SENG-5831 Lab Assignment #2

Introduction

For lab assignment number two, we were asked to implement a PID loop to control both the speed and the position of a motor. We were given the calculation, and not much else. I was able to get the position portion of the PID loop working well fairly quickly. Due to some misunderstandings by myself, and some fat-fingering in the code, the speed portion took me a little longer. The calculations are the same for both, but it was what to pass into the calculation that I had trouble with.

The next thing to figure out was the rate at which we wanted to update the PID loop. Based on the assigned readings, this value should be between 1/10th and 1/100th of our desired settling time. I picked a desired settling time of 500ms. I figured that would be a nice modest value for the equipment we were working with. Having selected 500ms, that would put our PID update loop at between 5ms and 50ms. One other thing to take into consideration was the speed at which our encoder counts are coming into the system. We don't want our PID loop to update faster than we're getting encoder counts. What I did to determine that was run the motor at the maximum speed, and the minimum speed, and got the differences between encoder counts. At the fastest speed, the encoder counts were coming in once every 130 microseconds. At the slowest speed, encoder counts were coming in once every 8 or so milliseconds. Based on the slower of the two numbers, 5ms would not be an appropriate PID update time, so I selected 10ms. This is a fairly fast number that is still within the range for my desired settling time. There will be opportunities to play with this number throughout the lab.

One thing I decided to do was to not have my velocity be in counts per second. I just calculated them in counts per 10ms, because the velocity would just have to be multiplied or divided by 100, and I couldn't really see the benefit of that, so when you enter 40 as your desired speed, that is 40 counts per 10ms, which would be 4000 counts per second. I may change my mind before the end of this lab, but for now, that is the way it is.

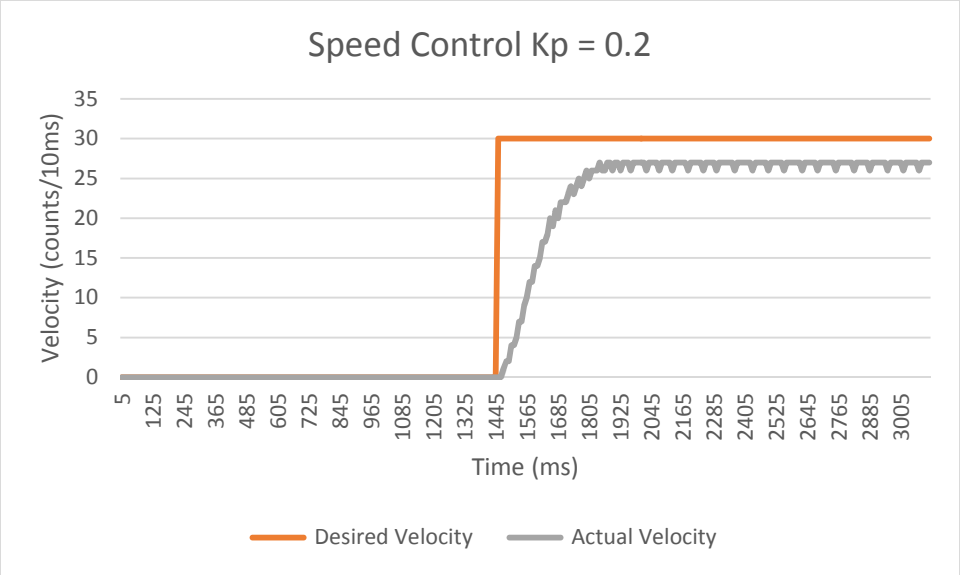
Without further ado, on to the experimentation!

Part 1 - Speed Controller

First we're set to use only proportional gain to maintain speed. I've set my P to increase by values of 0.2. I had this value set to 1, but there wasn't as much room for experimentation at that value. I then dropped it down to 0.1, but that is too big of a pain to increase Kp to larger values. For this portion of the lab, my desired velocity will be 3,000 counts per second.

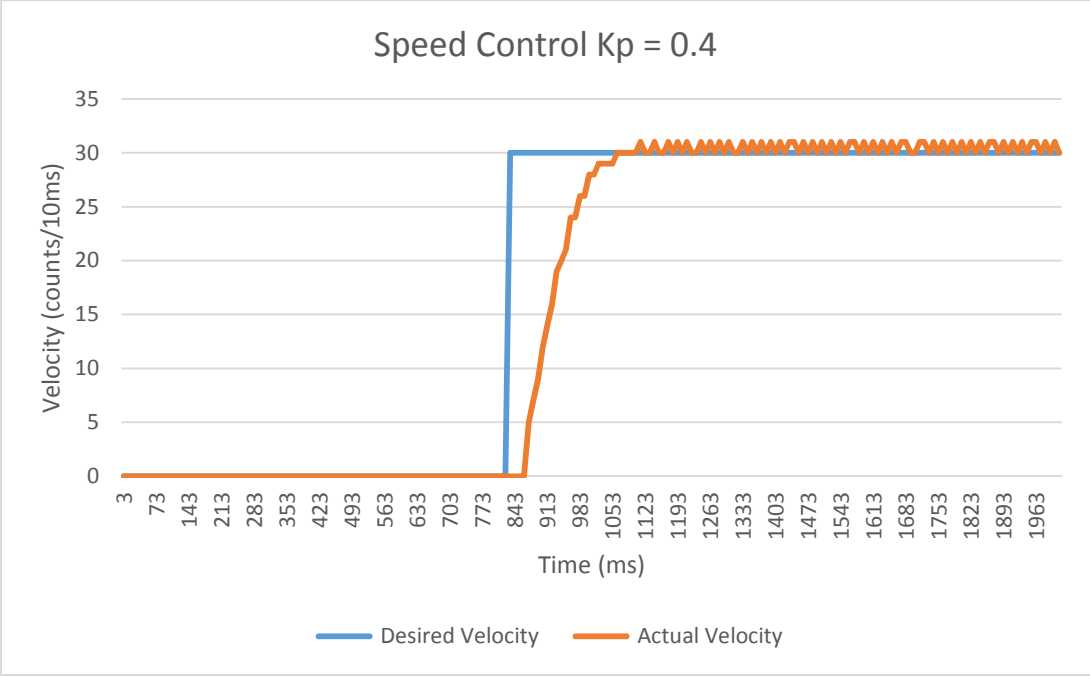
Use Only Proportional Gain

The first thing we'll do is set our proportional gain to the smallest value allowed in my system, which is 0.2. After running our command, we can graph our desired velocity and our actual velocity.

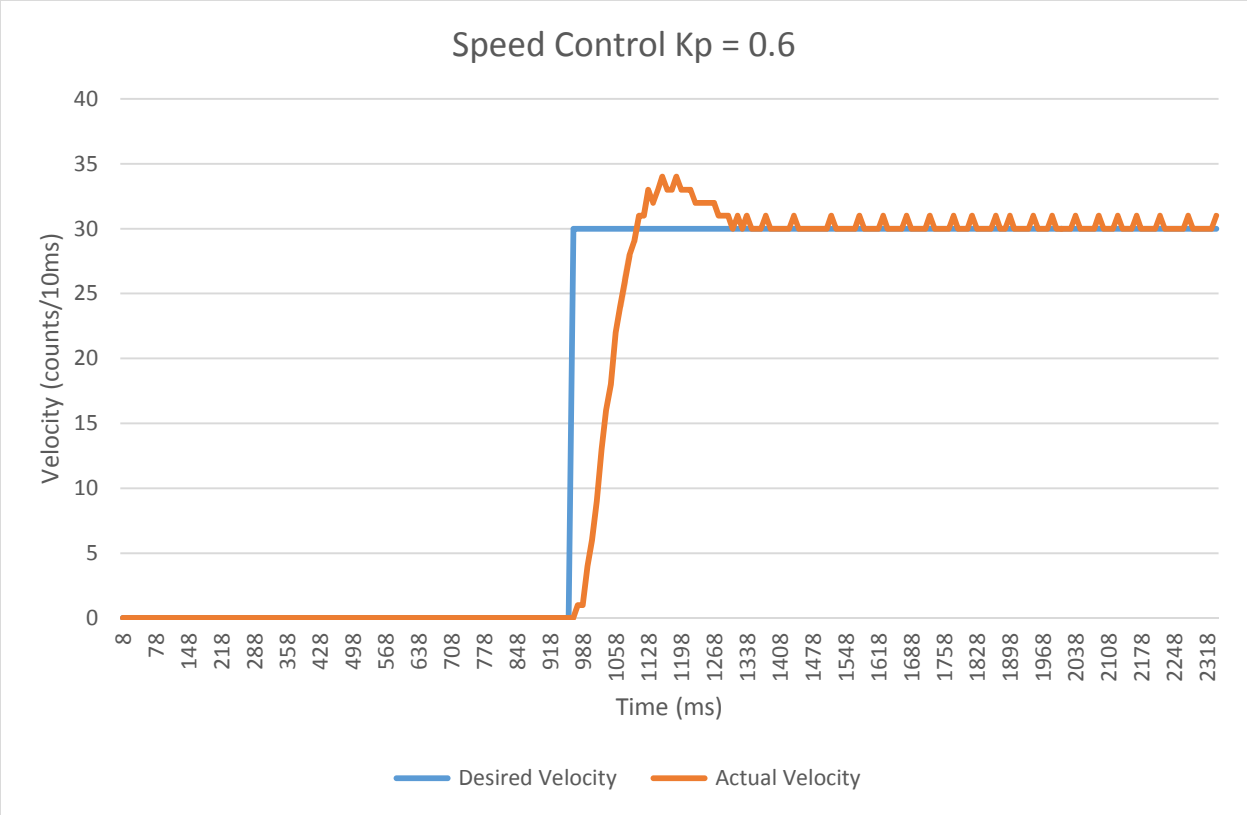


As we can see, the proportional gain isn't quite large enough to even get us to our desired velocity. As we get closer to our desired velocity, the error gets smaller, and as the error gets smaller, the calculation becomes so small that the result of the PID calculation isn't even enough to move the motor closer to the desired position.

