Christopher Sadlo 3/28/2015
Prof. Amy Larson SEng 5831 - Embedded

# PID Motor Control Lab Report

Part 1)

The first step I took was to determine what sort of timestep size seemed reasonable

180 RPM
180 / 60 = 3 RPS (per second)
3 * 2249 = 6747 encoder counts per second
20 samples per second so…..
6747 / 20 = 337.35 encoder counts per sample

5 RPM
5 / 60 = 0.083333333 RPS
0.0833333 * 2249 = 187.4166667 encoder counts per second
20 samples per second so……
187.4166667 / 20 = 9.370833333 encoder counts per sample

(at 50 ms sampling period)
9 encoder counts per sample = 10800 encoder counts per minute = 4.802 RPM
10 encoder counts per sample = 12000 counts per minute = 5.335 RPM

5.335 - 4.802 = 0.533 RPM precision

(at 20 ms sampling period)
5 RPM
5 / 60 = 0.083333 RPS
0.083333 * 2249 = 187.416667 encoder counts per second
50 samples per second so…..
187.4166667 / 50 = 3.7483 encoder counts per sample
3 counts per sample = 9000 counts per minute = 4.000 RPM
4 counts per sample = 12000 counts per minute = 5.335 RPM

5.335 - 4.000 = 1.335 RPM precision

I concluded that I wanted to use the larger timestep period because it would allow for more precision in calculating my measured RPM at slow speeds. I thought about creating some kind of rolling window to average measurements collected over the previous 2-3 timesteps and get a more accurate, albeit delayed RPM measurement, but I decided to do that later if I had time.

Part 1 continued)

Target speed is 50 RPM and a default 50 ms sampling rate

Adjusting Kp:

| Kp | Speed (RPM) | Speed (RPM) with 10 ms sampling rate | Speed (RPM) with 20 ms sampling rate |
|---|---|---|---|
| 1 | 19.70 | 21.34 | 20.00 |
| 2 | 28.81 | 29.34 | 29.34 |
| 3 | 34.14 | 34.68 | 33.34 |
| 4 | 35.21(Jittery) | 37.34 | 36.01 |
| 5 | oscillations | 40.01 | 38.68 |
| 6 | | 41.68 | 43.68 (Jittery) |
| 7 | | 43.68 (Jittery) | oscillation |
| 8 | | 50.68 (Jittery) | |
| 9 | | oscillations | |

At this point I realised that all of my problems with trying to tune the Kp value is caused by the fact that my PID formula was written like this:

voltage = (Kp*error + Ki*integral + Kd*derivative);

T = Kp(Pr - Pm) + Ki(SUM over time (Pr-Pm)*dt) - Kd( delta(Pr-Pm) / dt )

Pr = goal
Pm = measured
error = (Pr - Pm)

T = Kp(error) + Ki(SUM over time (error)*dt) - Kd( (error - prev_error) / dt )

integral += error*dt

The best set of PID gains I found for this formula were Kp=0 Ki=400 Kd=0

In the spirit of following along with the Lab instructions, I changed my PID formula for speed to this:

voltage += (Kp*error + Ki*integral + Kd*derivative);

Starting over…..

Target speed is 50 RPM and a default 50 ms sampling rate

Adjusting Kp:

| Kp | Speed (RPM) | Speed (RPM) with 10 ms sampling rate | Speed (RPM) with 20 ms sampling rate |
|---|---|---|---|
| 0.1 | 40.55 | 42.68 | 41.35 |
| 0.2 | 46.42 | 45.35 | 46.68 |
| 0.3 | 47.48 | 48.02 | 46.68 |
| 0.4 | 48.55 | 48.02 | 48.02 |
| 0.5 | 48.55 | 48.02 | 49.35 |
| 0.6 | 49.08 | 50.68 | 49.35 |
| 0.7 | 49.62 | 48.02 | 49.35 |
| 0.8 | 49.62 | 50.68 | 49.35 |
| 0.9 | 49.62 | 48.02 | 49.35 |
| 2.0 | strange whining | strange whining | strange whining |
| 3.4 | oscillations | oscillations | oscillations |

Observations: The strange whining noises are being caused by the voltage being fluxulated too quickly for the motor speed to change very much.

All of the experiments would overshoot their target 50 rpm and oscillate if the speed was started at 0 rpm and the Kp was more than 1.2.

It seems that by increasing the sampling rate, the RPM became less accurate and would jump around between speed at larger intervals.

Using a Kp = 2.0, I tested the accuracy of the speed

Adjusting Speed:

| Speed (Target) | Speed (Actual) |
|---|---|

| | |
|---|---|
| 5 | 4.80 |
| 20 | 19.74 |
| 40 | 40.01 |
| 60 | 59.75 |
| 80 | 80.03 |
| 120 | 120.05 |

I continued using a sampling rate of 50 ms and a Kp = 2.0.

I began to add in the Ki gain.
Adjusting Ki:

| Ki | Speed (RPM) |
|---|---|
| 1 | 38.4 |
| 2 | 28.81 |
| 3 | 34.14 |
| 4 | 35.21 |
| 5 | oscillations |
| 4.5 | jittery |

