.0, -5.0, 0.0, 0.0, -5.0);
        test_execute_fieldOrientedConversion(-2.3, 9.81, 0.0, -2.3, 9.81);
```

Now let’s start varying the field orientation value from the gyro. If we turn the robot 90 degrees counter-clockwise, the gyro will read an angle of PI / 2 radians. When we do that, then a field-oriented forward (+X) command should result in a robot-oriented command to the right (-Y) direction:

```
        test_execute_fieldOrientedConversion(5.0, 0.0, Math.PI/2.0, 0.0, -5.0);
```

We can turn the robot clockwise instead and make sure the robot moves in the positive Y direction:

```
        test_execute_fieldOrientedConversion(5.0, 0.0, -Math.PI/2.0, 0.0, 5.0);
```

Let’s try with the robot turned a full 180 degrees, or PI radians (in either direction) and make sure it commands the robot to move in the negative X robot-oriented direction:

```
        test_execute_fieldOrientedConversion(5.0, 0.0, Math.PI, -5.0, 0.0);
        test_execute_fieldOrientedConversion(5.0, 0.0, -Math.PI, -5.0, 0.0);
```

Then try a 3, 4, 5 triangle rotated clockwise by 90 degrees:

```
    test_execute_fieldOrientedConversion(3.0, 4.0, -Math.PI/2.0, -4.0, 3.0);
```

Make sure we try 45-degree turns as well:

```
    test_execute_fieldOrientedConversion(10.0, 0.0, Math.PI/4.0, 7.071, -7.071);
    test_execute_fieldOrientedConversion(10.0, 0.0, -Math.PI/4.0, 7.071, 7.071);
```

## Testing Rotation PID Control

The 2nd parameter of the execute() method is a field-oriented target heading. For instance, if the current field orientation from the gyro is 90 degrees (PI / 2 radians) and the target heading is zero (field forward) then we want the `FieldOrientedSwerve` class to set the robot-oriented rotation rate to turn the robot towards field forward. In this case that would be a negative rotation rate command, so that the robot would turn clockwise.

To implement this heading control, we've chosen to use a PID controller. The function of the PID controller is to measure how far we are away from our target, and set a control signal (in this case the rotation rate) to a value that will get us closer to the desired target.

The letters P, I, and D refer to the 3 potential components of a PID controller: Proportional, Integral, and Derivative. In this case we're only going to use the P component. This component will cause the target rotation rate sent to the robot-oriented control to increase the further we are away from our target. As we get closer to our target, the rotation rate will decrease towards zero, and when we're perfectly aligned with our target heading, the rotation rate command will be zero.

Create a new unit test helper function that lets us test this feature:

```
    private void test_execute_rotationPID(
            double fieldOrientedHeadingCommand_rad,
            double gyroFieldOrientation_rad,
            double rotationP,
            double maxRotationRate_rad_s,
            double expectedRotationRate_rad_s) {

        MockRobotOrientedSwerve robotOrientedSwerve
            = new MockRobotOrientedSwerve();
        MockGyro gyro = new MockGyro();
        gyro.setFieldOrientation_rad(gyroFieldOrientation_rad);

        final double MAX_SPEED_IN_S = 40000.0; // about 1 km/s
        final double MAX_ACCELERATION_IN_S2 = 1000000.0; // very high
        final double SCAN_TIME_S = 0.02; // typical
```

```
        FieldOrientedSwerve test = new FieldOrientedSwerve(
            robotOrientedSwerve,
            gyro,
            MAX_SPEED_IN_S,
            MAX_ACCELERATION_IN_S2,
            maxRotationRate_rad_s,
            SCAN_TIME_S,
            rotationP);

        test.execute(
            Vector2D.FromXY(0.0, 0.0),
            fieldOrientedHeadingCommand_rad,
            Vector2D.FromXY(0.0, 0.0),
            0.0);

        IVector2D translationResult
            = robotOrientedSwerve.getTranslationCommand_in_s_rad();
        double resultRotationRate
            = robotOrientedSwerve.getTargetRotationRate_rad_s();

        assertNotNull(translationResult);
        assertEquals(0.0, translationResult.getX(), 0.001);
        assertEquals(0.0, translationResult.getY(), 0.001);
        assertEquals(expectedRotationRate_rad_s, resultRotationRate, 0.001);
    }
```

Just for interest sake: the PID component “P” has units of Hz, or 1/s. That’s because, we’re going to measure how far we are away from our target, also known as our “error” (in radians) and we’re going to multiply that by our P value and that will give us our rotation rate in rad/s. If the error is in radians, and we multiply that by P to get a value in rad/s, then clearly P has units of “per second,” or 1/s. The unit of 1/s is also known by the more common unit, hertz (Hz).

So, let’s write some unit tests to check this. We’ll start by using a P value of 1.0, in which case our rotation rate in rad/s should be equal to how far we are away from our target. Start with the case where we’re pointed a field forward, and our target is field forward, meaning the rotation rate should be zero:

```
    @Test
     public void Test_execute_rotationPID() {
        test_execute_rotationPID(0.0, 0.0, 1.0, Math.PI, 0.0);
     }
```

Next, leave the gyro pointed at field forward and set the target heading to 45 (PI/4 radians) degrees counter-clockwise. That should cause a rotation of PI/4 rad/s in the counter-clockwise (positive) direction:

```
        test_execute_rotationPID(Math.PI/4.0, 0.0, 1.0, Math.PI, Math.PI/4.0);
```

If the set the target in the other direction, the rotation rate should also change sign:

```
        test_execute_rotationPID(-Math.PI/4.0, 0.0, 1.0, Math.PI, -Math.PI/4.0);
```

Now, let's cut the P value in half and make sure the rotation rate drops by a factor of ½:

```
        test_execute_rotationPID(Math.PI/4.0, 0.0, 0.5, Math.PI, Math.PI/8.0);
        test_execute_rotationPID(-Math.PI/4.0, 0.0, 0.5, Math.PI, -Math.PI/8.0);
```

The 4th parameter of our test helper function is the maximum allowed rotation rate. Let's try setting the error to PI / 2 and then set the rotation rate limit to PI / 4, and make sure it's limited:

```
    test_execute_rotationPID(Math.PI/2.0, 0.0, 1.0, Math.PI/4.0, Math.PI/4.0);
    test_execute_rotationPID(-Math.PI/2.0, 0.0, 1.0, Math.PI/4.0, -Math.PI/4.0);
```

Up until now we've used a field orientation of zero. Let's start moving the field orientation around and making sure the rotation command goes in the right direction. A good example is setting the field orientation to ¾ of PI (135 degrees), and the target to -¾ of PI (-135 degrees) and making sure the rotation rate is set to positive (that it decides to turn the short way, not the long way). I also doubled the P value so the rate should be PI rad/s.

```
test_execute_rotationPID(-0.75*Math.PI, 0.75*Math.PI, 2.0, Math.PI, Math.PI);
test_execute_rotationPID(0.75*Math.PI, -0.75*Math.PI, 2.0, Math.PI, -Math.PI);
```

## Testing Combined Commands

Next, write a test that combines robot- and field-oriented commands together. This is a long but fairly simple test. We set the robot facing field forward, so robot- and field-orientation are aligned. The two command vectors should simply add, and given a rotation P of 1.0, the rotation rates should add as well.

```
@Test
public void Test_execute_combined() {
    MockRobotOrientedSwerve robotOrientedSwerve
        = new MockRobotOrientedSwerve();
    MockGyro gyro = new MockGyro();
    gyro.setFieldOrientation_rad(0.0);

    final double MAX_SPEED_IN_S = 40000.0; // about 1 km/s
    final double MAX_ACCELERATION_IN_S2 = 1000000.0; // very high
    final double MAX_ROTATION_RATE_RAD_S = 100.0; // very high
    final double SCAN_TIME_S = 0.02; // typical
    final double ROTATION_P = 1.0;

    FieldOrientedSwerve test = new FieldOrientedSwerve(
        robotOrientedSwerve,
        gyro,
        MAX_SPEED_IN_S,
        MAX_ACCELERATION_IN_S2,
        MAX_ROTATION_RATE_RAD_S,
        SCAN_TIME_S,
        ROTATION_P);

    test.execute(
        Vector2D.FromXY(-1.3, 2.8),
        Math.PI/2.0,
        Vector2D.FromXY(6.55, -7.38),
        -Math.PI/4.0);

    IVector2D translationResult
        = robotOrientedSwerve.getTranslationCommand_in_s_rad();
    double resultRotationRate
        = robotOrientedSwerve.getTargetRotationRate_rad_s();

    double expectedRotationRate = Math.PI/2.0-Math.PI/4.0;
    assertNotNull(translationResult);
    assertEquals(6.55-1.3, translationResult.getX(), 0.001);
    assertEquals(2.8-7.38, translationResult.getY(), 0.001);
    assertEquals(expectedRotationRate, resultRotationRate, 0.001);
}
```

## Testing Maximum Rotation Rate Feature

The `FieldOrientedSwerve` class must impose a maximum rotation rate after combining the field- and robot-oriented components of motion. The typical case for limiting maximum rotation rate is during testing, when you want to limit motion to a "safe" (or at least reasonable) limit.

Create a unit test helper class to test this feature. Again, we have the robot facing forward, with a rotation P of 1.0, meaning that our field-oriented heading commands should create a rotation command of equal magnitude, and we also specify the robot rotation command, which will add together. We then specify a maximum, and an expected result:

```
private void test_execute_maximumRotationRate(
        double fieldRotation_rad_s,
        double robotRotation_rad_s,
        double maxRotationRate_rad_s,
        double expectedRotationRate_rad_s) {

    MockRobotOrientedSwerve robotOrientedSwerve
        = new MockRobotOrientedSwerve();
    MockGyro gyro = new MockGyro();

    final double MAX_SPEED_IN_S = 40000.0; // about 1 km/s
    final double MAX_ACCELERATION_IN_S2 = 1000000.0; // very high
    final double SCAN_TIME_S = 0.02; // typical
    final double ROTATION_P = 1.0;

    FieldOrientedSwerve test = new FieldOrientedSwerve(
        robotOrientedSwerve,
        gyro,
        MAX_SPEED_IN_S,
        MAX_ACCELERATION_IN_S2,
        maxRotationRate_rad_s,
        SCAN_TIME_S,
        ROTATION_P);

    test.execute(
        Vector2D.FromXY(0.0, 0.0),
        fieldRotation_rad_s,
        Vector2D.FromXY(0.0, 0.0),
        robotRotation_rad_s);

    double resultRotationRate
        = robotOrientedSwerve.getTargetRotationRate_rad_s();
    assertEquals(expectedRotationRate_rad_s, resultRotationRate, 0.001);
}
```

Start by creating a unit test method with zero commands, and a zero expected result, and a rotation limit of 1 rad/s:

```
    @Test
    public void Test_execute_maximumRotationRate() {
        test_execute_maximumRotationRate(0.0, 0.0, 1.0, 0.0);
        test_execute_maximumRotationRate(1.0, -1.0, 1.0, 0.0);
    }
```

Next, let's add some tests where we provide individual commands of 1 rad/s (both field- and robot-oriented) and make sure the output matches our input, since these are all within the 1.0 rad/s limit:

```
        test_execute_maximumRotationRate(1.0, 0.0, 1.0, 1.0);
        test_execute_maximumRotationRate(-1.0, 0.0, 1.0, -1.0);
        test_execute_maximumRotationRate(0.0, 1.0, 1.0, 1.0);
        test_execute_maximumRotationRate(0.0, -1.0, 1.0, -1.0);
```

To test the limit, copy those same tests, and drop the maximum rotation rate (3rd parameter) to 0.5 rad/s and make sure the actual rotation rate (4th parameter) also drops to + or – 0.5 rad/s:

```
        test_execute_maximumRotationRate(1.0, 0.0, 0.5, 0.5);
        test_execute_maximumRotationRate(-1.0, 0.0, 0.5, -0.5);
        test_execute_maximumRotationRate(0.0, 1.0, 0.5, 0.5);
        test_execute_maximumRotationRate(0.0, -1.0, 0.5, -0.5);
```

Now let's add the field- and robot-oriented rotation commands together (1.0 rad/s each, for a sum of + or – 2.0 rad/s) with a maximum rotation rate of 3 rad/s, and make sure the output matches the total:

```
        test_execute_maximumRotationRate(1.0, 1.0, 3.0, 2.0);
        test_execute_maximumRotationRate(-1.0, -1.0, 3.0, -2.0);
        test_execute_maximumRotationRate(1.0, 1.0, 3.0, 2.0);
        test_execute_maximumRotationRate(-1.0, -1.0, 3.0, -2.0);
```

Next, let's copy those same 4 tests but set the maximum rotation rate to 1.0 rad/s, and make sure the output is limited to that value:

```
        test_execute_maximumRotationRate(1.0, 1.0, 1.0, 1.0);
        test_execute_maximumRotationRate(-1.0, -1.0, 1.0, -1.0);
        test_execute_maximumRotationRate(1.0, 1.0, 1.0, 1.0);
        test_execute_maximumRotationRate(-1.0, -1.0, 1.0, -1.0);
```

## Test Maximum Speed Feature

The FieldOrientedSwerve class also needs to enforce a maximum robot speed. Again, this is generally used to limit the robot speed during testing, but you also need to limit speed if you have a limited acceleration/deceleration rate, or else you will get going so fast that you can't stop in a reasonable distance.

Again, create a unit test helper function. This one takes a field- and a robot-oriented speed and uses them as the X components of the two command vectors. We apply a given maximum speed, and make sure the result speed is whatever we predict.

```
    private void test_execute_maximumSpeed(
            double fieldSpeed_in_s,
            double robotSpeed_in_s,
            double maxSpeed_in_s,
            double expectedSpeed_in_s) {

        MockRobotOrientedSwerve robotOrientedSwerve
            = new MockRobotOrientedSwerve();
        MockGyro gyro = new MockGyro();

        final double MAX_ACCELERATION_IN_S2 = 1000000.0; // very high
        final double MAX_ROTATION_RATE_RAD_S = 100.0; // very high
        final double SCAN_TIME_S = 0.02; // typical
        final double ROTATION_P = 1.0;

        FieldOrientedSwerve test = new FieldOrientedSwerve(
            robotOrientedSwerve,
            gyro,
            maxSpeed_in_s,
            MAX_ACCELERATION_IN_S2,
            MAX_ROTATION_RATE_RAD_S,
            SCAN_TIME_S,
            ROTATION_P);

        test.execute(
            Vector2D.FromXY(fieldSpeed_in_s, 0.0),
            0.0,
            Vector2D.FromXY(robotSpeed_in_s, 0.0),
            0.0);

        IVector2D translationResult
            = robotOrientedSwerve.getTranslationCommand_in_s_rad();
        assertNotNull(translationResult);
        assertEquals(expectedSpeed_in_s, translationResult.getX(), 0.001);
        assertEquals(0.0, translationResult.getY(), 0.001);
    }
```

Next, let's start creating unit tests with a maximum speed of 10 in/s, but keeping our command speeds well within that maximum:

```
    @Test
    public void Test_execute_maximumSpeed() {
        test_execute_maximumSpeed(0.0, 0.0, 10.0, 0.0);
        test_execute_maximumSpeed(1.0, -1.0, 10.0, 0.0);
        test_execute_maximumSpeed(1.0, 1.0, 10.0, 2.0);
        test_execute_maximumSpeed(-1.0, -1.0, 10.0, -2.0);
    }
```

Then, let's add some more tests. Let's make sure each individual component (field- or robot-oriented component) is limited separately:

```
        test_execute_maximumSpeed(11.0, 0.0, 10.0, 10.0);
        test_execute_maximumSpeed(0.0, 11.0, 10.0, 10.0);
        test_execute_maximumSpeed(-11.0, 0.0, 10.0, -10.0);
        test_execute_maximumSpeed(0.0, -11.0, 10.0, -10.0);
```

Next let's make the sum of the two components greater than the maximum speed, even though each individual component is less than the maximum:

```
        test_execute_maximumSpeed(5.0, 6.0, 10.0, 10.0);
        test_execute_maximumSpeed(-5.0, -6.0, 10.0, -10.0);
```

Finally, let's make the two components oppose each other, just as a sense check:

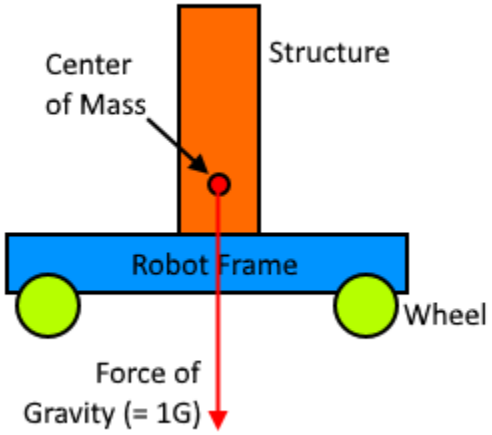
```
        test_execute_maximumSpeed(5.0, -6.0, 10.0, -1.0);
        test_execute_maximumSpeed(-5.0, 6.0, 10.0, 1.0);
```

## The Maximum Acceleration Feature

Robots are tippy. Most years the competition gives you some goal that asks you to reach up high, and tall robots… they tip over.

### Why Limit Acceleration?

When the robot is sitting still, gravity is pulling down from the center of the robot, which is (hopefully) inside of your wheels. Let's look at a robot from the side:




This is a stable configuration. Gravity pulls the robot down, and the wheels are (we hope) sitting firmly on the floor and the floor pushes back up on the wheels. Since the floor is concrete and sitting firmly on the Earth, the floor doesn't move, no matter how many robots you pile on top. To maintain this equilibrium, we can safely assume that the force exerted upwards by the floor is equal and opposite to the force exerted by gravity, and the robot doesn't move.

Next, let's apply a force due to the wheels. Assuming the wheels start a zero speed (not that hard to imagine) and we give them a command to move, such as 10 inches per second, our fancy swerve drive math will eventually tell the `DriveController` to move the drive motor at some RPM, the motor controller (let's assume it's a SparkMAX) will notice that the motor is currently at rest, deduce that it needs to increase the speed, and will apply current to the motor to do so. Electrical current in a motor is roughly proportional to torque, so the motor will try to spin, and since it's connected through various gears to the wheel, the wheel will attempt to start turning.

The wheel is certainly attached to the robot but it's not actually attached to the floor. Like, the wheel is definitely touching the floor, but it's not attached to the floor. As the wheel starts to turn, it exerts a horizontal force on the carpet. The maximum force that it can apply before the wheel loses contact with the carpet and starts to slip has to do with the materials that the wheel and the carpet are made of. Assuming we're using the 4 inch traction wheels that we did last year, and the same type of carpet, our research indicates that the maximum force you can apply is roughly 110% of the force pushing down on the wheel due to gravity. That is to say that it has a "coefficient of friction" of about 1.1.

Interestingly, the heavier the robot, the harder you can push with each wheel before it starts to spin. Note that if the wheel starts to spin relative to the carpet, then the amount of force we're applying drops of drastically, and we can no longer accelerate. Therefore, spinning wheels is bad. We don't want

to exceed the maximum force that we can apply due to "static" friction, which in this case is about 10% more than the force gravity is applying to the wheel from above.

Newton wrote down a useful law of motion that we should refer to now:

$$F=ma$$

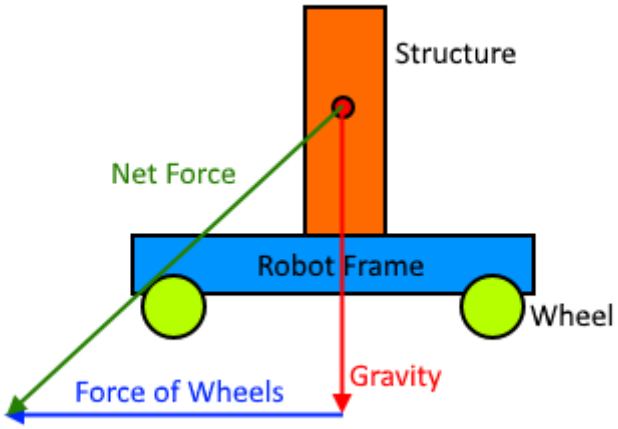
Force = mass x acceleration

Notice a couple of things… all things in free fall experience the same acceleration (9.8 $m/s^2$ or about 385.8 $in/s^2$) and we have a name for that: "1 G". Since the mass of the robot is fixed, and acceleration due to gravity is a constant, then the force of gravity pulling down on that robot is also constant.