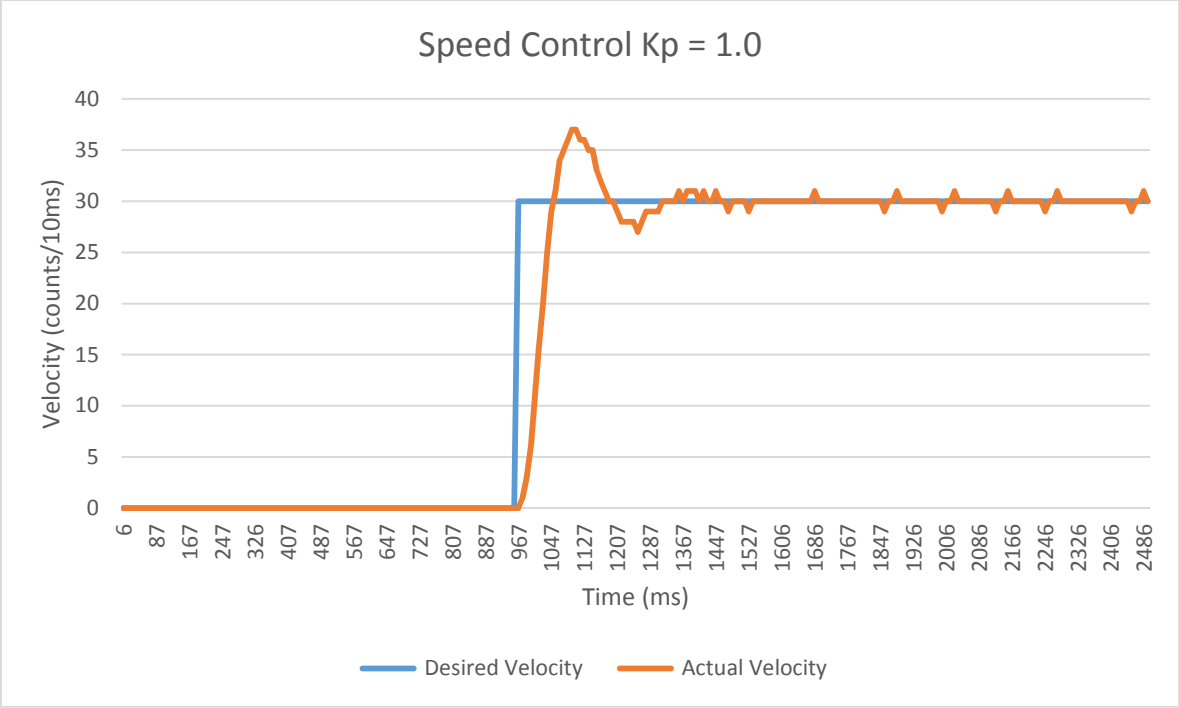
Let's try to bump Kd up another notch to 0.4 and run the same test.



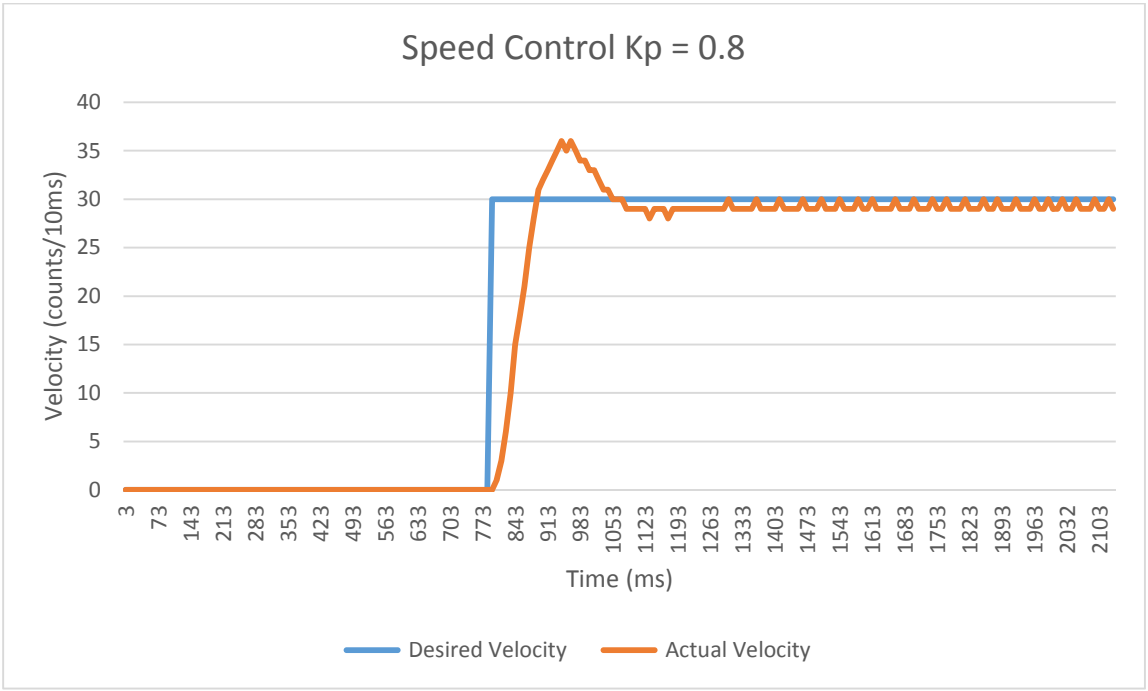
OK, now we're getting somewhere. We can see that the proportional gain is now strong enough to get us to our actual desired velocity. Let's bump it up a few more times and see if we can make the controller get there a little faster.



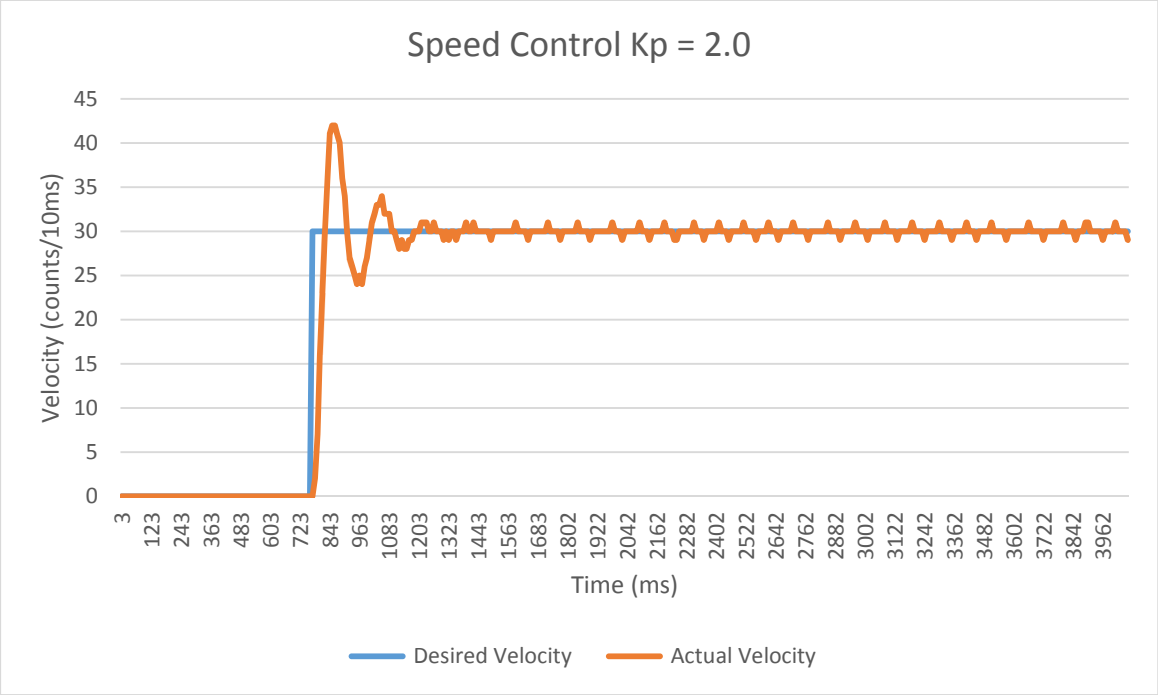
Alright, with a proportional gain of 0.6, we can see that we now overshoot our desired velocity by about 3 or 4 counts per 10ms. We can sort of expect this because our last graph went right up to the line. If you increase your gain by 50%, I think it is fair to expect you'll overshoot your desired value by a bit. Let's take it a little bit further and see if we can get our settling time a little bit lower before we start oscillating.



We can see that we've shortened our time to when we first reach the desired velocity by about half, but we haven't actually shortened the settling time. We've probably increased it a little bit. Let's take a look at a Kp setting of 0.8 and see if we're any better off.



It looks like we oscillate less, but not much less. We should probably be able to still with a Kp of 1.0, and add integral gain from there. First, let's get some real oscillation into this system. Let's try a Kp of 2.0.

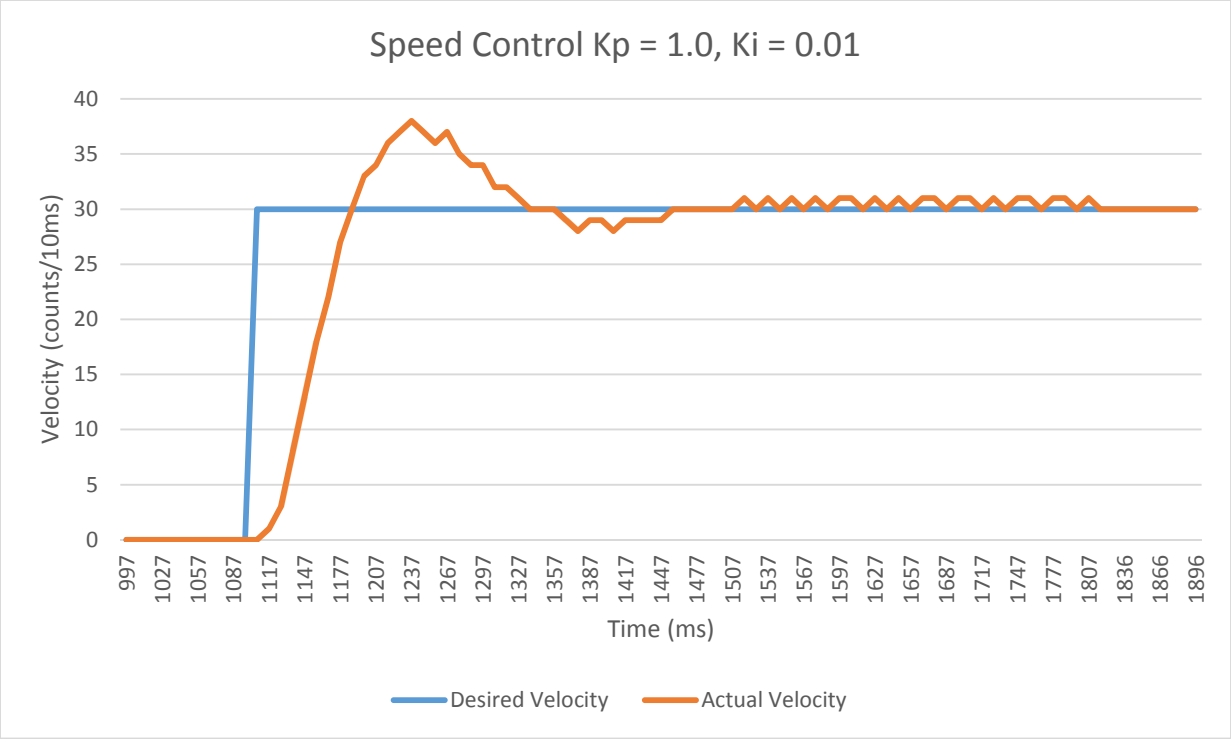


We can see here that we never really settle into our desired velocity. I could even hear the motor going back and forth even just this little bit. The proportional gain is now large enough that even when we get a little far away, we over compensate the other way. This happens because a smaller number times a larger number is still larger than we want, and we just go back and forth on there forever. We do, however, get up to our desired velocity very fast initially.