Chart: Target = 50 RPM Kp = 2.0 Ki = 0.0 Kd = 0.0

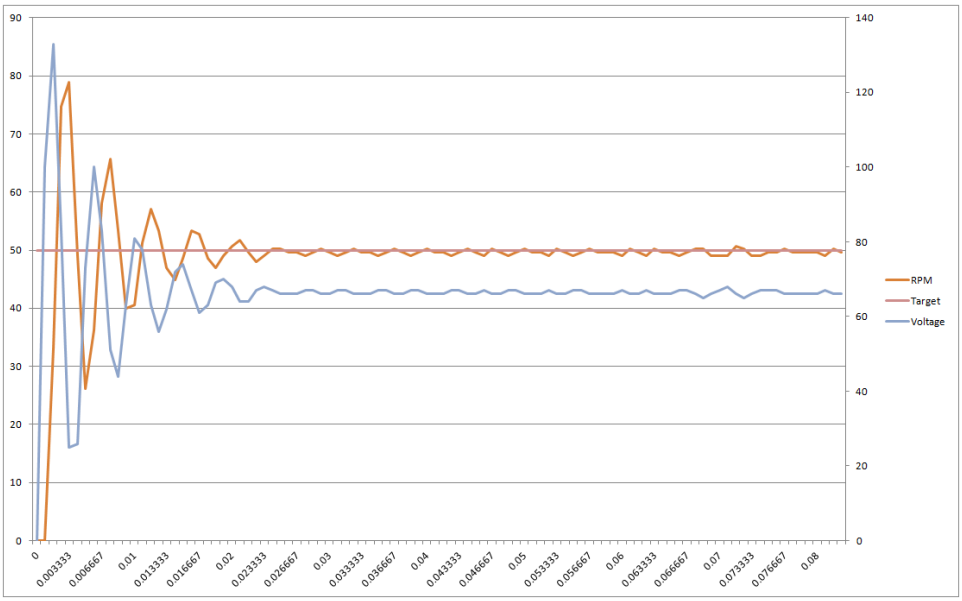
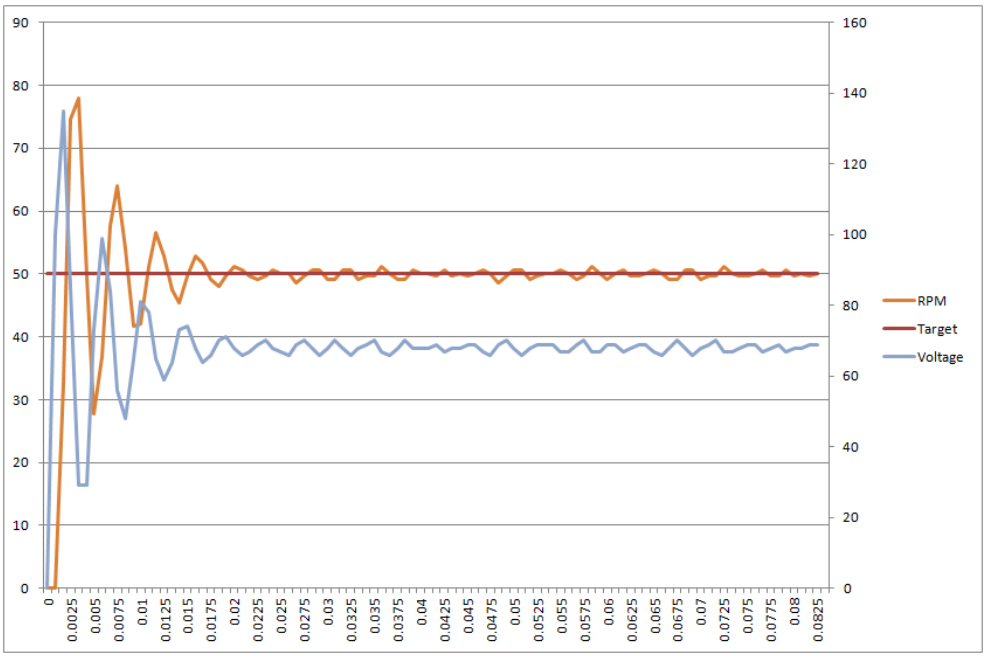
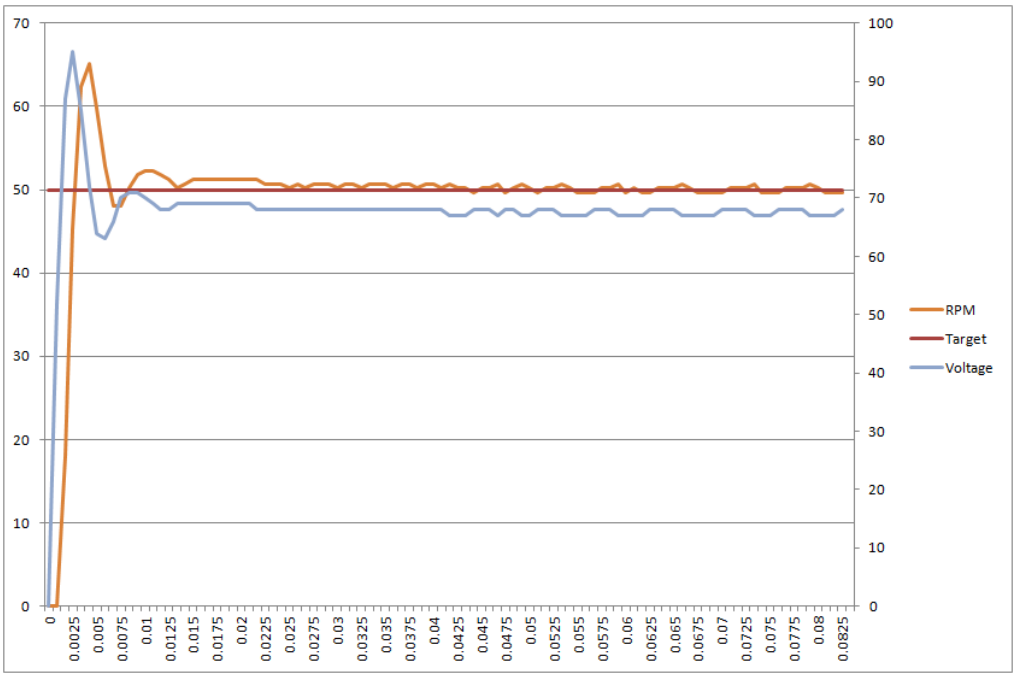


