First Robotics

Closed Loop Motor Control with the Talon SRX and Quadrature Encoders

Team 5109

11/4/2018

# Chapter 1: Introduction

Our motor controller of choice is the Talon SRX from Cross the Road Electronics™ (CTRE). Until now we’ve been using the Talon as a simple voltage controller- giving our motor -100% to 100% of the battery voltage in order to induce a speed to power to the motor. That’s great for Teleop, and your human driver becomes the feedback loop.

For Autonomous, we really need two functions:

1. Drive straight forward to an exact distance and stop
2. Turn to a specific angle

So far, we’ve been using 256 count Grayhill encoders, wired directly back to the Rio to keep track of how far we’ve travelled. Because we use iterativeRobot, this means we have a 20ms to 30ms time in between cycles, depending on how delayed our driverstation updates are. Here’s our first problem: Driverstation updates may not be exactly 30ms. A very temporary network backup could cause a delay of 10x that easily.

Our second problem is that we don’t make a correction until it’s too late. We’re basically watching for the encoders from each side to get out of sync; one has already sped up or slowed down, and we’re no longer straight! At best we end up oscillating between too far in each direction on our way to an unknown goal.

The purpose of this document is to illustrate another way. We’ll visit the basics of PID control, and explore ways to offload this functionality onto something that doesn’t depend on an unstable timebase. Basically, we’re going to find out how to make the machine work for us.

## How to use this document:

This kind of thing is best learned by example; when you have a problem you need to solve, and you’ve got a deadline. If you’ve got both of those, you’re set. Your most likely scenarios are:

1. You’re writing an autonomous, and need to drive straight quickly and accurately
2. You’re building some sort of elevator, gripper, or control arm that needs to go to an exact position quickly, consistently and accurately.

If you fall into ‘A’ then read chapters 2 and 3, then skip on out to Chapter 5, because driving straight means that you’re accurately controlling the velocity of your differential drive wheels. You won’t be tuning for position, but you DO need the terminology and understanding to proceed. If you fall into ‘B’ then proceed in order. Stop when you’ve had enough.

# Chapter 2: PID loops for position control

## P term:

PID is an acronym, it stands for ‘Proportional, Integral, differential’. Let’s run through these one at a time.

Let’s say you’re at position 10, and you WANT to get to position 500. That means we have 490 to go until we get there. 490 what?! What are we talking about? It doesn’t matter. Could be inches, could be encoder ticks. As long as your unit never changes, use whatever is convenient. For now, I’m going to use encoder ticks.

So getting back on point, we’re at 10. Our goal is 500. That makes our Error = (goal - position) = 490.

BEFORE we really get going on each term, I’d like to share a preview though: PID loops need to run on a stable timebase. Each time you recalculate Error, you need to know it’s based on the same amount of time since you LAST calculated error, and the same time will pass before you calculate error again. When controlling a bot with Java, we can do this with TimedRobot, which will always run every 20ms.

NOW we’re getting somewhere! I need to go forward 490 to get where I’m going. Proportional control (just the P in PID) can be realized as follows:

Control = Kp \* Error (remember Error = goal - position).

So what’s Kp? Turns out that’s up to you. Kp is a ‘Constant’ or a real number that we set to be our proportional term. It’s part of what we tune to get accurate control. Right now our Talon SRX can be set from -1023 to +1023, which is full reverse to full forward (0 is stop). If we set our Kp to, say, 1, our current motor power will be 490 (see what I did there? Motor power = Kp \* Error.) This might be fine if there’s 1000 counts per revolution. It may be way too little if there’s only 5 counts per revolution, and your motor will slow to a stop long before you’ve reached your goal.

You MAY be able to reach your intended position using ONLY the P term, and only perform proportional control. This is often true in very high ratio gearboxes, that you can’t backdrive. Hint: it probably won’t be all you need for FRC.

So, let’s talk about our next term: The Integral term.

## I term:

The I, or Integral term is the first term that “remembers” some of our error since the last time we ran an error. Let’s continue on our previous example: I’m now at 495 counts, and I want to get to 500. In this case, my Error = (500-495) = 5. OK, I’m only 5 away from my goal. Is that good enough? Well, it had better be, because with Proportional control (remember above we set our P term to 1) our motor power is only going to be 5. 5/1023 isn’t going to move the motor alone let alone a gearbox and a 120 pound robot. So what do we do? This is where the I term really shines:

Integral term = Ki \* (Error) + Integral term from last time through the loop

Here’s why it’s awesome: If you’re close and stalled, using an integral term, you can slowly crank up the power you need in order to get that motor moving towards its final goal.

Here’s why it sucks: The integral term can get out of hand VERY quickly. The example above was MADE for an I term, but horrible if you’re at 5 and want to reach 20000. The proportional term alone can get you to full power, so the I term just builds and builds each time with nowhere to go. How do you handle that situation? The answer is cheat. Yep, this is your code, and you can do what you want with it. Here’s some ways to deal with a runaway I term:

1. Make the I term = 0 every time your error changes sign (from + to - or vice versa). This indicates that you’ve already crossed your goal, and the Iterm is driving you further away.
2. Don’t even use an Iterm unless you’re close enough that the Pterm won’t get you there. This way, your Iterm can’t build unless it’s necessary.