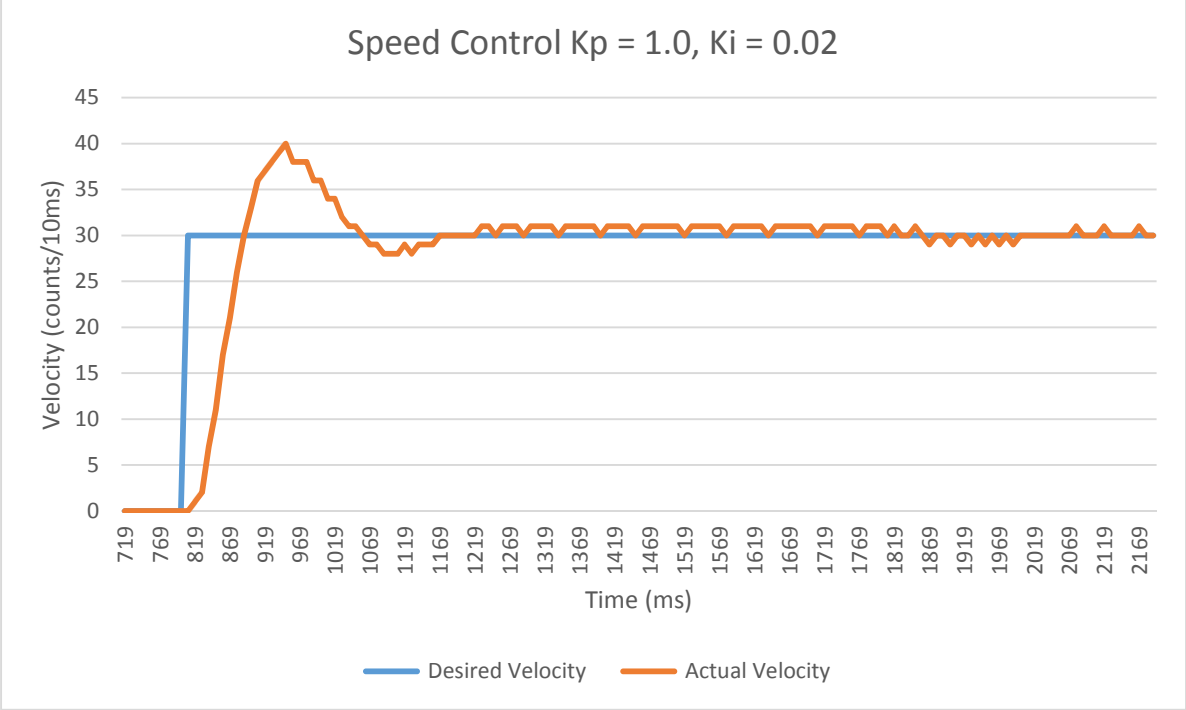
Add In Integral Gain

The integral gain is meant to solve the problem of the actual value never reaching the desired value by adding up the error over time and using that value as part of the final gain. What this means is that even if we are away from our desired value by just a small amount, that small amount will add up over time and add to our overall gain, moving us closer to our desired value. We can just raise the proportional gain to achieve the desired value quicker, but that come with undesired side effects such as overshoot and oscillation. For this portion of our lab, we'll use a proportional gain of 1.0. That has a good balance between speed to the desired velocity, and minimal overshoot. Hopefully our integral gain can help us even further.

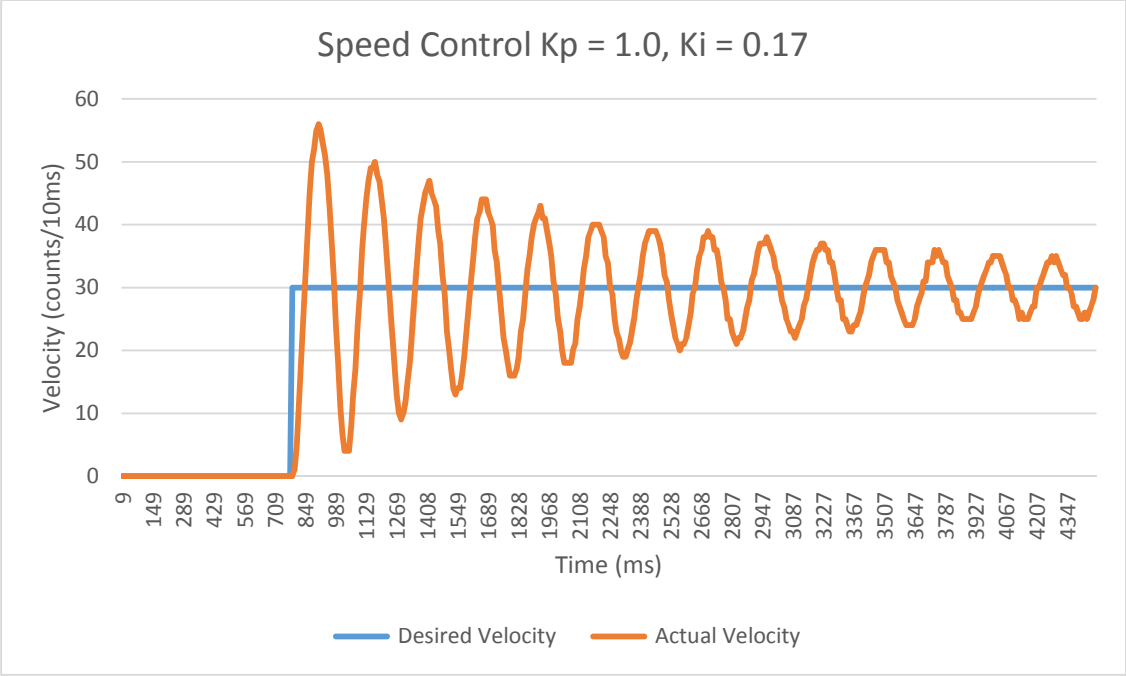
I've set the default change in Ki to be 0.01. This seems like a good value to start with. Using a Kd of 1.0, and a Ki of 0.01, our desired velocity vs actual velocity looks like the following.



We can see that our time to first reach our desired velocity is about 100ms, and we are pretty stable by about 450ms. Let's double Ki and see what the effects are.

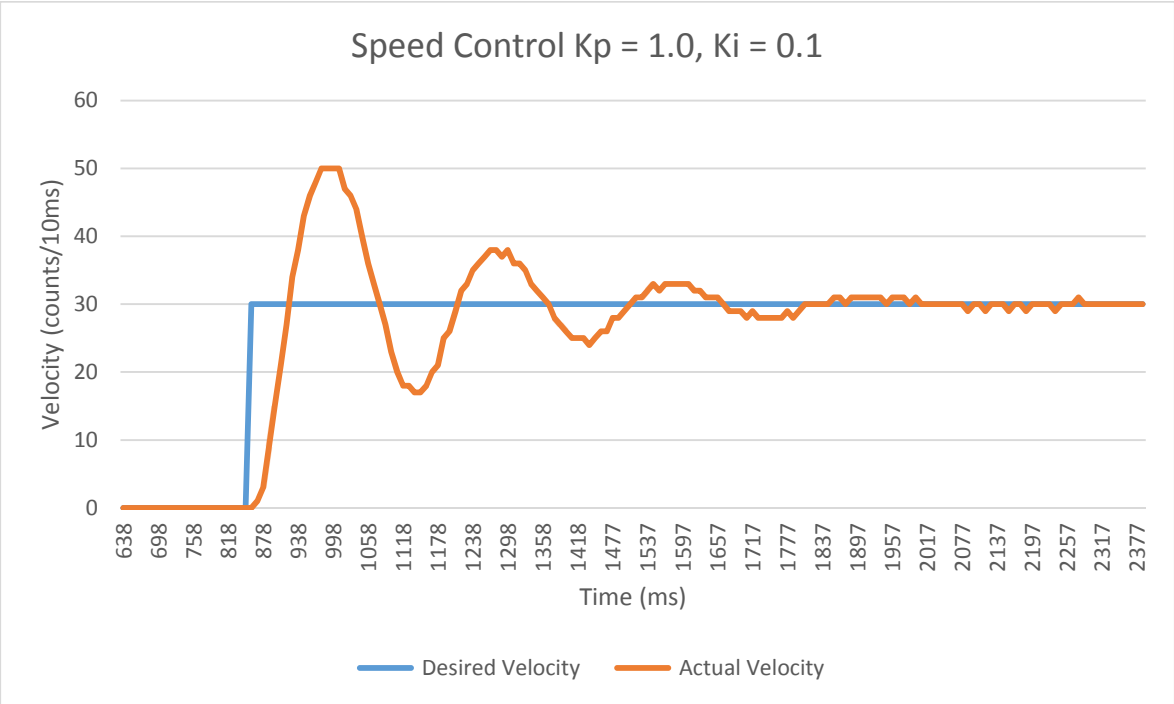


We can see we're a little better off here. We moved our settling time by about 10ms, but not much more than that. It appears we're not gaining much using integral gain. Let's see if we can get some oscillation in this guy.