Chart: Target = 50 RPM Kp = 2.0 Ki = 10.0 Kd = 0.0



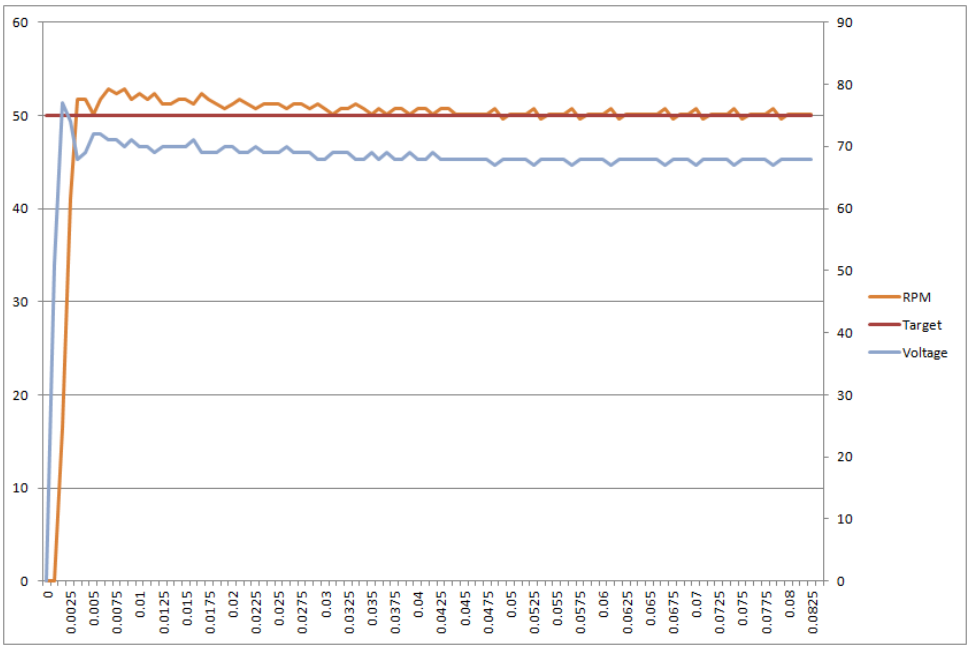

I've tried to increase the Ki gain for better results and I've tried values up to 200 but it hasn't been helpful. I've decided that it's quite difficult to tune with Kp so high, so I repeated an experiment with a lower Kp gain of 1.0 and the results were more promising. Basically the issue is that the Kp is so high that it kept overshooting the target goal.

Chart: Target = 50 RPM Kp = 1.0 Ki = 50.0 Kd = 0.0



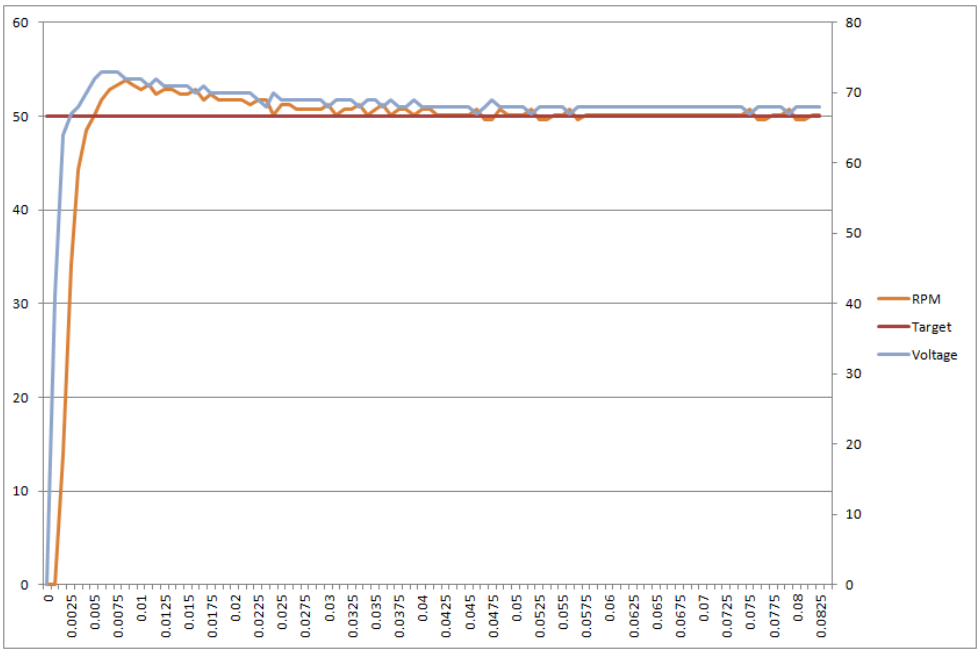
Once I had the Kp and Ki gains dialed in, I started to experiment with the Kd gain for dampening. I had to slowly adjust my Kd because even relatively large values would cause it to upset the motor and cause the board to shut down.

