# Documentation for Customization of FIRST FTC App for Improved Development.

***Release 2016.1***

**Phillip Tischler**

September 13, 2018

Contents

This site contains documentation for pmtischler/ftc_app, a repository containing customizations of the FIRST FTC code for an improved development experience. It is provided as open-source to provide no single team a competitive advantage.

This site contains a series of tutorials for interacting with the FIRST FTC code and the customizations. You shouldn't copy-paste code snippets found here, as that will diminish your learning experience and make it harder to develop new code built on top of these concepts.

CHAPTER 1

# Contents

## Getting Started

This page is intended to quickly go from nothing to a manual gamepad-controlled robot.

### FTC Guides

The following guides will walk you through installing all required software, and setting up all required hardware. By the end of this section you should have a gamepad-controlled robot and understand the ecosystem.

**FTC Game & Season Info** This page contains all the rules and regulations regarding robot construction and operation.

**FTC Technology Information & Resources** This page contains all software related documentation.

**FTC Robot Building Resources** This page contains all the hardware related documentation.

### Git & Github

This section describes Git and Github. By the end of this section you should know how to use Git and Github to download the base robot code, to make local changes, to commit those changes so you can go back to previous versions, and to push those changes to the remote server.

**Version Control Systems (VCS)** Professional software organizations use VCS to organize teams of Software Engineers working on a common project. It provides a means to backup the software remotely, look at the changes to code over time, revert back to a previous state, etc.

**Git** Git is a VCS that many companies and open source projects have decided to use. It is used by the FIRST FTC Competition and by pmtischler/ftc_app to release the base code for the robot.

**Github** Github is a company which hosts Git projects. It is the most popular choice for open source projects, and is used both by the competition and this page. In addition to a remote server for the Git repository (repo), it adds web-based systems for access-control, code reviews, Wiki pages, etc.

#### Learn Git

The following can be used to learn Git and Github.

**Github Hello World** This page will walk you through creating a Git repo hosted by Github, creating branches, changing and committing files, and performing a Pull Request (code review).

**Github Desktop App** This app simplifies working with Git repositories hosted by Github. Download and install it. Note this may already be installed by your teacher.

**Hello World with Desktop App** The Github Hello World tutorial walks you through the process using their web interface. Typically you do things using local command-line tools, or with the Desktop App. After completing the tutorial through the web interface, try editing files locally, commiting the changes with the Desktop App, and pushing the changes with the Desktop App instead. When it comes time to develop the robot's code, you'll use Android Studio to make changes, and the Github Desktop App to save changes in the cloud.

**Git Book** This page contains a book on all things Git. It is important to understand how Git works so you don't accidentally delete your code, and so that you can use the Git system to develop smarter and faster. Read through this book. You don't need to implement the things it describes.

### Fork pmtischler/ftc_app

The pmtischler/ftc_app is a Git repo hosted on Github. The repo is derived from the FIRST FTC repo, and builds on top of it to add additional robot features to improve the development process for teams.

Teams should fork this repo, or create a new repo which is initialized with this original repo. The team can then configure a remote upstream so that teams can synchronize the fork to pull changes from the original repo into the fork repo. Note that this process may already have been done by your teacher.

### Team Repo

**Clone Team Repo**. Once the team has a personal repo that will contain the code the team develops, and is initialized with the base robot code, the team can `clone` the repo locally, which will copy it to your machine. This can be done with the Github Desktop App.

**Import Project**. The next step is to open the Android Project in the repo to start making code changes. Open Android Studio, select `Import Project (Gradle)`, and select the folder for the repo. This will load the project using the `Gradle (*.gradle)` files in the repository which have been set up for you.

**Edit a File**. In Android Studio, use the `Project` pane to open the file `TeamCode/java/org.firstinspires.ftc.teamcode`. Edit the first line of this file to add your team name:

```
## TeamCode Module (YOUR TEAM NAME)
```

**Commit & Push**. Save the file you edited. Then use the Github Desktop App to commit the file and push the changes to the server. Once you've done it correctly, you should be able to see the changes at `https://github.com/USERNAME/REPONAME/blob/master/TeamCode/src/main/java/org/firs` (replace `USERNAME` and `REPONAME` based on how you forked).

You've successfully used your team repo to store changes to your code. If you were to switch computers (e.g. original machine broke, you use multiple computers, you have multiple team members), you could repeat the clone process to get your code back. You can `pull` from the repo to get any `push` made from another computer.

## Robot Program

Let's create the first Robot program which simply makes a motor run at full speed.

**Add a Robot Class**. In Android Studio, open the `Project` pane, right click on `TeamCode/java/org.firstinspires.ftc.teamcode`, and click `New > Java Class`. Set the name field to `FullPower` and the superclass to `com.qualcomm.robotcore.eventloop.opmode.OpMode`. This will add the file and open it in the editor.

**Add Imports**. Under the `package` line, add the following imports. This will allow you to use the code contained in those files.

```
import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;
```

**Register the Robot Class**. The base code uses annotations to register the class as an OpMode, which will allow it to be selected in the robot's UI. Add the annotation so that the class declaration looks like this:

```
@TeleOp(name="FullPower", group="FullPower")
public class FullPower extends OpMode {
    // ...
}
```

**Add Motor Variables**. We need a motor variable in order to refer to it when we set the power. Add the following member variable to the class.

```
private DcMotor motor = null;
```

initialize variables

**Initialize the Robot**. The next step is to initialize the robot and set the robot's motor variable. The `init` function is called when the robot is started. Add the following function to the class.

```
public void init() {
    motor = hardwareMap.dcMotor.get("motor");
}
```

↳ mapping motor onto the robot

**Set the Motor Power**. The next step is to set the motor power. The `loop` function is called repeatedly until the robot is stopped (e.g. match is over). Add tthe following function to the class.

```
public void loop() {
    motor.setPower(1.0);
}
```

