Making a One Degree Motor Arm with Potentiometer and PID

Version 1.0 June 8, 2018

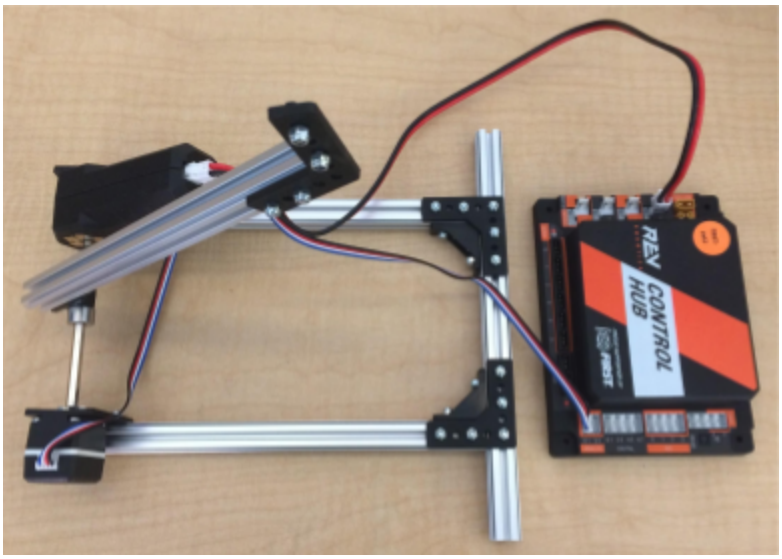
Introduction

This guide will demonstrate the coding steps to make a motor move to desired positions using a potentiometer. You could also use motor encoders, but for most mechanisms besides wheels, potentiometers are usually better. Whenever you start an opMode, encoder values are reset to 0. If you're trying to use encoders for something like a robot arm, then you have to reset the arm to a specific position or else you'll have a different 0 position.

Table of Contents

Hardware

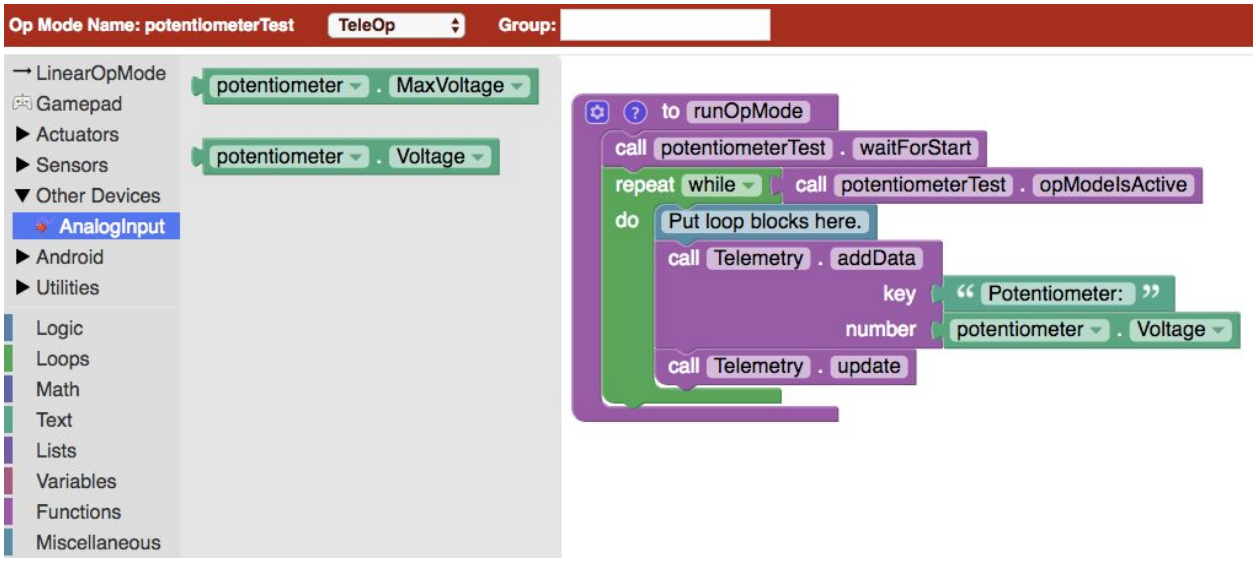
On the right is a picture of the hardware I used to test with. Basically it involves a Core Hex Motor attached on a hex axle. On that axle is a hex pillow bracket and a potentiometer on the other end. The motor is wired into the motor section, and the potentiometer is wired into analog input. When putting your potentiometer and axle together, keep in mind the physical limits (270°) of your potentiometer.



Basic Potentiometer Usage

Potentiometers give a reading of their rotation amount as a double with a range of 0.0-3.3. You can find the potentiometer's reading based on the physical orientation of the extrusion (or whatever else you have attached to the axle) by printing the potentiometer's voltage to telemetry. In Blocks, the potentiometer voltage block can be found under **Other Devices** and **AnalogInput**. In the robot configuration, I named my potentiometer "potentiometer" for

simplicity.