Chart: Target = 50 RPM Kp = 1.0 Ki = 50.0 Kd = 0.0005



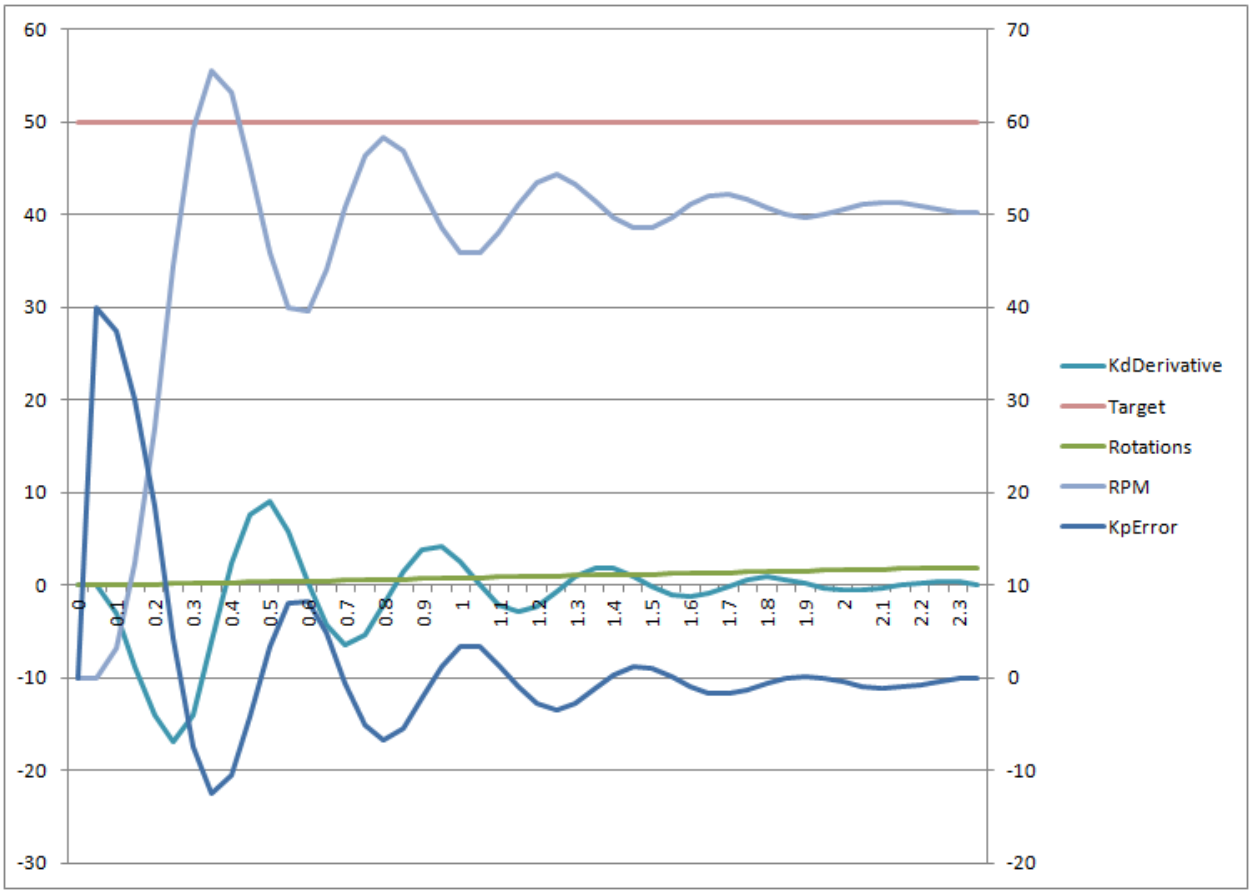

It appears in the above graph that the dampening effect of the Kd is not quite enough to eliminate oscillation completely. However, by lowering the Kp to 0.8 in the experiment below, the Kd was enough to compensate.

Chart: Target = 50 RPM Kp = 0.8 Ki = 50.0 Kd = 0.0005



Part 2)
Yes, my gains of Kp = 1.0 Ki = 10 and Kd = 0.0008 work well for both low speed (~5 RPM) and high speed (~180 RPM).

I tried changing the calculation frequency to every fourth pin change interrupt and the motor could not settle down. It kept oscillating between max voltage in one direction and max voltage in the other direction. A different idea I had was to use a rolling average of the current RPM measurement and the previous 3 measurements into an average RPM. Doing so should allow for the “RPM” to settle closer to the target during the steady state. Another possibility to try is to change the sampling period depending on if we’re targeting a position or a speed.




I attempted my second idea of using a rolling window to average the RPM with previous RPM values. This gave me more possible values of RPM to settle on. However, I did introduce oscillations into the experiment. They are nearly as bad as just calculating the RPM every fourth interrupt call. I believe these oscillations are caused by the fact that the derivative term is still be calculated using the current RPM. This is causing a delay in the derivative component that one can see in the above chart. The derivative simply isn’t there in time to afford any appreciable dampening effect. “Too little, too late” comes to mind.

Part 3)

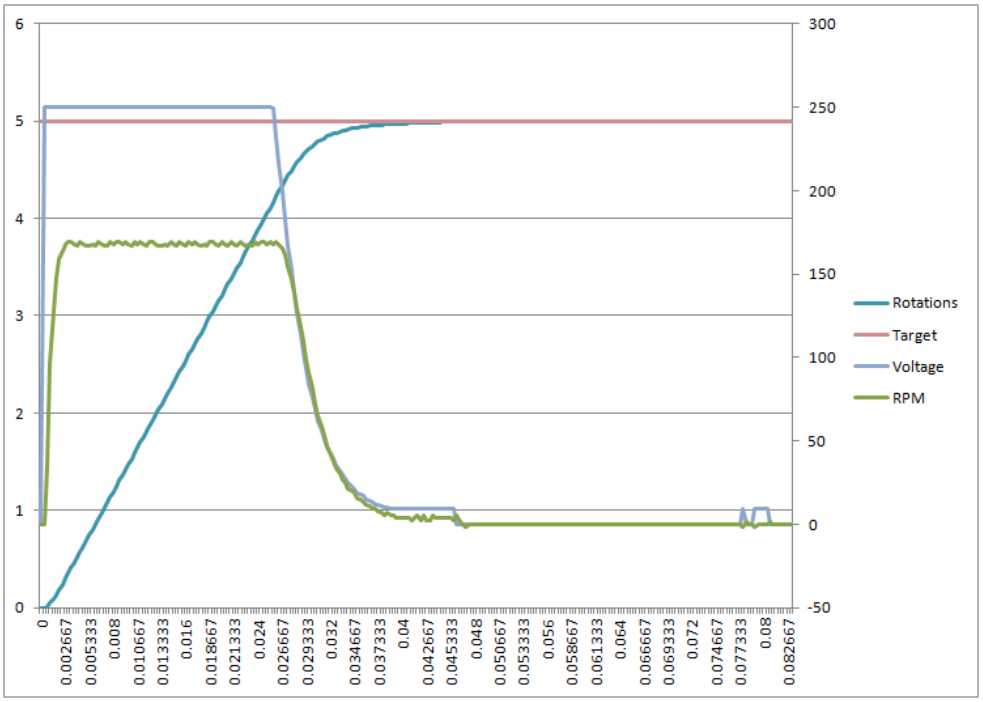
Target rotations is 2  forward and a default 50 ms sampling rate

