GE420 Laboratory Assignment #7

PI Motor Speed Control

Goals for this Lab Assignment:

1. Design a PI controller to regulate the speed of the DC motor with attached flywheel.
    a. Simulate the motor's dynamics and design a PI controller to meet the given specifications.
    b. In the lab, implement the PI control on the actual system.
2. Investigate integral windup.
3. Identify the friction in the motor and implement a method to compensate for the friction.

DSP/BIOS Objects Used:

Clock

Library Functions Used:

readEnc1, setEPWM3A

Matlab Functions Used:

Simulink Plotting

Prelab:

Read the entire lab so you understand what is envolved for this lab assignment.

Simulation:

1. You will design a proportional plus integral (PI) controller to regulate the speed of the DC motor. For reasons you must explain below, you will not be able to place the closed-loop poles of your system using classical techniques, or be able to determine from the closed-loop transfer function whether your controller meets given specifications; these approaches simply will not work. Instead, you must use a simulation's time response to design this controller 'by hand', iteratively tuning the Kp and Ki gains and checking the output of your model. You may wonder why traditional approaches do not work, so in preparation for this lab, we simulated the motor response of a PI controller (the block diagram is shown in Figure 2 below, with Kp and Ki = 1). To make the discussion easier on you, we eliminated the encoder quantization because this isn't the problem in this case. We tested the response for reference velocity commands at lower speeds of 1,2,3,4 rad/sec (left plot of Fig. 1 below) and at higher speeds of 100,200,300, 400 rad/sec (right plot of Figure 1 below). We divided the measured velocity by the amplitude of the reference so we can appropriately compare the responses. For example, for a reference signal of 400 rad/sec amplitude square-wave, the measured velocities are all divided by 400. The lower velocity responses (1,2,3 and 4 rad/sec reference commands) *all produced the same curve* while the higher velocity responses are all *separate curves*. Why does this result show you that classical control techniques will not work if we want to track responses in the range of 100 to 400 rad/sec?

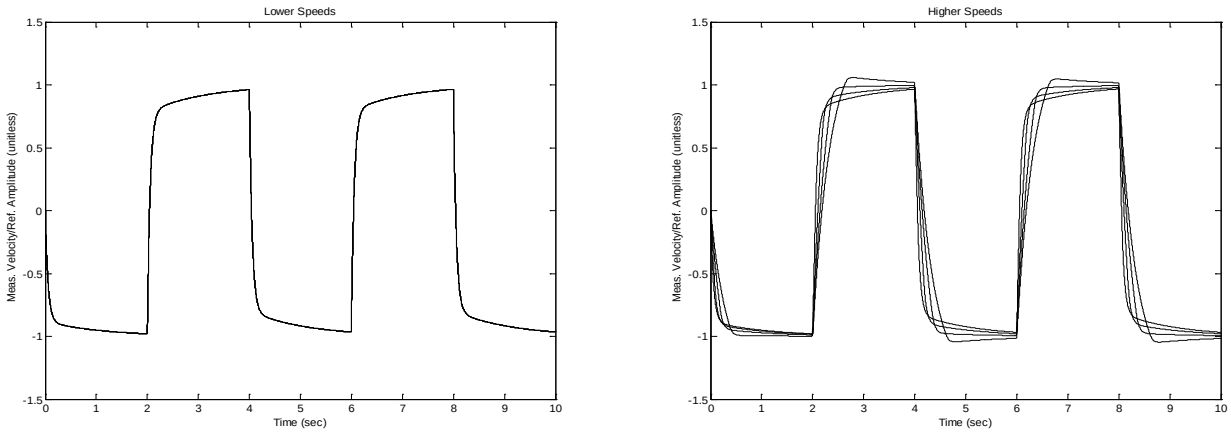

Figure 1: Normalized responses for reference amplitudes 1,2,3,4 rad/sec (left) and 100,200,300,400 rad/sec (right)

Use **Figure 2** as a guide to create a SIMULINK model, and use the values of c_1 and c_2 you identified, sampling every 1ms, in the previous lab for the dynamics of the motor. Using this diagram, design a controller with the following specifications:

a. Sample rate 1KHz.

b. Minimal overshoot (1% or less).

c. Rise time greater than 200 ms. but less than 300 ms.

d. The reference signal should be a square wave with amplitude 100 rad/s and a period of 4 seconds. This will command the motor to spin first at 100 rad/s for 2 seconds and then -100 rad/s for 2 seconds repeatedly.

e. Saturate your control signal with a saturation block so that it stays in the range –10 to +10. The DSP PWM output signal has the range limitation of –10 to +10.

f. Limit your gains to values less than 1.0.