If you're using Java like I will for a majority of this guide, you can access your potentiometer's position using **potentiometer.getVoltage()**.
While the above opMode is active, you can rotate your axle by hand until it reaches the desired position. Be sure to write down the number on the telemetry display!

Basic Motor Control

The simplest way to control your motor using a potentiometer is by setting a flat power. Here's a sample code of doing that.

```
if(potentiometer.getVoltage() < desiredPosition){
      arm.setPower(0.5);
}
else if(potentiometer.getVoltage() > desiredPosition){
      arm.setPower(-0.5);
}
```

On paper, this method looks like it works. If the axle isn't at desiredPosition, the motor will rotate the axle until it is. However, most of the time in testing this method fails miserably. The motor will overshoot desiredPosition, so it will switch directions. Then it will probably skip over desiredPosition and switch directions again. And again. And again. Forever. Even if you reduce the power, you'll run into the same problem since the potentiometer gives precise readings.

You could also try decreasing the potentiometer's precision by giving your code a range.

```
if(potentiometer.getVoltage() < desiredPosition - 0.1){
      arm.setPower(0.2);
}
else if(potentiometer.getVoltage() > desiredPosition + 0.1){
```

```
        arm.setPower(-0.2);
    }
    else{ //runs when axle is in a range of ± 0.1 of desiredPosition
        arm.setPower(0);
    }
```

This code will let your motor stop once it's close enough to your desiredPosition, but you lose speed and precision by decreasing the power and increasing the range of stopping. To help fix the speed problem you could have a higher power for your arm motor with a large range, but ultimately this is not the best method.

PID Controlled Movement (Java)

A PID (proportional-integral-derivative) controller factors in three different components of movement. There are much better guides than this one if you want to fully understand PID, but here's a simplified explanation. The proportional part of a PID looks at how far off from desiredPosition the axle is. The integral part looks to see if the error is actually changing, and if not, it will try to increase the motor's power. The derivative part looks at how quickly your error is changing. Each of these three components is multiplied by a different constant (k), and they are all added together to create your output, the number you set the arm's power to.

I'll demonstrate with further detail a PID using the function below.

```
//These variables are declared outside of the function so they won't be overwritten.
double errorPrior = 0;
double integralPrior = 0;
double lastLoopTime = 0;

public void pidControlArm(double desiredPosition){
   if(gamepad1.a){
       double changeInTime = (runtime.milliseconds() - lastLoopTime)/1000;
       double error = 0;
       double integral = 0;
       double derivative = 0;
       double output = 0;
       double kp = 1;
       double ki = 3.4;
       double kd = .04;

       error = -(desiredPosition - potentiometer.getVoltage());
       if(Math.abs(error) > 0.1){
           integralPrior = 0;
       }
```

```
        integral = integralPrior + error*changeInTime;
        derivative = (error-errorPrior)/changeInTime;
        output = kp*error + ki*integral + kd*derivative;
        if(Math.abs(output) > powerCap){
            output = Math.signum(output)*powerCap;
        }

        errorPrior = error;
        integralPrior = integral;
        lastLoopTime = runtime.milliseconds();

        arm.setPower(output);
    }
    else{
        lastLoopTime = runtime.milliseconds();
    }
}
```

errorPrior, integralPrior, and lastLoopTime are declared outside of the function because they need to be saved through the teleop loop. Every loop is one tick, and this function runs once every tick. We need to save these variables to compare the error, integral, and time of the last tick with the current tick.

```
public void pidControlArm(double desiredPosition)
```

is used to declare our function. Now in our main teleop loop we can write:

```
pidControlArm(0.5);
```

Which calls our function and tells it we want our arm to go to 0.5.

After that, we check to see if A is pressed. This means the PID controller will only run when A is pressed, and the motor will only move is A is pressed.

Next is the declaration of 8 variables. I'll get to each of their specific uses later. The last three (kp, ki, kd) are our constant multipliers. Changing these three lets us tune our PID controller to put more emphasis on error, integral, or the derivative.

The next four lines are the actual calculations for our PID controller. First is an error calculation to find the difference between the potentiometer value we want to reach, and the potentiometer value we currently have.

```
error = -(desiredPosition - potentiometer.getVoltage());
```

In the hardware setup you'll see the potentiometer and motor are facing opposite directions. The potentiometer difference is set negative to account for this.

```
integral = integralPrior + error*changeInTime;
```

At the end of the function integralPrior is set to the current integral to create the effect of adding up. This line of code adds up all the error*changeInTime since you started running the PID function. If the error stays large for a long period of time, the integral will add up to a large number. Once this happens, the arm moves with a lot of power which hopefully will cause the error to decrease.

```
derivative = (error-errorPrior)/changeInTime;
```

The derivative is $\frac{change\ in\ error}{change\ in\ time}$ which looks a lot like speed. The derivative component decreases the speed of the arm as it approaches the desired position.

```
output = kp*error + ki*integral + kd*derivative;
```

Finally, all three components are added together. Each one of them is multiplied by a constant in order to place emphasis on certain components. Finding k values can involve a lot of trial and error. For example, if you define **kp** to be much higher than **ki** and **kd**, the motor's power will be mostly based on how far away it is from the desired position. This causes the same problem we had in the Basic Motor Control section where the arm would oscillate back and forth, skipping over the desiredPosition. You may have to modify the constants you use based on how your robot is set up. If you need any help, contact me!