Adjusting Kp:

| Kp | Rotations | Rotations with 10 ms sampling rate | Rotations with 20 ms sampling rate |
|---|---|---|---|
| 1 | no movement | no movement | no movement |
| 10 | 1.953296 | 1.900872 | 1.902637 |
| 20 | 1.950638 | 1.950662 | 1.900859 |
| 50 | 1.985324 | 1.981331 | 1.980884 |
| 100 | 1.993774 | 1.990665 | 1.991554 |
| 300 | 2.000444 | 1.997777 | 1.998222 |

It appears that increasing the Kp gain has two positive effects. The first is that we approach our target rotation quicker. The second which is more surprising is that our steady state measured position is closer to our target than at smaller Kp values. I theorize that this is due to the Kp's ability at larger gains to correct for overshoot. However, because I have removed the Ki gain, it seems that the motor will virtually always undershoot the target position.

Chart: Position Target = 5 rotations  Kp = 300  Ki = 0.0  Kd = 0.0



In the chart above we can see that the controller is running at full speed to rotate a full 5 rotations. The little blip in the voltage at the far right end of the plot is me picking up the motor and accidentally turning the motor encoder wheel and the motor correcting to put things back at exactly 5 rotations.

The Ki terms seems to work against me depending on how many rotations I must pass through in order to match my target. The more rotations, then the worse the effect. It's because there is more error being accumulated into the integral component, causing it to take longer to reduce itself it the opposite direction. For fun, I modified the PID formula slightly such that error does not accumulate in the integral component unless the measured position is within one full rotation of the target position.

```
error = goal - rotations;
if (TARGET_ROTATIONS)
if (fabs(error) < 360*error_threshold) //goal within one rotation
     integral = integral + error*dt;
if (TARGET_RPM)       //then always accumulate error
     integral = integral + error*dt;
derivative = (error - pre_error)/dt;
voltage = (Kp*error + Ki*integral + Kd*derivative);
```

As we can compare in the charts below, the error integral term should only be of interest when we are getting close-ish to our target position. This is especially true if we are concerned by overshoot. The possibility of which would otherwise increase as we travel through many "intermediary" rotations on the way to the target. This adjustment to the PID formula would not be necessary if there was infinite motor speed available. But because the motor speed is limited to ~180 RPM, the motor could be running at full speed for quite a long time while error is accumulating in the integral component, and increasing the delay of it causing any benefit towards approaching the target position in the opposite direction after overshooting.

Chart: Position Target = 5 rotations  Kp = 300  Ki = 500.0  Kd = 0.001  Normal PID

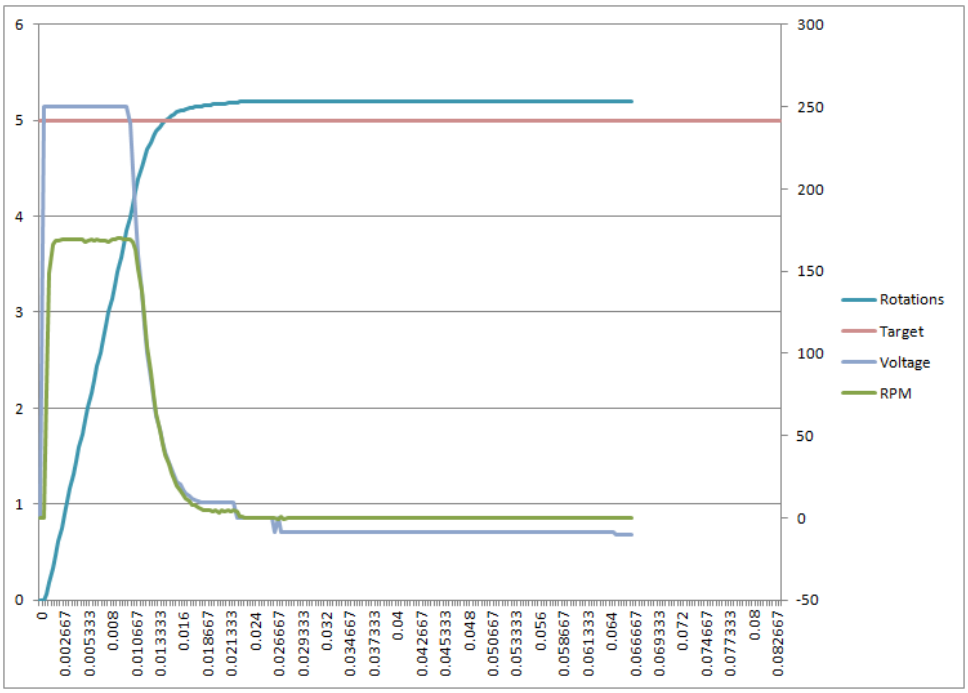
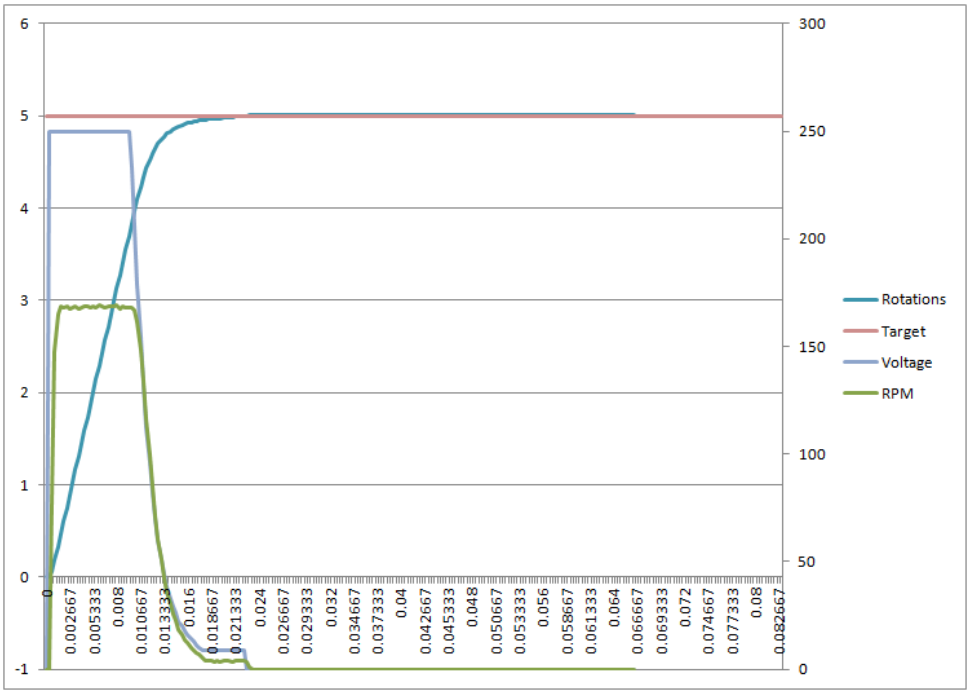