-1 = opp direction
0 = no movement
1 = direction

**Configure Robot, Run Program**. The final step is to run the program on the phone, create a robot configuration with a single motor named `motor`, and start the `FullPower OpMode`. If you have connected everything and programmed correctly, you should see the motor spin at full power. The following is the final code you should have.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;

@TeleOp(name="FullPower", group="FullPower")
public class FullPower extends OpMode {
```

```
    private DcMotor motor = null;

    public void init() {
        motor = hardwareMap.dcMotor.get("motor");
    }

    public void loop() {
        motor.setPower(1.0);
    }
}
```

You now have an end-to-end example of programming a robot. Save everything, commit it to Git, and push it so it's in the cloud. From here you can continue to more advanced things in the Tutorials.

# Tutorials

The following are tutorials that will walk you through the base FTC code, and through the customizations in pmtischler/ftc_app. Additional tutorials will be added over time.

## Tank Drive

This tutorial adds tank drive manual controls to a 4-wheel robot. That is, the left joystick will control the left motors and the right joystick will control the right motors. The robot should have 4 wheels: two on the left, and two on the right. Two of those motors should be in the front, and two in the back. You should name the robots the following:

- `leftFront`
- `leftBack`
- `rightFront`
- `rightBack`

**Skeleton**. The following code block contains the standard skeleton of code that every robot program will be based on. Create a new class under `TeamCode/java/org.firstinspires.ftc.teamcode` and call it `TankDrive.java`. Add the following code to the file:

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;

@TeleOp(name="TankDrive", group="TankDrive")
public class TankDrive extends OpMode {

    public void init() {
        // Run once when driver hits "INIT".
        // Robot setup code goes here.
    }

    public void start() {
        // Run once when driver hits "PLAY".
```

```
        // Robot start-of-match code goes here.
    }

    public void loop() {
        // Code executed continuously until robot end.
    }

    public void stop() {
        // Run once when driver hits "STOP" or time elapses.
        // Robot end-of-match code goes here.
    }
}
```

**Motor Variables**. After creating the robot class with the skeleton, the next step is to create variables for the 4 motors. These will later be initialized to the named motors above, and then set to the joystick inputs. Add the following as class member variables.

```
private DcMotor leftFrontMotor = null;
private DcMotor leftBackMotor = null;
private DcMotor rightFrontMotor = null;
private DcMotor rightBackMotor = null;
```

**Robot Initialization**. In the robot initialization step, you need to assign the specific motor values to the variables we created in the last step. We will also set the initial power to zero, so the robot doesn't start moving on power-up before we hit the play button. Replace the `init` function with the following:

```
public void init() {
    leftFrontMotor = hardwareMap.dcMotor.get("leftFront");
    leftBackMotor = hardwareMap.dcMotor.get("leftBack");
    rightFrontMotor = hardwareMap.dcMotor.get("rightFront");
    rightBackMotor = hardwareMap.dcMotor.get("rightBack");

    leftFrontMotor.setPower(0);
    leftBackMotor.setPower(0);
    rightFrontMotor.setPower(0);
    rightBackMotor.setPower(0);
}
```

**Joystick Controls**. The joystick can be accessed through the `gamepad` variable. Tank drive can be implemented by setting motor power to joystick y-axis values. When you push forward on the joystick, the motor power will be set to 100% and the robot will move forward. When you push backward on the youstick, the motor power will be set to -100% and the robot will move backward. With two joystics, each controlling one side of the robot, pushing forward on one joystick while pushing backward on the other joystick will cause the robot to turn in place. Replace the `loop` function with the following:

```
public void loop() {
    double left = gamepad1.left_stick_y;
    double right = gamepad1.right_stick_y;

    leftFrontMotor.setPower(left);
    leftBackMotor.setPower(left);
    rightFrontMotor.setPower(right);
    rightBackMotor.setPower(right);
}
```

**End of Match Stop**. At the end of the match you want the robot to stop moving. If the last call to `loop` at the end of the match had set a non-zero motor power, then the robot will continue to move in

that direction indefinitely, without you being able to stop it. For this reason, we always want to set the end-condition, for all motors to have zero power, in the `stop` function. Replace the `stop` function with the following:

```
public void stop() {
    leftFrontMotor.setPower(0);
    leftBackMotor.setPower(0);
    rightFrontMotor.setPower(0);
    rightBackMotor.setPower(0);
}
```


**Motor Polarity**. If you were to test the program, you would be able to control the motors with the gamepad, but the motors turn direction might be reversed. You can handle this type of problem two ways. First, you could reverse the wire connections (e.g. connect red header to black socket) from the motor to the motor controller, which will reverse the input signal to the motor. This is easier in terms of coding, but harder in terms of wiring. Second, you could modify the code to invert the control signal, so a 100% gamepad signal will set a -100% motor signal. The following shows an example of such:

```
double left = gamepad1.left_stick_y;
leftFrontMotor.setPower(-left);
```

Congratulations, you should now be able to drive a 4-wheel robot using tank-drive style controls!

## Mecanum Drive

Mecanum drive allows the robot to move at any angle and rotate in place. This will make it faster and more effective to maneuver the robot, like lining up to press a button. Mecanum wheels have lower grip, causing it to be less effective to go up/down ramps or climb over obstacles.

The roller component of the wheel should create a diagonal between the front-left and bottom-right, and between the front-right and bottom-left. The following equations can be used to control the mecanum wheels. See Simplistic Control of a Mecanum Drive for details.