We can see now that our heightened Ki has gotten us to our desired velocity initially in just under 60ms by summing up the error very vast. This gets us an increased torque, but there isn't much there to slow us down. Because of that, we start adding the positive error above the curve real fast, and immediately overshoot the other direction. This settles down a little bit, but Ki is high enough to keep us oscillating for evermore.

Let's see what we can do if we put our Ki to be 0.1.



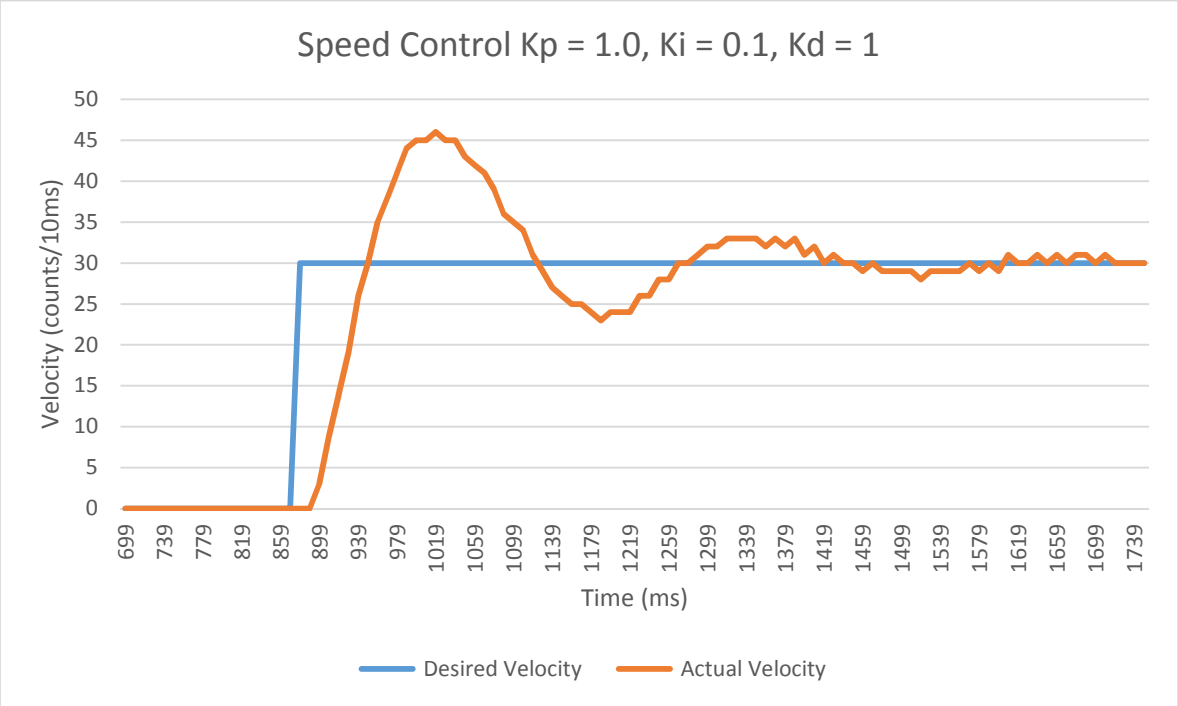
OK, so now we've got a very long settling time, but we get to our initial speed very fast, in about 65ms. We still have some oscillation, but we're going to try and take care of that with our derivative gain.

Add In Derivative Gain

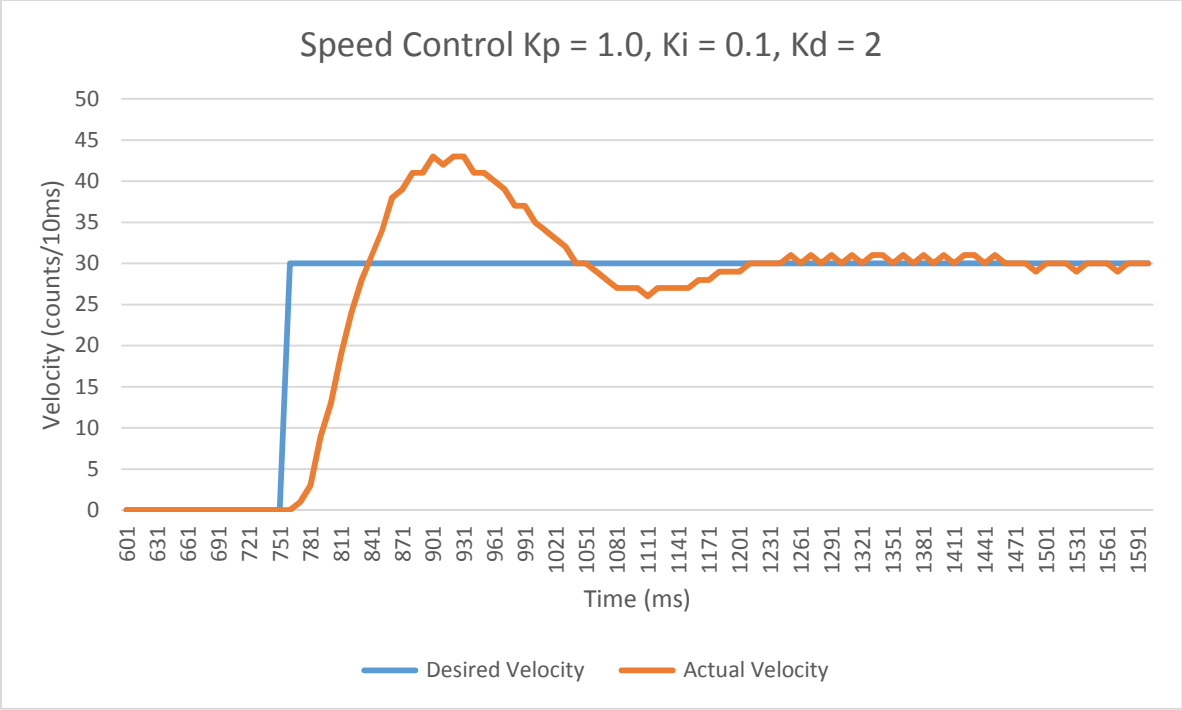
The derivative gain is going to use the slope of the velocity and remove that from the gain calculated using the proportional and integral gain. The theory here is that the faster you're already moving towards your goal, the less gain you want. As you get closer to your target, your velocity becomes less because of the proportional gain, and therefore the amount you take out due to your derivative gain also becomes less.

For our derivative gain, we are going to use an increment/decrement value of 1. I think that will be a sufficient value for our experimentation.

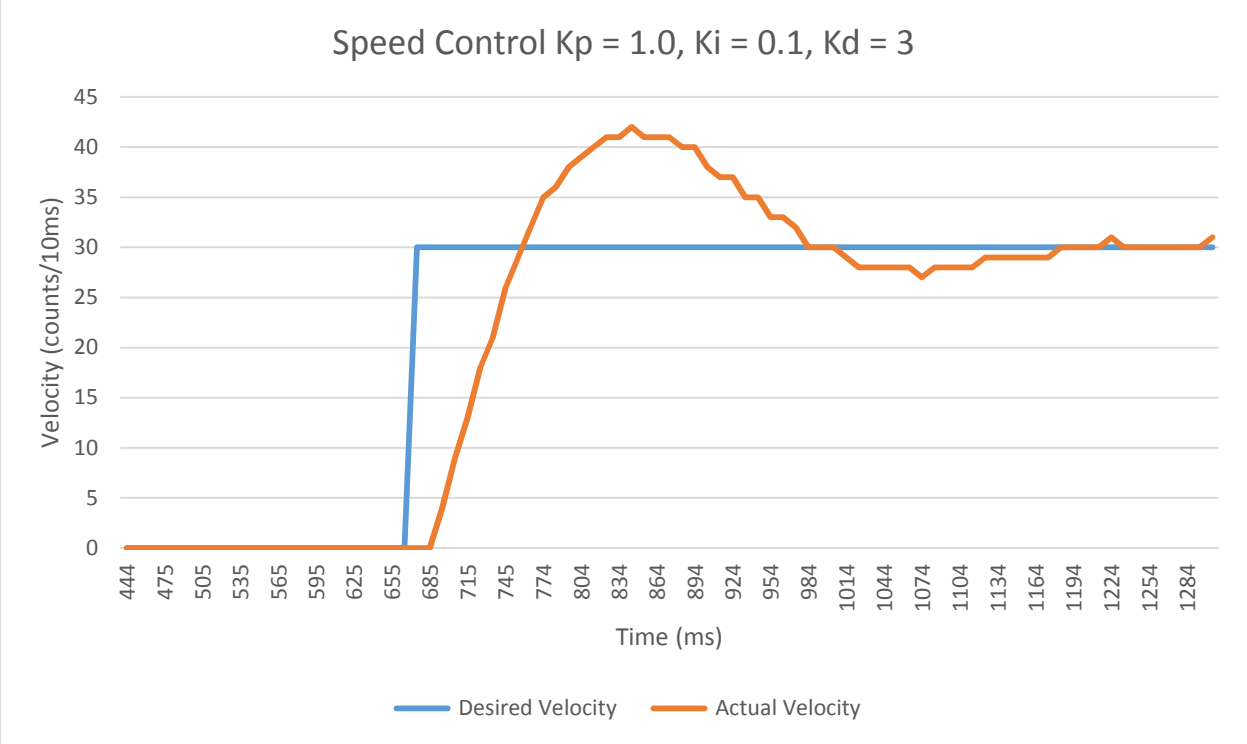
Let's see what we can do for our response with some derivative gain. We'll start with 1.



We can see we've improved our settling time considerably. Without derivative gain, we went back and forth about four different times. We our first stab at derivative gain, we've already limited that to just two obvious oscillations. Well, we can't stop there, so let's try doubling our Kd to 2.