Turns out the Talon SRX has built in methods for BOTH of the above tricks.

## D Term

The D term, or differential term, is how we apply the brakes. Up until now we’ve talked about how to GET to where we’re going, but our next challenge is to slow down in time to stop where we want. Your D term is calculated as follows:

Differential Term = Kd \* (Error Last time - Error now)

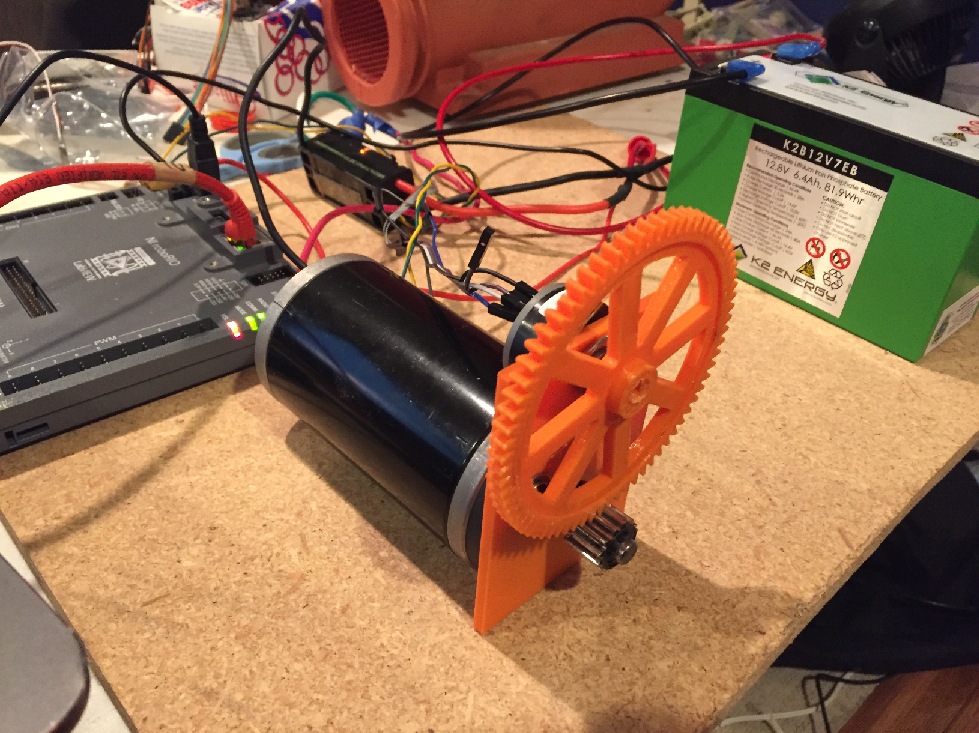
Looking at that, it’s a measure of how much we’ve IMPROVED on our position, and it subtracts power as we reach our goal. This way the entire PID is determined as:

PID=P + I - D

And THAT is where we set our motor power. Well, great. So how do you tune it?

# Chapter 3: Physical Setup

A position PID would be used for something like a servo, a control arm, an elevator; basically anything you want to be able to actively control the exact position of, especially where the load may be changing. In my case, I’ve got a motor, a 7:1 gearbox, and a 256 count greyhill encoder attached:



*Picture of the 7:1 CIM motor to encoder setup*

Something you might notice about my setup- the encoder is connected directly to the Talon! This is what enables the Talon to run in ‘Closed Loop’ mode. Here’s an interface cable to connect to the Talon SRX:

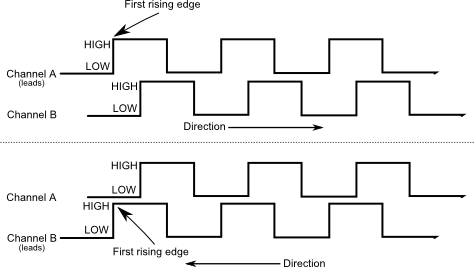
<http://www.ctr-electronics.com/talon-srx-data-cable-4-pack.html>

And while we’re at it, here’s a breakout board to connect to a quadrature encoder:

<http://www.ctr-electronics.com/talon-srx-encoder-breakout-board.html>

Notice that the breakout supplies power as well.

Here’s something interesting to note: Although we’re using a 256 count quadrature encoder, the Talon counts 1024 ticks per revolution. How? Well, usually when we’re using an encoder, we just pick one phase rising edge:



We can use Either A or B, but we watch for rising edges, and check the other phase to determine whether we’re going forwards or backwards. The Talon isn’t doing so many other things, so it watches each change (rising and falling) of each phase, and makes a count at each one. This gives us a 4X resolution, or 1024 counts per rev instead of just 256.

