Drive code

INTRODUCTION

In order for the robot to move, we have to be able to give it a few different instructions:

- Drive forwards
- Drive backwards
- Speed up

- Slow down
- Turn left
- Turn right
- Stop

Depending on what other mechanisms are on the robot (e.g., a pickup mechanism, a shooter, an elevator), we also want to control those but given that those change year to year, we're going to talk mostly about drive code here.

We can do all of this directly in code (and there are times where we will do this), but most of the time, we want a person to drive it directly.

There are any number of different ways you could do this, but most of them end up being pretty silly. You could make a button for everything. A speed up button. A slow down button. A turn left button. And so on. But that isn't great. The best drive code matches what people expect and matches the real world pretty closely. It would be nice if you don't have to think about it as much. If the control moves forward to move the robot forward or turns to make the robot turn.

So generally, you want to use joysticks (or the joystick portion of a XBox controller) instead. But saying that doesn't give you all that you need to know. In order to program the input. There are several different ways to program joysticks to make a robot drive. We're going to look at two of them.

JOYSTICKS

First we're going to look at what information you can get from joysticks before we look at how you can use it.

Constructor

First, let's take a look a the constructor. Joysticks are plugged in via USB port. So in order to create a Joystick object, you just need to tell it which USB port the joystick is plugged into. The library does the rest. The code looks like this:

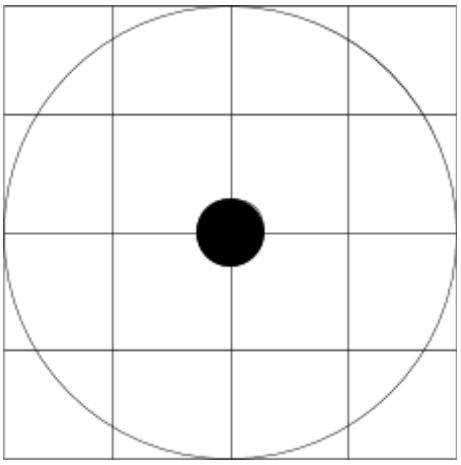
```
Joystick left = new Joystick(1);
```

Reading input

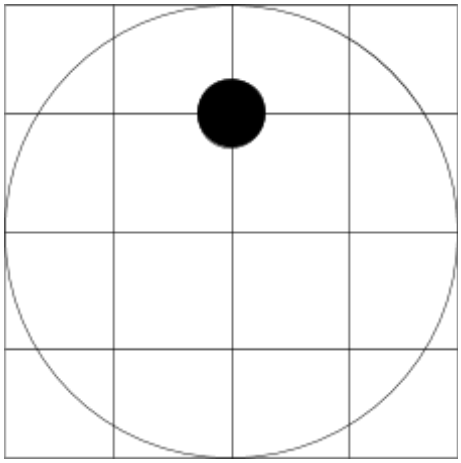
Square Grid

When you pull the raw values from the joystick, the values you can get are the direct values on the x and

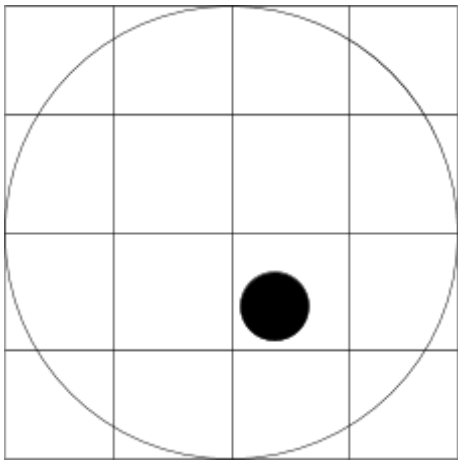
y axes, with 0 being in the middle and the far ends being at -1 and 1. So you can imagine a grid going over the whole joystick area. If we look down from the top it looks something like this:



This would be the joystick at (0, 0). If we were to push the joystick forward, the joystick would get start to get a positive y value. So this position would be (0, 0.5)

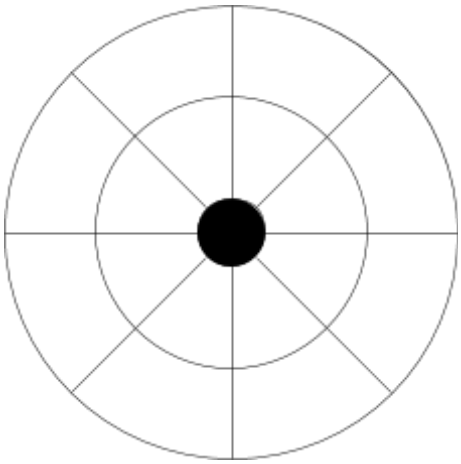


On the x-axis, left is negative, and right is positive. So if you were to go to the bottom right, so this would be (0.1, -0.4):