Chart: Position Target = 5 rotations  Kp = 300  Ki = 500.0  Kd = 0.001 w/special PID formula




I also was happy to see that the special modification to the PID formula was not causing any issues with the speed controller.

Yes, without the Ki term, there is a certain amount of undershoot which is very difficult to get rid of. I found that the best gains values I had for the position controller while using the special formula was Kp=400 Ki=200 and Kd=0.3

Chart: Target = 5 rotations Kp = 40 Ki = 20 Kd = 0.03 (Low Gains)

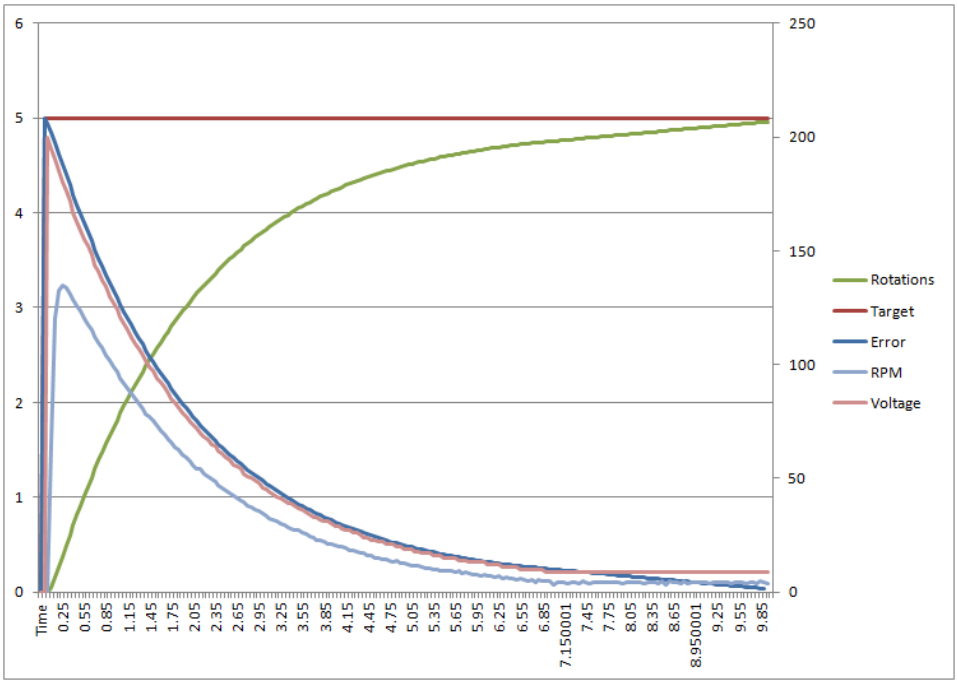



Chart: Target = 5 rotations Kp = 40 Ki = 0 Kd = 0.03 (Low Gains, no Ki)