The following mathematical description uses a robot frame where $x$ is forward and $y$ is to the left.

$$V_{front,left} = V_d sin\left(-\theta_d + \frac{\pi}{4}\right) - V_\theta$$
$$V_{front,right} = V_d cos\left(-\theta_d + \frac{\pi}{4}\right) + V_\theta$$
$$V_{back,left} = V_d cos\left(-\theta_d + \frac{\pi}{4}\right) - V_\theta$$
$$V_{back,right} = V_d sin\left(-\theta_d + \frac{\pi}{4}\right) + V_\theta$$

| Variable | Description |
|---|---|
| $V_x$ | Motor power for wheel $x$ [-2, 2]. |
| $V_d$ | The desired robot speed [-1, 1]. |
| $\theta_d$ | Desired robot angle while moving [0, $2\pi$]. |
| $V_\theta$ | Desired speed for changing direction [-1, 1]. |

The robot must be controlled by a gamepad with joysticks, so from these joysticks you need to determine $V_d$, $\theta_d$, $V_\theta$. The left joystick's direction will be used for $\theta_d$, and the joystick's magnitude for $V_d$. The right joystick's x-axis value will be used for $V_\theta$. This will provide first-person-shooter style control. The

following equations will provide this for joysticks $J_x$.

$$V_d = \sqrt{J_{left,x}^2 + J_{left,y}^2}$$
$$\theta_d = arctan(-J_{left,x}, J_{lefty})$$
$$V_\theta = -J_{right,x}$$

These functions don't account for bounded outputs. That is, the joystick inputs can yield an output power greater than 100%, which isn't possible. In order to maintain intent of control, we need to preserve the ratios of the motors. We can do this by scaling the motors uniformly by the max magnitude, effectively normalizing to 100%.

$$V_{max} = max_j V_{motor,j}$$
$$V_{motor,i} = \frac{V_{motor,i}}{V_{max}}$$

Mecanum. One of the custom additions in pmtischler/ftc_app is a class to perform the Mecanum calculation. It takes the desired robot speed, velocity angle, and rotation speed. It returns the speeds of each motor after clamping. In order to use it, you'll need to import the code.

```
import com.github.pmtischler.control.Mecanum;
```

**Add Mode Select**. We want the ability to control the robot with mecanum wheels or with regular wheels. Add the following member variable which will select between the two. This will allow you to select the control style by changing this variable, without having to change your code.

```
private final boolean shouldMecanumDrive = true;
```

cannot change this value elsewhere in code

**Add Mecanum Drive**. The mecanum drive is a straightforward implementation of the formulas above. Update the `loop` function with the following code.

```
public void loop() {
  if (shouldMecanumDrive) {
      // Convert joysticks to desired motion.
      Mecanum.Motion motion = Mecanum.joystickToMotion(
                          gamepad1.left_stick_x, gamepad1.left_stick_y,
                          gamepad1.right_stick_x, gamepad1.right_stick_y);

      // Convert desired motion to wheel powers, with power clamping.
      Mecanum.Wheels wheels = Mecanum.motionToWheels(motion);
      leftFrontMotor.setPower(wheels.frontLeft);
      rightFrontMotor.setPower(wheels.frontRight);
      leftBackMotor.setPower(wheels.backLeft);
      rightBackMotor.setPower(wheels.backRight);
  } else {
      // ... previous loop code.
  }
}
```

converts wheels to smooth movement at FULL PWR (no in betweens)

in case the smooth movement code doesn't work, it reverts back to normal, rigid movement

PLAN B

Congratulations, you now have the ability to drive with Mecanum wheels!

Manual
- Records the state of the wheels as you manually push a button

Autonomous
- Plays back the state of the wheels to mimic manual input

## Record & Playback

In this tutorial you will write your first simple autonomous program. This program will record the hardware state during each call to `loop` while the robot is in manual mode. When the program is in autonomous mode, it'll play back that recorded hardware log to mimic the manual operation.

Note this is not a robust autonomous method. It doesn't handle any changes in robot or environment. If the robot changes weight, changes air drag, etc the exact motor powers needed to turn will change, so the manual log previously recorded will not have the same affect. If the environment changes, the robot will not be able to adapt to avoid obstacles, deal with slippery conditions, etc. In future more advanced tutorials we'll learn how to make better autonomy programs.

BlackBox. One of the custom additions in pmtischler/ftc_app is a class to record the robot's hardware state and play it back. That is, it'll record the motors and servo states during Teleop mode, and then playback that recorded log during Autonomous mode. The robot will mimic the actions of the manual driver. This class uses the name of the hardware devices, so if you change the name of the devices you will need to re-record.

**Create Recorded Teleop Class.** First we will create a new class called `RecordedTeleop`. This class will inherit from the Teleop mode you've developed, like `TankDrive`. This will reuse all the code to drive the robot, and we'll extend it to record the hardware.

```
package org.firstinspires.ftc.teamcode;

import android.content.Context;
import com.github.pmtischler.base.BlackBox;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import java.io.FileOutputStream;

@Teleop(name="RecordedTeleop", group="RecordedTeleop")
public class RecordedTeleop extends TankDrive {
    // The output file stream.
    private FileOutputStream outputStream;
    // The hardware recorder.
    private BlackBox.Recorder recorder;
}
```

You'll notice we've imported the `BlackBox` code. We've also imported `Context` and `FileOutputStream` which will be needed to open a file for which to write the data.

**Open File, Construct Recorder**. We will override the `init` function to create a file to write data to, and then create a `BlackBox.Recorder` that will use the file to write hardware data. If this fails, we'll want to print the error and stop the robot, so we wrap the process in a try-catch.

```
public void init() {
    super.init();  // TankDrive teleop initialization.

    try {
        // Open a file named "recordedTeleop" in the app's folder.
        outputStream = hardwareMap.appContext.openFileOutput("recordedTeleop",
                                                             Context.MODE_PRIVATE);
        // Setup a hardware recorder.
        recorder = new BlackBox.Recorder(hardwareMap, outputStream);
    } catch (Exception e) {
        e.printStackTrace();
        requestOpModeStop();
    }
}
```

**Careful: File Overwriting**. The file we created will be created each time the program is run. This means that the second time you run the `OpMode` it'll overwrite the previously recorded file. You need to make sure you backup this file after recording so that you don't accidentally lose a good file.