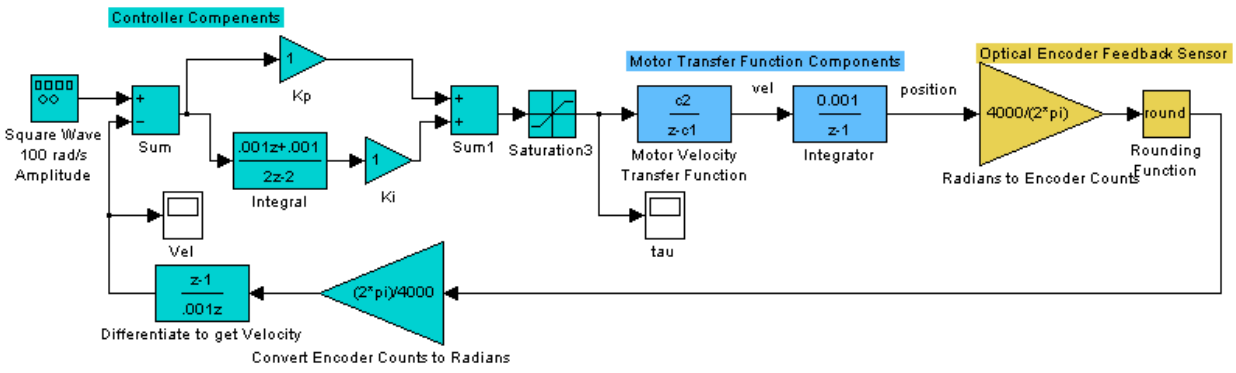

Figure 2: Simulink Block Diagram for PI control of DC Motor

Laboratory Exercise

Today's lab includes two exercises. The first exercise is to implement the PI controller you just simulated. You will hand tune the PI controller and update the speed controller's set point on the fly. In the second exercise we ask you to

implement a method of friction compensation for the DC motor. First you will be asked to identify the friction in DC motor. Then with those friction values you will implement a compensator to adjust for that friction.

In previous labs we used the GE420_serialread and GE420_serialwrite functions to upload and download values between the DSP and Matlab. In this lab you will be shown another way to do this. Simulink will be used to plot and save data in real-time and Code Composer Studio's Expressions Window will be used to change or tune controller parameters. In future labs you will be able to choose which ever method you prefer to exchange data with the DSP.

Exercise #1

1. Implement the PI control on the DSP/DC-Motor system. Your code should implement the following operations/specifications:
    a. Controller's sample period = 0.001 seconds.
    b. Read the motor's encoder position feedback.
    c. Compute motor's velocity from the encoder feedback (encoder channel 1).
    d. In code, create a set point that steps between 100 rad/s and –100 rad/s every 2 seconds.
    e. Calculate control effort.
    f. Output control effort to EPWM3A.
    g. Print the motor's angular velocity, the control effort's integral portion and the full control effort to the LCD every 100 ms.
    h. Transfer your angular velocity, control effort and reference over serial port to Simulink for plotting. Simulink will plot your data in real-time allowing for real-time tuning of your control gains. See how to do this in the below steps.
2. At first get your controller code working, controlling the speed of the motor and printing to the LCD screen. Don't worry about transferring the data yet to Simulink. Use the Kp and Ki gains you came up with in simulation. When working, your motor should spin for 2 seconds in the positive direction close to 100 rad/s and then the next 2 seconds at -100 rad/s.
3. Most of the code you need to transfer data from your DSP program up to Simulink has been given to you both in the DSP sources code and at the Simulink end. Your job will be to make a few changes to up load the data point you are interested. Find the function simulink_serialRX() at the bottom of your DSP C file. Then in that function you will see that there are the variables SIMU_Var1_toSIMU_32bit, SIMU_Var2_toSIMU_32bit, SIMU_Var1_toSIMU_16bit and SIMU_Var1_toSIMU_16bit. When Simulink requests data from the DSP these four variables are sent. So your job will be to assign these four variables the values you would like to plot in Simulink. Notice though that these variables are integers. Two 32bit integers and two 16bit integers. So if you would like to upload a floating point value you will need to scale it by a factor and then remember to scale it back down on Simulink's end. So for example you may want to multiply the number by 10000 on the DSP end and divide by 10000 in Simulink. This gives 4 decimal places of precision which will more than suffice for our plotting needs. Remember that the range of a 32bit integer is -2147483648 to 2147483647 and the range of a 16bit integer is -32768 to 32767. The default code uploads the optical encoder 32 bit count values and the ADC 12 bit values. Change these four lines of code so that:

    SIMU_Var1_toSIMU_32bit = 10000 times your calculated velocity

SIMU_Var2_toSIMU_32bit = 10000 times your calculated control effort

SIMU_Var1_toSIMU_16bit = 10 times your reference value

SIMU_Var2_toSIMU_16bit = 0 unused.