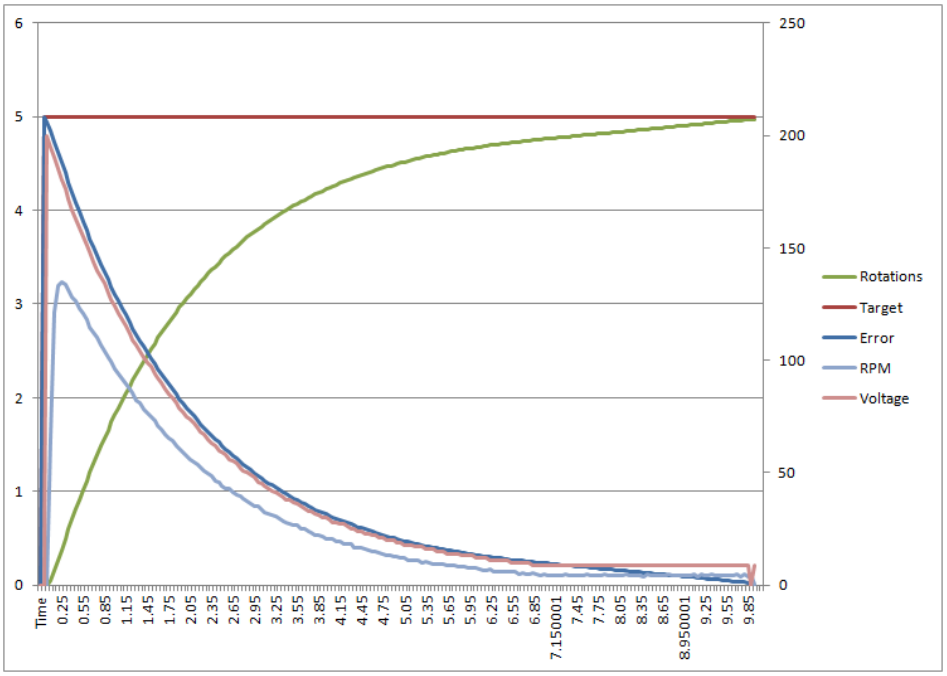
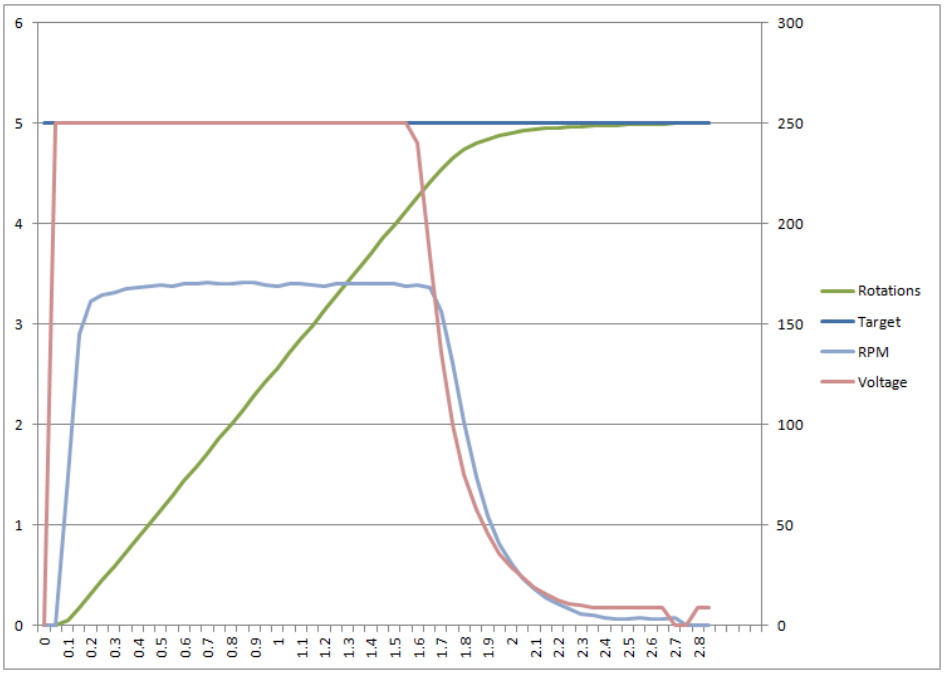
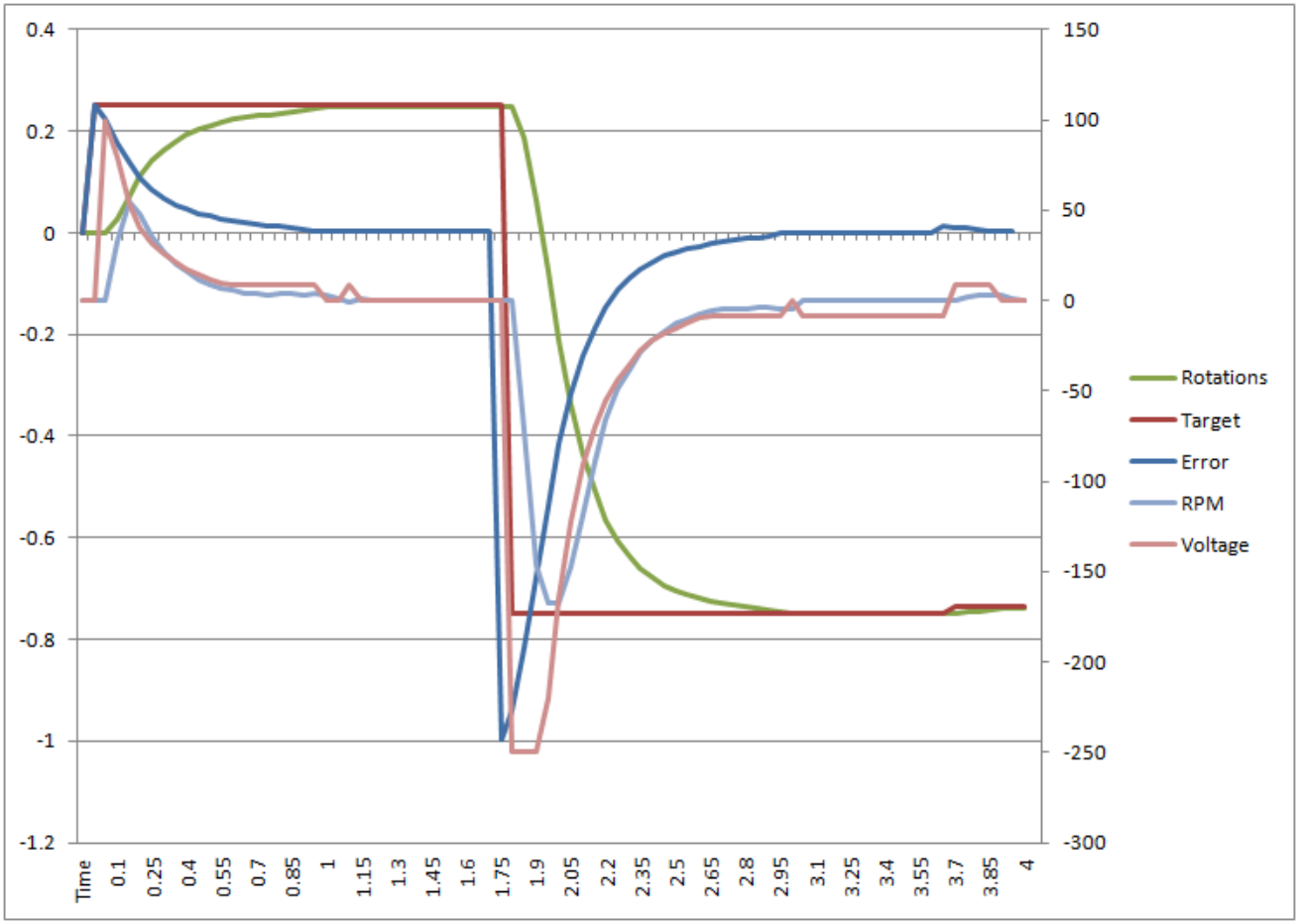