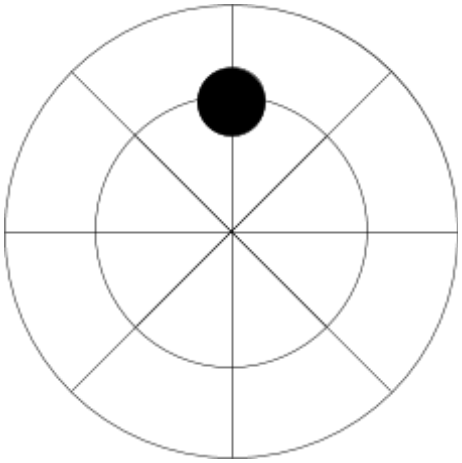
Circular Grid

The other way to get the values from a joystick is to use a circular grid. This grid doesn't use x and y axes, but tracks the angle and the distance from the center to that point.



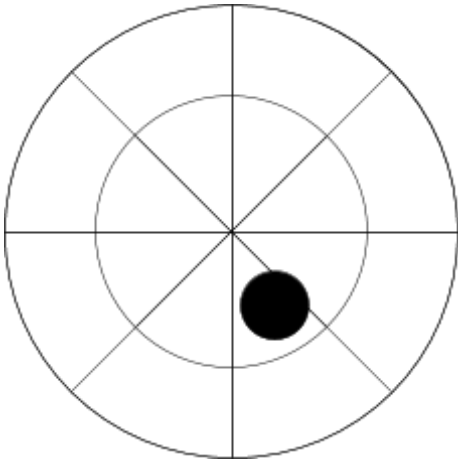
Again this is the center so it is at 0° with a distance (called magnitude) of 0. You can access these via `joystick.getDirectionDegrees()` and `joystick.getMagnitude()`

If we bring back our other examples



For magnitude, it works very similarly to the grid coordinates, where 0 is in the center and 1 is at the edge so this would have coordinates of 0° with a magnitude of 0.5.

The angle starts at 0° up top and goes around the circle clockwise. So this position would have coordinates 166° and magnitude of 0.41.



Buttons

There are also many different buttons that are on joysticks that you may need. Often these are used to activate specific abilities of the robot like shoot a ball or things outside of driving, but since we're talking joysticks we'll discuss this here.

Button functions

For every button that exists on these devices, there are a few different functions.

getXButton()

The first function takes the form `getX()`. So for the trigger button, it is `getTrigger()`, for the button on top it is `getTop()`, for any other button on a standard joystick it is `getRawButton(int buttonNumber)`, where each button is labelled with its number on the joystick.

This function returns a boolean of whether or not the button is currently being pressed.

XButtonPressed/XButtonReleased

Every button also has functions that can be called which indicate whether the button has been pressed or released since last check. So `getTriggerPressed()` indicates whether someone has pressed indicates whether or not someone has the trigger since the last check, whether or not is being held down, same with `getTriggerReleased()`.

Either set can be used. If you are using button pressed or button released, know when your last check was. Know that loop cannot be done effectively if using a button pressed/button released method, because in your second iteration it will no longer be true. (Even if someone is button mashing, they can't button mash faster than code can run.) So be aware of your choices and how they work.

XBOX CONTROLLER

Some of this carries over to the XBox controller as well, as it has buttons and joysticks as a part of it.

Constructor

It also plugs into a USB port, so the constructor ends up looking much the same, give the port that it is plugged into:

```
XboxController xbox = new XboxController(2);
```

Reading input

Square Grid

The Xbox controllers can only give you rectangular grid coordinates, but getting that information is a little different from getting it on a joystick. There are getX() and getY() functions, but you have to specify

which joystick you are talking about with an enum. So to get the Y value from the left joystick on the Xbox controller it would be

```
xbox.getY(GenericHID.Hand.kLeft);
```

For the right it would be

```
xbox.getY(GenericHID.Hand.kRight);
```

Circular Grid

You cannot directly get circular coordinates from the Xbox controller. If you want to have circular coordinates, you can do some math to give you that.

magnitude = $\sqrt{x^2 + y^2}$

angle = $tan^{-1}\left(\frac{y}{x}\right)$

Buttons

Buttons work the same way as on joysticks, all of them have the same three functions, and with the same naming scheme. So you can call `getStartButton()` or `getYButtonPressed()`, just like with the joysticks.

METHODS OF DRIVE CODE

Tank Drive

For tank drive, imagine you had tank treads on your robot. You now only have two things to control, the right side tank treads and the left side tank treads. One joystick controls each side.