That is all that is needed to upload those four items to Simulink. Now switch to Simulink. In Matlab make your Code Composer Studio project's "matlab" directory the current directory. Run the Simulink file simulink_plotAndGains.slx by typing its name without the .slx extension. The left half of the Simulink file is for plotting your four variables and the right half is for downloading values from Simulink. In this lab we are going to only use the four plotting variables. If you would like to play with the downloading capability talk to your instructor. So before running the Simulink file change the appropriate gains. For the two 32bit integers change the gain from 1 to 1/10000. For the first 16bit integer change the gain to 1/10. Then you are ready to start the real-time plotting. Start your DSP code and get the motor stepping back and forth. Then start the Simulink file. After it builds its real-time application, it will start plotting your data. If you setup the scope blocks to log data the date will be stored to variables accessed at Matlab's command prompt. Feel free to use the Mux block to plot multiple signals on the same plot. Notice that your step response looks similar to your simulation run but is a bit slower getting to steady state. You are going to hand tune our control gains in part 5 to make your actual controller meet the simulation specifications. NOTE: When you want to change your code in Code Composer Studio, make sure to stop the Simulink plotting by clicking the stop button. Simulink gets confused when the DSP is halted since no data is being sent back over the serial port. So try to remember to stop Simulink plotting in between debugging your code.

4. Investigate the issue of integral windup. Modify your code to have the reference remain constant at 100 rad/s. Also make sure the print to the LCD the value of your integral term Ik. Once your code is running, hold the motor so it does not spin. What happens when you let go of the motor? This is integral windup. Implement a way of solving this problem. When you are finished fixing this issue show your TA then change the code back to having your reference step back and forth between 100 rad/s and -100 rad/s.
5. As a final step, you will need to hand-tune your Kp and Ki gains to make your controller meet the simulation design specifications and match the simulation response as best you can. For this we are going to copy your simulation into the real-time plotting Simulink file and perform the discrete simulation at the same time we are uploading actual data from the DC-motor. Perform the following steps in your real-time plotting Simulink file:
    a. Copy your entire simulation blocks into the real-time plotting block diagram.
    b. Change the Simulation properties so that the sample rate is 0.001.
    c. Create a plot with a three port Mux to plot Actual Velocity, Simulated Velocity and Actual Reference.
    d. Get rid of the Signal Generator that made the simulation square wave and replace it with the actual reference value. But note that you need to place a "Rate Transition" block (found in the Signal Attributes library) on the actual reference signal saying that the output sample time of the rate transition is 0.001 seconds. This is needed since the simulation is running at 0.001 and the real-time plotting is occurring at 0.005.

e. Also place a "Rate Transition" block on the Simulated Velocity signal before entering the three port Mux for the three signal plot. Here the rate transition sample time output is 0.005 since the simulation is running at 0.001 and the plotting at 0.005.

Now tune your Kp and Ki gains from the Expressions Window in Code Composer Studio. Tune Kp and Ki so that the actual output matches the simulation output. Then when you think you have a good response show it to your TA and record these Kp and Ki gains along with changing your C file so these values are the default Kp and Ki gains.

Exercise #2

In this exercise you are going to implement a friction compensation algorithm to reduce the effects of friction on your system. Your first task will be to identify the friction acting on the motor. To accomplish this you will produce a plot of **steady state control effort** vs. **motor velocity** (rad/s). Your control law has the equation $u = Kp * e(t) + Ki * \int e(t)$. At steady state, e(t) goes to zero leaving the PI controller outputting the control effort needed to overcome the friction in the motor. Using this fact, we can command the motor to different speeds and record the control effort needed to hold the motor at that speed. Then by making a plot of your data, you can determine both the viscous and coulomb friction of the motor in both the positive and negative directions. You should be able to do steps 2-6 in MATLAB without changing your C-code.

1. Record the steady state control effort at the following speeds: -150, -100, -50, -30, -20, -10, -5, 5, 10, 20, 30, 50, 100, 150 rad/s.
2. Produce a plot of control effort vs. speed.
3. Find the two y-intercepts. These are your values for positive and negative coulomb friction. We will label these values Cpos and Cneg (note Cneg will be a negative value).
4. Find the slope of both the positive section and the negative section of the plot. These are your values for positive and negative viscous friction. We will label these values Vpos and Vneg (note Vneg will be a positive value).
5. With these identified parameters add the following friction compensation algorithm to you PI control code:

```
if (motorvel > 0.0) {
        u = u + Vpos*motorvel + Cpos;
}
else {
        u = u + Vneg*motorvel + Cneg;
}
```

6. As a starting point use 95% of the values identified for Vpos, Vneg, Cpos, and Cneg.
7. Make sure to create the variables Vpos, Vneg, Cpos and Cneg and set them to the 95% values you found because you will be tuning them in a few steps.
8. Also for the testing phase turn off the PI control (set Kp=0 and Ki=0) and just output the value from the friction compensator.
9. Your goal will be to tune the friction coefficients so that when you manually spin the flywheel it takes a long time for it to die out. Tune both the positive and negative coefficients so that when you give a positive or negative input spin the motor does not stop spinning for at least 20 seconds. After you find the tuned friction values make sure to record them somewhere for your keeping and set these variables in your source code to their tuned values.

10. Now with a tuned friction compensator add back in the PI controller but with an I-gain of zero. How does a P-only controller perform with added friction compensation?

Lab Check Off:

1. Show your PI simulation
2. Show implemented PI controller, and plot a step response. Explain and show what integral windup is and possible ways to solve the problem.
3. Produce a plot of the friction ID curves (control effort versus velocity).
4. Demonstrate your code with just your friction compensation working (no PI control).
5. Demonstrate a P control with added friction compensation.