The output is set to a power cap, so the arm does not move too quickly. Math.signum just checks if a number is positive, negative, or 0, and returns 1, -1, or 0. If a number is greater in magnitude (see the absolute value) than the power cap, the output is set to the power cap with the sign of the power maintained the same.

errorPrior, integralPrior, and lastLoopTime have to be set to the correct values for this tick, so in the next tick they will be considered the previous error, integral, and time.

Finally the motor's power is set to the output of our PID calculation.

In case we weren't pressing one of the buttons, the lastLoopTime is set to the current time anyways, so changeInTime is based on tick time difference and not press time difference.

PID for Multiple Buttons (Java)

I later realized the PID function doesn't work if you want to use multiple positions and multiple buttons (due to the way I declared errorPrior, integralPrior, and lastLoopTime). Here's a version of the code which turns those into arrays. The first time pidControlArm() is called in the teleop

loop, errorPrior[0], integralPrior[0], and lastLoopTime[0] are used for saving variables. The second time uses [1], third time [2], and so on.
In addition to that, this version of the PID controlled arm stops sending power to the motor after 50 ticks of it being within ±0.02 of the desired value.
[Github link to this teleop](#).

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

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.util.ElapsedTime;

@TeleOp(name="singleDegreeArmHoldtoMoveJava", group="Pushbot")
public class singleDegreeArmHoldtoMoveJava extends LinearOpMode {
    private ElapsedTime runtime = new ElapsedTime();

    double errorPrior[] = new double[4]; //Uses 4 because that's how many different buttons use
pidControlArm();
    double integralPrior[] = new double[4];
    double lastLoopTime[] = new double[4];
    int correctCount[] = new int[4];

    int pidCount = 0;
    int notPressedCount = 0;

    private AnalogInput potentiometer = null;
    private DcMotor arm = null;
    @Override
    public void runOpMode() {
        potentiometer = hardwareMap.analogInput.get("potentiometer");
        arm = hardwareMap.dcMotor.get("arm");

        while(!opModeIsActive()){}

        while(opModeIsActive()){
            pidControlArm(0.3, gamepad1.a, 1, pidCount);
            pidControlArm(0.8, gamepad1.b, 0.5, pidCount);
            pidControlArm(0.6, gamepad1.y, 0.2, pidCount);
            pidControlArm(1.0, gamepad1.x, 0.4, pidCount);
            if(notPressedCount == pidCount){
                arm.setPower(0);
            }
```

```
            pidCount = 0;
            notPressedCount = 0;
            telemetry.addData(">", potentiometer.getVoltage());
            telemetry.update();
        }
    }
    public void pidControlArm(double desiredPosition, boolean control, double powerCap, int
arrayPos){
        pidCount += 1;
        if(control){
            double changeInTime = (runtime.seconds() - lastLoopTime[arrayPos]);
            double error = 0;
            double integral = 0;
            double derivative = 0;
            double output = 0;
            double kp = 1;
            double ki = 3.4;
            double kd = .04;
            error = -(desiredPosition - potentiometer.getVoltage());
            if(Math.abs(error) > 0.1){
                integralPrior[arrayPos] = 0;
            }
            integral = integralPrior[arrayPos] + error*changeInTime;
            derivative = (error-errorPrior[arrayPos])/changeInTime;
            output = kp*error + ki*integral + kd*derivative;
            if(Math.abs(output) > powerCap){
                output = Math.signum(output)*powerCap;
            }
            telemetry.addData("p", kp*error);
            telemetry.addData("i", ki*integral);
            telemetry.addData("d", kd*derivative);
            errorPrior[arrayPos] = error;
            integralPrior[arrayPos] = integral;
            lastLoopTime[arrayPos] = runtime.seconds();
            if(Math.abs(error) <= 0.02){
                correctCount[arrayPos] += 1;
            }
            else{
                correctCount[arrayPos] = 0;
            }
            if(correctCount[arrayPos] <= 50){
                arm.setPower(output);
```

```
            }
            else{
                arm.setPower(0);
            }
        }
        else{
            notPressedCount += 1;
            lastLoopTime[arrayPos] = runtime.seconds();
        }
    }
}
```

PID (Blocks)

I'm not as experienced with coding in blocks, but I made a copy of the java code you see above. The download link is here.

Contact Information

Feel free to ask me anything about this guide or any other questions you have about robotics! Send emails to engineering@yale.edu or you can message me on Discord at nahn20#5234. I use Github to upload the example opmodes you'll see in these guides.