**Record over Time**. Now we want to record the state of the hardware to the file. The

`BlackBox.Recorder` has a `record` function which does exactly that. We will call it with the current time so the hardware state is written with the timestamp. Similar to setup, if it fails we wan't to print the error and stop the robot.

```
public void loop() {
    super.loop();  // TankDrive teleop control code.

    try {
        // Record the hardware state at the current time.
        recorder.record("leftFront", time);
        recorder.record("rightFront", time);
        recorder.record("leftBack", time);
        recorder.record("rightBack", time);
    } catch (Exception e) {
        e.printStackTrace();
        requestOpModeStop();
    }
}
```

method in recorder class that has a timestamp for the indicated motor's movement

**Close to Flush Contents**. When the Teleop mode is complete we need to close the file so the data is written. If there is a problem we will print the error, but as we are already stopping we don't need to request the `OpMode` stop.

```
public void stop() {
    super.stop();  // TankDrive stop code.

    try {
        // Close the file to write recorded data.
        outputStream.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

You now have a Teleop program which can use your base Teleop code to drive, and can use the `BlackBox` to record telemetry data. Run your program and test it out. Check the App's data folder to see that the file was written, and that it has data (has non-zero file size).

**Create Playback Autonomous Class**. Now it's time to create the autonomous `OpMode` which will playback the previously recorded file. Create a new class called `PlaybackAuto`. This time it will inherit from `OpMode` instead of a Teleop class.

```
package org.firstinspires.ftc.teamcode;

import com.github.pmtischler.base.BlackBox;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import java.io.FileInputStream;

@Autonomous(name="PlaybackAuto", group="PlaybackAuto")
public class PlaybackAuto extends OpMode {
    // The input file stream.
    private FileInputStream inputStream;
    // The hardware player.
    private BlackBox.Player player;
}
```

**Open File, Construct Player**. Similar to the recorder, we will now open the file and construct a player.

```
public void init() {
    try {
        // Open previously written file full of hardware data.
        inputStream = hardwareMap.appContext.openFileInput("recordedTeleop");
        // Create a player to playback the hardware log.
        player = new BlackBox.Player(inputStream, hardwareMap);
    } catch (Exception e) {
        e.printStackTrace();
        requestOpModeStop();
    }
}
```

**Playback Recorded Data**. Similar to the recorder, we will now call `playback` on the player. This will determine the state of the robot when it was recording at this time in the match, and then set the motors and servos to those values.

↳ can work w/ servos

```
public void loop() {
    try {
        // Update the hardware to mimic human during recorded Teleop.
        player.playback(time);
    } catch (Exception e) {
        e.printStackTrace();
        requestOpModeStop();
    }
}
```

**Close the File**. At the end of the mode, close the file. This is necessary to terminate cleanly, so the robot can run multiple times without error.

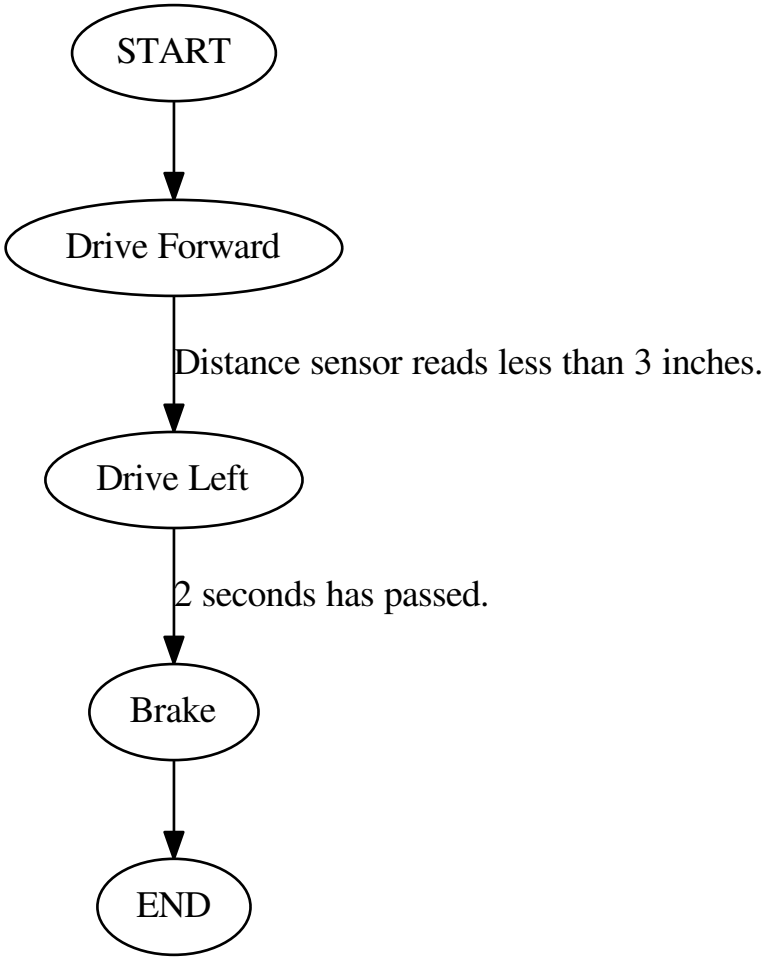
```
public void stop() {
    try {
        inputStream.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Congratulations! You now have an autonomous mode that can mimic a human driver. Test out the mode, see if it accurately mimics the driver. Does it do the same thing every time? How sensitive is it to initial conditions (e.g. initial heading or position on the field)?

## State Machine

In this tutorial you will implement a Finite State Machine. A state machine is made up of a series of states, and a series of transitions between states. A state can be something like "drive forward", and a transition can be something like "when a distance sensor reads less than 3 inches then change to the drive for time state". In this tutorial, we will implement an autonomous mode that drives forward until the distance sensor reads less than 3 inches, after which it will drive left for 2 seconds. It uses the previous Mecanum Drive tutorial for driving, and requires a *setDrive(vD, vTheta, thetaD, vTheta)* function.