At this point we have a 1024 count encoder on the output of a 7:1 gearbox. My base unit is going to be encoder counts, so when we talk about position, goal, and error we’re all using the same base unit- encoder counts.

# Chapter 4: Tuning for position

Now for the fun part. How do I TUNE a PID loop? I promise, it’s not that hard. Let’s start with the same layout we had previously: CIM motor connected to an encoder through a 7:1 geardown. The encoder is connected directly to the Talon. Here’s some simple code to set this up:

package org.usfirst.frc.team5109.robot;

import com.ctre.phoenix.motorcontrol.ControlMode;

import com.ctre.phoenix.motorcontrol.FeedbackDevice;

import com.ctre.phoenix.motorcontrol.can.TalonSRX;

import edu.wpi.first.wpilibj.Joystick;

import edu.wpi.first.wpilibj.TimedRobot;

import edu.wpi.first.wpilibj.smartdashboard.SendableChooser;

import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;

public class Robot extends TimedRobot {

//Let's define our constants here. Why? Because they're easy to find when tuning.

//They are implemented below

private static final double Kp=0.0; // <- when i say change Kp, it’s this!

private static final double Ki=0.0;

private static final double Kf=0.0; //no feed-forward on position control

private static final double Kd=0;

private static final int IZone=100; //IZone, this is explained below

// I only have one joystick. here it is:

Joystick joystick = new Joystick(0);

//I also only have one Talon connected:

TalonSRX \_motor = new TalonSRX(1);

/\*\*

\* This function is run when the robot is first started up and should be

\* used for any initialization code.

\*/

@Override

Notice the GREEN text above. These are the constants I’m going to set for closed loop control. At the moment, all of these are set to 0 because we haven’t started yet. We’ll start tuning in a minute, but let’s start some basic configuration first.

//Configuration stuff.

\_motor.configSelectedFeedbackSensor(FeedbackDevice.QuadEncoder);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Ramps. The argument is how many seconds it takes to go from 0 to 100%

\* during voltage control. There is a ramp for both open loop and closed

\* (controlled) loop

\*/

\_motor.configOpenloopRamp(0.2);

\_motor.configClosedloopRamp(0.2);

// Talon setup. Setting up + and - output for max and nominal.

//These should never really change

\_motor.configNominalOutputForward(0, 0);

\_motor.configNominalOutputReverse(0, 0);

\_motor.configPeakOutputForward(1,0);

\_motor.configPeakOutputReverse(-1, 0);

//set this if your encoder is hooked up backwards (A<--->B)

//mine is backwards, so I've set it to true

\_motor.setSensorPhase(true);

Above I have a line highlighted in yellow. This is because my encoder was connected backwards. This means that either my encoder is turning backwards (it is) OR my A and B signal wires are connected to the wrong spot. No big deal, just setSensorPhase(true). So here’s the catch: How do you know if your wires are backwards? Easy. Write a quick teleop program to drive your motor forward (or just read a joystick value and push forward), and read the counts back with this:

SmartDashboard.putNumber("Current Pos:", \_motor.getSelectedSensorPosition(0));

The above statement just writes the current encoder position (in counts) on the SmartDashboard. If you’re giving the motor + power (greater than 0) that number should climb. If it’s the opposite, setSensorPhase to the opposite of what it is now.

Great. Let’s write some more test code to help us tune a PID. just add this to what you’ve already got above! You’ll notice I haven’t commented anything we’re not using here.

//implementation of tuning constants. First argument is PID slot. We're only

//using slot 0. If we had multiple tuning constants for the same controller

// for different purposes, we can also have up to 3 slots pre-programmed.

// The last number is a timeout- we can set an error vector if for some reason

// we can’t program our talon and it times out. I’m not doing this here.

\_motor.config\_kF(0, Kf, 0); //No feed-forward on position control

\_motor.config\_kP(0, Kp, 0); //P factor!

\_motor.config\_kI(0, Ki, 0);

\_motor.config\_kD(0, Kd, 0); //note- these are defined above so they're easy to find and change.

}

@Override

public void autonomousInit() {

}

@Override

public void autonomousPeriodic() {

}

}

/\*\*

\* This function is called periodically during operator control.

\*/

@Override

public void teleopPeriodic() {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Here's what I'm doing during teleop: If you're NOT holding the trigger,

\* I operate like a normal motor- just piping Y axis of my joystick over

\* to the motor. Full stick forward= full motor power.

\* I'm also reading the Z axis on the joystick. This gives me a number from

\* -1 to 1. When you hit the trigger, I set the target of the Talon

\* to Z\*4096\*10 which sets a position target of anywhere from -40 to 40 full

\* Turns of the encoder wheel. I have the PID set to get me there as fast

\* as possible, and the talon takes care of it. You'll also notice there

\* are some SmartDashboard numbers thrown up there to check accuracy.