Chart: Target = 5 rotations Kp = 400 Ki = 200 Kd = 0.3 (High Gains)



Part 4)

Chart: Trajectory  Kp = 400  Ki = 200  Kd = 0.3  (High Gains)



Part 5)
Chart: Trajectory  Kp = 200  Ki = 200  Kd = 0.3  (Medium Gains)

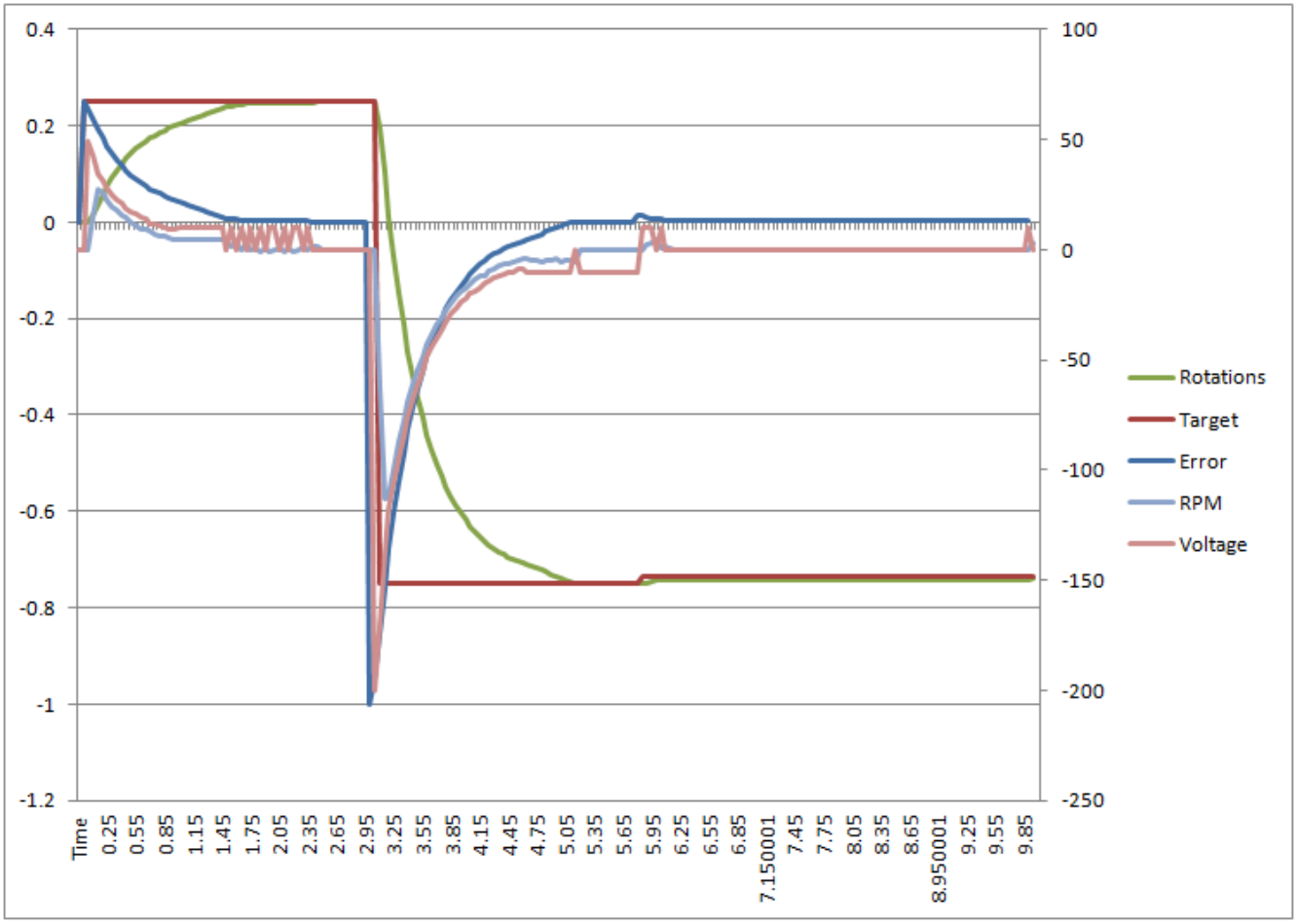



Chart: Trajectory  Kp = 100  Ki = 200  Kd = 0.3  (Low Gains)
Chart Not Available

I was not able to get a trajectory for the interpolator for very slow gains. It simply was taking too long to run.  My theory is that by reducing the Kp gain to such an extent, the integral component dominates for too long and dwarfs the proportional component.  This in turn is causing the PID formula to not respond quickly enough to sudden changes in the new position required.  I could “cheat” by having the integral component begin to accumulate/dissipate the integral component during the 500 ms rest. it would be as simple as setting the new positional goal BEFORE going on the 500 ms rest instead of AFTER.  I did make an additional modification to the PID formula that has been in effect during my whole lab but I thought I should point it out now. If the calculated voltage to use isn’t 0 but it’s within the range (eg. 1 < V < 9) to which there isn’t enough force to actually move the driveshaft, I round up or down to the nearest value (e.g. 9) that will affect a change. This allows the position to converge more quickly due to a reduction in the “dead zone” of the motor signal.