**Visualization**. State Machines are typically visualized as a Graph. A graph node is a state in the state machine, and an edge is a transition. The text on the node indicates what the state does, and the text on the edge indicates when the transition occurs. Here is the visualization of the state machine we will make:




**Program Structure**. The first step is to create a skeleton of the final program. Notice that there are two internal classes representing the states of the state machine, one for moving forward until a distance threshold is met, and another which moves left for an amount of time. Each state has 2 functions, *start* which is called when the state machine first enters the state, and *update* which updates the robot in the state and returns the next state to be in. The state machine transitions when the current state's *update* function returns a different state.

```
import com.github.pmtischler.base.StateMachine;
import com.github.pmtischler.base.StateMachine.State;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.hardware.DistanceSensor;
import org.firstinspires.ftc.robotcore.external.navigation.DistanceUnit;

@Autonomous(name="ForwardThenLeftStateMachine", group="ForwardThenLeftStateMachine")
public class ForwardThenLeftStateMachine extends MecanumDrive {

    /**
     * Moves forward until a distance threshold is met.
     */
```

two specific imports for this

```
    public class ForwardUntilDistance implements StateMachine.State {
        @Override
        public void start() {
        }

        @Override
        public State update() {
        }
    }

    /**
     * Moves left for a specific amount of time.
     */
    public class LeftForTime implements StateMachine.State {
        @Override
        public void start() {
        }

        @Override
        public State update() {
        }
    }

    /**
     * Initializes the state machine.
     */
    public void init() {
    }

    /**
     * Runs the state machine.
     */
    public void loop() {
    }

    // Distance sensor reading forward.
    DistanceSensor distanceSensor;

    // The state machine manager.
    private StateMachine machine;
    // Move forward until distance.
    private ForwardUntilDistance forwardUntilDistance;
    // Move left for time.
    private LeftForTime leftForTime;

}
```

**Setup & Run State Machine**. The next step is to setup and run the state machine. Setup involves instantiating each state, creating a state machine manager, and providing it with an initial state. Running involves telling the state machine manager to update, which will call the current state's *update* function, and track the returned state for the next update.

```
public void init() {
    super.init();  // Initialize mecanum drive.

    // Create the states.
    forwardUntilDistance = new ForwardUntilDistance();
```

```
    leftForTime = new LeftForTime();
    // Start the state machine with forward state.
    machine = new StateMachine(forwardUntilDistance);
}

public void loop() {
    machine.update();  // Run one update in state machine.
}
```

**Define Drive Until Distance State**. The next step is to define the first state, driving forward until a distance is met. There is no initialization needed for the state. Updating the state involves checking the distance sensor. If we haven't yet reached the desired distance, we command the motors to drive forward. If we have reached the desired distance, we return the next state: driving left until for an amount of time.

```
public class ForwardUntilDistance implements StateMachine.State {
    @Override
    public void start() {
    }

    @Override
    public State update() {
        if (distanceSensor.getDistance(DistanceUnit.INCH) > 3) {
            // Haven't yet reached distance, drive forward.
            setDrive(1, 0, 0); method w/ parameters
            return this;       (Drive forward)
        } else {
            // Reached distance, switch to left for time state.
            setDrive(0, 0, 0);
            return leftForTime; switch states
        }
    }
}
```

**Define Drive For Time State**. The next step is to define the second state, driving left for an amount of time. To initialize the state, save the time the state started. To update the state, drive left. The state is done when the difference between current time and start time is greater than 2 seconds. The effect will be to drive left for 2 seconds.

```
public class LeftForTime implements StateMachine.State {
    @Override
    public void start() {
        startTime = time;
    }

    @Override
    public State update() {
        if (time - startTime < 2) {
            // Less than 2 seconds elapsed, drive left.
            setDrive(1, Math.PI, 0);
            return this;
        } else {
            // 2 seconds elapsed, stop and terminate state machine.
            setDrive(0, 0, 0);
            return null; return nothing aka STOP moving
        }
    }
```

```
    private double startTime;
}
```

instance variable that indicates the beginning of a state's running

Congratulations! You can now use a state machine to define multiple robot states and when to transition between them. You can use this to implement much more complex state machines for complex autonomous behavior.

## PID Control

In this tutorial you will use a PID Controller to dynamically pick a motor power to achieve a specific motor speed. This can be used to remove the linearity assumption in the `feedback_linearization` tutorial, and to automatically adapt to changes in robot weight, air drag, etc.

PID Control. PID is a control feedback mechanism which standards for proportional integral derivative. It can help optimize situations where you have a variable of interest which you can measure, say speed of the motor, and a separate variable which you can control to achieve a speed, say motor power. The PID control takes as input the error signal $e(t)$, say difference between actual and desired motor speed, and a series of tuning constants. PID control then outputs the manipulated variable $O(t)$. The standard form of the PID equation:

$$O(t) = K_p \left( e(t) + \frac{1}{T_i} \int_0^t e(t)dt + T_d \frac{d}{dt} e(t) \right)$$

| Variable | Description |
|---|---|
| $t$ | Time |
| $O(t)$ | Output over time (e.g. motor power) |
| $e(t)$ | Error of desired variable (e.g. motor speed) |
| $T_i$ | Eliminate sum of errors within this time. |
| $T_d$ | Account for predicted error this time into future. |
| $K_p$ | Gain to control rise-time vs. overshoot. |

- essentially makes code adaptable for all scenarios involving the robot (i.e. adding weight)
- doesn't require robot to be fully built in order to accurately code

With the standard form, you set values for $T_i$ and $T_p$ based on desired behavior, and then you tune $K_p$ to an optimal value for your problem.

There is one common edge-case you need to think about. Let's say we were using Pid to control motor power to achieve a drive speed. If we held the robot in place to prevent the robot from achieving a desired speed, the integral term of Pid would grow continuously. When you release the robot, the integral term will dominate the others and prevent it from stopping. This phenomena is called Integral Windup. You can prevent this from happening by limiting the integral term to specific bounds. In practice you pick values which the integral term shouldn't need to go beyond, while simultaneously reacting fast enough once the robot is unblocked.