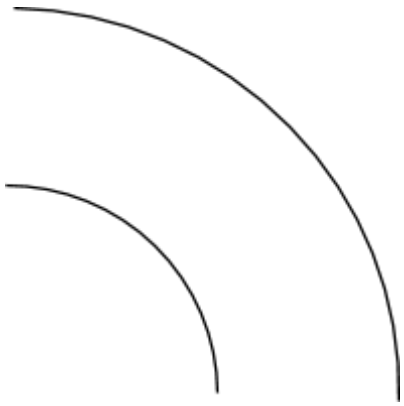
Let's run through the operations we listed earlier and see how these work in tank drive:

- Drive forwards: Move both joysticks forwards to the same level
- Drive backwards: Move both joysticks backwards to the same level
- Speed up: Move the joysticks further from the middle
- Slow down: Move the joysticks closer to the middle. Most of the time the robot will slow down on its own due to friction, so you can let go in most cases too.

- Turn left: Move the right joystick further forwards than the left joystick. If you want to turn in place, move the right joystick full forward and the left joystick full backward.
- Turn right: Move the left joystick further forward than the right joystick. If you want to turn in place, move the left joystick full forward and the right joystick full backward.
- Stop: Let go of the joysticks, or just return them to zero

Most of these should make intuitive sense, but if the turns are not obvious which goes which way or how one side going backwards helps you turn, try doing this by walking.

If you turn to the left, your right foot has more ground to cover. So if you go a little bit faster with your right foot, you'll end up turning left. Like how this path that takes a turn towards the left has a longer right side than left side.



If you want to turn in place, pick up your right foot and try turning without moving your left foot. Eventually you can't move anymore, and you have to step your left foot backwards in order to continue moving.

Making Tank Drive work in code

The best way to make Tank Drive work is to use rectangular coordinates. When you use getY() for each joystick, it gives you the values you need to map to the output of the motors.

Arcade drive

Arcade drive is based on the idea is that if you move the joystick up and down you go forwards and backwards, but if you move the joystick side to side that will be a turn. This can be done with two separate joysticks with one controlling forward/reverse motion and the other controlling turns, or it can be done with one joystick controlling both. It is called arcade drive because it looks a little more like how an arcade video game would use a joystick.

So if we return to our things that drive code needs to do, let's see how we would do them with an arcade drive robot. For this I will refer to having two different joysticks, but if you are using one joystick, it would all be the same

- Drive forwards: Move the forward/reverse forward
- Drive backwards: Move the forward/reverse backward
- Speed up: Move forward/reverse joystick further from the middle
- Slow down: Move forward/reverse joystick closer to the middle. Most of the time the robot will slow down on its own due to friction, so you can let go in most cases too.
- Turn left: Move the turn joystick left.
- Turn right: Move the turn joystick right.
- Stop: Let go of the joysticks, or just return them to zero

Making Arcade Drive work in code

Let's start by looking at the forward/backwards portion of this code. This ends up looking a lot like Tank Drive, but you control all the motors from one joystick. Take the `getY()` value from that joystick and assign it to all the motors.

Now if we think about just turning for a second, the further on the left we get on the x-axis, the more power we want to give a left turn, so we want to give that much power to reversing the left motors and pushing the right motors forward. So if the joystick is pushed half way out to the left, we have a value of -0.5 from `getX()`. So the left motors can get a value of x and the right motors get -x. Same thing happens for if we are at a value of 0.75, this means a ¾ power right hand turn. So left motors get a value of x and right motors get a value of -x.

Combining them ends up being super straight forward. Just add the forward/backward component together (make sure you don't go over the max for the motor though).

So the left motors get y + x and the right motors get y - x.

MORE THINGS TO CONSIDER

Don’t go max speed

We’ve been talking about a value at the edge of the joystick being 1. It is really easy to make that be mapped to full power to your motors, but maybe not the best idea. Consider whether you want full power or if you want 80% or 90% to be the most power you can give to your robot.

Dead zones

We’ve considered how 1 can be a problem, but so can 0. The only way the robot is not trying to move is if the joystick(s) are at exactly 0. If they register at 0.01, the robot is trying to move. And since this is hardware, it is probably not going to go exactly back to 0, especially when your driver is likely to keep their hands/thumbs on the controls the whole time. So you probably want values really close to 0 to be treated the same as 0. This is generally called a dead zone or dead band on your joystick.

Reversed motors

When we’ve been talking about mapping joystick values to the motor values, we’ve been talking about forward with respect to the robot. What you are really coding for is forward with respect to the motors. So any particular motor when you give it full power “forward” will go one way and “backward” it will go the other way. Depending on how it is mounted, this may not be the same as the front and back of the robot. If for whatever reason a particular motor is facing the opposite direction that you want it to be facing, just multiply the values you are giving it by -1.

Fighting motors

If there are multiple motors that need to work together for a specific purpose (as is common for drive, but also happens for other mechanisms), the two motors may be facing different directions. If you aren’t careful you can end up with the two motors fighting each other with the two motors on one side going opposite directions, which doesn’t end up with the robot going anywhere and can just leave you with burned out motors.

PID

These values we are giving are target speeds. Getting to that speed is not a sudden thing. Think about how a car is often said how fast it goes from 0-60, it usually takes a little bit of time. The process we use is PID. I want to call it out here, but the full details will be explained in its own lesson.