\*/

double targetPosition=joystick.getZ()\*4096\*10;

if (joystick.getTrigger())

{

\_motor.set(ControlMode.Position, targetPosition);

if (++\_loops >= 10) {

\_loops = 0;

SmartDashboard.putNumber("Trigger hit", targetPosition);

SmartDashboard.putNumber("Z value:",joystick.getZ());

}

}

//reguluar teleop. Just give the motor power wherever you have the joystick set

else if (!joystick.getRawButton(1)) {

double axis=joystick.getY();

\_motor.set(ControlMode.PercentOutput, axis);

if (++\_loops >= 10) {

\_loops = 0;

SmartDashboard.putNumber("Trigger hit", targetPosition);

}

}

SmartDashboard.putNumber("Current Pos:", \_motor.getSelectedSensorPosition(0));

}

/\*\*

\* This function is called periodically during test mode.

\*/

@Override

public void testPeriodic() {

}

}

Now we’ve got a workable system. Let’s start by tuning P, the proportional term of our PID loop.

## Tuning P

Tuning P is really easy. Seriously. Start by making Kp=0.5. Set a target with the Z axis and pull the switch. Chances are pretty good it’s not going to get there by the time the motor grinds to a halt. That’s OK! What we’re looking for in this case is oscillation. We want the motor to overshoot its goal, and swing back in the opposite direction. If it’s doing this multiple times, perfect. If not, double Kp and try again until it DOES do this. Keep doubling P until you have a good oscillating starting point.

## Tuning D

Remember our purpose for the D term is to SLOW down if we’re approaching the goal too quickly. It prevents overshoot, and it seems like exactly the behavior we need right now. Once you have an oscillation using only a P value, try setting your initial Kd to 10x to 100x of your P value. NOW comes the tuning. Has it stopped oscillating? If not, increase your D until it does. If not, reduce your D until it does, then add some back.

At this point, you’re just about critically damped. Especially from a distance, even with just a P and a D term. So why use an I term at all? Well, when you’re pretty close to your goal, but you REALLY want to be RIGHT ON the goal, a PD loop might not cut it. Before proceeding, try it now- set your goal to, say, 20 encoder clicks away from your current position (you can hand turn the wheel to achieve this). When you pull the trigger, the motor might not be getting enough power to move at all. Is it close enough? Maybe, this is up to you. If not, we have: (drum Roll Please)

## Tuning I (that’s a capital i)

Our system is stable, it just needs a bit of *oomph* at the end to get to its destination. Here’s the thing about the I term- it grows like the blob. If you’re REALLY far away, you don’t even want an I term, because it’s just going to grow unreasonably, and when you get close to home, it’s all bulked up, and it’ll cause wild overshoot. So how do we deal with this? We cheat. That’s right. The Talon has a built in function for JUST this occasion. You can set an IntegralZone, which is the amount of error beyond which you clear the integral term. Works great for preventing integral windup, but still corrects when you’re close to home. Here it is:

\_motor.config\_IntegralZone(0, IZone, 0);

That iZone in there? That’s an int, and you can set it in the same place you set your other parameters. Set that to just a little more than your maximum error is when you’re driving with a PD. Then set Ki (your I term) to about 0.001 of Kp and give it a try (a little goes a long way). If it’s too slow to correct near home, that’s OK, try doubling Ki, or increase your Integral Zone a little and repeat until you’re satisfied. PLEASE tune only one variable at a time.

# 

# Chapter 5: Using PID for velocity control

Velocity control is most obviously used for a shooter. For example: we need to spin a flywheel at a constant velocity, and if we’re rapid-firing, we need to supply the motor with more power so that it maintains it’s ideal speed for a consistent shot when we’re firing 1 object or 10 in a row. What’s NOT so obvious is that velocity control is what you need to drive a tank straight. Think this way: In order to go straight, I really need to turn the drive on both sides at exactly the same distance, at exactly the same rate. You also need to overcome things like extra friction, motor direction preference, and external obstacles. So far we’ve been doing ‘Drive Straight’ by counting pulses, and giving the motor that falls behind extra power. The biggest problem here is that we’re already behind, and we’ve noticed that when we try to go faster, it becomes unstable and we can’t correct for the error.

We need to react faster, but how? If you connect your encoder directly to the Talon, it computes velocity at each encoder tick by measuring the exact time since the last one. Additionally, it reads encoders at 4x the normal rate, so we’re getting velocity updates 1024 times per revolution of the encoder. Now that we’ve got fast updates, why don’t we let the Talon handle the control as well? All we have to do is set the rules for our controller to follow. You’ve already read about the PID and our test setup, so let’s jump right into tuning. Note that to apply this to a tank drive, the same tuning can apply to both sides.