**Motor Speed via Encoders**. To use PID to control a motor's speed, you need to measure the actual motors speed. You can measure motor speed using an encoder which measures the rotations per second, which can be obtained via `DcMotor.getCurrentPosition()`. Add the following member variables and code to compute motor speed. `ticksPerRevolution` is a constant you can obtain from your the motor's encoder datasheet, and it specifies the number of encoder positions in a single shaft revolution.

encoders essentially return info about the motor

# of values an encoder can return for 1 revolution (full circle)

```
private final double ticksPerRevolution = 1000;  // Get for your motor and gearing.
private double prevTime;  // The last time loop() was called.
private int prevLeftEncoderPosition;   // Encoder tick at last call to loop().
private int prevRightEncoderPosition;  // Encoder tick at last call to loop().
```

```
public void init() {
    // ... other code
    prevTime = 0;
    prevLeftEncoderPosition = leftFrontMotor.getCurrentPosition();
    prevRightEncoderPosition = RightFrontMotor.getCurrentPosition();
}

public void loop() {
    // ... other code

    // Compute speed of left,right motors.
    double deltaTime = time - prevTime;
    double leftSpeed = (leftFrontMotor.getCurrentPosition() - prevLeftEncoderPosition) /
                       deltaTime;
    double rightSpeed = (rightFrontMotor.getCurrentPosition() - prevRightEncoderPositior
                        deltaTime;
    // Track last loop() values.
    prevTime = time;
    prevLeftEncoderPosition = leftFrontMotor.getCurrentPosition();
    prevRightEncoderPosition = rightFrontMotor.getCurrentPosition();

    // ... other code
}
```

**PID Constants**. Per the equation, there are three constants to control operation of a PID controller. You will need to tune values for your specific use case. Add the following member variables to the code.

```
private final double drivePidKp = 1;     // Tuning variable for PID.
private final double drivePidTi = 1.0;   // Eliminate integral error in 1 sec.
private final double drivePidTd = 0.1;   // Account for error in 0.1 sec.
// Protect against integral windup by limiting integral term.
private final double drivePidIntMax = maxWheelSpeed;  // Limit to max speed.
private final double driveOutMax = 1.0;  // Motor output limited to 100%.
```

prevents the robot from winding up power

**PID Controller**. The custom code provides a Pid class. This will be used to achieve a specific motor speed while controlling motor power. Add the following code to use the Pid control.

```
// ... other code
import com.github.pmtischler.control.Pid;

@TeleOp(name="TankDrive", group="TankDrive")
public class TankDrive extends OpMode {
    // ... other code
    private Pid leftDrive = null;
    private Pid rightDrive = null;

    public void init() {
        // ... other code
        leftDrive = new Pid(drivePidKp, drivePidTi, drivePidTd,
                            -drivePidIntMax, drivePidIntMax,
                            -driveOutMax, driveOutMax);
        rightDrive = new Pid(drivePidKp, drivePidTi, drivePidTd,
                             -drivePidIntMax, drivePidIntMax,
                             -driveOutMax, driveOutMax);
    }

    public void loop() {
```

assures that speed desired = input given to motor

```
        // ... other code

        // Use Pid to compute motor powers to achieve wheel velocity.
        left = leftDrive.update(wheelVelocities.getX(), leftSpeed,
                                deltaTime);
        right = rightDrive.update(wheelVelocities.getY(), rightSpeed,
                                  deltaTime);
        // Clamp motor powers.
        Vector2d motorPower = new Vector2d(left, right);
        clampPowers(motorPower);
        left = motorPower.getX();
        right = motorPower.getY();
    }
}
```

Congratulations, you now have the ability to control a motor's speed using Pid! You've removed the linearity assumption, making it more accurate. You've removed hard-coded constants which wouldn't adapt to changes in robot weight or air drag.

You may need to tune the values of $K_p$, $T_i$, and $T_d$ for it to perform as desired. Tuning Pid controllers is a skill- one which will take time and patience to learn.

# Javadoc

## com.github.pmtischler.base

### BlackBox

public class **BlackBox**
Robot BlackBox recording and playback. Writes hardware state as a time series and plays it back.

### BlackBox.Player

public static class **Player**
Reads hardware state from a timeseries stream and applies it.

#### Constructors

**Player**
public **Player** (InputStream *inputStream*, HardwareMap *hardware*)
Creates the player.

**Parameters**

- **inputStream** – The input stream to read from.
- **hardware** – The hardware to manipulate.

#### Methods

**playback**

public void **playback**(double *time*)
Playbacks the hardware up to the time.

**Parameters**

- **time** – The time to playback up to (seconds).

### BlackBox.Recorder

public static class **Recorder**
Writes hardware state as a timeseries stream.

#### Constructors

**Recorder**
public **Recorder**(HardwareMap *hardware*, OutputStream *outputStream*)
Creates the recorder.

**Parameters**

- **hardware** – The hardware to record.
- **outputStream** – The output stream to write.

#### Methods

**record**
public void **record**(String *deviceName*, double *time*)
Records the hardware at the time.

**Parameters**

- **deviceName** – The device to record.
- **time** – The time to record hardware at (seconds).

### SimpleCamera

public class **SimpleCamera** implements Camera.PreviewCallback, Camera.PictureCallback
Camera for taking pictures. Manages the Android camera lifecycle and returns OpenCV images.

#### Constructors

**SimpleCamera**
public **SimpleCamera**(Context *context*)
Initializes the phone's camera. Attempts to get the first camera, which should be the back camera.

#### Methods

**onPictureTaken**
public void **onPictureTaken**(byte[] *data*, Camera *camera*)

**onPreviewFrame**
public void **onPreviewFrame**(byte[] *data*, Camera *camera*)

**startCapture**
public boolean **startCapture**()
Starts the process for capturing an image. The image will be available from takeImage().

**Returns** Whether the capture was started.

**stop**
public void **stop**()
Releases the camera. Should be called when done with the camera to release it for future use.

**takeImage**
public Mat **takeImage**()
Gets the previously taken image.

**Returns** The taken image, or null if the image is not available yet.

### StateMachine

public class **StateMachine**
State machine manager. Simplifies the development of finite state machines.