The force of gravity is pushing the wheels down onto the carpet, and it's a constant. The maximum force that we can apply horizontally to move the robot around on the field is therefore a constant, and it's roughly equal to 110% of the force due to gravity. This is a property of the wheel and carpet material and there's no way to exceed that. Since force is proportional to acceleration, and our maximum force is 110% of the gravitational force, then it stands to reason that the maximum acceleration of our robot on the field is around 110% of the acceleration due to gravity.

Since the acceleration due to gravity is 1 G, or 385.8 $in/s^2$, that means the maximum acceleration of our robot is always going to be less than or equal to 110% of 385.8, or 1.1G, or around 424.4 $in/s^2$. Assuming we have the highest traction wheels and everyone's running on the same carpet, this is actually a physical limit of the game. Nobody can accelerate faster than this.

Now let's see what happens when we try to accelerate our robot at 1.1G:




We can add the force of gravity (downward) to the force we're applying from the wheels (horizontally) by using our old friend, the vector. Adding them tip-to-tail, we can get the net force on the robot (in green) and unfortunately that net force no longer goes between the wheels. It goes *outside* the wheels. That's bad. That's how robots tip over. So, in this case, we actually need to limit the acceleration to something less than 1.1 G, or our driver's going to have their hands full.

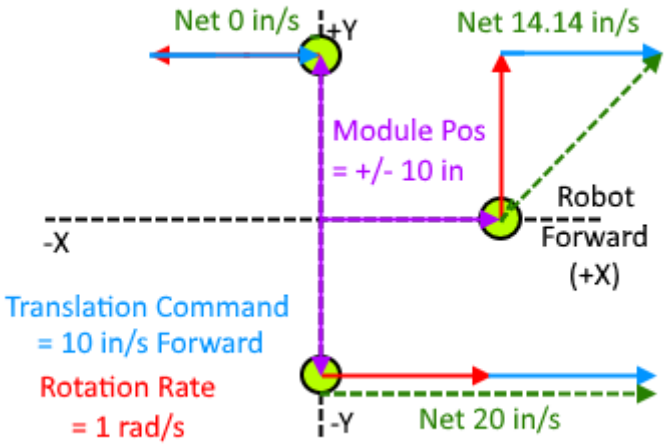
You may have noticed that an easier solution is to just move the center of mass down. Yes, if the center of mass is lower, you can accelerate faster without tipping over. Unfortunately, if the game requires you to reach up high, there's only so much you can do. You can mount your battery as close to the bottom as possible, and keep your motors and other heavy components in the base, if possible, but your gripper mechanism likely weighs a bit, and it's going to be up high sometimes, and that's going to move your center of mass up.

Just to complicate things, first, the center of mass is hard to calculate, and second, if you have an elevator or an arm, the center of mass moves up and down during the game.

In the real world, we can just start at a comfortably low acceleration limit (0.2 to 0.4 G), and we can slowly increase that during driver practice, up until the point where the robot is just about to tip over, and the driver is still comfortable driving it.

### Why Limit Acceleration in a Field-Oriented Frame of Reference?

Let's go back to a diagram from the `RobotOrientedSwerve` class:




If you recall, we said that in this situation, the robot is actually rotating around the left wheel. Remember that the center of mass of the robot is at coordinates 0,0. If that's the case, then the center of mass is moving in a circle around the left wheel of the robot.

Our robot-oriented translation command and rotation rate commands are constant (10 in/s forward and 1 rad/s counter-clockwise). Those are both speeds… if acceleration is the rate of change of speeds, and those speed commands aren't changing, then is the robot accelerating?

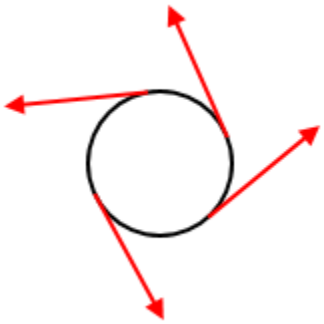
It turns out that it is. In order to move a mass at a constant speed around a circle, you have to constantly accelerate the mass towards the center of the circle. There's a fairly simple formula for circular acceleration: $v^2/r$, where v is the speed and r is the radius. We know the speed here is 10 in/s, and the radius is 10 in, so the acceleration is 10 x 10 / 10 = 10 in/s$^2$.

Most swerve drive programs will attempt to limit acceleration in the robot-oriented frame of reference (if at all). Unfortunately, as we've seen here, that becomes difficult. The root cause of the difficulty is

that our frame of reference (the robot) is itself accelerating, so it's not what physicists call "an inertial frame of reference." If you're standing in an accelerating frame of reference, such as sitting in a car going around a curve, and you do an experiment, you'll get different results than if you were standing on the side of the road.

Thankfully we do have an inertial frame of reference handy: the field. The field is not accelerating (well, not enough to worry about) and we have a gyro that we can use to convert the robot-oriented motion into field-oriented motion, and limit our acceleration in *that* frame.