## New Constant: Tuning F

F?! What the F is F?! It’s Feed-forward. Up until now, we’ve been discussing Feedback. Feedback is monitoring your system, and making corrections when it’s not where you want it to be. Feed FORWARD actually lets you make an adjustment BEFORE the disturbance happens. Let’s say you have a 15’ tall satellite dish sitting outside, and it’s following a moving satellite. You also have a bunch of wind speed sensors 30’ away, in a circle all around your dish. When the sensors pick up a wind change, you can compensate for it before the error occurs. Feedback would be waiting until the disturbance has occurred and THEN correcting for it.

In our case, since we’re controlling velocity, we’re going to use feed-forward to set our approximate initial speed. It’s a way to say “You want to go X velocity, I’m going to start by giving the motor X times Kf” (Kf is your feed-forward constant).

So let’s start finding our Kf. Using our same physical model, let’s have a teleop that lets you control motor speed with the Y axis, like this:

public void teleopPeriodic() {

//Output Encoder Values, joystick values, and velocity in encoder pulses per 100msec

double velocity = \_motor.getSelectedSensorVelocity(0);

//and put it up on the dashboard

SmartDashboard.putNumber("Motor Encoder Position", \_motor.getSelectedSensorPosition(0));

SmartDashboard.putNumber("Motor Encoder Velocity", velocity);

double axis=joystick.getY();

SmartDashboard.putNumber("Motor Encoder Velocity", axis);

\_motor.set(ControlMode.PercentOutput, axis);

}

This is run with the physical demo board above (pictured on page 5). Push your joystick to full forward, it’ll get up to speed pretty fast. The dashboard should report that you’re feeding a power setting of 1023 (full power). If you adapt this to a robot, make sure you’ve got enough running room to reach full speed in low gear, and you’re not still accelerating.

What you should have on your dashboard at full speed is the measured velocity (in encoder counts/100ms) when giving the motor controller full power. In my case, that number is 1500.

That velocity represents the number of encoder counts per 100ms. Kind of makes sense, after a 7:1 gear ratio, it means that my encoder is spinning at (1500/1024) \* 10 \*60sec/1min = 878rpm. The motor is running 878 \* 7 = 6152 rpm.

So what’s our Kf? Easy- it is equal to power to the motor in native units (1023) divided by measured velocity (1500), or in this case: 1023/1500 = 0.682. THAT is your Kf. We just measured it. Boom.

## Tuning P

We’ve got an F, but all this will do is set a basic power level depending on how fast you want to go. It won’t adjust accurately for every speed, and it won’t overcome unexpected friction or obstacles. Let’s add in a little P (proportional) response to make up for the difference.

Let’s change our teleop code a bit to help us tune. We’re going to use the Z axis of our joystick (which reads -1 to +1), and multiply it by 1500. When we hit the trigger of the joystick, we want to set the ‘Velocity Target’ to be Z \* 1500. Why 1500? Because 1500 is the fastest I could make my motor go at full power. No point in setting a target you can’t attain.

Here’s the new code:

public void teleopPeriodic() {

//Output Encoder Values, joystick values, and velocity in encoder pulses per 100msec

double velocity = \_motor.getSelectedSensorVelocity(0);

//info on the dashboard for the win!

SmartDashboard.putNumber("Motor Encoder Position", \_motor.getSelectedSensorPosition(0));

SmartDashboard.putNumber("Motor Encoder Velocity", velocity);

//trigger is hit, which means we're letting the Talon control the speed to wherever Z is

// talon will adjust power to maintain set velocity.

if (joystick.getRawButton(1)) //trigger is hit!

{

double targetSpeed=joystick.getZ()\*1500;

\_motor.set(ControlMode.Velocity, targetSpeed);

SmartDashboard.putNumber("Trigger hit", targetSpeed);

SmartDashboard.putNumber("Z value:",joystick.getZ());

}

//reguluar teleop. Just give the motor power wherever you have the joystick set

else { //trigger is not hit

double axis=joystick.getY();

\_motor.set(ControlMode.PercentOutput, axis);

}

}

For a full listing, check Appendix C: Velocity control in Java.

What’s missing above is where we set our control coefficients (K,P,I,D, and F). Let me show you that here:

\_motor.config\_kF(0, 0.683, 0); //set so no-load velocity = Z axis (-1500 to 1500)

\_motor.config\_kP(0, 0, 0); //a little P factor to make up for when loads are applied

\_motor.config\_kI(0, 0, 0);

\_motor.config\_kD(0, 0, 0);

You’ll note that I already plugged a 0.683 in for kF, which is what I had calculated above. From here, when you invoke this line, you’re in velocity controlled mode:

\_motor.set(ControlMode.Velocity, targetSpeed);

Give it a shot! Right now our Kp should be 0, which means that we’re running with no feedback. Since the above code puts your target and your current velocity on the dashboard, you’ll notice immediately that there’s plenty of error between the two (error is measured in counts/100ms). That error gets much worse when you apply even a slight load, because there’s no feedback to compensate. So, let’s compensate! A little proportional gain (P) should do the trick. Let’s start with a P of 0.1 meaning this:

\_motor.config\_kP(0, 0, 0); //a little P factor to make up for when loads are applied

Try it out! If it still lags when loads are applied, double P and try it again. At some point, your system will become unstable, meaning that it’s constantly overshooting. An overshoot or oscillation of say 3 (in encoder counts/100msec) is OK, an overshoot of 100 is probably way too much, and it’ll come out in a herky jerky motion as it causes a gearing backlash.