#### Constructors

**StateMachine**
public **StateMachine**(*State initial*)
Creates the state machine with the initial state.

**Parameters**

- **initial** – The initial state.

#### Methods

**update**
public void **update**()
Performs an update on the state machine.

### StateMachine.State

public static interface **State**
A state in the state machine.

#### Methods

**start**
public void **start**()
Called when the state first becomes the active state.

**update**
public *State* **update**()
Called on each update.

**Returns** The next state to run.

### TimeseriesStream

public class **TimeseriesStream**
Timeseries streaming. Writes and reads timeseries streams.

### TimeseriesStream.DataPoint

public static class **DataPoint** implements java.io.Serializable
Data point in a time series. A variable has a Datapoint's value until the next instance in the stream.

#### Fields

**timestamp**
public final double **timestamp**

**value**
public final double **value**

**varname**
public final String **varname**

#### Constructors

**DataPoint**
public **DataPoint**(String *varname*, double *timestamp*, double *value*)
Creates a DataPoint.

**Parameters**

- **varname** – The name of the variable.
- **timestamp** – The time of the data point.
- **value** – The value of the variable.

### TimeseriesStream.Reader

public static class **Reader**
Timeseries reader.

#### Constructors

**Reader**
public **Reader**(InputStream *inputStream*)
Creates the Reader.

**Parameters**

- **inputStream** – The input stream to read from.

#### Methods

**read**
public *DataPoint* **read**()
Reads a DataPoint from the input stream.

**Returns** DataPoint if available, null otherwise.

**readUntil**
public List<*DataPoint*> **readUntil**(double *time*)
Reads all DataPoint up to specific time.

**Parameters**

- **time** – The timestamp to read up to (seconds, inclusive).

**Returns** The DataPoint read.

### TimeseriesStream.Writer

public static class **Writer**
Timeseries writer.

#### Constructors

**Writer**
public **Writer**(OutputStream *outputStream*)
Creates the Writer.

**Parameters**

- **outputStream** – The output stream to write to.

#### Methods

**write**
public void **write**(*DataPoint point*)
Writes a DataPoint to the output stream. Calls to this function must be done with non-decreasing timestamps.

### Vector2d

public class **Vector2d**
Vector in 2 dimensions.

#### Constructors

**Vector2d**
public **Vector2d**(double *x*, double *y*)
Creates a vector.

**Parameters**

- **x** – The x position.
- **y** – The y position.

#### Methods

**add**
public void **add**(*Vector2d other*)
Adds the given vector to this vector.

**Parameters**

- **other** – The other vector to add to this one.

**div**
public void **div**(double *v*)
Divide the vector by the given constant.

**Parameters**

- **v** – The constant to divide by.

**getX**
public double **getX**()

**Returns** Gets the X position.

**getY**
public double **getY**()

**Returns** Gets the Y position.

**mul**
public void **mul** (double *v*)
Multiply the vector by the given constant.

**Parameters**

- **v** – The constant to multiply by.

**sub**
public void **sub** (*Vector2d other*)
Subtracts the given vector from this vector.

**Parameters**

- **other** – The other vector to subtract from this one.

## com.github.pmtischler.control

### FeedbackLinearizer

public class **FeedbackLinearizer**
Feedback linearization and wheel mapping to convert robot velocity to wheels. Feedback linearization to convert robot velocity to forward and angular. v_x = desired velocity x in robot frame. v_y = desired velocity y in robot frame. $\epsilon$ = feedback linearization epsilon. v_f = forward velocity. $\omega$ = angular velocity. Equation: v_f = v_x. $\omega$ = v_y / $\epsilon$. Convert forward and angular velocity into wheel velocities: $\omega$ = angular velocity. $\omega$_{L,R} = left and right wheel angular velocities. R = wheel radius. L = baseline between wheels. v_f = forward velocity. Characteristic equations: $\omega$ = R ($\omega$_R - $\omega$_L) / L. v_f = R ($\omega$_R + $\omega$_L) / 2. Solving for $\omega$_R and $\omega$_L: $\omega$_L = V_f / R - $\omega$ * L / (2R). $\omega$_R = V_f / R + $\omega$ * L / (2R).

#### Constructors

**FeedbackLinearizer**
public **FeedbackLinearizer** (double *wheelRadius*, double *wheelBaseline*, double *feedbackEpsilon*)
Creates a FeedbackLinearizer with robot and control paramters.

**Parameters**

- **wheelRadius** – The wheel radius; R in the equations.
- **wheelBaseline** – The distance between the wheels; L in the equations.
- **feedbackEpsilon** – The feedback algo epsilon; $\epsilon$ in the equations.

#### Methods

**convertToWheelVelocities**
public *Vector2d* **convertToWheelVelocities** (double *forwardVelocity*, double *angularVelocity*)
Convert a forward and angular velocities to wheel velocities.

**Parameters**

- **forwardVelocity** – The forward velocity.
- **angularVelocity** – The angular velocity (radians/sec).

**Returns** The left wheel (x) and right wheel (y) angular velocities.

**feedbackLinearize**
public *Vector2d* **feedbackLinearize** (*Vector2d velocity*)
Convert a velocity in robot frame to forward and angular velocities.

**Parameters**

- **velocity** – The desired velocity of the robot in robot frame.

**Returns** The forward (x) and angular (y) velocities (radians/sec).

**getWheelVelocitiesForRobotVelocity**
public *Vector2d* **getWheelVelocitiesForRobotVelocity** (*Vector2d velocity*)
Gets wheel velocities to match a desired robot velocity.

**Parameters**

- **velocity** – The desired velocity of the robot in robot frame.

**Returns** The left wheel (x) and right wheel (y) angular velocities.

### Mecanum

public class **Mecanum**
Mecanum wheel drive calculations. Input controls: V_d = desired robot speed. theta_d = desired robot velocity angle. V_theta = desired robot rotational speed. Characteristic equations: V_{front,left} = V_d sin(theta_d + pi/4) + V_theta V_{front,right} = V_d cos(theta_d + pi/4) - V_theta V_{back,left} = V_d cos(theta_d + pi/4) + V_theta V_{back,right} = V_d sin(theta_d + pi/4) - V_theta

#### Methods

**motionToWheels**
public static *Wheels* **motionToWheels** (double *vD*, double *thetaD*, double *vTheta*)
Gets the wheel powers corresponding to desired motion.