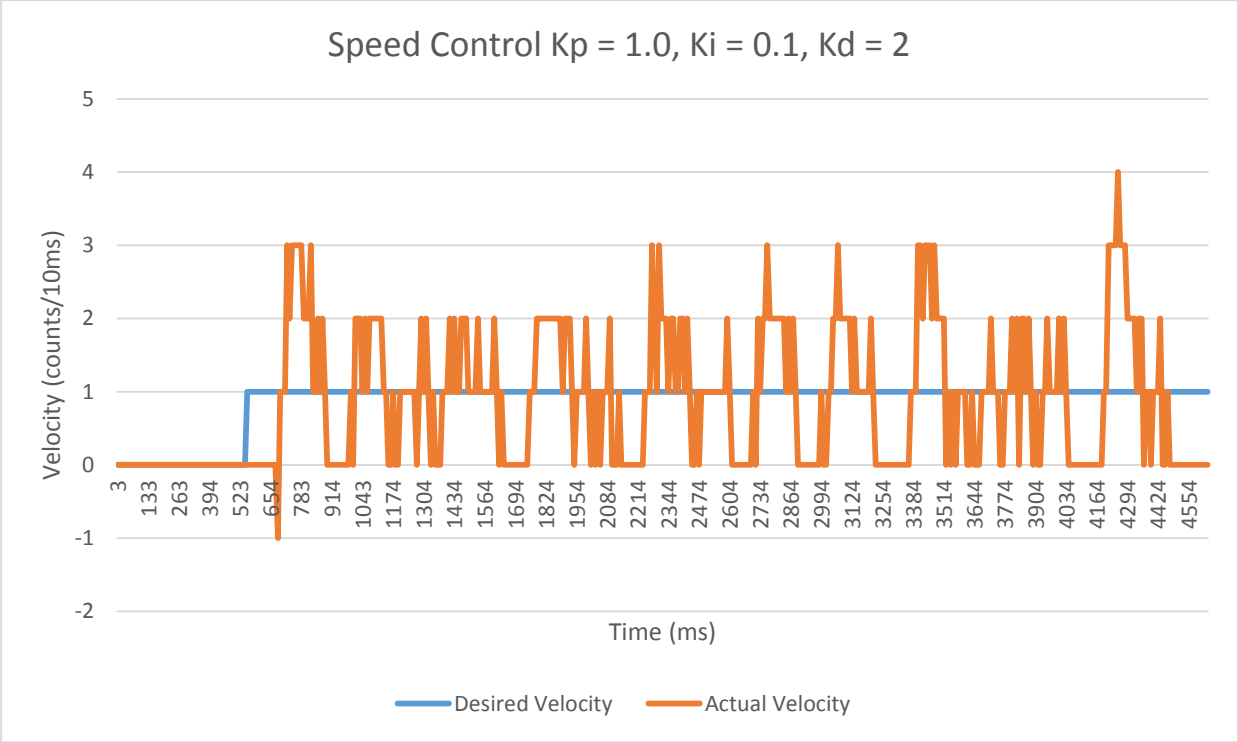
Alright, we're doing better here. We can see that we've only got about one oscillation here, and we're settling in at right about 450ms. We first reach our desired speed after about 70ms, so we lost a little bit of time there, but that's because we are subtracting out gain based on how fast we're going, and the greater Kd is, the more gain we're subtracting out. Let's see if a Kd of 3 helps us at all.



OK, we can see that our response is a little smoother, but at the cost of being a little bit slower. We are now hitting our initial desired velocity at around 85ms, and settling in around 500ms. I think we're going to say our best bet is to have a Kp of 1.0, a Ki of 0.1, and a Kd of 2. That seems to give us our best response, overall. We'll use those values in the next portion of this lab.

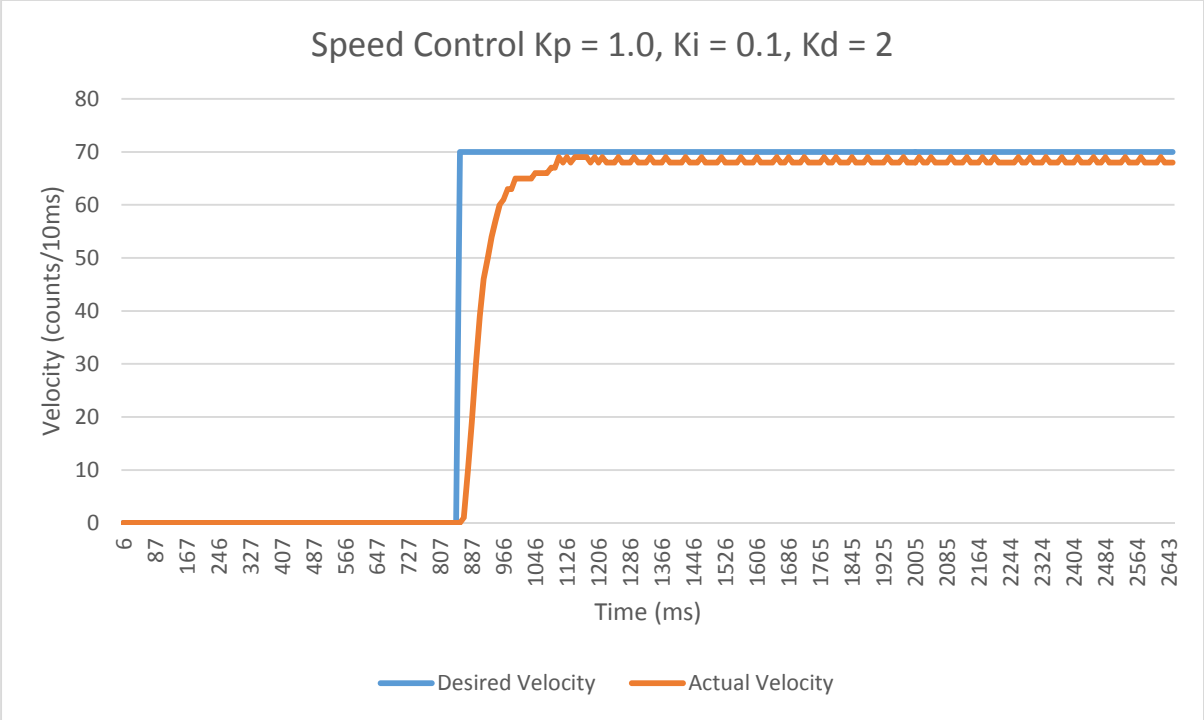
Part 2 – Play With PID Timing Speed

As mentioned previously, we're going to use a proportional gain of 1.0, an integral gain of 0.1, and a derivative gain of 2. Using these gains, I first tried to implement a very slow speed. I tried 100 counts per second. The results are shown below.



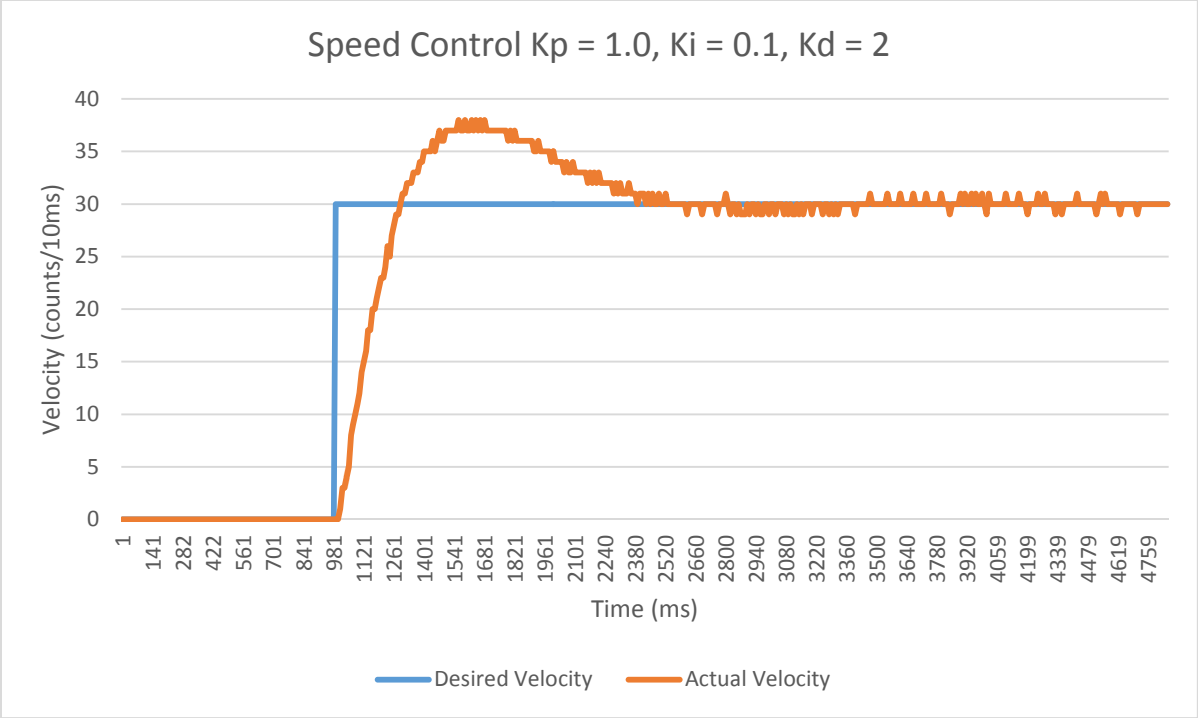
Now, if you'll remember from earlier in this document that my PID is being updated every 10ms. You might also remember that when the motor is driven as slow as the motor allows, the update time was between 8ms and 12ms. Because my PID loop is at times running faster than my counts are being updated, it is possible that I am going to see a velocity of zero. Another thing to consider is that at a 10ms update rate, my desired speed is one count per interrupt. If I happen to hit two counts in an interrupt, I'm going twice as fast as I'd like. There is not much room for error in this case, and that can be seen in the graph above. My actual velocity at times is zero, and even in one case is negative. I believe that is because of the PID update rate being so close to the encoder update rate.

Let's now try to run our motor at max speed. We will have to do a little bit of testing to even figure out where that even is. I'll do some experimentation and see when my motor stops being able to reach my desired speed. After doing some testing, it looks like our maximum speed is about 6900 counts per second. Using that speed, and the gain values from above, we get a graph that looks like this. Not that I set the max speed to be 7000 counts per second to make sure we were maxing out, which is why we don't quite reach our desired velocity.



As we can clearly see, we have much better control of the motor at high speeds. This can be explained because we are always going to be getting non-zero velocity every time we update our PID because our encoder counts are getting updated about once every 130 microseconds or so, so we should have up to eight counts in our velocity calculation. One interesting thing that was happening during this test was that the motor would randomly switch directions. I figured out that this was because the calculation wasn't being limited to the amount of torque the system could handle, which is 255. Because I'm using an integer to store that value, but the PID loop returns a double, which is bigger that an integer, the calculation was overflowing into a negative number, which would then force the motor backwards. This was easily fixed by limiting the torque to our allowable range, which is -255 to 255 inclusive.