What do the velocity vectors of a robot moving in a circle look like from the field frame of reference?

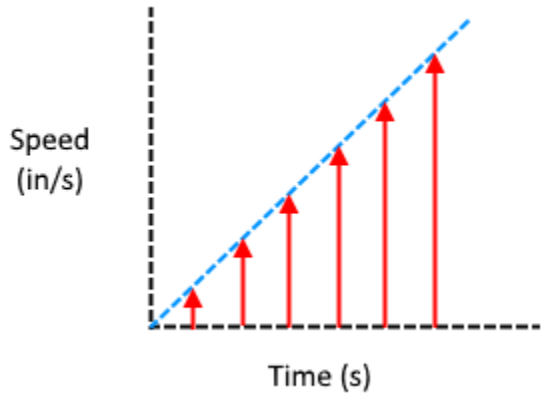


As the robot moves around the circle, the magnitude (speed) of the vector remains the same, but the direction is constantly changing.

## Acceleration as Speed Changing Over Time

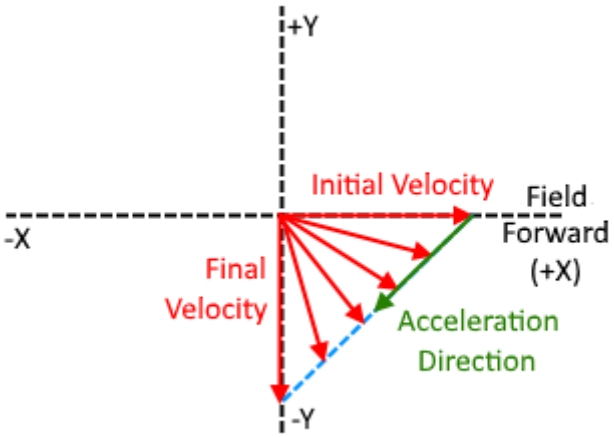
Understanding acceleration as the rate of change of speed over time is key. If you want to accelerate from a standstill to 10 in/s at 1 in/s$^2$, that'll take you 10 seconds, but if you can accelerate at 10 in/s$^2$, then you could get up to speed in 1 second (but it would take 10 times the force, 10 times the torque, and 10 times the electrical current, and so on).

What does it look like when a speed (or velocity) changes over time? Let's say we begin from a standstill and we want to accelerate to 10 in/s over 10 seconds. If we looked at the velocity every second, it would look like this:

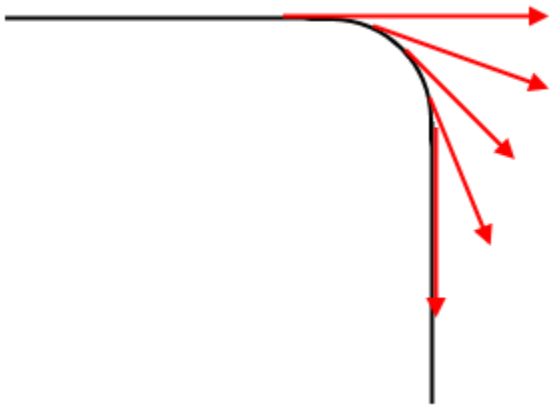



If we wanted to *de*celerate, it would be the opposite (the arrows would decrease in length over time).

Now what if we wanted to change directions? In our swerve drive example, imagine the robot is moving field forward at 10 in/s, and the driver gives the command to move to the right at 10 in/s. The velocity vector has to change from forward (+X) to right (-Y). Let's draw that the same way:



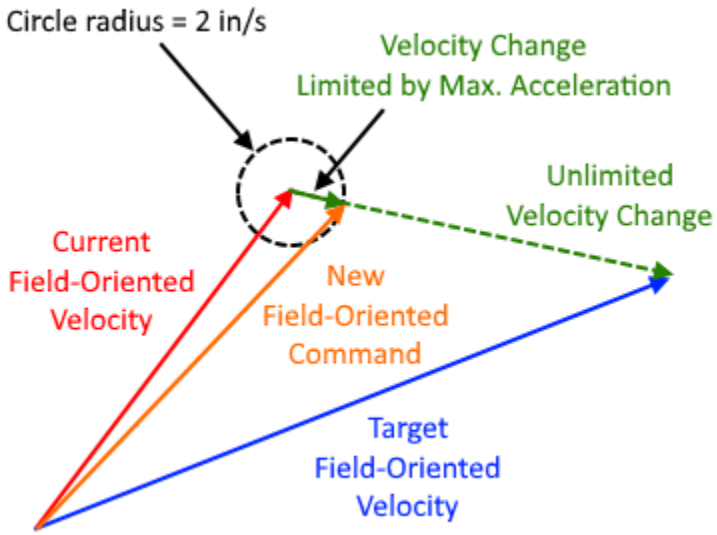
The motion that the robot takes will look like this:



Over time, the X component of the motion goes to zero (we stop moving forward) and the negative Y component increases (we start moving in the field-right direction).

Like these diagrams, our computer program sees “snapshots” of the world. Our robot program runs a continuous loop, and every time through the loop it has to give each subsystem (elevator, intake mechanism, swerve, etc.) a chance to run. In our case, it has to call the `execute()` method on our `FieldOrientedSwerve` class. These loops typically run on a scheduled timing. The default is around 20 ms (or 0.020 seconds) but it can be longer than that if we’re processing a lot of data each time through the loop. We call time interval the “scan time”.

If we only see a snapshot of the world every 0.02 seconds, and we want to limit the acceleration of our robot to, let's say, 100 in/s$^2$, then the maximum change of our field-oriented translation vector should never change by more than 0.02 x 100 in/s$^2$ = 2 in/s. So, to limit acceleration, we need to limit the change of the velocity vector to a circle no larger than 2 in/s every time `execute()` is called:




This is what our `FieldOrientedSwerve` class has to do. We give it the scan time and maximum acceleration in the constructor. We need to calculate the maximum amount that the velocity can change every time we call `execute()`.

Then we need to make sure that the new field-oriented translation command differs from the last field-oriented translation command by no more than this amount. How do we know what the last command was? We'll have to save it as a class variable.

As you can see, this adds some complexity to our tests because to make sure the maximum acceleration feature is working, we probably need to call the `execute()` method multiple times.

### Testing the Maximum Acceleration Feature

In order to test the maximum acceleration feature, we need to provide the maximum acceleration parameter and the scan time parameter, and then we need to give it a target translation velocity (which is a combination of the field-oriented translation command, the robot-oriented translation command, and the field heading from the gyro), and we need to robot-oriented translation command sent to the `IRobotOrientedSwerve` instance is the expected magnitude.

Start by creating a unit test helper function to run this test. In this test we setup all the parameters, call `execute()`, check the result, and then call `execute()` again and check the second result:

```
private void test_execute_maxAcceleration(
        double maxAcceleration_in_s2,
        double scanTime_s,
        double fieldOrientedCommand_in_s,
        double fieldOrientedCommand_rad,
        double robotOrientedCommand_in_s,
        double robotOrientedCommand_rad,
        double gyroFieldOrientation_rad,
        double expectedMagnitude1_in_s,
        double expectedMagnitude2_in_s,
        double expectedAngle_rad) {

    MockRobotOrientedSwerve robotOrientedSwerve
        = new MockRobotOrientedSwerve();
    MockGyro gyro = new MockGyro();
    gyro.setFieldOrientation_rad(gyroFieldOrientation_rad);

    final double MAX_SPEED_IN_S = 40000.0; // about 1 km/s
    final double MAX_ROTATION_RATE_RAD_S = 100.0; // very high
    final double ROTATION_P = 1.0;

    FieldOrientedSwerve test = new FieldOrientedSwerve(
        robotOrientedSwerve,
        gyro,
        MAX_SPEED_IN_S,
        maxAcceleration_in_s2,
        MAX_ROTATION_RATE_RAD_S,
        scanTime_s,
        ROTATION_P);

    Vector2D fieldOrientedCommand = Vector2D.FromPolar(
        fieldOrientedCommand_in_s, fieldOrientedCommand_rad);
    Vector2D robotOrientedCommand = Vector2D.FromPolar(
        robotOrientedCommand_in_s, robotOrientedCommand_rad);

    // First call to execute()
    test.execute(
        fieldOrientedCommand,
        gyroFieldOrientation_rad,
        robotOrientedCommand,
        0.0);
```

```
    IVector2D translationResult1
        = robotOrientedSwerve.getTranslationCommand_in_s_rad();
    assertNotNull(translationResult1);
    assertEquals(expectedMagnitude1_in_s,
        translationResult1.getMagnitude(), 0.001);
    assertEquals(expectedAngle_rad,
        translationResult1.getAngleRadians(), 0.001);

    // Second call to execute()
    test.execute(
        fieldOrientedCommand,
        gyroFieldOrientation_rad,
        robotOrientedCommand,
        0.0);

    IVector2D translationResult2
        = robotOrientedSwerve.getTranslationCommand_in_s_rad();
    assertNotNull(translationResult2);
    assertEquals(expectedMagnitude2_in_s,
        translationResult2.getMagnitude(), 0.001);
    assertEquals(expectedAngle_rad,
        translationResult2.getAngleRadians(), 0.001);
}
```

Next, start by writing a simple unit test where we accelerate forward towards a robot-oriented speed command of 3 in/s. We know that if we use a scan time of 0.02 seconds and a maximum acceleration of 100 in/s$^2$, then after the first call to `execute()`, the result should be a magnitude of 2 in/s, and after the second call it should be 3 in/s:

```
@Test
public void Test_execute_maxAcceleration() {
    test_execute_maxAcceleration(100.0, 0.02, 0.0, 0.0, 3.0, 0.0, 0.0, 2.0, 3.0, 0.0);
}
```

Then try cutting the acceleration rate in half and making sure the resulting magnitudes are 1.0 and 2.0:

```
    test_execute_maxAcceleration(50.0, 0.02, 0.0, 0.0, 3.0, 0.0, 0.0, 1.0, 2.0, 0.0);
```

Similarly, cut the scan time in half and the resulting magnitudes should be 1.0 and 2.0 (because in half the time it should change velocity half as much):

```
    test_execute_maxAcceleration(100.0, 0.01, 0.0, 0.0, 3.0, 0.0, 0.0, 1.0, 2.0, 0.0);
```

Next, turn the robot 90 degrees (PI / 2 radians) to the left, give it a 1 in/s field-oriented command forward and a 1 in/s robot-oriented command forward. The resulting total command in robot-oriented frame should be 1.414 in/s in the direction of 45 degrees to the right (-PI / 4). The result of the first execute will be limited to 1.0 in/s, and the final result will be the full speed of 1.414 in/s:

```
test_execute_maxAcceleration(100.0, 0.01, 1.0, 0.0, 1.0, 0.0, Math.PI/2.0, 1.0, 1.414, -Math.PI/4.0);
```

A couple more tests wouldn't hurt, but I'll stop here.

## Implementing the Execute Method

First, we're going to have to add some class level variables: one for holding a value we'll calculate in the constructor, and a second for holding the last translation vector from scan to scan. Add these inside the class, above the constructor, and after the existing class-level variables:

```
    private final double maximumTranslationChange_in_s;
    private Vector2D lastFieldTranslation = new Vector2D();
```

The first of those variables, `maximumTranslationChange_in_s`, we need to calculate once and set when our constructor runs. Put this right at the end of the existing constructor:

```
    this.maximumTranslationChange_in_s
        = this.maxAcceleration_in_s2 * this.scanTime_s;
```

Next, let's start implementing the `execute()` method. The first thing we want to do is read the gyro field orientation into a variable:

```
    double fieldOrientation_rad = this.gyro.getFieldOrientation_rad();
```

Next, everything has to be converted into field-orientation so that we can limit the acceleration (remember that we have to limit acceleration in field-orientation because it's an inertial frame of reference). First let's take the robot-oriented command and translate that into field-orientation by adding the field orientation angle we read from the gyro. This will create a new vector with the same magnitude but rotated:

```
    // convert the robot-oriented translation into a field-oriented translation
    double robotTranslationMagnitude_in_s =
        robotOrientedTranslationCommand_in_s_rad.getMagnitude();
    double robotTranslationAngle_rad = SwerveUtil.limitAngleFromNegativePItoPI_rad(
        robotOrientedTranslationCommand_in_s_rad.getAngleRadians() + fieldOrientation_rad);
    Vector2D robotTranslationInField =
        Vector2D.FromPolar(robotTranslationMagnitude_in_s, robotTranslationAngle_rad);
```

Next, we can add the converted robot-in-field oriented vector to the field-oriented command and that'll give us a total command in field-orientation:

```
    // combine the robot- and field-oriented translation commands
    Vector2D fieldTranslationTotal = new Vector2D();
    fieldTranslationTotal.add(
        fieldOrientedTranslationCommand_in_s_rad,
        robotTranslationInField);
```

Then we can start applying limits. First, let's apply the maximum speed limit (this just places a limit on the magnitude of the total command vector):

```
    // limit to maximum speed
    if(fieldTranslationTotal.getMagnitude() > this.maxSpeed_in_s) {
        fieldTranslationTotal.setMagnitude(this.maxSpeed_in_s);
    }
```

Then we can limit the acceleration. In order to limit the acceleration, we first have to calculate the difference between the desired field-oriented translation vector, and last scan's value:

```
    // limit acceleration
    Vector2D translationChange = new Vector2D();
    translationChange.subtract(fieldTranslationTotal, lastFieldTranslation);
```

This is how much we'd like to change the field-oriented translation vector by, if we had unlimited acceleration. To limit the acceleration, check the magnitude of this change against the maximum change value we calculated in the constructor, and if the change is greater, set the magnitude to the maximum allowable change:

```
    if(translationChange.getMagnitude() > this.maximumTranslationChange_in_s) {
        translationChange.setMagnitude(this.maximumTranslationChange_in_s);
    }
```

Now that we've calculated the limited change, add the change back to last scan's vector to compute the new field-oriented translation vector:

```
    fieldTranslationTotal.add(this.lastFieldTranslation, translationChange);
```

Then, make sure you save this value into the class-level variable so it can be used next scan:

```
    this.lastFieldTranslation.copy(fieldTranslationTotal);
```

We're done with the translation limiting, so it's time to convert the field-oriented translation back into a robot-oriented translation in preparation for sending that value to the `IRobotOrientedSwerve` instance. To convert from a field-oriented vector to a robot-oriented vector, we subtract the gyro field orientation angle, and keep the same magnitude:

```
    // convert back to robot-oriented
    double totalMagnitude_in_s = fieldTranslationTotal.getMagnitude();
    double totalAngle_rad = SwerveUtil.limitAngleFromNegativePItoPI_rad(
        fieldTranslationTotal.getAngleRadians() - fieldOrientation_rad);
    Vector2D robotTranslationTotal =
        Vector2D.FromPolar(totalMagnitude_in_s, totalAngle_rad);
```

That's everything for the translation component. Next, it's time to move on to the rotational component. Start by implementing the PID controller (in this case, just a P controller) to compute a rotation rate that will bring us towards our target heading. We start by computing how far we are away from our target heading (this is called the "error"):

```
    // rotation PID
    double rotationError_rad
        = fieldOrientedHeadingCommand_rad - fieldOrientation_rad;
    rotationError_rad = SwerveUtil
        .limitAngleFromNegativePItoPI_rad(rotationError_rad);
```

Then we need to multiply the error signal by our "P" gain to get the control signal (yes, our PID controller is literally 3 lines of code):

```
    double rotateToTarget_rad_s = rotationError_rad * this.rotationP;
```

Now we just add that to the raw rotation command we're getting as an input to the `execute()` method:

```
    // total rotation rate
    double totalRotationRate_rad_s =
        rotateToTarget_rad_s + targetRotationRate_rad_s;
```

Now that we have a total rotation rate, we can apply our rotation rate limiting:

```
    // limit to maximum rotation rate
    if(totalRotationRate_rad_s > this.maxRotationRate_rad_s) {
        totalRotationRate_rad_s = this.maxRotationRate_rad_s;
    }
    else if(totalRotationRate_rad_s < -this.maxRotationRate_rad_s) {
        totalRotationRate_rad_s = -this.maxRotationRate_rad_s;
    }
```

So, we have our robot-oriented translation component, and our rotation component, so the only thing that's left is to send that command to the `IRobotOrientedSwerve` instance:

```
    // send the result to the robot-oriented swerve class
    this.robotOrientedSwerve.execute(
        robotTranslationTotal,
        totalRotationRate_rad_s);
```

…and that's it! That's a field-oriented swerve drive.