**Parameters**

- **vD** – The desired robot speed. [-1, 1]
- **thetaD** – The angle at which the robot should move. [0, 2PI]
- **vTheta** – The desired rotation velocity. [-1, 1]

**Returns** The wheels with clamped powers. [-1, 1]

### Mecanum.Wheels

public static class **Wheels**
Mecanum wheels, used to get individual motor powers.

### Fields

**backLeft**
public final double **backLeft**

**backRight**
public final double **backRight**

**frontLeft**
public final double **frontLeft**

**frontRight**
public final double **frontRight**

### Constructors

**Wheels**
public **Wheels**(double *frontLeft*, double *frontRight*, double *backLeft*, double *backRight*)
Sets the wheels to the given values.

## Pid

public class **Pid**
PID Controller: kp * (e + (integral(e) / ti) + (td * derivative(e))). https://en.wikipedia.org/wiki/PID_controller#Ideal_versus_standard_PID_form

### Constructors

**Pid**
public **Pid**(double *kp*, double *ti*, double *td*, double *integralMin*, double *integralMax*)
Creates a PID Controller.

**Parameters**

- **kp** – Proportional factor to scale error to output.
- **ti** – The number of seconds to eliminate all past errors.
- **td** – The number of seconds to predict the error in the future.
- **integralMin** – The min of the running integral.
- **integralMax** – The max of the running integral.

### Methods

**clampValue**
public static double **clampValue**(double *value*, double *min*, double *max*)
Clamps a value to a given range.

**Parameters**

- **value** – The value to clamp.
- **min** – The min clamp.
- **max** – The max clamp.

**Returns** The clamped value.

**update**
public double **update**(double *desiredValue*, double *actualValue*, double *dt*)
Performs a PID update and returns the output control.

**Parameters**

- **desiredValue** – The desired state value (e.g. speed).
- **actualValue** – The actual state value (e.g. speed).
- **dt** – The amount of time (sec) elapsed since last update.

**Returns** The output which impacts state value (e.g. motor throttle).

## com.github.pmtischler.opmode

### DistanceStateMachine

public class **DistanceStateMachine** extends *MecanumDrive*
State machine to move up to distance.

#### Fields

**distanceSensor**
DistanceSensor **distanceSensor**

#### Methods

**init**
public void **init**()
Initializes the state machine.

**loop**
public void **loop**()
Runs the state machine.

### DistanceStateMachine.ForwardUntilDistance

public class **ForwardUntilDistance** implements *StateMachine*.*State*
Moves forward until a distance threshold is met.

#### Methods

**start**
public void **start**()

**update**
public *State* **update**()

### DistanceStateMachine.LeftForTime

public class **LeftForTime** implements *StateMachine*.*State*
Moves left for a specific amount of time.

#### Methods

**start**
public void **start**()

**update**
public *State* **update**()

### MecanumDrive

public class **MecanumDrive** extends *RobotHardware*
Mecanum Drive controls for Robot.

#### Methods

**loop**
public void **loop**()
Mecanum drive control program.

**setDrive**
public void **setDrive**(double *vD*, double *thetaD*, double *vTheta*)
Sets the drive chain power.

**Parameters**

- **vD** – The desired robot speed. [-1, 1]
- **thetaD** – The angle at which the robot should move. [0, 2PI]
- **vTheta** – The desired rotation velocity. [-1, 1]

### PlaybackAuto

public class **PlaybackAuto** extends OpMode

Playback autonomous mode. This mode playbacks the recorded values previously recorded by teleop.

#### Methods

**init**

public void **init**()

Creates the playback.

**loop**

public void **loop**()

Plays back the recorded hardware at the current time.

**stop**

public void **stop**()

Closes the file.

### RecordedTeleop

public class **RecordedTeleop** extends *MecanumDrive*

Recorded teleop mode. This mode records the hardware which can later be played back in autonomous. Select the manual control mode by changing the parent class.

#### Methods

**init**

public void **init**()

Extends teleop initialization to start a recorder.

**loop**

public void **loop**()

Extends teleop control to record hardware after loop.

**stop**

public void **stop**()

Closes the file to flush recorded data.

### RobotHardware

public abstract class **RobotHardware** extends OpMode

Hardware Abstraction Layer for Robot. Provides common variables and functions for the hardware.

#### Methods

**init**
public void **init**()
Initialize the hardware handles.

**setPower**
public void **setPower**(*MotorName* *motor*, double *power*)
Sets the power of the motor.

**Parameters**

- **motor** – The motor to modify.
- **power** – The power to set.

**stop**
public void **stop**()
End of match, stop all actuators.

### RobotHardware.MotorName

public enum **MotorName**

#### Enum Constants

**DRIVE_BACK_LEFT**
public static final *RobotHardware.MotorName* **DRIVE_BACK_LEFT**

**DRIVE_BACK_RIGHT**
public static final *RobotHardware.MotorName* **DRIVE_BACK_RIGHT**

**DRIVE_FRONT_LEFT**
public static final *RobotHardware.MotorName* **DRIVE_FRONT_LEFT**

**DRIVE_FRONT_RIGHT**
public static final *RobotHardware.MotorName* **DRIVE_FRONT_RIGHT**

### TankDrive

public class **TankDrive** extends *RobotHardware*
Tank Drive controls for Robot.

### Methods

**loop**
public void **loop**()
Tank drive control program.

**setDrive**
public void **setDrive**(double *left*, double *right*)
Sets the drive chain power.

**Parameters**

- **left** – The power for the left two motors.
- **right** – The power for the right two motors.

CHAPTER 2

# Indices & Tables

- genindex
- modindex
- search

## S



## T



## U



## V



## W