Our next task is to slow down the PID calculation. I'm going to do this by leaving the current interrupt at the same interval of 10ms, and simply adding in a variable that will increment every time the interrupt is triggered and then check if the value is divisible by four, and then update the PID that time. This will, in effect, slow our PID loop by one quarter. What I think is going to happen is that we won't be able to settle into our desired speed as fast as before because we won't be checking in on our speed as often. Basically, we could be four times closer to our destination, but not even realize it, and it would do it's best to recover, but it might be too late. For this task, we're going to put our desired velocity back to 3,000 counts per second. Below is the output of such a test.



Now, we can see that this graph looks about the same as the one with the correctly timed PID update with one very important difference. The time scale is about four times wider. We can see that while the response "looks" the same, it is taking about two and a half seconds to finally settle into its desired velocity. This is pretty much what I expected because it just can't update the torque values fast enough. For this reason, I think I've struck a pretty good balance between getting good response at high and low speeds. At a 10ms PID update, we can settle into our desired speeds around a half second, and still control speed pretty nicely all the way down to about 200 counts per second. If we need to go much slower than that, it's time to get ourselves a motor with a different gear ration. Either that, or we're going to have to sacrifice some of our settling time in the name of slower motor control.

We could try updating our PID loop more frequently at this point, but we already know what that is going to look like. It is going to have an even harder time getting non-zero velocities, and we are going to get a graph that looks like our 100 counts per second graph for speeds even higher than that.

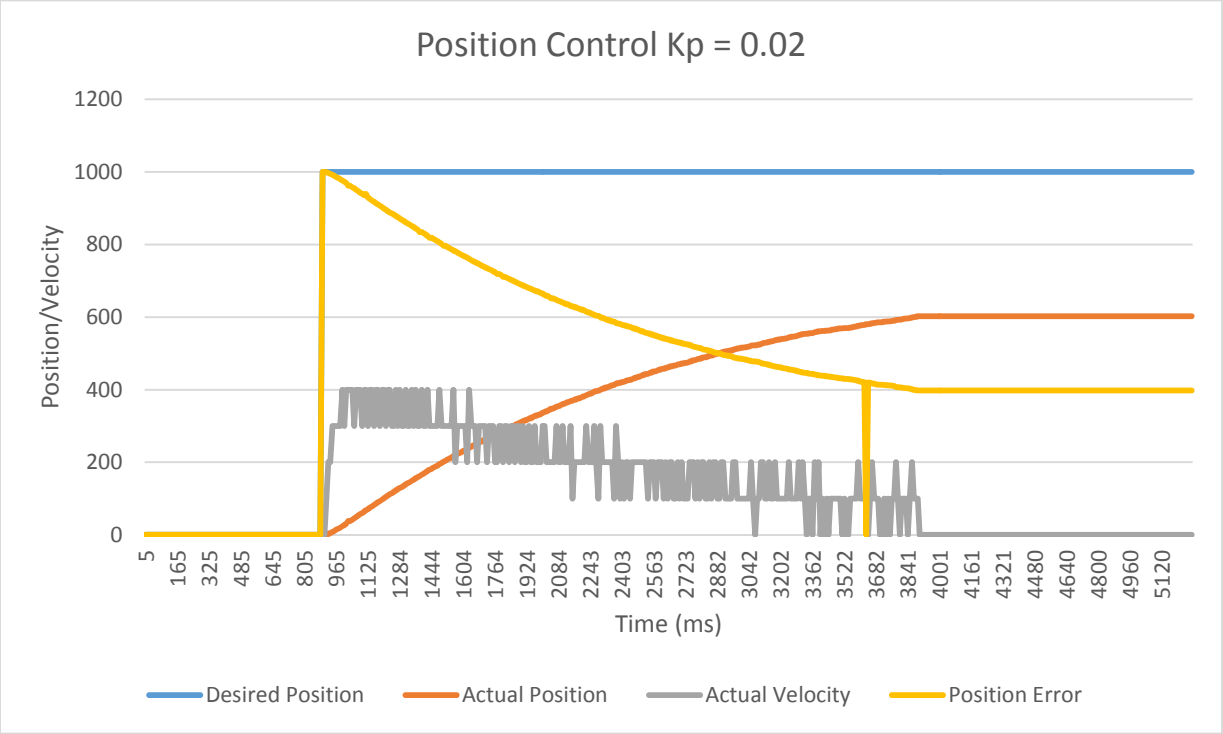
With that, I think we've seen and described what the effects of our PID update rate is. In the article "PID Without A PhD", the author talks about the update rate of a PID loop, and using his method, we've, I feel, correctly applied his theory and come up with a value that suits us well for our speed considerations, and our hardware considerations. On to the position control!

Part 3 – Position Controller

In my opinion, the position controller was much easier to implement. There was no worrying about what the previous torque was, nor did you have to worry about calculating the velocity outside of the PID controller. You could just enter your desired position, and your error, and it gave you the torque back. Pretty sweet deal.

For the position control, a lot of the explanations are going to be similar. The concepts of the proportional, integral and derivative are exactly the same, except the error is the difference in desired versus actual position instead of velocity. The nice part about that is that the PID equation is exactly the same, we just need to change the values that are passed into it.

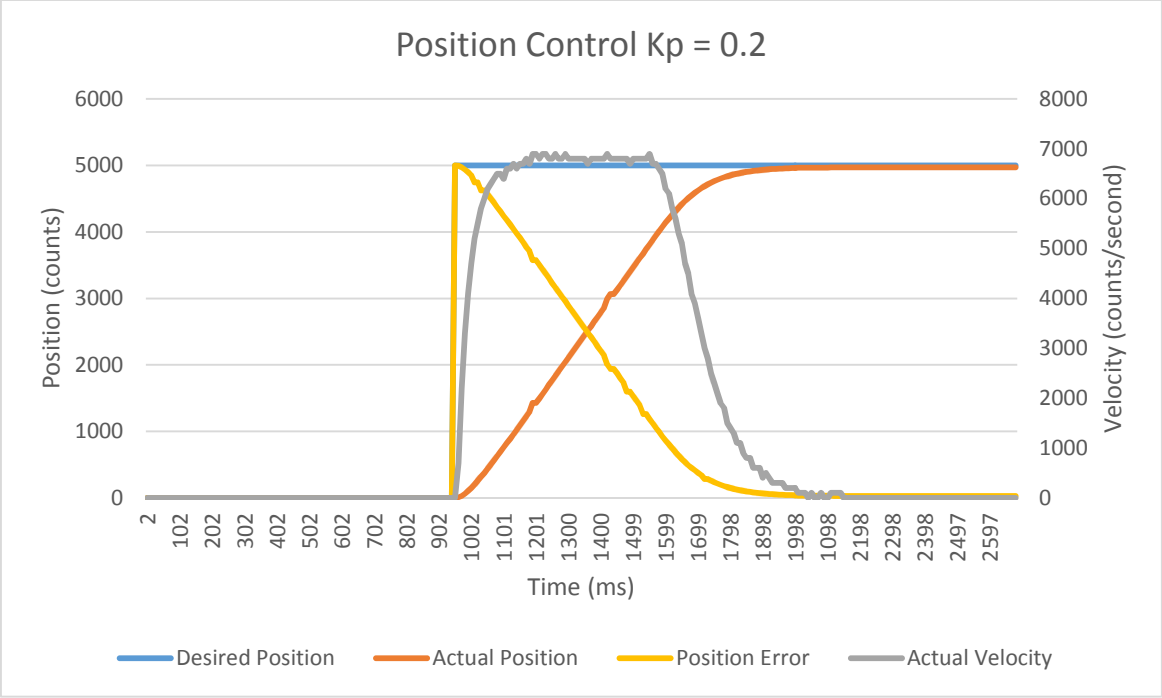
Our first task is to set the gains very small so the motor doesn't go at full speed. Based on my values for incrementing the proportional gain, I needed to drop my change from 0.2 all the way down to 0.01. At 0.2, my motor was already hitting full speed. I dropped the Kp down to 0.02 and got the graph below. One thing to note is that I now multiply my velocity from above by 100 to get the velocity in counts per second, where before my velocity was in counts per 10ms.



We can see from this graph that we don't come anywhere close to hitting full speed. Also, because our proportional gain is very small, we come up very short of our destination. This is because multiplying our error by 0.02 eventually gets to the point where our torque isn't enough to overcome the static friction to keep the motor moving. It looks like this happens at around 600 counts, which would leave our error at 400 counts. At 400 counts, multiplied by our 0.02 derivative gain, our torque would be 8, and that is right about the threshold where the motor will no longer continue to move forward.

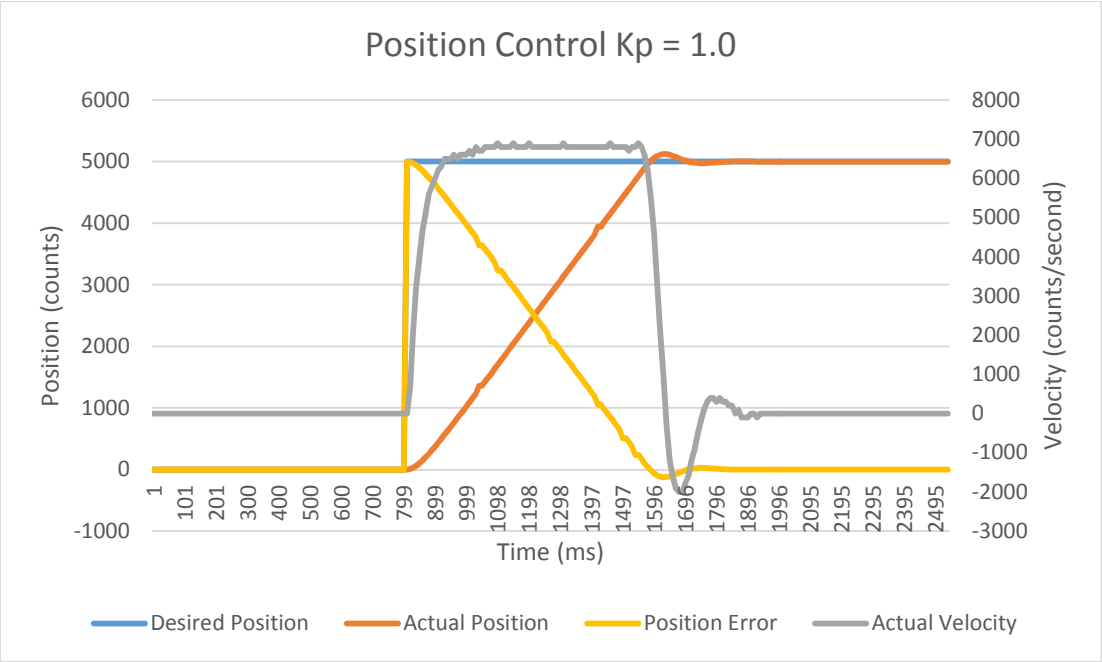
Let's increase the Kp gain and see if we can get a little better response than this. We'll put our increment value for proportional gain back to 0.2 and set it to that value. Setting our desired position to 1,000 counts gives us the following graph.