If you’ve found that your P goes from too little to too much, set it to too little, and give it a little bit of I. Try 1/100 of the current P gain (remember, a little I goes a long way).

## Drive Straight

Earlier, I had mentioned that we would use PID velocity control for drive straight; here we go.

1. Usually, a drive train has 2 motors on each side. This also means 2 motor controllers. Make the second one on each gearbox follow the first like this:
   1. \_motortwo.follow(\_motor);
2. Set the same factors on the left side as the right side. While they’re a little bit different, your proportional gain will make up for extra friction, direction preference, etc.
3. In your autonomous init, clear your encoders with:
   1. \_motor.setSelectedSensorPosition(0);
4. After setting your P, I, D, F (and IZone if applicable), set an identical velocity to each master motor controller.
   1. DON’T set it as fast as you can! One will have a higher top speed, so don’t set velocity above, say, 90% of your fastest ground speed.
5. Watch your encoders for a desired endpoint, and cut the power. If you like, you can even ramp up/down for soft start/stops.

# Appendix A: CTRE Phoenix installation

Link to CTRE Toolsuite:

<http://www.ctr-electronics.com/talon-srx.html#product_tabs_technical_resources>

Phoenix Framework Installer:

<http://www.ctr-electronics.com/hro.html#product_tabs_technical_resources>

Toolsuite Installation guide:

<https://www.ctr-electronics.com/downloads/pdf/CTRE%20Toolsuite%20Installation%20Guide.pdf>

All examples in this guide use the Phoenix Toolsuite in Java.

# Appendix B: Position PID in Java

/\*----------------------------------------------------------------------------\*/

/\* Position PID

\* Copyright (c) 2017-2018 FIRST. All Rights Reserved. \*/

/\* Open Source Software - may be modified and shared by FRC teams. The code \*/

/\* must be accompanied by the FIRST BSD license file in the root directory of \*/

/\* the project. \*/

/\*----------------------------------------------------------------------------\*/

package org.usfirst.frc.team5109.robot;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The following are the Talon libraries used for control of the Talon SRX

\* They are installed by the Phoenix toolsuit located

\* http://www.ctr-electronics.com/hro.html#product\_tabs\_technical\_resources

\*/

import com.ctre.phoenix.motorcontrol.ControlMode;

import com.ctre.phoenix.motorcontrol.FeedbackDevice;

import com.ctre.phoenix.motorcontrol.can.TalonSRX;

import edu.wpi.first.wpilibj.Joystick;

import edu.wpi.first.wpilibj.TimedRobot;

import edu.wpi.first.wpilibj.smartdashboard.SendableChooser;

import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;

/\*\*

\* The VM is configured to automatically run this class, and to call the

\* functions corresponding to each mode, as described in the IterativeRobot

\* documentation. If you change the name of this class or the package after

\* creating this project, you must also update the build.properties file in the

\* project.

\*/

// Check it out- TimedRobot instead of IterativeRobot- this one runs every 20ms no matter what!

public class Robot extends TimedRobot {

private static final String kDefaultAuto = "Default";

private static final String kCustomAuto = "My Auto";

private String m\_autoSelected;

private SendableChooser<String> m\_chooser = new SendableChooser<>();

int \_loops=0; // this is a temporary variable used to count off dashboard updates later

//Let's define our constants here. Why? Because they're easy to find when tuning.

//They are implemented below

private static final double Kp=2.0;

private static final double Ki=0.02; // .01 \* P

private static final double Kf=0.0; //no feed-forward on position control

private static final double Kd=200;

private static final int IZone=150;

// I only have one joystick. here it is:

Joystick joystick = new Joystick(0);

//I also only have one Talon connected:

TalonSRX \_motor = new TalonSRX(1);

/\*\*

\* This function is run when the robot is first started up and should be

\* used for any initialization code.

\*/

@Override

public void robotInit() {

m\_chooser.addDefault("Default Auto", kDefaultAuto);

m\_chooser.addObject("My Auto", kCustomAuto);

SmartDashboard.putData("Auto choices", m\_chooser);

//Configuration stuff.

\_motor.configSelectedFeedbackSensor(FeedbackDevice.QuadEncoder);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Ramps. The argument is how many seconds it takes to go from 0 to 100%

\* during voltage control. There is a ramp for both open loop and closed

\* (controlled) loop

\*/

\_motor.configOpenloopRamp(0.2);

\_motor.configClosedloopRamp(0.2);

// Talon setup. Setting up + and - output for max and nominal.

//These should never really change

\_motor.configNominalOutputForward(0, 0);

\_motor.configNominalOutputReverse(0, 0);

\_motor.configPeakOutputForward(1,0);

\_motor.configPeakOutputReverse(-1, 0);

//set this if your encoder is hooked up backwards (A<--->B)

//mine is backwards, so I've set it to true

\_motor.setSensorPhase(true);

//implementation of tuning constants. First argument is PID slot. We're only

//using slot 0. If we had multiple tuning constants for the same controller

// for different purposes, we can also have up to 3 slots pre-programmed

\_motor.config\_kF(0, Kf, 0); //No feed-forward on position control

\_motor.config\_kP(0, Kp, 0); //P factor!

\_motor.config\_kI(0, Ki, 0);

\_motor.config\_kD(0, Kd, 0); //note- these are defined above so they're easy to find and change.

//one last term- configIntegralZone

\_motor.config\_IntegralZone(0, IZone, 0);

}

/\*\*

\* This autonomous (along with the chooser code above) shows how to select

\* between different autonomous modes using the dashboard. The sendable

\* chooser code works with the Java SmartDashboard. If you prefer the

\* LabVIEW Dashboard, remove all of the chooser code and uncomment the

\* getString line to get the auto name from the text box below the Gyro

\*

\* <p>You can add additional auto modes by adding additional comparisons to

\* the switch structure below with additional strings. If using the

\* SendableChooser make sure to add them to the chooser code above as well.

\*/

@Override

public void autonomousInit() {

m\_autoSelected = m\_chooser.getSelected();

// m\_autoSelected = SmartDashboard.getString("Auto Selector",

// kDefaultAuto);

System.out.println("Auto selected: " + m\_autoSelected);

}

/\*\*

\* This function is called periodically during autonomous.

\*/

@Override

public void autonomousPeriodic() {

switch (m\_autoSelected) {

case kCustomAuto:

// Put custom auto code here

break;

case kDefaultAuto:

default:

// Put default auto code here

break;

}

}

/\*\*

\* This function is called periodically during operator control.

\*/

@Override

public void teleopPeriodic() {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Here's what I'm doing during teleop: If you're NOT holding the trigger,

\* I operate like a normal motor- just piping Y axis of my joystick over

\* to the motor. Full stick forward= full motor power.

\* I'm also reading the Z axis on the joystick. This gives me a number from

\* -1 to 1. When you hit the trigger, I set the target of the Talon

\* to Z\*4096\*10 which sets a position target of anywhere from -40 to 40 full

\* Turns of the encoder wheel. I have the PID set to get me there as fast

\* as possible, and the talon takes care of it. You'll also notice there

\* are some SmartDashboard numbers thrown up there to check accuracy.

\*/

double targetPosition=joystick.getZ()\*4096\*10;

if (joystick.getTrigger())

{

\_motor.set(ControlMode.Position, targetPosition);

if (++\_loops >= 10) {

\_loops = 0;

SmartDashboard.putNumber("Trigger hit", targetPosition);

SmartDashboard.putNumber("Z value:",joystick.getZ());

}

}

//reguluar teleop. Just give the motor power wherever you have the joystick set

else if (!joystick.getRawButton(1)) {

double axis=joystick.getY();

\_motor.set(ControlMode.PercentOutput, axis);

if (++\_loops >= 10) {

\_loops = 0;

SmartDashboard.putNumber("Trigger hit", targetPosition);

}

}

SmartDashboard.putNumber("Current Pos:", \_motor.getSelectedSensorPosition(0));

}

/\*\*

\* This function is called periodically during test mode.

\*/

@Override

public void testPeriodic() {

}

}

# Appendix C: Velocity control in Java

/\*----------------------------------------------------------------------------\*/

/\* Position PID

\* Copyright (c) 2017-2018 FIRST. All Rights Reserved. \*/

/\* Open Source Software - may be modified and shared by FRC teams. The code \*/

/\* must be accompanied by the FIRST BSD license file in the root directory of \*/

/\* the project. \*/

/\*----------------------------------------------------------------------------\*/

package org.usfirst.frc.team5109.robot;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The following are the Talon libraries used for control of the Talon SRX

\* They are installed by the Phoenix toolsuit located

\* http://www.ctr-electronics.com/hro.html#product\_tabs\_technical\_resources

\*/

import com.ctre.phoenix.motorcontrol.ControlMode;

import com.ctre.phoenix.motorcontrol.FeedbackDevice;

import com.ctre.phoenix.motorcontrol.can.TalonSRX;

import edu.wpi.first.wpilibj.Joystick;

import edu.wpi.first.wpilibj.TimedRobot;

import edu.wpi.first.wpilibj.smartdashboard.SendableChooser;

import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;

/\*\*

\* The VM is configured to automatically run this class, and to call the

\* functions corresponding to each mode, as described in the IterativeRobot

\* documentation. If you change the name of this class or the package after

\* creating this project, you must also update the build.properties file in the

\* project.