We can see that our speed does pretty good. We don't quite reach the maximum speed of around 6,900 counts per second, but we're a lot better off that we were last time. We're also not going very far, so we may not have had time to get up high enough. Let's try the same gains with a desired position of 5,000 counts.



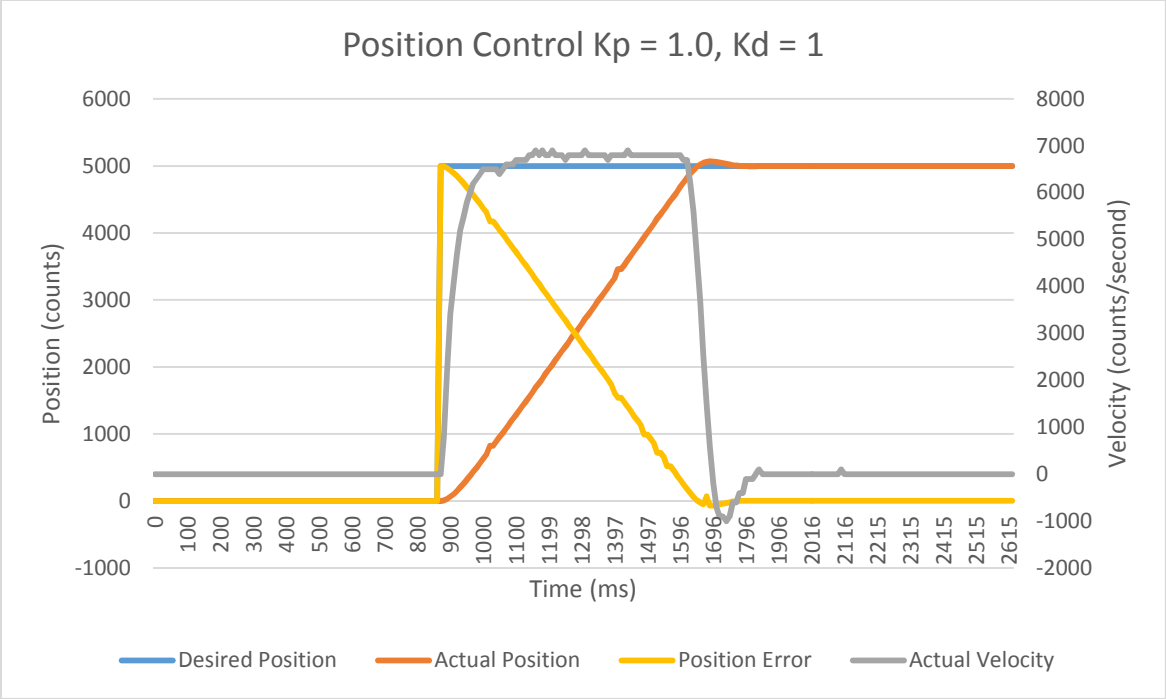
OK, now we can see that, given enough time, we are able to reach our maximum speed with a proportional gain of just 0.2. It is sort of hard to tell from the graph, but we still fall short of our desired position by about 30 counts. If we do the math, 30 counts multiplied by a proportional gain of 0.2 gives us a torque of 6, which is pretty close to the 8 we were at with the lower gain when we fell short before. I think we can explain the difference of two in the torque because we are going much faster, and that increased momentum will just naturally take us closer to our desired target. Actually, I proved that theory by manually moving the motor away from the desired position, and even at 40 counts away (torque of 8) it was still not able to move any closer to our desired target, so that is the same as our previous experiment.

Let's increase the proportional gain a little bit more. Actually, let's increase the proportional gain to 1.0.

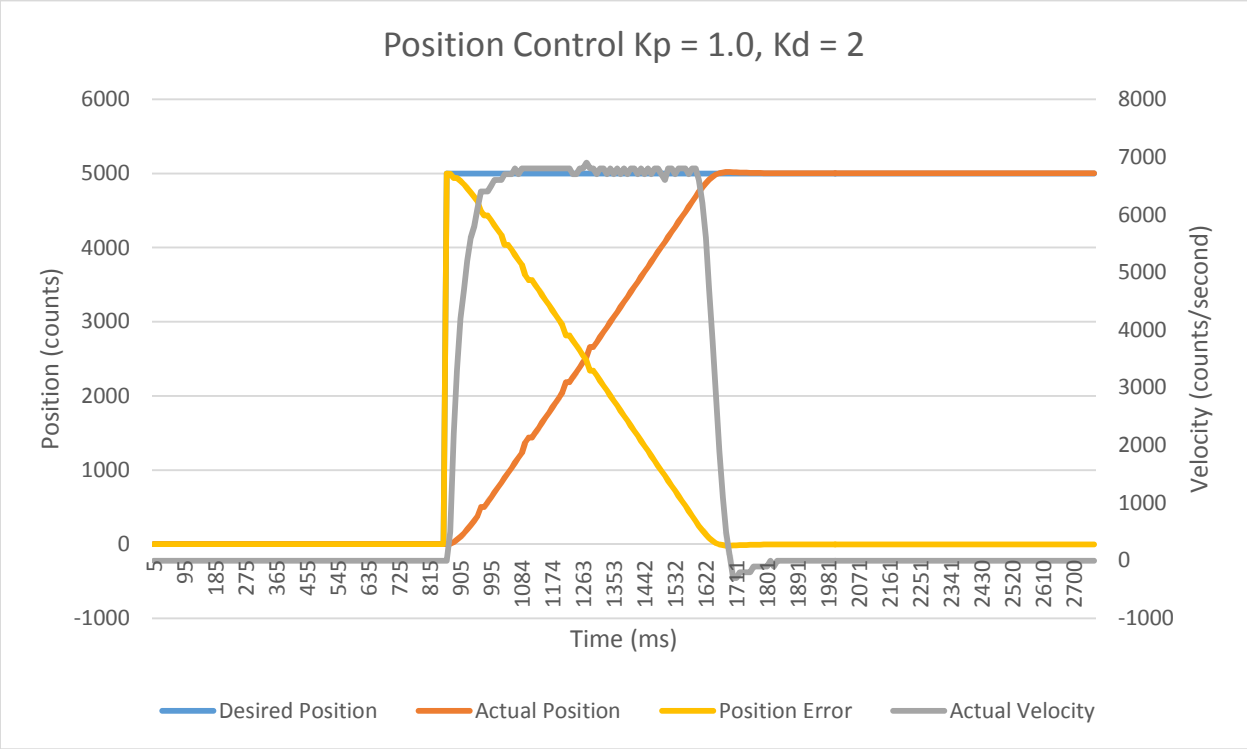


OK, we can see that we now get some overshoot, so it is starting to get exciting. We are once again hitting our maximum velocity, so that is good. Our overshoot is because our gain is pretty high at 1.0, and that is going to not allow us to recover in time as we move closer to our destination. I feel pretty good about our proportional gain. Let's see if we can take care of some of that overshoot by adding in derivative gain.

We will first add the minimum derivative gain allowed by our system, which is 1.



Well, our overshoot definitely improved. We are still getting a little bit of overshoot, and that could be because we aren't removing enough torque due to our derivative gain. Let's try increasing that and see what happens.



Here we can see that at a proportional gain of 1.0, and a derivative gain of 2, we are starting to look pretty good. Our velocity easily reaches the maximum, and our overshoot is very minimal. Our position settles in within 3 counts of our desired position, so I think we are sitting pretty here.

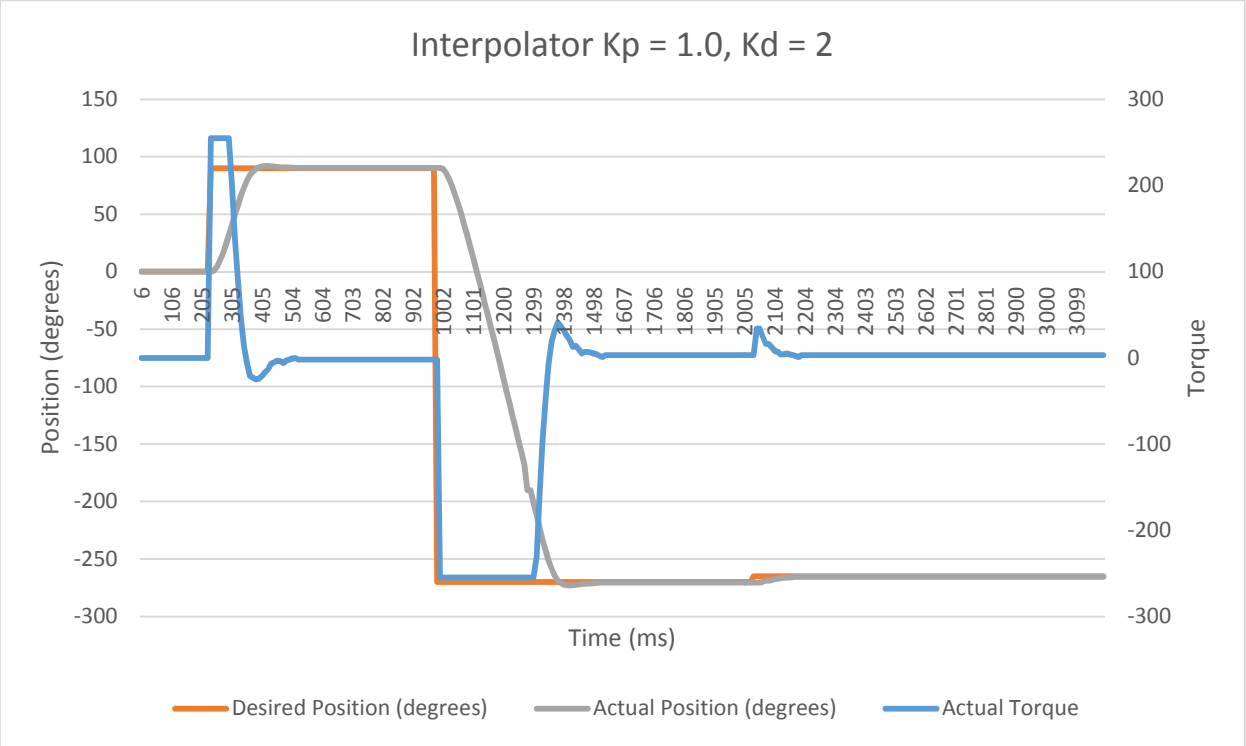
Our original problem with our overshoot was that we were going so fast that our proportional gain didn't have time to recover as we approached our target. The proportion gain would decrease, but not in enough time to stop the motor before overshooting. With the negative impact the derivative gain has on the overall torque, we can see that the faster we go, the more torque is being removed because of the derivative gain. This allows the proportional gain to recover in time to not overshoot our target by much. These values will be satisfactory for our interpolator in the next portion of the lab.

Part 4 – Trajectory Interpolator

We've now done enough experimentation to feel pretty good about our values for position control. We are going to update our PID loop every 10ms. This is a very good value based on our minimum speed and our desired settling time. From the previous experimentation in position control, we feel pretty good about using a proportional gain of 1.0, and a derivative gain of 2. With those values, and that update speed, we are able to get to a position relatively quickly, and settle into that position reliably.

I've implemented the interpolator, and it appears to be working. I've hard-coded all the values, so all a user has to do is tell it to start, and it does the rest. It is able to be run back to back quite reliably. The code isn't the prettiest, but you can't see the code when the motor is locking in on positions and humming along, so I'm not going to get too worked up about it.

Below is the graph of the interpolator at our optimal values for PID update speed, and the two gains we're using.

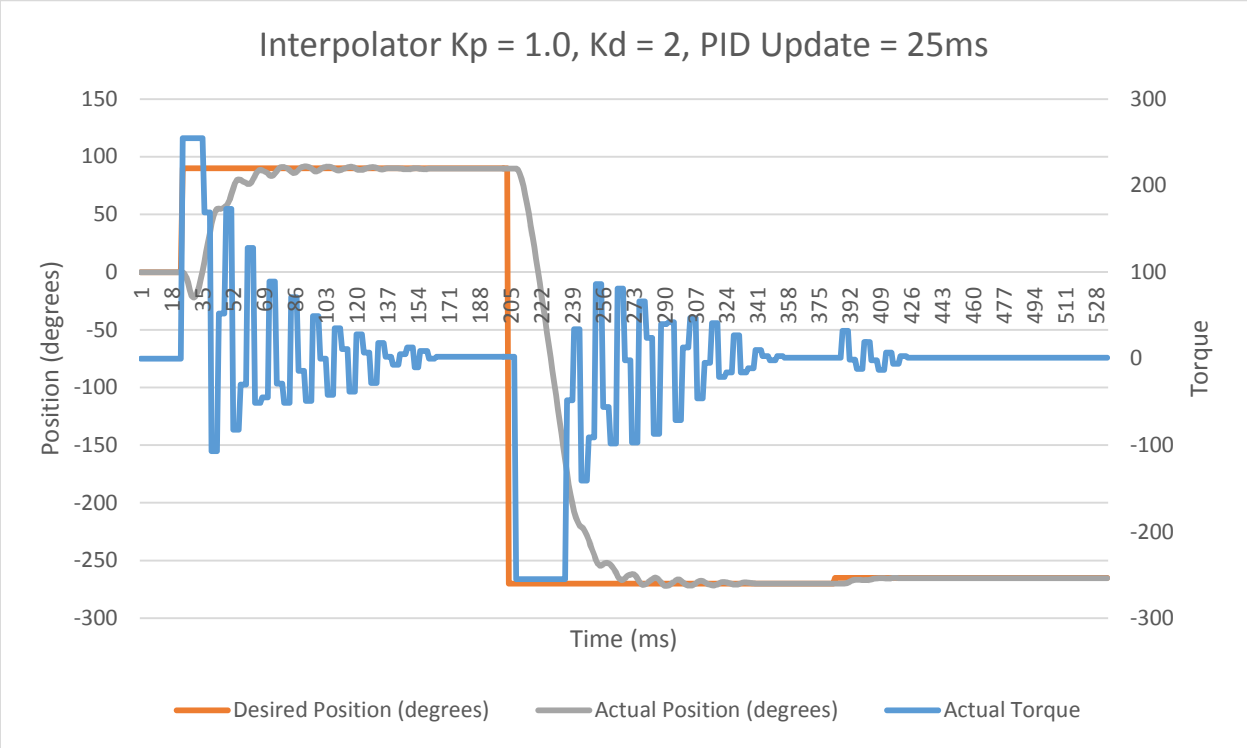


We can see that this graph looks pretty good. We settle into a position, sit there for half a second, head to our next destination, wait there for a half second, and then move just a little bit to our third and final destination. I don't think there is much to talk about here, as this graph is basically just three of the previous graphs put together end to end and shown on the same graph. We've got good settling time at all three positions, and we can see when given time, our torque reaches the maximum (and minimum) value, so we know we're going as fast as we can. Overall, I think this graph looks good, and I am happy with the results.

Part 5 – Play With Interpolator Timing Speed

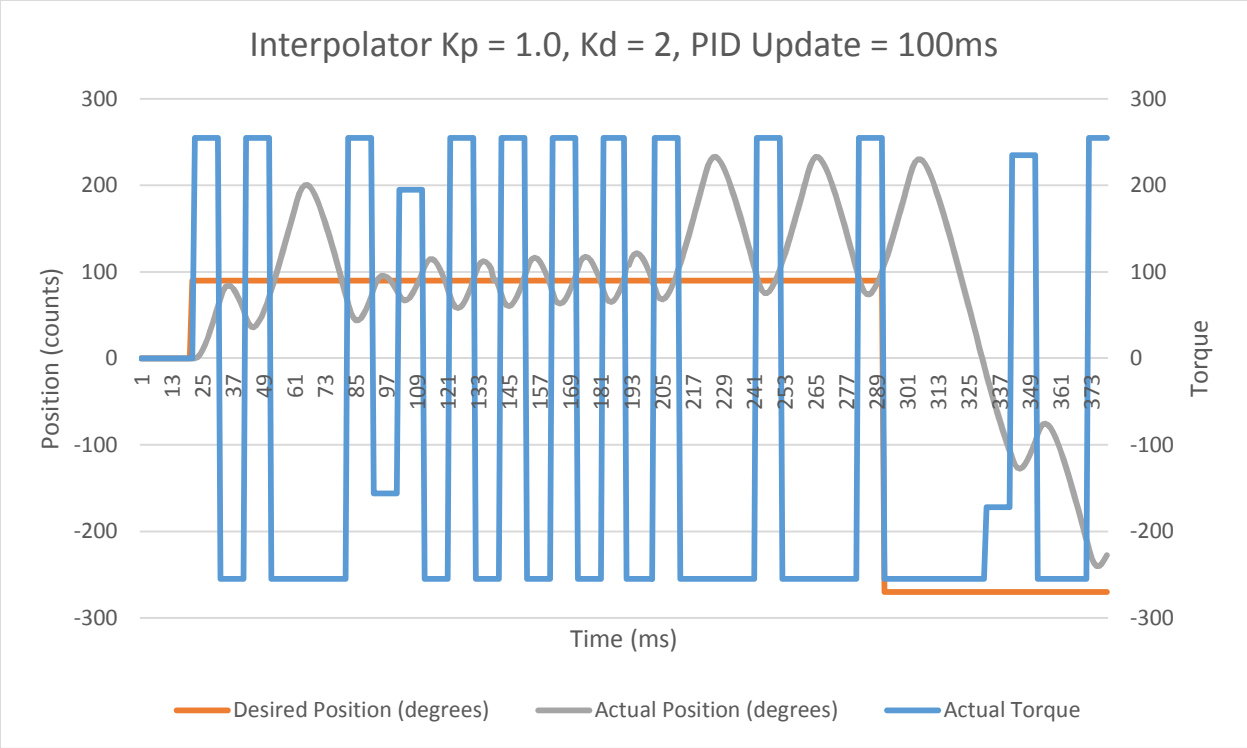
Our next thing to try is to slow down the PID update and graph those results. My assumption is that we are going to see the same results we saw when slowing down the PID on the speed controller, only with regards to position control. We are asked to graph it at a slow rate, and a very slow rate. The two rates I will choose will be 25ms update for a slow rate, and a 100ms update for a very slow rate. Those values represent cutting the update rate by four and by ten respectively.

First we'll cut our PID loop update to 25ms.



We can see some interesting results here on the graph, which sort of mimic the results from above. We can see that our torque is oscillating all over the place. This is because of the derivative gain we are subtracting out. We can see that we are oscillating around our desired position because we are not updating often enough to see when we are actually in position, and by the time we check again, we are past our position and we have to head back.

When we slow our PID update down to 100ms, I expect the above issue to be exacerbated.



As we can see, the problem is for sure exacerbated. We can see that with the longer time in-between updates, we get ever further away from our desired target, which allows our torque to become even bigger. We just oscillate back and forth forever around our target. As you can see from the graph, we never even finished our trajectory before we ran out of space in our arrays. I'm a little surprised we ever even made it past the first desired position. We must have just gotten lucky with the way I am calculating whether or not we're in position. That might be a possible update when I come back to improve this code, or if we use PID in our final project.

Conclusion

This lab was a lot of fun, and taught me a whole lot about PID loops. In my professional career, I've been responsible for tuning PID loops, but never fully understood how they worked, nor how easy they are to implement.

It was interesting to see what effect the different gains had on the response of the system. It really drove home the point of each of the gains, and what happens when those gains are increased or decreased.

I was very surprised to find out that the PID loop to control position and speed was exactly the same. It took me a little bit of time to really wrap my head around why that was the case. Once I was able to figure that out, it was clear to me how many situations a PID loop could be applied to.

One of the most interesting aspects of this lab, at least for me, was to see the results of playing with the PID update time. It is a little bit of an art to really pick the update time that works best for the system we're dealing with. While a 10ms update speed was not optimal for really slow motor speeds, it was pretty optimal for every other speed we could set our motor to. There is clearly a trade-off with regards

to the update speed, and I could see where you might run into issues deciding what the best update speed might be.

When all is said and done, it is clear how powerful a well thought out and tuned PID loop can be to control both speed and position.