\*/

public class Robot extends TimedRobot {

//Crap that the timedrobot sets up for you.

private static final String kDefaultAuto = "Default";

private static final String kCustomAuto = "My Auto";

private String m\_autoSelected;

private SendableChooser<String> m\_chooser = new SendableChooser<>();

int \_loops=0; // this is a temporary variable used to count off dashboard updates later

//Let's define our constants here. Why? Because they're easy to find when tuning.

//They are implemented below

private static final double Kf=0.680; // woo! Feed forward baby!

private static final double Kp=1.0; // and a little proportional to catch a load

private static final double Ki=0.02;

private static final double Kd=0.0;

private static final int IZone=50;

// I only have one joystick. here it is:

Joystick joystick = new Joystick(0);

//I also only have one Talon connected:

TalonSRX \_motor = new TalonSRX(1);

TalonSRX \_motor2 = new TalonSRX(2);

/\*\*

\* This function is run when the robot is first started up and should be

\* used for any initialization code.

\*/

@Override

public void robotInit() {

m\_chooser.addDefault("Default Auto", kDefaultAuto);

m\_chooser.addObject("My Auto", kCustomAuto);

SmartDashboard.putData("Auto choices", m\_chooser);

//Configuration stuff for Talon and Feedback encoder

\_motor.configSelectedFeedbackSensor(FeedbackDevice.QuadEncoder);

\_motor.setSensorPhase(true); //set this if your encoder is hooked up backwards (A<--->B)

//ramp for both open and closed loop. Slows you down, so you can't go 0-100% in an instant.

\_motor.configOpenloopRamp(0.0);

\_motor.configClosedloopRamp(0.0);

// Talon setup. Setting up + and - output for max and nominal.

// These never have to change

\_motor.configNominalOutputForward(0, 0);

\_motor.configNominalOutputReverse(0, 0);

\_motor.configPeakOutputForward(1,0);

\_motor.configPeakOutputReverse(-1, 0);

//PIDF (F= feedforward for the purpose of velocity control)

// setting slot 0 PID, because we're only implementing ONE pid on this Talon

\_motor.config\_kF(0, Kf, 0); //set so no-load velocity = Z axis (-1500 to 1500)

\_motor.config\_kP(0, Kp, 0); //a little P factor to make up for when loads are applied

\_motor.config\_kI(0, Ki, 0);

\_motor.config\_kD(0, Kd, 0);

\_motor.config\_IntegralZone(0, IZone, 0);

}

/\*\*

\* This autonomous (along with the chooser code above) shows how to select

\* between different autonomous modes using the dashboard. The sendable

\* chooser code works with the Java SmartDashboard. If you prefer the

\* LabVIEW Dashboard, remove all of the chooser code and uncomment the

\* getString line to get the auto name from the text box below the Gyro

\*

\* <p>You can add additional auto modes by adding additional comparisons to

\* the switch structure below with additional strings. If using the

\* SendableChooser make sure to add them to the chooser code above as well.

\*/

@Override

public void autonomousInit() {

m\_autoSelected = m\_chooser.getSelected();

// m\_autoSelected = SmartDashboard.getString("Auto Selector",

// kDefaultAuto);

System.out.println("Auto selected: " + m\_autoSelected);

}

/\*\*

\* This function is called periodically during autonomous.

\*/

@Override

public void autonomousPeriodic() {

switch (m\_autoSelected) {

case kCustomAuto:

// Put custom auto code here

break;

case kDefaultAuto:

default:

// Put default auto code here

break;

}

}

/\*\*

\* This function is called periodically during operator control.

\*/

@Override

public void teleopPeriodic() {

//Output Encoder Values, joystick values, and velocity in encoder pulses per 100msec

double velocity = \_motor.getSelectedSensorVelocity(0);

//info on the dashboard for the win!

SmartDashboard.putNumber("Motor Encoder Position", \_motor.getSelectedSensorPosition(0));

SmartDashboard.putNumber("Motor Encoder Velocity", velocity);

// Why the 1500 multiplyer? Empirically, i gave the motor full speed and found

// a velocity of 1500 coming back off the encoder.

//Velocity is measured in encoder counts flying by per 100ms

double targetSpeed=joystick.getZ()\*1500;

SmartDashboard.putNumber("Trigger hit", targetSpeed);

SmartDashboard.putNumber("Z value:",joystick.getZ());

//trigger is hit, which means we're letting the Talon control the speed to wherever Z is

// talon will adjust power to maintain set velocity.

if (joystick.getRawButton(1))

{

\_motor.set(ControlMode.Velocity, targetSpeed);

}

//reguluar teleop. Just give the motor power wherever you have the joystick set

else {

double axis=joystick.getY();

\_motor.set(ControlMode.PercentOutput, axis);

}

if (joystick.getRawButton(2))

{

\_motor.setSelectedSensorPosition(0);

}

}

/\*\*

\* This function is called periodically during test mode.

\*/

@Override

public void testPeriodic() {

}

}